Condor® Version 6.4.0 Manual

Condor Team, University of Wisconsin–Madison

September 19, 2002

CONTENTS

1 Overview **1**

1.1 High-Throughput Computing (HTC) and its Requirements 1

1.2 Condor's Power 2

1.3 Exceptional Features 3

1.4 Current Limitations 4

1.5 Availability 5

1.6 Contact Information 6

2 Users' Manual **7**

2.1 Welcome to Condor 7

2.2 What does Condor do? 7

2.3 Condor Matchmaking with ClassAds 8

2.3.1 Inspecting Machine ClassAds with condor_status 9

2.4 Road-map for running jobs with Condor 10

2.4.1 Choosing a Condor Universe 11

2.5 Submitting a Job to Condor 15

2.5.1 Sample submit description files 15

2.5.2 About Requirements and Rank 17

2.5.3 Heterogeneous Submit: Execution on Differing Architectures 25

2.5.4 Vanilla Universe Example for Execution on Differing Architectures 25
2.5.5 Standard Universe Example for Execution on Differing Architectures . . . 27
2.6 Managing a Condor Job 28
2.6.1 Checking on the progress of jobs 29
2.6.2 Removing a job from the queue 31
2.6.3 Changing the priority of jobs 32
2.6.4 Why does the job not run? 32
2.6.5 Job Completion 34
2.7 Priorities in Condor 35
2.7.1 Job Priority 35
2.7.2 User priority 35
2.7.3 Environment Variables 36
2.8 Java Applications 37
2.9 Parallel Applications in Condor: Condor-PVM 41
2.9.1 Effective Usage: the Master-Worker Paradigm 42
2.9.2 Binary Compatibility and Runtime Differences 42
2.9.3 Sample PVM submit file 45
2.10 MPI Support 46
2.10.1 MPI Details of Set Up 47
2.10.2 MPI Job Submission 47
2.11 Extending your Condor pool with Glidein 50
2.11.1 *condor_glidein* Requirements 50
2.11.2 What *condor_glidein* Does 51
2.12 Inter-job Dependencies: DAGMan Meta-Scheduler 52
2.12.1 Input File describing the DAG 52
2.12.2 Condor Submit Description File 56
2.12.3 Job Submission 57
2.12.4 Job Monitoring 57

2.12.5 Job Failure and Job Removal 57
2.12.6 Job Recovery: The Rescue DAG 58
2.13 UserLogViewer 59
2.13.1 Transition States 59
2.13.2 Events 59
2.13.3 Zooming 60
2.13.4 Zooming 60
2.13.5 Keyboard and Mouse Shortcuts 60
2.14 About How Condor Jobs Vacate Machines 60
2.15 Special Environment Considerations 61
2.15.1 AFS 61
2.15.2 NFS Automounter 62
2.15.3 Condor Daemons That Do Not Run as root 62
2.16 Potential Problems 63
2.16.1 Renaming of argv[0] 63

3 Administrators' Manual 64

3.1 Introduction 64
3.1.1 The Different Roles a Machine Can Play 65
3.1.2 The Condor Daemons 66
3.2 Installation of Condor 68
3.2.1 Obtaining Condor 68
3.2.2 Condor Distribution Contents 69
3.2.3 Preparation 69
3.2.4 Installation Procedure 75
3.2.5 Condor is installed... now what? 79
3.2.6 Starting up the Condor daemons 80
3.2.7 The Condor daemons are running... now what? 81

3.3 Configuring Condor 82
3.3.1 Introduction to Configuration Files 82
3.3.2 Condor-wide Configuration File Entries 86
3.3.3 Daemon Logging Config File Entries 89
3.3.4 DaemonCore Config File Entries 91
3.3.5 Shared File System Configuration File Macros 93
3.3.6 Checkpoint Server Configuration File Macros 96
3.3.7 condor_master Configuration File Macros 97
3.3.8 condor_startd Configuration File Macros 100
3.3.9 condor_schedd Configuration File Entries 104
3.3.10 condor_shadow Configuration File Entries 107
3.3.11 condor_shadow.pvm Configuration File Entries 107
3.3.12 condor_starter Configuration File Entries 108
3.3.13 condor_submit Configuration File Entries 109
3.3.14 condor_preen Configuration File Entries 110
3.3.15 condor_collector Configuration File Entries 110
3.3.16 condor_negotiator Configuration File Entries 112
3.3.17 condor_eventd Configuration File Entries 113
3.4 Installing Contrib Modules 113
3.4.1 Installing the CondorView Client Contrib Module 114
3.4.2 Installing a Checkpoint Server 116
3.4.3 Installing PVM Support in Condor 121
3.4.4 Installing the Condor Event Daemon 122
3.4.5 Installing the Condor-G contrib module 124
3.5 User Priorities in the Condor System 124
3.5.1 Real User Priority (RUP) 125
3.5.2 Effective User Priority (EUP) 125
3.5.3 Priorities and Preemption 126

3.5.4 Priority Calculation 126
3.6 Configuring The Startd Policy 127
3.6.1 Startd ClassAd Attributes 127
3.6.2 Job ClassAd Attributes 131
3.6.3 The START expression 132
3.6.4 The RANK expression 133
3.6.5 Machine States 135
3.6.6 Machine Activities 135
3.6.7 State and Activity Transitions 138
3.6.8 State/Activity Transition Expression Summary 143
3.6.9 Policy Settings 144
3.6.10 Differences from the Version 6.0 Policy Settings 149
3.7 Security In Condor 149
3.7.1 UIDs in Condor 150
3.7.2 Running Condor as Non-Root 152
3.7.3 Security Configuration 153
3.7.4 Authentication 161
3.7.5 Authorization 163
3.8 DaemonCore 173
3.8.1 DaemonCore and Unix signals 173
3.8.2 DaemonCore and Command-line Arguments 174
3.9 Managing your Condor Pool 175
3.9.1 Shutting Down and Restarting your Condor Pool 176
3.9.2 Reconfiguring Your Condor Pool 178
3.10 Setting up Condor for Special Environments 178
3.10.1 Using Condor with AFS 178
3.10.2 Configuring Condor for Multiple Platforms 180
3.10.3 Full Installation of condor_compile 182

3.10.4 Installing the *condor_kbdd* 184
3.10.5 Installing a Checkpoint Server 185
3.10.6 Configuring The CondorView Server 190
3.10.7 Flocking: Configuring a Schedd to Submit to Multiple Pools 192
3.10.8 Configuring The Startd for SMP Machines 192
3.10.9 Configuring Condor for Machines With Multiple Network Interfaces . . . 198
3.10.10 Allocating Bandwidth 199
3.10.11 Configuring Condor for Running Dedicated Jobs 202
3.11 Installing Java Support in Condor 206

4 Miscellaneous Concepts **208**
4.1 An Introduction to Condor's ClassAd Mechanism 208
4.1.1 Syntax 209
4.1.2 Evaluation Semantics 210
4.1.3 ClassAds in the Condor System 212
4.2 An Introduction to Condor's Checkpointing Mechanism 214
4.2.1 Standalone Checkpointing 216
4.2.2 Checkpoint Safety 216
4.2.3 Checkpoint Warnings 217
4.2.4 Checkpoint Library Interface 217
4.3 The Condor Perl Module 218
4.3.1 Subroutines 219
4.3.2 Examples 221

5 Condor-G **228**
5.1 Condor-G Introduction 228
5.2 Working with Globus 229
5.2.1 Globus Protocols 229
5.3 Using the Globus Universe 229

5.3.1 Accessing the Grid with Condor-G 229
5.3.2 Condor-G Installation 231
5.3.3 Running a Globus Universe Job 232
5.3.4 Configuration and Credential Management 237
5.4 Limitations of Condor-G 237

6 Condor for Microsoft Windows NT 4.0 239
6.1 Introduction to Condor NT Version 6.4.0 239
6.2 Release Notes for Condor NT Version 6.4.0 240
6.2.1 Details on how Condor NT starts/stops a job 241
6.2.2 Security considerations in Condor NT 243
6.2.3 Interoperability between Condor for Unix and Condor NT 244
6.2.4 Some differences between Condor for Unix -vs- Condor NT 244
6.3 Installation of Condor on Windows NT 245
6.3.1 Installation Requirements 245
6.3.2 Preparing to Install Condor under Windows NT 245
6.3.3 Installation Procedure using the included Setup Program 246
6.3.4 Manual Installation Condor on Windows NT 249
6.3.5 Condor is installed... now what? 250
6.3.6 Condor is running... now what? 251

7 Frequently Asked Questions (FAQ) 252
7.1 Obtaining & Installing Condor 252
7.1.1 Where can I download Condor? 252
7.1.2 When I click to download Condor, it sends me back to the downloads page! 252
7.1.3 What platforms do you support? 252
7.1.4 Do you distribute source code? 253
7.1.5 How do I upgrade the Unix machines in my pool from 6.2.x to 6.4.x? . . . 253
7.1.6 What is “Personal Condor”? 255

7.1.7 What do I do now? My installation of Condor does not work. 255

7.2 Setting up Condor 256

7.2.1 How do I set up a central manager on a machine with multiple network interfaces? 256

7.2.2 How do I get more than one job to run on my SMP machine? 256

7.2.3 How do I set up my machines so that only certain users's jobs will run on them? 256

7.2.4 How do I configure Condor to run my jobs only on machines that have the right packages installed? 257

7.2.5 How do I configure Condor to only run jobs at night? 257

7.2.6 Why will the *condor_master* not run when a local configuration file is missing? 258

7.3 Running Condor Jobs 258

7.3.1 I'm at the University of Wisconsin-Madison Computer Science Dept., and I am having problems! 258

7.3.2 I'm getting a lot of email from Condor. Can I just delete it all? 258

7.3.3 Why will my vanilla jobs only run on the machine where I submitted them from? 259

7.3.4 My job starts but exits right away with signal 9. 259

7.3.5 Why aren't any or all of my jobs running? 259

7.3.6 Can I submit my standard universe SPARC Solaris 2.6 jobs and have them run on a SPARC Solaris 2.7 machine? 260

7.3.7 Why do my vanilla jobs keep cycling between suspended and unsuspended? 260

7.3.8 Why might my job be preempted (evicted)? 260

7.4 Condor on Windows NT / Windows 2000 261

7.4.1 Will Condor work on a network of mixed Unix and NT machines? 261

7.4.2 When I run *condor_status* I get a communication error, or the Condor daemon log files report a failure to bind. 262

7.4.3 My job starts but exits right away with status 128. 262

7.4.4 Why does the startd crash on CondorNT with the error "caInsert: Can't insert CpuBusy into target classad."? 262

7.4.5 How can I access network files with Condor on NT? 262

7.4.6 Does Condor run under Windows 2000? . . . 270
7.5 Troubleshooting . . . 270
7.5.1 What happens if the central manager crashes? . . . 270
7.5.2 When I ssh/telnet to a machine to check particulars of how Condor is doing something, it is always vacating or unclaimed when I know a job had been running there! . . . 270
7.6 Other questions . . . 270
7.6.1 Is Condor Y2K-compliant? . . . 270
7.6.2 Is there a Condor mailing-list? . . . 271
7.6.3 Do you support Globus? . . . 271
7.6.4 What is the process for upgrading from 6.0 to 6.2? . . . 271
7.6.5 My question isn't in the FAQ! . . . 271

8 Condor Version History 272

8.1 Introduction to Condor Versions . . . 272
8.1.1 Condor Version Number Scheme . . . 272
8.1.2 The Stable Release Series . . . 273
8.1.3 The Development Release Series . . . 273
8.2 Stable Release Series 6.4 . . . 273
8.2.1 Version 6.4.3 . . . 274
8.2.2 Version 6.4.2 . . . 274
8.2.3 Version 6.4.1 . . . 274
8.2.4 Version 6.4.0 . . . 275
8.3 Development Release Series 6.3 . . . 275
8.3.1 Version 6.3.4 . . . 276
8.3.2 Version 6.3.3 . . . 276
8.3.3 Version 6.3.2 . . . 279
8.4 Stable Release Series 6.2 . . . 283
8.4.1 Version 6.2.2 . . . 283

8.4.2 Version 6.2.1 285
8.5 Development Release Series 6.1 289
8.5.1 Version 6.1.17 289
8.5.2 Version 6.1.16 291
8.5.3 Version 6.1.15 292
8.5.4 Version 6.1.14 294
8.5.5 Version 6.1.13 295
8.5.6 Version 6.1.12 299
8.5.7 Version 6.1.11 300
8.5.8 Version 6.1.10 302
8.5.9 Version 6.1.9 303
8.5.10 Version 6.1.8 305
8.5.11 Version 6.1.7 308
8.5.12 Version 6.1.6 308
8.5.13 Version 6.1.5 310
8.5.14 Version 6.1.4 312
8.5.15 Version 6.1.3 312
8.5.16 Version 6.1.2 314
8.5.17 Version 6.1.1 315
8.5.18 Version 6.1.0 315
8.6 Stable Release Series 6.0 316
8.6.1 Version 6.0.3 316
8.6.2 Version 6.0.2 317
8.6.3 Version 6.0.1 317
8.6.4 Version 6.0 pl4 318
8.6.5 Version 6.0 pl3 319
8.6.6 Version 6.0 pl2 320
8.6.7 Version 6.0 pl1 321

8.6.8 Version 6.0 pl0 321

9 Command Reference Manual (man pages) 322

condor_advertise 323

condor_checkpoint 325

condor_compile 328

condor_config_val 331

condor_findhost 335

condor_glidein 338

condor_history 342

condor_hold 344

condor_master 347

condor_master_off 348

condor_off 350

condor_on 353

condor_preen 356

condor_prio 358

condor_q 360

condor_qedit 366

condor_reconfig 368

condor_reconfig_schedd 371

condor_release 373

condor_reschedule 375

condor_restart 377

condor_rm 380

condor_run 382

condor_stats 386

condor_status 390

condor_submit 395

condor_submit_dag 411

condor_userlog 414

condor_userprio 418

condor_vacate 421

Copyright and Disclaimer

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program Object Code (Condor) is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

Some distributions of Condor include a compiled, unmodified version of the GNU C library. The complete source code to GNU glibc can be found at http://www.gnu.org/software/libc/.

Allowed Uses: User may use Condor only in accordance with the appropriate Usage License, which are detailed below. Academic institutions should agree to the *Academic Use License for Condor*, while all others should agree to the *Internal Use License for Condor*.

Use Restrictions: User may not and User may not permit others to (a) decipher, disassemble, decompile, translate, reverse engineer or otherwise derive source code from Condor, (b) modify or prepare derivative works of Condor, (c) copy Condor, except to make a single copy for archival purposes only, (d) rent or lease Condor, (e) distribute Condor electronically, (f) use Condor in any manner that infringes the intellectual property or rights of another party, or (g) transfer Condor or any copy thereof to another party.

Warranty Disclaimer: USER ACKNOWLEDGES AND AGREES THAT: (A) NEITHER THE Condor TEAM NOR THE BOARD OF REGENTS OF THE UNIVERSITY OF WISCONSIN SYSTEM (REGENTS) MAKE ANY REPRESENTATIONS OR WARRANTIES WHATSOEVER ABOUT THE SUITABILITY OF Condor FOR ANY PURPOSE; (B) Condor IS PROVIDED ON AN ”AS IS, WITH ALL DEFECTS” BASIS WITHOUT EXPRESS OR IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT; (C) NEITHER THE Condor TEAM NOR THE REGENTS SHALL BE LIABLE FOR ANY DAMAGE OR LOSS OF ANY KIND ARISING OUT OF OR RESULTING FROM USER’S POSSESSION OR USE OF Condor (INCLUDING DATA LOSS OR CORRUPTION), REGARDLESS OF WHETHER SUCH LIABILITY IS BASED IN TORT, CONTRACT, OR OTHERWISE; AND (D) NEITHER THE Condor TEAM NOR THE REGENTS HAVE AN OBLIGATION TO PROVIDE DEBUGGING, MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Damages Disclaimer: USER ACKNOWLEDGES AND AGREES THAT IN NO EVENT WILL THE Condor TEAM OR THE REGENTS BE LIABLE TO USER FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT OR SIMILAR DAMAGES, INCLUDING ANY LOST PROFITS OR LOST DATA ARISING OUT OF THE USE OR INABILITY TO USE Condor EVEN IF THE Condor TEAM OR THE REGENTS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Attribution Requirement: User agrees that any reports, publications, or other disclosure of results obtained

with Condor will attribute its use by an appropriate citation. The appropriate reference for Condor is "The Condor Software Program (Condor) was developed by the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison. All rights, title, and interest in Condor are owned by the Condor Team."

Compliance with Applicable Laws: User agrees to abide by copyright law and all other applicable laws of the United States including, but not limited to, export control laws. User acknowledges that Condor in source code form remains a confidential trade secret of the Condor Team and/or its licensors and therefore User agrees not to modify Condor or attempt to decipher, decompile, disassemble, translate, or reverse engineer Condor, except to the extent applicable laws specifically prohibit such restriction.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

Condor Usage Licenses

Following are licenses for use of Condor Version 6. Academic institutions should agree to the Academic Use License for Condor, while all others should agree to the Internal Use License for Condor.

Internal Use License for Condor

This is an Internal Use License for Condor Version 6. This License is to be signed by RECIPIENT (the "RECIPIENT"), and returned to the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison (the "PROVIDER"). The Condor Version 6 software program was developed by the Condor Team. All rights, title, and interest in Condor Version 6 are owned by the Condor Team. The subject computer program, including source code, executables, and documentation shall be referred to as the "SOFTWARE."

RECIPIENT and PROVIDER agree as follows:

1. Definitions.
   (a) The "Object Code" of the SOFTWARE means the SOFTWARE assembled or compiled in magnetic or electronic binary form on software media, which are readable and usable by machines, but not generally readable by humans without reverse assembly, reverse compiling, or reverse engineering.
   (b) The "Source Code" of the SOFTWARE means the SOFTWARE written in programming languages, such as C and FORTRAN, including all comments and procedural code, such as job control language statements, in a form intelligible to trained programmers and capable of being translated into Object Code for operation on computer equipment through assembly or compiling, and accompanied by documentation, including flow charts, schematics, statements of principles of operations, and architecture standards, describing the data flows, data structures, and control logic of the SOFTWARE in sufficient detail to enable a trained programmer through study of such documentation to maintain and/or modify the SOFTWARE without undue experimentation.

(c) A "Derivative Work" means a work that is based on one or more preexisting works, such as a revision, enhancement, modification, translation, abridgment, condensation, expansion, or any other form in which such preexisting works may be recast, transformed, or adapted, and that, if prepared without authorization of the owner of the copyright in such preexisting work, would constitute a copyright infringement. For purposes hereof, a Derivative Work shall also include any compilation that incorporates such a preexisting work. Unless otherwise provided in this License, all references to the SOFTWARE include any Derivative Works provided by PROVIDER or authorized to be made by RECIPIENT hereunder.

(d) "Support Materials" means documentation that describes the function and use of the SOFTWARE in sufficient detail to permit use of the SOFTWARE.

2. Copying of SOFTWARE and Support Materials. PROVIDER grants RECIPIENT a non-exclusive, non-transferable use license to copy and distribute internally the SOFTWARE and related Support Materials in support of RECIPIENT's use of the SOFTWARE. RECIPIENT agrees to include all copyright, trademark, and other proprietary notices of PROVIDER in each copy of the SOFTWARE as they appear in the version provided to RECIPIENT by PROVIDER. RECIPIENT agrees to maintain records of the number of copies of the SOFTWARE that RECIPIENT makes, uses, or possesses.

3. Use of Object Code. PROVIDER grants RECIPIENT a royalty-free, non-exclusive, non-transferable use license in and to the SOFTWARE, in Object Code form only, to:

   (a) Install the SOFTWARE at RECIPIENT's offices listed below;

   (b) Use and execute the SOFTWARE for research or other internal purposes only;

   (c) In support of RECIPIENT's authorized use of the SOFTWARE, physically transfer the SOFTWARE from one (1) computer to another; store the SOFTWARE's machine-readable instructions or data on a temporary basis in main memory, extended memory, or expanded memory of such computer system as necessary for such use; and transmit such instructions or data through computers and associated devices.

4. Delivery. PROVIDER will deliver to RECIPIENT one (1) executable copy of the SOFTWARE in Object Code form, one (1) full set of the related documentation, and one (1) set of Support Materials relating to the SOFTWARE within fifteen (15) business days after the receipt of the signed License.

5. Back-up Copies. RECIPIENT may make up to two (2) copies of the SOFTWARE in Object Code form for nonproductive backup purposes only.

6. Term of License. The term of this License shall be one (1) year from the date of this License. However, PROVIDER may terminate RECIPIENT's License without cause at any time. All copies of the SOFTWARE, or Derivative Works thereof, shall be destroyed by the RECIPIENT upon termination of this License.

7. Proprietary Protection. PROVIDER shall have sole and exclusive ownership of all right, title, and interest in and to the SOFTWARE and Support Materials, all copies thereof, and all modifications and enhancements thereto (including ownership of all copyrights and other intellectual property rights pertaining thereto). Any modifications or Derivative Works based on the SOFTWARE shall be considered a part of the SOFTWARE and ownership thereof shall be retained by the PROVIDER and shall be made available to the PROVIDER upon request. This License does not provide RECIPIENT with title or ownership of the SOFTWARE, but only a right of internal use.

8. Limitations on Use, Etc. RECIPIENT may not use, copy, modify, or distribute the SOFTWARE (electronically or otherwise) or any copy, adaptation, transcription, or merged portion thereof, except as expressly authorized in this License. RECIPIENT's license may not be transferred, leased, assigned, or sublicensed without PROVIDER's prior express authorization.

9. Data. RECIPIENT acknowledges that data conversion is subject to the likelihood of human and machine errors, omissions, delays, and losses, including inadvertent loss of data or damage to media, that may give rise to loss or damage. PROVIDER shall not be liable for any such errors, omissions, delays, or losses, whatsoever. RECIPIENT is also responsible for complying with all local, state, and federal laws pertaining to the use and disclosure of any data.
10. Warranty Disclaimer. RECIPIENT ACKNOWLEDGES AND AGREES THAT: (A) NEITHER THE CONDOR TEAM NOR THE BOARD OF REGENTS OF THE UNIVERSITY OF WISCONSIN SYSTEM (REGENTS) MAKE ANY REPRESENTATIONS OR WARRANTIES WHATSOEVER ABOUT THE SUITABILITY OF THE SOFTWARE FOR ANY PURPOSE; (B) THE SOFTWARE IS PROVIDED ON AN "AS IS, WITH ALL DEFECTS" BASIS WITHOUT EXPRESS OR IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT; (C) NEITHER THE CONDOR TEAM NOR THE REGENTS SHALL BE LIABLE FOR ANY DAMAGE OR LOSS OF ANY KIND ARISING OUT OF OR RESULTING FROM RECIPIENT'S POSSESSION OR USE OF THE SOFTWARE (INCLUDING DATA LOSS OR CORRUPTION), REGARDLESS OF WHETHER SUCH LIABILITY IS BASED IN TORT, CONTRACT, OR OTHERWISE; AND (D) NEITHER THE CONDOR TEAM NOR THE REGENTS HAVE AN OBLIGATION TO PROVIDE DEBUGGING, MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
11. Damages Disclaimer. RECIPIENT ACKNOWLEDGES AND AGREES THAT IN NO EVENT WILL THE CONDOR TEAM OR THE REGENTS BE LIABLE TO RECIPIENT FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT OR SIMILAR DAMAGES, INCLUDING ANY LOST PROFITS OR LOST DATA ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF THE CONDOR TEAM OR THE REGENTS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
12. Compliance with Applicable Laws. RECIPIENT agrees to abide by copyright law and all other applicable laws of the United States including, but not limited to, export control laws.
13. U.S. Government Rights Restrictions. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.
14. Governing Law. This License shall be governed by and construed and enforced in accordance with the laws of the State of Wisconsin as it applies to a contract made and performed in such state, except to the extent such laws are in conflict with federal law.
15. Modifications and Waivers. This License may not be modified except by a writing signed by authorized representatives of both parties. A waiver by either party of its rights hereunder shall not be binding unless contained in a writing signed by an authorized representative of the party waiving its rights. The nonenforcement or waiver of any provision on one (1) occasion shall not constitute a waiver of such provision on any other occasions unless expressly so agreed in writing. It is agreed that no use of trade or other regular practice or method of dealing between the parties hereto shall be used to modify, interpret, supplement, or alter in any manner the terms of this License.

Academic Use License for Condor

This is an Academic Object Code Use License for Condor. This license is between you (the "RECIPIENT"), and the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison (the

”PROVIDER”). The Condor software program was developed by the Condor Team. All rights, title, and interest in Condor are owned by the Condor Team. The subject computer program, including executables and supporting documentation, shall be referred to as the ”SOFTWARE”.

RECIPIENT and PROVIDER agree as follows:

1. A non-exclusive, non-transferable academic use license is granted to the RECIPIENT to install and use the SOFTWARE on any appropriate computer systems located at the RECIPIENT's institution to which the RECIPIENT has authorized access. Use of the SOFTWARE is restricted to the RECIPIENT and collaborators at RECIPIENT's institution who have agreed to accept the terms of this license.
2. The PROVIDER shall retain ownership of all materials (including magnetic tape, unless provided by the RECIPIENT) and SOFTWARE delivered to the RECIPIENT. Any modifications or derivative works based on the SOFTWARE shall be considered part of the SOFTWARE and ownership thereof shall be retained by the PROVIDER and shall be made available to the PROVIDER upon request.
3. The RECIPIENT may make a reasonable number of copies of the SOFTWARE for the purpose of backup and maintenance of the SOFTWARE, or for development of derivative works based on the SOFTWARE. The RECIPIENT agrees to include all copyright or trademark notices on any copies of the SOFTWARE or derivatives thereof. All copies of the SOFTWARE, or derivatives thereof, shall be destroyed by the RECIPIENT upon termination of this license.
4. The RECIPIENT shall use the SOFTWARE for research, educational, or other non-commercial purposes only. The RECIPIENT acknowledges that this license grants no rights whatsoever for commercial use of the SOFTWARE or in any commercial version(s) of the SOFTWARE. The RECIPIENT is strictly prohibited from deciphering, disassembling, decompiling, translating, reverse engineering or otherwise deriving source code from the SOFTWARE, except to the extent applicable laws specifically prohibit such restriction.
5. The RECIPIENT shall not disclose in any form either the delivered SOFTWARE or any modifications or derivative works based on the SOFTWARE to any third party without prior express authorization from the PROVIDER.
6. If the RECIPIENT receives a request to furnish all or any portion of the SOFTWARE to any third party, RECIPIENT shall not fulfill such a request, and further agrees to refer the request to the PROVIDER.
7. The RECIPIENT agrees that the SOFTWARE is furnished on an ”as is, with all defects” basis, without maintenance, debugging, support or improvement, and that neither the PROVIDER nor the Board of Regents of the University of Wisconsin System warrant the SOFTWARE or any of its results and are in no way liable for any use that the RECIPIENT makes of the SOFTWARE.
8. The RECIPIENT agrees that any reports, publications, or other disclosure of results obtained with the SOFTWARE will acknowledge its use by an appropriate citation. The appropriate reference for the SOFTWARE is ”The Condor Software Program (Condor) was developed by the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison. All rights, title, and interest in Condor are owned by the Condor Team.”
9. The term of this license shall not be limited in time. However, PROVIDER may terminate RECIPIENT's license without cause at any time.
10. Source code for the SOFTWARE is available upon request and at the sole discretion of the PROVIDER.
11. This license shall be construed and governed in accordance with the laws of the State of Wisconsin.

For more information:
Condor Team

Attention: Professor Miron Livny
7367 Computer Sciences
1210 W. Dayton St.
Madison, WI 53706-1685
miron@cs.wisc.edu
http://www.cs.wisc.edu/~miron/miron.html

CHAPTER

ONE

Overview

1.1 High-Throughput Computing (HTC) and its Requirements

For many research and engineering projects, the quality of the research or the product is heavily dependent upon the quantity of computing cycles available. It is not uncommon to find problems that require weeks or months of computation to solve. Scientists and engineers engaged in this sort of work need a computing environment that delivers large amounts of computational power over a long period of time. Such an environment is called a High-Throughput Computing (HTC) environment. In contrast, High Performance Computing (HPC) environments deliver a tremendous amount of compute power over a short period of time. HPC environments are often measured in terms of FLoating point Operations Per Second (FLOPS). A growing community is not concerned about operations per second, but operations per month or per year. Their problems are of a much larger scale. They are more interested in how many jobs they can complete over a long period of time instead of how fast an individual job can complete.

The key to HTC is to efficiently harness the use of all available resources. Years ago, the engineering and scientific community relied on a large, centralized mainframe or a supercomputer to do computational work. A large number of individuals and groups needed to pool their financial resources to afford such a machine. Users had to wait for their turn on the mainframe, and they had a limited amount of time allocated. While this environment was inconvenient for users, the utilization of the mainframe was high; it was busy nearly all the time.

As computers became smaller, faster, and cheaper, users moved away from centralized mainframes and purchased personal desktop workstations and PCs. An individual or small group could afford a computing resource that was available whenever they wanted it. The personal computer is slower than the large centralized machine, but it provides exclusive access. Now, instead of one giant computer for a large institution, there may be hundreds or thousands of personal computers. This

is an environment of distributed ownership, where individuals throughout an organization own their own resources. The total computational power of the institution as a whole may rise dramatically as the result of such a change, but because of distributed ownership, individuals have not been able to capitalize on the institutional growth of computing power. And, while distributed ownership is more convenient for the users, the utilization of the computing power is lower. Many personal desktop machines sit idle for very long periods of time while their owners are busy doing other things (such as being away at lunch, in meetings, or at home sleeping).

1.2 Condor's Power

Condor is a software system that creates a High-Throughput Computing (HTC) environment. It effectively utilizes the computing power of workstations that communicate over a network. Condor can manage a dedicated cluster of workstations. Its power comes from the ability to effectively harness non-dedicated, preexisting resources under distributed ownership.

A user submits the job to Condor. Condor finds an available machine on the network and begins running the job on that machine. Condor has the capability to detect that a machine running a Condor job is no longer available (perhaps because the owner of the machine came back from lunch and started typing on the keyboard). It can checkpoint the job and move (migrate) the jobs to a different machine which would otherwise be idle. Condor continues job on the new machine from precisely where it left off.

In those cases where Condor can checkpoint and migrate a job, Condor makes it easy to maximize the number of machines which can run a job. In this case, there is no requirement for machines to share file systems (for example, with NFS or AFS), so that machines across an entire enterprise can run a job, including machines in different administrative domains.

Condor can be a real time saver when a job must be run many (hundreds of) different times, perhaps with hundreds of different data sets. With one command, all of the hundreds of jobs are submitted to Condor. Depending upon the number of machines in the Condor pool, dozens or even hundreds of otherwise idle machines can be running the job at any given moment.

Condor does not require an account (login) on machines where it runs a job. Condor can do this because of its *remote system call* technology, which traps library calls for such operations as reading or writing from disk files. The calls are transmitted over the network to be performed on the machine where the job was submitted.

Condor provides powerful resource management by match-making resource owners with resource consumers. This is the cornerstone of a successful HTC environment. Other compute cluster resource management systems attach properties to the job queues themselves, resulting in user confusion over which queue to use as well as administrative hassle in constantly adding and editing queue properties to satisfy user demands. Condor implements *ClassAds*, a clean design that simplifies the user's submission of jobs.

ClassAds work in a fashion similar to the newspaper classified advertising want-ads. All machines in the Condor pool advertise their resource properties, both static and dynamic, such as

available RAM memory, CPU type, CPU speed, virtual memory size, physical location, and current load average, in a *resource offer* ad. A user specifies a *resource request* ad when submitting a job. The request defines both the required and a desired set of properties of the resource to run the job. Condor acts as a broker by matching and ranking resource offer ads with resource request ads, making certain that all requirements in both ads are satisfied. During this match-making process, Condor also considers several layers of priority values: the priority the user assigned to the resource request ad, the priority of the user which submitted the ad, and desire of machines in the pool to accept certain types of ads over others.

1.3 Exceptional Features

Checkpoint and Migration. Where programs can be linked with Condor libaries, users of Condor may be assured that their jobs will eventually complete, even in the ever changing environment that Condor utilizes. As a machine running a job submitted to Condor becomes unavailable, the job can be checkpointed. The job may continue after migrating to another machine. Condor's periodic checkpoint feature periodically checkpoints a job even in lieu of migration in order to safeguard the accumulated computation time on a job from being lost in the event of a system failure such as the machine being shutdown or a crash.

Remote System Calls. Despite running jobs on remote machines, the Condor standard universe execution mode preserves the local execution environment via remote system calls. Users do not have to worry about making data files available to remote workstations or even obtaining a login account on remote workstations before Condor executes their programs there. The program behaves under Condor as if it were running as the user that submitted the job on the workstation where it was originally submitted, no matter on which machine it really ends up executing on.

No Changes Necessary to User's Source Code. No special programming is required to use Condor. Condor is able to run non-interactive programs. The checkpoint and migration of programs by Condor is transparent and automatic, as is the use of remote system calls. If these facilities are desired, the user only re-links the program. The code is neither recompiled nor changed.

Pools of Machines can be Hooked Together. Flocking is a feature of Condor that allows jobs submitted within a first pool of Condor machines to execute on a second pool. The mechanism is flexible, following requests from the job submission, while allowing the second pool, or a subset of machines within the second pool to set policies over the conditions under which jobs are executed.

Jobs can be Ordered. The ordering of job execution required by dependencies among jobs in a set is easily handled. The set of jobs is specified using a directed acyclic graph, where each job is a node in the graph. Jobs are submitted to Condor following the dependencies given by the graph.

Condor Enables Grid Computing. As grid computing becomes a reality, Condor is already there. The technique of glidein allows jobs submitted to Condor to be executed on grid machines

in various locations worldwide. As the details of grid computing evolve, so does Condor's ability, starting with Globus-controlled resources.

Sensitive to the Desires of Machine Owners. The owner of a machine has complete priority over the use of the machine. An owner is generally happy to let others compute on the machine while it is idle, but wants it back promptly upon returning. The owner does not want to take special action to regain control. Condor handles this automatically.

ClassAds. The ClassAd mechanism in Condor provides an extremely flexible, expressive framework for matchmaking resource requests with resource offers. Users can easily request both job requirements and job desires. For example, a user can require that a job run on a machine with 64 Mbytes of RAM, but state a preference for 128 Mbytes, if available. A workstation owner can state a preference that the workstation runs jobs from a specified set of users. The owner can also require that there be no interactive workstation activity detectable at certain hours before Condor could start a job. Job requirements/preferences and resource availability constraints can be described in terms of powerful expressions, resulting in Condor's adaptation to nearly any desired policy.

1.4 Current Limitations

Limitations on Jobs which can Checkpointed Although Condor can schedule and run any type of process, Condor does have some limitations on jobs that it can transparently checkpoint and migrate:

1. Multi-process jobs are not allowed. This includes system calls such as `fork()`, `exec()`, and `system()`.
2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.
3. Network communication must be brief. A job *may* make network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpointing and migration.
4. Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals *is* allowed.
5. Alarms, timers, and sleeping are not allowed. This includes system calls such as `alarm()`, `getitimer()`, and `sleep()`.
6. Multiple kernel-level threads are not allowed. However, multiple user-level threads *are* allowed.
7. Memory mapped files are not allowed. This includes system calls such as `mmap()` and `munmap()`.
8. File locks are allowed, but not retained between checkpoints.

9. All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.
10. A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special *checkpoint server* can be designated for storing all the checkpoint images for a pool.
11. On Digital Unix (OSF/1), HP-UX, and Linux, your job must be statically linked. Dynamic linking is allowed on all other platforms.

Note: these limitations *only* apply to jobs which Condor has been asked to transparently checkpoint. If job checkpointing is not desired, the limitations above do not apply.

Security Implications. Condor does a significant amount of work to prevent security hazards, but loopholes are known to exist. Condor can be instructed to run user programs only as the UNIX user nobody, a user login which traditionally has very restricted access. But even with access solely as user nobody, a sufficiently malicious individual could do such things as fill up /tmp (which is world writable) and/or gain read access to world readable files. Furthermore, where the security of machines in the pool is a high concern, only machines where the UNIX user root on that machine can be trusted should be admitted into the pool. Condor provides the administrator with IP-based security mechanisms to enforce this.

Jobs Need to be Re-linked to get Checkpointing and Remote System Calls Although typically no source code changes are required, Condor requires that the jobs be re-linked with the Condor libraries to take advantage of checkpointing and remote system calls. This often precludes commercial software binaries from taking advantage of these services because commercial packages rarely make their object code available. Condor's other services are still available for these commercial packages.

1.5 Availability

Condor is currently available as a free download from the Internet via the World Wide Web at URL http://www.cs.wisc.edu/condor/downloads. Binary distributions of Condor version 6.x are available for the platforms detailed in Table 1.1. A platform is an architecture/operating system combination. Condor binaries are available most major versions of UNIX, as well as Windows NT.

In the table, *clipped* means that Condor does not support checkpointing or remote system calls on the given platform. This means that *standard* jobs are not supported, only *vanilla* jobs. See section 2.4.1 on page 11 for more details on job universes within Condor and their abilities and limitations.

The Condor source code is no longer available for public download from the Internet. If you desire the Condor source code, please contact the Condor Team in order to discuss it further (see Section 1.6, on page 6).

Architecture	*Operating System*
Hewlett Packard PA-RISC (both PA7000 and PA8000 series)	HPUX 10.20
Sun SPARC Sun4m,c, Sun UltraSPARC	Solaris 2.5.x, 2.6, 2.7, 2.8
Silicon Graphics MIPS (R4400, R4600, R8000, R10000)	IRIX 6.5
Intel x86	RedHat Linux 5.2, 6.x, 7.1 Solaris 2.5.x, 2.6, 2.7 Windows NT 4.0 (“clipped”)
Digital ALPHA	OSF/1 (Digital Unix) 4.x to 5.0a Linux 2.2.x

Table 1.1: Condor Version 6.4.0 supported platforms

NOTE: Other Linux distributions (Debian, etc.) may work, but are not tested or supported.

1.6 Contact Information

The latest software releases, publications/papers regarding Condor and other High-Throughput Computing research can be found at the official web site for Condor at http://www.cs.wisc.edu/condor.

In addition, there is an e-mail list at condor-world@cs.wisc.edu. The Condor Team uses this e-mail list to announce new releases of Condor and other major Condor-related news items. Membership into condor-world is automated by MajorDomo software. To subscribe or unsubscribe from the the list, follow the instructions at http://www.cs.wisc.edu/condor/condor-world/condor-world.html. Because many of us receive too much e-mail as it is, you’ll be happy to know that the condor-world e-mail listgroup is moderated and only major announcements of wide interest are distributed.

Finally, you can reach the Condor Team directly. The Condor Team is comprised of the developers and administrators of Condor at the University of Wisconsin-Madison. Condor questions, comments, pleas for help, and requests for commercial contract consultation or support are all welcome; just send Internet e-mail to mailto:condor-admin@cs.wisc.edu. Please include your name, organization, and telephone number in your message. If you are having trouble with Condor, please help us troubleshoot by including as much pertinent information as you can, including snippets of Condor log files.

CHAPTER

TWO

Users' Manual

2.1 Welcome to Condor

Presenting Condor Version 6.4.0! Condor is developed by the Condor Team at the University of Wisconsin-Madison (UW-Madison), and was first installed as a production system in the UW-Madison Computer Sciences department nearly 10 years ago. This Condor pool has since served as a major source of computing cycles to UW faculty and students. For many, it has revolutionized the role computing plays in their research. An increase of one, and sometimes even two, orders of magnitude in the computing throughput of a research organization can have a profound impact on its size, complexity, and scope. Over the years, the Condor Team has established collaborations with scientists from around the world and has provided them with access to surplus cycles (one of whom has consumed 100 CPU years!). Today, our department's pool consists of more than 700 desktop Unix workstations. On a typical day, our pool delivers more than 500 CPU days to UW researchers. Additional Condor pools have been established over the years across our campus and the world. Groups of researchers, engineers, and scientists have used Condor to establish compute pools ranging in size from a handful to hundreds of workstations. We hope that Condor will help revolutionize your compute environment as well.

2.2 What does Condor do?

In a nutshell, Condor is a specialized batch system for managing compute-intensive jobs. Like most batch systems, Condor provides a queueing mechanism, scheduling policy, priority scheme, and resource classifications. Users submit their compute jobs to Condor, Condor puts the jobs in a queue, runs them, and then informs the user as to the result.

Batch systems normally operate only with dedicated machines. Often termed compute servers, these dedicated machines are typically owned by one organization and dedicated to the sole purpose of running compute jobs. Condor can schedule jobs on dedicated machines. But unlike traditional batch systems, Condor is also designed to effectively utilize non-dedicated machines to run jobs. By being told to only run compute jobs on machines which are currently not being used (no keyboard activity, no load average, no active telnet users, etc), Condor can effectively harness otherwise idle machines throughout a pool of machines. This is important because often times the amount of compute power represented by the aggregate total of all the non-dedicated desktop workstations sitting on people's desks throughout the organization is far greater than the compute power of a dedicated central resource.

Condor has several unique capabilities at its disposal which are geared towards effectively utilizing non-dedicated resources that are not owned or managed by a centralized resource. These include transparent process checkpoint and migration, remote system calls, and ClassAds. Read section 1.2 for a general discussion of these features before reading any further.

2.3 Condor Matchmaking with ClassAds

Before you learn about how to submit a job, it is important to understand how Condor allocates resources. Understanding the unique framework by which Condor matches submitted jobs with machines is the key to getting the most from Condor's scheduling algorithm.

Condor simplifies job submission by acting as a matchmaker of ClassAds. Condor's ClassAds are analogous to the classified advertising section of the newspaper. Sellers advertise specifics about what they have to sell, hoping to attract a buyer. Buyers may advertise specifics about what they wish to purchase. Both buyers and sellers list constraints that need to be satisfied. For instance, a buyer has a maximum spending limit, and a seller requires a minimum purchase price. Furthermore, both want to rank requests to their own advantage. Certainly a seller would rank one offer of $50 dollars higher than a different offer of $25. In Condor, users submitting jobs can be thought of as buyers of compute resources and machine owners are sellers.

All machines in a Condor pool advertise their attributes, such as available RAM memory, CPU type and speed, virtual memory size, current load average, along with other static and dynamic properties. This machine ClassAd also advertises under what conditions it is willing to run a Condor job and what type of job it would prefer. These policy attributes can reflect the individual terms and preferences by which all the different owners have graciously allowed their machine to be part of the Condor pool. You may advertise that your machine is only willing to run jobs at night and when there is no keyboard activity on your machine. In addition, you may advertise a preference (rank) for running jobs submitted by you or one of your co-workers.

Likewise, when submitting a job, you specify a ClassAd with your requirements and preferences. The ClassAd includes the type of machine you wish to use. For instance, perhaps you are looking for the fastest floating point performance available. You want Condor to rank available machines based upon floating point performance. Or, perhaps you care only that the machine has a minimum of 128 Mbytes of RAM. Or, perhaps you will take any machine you can get! These job attributes

and requirements are bundled up into a job ClassAd.

Condor plays the role of a matchmaker by continuously reading all the job ClassAds and all the machine ClassAds, matching and ranking job ads with machine ads. Condor makes certain that all requirements in both ClassAds are satisfied.

2.3.1 Inspecting Machine ClassAds with condor_status

Once Condor is installed, you will get a feel for what a machine ClassAd does by trying the *condor_status* command. Try the *condor_status* command to get a summary of information from ClassAds about the resources available in your pool. Type *condor_status* and hit enter to see a summary similar to the following:

```
Name       Arch     OpSys        State      Activity   Loa-
dAv Mem  ActvtyTime

adriana.cs INTEL    SOLARIS251   Claimed    Busy       1.000  64    0+01:10:00
alfred.cs. INTEL    SOLARIS251   Claimed    Busy       1.000  64    0+00:40:00
amul.cs.wi SUN4u    SOLARIS251   Owner      Idle       1.000  128   0+06:20:04
anfrom.cs. SUN4x    SOLARIS251   Claimed    Busy       1.000  32    0+05:16:22
anthrax.cs INTEL    SOLARIS251   Claimed    Busy       0.285  64    0+00:00:00
astro.cs.w INTEL    SOLARIS251   Claimed    Busy       0.949  64    0+05:30:00
aura.cs.wi SUN4u    SOLARIS251   Owner      Idle       1.043  128   0+14:40:15

                                ...
```

The *condor_status* command has options that summarize machine ads in a variety of ways. For example,

condor_status -available shows only machines which are willing to run jobs now.

condor_status -run shows only machines which are currently running jobs.

condor_status -l lists the machine ClassAds for all machines in the pool.

Refer to the *condor_status* command reference page located on page 390 for a complete description of the *condor_status* command.

Figure 2.1 shows the complete machine ClassAd for a single workstation: alfred.cs.wisc.edu. Some of the listed attributes are used by Condor for scheduling. Other attributes are for information purposes. An important point is that *any* of the attributes in a machine ad can be utilized at job submission time as part of a request or preference on what machine to use. Additional attributes can be easily added. For example, your site administrator can add a physical location attribute to your machine ClassAds.

```
MyType = "Machine"
TargetType = "Job"
Name = "alfred.cs.wisc.edu"
Machine = "alfred.cs.wisc.edu"
StartdIpAddr = "<128.105.83.11:32780>"
Arch = "INTEL"
OpSys = "SOLARIS251"
UidDomain = "cs.wisc.edu"
FileSystemDomain = "cs.wisc.edu"
State = "Unclaimed"
EnteredCurrentState = 892191963
Activity = "Idle"
EnteredCurrentActivity = 892191062
VirtualMemory = 185264
Disk = 35259
KFlops = 19992
Mips = 201
LoadAvg = 0.019531
CondorLoadAvg = 0.000000
KeyboardIdle = 5124
ConsoleIdle = 27592
Cpus = 1
Memory = 64
AFSCell = "cs.wisc.edu"
START = LoadAvg - CondorLoadAvg <= 0.300000 && KeyboardI-
dle > 15 * 60
Requirements = TRUE
Rank = Owner == "johndoe" || Owner == "friendofjohn"
CurrentRank =  - 1.000000
LastHeardFrom = 892191963
```

Figure 2.1: Sample output from *condor_status -l alfred*

2.4 Road-map for running jobs with Condor

The road to using Condor effectively is a short one. The basics are quickly and easily learned.

Here are all the steps needed to run a job using Condor.

Code Preparation. A job run under Condor must be able to run as a background batch job. Condor runs the program unattended and in the background. A program that runs in the background will not be able to do interactive input and output. Condor can redirect console output (stdout and stderr) and keyboard input (stdin) to and from files for you. Create any needed files that contain the proper keystrokes needed for program input. Make certain the program will run correctly with the files.

The Condor Universe. Condor has several runtime environments (called a *universe*) from which to choose. Of the universes, two are likely choices when learning to submit a job to Condor: the standard universe and the vanilla universe. The standard universe allows a job running under Condor to handle system calls by returning them to the machine where the job was submitted. The standard universe also provides the mechanisms necessary to take a checkpoint and migrate a partially completed job, should the machine on which the job is executing become unavailable. To use the standard universe, it is necessary to relink the program with the Condor library using the *condor_compile* command. The manual page for *condor_compile* on page 328 has details.

The vanilla universe provides a way to run jobs that cannot be relinked. It depends on a shared file system for access to input and output files, and there is no way to take a checkpoint or migrate a job executed under the vanilla universe.

Choose a universe under which to run the Condor program, and re-link the program if necessary.

Submit description file. Controlling the details of a job submission is a submit description file. The file contains information about the job such as what executable to run, the files to use for keyboard and screen data, the platform type required to run the program, and where to send e-mail when the job completes. You can also tell Condor how many times to run a program; it is simple to run the same program multiple times with multiple data sets.

Write a submit description file to go with the job, using the examples provided in section 2.5.1 for guidance.

Submit the Job. Submit the program to Condor with the *condor_submit* command.

Once submitted, Condor does the rest toward running the job. Monitor the job's progress with the *condor_q* and *condor_status* commands. You may modify the order in which Condor will run your jobs with *condor_prio*. If desired, Condor can even inform you in a log file every time your job is checkpointed and/or migrated to a different machine.

When your program completes, Condor will tell you (by e-mail, if preferred) the exit status of your program and various statistics about its performances, including time used and I/O performed. If you are using a log file for the job(which is recommended) the exit status will be recorded in the log file. You can remove a job from the queue prematurely with *condor_rm*.

2.4.1 Choosing a Condor Universe

A *universe* in Condor defines an execution environment. Condor Version 6.4.0 supports several different universes for user jobs:

- Standard
- Vanilla
- PVM

- MPI
- Globus
- Java

The `Universe` attribute is specified in the submit description file. If a universe is not specified, the default is `standard`.

The `standard` universe provides migration and reliability, but has some restrictions on the programs that can be run. The `vanilla` universe provides fewer services, but has very few restrictions. The `PVM` universe is for programs written to the Parallel Virtual Machine interface. See section 2.9 for more about PVM and Condor. The `MPI` universe is for programs written to the MPICH interface. See section 2.10 for more about MPI and Condor. The Globus universe allows users to submit Globus jobs through the Condor interface. See http://www.globus.org for more about Globus. The Java universe allows users to run jobs written for the Java Virtual Machine (JVM).

Standard Universe

In the standard universe, Condor provides *checkpointing* and *remote system calls*. These features make a job more reliable and allow it uniform access to resources from anywhere in the pool. To prepare a program as a standard universe job, it must be relinked with *condor_compile*. Most programs can be prepared as a standard universe job, but there are a few restrictions.

Condor checkpoints a job at regular intervals. A *checkpoint image* is essentially a snapshot of the current state of a job. If a job must be migrated from one machine to another, Condor makes a checkpoint image, copies the image to the new machine, and restarts the job continuing the job from where it left off. If a machine should crash or fail while it is running a job, Condor can restart the job on a new machine using the most recent checkpoint image. In this way, jobs can run for months or years even in the face of occasional computer failures.

Remote system calls make a job perceive that it is executing on its home machine, even though the job may execute on many different machines over its lifetime. When a job runs on a remote machine, a second process, called a *condor_shadow* runs on the machine where the job was submitted. When the job attempts a system call, the *condor_shadow* performs the system call instead and sends the results to the remote machine. For example, if a job attempts to open a file that is stored on the submitting machine, the *condor_shadow* will find the file, and send the data to the machine where the job is running.

To convert your program into a standard universe job, you must use *condor_compile* to relink it with the Condor libraries. Put *condor_compile* in front of your usual link command. You do not need to modify the program's source code, but you do need access to the unlinked object files. A commercial program that is packaged as a single executable file cannot be converted into a standard universe job.

For example, if you would have linked the job by executing:

```
% cc main.o tools.o -o program
```

Then, relink the job for Condor with:

```
% condor_compile cc main.o tools.o -o program
```

There are a few restrictions on standard universe jobs:

1. Multi-process jobs are not allowed. This includes system calls such as `fork()`, `exec()`, and `system()`.
2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.
3. Network communication must be brief. A job *may* make network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpointing and migration.
4. Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals *is* allowed.
5. Alarms, timers, and sleeping are not allowed. This includes system calls such as `alarm()`, `getitimer()`, and `sleep()`.
6. Multiple kernel-level threads are not allowed. However, multiple user-level threads *are* allowed.
7. Memory mapped files are not allowed. This includes system calls such as `mmap()` and `munmap()`.
8. File locks are allowed, but not retained between checkpoints.
9. All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.
10. A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special *checkpoint server* can be designated for storing all the checkpoint images for a pool.
11. On Digital Unix (OSF/1), HP-UX, and Linux, your job must be statically linked. Dynamic linking is allowed on all other platforms.

Vanilla Universe

The vanilla universe in Condor is intended for programs which cannot be successfully re-linked. Shell scripts are another case where the vanilla universe is useful. Unfortunately, jobs run under the vanilla universe cannot checkpoint or use remote system calls. This has unfortunate consequences for a job that is partially completed when the remote machine running a job must be returned to its owner. Condor has only two choices. It can suspend the job, hoping to complete it at a later time, or it can give up and restart the job *from the beginning* on another machine in the pool.

Notice:

Under Unix, jobs submitted as vanilla universe jobs rely on an external mechanism for accessing data files, such as NFS or AFS. The job *must* be able to access the data files from any machine on which it could potentially run. As an example, suppose a job is submitted from blackbird.cs.wisc.edu, and the job requires a particular data file called `/u/p/s/psilord/data.txt`. If the job were to run on cardinal.cs.wisc.edu, the file `/u/p/s/psilord/data.txt` must be available through either NFS or AFS for the job to run correctly.
Condor deals with this restriction imposed by the vanilla universe by using the `FileSystemDomain` and `UidDomain` machine ClassAd attributes. These attributes reflect the reality of the pool's disk mounting structure. For a large pool spanning multiple `UidDomain` and/or `FileSystemDomain`s, the job must specify its `requirements` to use the correct `UidDomain` and/or `FileSystemDomain`s.
This mechanism is not required under Windows NT. The vanilla universe does *not* require a shared file system due to the Condor File Transfer mechanism. Please see chapter 6 for more details about Condor NT.

PVM Universe

The PVM universe allows programs written for the Parallel Virtual Machine interface to be used within the opportunistic Condor environment. Please see section 2.9 for more details.

MPI Universe

The MPI universe allows programs written to the MPICH interface to be used within the opportunistic Condor environment. Please see section 2.10 for more details.

Globus Universe

The Globus universe in Condor is intended to provide the standard Condor interface to users who wish to start Globus system jobs from Condor. Each job queued in the job submission file is translated into a Globus RSL and subsequently submitted to Globus via the GRAM protocol. The manual page for *condor_submit* on page 395 has detailed descriptions of the Globus-related attributes.

Java Universe

A program submitted to the Java universe may run on any sort of machine with a JVM regardless of its location, owner, or JVM version. Condor will take care of all the details such as finding the JVM binary and setting the classpath.

2.5 Submitting a Job to Condor

A job is submitted for execution to Condor using the *condor_submit* command. *condor_submit* takes as an argument the name of a file called a submit description file. This file contains commands and keywords to direct the queuing of jobs. In the submit description file, Condor finds everything it needs to know about the job. Items such as the name of the executable to run, the initial working directory, and command-line arguments to the program all go into the submit description file. *condor_submit* creates a job ClassAd based upon the information, and Condor works toward running the job.

The contents of a submit file can save time for Condor users. It is easy to submit multiple runs of a program to Condor. To run the same program 500 times on 500 different input data sets, arrange your data files accordingly so that each run reads its own input, and each run writes its own output. Each individual run may have its own initial working directory, stdin, stdout, stderr, command-line arguments, and shell environment. A program that directly opens its own files will read the file names to use either from stdin or from the command line. A program that opens a static filename every time will need to use a separate subdirectory for the output of each run.

The *condor_submit* manual page is on page 395 and contains a complete and full description of how to use *condor_submit*.

2.5.1 Sample submit description files

In addition to the examples of submit description files given in the *condor_submit* manual page, here are a few more.

Example 1

Example 1 is the simplest submit description file possible. It queues up one copy of the program *foo*(which had been created by *condor_compile*) for execution by Condor. Since no platform is specified, Condor will use its default, which is to run the job on a machine which has the same architecture and operating system as the machine from which it was submitted. No `input`, `output`, and `error` commands are given in the submit description file, so the files `stdin`, `stdout`, and `stderr` will all refer to `/dev/null`. The program may produce output by explicitly opening a file and writing to it. A log file, `foo.log`, will also be produced that contains events the job had

during its lifetime inside of Condor. When the job finishes, its exit conditions will be noted in the log file. It is recommended that you always have a log file so you know what happened to your jobs.

```
####################
#
# Example 1
# Simple condor job description file
#
####################

Executable     = foo
Log            = foo.log
Queue
```

Example 2

Example 2 queues two copies of the program *mathematica*. The first copy will run in directory `run_1`, and the second will run in directory `run_2`. For both queued copies, `stdin` will be `test.data`, `stdout` will be `loop.out`, and `stderr` will be `loop.error`. There will be two sets of files written, as the files are each written to their own directories. This is a convenient way to organize data if you have a large group of Condor jobs to run. The example file shows program submission of *mathematica* as a vanilla universe job. This may be necessary if the source and/or object code to program *mathematica* is not available.

```
####################
#
# Example 2: demonstrate use of multiple
# directories for data organization.
#
####################

Executable     = mathematica
Universe = vanilla
input   = test.data
output  = loop.out
error   = loop.error
Log     = loop.log

Initialdir     = run_1
Queue

Initialdir     = run_2
Queue
```

Example 3

The submit description file for Example 3 queues 150 runs of program *foo* which has been compiled and linked for Silicon Graphics workstations running IRIX 6.5. This job requires Condor to run the program on machines which have greater than 32 megabytes of physical memory, and expresses a preference to run the program on machines with more than 64 megabytes, if such machines are available. It also advises Condor that it will use up to 28 megabytes of memory when running. Each of the 150 runs of the program is given its own process number, starting with process number 0. So, files `stdin`, `stdout`, and `stderr` will refer to `in.0`, `out.0`, and `err.0` for the first run of the program, `in.1`, `out.1`, and `err.1` for the second run of the program, and so forth. A log file containing entries about when and where Condor runs, checkpoints, and migrates processes for the 150 queued programs will be written into file `foo.log`.

```
####################
#
# Example 3: Show off some fancy features including
# use of pre-defined macros and logging.
#
####################

Executable     = foo
Requirements   = Memory >= 32 && OpSys == "IRIX65" && Arch =="SGI"
Rank  = Memory >= 64
Image_Size     = 28 Meg

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log = foo.log

Queue 150
```

2.5.2 About Requirements and Rank

The `requirements` and `rank` commands in the submit description file are powerful and flexible. Using them effectively requires care, and this section presents those details.

Both `requirements` and `rank` need to be specified as valid Condor ClassAd expressions, however, default values are set by the *condor_submit* program if these are not defined in the submit description file. From the *condor_submit* manual page and the above examples, you see that writing ClassAd expressions is intuitive, especially if you are familiar with the programming language C. There are some pretty nifty expressions you can write with ClassAds. A complete description of ClassAds and their expressions can be found in section 4.1 on page 208.

All of the commands in the submit description file are case insensitive, *except* for the ClassAd

attribute string values. ClassAds attribute names are case insensitive, but ClassAd string values are always *case sensitive*. The correct specification for an architecture is

```
requirements = arch == "ALPHA"
```

so an accidental specification of

```
requirements = arch == "alpha"
```

will not work due to the incorrect case.

The allowed ClassAd attributes are those that appear in a machine or a job ClassAd. To see all of the machine ClassAd attributes for all machines in the Condor pool, run *condor_status -l*. The *-l* argument to *condor_status* means to display all the complete machine ClassAds. The job ClassAds, if there jobs in the queue, can be seen with the *condor_q -l* command. This will show you all the available attributes you can play with.

To help you out with what these attributes all signify, descriptions follow for the attributes which will be common to every machine ClassAd. Remember that because ClassAds are flexible, the machine ads in your pool may include additional attributes specific to your site's installation and policies.

ClassAd Machine Attributes

`Activity` : String which describes Condor job activity on the machine. Can have one of the following values:

`"Idle"` : There is no job activity

`"Busy"` : A job is busy running

`"Suspended"` : A job is currently suspended

`"Vacating"` : A job is currently checkpointing

`"Killing"` : A job is currently being killed

`"Benchmarking"` : The startd is running benchmarks

`Arch` : String with the architecture of the machine. Typically one of the following:

`"INTEL"` : Intel x86 CPU (Pentium, Xeon, etc).

`"ALPHA"` : Digital Alpha CPU

`"SGI"` : Silicon Graphics MIPS CPU

`"SUN4u"` : Sun UltraSparc CPU

`"SUN4x"` : A Sun Sparc CPU other than an UltraSparc, i.e. sun4m or sun4c CPU found in older Sparc workstations such as the Sparc 10, Sparc 20, IPC, IPX, etc.

"HPPA1" : Hewlett Packard PA-RISC 1.x CPU (i.e. PA-RISC 7000 series CPU) based workstation

"HPPA2" : Hewlett Packard PA-RISC 2.x CPU (i.e. PA-RISC 8000 series CPU) based workstation

ClockDay : The day of the week, where 0 = Sunday, 1 = Monday, . . ., 6 = Saturday.

ClockMin : The number of minutes passed since midnight.

CondorLoadAvg : The portion of the load average generated by Condor (either from remote jobs or running benchmarks).

ConsoleIdle : The number of seconds since activity on the system console keyboard or console mouse has last been detected.

Cpus : Number of CPUs in this machine, i.e. 1 = single CPU machine, 2 = dual CPUs, etc.

CurrentRank : A float which represents this machine owner's affinity for running the Condor job which it is currently hosting. If not currently hosting a Condor job, `CurrentRank` is -1.0.

Disk : The amount of disk space on this machine available for the job in kbytes (e.g. 23000 = 23 megabytes). Specifically, this is the amount of disk space available in the directory specified in the Condor configuration files by the `EXECUTE` macro, minus any space reserved with the `RESERVED_DISK` macro.

EnteredCurrentActivity : Time at which the machine entered the current Activity (see `Activity` entry above). On all platforms (including NT), this is measured in the number of seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

FileSystemDomain : A "domain" name configured by the Condor administrator which describes a cluster of machines which all access the same, uniformly-mounted, networked file systems usually via NFS or AFS. This is useful for Vanilla universe jobs which require remote file access.

KeyboardIdle : The number of seconds since activity on any keyboard or mouse associated with this machine has last been detected. Unlike `ConsoleIdle`, `KeyboardIdle` also takes activity on pseudo-terminals into account (i.e. virtual "keyboard" activity from telnet and rlogin sessions as well). Note that `KeyboardIdle` will always be equal to or less than `ConsoleIdle`.

KFlops : Relative floating point performance as determined via a Linpack benchmark.

LastHeardFrom : Time when the Condor central manager last received a status update from this machine. Expressed as seconds since the epoch (integer value). Note: This attribute is only inserted by the central manager once it receives the ClassAd. It is not present in the *condor_startd* copy of the ClassAd. Therefore, you could not use this attribute in defining *condor_startd* expressions (and you would not want to).

LoadAvg : A floating point number with the machine's current load average.

Machine : A string with the machine's fully qualified hostname.

Memory : The amount of RAM in megabytes.

Mips : Relative integer performance as determined via a Dhrystone benchmark.

MyType : The ClassAd type; always set to the literal string `"Machine"`.

Name : The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate virtual machines, each with with a unique name. These names will be of the form "vm#@full.hostname", for example, "vm1@vulture.cs.wisc.edu", which signifies virtual machine 1 from vulture.cs.wisc.edu.

OpSys : String describing the operating system running on this machine. For Condor Version 6.4.0 typically one of the following:

"HPUX10" : for HPUX 10.20

"IRIX6" : for IRIX 6.2, 6.3, or 6.4

"IRIX65" : for IRIX 6.5

"LINUX" : for LINUX 2.0.x or LINUX 2.2.x kernel systems

"OSF1" : for Digital Unix 4.x

"SOLARIS251"

"SOLARIS26"

"SOLARIS27"

"SOLARIS28"

"WINNT40" : for Windows NT 4.0

Requirements : A boolean, which when evaluated within the context of the machine ClassAd and a job ClassAd, must evaluate to TRUE before Condor will allow the job to use this machine.

StartdIpAddr : String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd.

State : String which publishes the machine's Condor state. Can be:

"Owner" : The machine owner is using the machine, and it is unavailable to Condor.

"Unclaimed" : The machine is available to run Condor jobs, but a good match is either not available or not yet found.

"Matched" : The Condor central manager has found a good match for this resource, but a Condor scheduler has not yet claimed it.

"Claimed" : The machine is claimed by a remote *condor_schedd* and is probably running a job.

"Preempting" : A Condor job is being preempted (possibly via checkpointing) in order to clear the machine for either a higher priority job or because the machine owner wants the machine back.

TargetType : Describes what type of ClassAd to match with. Always set to the string literal "Job", because machine ClassAds always want to be matched with jobs, and vice-versa.

UidDomain : a domain name configured by the Condor administrator which describes a cluster of machines which all have the same passwd file entries, and therefore all have the same logins.

VirtualMemory : The amount of currently available virtual memory (swap space) expressed in kbytes.

In addition, there are a few attributes that are automatically inserted into the machine ClassAd whenever a resource is in the Claimed state:

ClientMachine : The hostname of the machine that has claimed this resource

CurrentRank : The value of the RANK expression when evaluated against the ClassAd of the "current" job using this machine. If the resource has been claimed but no job is running, the "current" job ClassAd is the one that was used when claiming the resource. If a job is currently running, that job's ClassAd is the one that was used when claiming the resource. If a job is currently running, that job's ClassAd is the "current" one. If the resource is between jobs, the ClassAd of the last job that was run is used for CurrentRank.

RemoteOwner : The name of the user who originally claimed this resource.

RemoteUser : The name of the user who is currently using this resource. In general, this will always be the same as the RemoteOwner, but in some cases, a resource can be claimed by one entity that hands off the resource to another entity which uses it. In that case, RemoteUser would hold the name of the entity currently using the resource, while RemoteOwner would hold the name of the entity that claimed the resource.

Finally, there are a few attributes that are only inserted into the machine ClassAd if a job is currently executing. If the resource is claimed but no job are running, none of these attributes will be defined.

JobId : The job's identifier (for example,

```
152.3
```

), like you would see in *condor_q* on the submitting machine.

JobStart : The timestamp of when the job began executing.

LastPeriodicCheckpoint : If the job has performed a periodic checkpoint, this attribute will be defined and will hold the timestamp of when the last periodic checkpoint was begun. If the job has yet to perform a periodic checkpoint, or cannot checkpoint at all, the LastPeriodicCheckpoint attribute will not be defined.

ClassAd Job Attributes

CkptArch : String describing the architecture of the machine where this job last checkpointed. If the job has never checkpointed, this attribute is UNDEFINED.

CkptOpSys : String describing the operating system of the machine where this job last checkpointed. If the job has never checkpointed, this attribute is UNDEFINED.

ClusterId : Integer cluster identifier for this job. A "cluster" is a group of jobs that were submitted together. Each job has its own unique job identifier within the cluser, but shares a common cluster identifier.

CumulativeSuspensionTime : A running total of the number of seconds the job has spent in suspension for the life of the job.

ExecutableSize : Size of the executable in kbytes.

ExitBySignal : When a user job exits, this is true if the job exited via a signal, and false otherwise. Currently this is available for use only in Standard and Vanilla universe jobs.

ExitCode : When a user job exits by means other than a signal, this is the exit return code of the user job. Currently this is available for use only in Standard and Vanilla universe jobs.

ExitSignal : When a user job exits by means of an unhandled signal, this attribute takes on the numeric value of the signal. Currently this is available for use only in Standard and Vanilla universe jobs.

ImageSize : Estimate of the memory image size of the job in kbytes. The initial estimate may be specified in the job submit file. Otherwise, the initial value is equal to the size of the executable. When the job checkpoints, the `ImageSize` attribute is set to the size of the checkpoint file (since the checkpoint file contains the job's memory image).

JobPrio : Integer priority for this job, set by *condor_submit* or *condor_prio*. The default value is 0. The higher the number, the worse the priority.

JobStartDate : Time at which the job first began running. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

JobStatus : Integer which indicates the current status of the job, where 1 = Idle, 2 = Running, 3 = Removed, 4 = Completed, and 5 = Held.

JobUniverse : Integer which indicates the job universe, where 1 = Standard, 4 = PVM, 5 = Vanilla, and 7 = Scheduler.

LastCkptServer : Hostname of the last checkpoint server used by this job. When a pool is using multiple checkpoint servers, this tells the job where to find its checkpoint file.

LastCkptTime : Time at which the job last performed a successful checkpoint. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

LastSuspensionTime : Time at which the job last performed a successful suspension. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

LastVacateTime : Time at which the job was last evicted from a remote workstation. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

NumCkpts : A count of the number of checkpoints written by this job during its lifetime.

NumRestarts : A count of the number of restarts from a checkpoint attempted by this job during its lifetime.

NiceUser : Boolean value which indicates whether this is a nice-user job.

Owner : String describing the user who submitted this job.

ProcId : Integer process identifier for this job. In a cluster of many jobs, each job will have the same ClusterId but will have a unique ProcId.

TotalSuspensions : A count of the number of times this job has been suspended during its lifetime.

QDate : Time at which the job was submitted to the job queue. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

Rank Expression Examples

When considering the match between a job and a machine, rank is used to choose a match from among all machines that satisfy the job's requirements and are available to the user, after accounting for the user's priority and the machine's rank of the job. The rank expressions, simple or complex, define a numerical value that expresses preferences.

The job's `rank` expression evaluates to one of three values. It can be UNDEFINED, ERROR, or a floating point value. If `rank` evaluates to a floating point value, the best match will be the one with the largest, positive value. If no `rank` is given in the submit description file, then Condor substitutes a default value of 0.0 when considering machines to match. If the job's `rank` of a given machine evaluates to UNDEFINED or ERROR, this same value of 0.0 is used. Therefore, the machine is still considered for a match, but has no rank above any other.

A boolean expression evaluates to the numerical value of 1.0 if true, and 0.0 if false.

The following `rank` expressions provide examples to follow.

For a job that desires the machine with the most available memory:

```
Rank = memory
```

For a job that prefers to run on a friend's machine on Saturdays and Sundays:

```
Rank = ( (clockday == 0) || (clockday == 6) )
       && (machine == "friend.cs.wisc.edu")
```

For a job that prefers to run on one of three specific machines:

```
Rank = (machine == "friend1.cs.wisc.edu") ||
       (machine == "friend2.cs.wisc.edu") ||
       (machine == "friend3.cs.wisc.edu")
```

For a job that wants the machine with the best floating point performance (on Linpack benchmarks):

```
Rank = kflops
```

This particular example highlights a difficulty with rank expression evaluation as currently defined. While all machines have floating point processing ability, not all machines will have the `kflops` attribute defined. For machines where this attribute is not defined, `Rank` will evaluate to the value UNDEFINED, and Condor will use a default rank of the machine of 0.0. The `rank` attribute will only rank machines where the attribute is defined. Therefore, the machine with the highest floating point performance may not be the one given the highest rank.

So, it is wise when writing a `rank` expression to check if the expression's evaluation will lead to the expected resulting ranking of machines. This can be accomplished using the *condor_status* command with the [*-constraint*] argument. This allows the user to see a list of machines that fit a constraint. To see which machines in the pool have `kflops` defined, use

```
condor_status -constraint kflops
```

Alternatively, to see a list of machines where `kflops` is not defined, use

```
condor_status -constraint "kflops=?=undefined"
```

For a job that prefers specific machines in a specific order:

```
Rank = ((machine == "friend1.cs.wisc.edu")*3) +
       ((machine == "friend2.cs.wisc.edu")*2) +
        (machine == "friend3.cs.wisc.edu")
```

If the machine being ranked is `"friend1.cs.wisc.edu"`, then the expression

```
(machine == "friend1.cs.wisc.edu")
```

is true, and gives the value 1.0. The expressions

```
(machine == "friend2.cs.wisc.edu")
```

and

```
(machine == "friend3.cs.wisc.edu")
```

are false, and give the value 0.0. Therefore, `rank` evaluates to the value 3.0. In this way, machine `"friend1.cs.wisc.edu"` is ranked higher than machine `"friend2.cs.wisc.edu"`, machine `"friend2.cs.wisc.edu"` is ranked higher than machine `"friend3.cs.wisc.edu"`, and all three of these machines are ranked higher than others.

2.5.3 Heterogeneous Submit: Execution on Differing Architectures

If executables are available for the different platforms of machines in the Condor pool, Condor can be allowed the choice of a larger number of machines when allocating a machine for a job. Modifications to the submit description file allow this choice of platforms.

A simplified example is a cross submission. An executable is available for one platform, but the submission is done from a different platform. Given the correct executable, the `requirements` command in the submit description file specifies the target architecture. For example, an executable compiled for a Sun 4, submitted from an Intel architecture running Linux would add the `requirement`

```
requirements = Arch == "SUN4x" && OpSys == "SOLARIS251"
```

Without this `requirement`, *condor_submit* will assume that the program is to be executed on a machine with the same platform as the machine where the job is submitted.

Cross submission works for both `standard` and `vanilla` universes. The burden is on the user to both obtain and specify the correct executable for the target architecture. To list the architecture and operating systems of the machines in a pool, run *condor_status*.

2.5.4 Vanilla Universe Example for Execution on Differing Architectures

A more complex example of a heterogeneous submission occurs when a job may be executed on many different architectures to gain full use of a diverse architecture and operating system pool. If the executables are available for the different architectures, then a modification to the submit description file will allow Condor to choose an executable after an available machine is chosen.

A special-purpose MachineAd substitution macro can be used in the `executable`, `environment`, and `arguments` attributes in the submit description file. The macro has the form

```
$$(MachineAdAttribute)
```

Note that this macro is ignored in all other submit description attributes. The $$() informs Condor to substitute the requested `MachineAdAttribute` from the machine where the job will be executed.

An example of the heterogeneous job submission has executables available for three platforms: LINUX Intel, Solaris26 Intel, and Irix 6.5 SGI machines. This example uses *povray* to render images using a popular free rendering engine.

The substitution macro chooses a specific executable after a platform for running the job is chosen. These executables must therefore be named based on the machine attributes that describe a platform. The executables named

```
povray.LINUX.INTEL
povray.SOLARIS26.INTEL
povray.IRIX65.SGI
```

will work correctly for the macro

```
povray.$$(OpSys).$$(Arch)
```

The executables or links to executables with this name are placed into the initial working directory so that they may be found by Condor. A submit description file that queues three jobs for this example:

```
####################
#
# Example of heterogeneous submission
#
####################

universe     = vanilla
Executable   = povray.$$(OpSys).$$(Arch)
Log          = povray.log
Output       = povray.out.$(Process)
Error        = povray.err.$(Process)

Requirements = (Arch == "INTEL" && OpSys == "LINUX") || \
               (Arch == "INTEL" && OpSys =="SOLARIS26") || \
               (Arch == "SGI" && OpSys == "IRIX65")

Arguments    = +W1024 +H768 +Iimage1.pov
Queue

Arguments    = +W1024 +H768 +Iimage2.pov
Queue
```

```
  Arguments     = +W1024 +H768 +Iimage3.pov
  Queue
```

These jobs are submitted to the vanilla universe to assure that once a job is started on a specific platform, it will finish running on that platform. Switching platforms in the middle of job execution cannot work correctly.

There are two common errors made with the substitution macro. The first is the use of a non-existent `MachineAdAttribute`. If the specified `MachineAdAttribute` does not exist in the machine's ClassAd, then Condor will place the job in the machine state of hold until the problem is resolved.

The second common error occurs due to an incomplete job set up. For example, the submit description file given above specifies three available executables. If one is missing, Condor report back that an executable is missing when it happens to match the job with a resource that requires the missing binary.

2.5.5 Standard Universe Example for Execution on Differing Architectures

Jobs submitted to the standard universe may produce checkpoints. A checkpoint can then be used to start up and continue execution of a partially completed job. For a partially completed job, the checkpoint and the job are specific to a platform. If migrated to a different machine, correct execution requires that the platform must remain the same.

In previous versions of Condor, the author of the heterogeneous submission file would need to write extra policy expressions in the `requirements` expression to force Condor to choose the same type of platform when continuing a checkpointed job. However, since it is needed in the common case, this additional policy is now automatically added to the `requirements` expression. The additional expression is added provided the user does not use `CkptArch` in the `requirements` expression. Condor will remain backwards compatible for those users who have explicitly specified `CkptRequirements`–implying use of `CkptArch`, in their `requirements` expression.

The expression added when the attribute `CkptArch` is not specified will default to

```
  # Added by Condor
  CkptRequirements = ((CkptArch == Arch) || (CkptArch =?= UNDE-
FINED)) && \
                     ((CkptOpSys == OpSys) || (CkptOp-
Sys =?= UNDEFINED))

  Requirements = (<user specified policy>) && $(CkptRequirements)
```

The behavior of the `CkptRequirements` expressions and its addition to `requirements` is as follows. The `CkptRequirements` expression guarantees correct operation in the two possible cases for a job. In the first case, the job has not produced a checkpoint. The ClassAd attributes

CkptArch and CkptOpSys will be undefined, and therefore the meta operator (=?=) evaluates to true. In the second case, the job has produced a checkpoint. The Machine ClassAd is restricted to require further execution only on a machine of the same platform. The attributes CkptArch and CkptOpSys will be defined, ensuring that the platform chosen for further execution will be the same as the one used just before the checkpoint.

Note that this restriction of platforms also applies to platforms where the executables are binary compatible.

The complete submit description file for this example:

```
  ####################
  #
  # Example of heterogeneous submission
  #
  ####################

  universe     = standard
  Executable   = povray.$$(OpSys).$$(Arch)
  Log          = povray.log
  Output       = povray.out.$(Process)
  Error        = povray.err.$(Process)

  # Condor automatically adds the correct expressions to in-
sure that the
  # checkpointed jobs will restart on the correct platform types.
  Requirements = ( (Arch == "INTEL" && OpSys == "LINUX") || \
                 (Arch == "INTEL" && OpSys =="SOLARIS26") || \
                 (Arch == "SGI" && OpSys == "IRIX65") )

  Arguments    = +W1024 +H768 +Iimage1.pov
  Queue

  Arguments    = +W1024 +H768 +Iimage2.pov
  Queue

  Arguments    = +W1024 +H768 +Iimage3.pov
  Queue
```

2.6 Managing a Condor Job

This section provides a brief summary of what can be done once jobs are submitted. The basic mechanisms for monitoring a job are introduced, but the commands are discussed briefly. You are encouraged to look at the man pages of the commands referred to (located in Chapter 9 beginning

on page 322) for more information.

When jobs are submitted, Condor will attempt to find resources to run the jobs. A list of all those with jobs submitted may be obtained through *condor_status* with the *-submitters* option. An example of this would yield output similar to:

```
%  condor_status -submitters

Name                 Machine      Running IdleJobs HeldJobs

ballard@cs.wisc.edu  bluebird.c         0       11        0
nice-user.condor@cs. cardinal.c         6      504        0
wright@cs.wisc.edu   finch.cs.w         1        1        0
jbasney@cs.wisc.edu  perdita.cs         0        0        5

                           RunningJobs           Idle-
Jobs           HeldJobs

 ballard@cs.wisc.edu                0                 11                  0
 jbasney@cs.wisc.edu                0                  0                  5
nice-user.condor@cs.                6                504                  0
  wright@cs.wisc.edu                1                  1                  0

               Total                7                516                  5
```

2.6.1 Checking on the progress of jobs

At any time, you can check on the status of your jobs with the *condor_q* command. This command displays the status of all queued jobs. An example of the output from *condor_q* is

```
%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -
10 1.2  hello.remote
 127.0   raman           4/11 15:35   0+00:00:00 R  0   1.4  hello
 128.0   raman           4/11 15:35   0+00:02:33 I  0   1.4  hello

3 jobs; 2 idle, 1 running, 0 held
```

This output contains many columns of information about the queued jobs. The `ST` column (for status) shows the status of current jobs in the queue. An `R` in the status column means the the job is

currently running. An `I` stands for idle. The job is not running right now, because it is waiting for a machine to become available. The status `H` is the hold state. In the hold state, the job will not be scheduled to run until it is released (see condor_hold and condor_release man pages). Older versions of Condor used a `U` in the status column to stand for unexpanded. In this state, a job has never checkpointed and when it starts running, it will start running from the beginning. Newer versions of Condor do not use the `U` state.

The `CPU_USAGE` time reported for a job is the time that has been committed to the job. It is not updated for a job until the job checkpoints. At that time, the job has made guaranteed forward progress. Depending upon how the site administrator configured the pool, several hours may pass between checkpoints, so do not worry if you do not observe the `CPU_USAGE` entry changing by the hour. Also note that this is actual CPU time as reported by the operating system; it is not time as measured by a wall clock.

Another useful method of tracking the progress of jobs is through the user log. If you have specified a `log` command in your submit file, the progress of the job may be followed by viewing the log file. Various events such as execution commencement, checkpoint, eviction and termination are logged in the file. Also logged is the time at which the event occurred.

When your job begins to run, Condor starts up a *condor_shadow* process on the submit machine. The shadow process is the mechanism by which the remotely executing jobs can access the environment from which it was submitted, such as input and output files.

It is normal for a machine which has submitted hundreds of jobs to have hundreds of shadows running on the machine. Since the text segments of all these processes is the same, the load on the submit machine is usually not significant. If, however, you notice degraded performance, you can limit the number of jobs that can run simultaneously through the `MAX_JOBS_RUNNING` configuration parameter. Please talk to your system administrator for the necessary configuration change.

You can also find all the machines that are running your job through the *condor_status* command. For example, to find all the machines that are running jobs submitted by "breach@cs.wisc.edu," type:

```
%  condor_status -constraint 'RemoteUser == "breach@cs.wisc.edu"'

Name       Arch     OpSys        State      Activity   Loa-
dAv Mem  ActvtyTime

alfred.cs. INTEL    SOLARIS251   Claimed    Busy       0.980  64    0+07:10:02
biron.cs.w INTEL    SOLARIS251   Claimed    Busy       1.000  128   0+01:10:00
cambridge. INTEL    SOLARIS251   Claimed    Busy       0.988  64    0+00:15:00
falcons.cs INTEL    SOLARIS251   Claimed    Busy       0.996  32    0+02:05:03
happy.cs.w INTEL    SOLARIS251   Claimed    Busy       0.988  128   0+03:05:00
istat03.st INTEL    SOLARIS251   Claimed    Busy       0.883  64    0+06:45:01
istat04.st INTEL    SOLARIS251   Claimed    Busy       0.988  64    0+00:10:00
istat09.st INTEL    SOLARIS251   Claimed    Busy       0.301  64    0+03:45:00
...
```

To find all the machines that are running any job at all, type:

```
%  condor_status -run

Name       Arch     OpSys        LoadAv Remo-
teUser           ClientMachine

adriana.cs INTEL    SOLARIS251   0.980  hepcon@cs.wisc.edu   chevre.cs.wisc.
alfred.cs. INTEL    SOLARIS251   0.980  breach@cs.wisc.edu   neufchatel.cs.w
amul.cs.wi SUN4u    SOLARIS251   1.000  nice-
user.condor@cs. chevre.cs.wisc.
anfrom.cs. SUN4x    SOLARIS251   1.023  ashoks@jules.ncsa.ui jules.ncsa.uiuc
anthrax.cs INTEL    SOLARIS251   0.285  hepcon@cs.wisc.edu   chevre.cs.wisc.
astro.cs.w INTEL    SOLARIS251   1.000  nice-
user.condor@cs. chevre.cs.wisc.
aura.cs.wi SUN4u    SOLARIS251   0.996  nice-
user.condor@cs. chevre.cs.wisc.
balder.cs. INTEL    SOLARIS251   1.000  nice-
user.condor@cs. chevre.cs.wisc.
bamba.cs.w INTEL    SOLARIS251   1.574  dmarino@cs.wisc.edu  riola.cs.wisc.e
bardolph.c INTEL    SOLARIS251   1.000  nice-
user.condor@cs. chevre.cs.wisc.
...
```

2.6.2 Removing a job from the queue

A job can be removed from the queue at any time by using the *condor_rm* command. If the job that is being removed is currently running, the job is killed without a checkpoint, and its queue entry is removed. The following example shows the queue of jobs before and after a job is removed.

```
%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -
10 1.2  hello.remote
 132.0   raman           4/11 16:57   0+00:00:00 R  0   1.4  hello

2 jobs; 1 idle, 1 running, 0 held

%  condor_rm 132.0
Job 132.0 removed.

%  condor_q
```

```
-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -
10 1.2  hello.remote

1 jobs; 1 idle, 0 running, 0 held
```

2.6.3 Changing the priority of jobs

In addition to the priorities assigned to each user, Condor also provides each user with the capability of assigning priorities to each submitted job. These job priorities are local to each queue and range from -20 to +20, with higher values meaning better priority.

The default priority of a job is 0, but can be changed using the *condor_prio* command. For example, to change the priority of a job to -15,

```
%  condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 I  0   0.3  hello

1 jobs; 1 idle, 0 running, 0 held

%  condor_prio -p -15 126.0

%  condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 I  -15 0.3  hello

1 jobs; 1 idle, 0 running, 0 held
```

It is important to note that these *job* priorities are completely different from the *user* priorities assigned by Condor. Job priorities do not impact user priorities. They are only a mechanism for the user to identify the relative importance of jobs among all the jobs submitted by the user to that specific queue.

2.6.4 Why does the job not run?

Users sometimes find that their jobs do not run. There are several reasons why a specific job does not run. These reasons include failed job or machine constraints, bias due to preferences, insuffi-

cient priority, and the preemption throttle that is implemented by the *condor_negotiator* to prevent thrashing. Many of these reasons can be diagnosed by using the *-analyze* option of *condor_q*. For example, the following job submitted by user jbasney was found to have not run for several days.

```
% condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -
10 1.2  hello.remote

1 jobs; 1 idle, 0 running, 0 held
```

Running *condor_q*'s analyzer provided the following information:

```
%  condor_q 125.0 -analyze

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
---
125.000:  Run analysis summary.  Of 323 resource offers,
          323 do not satisfy the request's constraints
            0 resource offer constraints are not satis-
fied by this request
            0 are serving equal or higher priority customers
            0 are serving more preferred customers
            0 cannot preempt because preemption has been held
            0 are available to service your request

WARNING:  Be advised:
   No resources matched request's constraints
   Check the Requirements expression below:

Requirements = Arch == "INTEL" && OpSys == "IRIX6" &&
  Disk >= ExecutableSize && VirtualMemory >= ImageSize
```

For this job, the `Requirements` expression specifies a platform that does not exist. Therefore, the expression always evaluates to false.

While the analyzer can diagnose most common problems, there are some situations that it cannot reliably detect due to the instantaneous and local nature of the information it uses to detect the problem. Thus, it may be that the analyzer reports that resources are available to service the request, but the job still does not run. In most of these situations, the delay is transient, and the job will run during the next negotiation cycle.

If the problem persists and the analyzer is unable to detect the situation, it may be that the job begins to run but immediately terminates due to some problem. Viewing the job's error and log files

(specified in the submit command file) and Condor's SHADOW_LOG file may assist in tracking down the problem. If the cause is still unclear, please contact your system administrator.

2.6.5 Job Completion

When your Condor job completes(either through normal means or abnormal termination by signal), Condor will remove it from the job queue (i.e., it will no longer appear in the output of *condor_q*) and insert it into the job history file. You can examine the job history file with the *condor_history* command. If you specified a log file in your submit description file, then the job exit status will be recorded there as well.

By default, Condor will send you an email message when your job completes. You can modify this behavior with the *condor_submit* "notification" command. The message will include the exit status of your job (i.e., the argument your job passed to the exit system call when it completed) or notification that your job was killed by a signal. It will also include the following statistics (as appropriate) about your job:

Submitted at: when the job was submitted with *condor_submit*

Completed at: when the job completed

Real Time: elapsed time between when the job was submitted and when it completed (days hours:minutes:seconds)

Run Time: total time the job was running (i.e., real time minus queueing time)

Committed Time: total run time that contributed to job completion (i.e., run time minus the run time that was lost because the job was evicted without performing a checkpoint)

Remote User Time: total amount of committed time the job spent executing in user mode

Remote System Time: total amount of committed time the job spent executing in system mode

Total Remote Time: total committed CPU time for the job

Local User Time: total amount of time this job's *condor_shadow* (remote system call server) spent executing in user mode

Local System Time: total amount of time this job's *condor_shadow* spent executing in system mode

Total Local Time: total CPU usage for this job's *condor_shadow*

Leveraging Factor: the ratio of total remote time to total system time (a factor below 1.0 indicates that the job ran inefficiently, spending more CPU time performing remote system calls than actually executing on the remote machine)

Virtual Image Size: memory size of the job, computed when the job checkpoints

Checkpoints written: number of successful checkpoints performed by the job

Checkpoint restarts: number of times the job successfully restarted from a checkpoint

Network: total network usage by the job for checkpointing and remote system calls

Buffer Configuration: configuration of remote system call I/O buffers

Total I/O: total file I/O detected by the remote system call library

I/O by File: I/O statistics per file produced by the remote system call library

Remote System Calls: listing of all remote system calls performed (both Condor-specific and Unix system calls) with a count of the number of times each was performed

2.7 Priorities in Condor

Condor has two independent priority controls: *job* priorities and *user* priorities.

2.7.1 Job Priority

Job priorities allow the assignment of a priority level to each submitted Condor job in order to control order of execution. To set a job priority, use the *condor_prio* command — see the example in section 2.6.3, or the command reference page on page 358. Job priorities do not impact user priorities in any fashion. Job priorities range from -20 to +20, with -20 being the worst and with +20 being the best.

2.7.2 User priority

Machines are allocated to users based upon a user's priority. A lower numerical value for user priority means higher priority, so a user with priority 5 will get more resources than a user with priority 50. User priorities in Condor can be examined with the *condor_userprio* command (see page 418). Condor administrators can set and change individual user priorities with the same utility.

Condor continuously calculates the share of available machines that each user should be allocated. This share is inversely related to the ratio between user priorities. For example, a user with a priority of 10 will get twice as many machines as a user with a priority of 20. The priority of each individual user changes according to the number of resources the individual is using. Each user starts out with the best possible priority: 0.5. If the number of machines a user currently has is greater than the user priority, the user priority will worsen by numerically increasing over time. If the number of machines is less then the priority, the priority will improve by numerically decreasing over time. The long-term result is fair-share access across all users. The speed at which Condor adjusts the priorities is controlled with the configuration macro `PRIORITY_HALFLIFE` , an exponential half-life value. The default is one day. If a user that has user priority of 100 and is utilizing

100 machines removes all his/her jobs, one day later that user's priority will be 50, and two days later the priority will be 25.

Condor enforces that each user gets his/her fair share of machines according to user priority both when allocating machines which become available and by priority preemption of currently allocated machines. For instance, if a low priority user is utilizing all available machines and suddenly a higher priority user submits jobs, Condor will immediately checkpoint and vacate jobs belonging to the lower priority user. This will free up machines that Condor will then give over to the higher priority user. Condor will not starve the lower priority user; it will preempt only enough jobs so that the higher priority user's fair share can be realized (based upon the ratio between user priorities). To prevent thrashing of the system due to priority preemption, the Condor site administrator can define a `PREEMPTION_REQUIREMENTS` expression in Condor's configuration. The default expression that ships with Condor is configured to only preempt lower priority jobs that have run for at least one hour. So in the previous example, in the worse case it could take up to a maximum of one hour until the higher priority user receives his fair share of machines.

User priorities are keyed on "username@domain", for example "johndoe@cs.wisc.edu". The domain name to use, if any, is configured by the Condor site administrator. Thus, user priority and therefore resource allocation is not impacted by which machine the user submits from or even if the user submits jobs from multiple machines.

An extra feature is the ability to submit a job as a "nice" job (see page 401). Nice jobs artificially boost the user priority by one million just for the nice job. This effectively means that nice jobs will only run on machines that no other Condor job (that is, non-niced job) wants. In a similar fashion, a Condor administrator could set the user priority of any specific Condor user very high. If done, for example, with a guest account, the guest could only use cycles not wanted by other users of the system.

2.7.3 Environment Variables

Condor provides several tidbits of information to your job that allow it to make sensible decisions when running in a batch mode. These are provided in the form of environment variables that your job may read in the manner provided by your language.

- `CONDOR_SCRATCH_DIR` - This variable gives the name of a directory where your job may place temporary data files. This directory is unique for every job that is run, and is always deleted, no matter how your job completes.
- `CONDOR_VM` - This variable gives the name of the virtual machine, if any, on which your job is run.

2.8 Java Applications

Condor allows users to access a wide variety of machines distributed around the world. The Java Virtual Machine (JVM) provides a uniform platform on any machine, regardless of the machine's architecture or operating system. The Condor Java universe brings together these two features to create a distributed, homogeneous computing environment.

Compiled Java programs can be submitted to Condor, and Condor can execute the programs on any machine in the pool that will run the Java Virtual Machine.

The *condor_status* command can be used to see a list of machines in the pool for which Condor can use the Java Virtual Machine.

```
% condor_status -java

Name          JavaVendor  Ver    State      Activity   Loa-
dAv Mem   ActvtyTime

coral.cs.wisc Sun Microsy 1.2.2  Unclaimed  Idle       0.000   511  0+02:28:04
doc.cs.wisc.e Sun Microsy 1.2.2  Unclaimed  Idle       0.000   511  0+01:05:04
dsonokwa.cs.w Sun Microsy 1.2.2  Unclaimed  Idle       0.000   511  0+01:05:04
...
```

If there is no output from the *condor_status* command, then Condor does not know the location details of the Java Virtual Machine on machines in the pool, or no machines have Java correctly installed. In this case, contact your system administrator or see section 3.11 for more information on getting Condor to work together with Java.

Here is a complete, if simple, example. Start with a simple Java program, `Hello.java`:

```
public class Hello {
        public static void main( String [] args ) {
                System.out.println("Hello, world!\n");
        }
}
```

Build this program using your Java compiler. On most platforms, this is accomplished with the command

```
javac Hello.java
```

Submission to Condor requires a submit description file. This is a simple one that works:

```
  ####################
```

```
#
# Example 1
# Execute a single Java class
#
####################

universe       = java
executable     = Hello.class
arguments      = Hello
output         = Hello.output
error          = Hello.error
queue
```

The Java universe must be explicitly selected.

The main class of the program is given in the `executable` statement. This is a file name which contains the entry point of the program. The name of the main class (not a file name) must be specified as the first argument to the program.

To submit the job, where the submit description file is named `Hello.cmd`, execute

```
condor_submit Hello.cmd
```

To monitor the job, the commands *condor_q* and *condor_rm* are used as with all jobs.

For programs that consist of more than one `.class` file, one option that tells Condor of the additional files adds this line to the submit description file:

```
transfer_input_files = Larry.class Curly.class Moe.class
```

If the program consists of a large number of class files, it may be easier to collect them all together into a single Java Archive (JAR) file. A JAR can be created with:

```
% jar cvf Library.jar Larry.class Curly.class Moe.class
```

Condor must then be told where to find the JAR by adding the following to the submit description file:

```
jar_files = Library.jar
```

Note that the JVM must know whether it is receiving JAR files or class files. Therefore, Condor must also be informed, in order to pass the information on to the JVM. That is why there is a difference in submit description file commands for the two ways of specifying files (`transfer_input_files` and `jar_files`).

If the program uses Java features found only in certain JVMs, then inform Condor by adding a `requirements` statement to the submit description file. For example, to require version 3.2, add to the submit description file:

```
requirements = (JavaVersion=="3.2")
```

Each machine with Java capability in a Condor pool will execute a benchmark to determine its speed. The benchmark is taken when Condor is started on the machine, and it uses the SciMark2 (http://math.nist.gov/scimark2) benchmark. The result of the benchmark is held as an attribute within the machine ClassAd. The attribute is called `JavaMFlops`. Jobs that are run under the Java universe (as all other Condor jobs) may prefer or require a machine of a specific speed by setting `rank` or `requirements` in the submit description file. As an example, to execute only on machines of a minimum speed:

```
requirements = (JavaMFlops>4.5)
```

By default, Condor moves all input files to each execution site before the job runs, and it moves all output files back when the job completes.

If a job has more sophisticated I/O requirements, then you may use a facility called Chirp. Chirp has two advantages over simple whole-file transfers. First, it permits the input files to be decided upon at run-time rather than submit time, and second, it permits partial-file I/O with results than can be seen as the program executes. However, you must make small changes to the program in order to take advantage of Chirp. Depending on the style of the program, use either Chirp I/O streams or UNIX-like I/O functions.

Chirp I/O streams are the easiest way to get started. Modify the program to use the objects `ChirpInputStream` and `ChirpOutputStream` instead of `FileInputStream` and `FileOutputStream`. These classes are completely documented in the Condor Software Developer's Kit (SDK). Here is a simple code example:

```
import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

   public static void main( String args[] ) {

      try {
         BufferedReader in = new BufferedReader(
            new InputStreamReader(
               new ChirpInputStream("input")));

         PrintWriter out = new PrintWriter(
            new OutputStreamWriter(
```

```
                new ChirpOutputStream("output")));

            while(true) {
               String line = in.readLine();
               if(line==null) break;
               out.println(line);
            }
         } catch( IOException e ) {
            System.out.println(e);
         }
      }
}
```

To perform UNIX-like I/O with Chirp, create a `ChirpClient` object. This object supports familiar operations such as `open`, `read`, `write`, and `close`. Exhaustive detail of the methods may be found in the Condor SDK, but here is a brief example:

```
import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

   public static void main( String args[] ) {

      try {
         ChirpClient client = new ChirpClient();
         String message = "Hello, world!\n";
         byte [] buffer = message.getBytes();

         // Note that we should check that actual==length.
         // However, skip it for clarity.

         int fd = client.open("output","wct",0777);
         int actual = client.write(fd,buffer,0,buffer.length);
         client.close(fd);

         client.rename("output","output.new");
         client.unlink("output.new");

      } catch( IOException e ) {
         System.out.println(e);
      }
   }
}
```

Regardless of which I/O style, the Chirp library must be specified and included with the job. The Chirp JAR (`Chirp.jar`) is found in the `lib` directory of the Condor installation. Copy it into your working directory in order to compile the program after modification to use Chirp I/O.

```
% condor_config_val LIB
/usr/local/condor/lib
% cp /usr/local/condor/lib/Chirp.jar .
```

Rebuild the program with the Chirp JAR file in the class path.

```
% javac -classpath Chirp.jar:. TestChirp.java
```

The Chirp JAR file must be specified in the submit description file. Here is an example submit description file that works for both of the given test programs:

```
universe = java
executable = TestChirp.class
arguments = TestChirp
jar_files = Chirp.jar
queue
```

2.9 Parallel Applications in Condor: Condor-PVM

Applications that use PVM (Parallel Virtual Machine) may use Condor. PVM offers a set of message passing primitives for use in C and C++ language programs. The primitives, together with the PVM environment allow parallelism at the program level. Multiple processes may run on multiple machines, while communicating with each other. More information about PVM is available at http://www.epm.ornl.gov/pvm/.

Condor-PVM provides a framework to run PVM applications in Condor's opportunistic environment. Where PVM needs dedicated machines to run PVM applications, Condor does not. Condor can be used to dynamically construct PVM virtual machines from a Condor pool of machines.

In Condor-PVM, Condor acts as the resource manager for the PVM daemon. Whenever a PVM program asks for nodes (machines), the request is forwarded to Condor. Condor finds a machine in the Condor pool using usual mechanisms, and adds it to the virtual machine. If a machine needs to leave the pool, the PVM program is notified by normal PVM mechanisms.

NOTE: Condor-PVM is an optional Condor module. It is not automatically installed with Condor. To check and see if it has been installed at your site, enter the command:

```
ls -l `condor_config_val PVMD`
```

Please note the use of back ticks in the above command. They specify to run the *condor_config_val* program. If the result of this program shows the file `condor_pvmd` on your system, then the Condor-PVM module is installed. If not, ask your site administrator to download and install Condor-PVM from http://www.cs.wisc.edu/condor/downloads/.

2.9.1 Effective Usage: the Master-Worker Paradigm

There are several different parallel programming paradigms. One of the more common is the *master-worker* (or *pool of tasks*) arrangement. In a master-worker program model, one node acts as the controlling master for the parallel application and sends pieces of work out to worker nodes. The worker node does some computation, and it sends the result back to the master node. The master has a pool of work that needs to be done, so it assigns the next piece of work out to the next worker that becomes available.

Condor-PVM is designed to run PVM applications which follow the master-worker paradigm. Condor runs the master application on the machine where the job was submitted and will not preempt it. Workers are pulled in from the Condor pool as they become available.

Not all parallel programming paradigms lend themselves to Condor's opportunistic environment. In such an environment, any of the nodes could be preempted and disappear at any moment. The master-worker model does work well in this environment. The master keeps track of which piece of work it sends to each worker. The master node is informed of the addition and disappearance of nodes. If the master node is informed that a worker node has disappeared, the master places the unfinished work it had assigned to the disappearing node back into the pool of tasks. This work is sent again to the next available worker node. If the master notices that the number of workers has dropped below an acceptable level, it requests more workers (using `pvm_addhosts()`). Alternatively, the master requests a replacement node every time it is notified that a worker has gone away. The benefit of this paradigm is that the number of workers is not important and changes in the size of the virtual machine are easily handled.

A tool called *MW* has been developed to assist in the development of master-worker style applications for distributed, opportunistic environments like Condor. MW provides a C++ API which hides the complexities of managing a master-worker Condor-PVM application. We suggest that you consider modifying your PVM application to use MW instead of developing your own dynamic PVM master from scratch. Additional information about MW is available at http://www.cs.wisc.edu/condor/mw/.

2.9.2 Binary Compatibility and Runtime Differences

Condor-PVM does not define a new API (application program interface); programs use the existing resource management PVM calls such as `pvm_addhosts()` and `pvm_notify()`. Because of this, some master-worker PVM applications are ready to run under Condor-PVM with no changes at all. Regardless of using Condor-PVM or not, it is good master-worker design to handle the case of a disappearing worker node, and therefore many programmers have already constructed their master

program with all the necessary fault tolerant logic.

Regular PVM and Condor-PVM are *binary compatible*. The same binary which runs under regular PVM will run under Condor, and vice-versa. There is no need to re-link for Condor-PVM. This permits easy application development (develop your PVM application interactively with the regular PVM console, XPVM, etc.) as well as binary sharing between Condor and some dedicated MPP systems.

This release of Condor-PVM is based on PVM 3.4.2. PVM versions 3.4.0 through 3.4.2 are all supported. The vast majority of the PVM library functions under Condor maintain the same semantics as in PVM 3.4.2, including messaging operations, group operations, and `pvm_catchout()`.

The following list is a summary of the changes and new features of PVM running within the Condor environment:

- Condor introduces the concept of machine class. A pool of machines is likely to contain machines of more than one platform. Under Condor-PVM, machines of different architectures belong to different machine classes. With the concept machine class, Condor can be told what type of machine to allocate. Machine classes are assigned integer values, starting with 0. A machine class is specified in a submit description file when the job is submitted to Condor.

- `pvm_addhosts()`. When an application adds a host machine, it calls `pvm_addhosts()`. The first argument to `pvm_addhosts()` is a string that specifies the machine class. For example, to specify class 0, a pointer to the string "0" is the first argument. Condor finds a machine that satisfies the requirements of class 0 and adds it to the PVM virtual machine.

  The function `pvm_addhosts()` does not block. It returns immediately, before hosts are added to the virtual machine. In a non-dedicated environment the amount of time it takes until a machine becomes available is not bound. A program should call `pvm_notify()` before calling `pvm_addhosts()`. When a host is added later, the program will be notified in the usual PVM fashion (with a `PvmHostAdd` notification message).

  After receiving a `PvmHostAdd` notification, the PVM master can unpack the following information about the added host: an integer specifying the TID of the new host, a string specifying the name of the new host, followed by a string specifying the machine class of the new host. The PVM master can then call `pvm_spawn()` to start a worker process on the new host, specifying this machine class as the architecture and using the appropriate executable path for this machine class. Note that the name of the host is given by the startd and may be of the form "vmN@hostname" on SMP machines.

- `pvm_notify()`. Under Condor, there are two additional possible notification types to the function `pvm_notify()`. They are `PvmHostSuspend` and `PvmHostResume`. The program calls `pvm_notify()` with a host tid and `PvmHostSuspend` (or `PvmHostResume`) as arguments, and the program will receive a notification for the event of a host being suspended. Note that a notification occurs only once for each request. As an example, a `PvmHostSuspend` notification request for tid 4 results in a single `PvmHostSuspend` message for tid 4. There will not be another `PvmHostSuspend` message for that tid without another notification request.

The easiest way to handle this is the following: When a worker node starts up, set up a notification for `PvmHostSuspend` on its tid. When that node gets suspended, set up a `PvmHostResume` notification. When it resumes, set up a `PvmHostSuspend` notification.

If your application uses the `PvmHostSuspend` and `PvmHostResume` notification types, you will need to modify your PVM distribution to support them as follows. First, go to your $(PVM_ROOT). In `include/pvm3.h`, add

```
#define PvmHostSuspend  6   /* condor suspension */
#define PvmHostResume   7   /* condor resumption */
```

to the list of "pvm_notify kinds". In `src/lpvmgen.c`, in `pvm_notify()`, change

```
} else {
        switch (what) {
        case PvmHostDelete:
        ....
```

to

```
} else {
        switch (what) {
        case PvmHostSuspend:  /* for condor */
        case PvmHostResume:   /* for condor */
        case PvmHostDelete:
        ....
```

And that's it. Re-compile, and you're done.

- `pvm_spawn()`. If the flag in `pvm_spawn()` is `PvmTaskArch`, then a machine class string should be used. If there is only one machine class in a virtual machine, "0" is the string for the desired architecture.

  Under Condor, only one PVM task spawned per node is currently allowed, due to Condor's machine load checks. Most Condor sites will suspend or vacate a job if the load on its machine is higher than a specified threshold. Having more than one PVM task per node pushes the load higher than the threshold.

  Also, Condor only supports starting one copy of the executable with each call to `pvm_spawn()` (i.e., the fifth argument must always be equal to one). To spawn multiple copies of an executable in Condor, you must call `pvm_spawn()` once for each copy.

  A good fault tolerant program will be able to deal with `pvm_spawn()` failing. It happens more often in opportunistic environments like Condor than in dedicated ones.

- `pvm_exit()`. If a PVM task calls `pvm_catchout()` during its run to catch the output of child tasks, `pvm_exit()` will attempt to gather the output of all child tasks before returning. Due to the dynamic nature of the virtual machine in Condor, this cleanup procedure (in the PVM library and daemon) is error-prone and should be avoided. So, any PVM tasks which call `pvm_catchout()` should be sure to call it again with a NULL argument to disable output collection before calling `pvm_exit()`.

2.9.3 Sample PVM submit file

PVM jobs are submitted to the PVM universe. The following is an example of a submit description file for a PVM job. This job has a master PVM program called `master.exe`.

```
############################################################
# sample_submit
# Sample submit file for PVM jobs.
############################################################

# The job is a PVM universe job.
universe = PVM

# The executable of the master PVM program is ``master.exe''.
executable = master.exe

input = "in.dat"
output = "out.dat"
error = "err.dat"

###################  Machine class 0  ##################

Requirements = (Arch == "INTEL") && (OpSys == "LINUX")

# We want at least 2 machines in class 0 before starting the
# program.  We can use up to 4 machines.
machine_count = 2..4
queue

###################  Machine class 1  ##################

Requirements = (Arch == "SUN4x") && (OpSys == "SOLARIS26")

# We need at least 1 machine in class 1 before starting the
# executable.  We can use up to 3 to start with.
machine_count = 1..3
queue

###################  Machine class 2  ##################

Requirements = (Arch == "INTEL") && (OpSys == "SOLARIS26")

# We don't need any machines in this class at startup, but we can use
# up to 3.
machine_count = 0..3
```

```
queue

##################################################################
# note: the program will not be started until the least
#       requirements in all classes are satisfied.
##################################################################
```

In this sample submit file, the command `universe = PVM` specifies that the jobs should be submitted into PVM universe.

The command `executable = master.exe` tells Condor that the PVM master program is *master.exe*. This program will be started on the submitting machine. The workers should be spawned by this master program during execution.

The `input`, `output`, and `error` commands specify files that should be redirected to the standard in, out, and error of the PVM master program. Note that these files will not include output from worker processes unless the master calls `pvm_catchout()`.

This submit file also tells Condor that the virtual machine consists of three different classes of machine. Class 0 contains machines with INTEL processors running LINUX; class 1 contains machines with SUN4x (SPARC) processors running SOLARIS26; class 2 contains machines with INTEL processors running SOLARIS26.

By using `machine_count = <min>..<max>`, the submit file tells Condor that before the PVM master is started, there should be at least `<min>` number of machines of the current class. It also asks Condor to give it as many as `<max>` machines. During the execution of the program, the application may request more machines of each of the class by calling `pvm_addhosts()` with a string specifying the machine class. It is often useful to specify `<min>` of 0 for each class, so the PVM master will be started immediately when the first host from any machine class is allocated.

The `queue` command should be inserted after the specifications of each class.

2.10 MPI Support

MPI stands for Message Passing Interface. It provides an environment under which parallel programs may synchronize, by providing communication support. Running the MPI-based parallel programs within Condor eases the programmer's effort. Condor dedicates machines for running the programs, and it does so using the same interface used when submitting non-MPI jobs.

Condor currently supports MPICH version ch_p4, the implementation offered by Argonne National Labs. This version is available without charge by download. See the web page at http://www-unix.mcs.anl.gov/mpi/mpich/ for details and availability. Programs to be submitted for execution under Condor will have been compiled using *mpicc*. No further compilation or linking is necessary to run jobs under Condor.

2.10.1 MPI Details of Set Up

Administratively, Condor must be configured such that resources (machines) running MPI jobs are dedicated. Dedicated machines are ones that, once they begin execution of a program, will continue executing the program until the program ends. The program will not be preempted (to run another program) or suspended. Since Condor is not ordinarily used in this manner (Condor uses opportunistic scheduling), machines that are to be used as dedicated resources must be configured as such. Section 3.10.11 of Administrator's Manual describes the necessary configuration and provides detailed examples.

To simplify the dedicated scheduling of resources, a single machine becomes the scheduler of dedicated resources. This leads to a further restriction that jobs submitted to execute under the MPI universe (with dedicated machines) must be submitted from the machine running as the dedicated scheduler.

2.10.2 MPI Job Submission

Once the programs are written and compiled, and Condor resources are correctly configured, jobs may be submitted. Each Condor job requires a submit description file. The simplest submit description file for an MPI job:

```
############################################
##   submit description file for mpi_program
############################################
universe = MPI
executable = mpi_program
machine_count = 4
queue
```

This job specifies the `universe` as `mpi`, letting Condor know that dedicated resources will be required. The `machine_count` command identifies the number of machines required by the job. The four machines that run the program will default to be of the same architecture and operating system as the machine on which the job is submitted, since a platform is not specified as a requirement.

The simplest example does not specify an input or output, meaning that the computation completed is useless, since both input comes from and the output goes to `/dev/null`. A more complex example of a submit description file utilizes other features.

```
#####################################
## MPI example submit description file
#####################################
universe = MPI
executable = simplempi
```

```
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
machine_count = 4
queue
```

The specification of the input, output, and error files utilize a predefined macro that is only relevant to mpi universe jobs. See the *condor_submit* manual page on page 395 for further description of predefined macros. The `$(NODE)` macro is given a unique value as programs are assigned to machines. This value is what the MPICH version ch_p4 implementation terms the rank of a program. Note that this term is unrelated and independent of the Condor term rank. The `$(NODE)` value is fixed for the entire length of the job. It can therefore be used to identify individual aspects of the computation. In this example, it is used to give unique names to input and output files.

If your site does NOT have a shared filesystem across all the nodes where your MPI computation will execute, you can use Condor's file transfer mechanism. You can find out more details about these settings by reading the *condor_submit* man page. Assuming your job only reads input from STDIN, here is an example submit file for a site without a shared filesystem:

```
######################################
## MPI example submit description file
## without using a shared filesystem
######################################
universe = MPI
executable = simplempi
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
machine_count = 4
transfer_files = on_exit
queue
```

Consider the following C program that uses this example submit description file.

```
/**************
 * simplempi.c
 **************/
#include <stdio.h>
#include "mpi.h"

int main(argc,argv)
    int argc;
    char *argv[];
```

```
{
    int myid;
    char line[128];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    fprintf ( stdout, "Printing to stdout...%d\n", myid );
    fprintf ( stderr, "Printing to stderr...%d\n", myid );
    fgets ( line, 128, stdin );
    fprintf ( stdout, "From stdin: %s", line );

    MPI_Finalize();
    return 0;
}
```

Here is a makefile that works with the example. It would build the MPI executable, using the MPICH version ch_p4 implementation.

```
#####################################################################
## This is a very basic Makefile                                   ##
#####################################################################

# the location of the MPICH compiler
CC          = /usr/local/bin/mpicc
CLINKER     = $(CC)

CFLAGS    = -g
EXECS     = simplempi

all: $(EXECS)

simplempi: simplempi.o
        $(CLINKER) -o simplempi simplempi.o -lm

.c.o:
        $(CC) $(CFLAGS) -c $*.c
```

The submission to Condor requires exactly four machines, and queues four programs. Each of these programs requires an input file (correctly named) and produces an output file.

If input file for `$(NODE)` = 0 (called `infile.0`) contains

```
Hello number zero.
```

and the input file for `$(NODE)` = 1 (called `infile.1`) contains

```
Hello number one.
```

then after the job is submitted to Condor, there will be eight files created: `errfile.[0-3]` and `outfile.[0-3]`. `outfile.0` will contain

```
Printing to stdout...0
From stdin: Hello number zero.
```

and `errfile.0` will contain

```
Printing to stderr...0
```

2.11 Extending your Condor pool with Glidein

Condor works together with Globus software to provide the capability of submitting Condor jobs to remote computer systems. Globus software provides mechanisms to access and utilize remote resources.

condor_glidein is a program that can be used to add Globus resources to a Condor pool on a temporary basis. During this period, these resources are visible to users of the pool, but only the user that added the resources is allowed to use them. The machine in the Condor pool is referred to herein as the local node, while the resource added to the local Condor pool is referred to as the remote node.

These requirements are general to using any Globus resource:

1. An X.509 certificate issued by a Globus certificate authority.
2. Access to a Globus resource. You must be a valid Globus user and be mapped to a valid login account by the site's Globus administrator on every Globus resource that will be added to the local Condor pool using *condor_glidein*. More information can be found at http://www.globus.org
3. The environment variables `HOME` and either `GLOBUS_INSTALL_PATH` or `GLOBUS_DEPLOY_PATH` must be set.

2.11.1 *condor_glidein* Requirements

In order to use *condor_glidein* to add a Globus resource to the local Condor pool, there are several requirements beyond the general Globus requirements given above.

1. Use Globus v1.1 or better.

2. Have *gsincftp* installed. This program is an ftp client modified to use Globus X.509 authentication. More information ca be found at http://www.globus.org/datagrid/deliverables/gsiftp-tools.html.
3. Be an authorized user of the local Condor pool.
4. The local Condor pool configuration file(s) must give `HOSTALLOW_WRITE` permission to every resource that will be added using *condor_glidein*. Wildcards are permitted in this specification. An example is of adding every machine at cs.wisc.edu by adding *.cs.wisc.edu to the `HOSTALLOW_WRITE` list. Recall that the changes take effect when all machines in the local pool are sent a reconfigure command.
5. The local Condor pool's configuration file(s) must set `GLOBUSRUN` to be the path of *globusrun* and `SHADOW_GLOBUS` to be the path of the *condor_shadow.globus*.
6. Included in the `PATH` must be the common user programs directory `/bin`, globus tools, and the Condor user program directory.

2.11.2 What *condor_glidein* Does

condor_glidein first checks that there is a valid proxy and that the necessary files are available to *condor_glidein*.

condor_glidein then contacts the Globus resource and checks for the presence of the necessary configuration files and Condor executables. If the executables are not present for the machine architecture, operating system version, and Condor version required, a server running at UW is contacted to transfer the needed executables. To gain access to the server, send email to condor-admin@cs.wisc.edu that includes the name of your X.509 certificate.

When the files are correctly in place, Condor daemons are started. *condor_glidein* does this by creating a submit description file for *condor_submit*, which runs the *condor_master* under the Globus universe. This implies that execution of the *condor_master* is started on the Globus resource. The Condor daemons exit gracefully when no jobs run on the daemons for a configurable period of time. The default length of time is 20 minutes.

The Condor executables on the Globus resource contact the local pool and attempt to join the pool. The `START` expression for the *condor_startd* daemon requires that the username of the person running *condor_glidein* matches the username of the jobs submitted through Condor.

After a short length of time, the Globus resource can be seen in the local Condor pool, as with this example.

```
% condor_status | grep denal
7591386@denal IRIX65      SGI    Unclaimed  Idle       3.700  24064  0+00:06:35
```

Once the Globus resource has been added to the local Condor pool with *condor_glidein*, job(s) may be submitted. To force a job to run on the Globus resource, specify that Globus resource as

a machine requirement in the submit description file. Here is an example from within the submit description file that forces submission to the Globus resource denali.mcs.anl.gov:

```
requirements = ( machine == "denali.mcs.anl.gov" ) \
   && FileSystemDomain != "" \
   && Arch != "" && OpSys != ""
```

This example requires that the job run only on denali.mcs.anl.gov, and it prevents Condor from inserting the filesystem domain, architecture, and operating system attributes as requirements in the matchmaking process. Condor must be told not to use the submission machine's attributes in those cases where the Globus resource's attributes do not match the submission machine's attributes.

2.12 Inter-job Dependencies: DAGMan Meta-Scheduler

A directed acyclic graph (DAG) can be used to represent a set of programs where the input, output, or execution of one or more programs is dependent on one or more other programs. The programs are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies. Condor finds machines for the execution of programs, but it does not schedule programs (jobs) based on dependencies. The Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for Condor jobs. DAGMan submits jobs to Condor in an order represented by a DAG and processes the results. An input file defined prior to submission describes the DAG, and a Condor submit description file for each program in the DAG is used by Condor.

Each node (program) in the DAG needs its own Condor submit description file. As DAGMan submits jobs to Condor, it uses a single Condor log file to enforce the ordering required for the DAG. The DAG itself is defined by the contents of a DAGMan input file. DAGMan is responsible for scheduling, recovery, and reporting for the set of programs submitted to Condor.

The following sections specify the use of DAGMan.

2.12.1 Input File describing the DAG

The input file used by DAGMan specifies four items:

1. A list of the programs in the DAG. This serves to name each program and specify each program's Condor submit description file.
2. Processing that takes place before submission of any program in the DAG to Condor or after Condor has completed execution of any program in the DAG.
3. Description of the dependencies in the DAG.
4. Number of times to retry if a node within the DAG fails.

Comments may be placed in the input file that describes the DAG. The pound character (#) as the first character on a line identifies the line as a comment. Comments do not span lines.

An example input file for DAGMan is

```
# Filename: diamond.dag
#
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
Script PRE  A top_pre.csh
Script PRE  B mid_pre.perl  $JOB
Script POST B mid_post.perl $JOB $RETURN
Script PRE  C mid_pre.perl  $JOB
Script POST C mid_post.perl $JOB $RETURN
Script PRE  D bot_pre.csh
PARENT A CHILD B C
PARENT B C CHILD D
Retry  C 3
```

This input file describes the DAG shown in Figure 2.2.

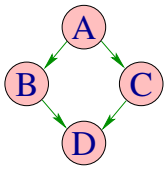


Figure 2.2: Diamond DAG

The first section of the input file lists all the programs that appear in the DAG. Each program is described by a single line called a Job Entry. The syntax used for each Job Entry is

JOB JobName CondorSubmitDescriptionFile [*DONE*]

A Job Entry maps a *JobName* to a Condor submit description file. The *JobName* uniquely identifies nodes within the DAGMan input file and within output messages.

The keyword *JOB* and the *JobName* are not case sensitive. A *JobName* of *joba* is equivalent to *JobA*. The *CondorSubmitDescriptionFile* is case sensitive, since the UNIX file system is case sensitive. The *JobName* can be any string that contains no white space.

The optional *DONE* identifies a job as being already completed. This is useful in situations where the user wishes to verify results, but does not need all programs within the dependency graph to be executed. The *DONE* feature is also utilized when an error occurs causing the DAG to not be

completed. DAGMan generates a Rescue DAG, a DAGMan input file that can be used to restart and complete a DAG without re-executing completed programs.

The second type of item in a DAGMan input file enumerates processing that is done either before a program within the DAG is submitted to Condor for execution or after a program within the DAG completes its execution. Processing done before a program is submitted to Condor is called a *PRE* script. Processing done after a program successfully completes its execution under Condor is called a *POST* script. A node in the DAG is comprised of the program together with *PRE* and/or *POST* scripts. The dependencies in the DAG are enforced based on nodes.

Syntax for *PRE* and *POST* script lines within the input file

SCRIPT PRE JobName ExecutableName [*arguments*]

SCRIPT POST JobNameExecutableName [*arguments*]

The *SCRIPT* keyword identifies the type of line within the DAG input file. The *PRE* or *POST* keyword specifies the relative timing of when the script is to be run. The *JobName* specifies the node to which the script is attached. The *ExecutableName* specifies the script to be executed, and it may be followed by any command line arguments to that script. The *ExecutableName* and optional *arguments* have their case preserved.

Scripts are optional for each job, and any scripts are executed on the machine to which the DAG is submitted.

The PRE and POST scripts are commonly used when files must be placed into a staging area for the job to use, and files are cleaned up or removed once the job is finished running. An example using PRE/POST scripts involves staging files that are stored on tape. The PRE script reads compressed input files from the tape drive, and it uncompresses them, placing the input files in the current directory. The program within the DAG node is submitted to Condor, and it reads these input files. The program produces output files. The POST script compresses the output files, writes them out to the tape, and then deletes the staged input and output files.

DAGMan takes note of the exit value of the scripts as well as the program. If the PRE script fails (exit value != 0), then neither the program nor the POST script runs, and the node is marked as failed.

If the PRE script succeeds, the program is submitted to Condor. If the program fails and there is no POST script, the DAG node is marked as failed. An exit value not equal to 0 indicates program failure. It is therefore important that the program returns the exit value 0 to indicate the program did not fail.

If the program fails and there is a POST script, node failure is determined by the exit value of the POST script. A failing value from the POST script marks the node as failed. A succeeding value from the POST script (even with a failed program) marks the node as successful. Therefore, the POST script may need to consider the return value from the program.

By default, the POST script is run regardless of the program's return value. To prevent POST scripts from running after failed jobs, pass the *-NoPostFail* argument to *condor_submit_dag*.

A node not marked as failed at any point is successful.

Two variables are available to ease script writing. The `$JOB` variable evaluates to *JobName*. For POST scripts, the `$RETURN` variable evaluates to the return value of the program. The variables may be placed anywhere within the arguments.

As an example, suppose the *PRE* script expands a compressed file named `JobName.gz`. The *SCRIPT* entry for jobs A, B, and C are

```
SCRIPT PRE  A  pre.csh $JOB .gz
SCRIPT PRE  B  pre.csh $JOB .gz
SCRIPT PRE  C  pre.csh $JOB .gz
```

The script `pre.csh` may use these arguments

```
#!/bin/csh
gunzip $argv[1]$argv[2]
```

The third type of item in the DAG input file describes the dependencies within the DAG. Nodes are parents and/or children within the DAG. A parent node must be completed successfully before any child node may be started. A child node is started once all its parents have successfully completed.

The syntax of a dependency line within the DAG input file:

PARENT ParentJobName... CHILD ChildJobName...

The *PARENT* keyword is followed by one or more *ParentJobName*s. The *CHILD* keyword is followed by one or more *ChildJobName*s. Each child job depends on every parent job on the line. A single line in the input file can specify the dependencies from one or more parents to one or more children. As an example, the line

```
PARENT p1 p2 CHILD c1 c2
```

produces four dependencies:

1. `p1` to `c1`
2. `p1` to `c2`
3. `p2` to `c1`
4. `p2` to `c2`

The fourth type of item in the DAG input file provides a way (optional) to retry failed nodes. The syntax for retry is

Retry JobName NumberOfRetries

where the *JobName* is the same as the name given in a Job Entry line, and *NumberOfRetries* is an integer, the number of times to retry the node after failure. The default number of retries for any node is 0, the same as not having a retry line in the file.

In the event of retry, all parts of a node within the DAG are redone, following the same rules regarding node failure as given above. The PRE script is executed first, followed by submitting the program to Condor upon success of the PRE script. Failure of the node is then determined by the return value of the program, the existence and return value of a POST script.

2.12.2 Condor Submit Description File

Each node in a DAG may be a unique executable, each with a unique Condor submit description file. Each program may be submitted to a different universe within Condor, for example standard, vanilla, or DAGMan.

Two limitations exist. First, each Condor submit description file must submit only one job. There may not be multiple `queue` lines, or DAGMan will fail. The second limitation is that the submit description file for all jobs within the DAG must specify the same log. DAGMan enforces the dependencies within a DAG using the events recorded in the log file produced by job submission to Condor.

Here is an example Condor submit description file to go with the diamond-shaped DAG example.

```
# Filename: diamond_job.condor
#
executable   = /path/diamond.exe
output       = diamond.out.$(cluster)
error        = diamond.err.$(cluster)
log          = diamond_condor.log
universe     = vanilla
notification = NEVER
queue
```

This example uses the same Condor submit description file for all the jobs in the DAG. This implies that each node within the DAG runs the same program. The `$(cluster)` macro is used to produce unique file names for each program's output. Each job is submitted separately, into its own cluster, so this provides unique names for the output files.

The notification is set to `NEVER` in this example. This tells Condor not to send e-mail about the completion of a program submitted to Condor. For DAGs with many nodes, this is recommended to reduce or eliminate excessive numbers of e-mails.

2.12.3 Job Submission

A DAG is submitted using the program *condor_submit_dag*. See the manual page 395 for complete details. A simple submission has the syntax

condor_submit_dag DAGInputFileName

The example may be submitted with

```
condor_submit_dag diamond.dag
```

In order to guarantee recoverability, the DAGMan program itself is run as a Condor job. As such, it needs a submit description file. *condor_submit_dag* produces the needed file, naming it by appending the *DAGInputFileName* with `.condor.sub`. This submit description file may be editted if the DAG is submitted with

```
condor_submit_dag -no_submit diamond.dag
```

causing *condor_submit_dag* to generate the submit description file, but not submit DAGMan to Condor. To submit the DAG, once the submit description file is editted, use

```
condor_submit diamond.dag.condor.sub
```

An optional argument to *condor_submit_dag*, *-maxjobs*, is used to specify the maximum number of Condor jobs that DAGMan may submit to Condor at one time. It is commonly used when there is a limited amount of input file staging capacity. As a specific example, consider a case where each job will require 4 Mbytes of input files, and the jobs will run in a directory with a volume of 100 Mbytes of free space. Using the argument *-maxjobs 25* guarantees that a maximum of 25 jobs, using a maximum of 100 Mbytes of space, will be submitted to Condor at one time.

While the *-maxjobs* argument is used to limit the number of Condor jobs submitted at one time, it may be desirable to limit the number of scripts running at one time. The optional *-maxpre* argument limits the number of PRE scripts that may be running at one time, while the optional *-maxpost* argument limits the number of POST scripts that may be running at one time.

2.12.4 Job Monitoring

After submission, the progress of the DAG can be monitored by looking at the common log file, observing the e-mail that program submission to Condor causes, or by using *condor_q -dag*.

2.12.5 Job Failure and Job Removal

condor_submit_dag attempts to check the DAG input file to verify that all the nodes in the DAG specify the same log file. If a problem is detected, *condor_submit_dag* prints out an error message

and aborts.

To omit the check that all nodes use the same log file, as may be desired in the case where there are thousands of nodes, submit the job with the *-log* option. An example of this submission:

```
condor_submit_dag -log diamond_condor.log
```

This option tells *condor_submit_dag* to omit the verification step and use the given file as the log file.

To remove an entire DAG, consisting of DAGMan plus any jobs submitted to Condor, remove the DAGMan job running under Condor. *condor_q* will list the job number. Use the job number to remove the job, for example

```
% condor_q
-- Submitter: turunmaa.cs.wisc.edu : <128.105.175.125:36165> : turunmaa.cs.wisc.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
    9.0   smoler          10/12 11:47   0+00:01:32 R  0   8.7  con-
dor_dagman -f -
    11.0   smoler          10/12 11:48   0+00:00:00 I  0   3.6  B.out
    12.0   smoler          10/12 11:48   0+00:00:00 I  0   3.6  C.out

          3 jobs; 2 idle, 1 running, 0 held

% condor_rm 9.0
```

Before the DAGMan job stops running, it uses *condor_rm* to remove any Condor jobs within the DAG that are running.

In the case where a machine is scheduled to go down, DAGMan will clean up memory and exit. However, it will leave any submitted jobs in Condor's queue.

2.12.6 Job Recovery: The Rescue DAG

DAGMan can help with the resubmission of uncompleted portions of a DAG when one or more nodes resulted in failure. If any node in the DAG fails, the remainder of the DAG is continued until no more forward progress can be made based on the DAG's dependencies. At this point, DAGMan produces a file called a Rescue DAG.

The Rescue DAG is a DAG input file, functionally the same as the original DAG file. It additionally contains indication of successfully completed nodes using the *DONE* option in the input description file. If the DAG is resubmitted using this Rescue DAG input file, the nodes marked as completed will not be reexecuted.

The Rescue DAG is automatically generated by DAGMan when a node within the DAG fails. The file is named using the *DAGInputFileName*, and appending the suffix `.rescue` to it. Statistics

about the failed DAG execution are presented as comments at the beginning of the Rescue DAG input file.

If the Rescue DAG file is generated before all retries of a node are completed, then the Rescue DAG file will also contain Retry entries. The number of retries will be set to the appropriate remaining number of retries.

2.13 UserLogViewer

The Condor User Log Viewer is a Java application designed to allow users to view user log files created by the Condor project at the University of Wisconsin.

To view a user log file, select it using the open file command in the File menu. After the file is parsed, it will be visually represented. Each horizontal line represents an individual job. The x-axis is time. Whether a job is running at a particular time is represented by its color at that time – white for running, black for idle. For example, a job which appears predominantly white has made efficient progress, whereas a job which appears predominantly black has received an inordinately small proportion of computational time.

2.13.1 Transition States

A transistion state is the state of a job at any time. It is called a ”transistion” because it is defined by the two events which bookmark it. There are two basic transistion states: running and idle. An idle job typically is a job which has just been submitted into the Condor pool and is waiting to be matched with an appropriate machine or a job which has vacated from a machine and has been returned to the pool. A running job, by contrast, is a job which is making active progress.

Advanced users may want a visual distinction between two types of running transistions: ”goodput” or ”badput”. Goodput is the transistion state preceding an eventual job completion or checkpoint. Badput is the transistion state preceding a non-checkpointed eviction event. Note that ”badput” is potentially a misleading nonmenclature; a job which is not checkpointed by the Condor program may checkpoint itself or make progress in some other way. To view these two transistion as distinct transistions, select the appropriate option from the ”View” menu.

2.13.2 Events

There are two basic kinds of events: checkpoint events and error events. Plus advanced users can ask to see more events.

2.13.3 Zooming

To view any arbitrary selection of jobs in a job file, use the job selector tool. Jobs appear visually by order of appearence within the actual text log file. For example, the log file might contain jobs 775.1, 775.2, 775.3, 775.4, and 775.5, which appear in that order. A user who wishes to see only jobs 775.2 and 775.5 can select only these two jobs in the job selector tool and click the "Ok" or "Apply" button. The job selector supports double clicking; double click on any single job to see it drawn in isolation.

2.13.4 Zooming

To view a small area of the log file, zoom in on the area which you would like to see in greater detail. You can zoom in, out and do a full zoom. A full zoom redraws the log file in its entirety. For example, if you have zoomed in very close and would like to go all the way back out, you could do so with a succession of zoom outs or with one full zoom.

There is a difference between using the menu driven zooming and the mouse driven zooming. The menu driven zooming will recenter itself around the current center, whereas mouse driven zooming will recenter itself (as much as possible) around the mouse click. To help you refind the clicked area, a box will flash after the zoom. This is called the "zoom finder" and it can be turned off in the zoom menu if you prefer.

2.13.5 Keyboard and Mouse Shortcuts

1. The Keyboard shortcuts:
   - Arrows - an approximate ten percent scrollbar movement
   - PageUp and PageDown - an approximate one hundred percent scrollbar movemnet
   - Control + Left or Right - approximate one hundred percent scrollbar movement
   - End and Home - scrollbar movement to the vertical extreme
   - Others - as seen beside menu items
2. The mouse shortcuts:
   - Control + Left click - zoom in
   - Control + Right click - zoom out
   - Shift + left click - re-center

2.14 About How Condor Jobs Vacate Machines

When Condor needs a job to vacate a machine for whatever reason, it sends the job an asynchronous signal specified in the `KillSig` attribute of the job's ClassAd. The value of this attribute can be

specified by the user at submit time by placing the **kill_sig** option in the Condor submit description file.

If a program wanted to do some special work when required to vacate a machine, the program may set up a signal handler to use a trappable signal as an indication to clean up. When submitting this job, this clean up signal is specified to be used with **kill_sig**. Note that the clean up work needs to be quick. If the job takes too long to go away, Condor follows up with a SIGKILL signal which immediately terminates the process.

A job that is linked using *condor_compile* and is subsequently submitted into the standard universe, will checkpoint and exit upon receipt of a SIGTSTP signal. Thus, SIGTSTP is the default value for `KillSig` when submitting to the standard universe. The user's code may still checkpoint itself at any time by calling one of the following functions exported by the Condor libraries:

`ckpt()` Performs a checkpoint and then returns.

`ckpt_and_exit()` Checkpoints and exits; Condor will then restart the process again later, potentially on a different machine.

For jobs submitted into the vanilla universe, the default value for `KillSig` is SIGTERM, the usual method to nicely terminate a Unix program.

2.15 Special Environment Considerations

2.15.1 AFS

The Condor daemons do not run authenticated to AFS; they do not possess AFS tokens. Therefore, no child process of Condor will be AFS authenticated. The implication of this is that you must set file permissions so that your job can access any necessary files residing on an AFS volume without relying on having your AFS permissions.

If a job you submit to Condor needs to access files residing in AFS, you have the following choices:

1. Copy the needed files from AFS to either a local hard disk where Condor can access them using remote system calls (if this is a standard universe job), or copy them to an NFS volume.
2. If you must keep the files on AFS, then set a host ACL (using the AFS *fs setacl* command) on the subdirectory to serve as the current working directory for the job. If a standard universe job, then the host ACL needs to give read/write permission to any process on the submit machine. If vanilla universe job, then you need to set the ACL such that any host in the pool can access the files without being authenticated. If you do not know how to use an AFS host ACL, ask the person at your site responsible for the AFS configuration.

The Condor Team hopes to improve upon how Condor deals with AFS authentication in a subsequent release.

Please see section 3.10.1 on page 179 in the Administrators Manual for further discussion of this problem.

2.15.2 NFS Automounter

If your current working directory when you run *condor_submit* is accessed via an NFS automounter, Condor may have problems if the automounter later decides to unmount the volume before your job has completed. This is because *condor_submit* likely has stored the dynamic mount point as the job's initial current working directory, and this mount point could become automatically unmounted by the automounter.

There is a simple work around: When submitting your job, use the *initialdir* command in your submit description file to point to the stable access point. For example, suppose the NFS automounter is configured to mount a volume at mount point `/a/myserver.company.com/vol1/johndoe` whenever the directory `/home/johndoe` is accessed. Adding the following line to the submit description file solves the problem.

```
initialdir = /home/johndoe
```

2.15.3 Condor Daemons That Do Not Run as root

Condor is normally installed such that the Condor daemons have root permission. This allows Condor to run the condor_shadow process and your job with your UID and file access rights. When Condor is started as root, your Condor jobs can access whatever files you can.

However, it is possible that whomever installed Condor did not have root access, or decided not to run the daemons as root. That is unfortunate, since Condor is designed to be run as the Unix user root. To see if Condor is running as root on a specific machine, enter the command

```
condor_status -master -l <machine-name>
```

where `machine-name` is the name of the specified machine. This command displays a condor_master ClassAd; if the attribute `RealUid` equals zero, then the Condor daemons are indeed running with root access. If the `RealUid` attribute is not zero, then the Condor daemons do not have root access.

NOTE: The Unix program *ps* is *not* an effective method of determining if Condor is running with root access. When using *ps*, it may often appear that the daemons are running as the condor user instead of root. However, note that the *ps*, command shows the current *effective* owner of the process, not the *real* owner. (See the *getuid*(2) and *geteuid*(2) Unix man pages for details.) In Unix, a process running under the real UID of root may switch its effective UID. (See the *seteuid*(2) man

page.) For security reasons, the daemons only set the effective uid to root when absolutely necessary (to perform a privileged operation).

If they are not running with root access, you need to make any/all files and/or directories that your job will touch readable and/or writable by the UID (user id) specified by the RealUid attribute. Often this may mean using the Unix command `chmod 777` on the directory where you submit your Condor job.

2.16 Potential Problems

2.16.1 Renaming of argv[0]

When Condor starts up your job, it renames argv[0] (which usually contains the name of the program) to condor_exec. This is convenient when examining a machine's processes with the Unix command *ps*; the process is easily identified as a Condor job.

Unfortunately, some programs read argv[0] expecting their own program name and get confused if they find something unexpected like condor_exec.

CHAPTER

THREE

Administrators' Manual

3.1 Introduction

This is the Condor Administrator's Manual for Unix. Its purpose is to aid in the installation and administration of a Condor pool. For help on using Condor, see the Condor User's Manual.

A Condor pool is comprised of a single machine which serves as the *central manager*, and an arbitrary number of other machines that have joined the pool. Conceptually, the pool is a collection of resources (machines) and resource requests (jobs). The role of Condor is to match waiting requests with available resources. Every part of Condor sends periodic updates to the central manager, the centralized repository of information about the state of the pool. Periodically, the central manager assesses the current state of the pool and tries to match pending requests with the appropriate resources.

Each resource has an owner, the user who works at the machine. This person has absolute power over their own resource and Condor goes out of its way to minimize the impact on this owner caused by Condor. It is up to the resource owner to define a policy for when Condor requests will serviced and when they will be denied.

Each resource request has an owner as well: the user who submitted the job. These people want Condor to provide as many CPU cycles as possible for their work. Often the interests of the resource owners are in conflict with the interests of the resource requesters.

The job of the Condor administrator is to configure the Condor pool to find the happy medium that keeps both resource owners and users of resources satisfied. The purpose of this manual is to help you understand the mechanisms that Condor provides to enable you to find this happy medium for your particular set of users and resource owners.

3.1.1 The Different Roles a Machine Can Play

Every machine in a Condor pool can serve a variety of roles. Most machines serve more than one role simultaneously. Certain roles can only be performed by single machines in your pool. The following list describes what these roles are and what resources are required on the machine that is providing that service:

Central Manager There can be only one central manager for your pool. The machine is the collector of information, and the negotiator between resources and resource requests. These two halves of the central manager's responsibility are performed by separate daemons, so it would be possible to have different machines providing those two services. However, normally they both live on the same machine. This machine plays a very important part in the Condor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the Condor system (although all current matches remain in effect until they are broken by either party involved in the match). Therefore, choose for central manager a machine that is likely to be online all the time, or at least one that will be rebooted quickly if something goes wrong. The central manager will ideally have a good network connection to all the machines in your pool, since they all send updates over the network to the central manager. All queries go to the central manager.

Execute Any machine in your pool (including your Central Manager) can be configured for whether or not it should execute Condor jobs. Obviously, some of your machines will have to serve this function or your pool won't be very useful. Being an execute machine doesn't require many resources at all. About the only resource that might matter is disk space, since if the remote job dumps core, that file is first dumped to the local disk of the execute machine before being sent back to the submit machine for the owner of the job. However, if there isn't much disk space, Condor will simply limit the size of the core file that a remote job will drop. In general the more resources a machine has (swap space, real memory, CPU speed, etc.) the larger the resource requests it can serve. However, if there are requests that don't require many resources, any machine in your pool could serve them.

Submit Any machine in your pool (including your Central Manager) can be configured for whether or not it should allow Condor jobs to be submitted. The resource requirements for a submit machine are actually much greater than the resource requirements for an execute machine. First of all, every job that you submit that is currently running on a remote machine generates another process on your submit machine. So, if you have lots of jobs running, you will need a fair amount of swap space and/or real memory. In addition all the checkpoint files from your jobs are stored on the local disk of the machine you submit from. Therefore, if your jobs have a large memory image and you submit a lot of them, you will need a lot of disk space to hold these files. This disk space requirement can be somewhat alleviated with a checkpoint server (described below), however the binaries of the jobs you submit are still stored on the submit machine.

Checkpoint Server One machine in your pool can be configured as a checkpoint server. This is optional, and is not part of the standard Condor binary distribution. The checkpoint server is a centralized machine that stores all the checkpoint files for the jobs submitted in your pool.

This machine should have lots of disk space and a good network connection to the rest of your pool, as the traffic can be quite heavy.

Now that you know the various roles a machine can play in a Condor pool, we will describe the actual daemons within Condor that implement these functions.

3.1.2 The Condor Daemons

The following list describes all the daemons and programs that could be started under Condor and what they do:

condor_master This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send e-mail to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

condor_startd This daemon represents a given resource (namely, a machine capable of running jobs) to the Condor pool. It advertises certain attributes about that resource that are used to match it with pending resource requests. The startd will run on any machine in your pool that you wish to be able to execute jobs. It is responsible for enforcing the policy that resource owners configure which determines under what conditions remote jobs will be started, suspended, resumed, vacated, or killed. When the startd is ready to execute a Condor job, it spawns the *condor_starter*, described below.

condor_starter This program is the entity that actually spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the starter notices this, sends back any status information to the submitting machine, and exits.

condor_schedd This daemon represents resources requests to the Condor pool. Any machine that you wish to allow users to submit jobs from needs to have a *condor_schedd* running. When users submit jobs, they go to the schedd, where they are stored in the *job queue*, which the schedd manages. Various tools to view and manipulate the job queue (such as *condor_submit*, *condor_q*, or *condor_rm*) all must connect to the schedd to do their work. If the schedd is down on a given machine, none of these commands will work.

The schedd advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a schedd has been matched with a given resource, the schedd spawns a *condor_shadow* (described below) to serve that particular request.

condor_shadow This program runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's standard universe, which perform remote system calls, do so via the *condor_shadow*. Any system call performed on the remote execute machine is sent over the network, back to the *condor_shadow* which actually performs the system call (such as file I/O) on the submit machine, and the result is sent back over the network to the remote job. In addition, the shadow is responsible for making decisions about the request (such as where checkpoint files should be stored, how certain files should be accessed, etc).

condor_collector This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons (except the negotiator) periodically send ClassAd updates to the collector. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool (such as jobs that have been submitted to a given schedd). The *condor_status* command can be used to query the collector for specific information about various parts of Condor. In addition, the Condor daemons themselves query the collector for important information, such as what address to use for sending commands to a remote machine.

condor_negotiator This daemon is responsible for all the match-making within the Condor system. Periodically, the negotiator begins a *negotiation cycle*, where it queries the collector for the current state of all the resources in the pool. It contacts each schedd that has waiting resource requests in priority order, and tries to match available resources with those requests. The negotiator is responsible for enforcing user priorities in the system, where the more resources a given user has claimed, the less priority they have to acquire more resources. If a user with a better priority has jobs that are waiting to run, and resources are claimed by a user with a worse priority, the negotiator can preempt that resource and match it with the user with better priority.

NOTE: A higher numerical value of the user priority in Condor translate into worse priority for that user. The best priority you can have is 0.5, the lowest numerical value, and your priority gets worse as this number grows.

condor_kbdd This daemon is only needed on Digital Unix and IRIX. On these platforms, the *condor_startd* cannot determine console (keyboard or mouse) activity directly from the system. The *condor_kbdd* connects to the X Server and periodically checks to see if there has been any activity. If there has, the kbdd sends a command to the startd. That way, the startd knows the machine owner is using the machine again and can perform whatever actions are necessary, given the policy it has been configured to enforce.

condor_ckpt_server This is the checkpoint server. It services requests to store and retrieve checkpoint files. If your pool is configured to use a checkpoint server but that machine (or the server itself is down) Condor will revert to sending the checkpoint files for a given job back to the submit machine.

See figure 3.1 for a graphical representation of the pool architecture.

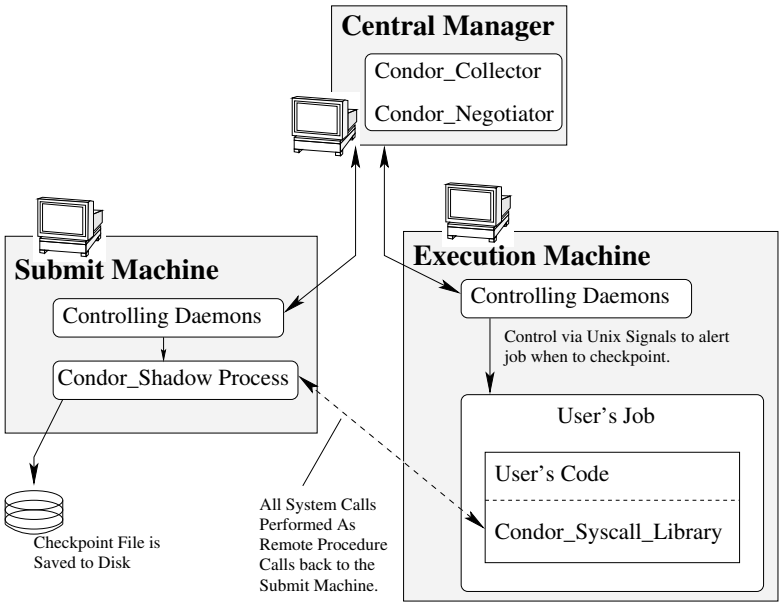


Figure 3.1: Pool Architecture

3.2 Installation of Condor

This section contains the instructions for installing Condor at your Unix site. Read this entire section before starting installation. The installation will have a default configuration that can be customized. Sections of the manual that follow this one explain customization.

Please read the copyright and disclaimer information in section on page xii of the manual, or in the file `LICENSE.TXT`, before proceeding. Installation and use of Condor is acknowledgment that you have read and agree to the terms.

3.2.1 Obtaining Condor

The first step to installing Condor is to download it from the Condor web site, http://www.cs.wisc.edu/condor. The downloads are available from the downloads page, at http://www.cs.wisc.edu/condor/downloads/.

The platform-dependent Condor files are currently available from two sites. The main site is at the University of Wisconsin–Madison, Madison, Wisconsin, USA. A second site is the Istituto Nazionale di Fisica Nucleare Sezione di Bologna, Bologna, Italy. Please choose the site nearest you.

Make note of the location of where you download the binary into.

3.2.2 Condor Distribution Contents

The Condor binary distribution is packaged in the following 5 files and 2 directories:

DOC directions on where to find Condor documentation

INSTALL these installation directions

LICENSE.TXT the licensing agreement. By installing Condor, you agree to the contents of this file

README general information

condor_install the Perl script used to install and configure Condor

examples directory containing C, Fortran and C++ example programs to run with Condor

release.tar tar file of the release directory, which contains the Condor binaries and libraries

Before you install, please consider joining the condor-world mailing list. Traffic on this list is kept to an absolute minimum. It is only used to announce new releases of Condor. To subscribe, send a message to majordomo@cs.wisc.edu with the body:

```
subscribe condor-world
```

3.2.3 Preparation

Before installation, make a few important decisions about the basic layout of your pool. The decisions answer the questions:

1. What machine will be the central manager?
2. Will Condor run as root or not?
3. Who will be administering Condor on the machines in your pool?
4. Will you have a Unix user named condor and will its home directory be shared?
5. Where should the machine-specific directories for Condor go?
6. Where should the parts of the Condor system be installed?
   - Config files
   - Release directory
     - user binaries
     - system binaries

  - `lib` directory
  - `etc` directory
- Documentation

7. Am I using AFS?
8. Do I have enough disk space for Condor?

If you feel you already know the answers to these questions, you can skip to the Installation Procedure section below, section 3.2.4 on page 75. If you are unsure about any of them, read on.

What machine will be the central manager?

One machine in your pool must be the central manager. Install Condor on this machine first. This is the centralized information repository for the Condor pool, and it is also the machine that does match-making between available machines and submitted jobs. If the central manager machine crashes, any currently active matches in the system will keep running, but no new matches will be made. Moreover, most Condor tools will stop working. Because of the importance of this machine for the proper functioning of Condor, install the central manager on a machine that is likely to stay up all the time, or on one that will be rebooted quickly if it does crash. Also consider network traffic and your network layout when choosing your central manager. All the daemons send updates (by default, every 5 minutes) to this machine.

Will Condor run as root or not?

Start up the Condor daemons as the Unix user root. Without this, Condor can do very little to enforce security and policy decisions. You can install Condor as any user, however there are both serious security and performance consequences. Please see section 3.7.2 on page 152 in the manual for the details and ramifications of on running Condor as a Unix user other than root.

Who will administer Condor?

Either root will be administering Condor directly, or someone else would be acting as the Condor administrator. If root has delegated the responsibility to another person but doesn't want to grant that person root access, root can specify a `condor_config.root` file that will override settings in the other condor configuration files. This way, the global `condor_config` file can be owned and controlled by whoever is condor-admin, and the condor_config.root can be owned and controlled only by root. Settings that would compromise root security (such as which binaries are started as root) can be specified in the `condor_config.root` file while other settings that only control policy or condor-specific settings can still be controlled without root access.

Will you have a Unix user named condor, and will its home directory be shared?

To simplify installation of Condor, create a Unix user named condor on all machines in the pool. The Condor daemons will create files (such as the log files) owned by this user, and the home directory can be used to specify the location of files and directories needed by Condor. The home directory of this user can either be shared among all machines in your pool, or could be a separate home directory on the local partition of each machine. Both approaches have advantages and disadvantages. Having the directories centralized can make administration easier, but also concentrates the resource usage such that you potentially need a lot of space for a single shared home directory. See the section below on machine-specific directories for more details.

If you choose not to create a user named condor, then you must specify via the CONDOR_IDS environment variable which uid.gid pair should be used for the ownership of various Condor files. See section 3.7.1 on UIDs in Condor on page 150 in the Administrator's Manual for details.

Where should the machine-specific directories for Condor go?

Condor needs a few directories that are unique on every machine in your pool. These are `spool`, `log`, and `execute`. Generally, all three are subdirectories of a single machine specific directory called the local directory (specified by the `LOCAL_DIR` macro in the configuration file).

If you have a Unix user named condor with a local home directory on each machine, the `LOCAL_DIR` could just be user condor's home directory (`LOCAL_DIR = $(TILDE)` in the configuration file). If this user's home directory is shared among all machines in your pool, you would want to create a directory for each host (named by host name) for the local directory (for example, `LOCAL_DIR = $(TILDE)`/hosts/`$(HOSTNAME)`). If you do not have a condor account on your machines, you can put these directories wherever you'd like. However, where to place them will require some thought, as each one has its own resource needs:

`execute` This is the directory that acts as the current working directory for any Condor jobs that run on a given execute machine. The binary for the remote job is copied into this directory, so there must be enough space for it. (Condor will not send a job to a machine that does not have enough disk space to hold the initial binary). In addition, if the remote job dumps core for some reason, it is first dumped to the execute directory before it is sent back to the submit machine. So, put the execute directory on a partition with enough space to hold a possible core file from the jobs submitted to your pool.

`spool` The `spool` directory holds the job queue and history files, and the checkpoint files for all jobs submitted from a given machine. As a result, disk space requirements for the `spool` directory can be quite large, particularly if users are submitting jobs with very large executables or image sizes. By using a checkpoint server (see section 3.10.5 on Installing a Checkpoint Server on page 185 for details), you can ease the disk space requirements, since all checkpoint files are stored on the server instead of the spool directories for each machine. However, the initial checkpoint files (the executables for all the clusters you submit) are still stored in the spool directory, so you will need some space, even with a checkpoint server.

log Each Condor daemon writes its own log file, and each log file is placed in the `log` directory. You can specify what size you want these files to grow to before they are rotated, so the disk space requirements of the directory are configurable. The larger the log files, the more historical information they will hold if there is a problem, but the more disk space they use up. If you have a network file system installed at your pool, you might want to place the log directories in a shared location (such as `/usr/local/condor/logs/$(HOSTNAME)`), so that you can view the log files from all your machines in a single location. However, if you take this approach, you will have to specify a local partition for the `lock` directory (see below).

lock Condor uses a small number of lock files to synchronize access to certain files that are shared between multiple daemons. Because of problems encountered with file locking and network file systems (particularly NFS), these lock files should be placed on a local partition on each machine. By default, they are placed in the `log` directory. If you place your `log` directory on a network file system partition, specify a local partition for the lock files with the `LOCK` parameter in the configuration file (such as `/var/lock/condor`).

Generally speaking, it is recommended that you do not put these directories (except `lock`) on the same partition as `/var`, since if the partition fills up, you will fill up `/var` as well. This will cause lots of problems for your machines. Ideally, you will have a separate partition for the Condor directories. Then, the only consequence of filling up the directories will be Condor's malfunction, not your whole machine.

Where should the parts of the Condor system be installed?

- Configuration Files
- Release directory
  - User Binaries
  - System Binaries
  - `lib` Directory
  - `etc` Directory
- Documentation

Configuration Files There are a number of configuration files that allow you different levels of control over how Condor is configured at each machine in your pool. The global configuration file is shared by all machines in the pool. For ease of administration, this file should be located on a shared file system, if possible. In addition, there is a local configuration file for each machine, where you can override settings in the global file. This allows you to have different daemons running, different policies for when to start and stop Condor jobs, and so on. You can also have configuration files specific to each platform in your pool. See section 3.10.2 on page 180 about Configuring Condor for Multiple Platforms for details.

In addition, because we recommend that you start the Condor daemons as root, we allow you to create configuration files that are owned and controlled by root that will override any other Condor settings. This way, if the Condor administrator is not root, the regular Condor configuration files can be owned and writable by condor-admin, but root does not have to grant root access to this person. See section **??** on page **??** in the manual for a detailed discussion of the root configuration files, if you should use them, and what settings should be in them.

In general, there are a number of places that Condor will look to find its configuration files. The first file it looks for is the global configuration file. These locations are searched in order until a configuration file is found. If none contain a valid configuration file, Condor will print an error message and exit:

1. File specified in `CONDOR_CONFIG` environment variable
2. `/etc/condor/condor_config`
3. `~condor/condor_config`

If you specify a file in the `CONDOR_CONFIG` environment variable and there's a problem reading that file, Condor will print an error message and exit right away, instead of continuing to search the other options. However, if no `CONDOR_CONFIG` environment variable is set, Condor will search through the other options.

Next, Condor tries to load the local configuration file(s). The only way to specify the local configuration file(s) is in the global configuration file, with the `LOCAL_CONFIG_FILE` macro. If that macro is not set, no local configuration file is used. This macro can be a list of files or a single file.

The root configuration files come in last. The global file is searched for in the following places:

1. `/etc/condor/condor_config.root`
2. `~condor/condor_config.root`

The local root configuration file(s) are found with the `LOCAL_ROOT_CONFIG_FILE` macro. If that is not set, no local root configuration file is used. This macro can be a list of files or a single file.

Release Directory Every binary distribution contains a `release.tar` file that contains four subdirectories: `bin`, `etc`, `lib` and `sbin`. Wherever you choose to install these 4 directories we call the release directory (specified by the `RELEASE_DIR` macro in the configuration file). Each release directory contains platform-dependent binaries and libraries, so you will need to install a separate one for each kind of machine in your pool. For ease of administration, these directories should be located on a shared file system, if possible.

- User Binaries:
  All of the files in the `bin` directory are programs the end Condor users should expect to have in their path. You could either put them in a well known location (such as `/usr/local/condor/bin`) which you have Condor users add to their `PATH` environment variable, or copy those files directly into a well known place already in user's PATHs (such as `/usr/local/bin`). With the above examples, you could

also leave the binaries in `/usr/local/condor/bin` and put in soft links from `/usr/local/bin` to point to each program.

- System Binaries:
  All of the files in the `sbin` directory are Condor daemons and agents, or programs that only the Condor administrator would need to run. Therefore, add these programs only to the `PATH` of the Condor administrator.
- `lib` Directory:
  The files in the `lib` directory are the Condor libraries that must be linked in with user jobs for all of Condor's checkpointing and migration features to be used. `lib` also contains scripts used by the *condor_compile* program to help re-link jobs with the Condor libraries. These files should be placed in a location that is world-readable, but they do not need to be placed in anyone's `PATH`. The *condor_compile* script checks the configuration file for the location of the `lib` directory.
- `etc` Directory:
  `etc` contains an `examples` subdirectory which holds various example configuration files and other files used for installing Condor. `etc` is the recommended location to keep the master copy of your configuration files. You can put in soft links from one of the places mentioned above that Condor checks automatically to find its global configuration file.

Documentation The documentation provided with Condor is currently available in HTML, Postscript and PDF (Adobe Acrobat). It can be locally installed wherever is customary at your site. You can also find the Condor documentation on the web at: http://www.cs.wisc.edu/condor/manual.

Am I using AFS?

If you are using AFS at your site, be sure to read the section 3.10.1 on page 178 in the manual. Condor does not currently have a way to authenticate itself to AFS. A solution is not ready for Version 6.4.0. This implies that you are probably not going to want to have the `LOCAL_DIR` for Condor on AFS. However, you can (and probably should) have the Condor `RELEASE_DIR` on AFS, so that you can share one copy of those files and upgrade them in a centralized location. You will also have to do something special if you submit jobs to Condor from a directory on AFS. Again, read manual section 3.10.1 for all the details.

Do I have enough disk space for Condor?

Condor takes up a fair amount of space. This is another reason why it is a good idea to have it on a shared file system. The size requirements for the downloads are given on the downloads page. They currently vary from about 20 Mbytes (statically linked HP Unix on a PA RISC) to more than 50 Mbytes (dynamically linked Irix on an SGI).

In addition, you will need a lot of disk space in the local directory of any machines that are submitting jobs to Condor. See question 5 above for details on this.

3.2.4 Installation Procedure

IF YOU HAVE DECIDED TO CREATE A condor USER AND GROUP, DO THAT ON ALL YOUR MACHINES BEFORE YOU DO ANYTHING ELSE.

The easiest way to install Condor is to use one or both of the scripts provided to help you: *condor_install* and *condor_init*. Run these scripts as the user that you are going to run the Condor daemons as. First, run *condor_install* on the machine that will be a file server for shared files used by Condor, such as the release directory, and possibly the condor user's home directory. When you do, choose the "full-install" option in step #1 described below.

Once you have run *condor_install* on a file server to set up your release directory and configure Condor for your site, you should run *condor_init* on any other machines in your pool to create any locally used files that are not created by *condor_install*. In the most simple case, where nearly all of Condor is installed on a shared file system, even though *condor_install* will create nearly all the files and directories you need, you will still need to use *condor_init* to create the `LOCK` directory on the local disk of each machine. If you have a shared release directory, but the `LOCAL_DIR` is local on each machine, *condor_init* will create all the directories and files needed in `LOCAL_DIR`. In addition, *condor_init* will create any soft links on each machine that are needed so that Condor can find its global configuration file.

If you do not have a shared file system, you need to run *condor_install* on each machine in your pool to set up Condor. In this case, there is no need to run *condor_init* at all.

In addition, you will want to run *condor_install* on your central manager machine if that machine is different from your file server, using the "central-manager" option in step #1 described below. Run *condor_install* on your file server first, then on your central manager. If this step fails for some reason (NFS permissions, etc), you can do it manually quite easily. All this does is copy the `condor_config.local.central.manager` file from `<release_dir>/etc/examples` to the proper location for the local configuration file of your central manager machine. If your central manager is an Alpha or an SGI, you might want to add `KBDD` to the `$(DAEMON_LIST)` macro. See section 3.3 Configuring Condor on page 82 of the manual for details.

condor_install assumes you have perl installed in `/usr/bin/perl`. If this is not the case, you can either edit the script to put in the right path, or you will have to invoke perl directly from your shell (assuming perl is in your PATH):

```
% perl condor_install
```

condor_install breaks down the installation procedure into various steps. Each step is clearly numbered. The following section explains what each step is for, and suggests how to answer the questions *condor_install* will ask you for each one.

***condor_install*, step-by-step**

STEP 1: What type of Condor installation do you want? There are three types of Condor installation you might choose: 'submit-only', 'full-install', and 'central-manager'. A submit-only machine can submit jobs to a Condor pool, but Condor jobs will not run on it. A full-install machine can both submit and run Condor jobs.

If you are planning to run Condor jobs on your machines, you should either install and run Condor as root, or as the Unix user condor.

If you are planning to set up a submit-only machine, you can either install Condor machine-wide as root or user condor, or, you can install Condor as yourself into your home directory.

The other possible installation type is setting up a machine as a central manager. If you do a full-install and you say that you want the local host to be your central manager, this step will be done automatically. You should only choose the central-manager option at step 1 if you have already run *condor_install* on your file server and you now want to run *condor_install* on a different machine that will be your central manager.

STEP 2: How many machines are you setting up this way? If you are installing Condor for multiple machines and you have a shared file system, then *condor_install* will prompt you for the host names of each machine you want to add to your Condor pool. If you do not have a shared file system, you will have to run *condor_install* locally on each machine, so *condor_install* does not ask for the names. If you provide a list, it will use the names to automatically create directories and files later. At the end, *condor_install* will dump out this list to a `roster` file which can be used by scripts to help maintain your Condor pool.

If you are only installing Condor on 1 machine, you would answer no to the first question, and move on.

STEP 3: Install the Condor release directory The release directory contains four subdirectories: `bin`, `etc`, `lib` and `sbin`. `bin` contains user-level executable programs. `etc` is the recommended location for your Condor configuration files, and it also includes an `examples` directory with default configuration files and other default files used for installing Condor. `lib` contains libraries to link Condor user programs and scripts used by the Condor system. `sbin` contains all administrative executable programs and the Condor daemons.

If you have multiple machines with a shared file system that will be running Condor, put the release directory on that shared file system so you only have one copy of all the binaries, and so that when you update them, you can do so in one place. Note that the release directory is architecture dependent, so download separate binary distributions for every platform in your pool.

condor_install tries to find an already installed release directory. If it cannot find one, it asks if you have installed one already. If you have not installed one, it tries to do so for you by untarring the `release.tar` file from the binary distribution.

NOTE: If you are only setting up a central manager (you chose 'central manager' in STEP 1), STEP 3 is the last question you will need to answer.

STEP 4: How and where should Condor send e-mail if things go wrong? Various parts of Condor will send e-mail to a condor administrator if something goes wrong that needs human attention. You will need to specify the e-mail address of this administrator.

You also specify the full path to a mail program that Condor will use to send the e-mail. This program needs to understand the **-s** option, to specify a subject for the outgoing message. The default on most platforms will probably be correct. On Linux machines, since there is such variation in Linux distributions and installations, verify that the default works. If the script complains that it cannot find the mail program that was specified, try

```
% which mail
```

to see what mail program is currently in your PATH. If there is none, try

```
% which mailx
```

If you still cannot find anything, ask your system administrator. Verify that the program you use supports **-s**. The man page for that program will probably tell you.

STEP 5: File system and UID domains. While Condor does not depend on a shared file system or common UID space for running jobs in the standard universe, vanilla jobs (ones that are not relinked with the Condor libraries) do need a shared file system and a common UID space. Therefore, it is very important for you to correctly configure Condor with respect to a shared file system. For complete details on what these settings do and how you should answer the questions, read section 3.3.5, Shared File System Configuration File Entries", on page 93.

You will be asked if you have a shared file system. If so, condor_install will configure your `FILESYSTEM_DOMAIN` setting to be set to the domain name of the machine running condor_install. If not, `FILESYSTEM_DOMAIN` will be set to `$(FULL_HOSTNAME)`, indicating that each machine is in its own domain.

For the UID domain, Condor needs to know if all users across all the machines in your pool have a unique UID. If so, `UID_DOMAIN` will be set to the domain name of the machine running condor_install. If not, `UID_DOMAIN` will be set to `$(FULL_HOSTNAME)`, indicating that each machine is in its own domain.

If you have a common `UID_DOMAIN`, condor_install will ask you if have a *soft UID domain*, meaning that although you have unique UIDs, not every machine in your pool has all the users in their individual password files. Please see the description of `SOFT_UID_DOMAIN` in section 3.3.5 on page 94 for details.

STEP 6: Java Universe support in Condor. Condor has the ability to run Java programs with remote i/o, but no checkpointing. If you would like to enable this feature in Condor, then select yes for enabling of Java Universe. The installer will try to determine if you have a valid JVM and prompt you if nothing suitable can be found. If you do not care to use the Java Universe, then it is safe to say no here.

STEP 7: Where should public programs be installed? It is recommended that you install the user-level Condor programs in the release directory, (where they go by default). This way, when you want to install a new version of the Condor binaries, you can just replace your release directory and everything will be updated at once. So, one option is to have Condor users add `<release_dir>/bin` to their PATH, so that they can access the programs. However, we recommend putting in soft links from some directory already in their PATH (such

as `/usr/local/bin`) that point back to the Condor user programs. *condor_install* will do this for you. All you do is tell it what directory to put these links into. This way, users do not have to change their PATH to use Condor, and you can still have the binaries installed in their own location.

If you are installing Condor as neither root nor condor, there is a perl script wrapper to all the Condor tools that is created which sets some appropriate environment variables and automatically passes certain options to the tools. This is all created automatically by *condor_install*. So, you need to tell *condor_install* where to put this perl script. The script itself is linked to itself with many different names, since it is the name that determines the behavior of the script. This script should go somewhere that is in your PATH already, if possible (such as ˜bin).

At this point, the remaining steps differ based on the whether the installation is a full install or a submit-only. Skip to the appropriate section below, based on the kind of installation.

Full Install

STEP 8: What machine will be your central manager? Type in the full host name of the machine you have chosen for your central manager. If *condor_install* cannot find information about the host you typed by querying your name server, it will print out an error message and ask you to confirm.

STEP 9: Where will the local directory go? This is the directory discussed in question 5 of the installation introduction. *condor_install* tries to make some educated guesses as to what directory you want to use for the purpose. Agree to the correct guess, or (when *condor_install* has run out of guesses) type in what you want. Since this directory needs to be unique, it is common to use the host name of each machine in its name. When typing in your own path, you can use '`$(HOSTNAME)`' which *condor_install* (and the Condor configuration files) will expand to the host name of the machine you are currently on. *condor_install* will try to create the corresponding directories for all the machines you told it about in STEP 2 above.

Once you have selected the local directory, *condor_install* creates all the needed subdirectories of each one with the proper permissions. They should have the following permissions and ownerships:

```
drwxr-xr-x   2 condor   root         1024 Mar  6 01:30 execute/
drwxr-xr-x   2 condor   root         1024 Mar  6 01:30 log/
drwxr-xr-x   2 condor   root         1024 Mar  6 01:30 spool/
```

If your local directory is on a shared file system, *condor_install* will prompt you for the location of your lock files, as discussed in question #5 above. In this case, when *condor_install* is finished, you will have to run *condor_init* on each machine in your pool to create the `lock` directory before you can start up Condor.

STEP 10: Where will the local (machine-specific) configuration files go? As discussed in question STEP 6 above, there are a few different levels of Condor configuration files. There is the global configuration file that will be installed in

<release_dir>/etc/condor_config, and there are machine-specific, or local configuration files, that override the settings in the global file. If you are installing on multiple machines or are configuring your central manager machine, you must select a location for your local configuration files.

The two main options are to have a single directory that holds all the local configuration files, each one named $(HOSTNAME).local, or to have the local configuration files go into the individual local directories for each machine. Given a shared file system, we recommend the first option, since it makes it easier to configure your pool from a centralized location.

STEP 11: How shall Condor find its configuration file? Since there are a few known places Condor looks to find your configuration file, we recommend that you put a soft link from one of them to point to <release_dir>/etc/condor_config. This way, you can keep your Condor configuration in a centralized location, but all the Condor daemons and tools will be able to find their configuration files. Alternatively, you can set the CONDOR_CONFIG environment variable to contain <release_dir>/etc/condor_config.

condor_install will ask you if you want to create a soft link from either of the two fixed locations that Condor searches.

Once you have completed STEP 11, you are done. *condor_install* prints out a messages describing what to do next. Please skip to section 3.2.5.

Submit Only

A submit-only installation of Condor implies that the machine will be submitting jobs to one or more established Condor pools. Configuration for this installation needs to account for the other pools.

For the submit-only installation, STEP 6 continues and completes the installation.

STEP 6: continued. A submit-only machine has the option of submission to more than one Condor pool. The full host name of the central manager is required for each pool. The first entered becomes the default for start up and job submission.

There is a separate configuration file for each pool. The location of each file is specified.

Identification of each pool requires a unique name. A final question sets a name for each pool. The name will be the argument for **-pool** command line options.

3.2.5 Condor is installed... now what?

Now that Condor has been installed on your machine(s), there are a few things you should check before you start up Condor.

1. Read through the <release_dir>/etc/condor_config file. There are a lot of possible settings and you should at least take a look at the first two main sections to make sure

everything looks okay. In particular, you might want to set up security for Condor. See the section 3.7.3 on page 153 to learn how to do this.

2. Condor can monitor the activity of your mouse and keyboard, provided that you tell it where to look. You do this with the CONSOLE_DEVICES entry in the condor_startd section of the configuration file. On most platforms, reasonable defaults are provided. For example, the default device for the mouse on Linux is 'mouse', since most Linux installations have a soft link from /dev/mouse that points to the right device (such as tty00 if you have a serial mouse, psaux if you have a PS/2 bus mouse, etc). If you do not have a /dev/mouse link, you should either create one (you will be glad you did), or change the CONSOLE_DEVICES entry in Condor's configuration file. This entry is a comma separated list, so you can have any devices in /dev count as 'console devices' and activity will be reported in the condor_startd's ClassAd as ConsoleIdleTime.

3. (Linux only) Condor needs to be able to find the utmp file. According to the Linux File System Standard, this file should be /var/run/utmp. If Condor cannot find it there, it looks in /var/adm/utmp. If it still cannot find it, it gives up. So, if your Linux distribution places this file somewhere else, be sure to put a soft link from /var/run/utmp to point to the real location.

3.2.6 Starting up the Condor daemons

To start up the Condor daemons, execute <release_dir>/sbin/condor_master. This is the Condor master, whose only job in life is to make sure the other Condor daemons are running. The master keeps track of the daemons, restarts them if they crash, and periodically checks to see if you have installed new binaries (and if so, restarts the affected daemons).

If you are setting up your own pool, you should start Condor on your central manager machine first. If you have done a submit-only installation and are adding machines to an existing pool, the start order does not matter.

To ensure that Condor is running, you can run either:

```
ps -ef | egrep condor_
```

or

```
ps -aux | egrep condor_
```

depending on your flavor of Unix. On your central manager machine you should have processes for:

- condor_master
- condor_collector
- condor_negotiator

- condor_startd
- condor_schedd

On all other machines in your pool you should have processes for:

- condor_master
- condor_startd
- condor_schedd

(NOTE: On Alphas and IRIX machines, there will also be a condor_kbdd – see section 3.10.4 on page 184 of the manual for details.) If you have set up a submit-only machine, you will only see:

- condor_master
- condor_schedd

Once you are sure the Condor daemons are running, check to make sure that they are communicating with each other. You can run *condor_status* to get a one line summary of the status of each machine in your pool.

Once you are sure Condor is working properly, you should add *condor_master* into your startup/bootup scripts (i.e. `/etc/rc`) so that your machine runs *condor_master* upon bootup. *condor_master* will then fire up the necessary Condor daemons whenever your machine is rebooted.

If your system uses System-V style init scripts, you can look in `<release_dir>/etc/examples/condor.boot` for a script that can be used to start and stop Condor automatically by init. Normally, you would install this script as `/etc/init.d/condor` and put in soft link from various directories (for example, `/etc/rc2.d`) that point back to `/etc/init.d/condor`. The exact location of these scripts and links will vary on different platforms.

If your system uses BSD style boot scripts, you probably have an `/etc/rc.local` file. Add a line to start up `<release_dir>/sbin/condor_master`.

3.2.7 The Condor daemons are running... now what?

Now that the Condor daemons are running, there are a few things you can and should do:

1. (Optional) Do a full install for the *condor_compile* script. condor_compile assists in linking jobs with the Condor libraries to take advantage of all of Condor's features. As it is currently installed, it will work by placing it in front of any of the following commands that you would normally use to link your code: gcc, g++, g77, cc, acc, c89, CC, f77, fort77 and ld. If you complete the full install, you will be able to use condor_compile with any command whatsoever, in particular, make. See section 3.10.3 on page 182 in the manual for directions.

2. Try building and submitting some test jobs. See `examples/README` for details.
3. If your site uses the AFS network file system, see section 3.10.1 on page 178 in the manual.
4. We strongly recommend that you start up Condor (run the *condor_master* daemon) as user root. If you must start Condor as some user other than root, see section 3.7.2 on page 152.

3.3 Configuring Condor

This section describes how to configure all parts of the Condor system. General information about the configuration files and their syntax is followed by a description of settings that affect all Condor daemons and tools. At the end is a section describing the settings for each part of Condor. The settings that control the policy under which Condor will start, suspend, resume, vacate or kill jobs are described in section 3.6 on Configuring Condor's Job Execution Policy.

3.3.1 Introduction to Configuration Files

The Condor configuration files are used to customize how Condor operates at a given site. The basic configuration as shipped with Condor works well for most sites, with few exceptions.

See section 3.2 on page 68 for details on where Condor's configuration files are found.

Each Condor program will, as part of its initialization process, configure itself by calling a library routine which parses the various configuration files that might be used including pool-wide, platform-specific, machine-specific, and root-owned configuration files. The result is a list of constants and expressions which are evaluated as needed at run time.

The order in which attributes are defined is important, since later definitions will override existing definitions. This is particularly important if configuration files are broken up using the `LOCAL_CONFIG_FILE` setting described in sections 3.3.2 and 3.10.2 below.

Config File Macros

Macro definitions are of the form:

```
<macro_name> = <macro_definition>
```

NOTE: You must have white space between the macro name, the "=" sign, and the macro definition.

Macro invocations are of the form:

```
$(macro_name)
```

Macro definitions may contain references to other macros, even ones that aren't yet defined (so long as they are eventually defined in your config files somewhere). All macro expansion is done after all config files have been parsed (with the exception of macros that reference themselves, described below).

```
A = xxx
C = $(A)
```

is a legal set of macro definitions, and the resulting value of `C` is `xxx`. Note that `C` is actually bound to `$(A)`, not its value.

As a further example,

```
A = xxx
C = $(A)
A = yyy
```

is also a legal set of macro definitions, and the resulting value of `C` is `yyy`.

A macro may be incrementally defined by invoking itself in its definition. For example,

```
A = xxx
B = $(A)
A = $(A)yyy
A = $(A)zzz
```

is a legal set of macro definitions, and the resulting value of `A` is `xxxyyyzzz`. Note that invocations of a macro in its own definition are immediately expanded. `$(A)` is immediately expanded in line 3 of the example. If it were not, then the definition would be impossible to evaluate.

NOTE: Macros should not be incrementally defined in the `LOCAL_ROOT_CONFIG_FILE` for security reasons.

NOTE: Condor used to distinguish between "macros" and "expressions" in its config files. Begining with Condor version 6.1.13, this distinction has been removed. For backwards compatibility, you can still use ":" instead of "=" in your config files, and these attributes will just be treated as macros.

Config File Environment References

References to the Condor process's environment are also allowed in the configuration file. Environment references are of the form:

```
$ENV(environment_variable_name)
```

For example,

```
A = $ENV(HOME)
```

binds A to the value of the HOME environment variable. Environment references are not currently used in standard Condor configurations. However, they can sometimes be useful in custom configurations.

Comments and Line Continuations

A Condor configuration file can also contain comments or line continuations. A comment is any line beginning with a "#" character. A continuation is any entry that continues across multiples lines. Line continuation is accomplished by placing the "\" character at the end of any line to be continued onto another. Valid examples of line continuation are

```
START = (KeyboardIdle > 15 * $(MINUTE)) && \
        ((LoadAvg - CondorLoadAvg) <= 0.3)
```

and

```
ADMIN_MACHINES = condor.cs.wisc.edu, raven.cs.wisc.edu, \
                 stork.cs.wisc.edu, ostrich.cs.wisc.edu, \
                 bigbird.cs.wisc.edu
HOSTALLOW_ADMIN = $(ADMIN_MACHINES)
```

Pre-Defined Macros

Condor provides pre-defined macros that help configure Condor. Pre-defined macros are listed as `$(macro_name)`.

This first set are entries whose values are determined at run time and cannot be overwritten. These are inserted automatically by the library routine which parses the configuration files.

$(FULL_HOSTNAME) The fully qualified hostname of the local machine (hostname plus domain name).

$(HOSTNAME) The hostname of the local machine (no domain name).

$(IP_ADDRESS) The ASCII string version of the local machine's IP address.

$(TILDE) The full path to the home directory of the Unix user condor, if such a user exists on the local machine.

$(SUBSYSTEM) The subsystem name of the daemon or tool that is evaluating the macro. This is a unique string which identifies a given daemon within the Condor system. The possible subsystem names are:

- STARTD
- SCHEDD
- MASTER
- COLLECTOR
- NEGOTIATOR
- KBDD
- SHADOW
- STARTER
- CKPT_SERVER
- SUBMIT
- GRIDMANAGER

This second set of macros are entries whose default values are determined automatically at run-time but which can be overwritten.

$(ARCH) Defines the string used to identify the architecture of the local machine to Condor. The *condor_startd* will advertise itself with this attribute so that users can submit binaries compiled for a given platform and force them to run on the correct machines. *condor_submit* will append a requirement to the job ClassAd that it must run on the same `ARCH` and `OPSYS` of the machine where it was submitted, unless the user specifies `ARCH` and/or `OPSYS` explicitly in their submit file. See the the *condor_submit* manual page on page 395 for details.

$(OPSYS) Defines the string used to identify the operating system of the local machine to Condor. If it is not defined in the configuration file, Condor will automatically insert the operating system of this machine as determined by *uname*.

$(FILESYSTEM_DOMAIN) Defaults to the fully qualified hostname of the machine it is evaluated on. See section 3.3.5, Shared File System Configuration File Entries for the full description of its use and under what conditions you would want to change it.

$(UID_DOMAIN) Defaults to the fully qualified hostname of the machine it is evaluated on. See section 3.3.5 on "Shared File System Configuration File Entries" for the full description of its use and under what conditions you would want to change it.

Since `$(ARCH)` and `$(OPSYS)` will automatically be set to the correct values, we recommend that you do not overwrite them. Only do so if you know what you are doing.

3.3.2 Condor-wide Configuration File Entries

This section describes settings which affect all parts of the Condor system.

CONDOR_HOST This macro is used to define the $(NEGOTIATOR_HOST) and $(COLLECTOR_HOST) macros. Normally the *condor_collector* and *condor_negotiator* would run on the same machine. If for some reason they were not run on the same machine, $(CONDOR_HOST) would not be needed. Some of the host-based security macros use $(CONDOR_HOST) by default. See section 3.7.5, Setting up IP/host-based security in Condor for details.

COLLECTOR_HOST The hostname of the machine where the *condor_collector* is running for your pool. Normally it is defined with the $(CONDOR_HOST) macro described above.

NEGOTIATOR_HOST The hostname of the machine where the *condor_negotiator* is running for your pool. Normally it is defined with the $(CONDOR_HOST) macro described above.

RELEASE_DIR The full path to the Condor release directory, which holds the bin, etc, lib, and sbin directories. Other macros are defined relative to this one.

BIN This directory points to the Condor directory where user-level programs are installed. It is usually defined relative to the $(RELEASE_DIR) macro.

LIB This directory points to the Condor directory where libraries used to link jobs for Condor's standard universe are stored. The *condor_compile* program uses this macro to find these libraries, so it must be defined. $(LIB) is usually defined relative to the $(RELEASE_DIR) macro.

SBIN This directory points to the Condor directory where Condor's system binaries (such as the binaries for the Condor daemons) and administrative tools are installed. Whatever directory $(SBIN) points to ought to be in the PATH of users acting as Condor administrators.

LOCAL_DIR The location of the local Condor directory on each machine in your pool. One common option is to use the condor user's home directory which may be specified with $(TILDE). For example:

```
LOCAL_DIR = $(tilde)
```

On machines with a shared file system, where either the $(TILDE) directory or another directory you want to use is shared among all machines in your pool, you might use the $(HOSTNAME) macro and have a directory with many subdirectories, one for each machine in your pool, each named by hostnames. For example:

```
LOCAL_DIR = $(tilde)/hosts/$(hostname)
```

or:

```
LOCAL_DIR = $(release_dir)/hosts/$(hostname)
```

LOG Used to specify the directory where each Condor daemon writes its log files. The names of the log files themselves are defined with other macros, which use the `$(LOG)` macro by default. The log directory also acts as the current working directory of the Condor daemons as the run, so if one of them should produce a core file for any reason, it would be placed in the directory defined by this macro. Normally, `$(LOG)` is defined in terms of `$(LOCAL_DIR)`.

SPOOL The spool directory is where certain files used by the *condor_schedd* are stored, such as the job queue file and the initial executables of any jobs that have been submitted. In addition, for systems not using a checkpoint server, all the checkpoint files from jobs that have been submitted from a given machine will be store in that machine's spool directory. Therefore, you will want to ensure that the spool directory is located on a partition with enough disk space. If a given machine is only set up to execute Condor jobs and not submit them, it would not need a spool directory (or this macro defined). Normally, `$(SPOOL)` is defined in terms of `$(LOCAL_DIR)`.

EXECUTE This directory acts as the current working directory of any Condor job that is executing on the local machine. If a given machine is only set up to only submit jobs and not execute them, it would not need an execute directory (or this macro defined). Normally, `$(EXECUTE)` is defined in terms of `$(LOCAL_DIR)`.

LOCAL_CONFIG_FILE The location of the local, machine-specific configuration file for each machine in your pool. The two most common options would be putting this file in the `$(LOCAL_DIR)`, or putting all local configuration files for your pool in a shared directory, each one named by hostname. For example,

```
LOCAL_CONFIG_FILE = $(LOCAL_DIR)/condor_config.local
```

or,

```
LOCAL_CONFIG_FILE = $(release_dir)/etc/$(hostname).local
```

or, not using your release directory

```
LOCAL_CONFIG_FILE = /full/path/to/configs/$(hostname).local
```

Beginning with Condor version 6.0.1, the `$(LOCAL_CONFIG_FILE)` is treated as a list of files, not a single file. You can use either a comma or space separated list of files as its value. This allows you to specify multiple files as the local configuration file and each one will be processed in the order given (with parameters set in later files overriding values from previous files). This allows you to use one global configuration file for multiple platforms in your pool, define a platform-specific configuration file for each platform, and use a local configuration file for each machine. For more information on this, see section 3.10.2 about Configuring Condor for Multiple Platforms on page 180.

CONDOR_ADMIN The email address that Condor will send mail to if something goes wrong in your pool. For example, if a daemon crashes, the *condor_master* can send an *obituary* to this address with the last few lines of that daemon's log file and a brief message that describes what signal or exit status that daemon exited with.

CONDOR_SUPPORT_EMAIL The email address to be included at the bottom of all email Condor sends out under the label "Email address of the local Condor administrator:". This is the address where Condor users at your site should send their questions about Condor and get technical support. If this setting is not defined, Condor will use the address specified in CONDOR_ADMIN (described above).

MAIL The full path to a mail sending program that uses **-s** to specify a subject for the message. On all platforms, the default shipped with Condor should work. Only if you installed things in a non-standard location on your system would you need to change this setting.

RESERVED_SWAP Determines how much swap space you want to reserve for your own machine. Condor will not start up more *condor_shadow* processes if the amount of free swap space on your machine falls below this level.

RESERVED_DISK Determines how much disk space you want to reserve for your own machine. When Condor is reporting the amount of free disk space in a given partition on your machine, it will always subtract this amount. An example is the *condor_startd*, which advertises the amount of free space in the $(EXECUTE) directory.

LOCK Condor needs to create lock files to synchronize access to various log files. Because of problems with network file systems and file locking over the years, we *highly* recommend that you put these lock files on a local partition on each machine. If you do not have your $(LOCAL_DIR) on a local partition, be sure to change this entry. Whatever user or group Condor is running as needs to have write access to this directory. If you are not running as root, this is whatever user you started up the *condor_master* as. If you are running as root, and there is a condor account, it is most likely condor. Otherwise, it is whatever you set in the CONDOR_IDS environment variable. See section 3.7.1 on UIDs in Condor for details.

HISTORY Defines the location of the Condor history file, which stores information about all Condor jobs that have completed on a given machine. This macro is used by both the *condor_schedd* which appends the information and *condor_history*, the user-level program used to view the history file.

DEFAULT_DOMAIN_NAME If you do not use a fully qualified name in file /etc/hosts (or NIS, etc.) for either your official hostname or as an alias, Condor would not normally be able to use fully qualified names in places that it wants to. You can set this macro to the domain to be appended to your hostname, if changing your host information is not a good option. This macro must be set in the global configuration file (not the $(LOCAL_CONFIG_FILE). The reason for this is that the special $(FULL_HOSTNAME) macro is used by the configuration file code in Condor needs to know the full hostname. So, for $(DEFAULT_DOMAIN_NAME) to take effect, Condor must already have read in its value. However, Condor must set the $(FULL_HOSTNAME) special macro since you might use that to define where your local configuration file is. After reading the global configuration file, Condor figures out the right values for $(HOSTNAME) and $(FULL_HOSTNAME) and inserts them into its configuration table.

NETWORK_INTERFACE For systems with multiple network interfaces, Condor chooses the first one defined. To choose a network interface other than the first one, this macro is defined by giving the IP address to use.

CM_IP_ADDR If neither COLLECTOR_HOST nor COLLECTOR_IP_ADDR macros are defined, then this macro will be used to determine the IP address of the central manager (collector daemon). This macro is defined by an IP address.

HIGHPORT Specifies an upper limit of given port numbers for Condor to use, such that Condor is restricted to a range of port numbers. If this macro is not explicitly specified, then Condor will not restrict the port numbers that it uses. Condor will use system-assigned port numbers. For this macro to work, both HIGHPORT and LOWPORT (given below) must be defined.

LOWPORT Specifies a lower limit of given port numbers for Condor to use, such that Condor is restricted to a range of port numbers. If this macro is not explicitly specified, then Condor will not restrict the port numbers that it uses. Condor will use system-assigned port numbers. For this macro to work, both HIGHPORT (given above) and LOWPORT must be defined.

EMAIL_DOMAIN By default, if a user does not specify notify_user in the submit description file, any email Condor sends about that job will go to "username@UID_DOMAIN". If your machines all share a common UID domain (so that you would set UID_DOMAIN to be the same across all machines in your pool), but email to user@UID_DOMAIN is not the right place for Condor to send email for your site, you can define the default domain to use for email. A common example would be to set EMAIL_DOMAIN to the fully qualified hostname of each machine in your pool, so users submitting jobs from a specific machine would get email sent to user@machine.your.domain, instead of user@your.domain. You would do this by setting EMAIL_DOMAIN to $(FULL_HOSTNAME). In general, you should leave this setting commented out unless two things are true: 1) UID_DOMAIN is set to your domain, not $(FULL_HOSTNAME), and 2) email to user@UID_DOMAIN will not work.

CREATE_CORE_FILES Defines whether or not Condor daemons are to create a core file if something really bad happens. It is used to set the resource limit for the size of a core file. If not defined, it leaves in place whatever limit was in effect when you started the Condor daemons (normally the *condor_master*). If this parameter is set and TRUE, the limit is increased to the maximum. If it is set to FALSE, the limit is set at 0 (which means that no core files are created). Core files greatly help the Condor developers debug any problems you might be having. By using the parameter, you do not have to worry about tracking down where in your boot scripts you need to set the core limit before starting Condor. You set the parameter to whatever behavior you want Condor to enforce. This parameter has no default value, and is commented out in the default configuration file.

3.3.3 Daemon Logging Config File Entries

These entries control how and where the Condor daemons write their log files. Each of the entries in this section represents multiple macros. There is one for each subsystem (listed in section 3.3.1). The macro name for each substitutes SUBSYS with the name of the subsystem corresponding to the daemon.

SUBSYS_LOG The name of the log file for a given subsystem. For example, $(STARTD_LOG) gives the location of the log file for *condor_startd*. The name is defined relative to the

$(LOG) macro described above. The actual names of the files are also used in the $(VALID_LOG_FILES) entry used by *condor_preen*. A change to one of the file names with this setting requires a change to the $(VALID_LOG_FILES) entry as well, or *condor_preen* will delete your newly named log files.

MAX_SUBSYS_LOG Controls the maximum length in bytes to which a log will be allowed to grow. Each log file will grow to the specified length, then be saved to a file with the suffix .old. The .old files are overwritten each time the log is saved, thus the maximum space devoted to logging for any one program will be twice the maximum length of its log file. A value of 0 specifies that the file may grow without bounds. The default is 64 Kbytes.

TRUNC_SUBSYS_LOG_ON_OPEN If this macro is defined and set to TRUE, the affected log will be truncated and started from an empty file with each invocation of the program. Otherwise, new invocations of the program will append to the previous log file. By default this setting is FALSE for all daemons.

SUBSYS_LOCK This macro specifies the lock file used to synchronize append operations to the log file for this subsystem. It must be a separate file from the $(SUBSYS_LOG) file, since the $(SUBSYS_LOG) file may be rotated and you want to be able to synchronize access across log file rotations. A lock file is only required for log files which are accessed by more than one process. Currently, this includes only the SHADOW subsystem. This macro is defined relative to the $(LOCK) macro. If, for some strange reason, you decide to change this setting, be sure to change the $(VALID_LOG_FILES) entry that *condor_preen* uses as well.

SUBSYS_DEBUG All of the Condor daemons can produce different levels of output depending on how much information you want to see. The various levels of verbosity for a given daemon are determined by this macro. All daemons have the default level D_ALWAYS, and log messages for that level will be printed to the daemon's log, regardless of this macro's setting. The other possible debug levels are:

D_FULLDEBUG This level provides very verbose output in the log files. Only exceptionally frequent log messages for very specific debugging purposes would be excluded. In those cases, the messages would be viewed by having that another flag and D_FULLDEBUG both listed in the configuration file.

D_DAEMONCORE Provides log file entries specific to DaemonCore, such as timers the daemons have set and the commands that are registered. If both D_FULLDEBUG and D_DAEMONCORE are set, expect *very* verbose output.

D_PRIV This flag provides log messages about the *privilege state* switching that the daemons do. See section 3.7.1 on UIDs in Condor for details.

D_COMMAND With this flag set, any daemon that uses DaemonCore will print out a log message whenever a command comes in. The name and integer of the command, whether the command was sent via UDP or TCP, and where the command was sent from are all logged. Because the messages about the command used by *condor_kbdd* to communicate with the *condor_startd* whenever there is activity on the X server, and the command used for keep-alives are both only printed with D_FULLDEBUG enabled, it is best if this setting is used for all daemons.

D_LOAD The *condor_startd* keeps track of the load average on the machine where it is running. Both the general system load average, and the load average being generated by Condor's activity there are determined. With this flag set, the *condor_startd* will log a message with the current state of both of these load averages whenever it computes them. This flag only affects the *condor_startd*.

D_KEYBOARD With this flag set, the *condor_startd* will print out a log message with the current values for remote and local keyboard idle time. This flag affects only the *condor_startd*.

D_JOB When this flag is set, the *condor_startd* will send to its log file the contents of any job ClassAd that the *condor_schedd* sends to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.

D_MACHINE When this flag is set, the *condor_startd* will send to its log file the contents of its resource ClassAd when the *condor_schedd* tries to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.

D_SYSCALLS This flag is used to make the *condor_shadow* log remote syscall requests and return values. This can help track down problems a user is having with a particular job by providing the system calls the job is performing. If any are failing, the reason for the failure is given. The *condor_schedd* also uses this flag for the server portion of the queue management code. With `D_SYSCALLS` defined in `SCHEDD_DEBUG` there will be verbose logging of all queue management operations the *condor_schedd* performs.

D_MATCH When this flag is set, the negotiator logs a message for every match.

D_NETWORK When this flag is set, all Condor daemons will log a message on every TCP accept, connect, and close, and on every UDP send and receive. This flag is not yet fully supported in the *condor_shadow*.

ALL_DEBUG To make all subsystems share a debug flag, simply set the parameter `ALL_DEBUG` instead of changing all of the individual parameters. For example, to turn on network debugging in all subsystems, set `ALL_DEBUG = D_NETWORK`.

Log files may optionally be specified per debug level as follows:

SUBSYS_LEVEL_LOG This is the name of a log file for messages at a specific debug level for a specific subsystem. If the debug level is included in `$(SUBSYS_DEBUG)`, then all messages of this debug level will be written both to the `$(SUBSYS_LOG)` file and the `$(SUBSYS_LEVEL_LOG)` file. For example, `$(SHADOW_SYSCALLS_LOG)` specifies a log file for all remote system call debug messages.

MAX_SUBSYS_LEVEL_LOG Similar to `MAX_SUBSYS_LOG` .

TRUNC_SUBSYS_LEVEL_LOG_ON_OPEN Similar to `TRUNC_SUBSYS_LOG_ON_OPEN` .

3.3.4 DaemonCore Config File Entries

Please read section 3.8 for details on DaemonCore. There are certain configuration file settings that DaemonCore uses which affect all Condor daemons (except the checkpoint server, shadow, and

starter, none of which use DaemonCore yet).

HOSTALLOW... All macros that begin with either HOSTALLOW or HOSTDENY are settings for Condor's host-based security. See section 3.7.5 on Setting up IP/host-based security in Condor for details on these macros and how to configure them.

SETTABLE_ATTRS... All macros that begin with SETTABLE_ATTRS or SUBSYS_SETTABLE_ATTRS are settings used to restrict the configuration values that can be changed using the *condor_config_val* command. Section 3.7.5 on Setting up IP/Host-Based Security in Condor for details on these macros and how to configure them. In particular, section **??** on page **??** contains details specific to these macros.

SHUTDOWN_GRACEFUL_TIMEOUT Determines how long Condor will allow daemons try their graceful shutdown methods before they do a hard shutdown. It is defined in terms of seconds. The default is 1800 (30 minutes).

AUTHENTICATION_METHODS There are many instances when the Condor system needs to authenticate the identity of the user. For instance, when a job is submitted with *condor_submit*, Condor needs to authenticate the user so that the job goes into the queue and runs with the proper credentials. The AUTHENTICATION_METHODS parameter should be a list of permitted authentication methods. The list should be ordered by preference. The actual authentication method used is the first method in this list that both the server and client are able to perform. Possible values are:

- NTSSPI Use NT's standard LAN-MANAGER challenge-reponse protocol. NOTE: This is the default method used on Windows NT.
- FS Use the filesystem to authenticate the user. The server requests the client to create a specified temporary file, then the server verifies the ownership of that file. NOTE: This is the default method used on Unix systems.
- FS_REMOTE Use a shared filesystem to authenticate the user. This is useful for submitting jobs to a remote schedd. Similar to FS authentication, except the temporary file to be created by the user must be on a shared filesystem (AFS, NFS, etc.) If the client's submit description file does not define the command **rendezvousdir**, the **initialdir** value is used as the default directory in which to create the temporary file. NOTE: Normal AFS issues apply here: Condor must be able to write to the directory used.
- GSS Use Generic Security Services, which is implemented in Condor with X.509 certificates. See section 3.7.4. These X.509 certificates are compatible with the Globus system from Argonne National Labs.
- CLAIMTOBE The server should simply trust the client. NOTE: You had better trust all users who have access to your Condor pool if you enable CLAIMTOBE authentication.

SHUTDOWN_GRACEFUL_TIMEOUT This entry determines how long you are willing to let daemons try their graceful shutdown methods

SUBSYS_ADDRESS_FILE Every Condor daemon that uses DaemonCore has a command port where commands are sent. The IP/port of the daemon is put in that daemon's ClassAd so that other machines in the pool can query the *condor_collector* (which listens on a well-known

port) to find the address of a given daemon on a given machine. However, tools and daemons executing on the same machine they wish to communicate with are not required to query the collector. They look in a file on the local disk to find the IP/port. Setting this macro will cause daemons to write the IP/port of their command socket to a specified file. In this way, local tools will continue to operate, even if the machine running the *condor_collector* crashes. Using this file will also generate slightly less network traffic in your pool (since *condor_q*, *condor_rm*, and others do not have to send any messages over the network to locate the *condor_schedd*). This macro is not needed for the collector or negotiator, since their command sockets are at well-known ports.

SUBSYS_EXPRS Allows any DaemonCore daemon to advertise arbitrary expressions from the configuration file in its ClassAd. Give the comma-separated list of entries from the configuration file you want in the given daemon's ClassAd.

NOTE: The *condor_negotiator* and *condor_kbdd* do not send ClassAds now, so this entry does not affect them. The *condor_startd*, *condor_schedd*, *condor_master*, and *condor_collector* do send ClassAds, so those would be valid subsystems to set this entry for.

Setting `$(SUBMIT_EXPRS)` has the slightly different effect of having the named expressions inserted into all the job ClassAds that *condor_submit* creates. This is equivalent to the "+" syntax in submit files. See the the *condor_submit* manual page on page 395 for details.

Because of the different syntax of the configuration file and ClassAds, a little extra work is required to get a given entry into a ClassAd. In particular, ClassAds require quote marks (") around strings. Numeric values and boolean expressions can go in directly. For example, if the startd is to advertise a string macro, a numeric macro, and a boolean expression, do something similar to:

```
STRING = This is a string
NUMBER = 666
BOOL1 = True
BOOL2 = CurrentTime >= $(NUMBER_MACRO) || $(BOOL1)
MY_STRING = "$(STRING_MACRO)"
STARTD_EXPRS = MY_STRING, NUMBER, BOOL1, BOOL2
```

3.3.5 Shared File System Configuration File Macros

These macros control how Condor interacts with various shared and network filesystems. If you are using AFS as your shared filesystem, be sure to read section 3.10.1 on Using Condor with AFS.

UID_DOMAIN The `UID_DOMAIN` macro is used to decide under which user to run your jobs. If the `$(UID_DOMAIN)` on the submitting machine is different than the `$(UID_DOMAIN)` on the machine that runs your job, then Condor will run the job as the user called "nobody". For example, if the submit machine has the `$(UID_DOMAIN)` "flippy.cs.wisc.edu" and the machine where the job will execute has the `$(UID_DOMAIN)` "cs.wisc.edu", the job will run as user nobody, because the two `$(UID_DOMAIN)`s are not the same. If the `$(UID_DOMAIN)`

is the same on both the submit and execute machines, then Condor will run the job as the user that submitted the job.

A further check attempts to assure that the submitting machine can not lie about its `$(UID_DOMAIN)`. Condor compares the submit machine's claimed `$(UID_DOMAIN)` to its fully qualified name. If the two do not end the same, then the submit machine is presumed to be lying about its `$(UID_DOMAIN)`. In this case, Condor will run the job as user nobody. For example, a job submission to the Condor pool at the UW Madison from "flippy.example.com", claiming a `$(UID_DOMAIN)` of "cs.wisc.edu", will run the job as the user nobody.

Because of this verification, you need to set your `$(UID_DOMAIN)` to a real domain name. At the Computer Sciences department at the UW Madison, we set the `$(UID_DOMAIN)` to be "cs.wisc.edu" to indicate that whenever someone submits from a department machine, we will run the job as the user who submits it.

Also see `SOFT_UID_DOMAIN` below for information about one more check that Condor performs before running a job as a given user.

A few details:

You could set `$(UID_DOMAIN)` to "*". This will match all domains, but it is a gaping security hole. It is not recommended.

You can set `$(UID_DOMAIN)` to "none" or leave it undefined. This will force Condor to always run jobs as user nobody. Running standard universe jobs as user nobody enhances your security and should cause no problems, because the jobs use remote I/O to access all of their files. However, if vanilla jobs are run as user nobody, then files that need to be accessed by the job will need to be marked as world readable/writable so the user nobody can access them.

When Condor sends e-mail about a job, Condor sends the e-mail to `user@UID_DOMAIN`. If `$(UID_DOMAIN)` is set to "none" or it is undefined, the e-mail is sent to `user@submitmachinename`.

SOFT_UID_DOMAIN When Condor is about to run a job as a particular user (instead of the user nobody), it verifies that the UID given for the user is in the password file and actually matches the given user name. However, some installations may not have every user in every machine's password file, so this check will fail. If you prefer that Condor not do this check, because users are not in every password file, set this attribute to True.

FILESYSTEM_DOMAIN The `FILESYSTEM_DOMAIN` macro is an arbitrary string that is used to decide if two machines (a submitting machine and an execute machine) share a file system. Although the macro name contains the word "DOMAIN", the macro is not required to be a domain name, although it often is.

Vanilla Unix jobs currently require a shared file system in order to share any data files or see the output of the program. Condor decides if there is a shared filesystem by comparing the values of `$(FILESYSTEM_DOMAIN)` of both the submitting and execute machines. If the values are the same, Condor assume there is a shared file system. Condor implements the check by extending the Requirements for your job. You can see these requirements by using the [*-v*] argument to *condor_submit*.

Note that this implementation is not ideal: machines may share some file systems but not others. Condor currently has no way to express this automatically. You can express the need to use a particular file system by adding additional attributes to your machines and submit files, similar to the example given in Frequently Asked Questions, section 7 on how to run jobs only on machines that have certain software installations.

Note that if you do not set `$(FILESYSTEM_DOMAIN)`, Condor will automatically set the macro's value to be the fully qualified hostname of the local machine. Since each machine will have a different `$(FILESYSTEM_DOMAIN)`, they will not be considered to have shared file systems.

HAS_AFS Set this macro to TRUE if all the machines you plan on adding in your pool can all access a common set of AFS fileservers. Otherwise, set it to FALSE.

RESERVE_AFS_CACHE If your machine is running AFS and the AFS cache lives on the same partition as the other Condor directories, and you want Condor to reserve the space that your AFS cache is configured to use, set this macro to TRUE. It defaults to FALSE.

USE_NFS This macro influences how Condor jobs running in the standard universe access their files. Condor will redirect the file I/O requests of standard universe jobs to be executed on the machine which submitted the job. Because of this, as a Condor job migrates around the network, the file system always appears to be identical to the file system where the job was submitted. However, consider the case where a user's data files are sitting on an NFS server. The machine running the user's program will send all I/O over the network to the machine which submitted the job, which in turn sends all the I/O over the network a second time back to the NFS file server. Thus, all of the program's I/O is being sent over the network twice.

If this macro to TRUE, then Condor will attempt to read/write files without redirecting I/O back to the submitting machine if both the submitting machine and the machine running the job are both accessing the same NFS servers (*if* they are both in the same `$(FILESYSTEM_DOMAIN)` and in the same `$(UID_DOMAIN)`, as described above). The result is I/O performed by Condor standard universe jobs is only sent over the network once. While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth as at a very high premium, practical experience reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting `$(USE_NFS)` to FALSE is always safe. It may result in slightly more network traffic, but Condor jobs are most often heavy on CPU and light on I/O. It also ensures that a remote standard universe Condor job will always use Condor's remote system calls mechanism to reroute I/O and therefore see the exact same file system that the user sees on the machine where she/he submitted the job.

Some gritty details for folks who want to know: If the you set `$(USE_NFS)` to TRUE, and the `$(FILESYSTEM_DOMAIN)` of both the submitting machine and the remote machine about to execute the job match, and the `$(FILESYSTEM_DOMAIN)` claimed by the submit machine is indeed found to be a subset of what an inverse lookup to a DNS (domain name server) reports as the fully qualified domain name for the submit machine's IP address (this security measure safeguards against the submit machine from lying), *then* the job will access files using a local system call, without redirecting them to the submitting machine (with NFS). Otherwise, the

system call will get routed back to the submitting machine using Condor's remote system call mechanism. NOTE: When submitting a vanilla job, *condor_submit* will, by default, append requirements to the Job ClassAd that specify the machine to run the job must be in the same `$(FILESYSTEM_DOMAIN)` and the same `$(UID_DOMAIN)`.

USE_AFS If your machines have AFS, this macro determines whether Condor will use remote system calls for standard universe jobs to send I/O requests to the submit machine, or if it should use local file access on the execute machine (which will then use AFS to get to the submitter's files). Read the setting above on `$(USE_NFS)` for a discussion of why you might want to use AFS access instead of remote system calls.

One important difference between `$(USE_NFS)` and `$(USE_AFS)` is the AFS cache. With `$(USE_AFS)` set to TRUE, the remote Condor job executing on some machine will start modifying the AFS cache, possibly evicting the machine owner's files from the cache to make room for its own. Generally speaking, since we try to minimize the impact of having a Condor job run on a given machine, we do not recommend using this setting.

While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth as at a very high premium, practical experience reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting `$(USE_AFS)` to FALSE is always safe. It may result in slightly more network traffic, but Condor jobs are usually heavy on CPU and light on I/O. FALSE ensures that a remote standard universe Condor job will always see the exact same file system that the user on sees on the machine where he/she submitted the job. Plus, it will ensure that the machine where the job executes does not have its AFS cache modified as a result of the Condor job being there.

However, things may be different at your site, which is why the setting is there.

3.3.6 Checkpoint Server Configuration File Macros

These macros control whether or not Condor uses a checkpoint server. If you are using a checkpoint server, this section describes the settings that the checkpoint server itself needs defined. A checkpoint server is installed separately. It is not included in the main Condor binary distribution or installation procedure. See section 3.10.5 on Installing a Checkpoint Server for details on installing and running a checkpoint server for your pool.

NOTE: If you are setting up a machine to join the UW-Madison CS Department Condor pool, you *should* configure the machine to use a checkpoint server, and use "condor-ckpt.cs.wisc.edu" as the checkpoint server host (see below).

CKPT_SERVER_HOST The hostname of a checkpoint server.

STARTER_CHOOSES_CKPT_SERVER If this parameter is TRUE or undefined on the submit machine, the checkpoint server specified by `$(CKPT_SERVER_HOST)` on the execute machine is used. If it is FALSE on the submit machine, the checkpoint server specified by `$(CKPT_SERVER_HOST)` on the submit machine is used.

CKPT_SERVER_DIR The checkpoint server needs this macro defined to the full path of the directory the server should use to store checkpoint files. Depending on the size of your pool and the size of the jobs your users are submitting, this directory (and its subdirectories) might need to store many Mbytes of data.

USE_CKPT_SERVER A boolean which determines if you want a given submit machine to use a checkpoint server if one is available. If a checkpoint server isn't available or `USE_CKPT_SERVER` is set to False, checkpoints will be written to the local `$(SPOOL)` directory on the submission machine.

MAX_DISCARDED_RUN_TIME If the shadow is unable to read a checkpoint file from the checkpoint server, it keeps trying only if the job has accumulated more than this many seconds of CPU usage. Otherwise, the job is started from scratch. Defaults to 3600 (1 hour). This setting is only used if `$(USE_CKPT_SERVER)` is TRUE.

3.3.7 condor_master Configuration File Macros

These macros control the *condor_master*.

DAEMON_LIST This macro determines what daemons the *condor_master* will start and keep its watchful eyes on. The list is a comma or space separated list of subsystem names (listed in section 3.3.1). For example,

```
DAEMON_LIST = MASTER, STARTD, SCHEDD
```

NOTE: On your central manager, your `$(DAEMON_LIST)` will be different from your regular pool, since it will include entries for the *condor_collector* and *condor_negotiator*.

NOTE: On machines running Digital Unix or IRIX, your `$(DAEMON_LIST)` will also include KBDD, for the *condor_kbdd*, which is a special daemon that runs to monitor keyboard and mouse activity on the console. It is only with this special daemon that we can acquire this information on those platforms.

DC_DAEMON_LIST This macro lists the daemons in `DAEMON_LIST` which use the Condor DaemonCore library. The *condor_master* must differentiate between daemons that use DaemonCore and those that don't so it uses the appropriate inter-process communication mechanisms. This list currently includes all Condor daemons except the checkpoint server by default.

SUBSYS Once you have defined which subsystems you want the *condor_master* to start, you must provide it with the full path to each of these binaries. For example:

```
MASTER          = $(SBIN)/condor_master
STARTD          = $(SBIN)/condor_startd
SCHEDD          = $(SBIN)/condor_schedd
```

These are most often defined relative to the `$(SBIN)` macro.

DAEMONNAME_ENVIRONMENT For each subsystem defined in DAEMON_LIST , you may specify changes to the environment that daemon is started with by setting DAEMONNAME_ENVIRONMENT , where DAEMONNAME is the name of a daemon listed in DAEMON_LIST . It should be set to a semicolon delimited list of name=value pairs. For example, if you wish to redefine the TMP and CONDOR_CONFIG environment variables seen by the *condor_schedd*, you could place the following in the config file:

```
SCHEDD_ENVIRONMENT = TMP=/new/value;CONDOR_CONFIG=/special/config
```

When the *condor_schedd* was started by the *condor_master*, it would see the specified values of TMP and CONDOR_CONFIG.

SUBSYS_ARGS This macro allows the specification of additional command line arguments for any process spawned by the *condor_master*. List the desired arguments, as typing the command line into the configuration file. Set the arguments for a specific daemon with this macro, and the macro will affect only that daemon. Define one of these for each daemon the *condor_master* is controlling. For example, set $(STARTD_ARGS) to specify any extra command line arguments to the *condor_startd*.

PREEN In addition to the daemons defined in $(DAEMON_LIST), the *condor_master* also starts up a special process, *condor_preen* to clean out junk files that have been left laying around by Condor. This macro determines where the *condor_master* finds the *condor_preen* binary. Comment out this macro, and *condor_preen* will not run.

PREEN_ARGS Controls how *condor_preen* behaves by allowing the specification of command-line arguments. This macro works as $(SUBSYS_ARGS) does. The difference is that you must specify this macro for *condor_preen* if you want it to do anything. *condor_preen* takes action only because of command line arguments. **-m** means you want e-mail about files *condor_preen* finds that it thinks it should remove. **-r** means you want *condor_preen* to actually remove these files.

PREEN_INTERVAL This macro determines how often *condor_preen* should be started. It is defined in terms of seconds and defaults to 86400 (once a day).

PUBLISH_OBITUARIES When a daemon crashes, the *condor_master* can send e-mail to the address specified by $(CONDOR_ADMIN) with an obituary letting the administrator know that the daemon died, the cause of death (which signal or exit status it exited with), and (optionally) the last few entries from that daemon's log file. If you want obituaries, set this macro to TRUE.

OBITUARY_LOG_LENGTH This macro controls how many lines of the log file are part of obituaries.

START_MASTER If this setting is defined and set to FALSE when the *condor_master* starts up, the first thing it will do is exit. This appears strange, but perhaps you do not want Condor to run on certain machines in your pool, yet the boot scripts for your entire pool are handled by a centralized system that starts up the *condor_master* automatically. This is an entry you would most likely find in a local configuration file, not a global configuration file.

START_DAEMONS This macro is similar to the `$(START_MASTER)` macro described above. However, the *condor_master* does not exit; it does not start any of the daemons listed in the `$(DAEMON_LIST)`. The daemons may be started at a later time with a *condor_on* command.

MASTER_UPDATE_INTERVAL This macro determines how often the *condor_master* sends a ClassAd update to the *condor_collector*. It is defined in seconds and defaults to 300 (every 5 minutes).

MASTER_CHECK_NEW_EXEC_INTERVAL This macro controls how often the *condor_master* checks the timestamps of the running daemons. If any daemons have been modified, the master restarts them. It is defined in seconds and defaults to 300 (every 5 minutes).

MASTER_NEW_BINARY_DELAY Once the *condor_master* has discovered a new binary, this macro controls how long it waits before attempting to execute the new binary. This delay exists because the *condor_master* might notice a new binary while it is in the process of being copied, in which case trying to execute it yields unpredictable results. The entry is defined in seconds and defaults to 120 (2 minutes).

SHUTDOWN_FAST_TIMEOUT This macro determines the maximum amount of time daemons are given to perform their fast shutdown procedure before the *condor_master* kills them outright. It is defined in seconds and defaults to 300 (5 minutes).

MASTER_BACKOFF_FACTOR If a daemon keeps crashing, an exponential backoff waits longer and longer before restarting it. At the end of this section, there is an example that shows how all these settings work. This setting is the base of the exponent used to determine how long to wait before starting the daemon again. It defaults to 2 seconds.

MASTER_BACKOFF_CEILING This entry determines the maximum amount of time you want the master to wait between attempts to start a given daemon. (With 2.0 as the `$(MASTER_BACKOFF_FACTOR)`, 1 hour is obtained in 12 restarts). It is defined in terms of seconds and defaults to 3600 (1 hour).

MASTER_RECOVER_FACTOR A macro to set How long a daemon needs to run without crashing before it is considered *recovered*. Once a daemon has recovered, the number of restarts is reset, so the exponential backoff stuff returns to its initial state. The macro is defined in terms of seconds and defaults to 300 (5 minutes).

For clarity, the following is an example of the workings of the exponential backoff settings. The example is worked out assuming the default settings.

When a daemon crashes, it is restarted in 10 seconds. If it keeps crashing, a longer amount of time is waited before restarting. The length of time is based on how many times it has been restarted. Take the `$(MASTER_BACKOFF_FACTOR)` (defaults to 2) to the power the number of times the daemon has restarted, and add 9. An example:

```
1st crash:  restarts == 0, so, 9 + 2^0 = 9 + 1 = 10 seconds
2nd crash:  restarts == 1, so, 9 + 2^1 = 9 + 2 = 11 seconds
3rd crash:  restarts == 2, so, 9 + 2^2 = 9 + 4 = 13 seconds
```

```
...
6th crash:  restarts == 5, so, 9 + 2^5 = 9 + 32 = 41 seconds
...
9th crash:  restarts == 8, so, 9 + 2^8 = 9 + 256 = 265 seconds
```

After the 13th crash, it would be:

```
13th crash:  restarts == 12, so, 9 + 2^12 = 9 + 4096 = 4105 seconds
```

This is bigger than the $(MASTER_BACKOFF_CEILING), which defaults to 3600, so the daemon would really be restarted after only 3600 seconds, not 4105. The *condor_master* tries again every hour (since the numbers would get larger and would always be capped by the ceiling). Eventually, imagine that daemon finally started and did not crash. This might happen if, for example, an administrator reinstalled an accidentally deleted binary after receiving e-mail about the daemon crashing. If it stayed alive for $(MASTER_RECOVER_FACTOR) seconds (defaults to 5 minutes), the count of how many restarts this daemon has performed is reset to 10 seconds.

The moral of the example is that the defaults work quite well, and you probably will not want to change them for any reason.

MASTER_EXPRS This macro is described in section 3.3.4 as SUBSYS_EXPRS .

MASTER_DEBUG This macro is described in section 3.3.3 as SUBSYS_DEBUG .

MASTER_ADDRESS_FILE This macro is described in section 3.3.4 as SUBSYS_ADDRESS_FILE

SECONDARY_COLLECTOR_LIST This macro lists the hostnames of secondary collectors. A secondary collector is a machine running a *condor_collector* daemon that is not the central manager. A secondary collector makes it possible to execute administrative commands in the pool when the central manager is down by using the **-pool** argument to specify the name of a secondary collector to use to locate the *condor_master* daemon.

ALLOW_ADMIN_COMMANDS If set to NO for a given host, this macro disables administrative commands, such as *condor_restart*, *condor_on*, and *condor_off*, to that host.

3.3.8 condor_startd Configuration File Macros

NOTE: If you are running Condor on a multi-CPU machine, be sure to also read section 3.10.8 on page 192 which describes how to setup and configure Condor on SMP machines.

These settings control general operation of the *condor_startd*. Information on how to configure the *condor_startd* to start, suspend, resume, vacate and kill remote Condor jobs is found in section 3.6 on Configuring The Startd Policy. In that section is information on the startd's *states* and *activities*. Macros in the configuration file not described here are ones that control state or activity transitions within the *condor_startd*.

STARTER This macro holds the full path to the *condor_starter* binary that the startd should spawn. It is normally defined relative to `$(SBIN)`.

ALTERNATE_STARTER_1 This macro holds the full path to the *condor_starter.pvm* binary that the startd spawns to service PVM jobs. It is normally defined relative to `$(SBIN)`, since by default, *condor_starter.pvm* is installed in the regular Condor release directory.

POLLING_INTERVAL When a startd enters the claimed state, this macro determines how often the state of the machine is polled to check the need to suspend, resume, vacate or kill the job. It is defined in terms of seconds and defaults to 5.

UPDATE_INTERVAL Determines how often the startd should send a ClassAd update to the *condor_collector*. The startd also sends update on any state or activity change, or if the value of its `START` expression changes. See section 3.6.5 on condor_startd States, section 3.6.6 on condor_startd Activities, and section 3.6.3 on condor_startd `START` expression for details on states, activities, and the `START` expression. This macro is defined in terms of seconds and defaults to 300 (5 minutes).

STARTD_HAS_BAD_UTMP When the startd is computing the idle time of all the users of the machine (both local and remote), it checks the `utmp` file to find all the currently active ttys, and only checks access time of the devices associated with active logins. Unfortunately, on some systems, `utmp` is unreliable, and the startd might miss keyboard activity by doing this. So, if your `utmp` is unreliable, set this macro to TRUE and the startd will check the access time on all tty and pty devices.

CONSOLE_DEVICES This macro allows the startd to monitor console (keyboard and mouse) activity by checking the access times on special files in `/dev`. Activity on these files shows up as `ConsoleIdle` time in the startd's ClassAd. Give a comma-separated list of the names of devices considered the console, without the `/dev/` portion of the pathname. The defaults vary from platform to platform, and are usually correct.

One possible exception to this is on Linux, where we use "mouse" as one of the entries. Most Linux installations put in a soft link from `/dev/mouse` that points to the appropriate device (for example, `/dev/psaux` for a PS/2 bus mouse, or `/dev/tty00` for a serial mouse connected to com1). However, if your installation does not have this soft link, you will either need to put it in (you will be glad you did), or change this macro to point to the right device.

Unfortunately, there are no such devices on Digital Unix or IRIX (don't be fooled by `/dev/keyboard0`; the kernel does not update the access times on these devices), so this macro is not useful in these cases, and we must use the *condor_kbdd* to get this information by connecting to the X server.

STARTD_JOB_EXPRS When the machine is claimed by a remote user, the startd can also advertise arbitrary attributes from the job ClassAd in the machine ClassAd. List the attribute names to be advertised. NOTE: Since these are already ClassAd expressions, do not do anything unusual with strings.

STARTD_EXPRS This macro is described in section 3.3.4 as `SUBSYS_EXPRS`.

STARTD_DEBUG This macro (and other settings related to debug logging in the startd) is described in section 3.3.3 as `SUBSYS_DEBUG`.

STARTD_ADDRESS_FILE This macro is described in section 3.3.4 as SUBSYS_ADDRESS_FILE

NUM_CPUS This macro can be used to "lie" to the startd about how many CPUs your machine has. If you set this, it will override Condor's automatic computation of the number of CPUs in your machine, and Condor will use whatever integer you specify here. In this way, you can allow multiple Condor jobs to run on a single-CPU machine by having that machine treated like an SMP machine with multiple CPUs, which could have different Condor jobs running on each one. Or, you can have an SMP machine advertise more virtual machines than it has CPUs. However, using this parameter will hurt the performance of the jobs, since you would now have multiple jobs running on the same CPU, competing with each other. The option is only meant for people who specifically want this behavior and know what they are doing. It is disabled by default.

NOTE: This setting cannot be changed with a simple reconfig (either by sending a SIGHUP or using *condor_reconfig*. If you change this, you must restart the *condor_startd* for the change to take effect (by using "condor_restart -startd").

NOTE: If you use this setting on a given machine, you should probably advertise that fact in the machine's ClassAd by using the STARTD_EXPRS setting (described above). This way, jobs submitted in your pool could specify that they did or did not want to be matched with machines that were only really offering "fractional CPUs".

MEMORY Normally, Condor will automatically detect the amount of physical memory available on your machine. Define MEMORY to tell Condor how much physical memory (in MB) your machine has, overriding the value Condor computes automatically.

RESERVED_MEMORY How much memory would you like reserved from Condor? By default, Condor considers all the physical memory of your machine as available to be used by Condor jobs. If RESERVED_MEMORY is defined, Condor subtracts it from the amount of memory it advertises as available.

These macros only apply to the startd when it is running on an SMP machine. See section 3.10.8 on page 192 on Configuring The Startd for SMP Machines for details.

VIRTUAL_MACHINES_CONNECTED_TO_CONSOLE An integer which indicates how many of the virtual machines the startd is representing should be "connected" to the console (in other words, notice when there's console activity). This defaults to all virtual machines (N in a machine with N CPUs).

VIRTUAL_MACHINES_CONNECTED_TO_KEYBOARD An integer which indicates how many of the virtual machines the startd is representing should be "connected" to the keyboard (for remote tty activity, as well as console activity). Defaults to 1.

DISCONNECTED_KEYBOARD_IDLE_BOOST If there are virtual machines not connected to either the keyboard or the console, the corresponding idle time reported will be the time since the startd was spawned, plus the value of this macro. It defaults to 1200 seconds (20 minutes). We do this because if the virtual machine is configured not to care about keyboard activity,

we want it to be available to Condor jobs as soon as the startd starts up, instead of having to wait for 15 minutes or more (which is the default time a machine must be idle before Condor will start a job). If you do not want this boost, set the value to 0. If you change your START expression to require more than 15 minutes before a job starts, but you still want jobs to start right away on some of your SMP nodes, increase this macro's value.

The following settings control the number of virtual machines reported for a given SMP host, and what attributes each one has. They are only needed if you do not want to have an SMP machine report to Condor with a seperate virtual machine for each CPU, with all shared system resources evenly divided among them. Please read section 3.10.8 on page 193 for details on how to properly configure these settings to suit your needs.

NOTE: You can only change the number of each type of virtual machine the *condor_startd* is reporting with a simple reconfig (such as sending a SIGHUP signal, or using the *condor_reconfig* command). You cannot change the definition of the different virtual machine types with a reconfig. If you change them, you must restart the *condor_startd* for the change to take effect (for example, using "condor_restart -startd").

MAX_VIRTUAL_MACHINE_TYPES The maximum number of different virtual machine types. Note: this is the maximum number of different *types*, not of actual virtual machines. Defaults to 10. (You should only need to change this setting if you define more than 10 seperate virtual machine types, which would be pretty rare.)

VIRUAL_MACHINE_TYPE_<N> This setting defines a given virtual machine type, by specifying what part of each shared system resource (like RAM, swap space, etc) this kind of virtual machine gets. N can be any integer from 1 to the value of `$(MAX_VIRTUAL_MACHINE_TYPES)`, such as `VIRTUAL_MACHINE_TYPE_1`. The format of this entry can be somewhat complex, so please refer to section 3.10.8 on page 193 for details on the different possibilities.

NUM_VIRUAL_MACHINES_TYPE_<N> This macro controls how many of a given virtual machine type are actually reported to Condor. There is no default.

NUM_VIRUAL_MACHINES If your SMP machine is being evenly divided, and the virtual machine type settings described above are not being used, this macro controls how many virtual machines will be reported. The default is one virtual machine for each CPU. This setting can be used to reserve some CPUs on an SMP which would not be reported to the Condor pool.

The following macros control the optional computation of resource availability statistics in the startd.

STARTD_COMPUTE_AVAIL_STATS A boolean that determines if the startd computes resource availability statistics. The default is False.

STARTD_AVAIL_CONFIDENCE A floating point number that sets the confidence level of the startd's AvailTime estimate. By default, the estimate is based on the 80th percentile of past values (i.e., the macro is set to 0.8).

STARTD_MAX_AVAIL_PERIOD_SAMPLES An integer that limits the number of samples of past available intervals stored by the startd to limit memory and disk consumption. Each sample requires 4 bytes of memory and approximately 10 bytes of disk space.

If STARTD_COMPUTE_AVAIL_STATS = True, the startd will define the following ClassAd attributes for resources:

AvailTime What proportion of the time (between 0.0 and 1.0) has this resource been in a state other than "Owner"?

LastAvailInterval What was the duration (in seconds) of the last period between "Owner" states?

The following attributes will also be included if the resource is not in the "Owner" state:

AvailSince At what time did the resource last leave the "Owner" state? Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

AvailTimeEstimate Based on past history, this is an estimate of how long the current period between "Owner" states will last.

3.3.9 condor_schedd Configuration File Entries

These macros control the *condor_schedd*.

SHADOW This macro determines the full path of the *condor_shadow* binary that the *condor_schedd* spawns. It is normally defined in terms of $(SBIN).

SHADOW_PVM This macro determines the full path of the special *condor_shadow.pvm* binary used for supporting PVM jobs that the *condor_schedd* spawns. It is normally defined in terms of $(SBIN).

MAX_JOBS_RUNNING This macro controls the maximum number of *condor_shadow* processes a given *condor_schedd* is allowed to spawn. The actual number of *condor_shadow*s may be less if you have reached your $(RESERVED_SWAP) limit.

MAX_SHADOW_EXCEPTIONS This macro controls the maximum number of times that *condor_shadow* processes can have a fatal error (exception) before the *condor_schedd* will relinquish the match associated with the dying shadow. Defaults to 5.

SCHEDD_INTERVAL This macro determines how often the *condor_schedd* sends a ClassAd update to the *condor_collector*. It is defined in terms of seconds and defaults to 300 (every 5 minutes).

JOB_START_DELAY When the *condor_schedd* has finished negotiating and has many new machines that it has claimed, the *condor_schedd* can wait for a delay period before starting up a *condor_shadow* for each job it is going to run. The delay prevents a sudden, large load on the submit machine as it spawns many shadows simultaneously. It prevents having to deal with their startup activity all at once. This macro determines how how long the *condor_schedd* should wait in between spawning each *condor_shadow*. Similarly, this macro is also used during the graceful shutdown of the *condor_schedd*. During graceful shutdown, this macro determines how long to wait in between asking each *condor_shadow* to gracefully shutdown. Defined in terms of seconds and defaults to 2.

ALIVE_INTERVAL This macro determines how often the *condor_schedd* should send a keep alive message to any startd it has claimed. When the schedd claims a startd, it tells the startd how often it is going to send these messages. If the startd does not get one of these messages after 3 of these intervals has passed, the startd releases the claim, and the schedd is no longer paying for the resource (in terms of priority in the system). The macro is defined in terms of seconds and defaults to 300 (every 5 minutes).

SHADOW_SIZE_ESTIMATE This macro sets the estimated virtual memory size of each *condor_shadow* process. Specified in kilobytes. The default varies from platform to platform.

SHADOW_RENICE_INCREMENT When the schedd spawns a new *condor_shadow*, it can do so with a *nice-level*. A nice-level is a Unix mechanism that allows users to assign their own processes a lower priority so that the processes do not interfere with interactive use of the machine. This is very handy for keeping a submit machine with lots of shadows running still useful to the owner of the machine. The value can be any integer between 0 and 19, with a value of 19 being the lowest priority. It defaults to 10.

QUEUE_CLEAN_INTERVAL The schedd maintains the job queue on a given machine. It does so in a persistent way such that if the schedd crashes, it can recover a valid state of the job queue. The mechanism it uses is a transaction-based log file (the `job_queue.log` file, not the `SchedLog` file). This file contains an initial state of the job queue, and a series of transactions that were performed on the queue (such as new jobs submitted, jobs completing, and checkpointing). Periodically, the schedd will go through this log, truncate all the transactions and create a new file with containing only the new initial state of the log. This is a somewhat expensive operation, but it speeds up when the schedd restarts since there are fewer transactions it has to play to figure out what state the job queue is really in. This macro determines how often the schedd should rework this queue to cleaning it up. It is defined in terms of seconds and defaults to 86400 (once a day).

WALL_CLOCK_CKPT_INTERVAL The job queue contains a counter for each job's "wall clock" run time, i.e., how long each job has executed so far. This counter is displayed by *condor_q*. The counter is updated when the job is evicted or when the job completes. When the schedd crashes, the run time for jobs that are currently running will not be added to the counter (and so, the run time counter may become smaller than the cpu time counter). The schedd saves run time "checkpoints" periodically for running jobs so if the schedd crashes, only run time since the last checkpoint is lost. This macro controls how often the schedd saves run time checkpoints. It is defined in terms of seconds and defaults to 3600 (one hour). A value of 0 will disable wall clock checkpoints.

ALLOW_REMOTE_SUBMIT Starting with Condor Version 6.0, users can run *condor_submit* on one machine and actually submit jobs to another machine in the pool. This is called a *remote submit*. Jobs submitted in this way are entered into the job queue owned by the Unix user nobody. This macro determines whether this is allowed. It defaults to FALSE.

QUEUE_SUPER_USERS This macro determines what user names on a given machine have *super-user access* to the job queue, meaning that they can modify or delete the job ClassAds of other users. (Normally, you can only modify or delete ClassAds from the job queue that you own). Whatever user name corresponds with the UID that Condor is running as (usually the Unix user condor) will automatically be included in this list because that is needed for Condor's proper functioning. See section 3.7.1 on UIDs in Condor for more details on this. By default, we give root the ability to remove other user's jobs, in addition to user condor.

SCHEDD_LOCK This macro specifies what lock file should be used for access to the SchedLog file. It must be a separate file from the SchedLog, since the SchedLog may be rotated and synchronization across log file rotations is desired. This macro is defined relative to the $(LOCK) macro. If you decide to change this setting (not recommended), be sure to change the $(VALID_LOG_FILES) entry that *condor_preen* uses as well.

SCHEDD_EXPRS This macro is described in section 3.3.4 as SUBSYS_EXPRS.

SCHEDD_DEBUG This macro (and other settings related to debug logging in the schedd) is described in section 3.3.3 as SUBSYS_DEBUG.

SCHEDD_ADDRESS_FILE This macro is described in section 3.3.4 as SUBSYS_ADDRESS_FILE.

FLOCK_NEGOTIATOR_HOSTS This macro defines a list of negotiator hostnames (not including the local $(NEGOTIATOR_HOST) machine) for pools in which the schedd should attempt to run jobs. Hosts in the list should be in order of preference. The schedd will only send a request to a central manager in the list if the local pool and pools earlier in the list are not satisfying all the job requests. $(HOSTALLOW_NEGOTIATOR_SCHEDD) (see section 3.3.4) must also be configured to allow negotiators from all of the $(FLOCK_NEGOTIATOR_HOSTS) to contact the schedd. Please make sure the $(NEGOTIATOR_HOST) is first in the $(HOSTALLOW_NEGOTIATOR_SCHEDD) list. Similarly, the central managers of the remote pools must be configured to listen to requests from this schedd.

FLOCK_COLLECTOR_HOSTS This macro defines a list of collector hostnames for pools in which the schedd should attempt to run jobs. The collectors must be specified in order, corresponding to the $(FLOCK_NEGOTIATOR_HOSTS) list. In the typical case, where each pool has the collector and negotiator running on the same machine, $(FLOCK_COLLECTOR_HOSTS) should have the same definition as $(FLOCK_NEGOTIATOR_HOSTS).

FLOCK_VIEW_SERVERS This macro defines a list of hostnames where the condor-view server is running in the pools to which you want your jobs to flock. The order of this list must correspond to the order of the $(FLOCK_COLLECTOR_HOSTS) and $(FLOCK_NEGOTIATOR_HOSTS) lists. List items may be empty for pools which don't use a separate condor-view server. $(FLOCK_VIEW_SERVER) may be left undefined if no remote pools use separate condor-view servers. Note: It is required that the same hostname does not appear twice in the $(FLOCK_VIEW_SERVERS) list and that the $(CONDOR_VIEW_HOST) does not appear in the $(FLOCK_VIEW_SERVERS) list.

NEGOTIATE_ALL_JOBS_IN_CLUSTER If this macro is set to False (the default), when the schedd fails to start an idle job, it will not try to start any other idle jobs in the same cluster during that negotiation cycle. This makes negotiation much more efficient for large job clusters. However, in some cases other jobs in the cluster can be started even though an earlier job can't. For example, the jobs' requirements may differ, because of different disk space, memory, or operating system requirements. Or, machines may be willing to run only some jobs in the cluster, because their requirements reference the jobs' virtual memory size or other attribute. Setting this macro to True will force the schedd to try to start all idle jobs in each negotiation cycle. This will make negotiation cycles last longer, but it will ensure that all jobs that can be started will be started.

3.3.10 condor_shadow Configuration File Entries

These settings affect the *condor_shadow*.

SHADOW_LOCK This macro specifies the lock file to be used for access to the `ShadowLog` file. It must be a separate file from the `ShadowLog`, since the `ShadowLog` may be rotated and you want to synchronize access across log file rotations. This macro is defined relative to the `$(LOCK)` macro. If you decide to change this setting (not recommended), be sure to change the `$(VALID_LOG_FILES)` entry that *condor_preen* uses as well.

SHADOW_DEBUG This macro (and other settings related to debug logging in the shadow) is described in section 3.3.3 as `SUBSYS_DEBUG`.

COMPRESS_PERIODIC_CKPT This boolean macro specifies whether the shadow should instruct applications to compress periodic checkpoints (when possible). The default is FALSE.

COMPRESS_VACATE_CKPT This boolean macro specifies whether the shadow should instruct applications to compress vacate checkpoints (when possible). The default is FALSE.

PERIODIC_MEMORY_SYNC This boolean macro specifies whether the shadow should instruct applications to commit dirty memory pages to swap space during a periodic checkpoint. The default is FALSE. This potentially reduces the number of dirty memory pages at vacate time, thereby reducing swapping activity on the remote machine.

SLOW_CKPT_SPEED This macro specifies the speed at which vacate checkpoints should be written, in kilobytes per second. If zero (the default), vacate checkpoints are written as fast as possible. Writing vacate checkpoints slowly can avoid overwhelming the remote machine with swapping activity.

3.3.11 condor_shadow.pvm Configuration File Entries

These macros control the *condor_shadow.pvm*, the special shadow that supports PVM jobs inside Condor. See section 3.4.3 on Installing PVM Support in Condor for details. *condor_shadow* macros also apply to this special shadow. See section 3.3.10.

PVMD This macro holds the full path to the special *condor_pvmd*, the Condor PVM daemon. This daemon is installed in the regular Condor release directory by default, so the macro is usually defined in terms of `$(SBIN)`.

PVMGS This macro holds the full path to the special *condor_pvmgs*, the Condor PVM Group Server daemon, which is needed to support PVM groups. This daemon is installed in the regular Condor release directory by default, so the macro is usually defined in terms of `$(SBIN)`.

3.3.12 condor_starter Configuration File Entries

These settings affect the *condor_starter*.

EXEC_TRANSFER_ATTEMPTS Sometimes due to a router misconfiguration, kernel bug, or other Act of God network problem, the transfer of the initial checkpoint from the submit machine to the execute machine will fail midway through. This parameter allows a retry of the transfer a certain number of times that must be equal to or greater than 1. If this parameter is not specified, or specified incorrectly, then it will default to three. If the transfer of the initial executable fails every attempt, then the job goes back into the idle state until the next renegotiation cycle.

NOTE: : This parameter does not exist in the NT starter.

JOB_RENICE_INCREMENT When the starter spawns a Condor job, it can do so with a *nice-level*. A nice-level is a Unix mechanism that allows users to assign their own processes a lower priority so that the processes do not interfere with interactive use of the machine. If you have machines with lots of real memory and swap space so the only scarce resource is CPU time, you may use this macro in conjunction with a policy that always allowed Condor to start jobs on your machines so that Condor jobs would always run, but interactive response on your machines would never suffer. You most likely will not notice Condor is running jobs. See section 3.6 on Configuring The Startd Policy for more details on setting up a policy for starting and stopping jobs on a given machine. The entry can be any integer between 0 and 20, with a value of 19 being the lowest priority. It is commented out by default.

STARTER_LOCAL_LOGGING This macro determines whether the starter should do local logging to its own log file, or send debug information back to the *condor_shadow* where it will end up in the ShadowLog. It defaults to TRUE.

STARTER_DEBUG This setting (and other settings related to debug logging in the starter) is described above in section 3.3.3 as `$(SUBSYS_DEBUG)`.

USER_JOB_WRAPPER This macro allows the administrator to specify a "wrapper" script to handle the execution of all user jobs. If specified, Condor will never directly execute a job but instead will invoke the program specified by this macro. The command-line arguments passed to this program will include the full-path to the actual user job which should be executed, followed by all the command-line parameters to pass to the user job. This wrapper program must ultimately replace its image with the user job; in other words, it must *exec()* the user job, not *fork()* it. For instance, if the wrapper program is a Bourne/C/Korn shell script, the last line of execution should be:

```
exec $*
```

3.3.13 condor_submit Configuration File Entries

DEFAULT_UNIVERSE The universe under which a job is executed may be specified in the submit description file. If it is not specified in the submit description file, then this variable specifies the universe (when defined). If the universe is not specified in the submit description file, and if this variable is not defined, then the default universe for a job will be the standard universe.

If you want *condor_submit* to automatically append an expression to the `Requirements` expression or `Rank` expression of jobs at your site use the following macros:

APPEND_REQ_VANILLA Expression to be appended to vanilla job requirements.

APPEND_REQ_STANDARD Expression to be appended to standard job requirements.

APPEND_REQUREMENTS Expression to be appended to any type of universe jobs. However, if `APPEND_REQ_VANILLA` or `APPEND_REQ_STANDARD` is defined, then ignore the `APPEND_REQUREMENTS` for those universes.

APPEND_RANK_STANDARD Expression to be appended to standard job rank.

APPEND_RANK_VANILLA Expression to append to vanilla job rank.

NOTE: The `APPEND_RANK_STANDARD` and `APPEND_RANK_VANILLA` macros were called `APPEND_PREF_STANDARD` and `APPEND_PREF_VANILLA` in previous versions of Condor.

In addition, you may provide default `Rank` expressions if your users do nt specify their own with:

DEFAULT_RANK_VANILLA Default Rank for vanilla jobs.

DEFAULT_RANK_STANDARD Default Rank for standard jobs.

Both of these macros default to the jobs preferring machines where there is more main memory than the image size of the job, expressed as:

```
((Memory*1024) > Imagesize)
```

GLOBUSRUN This macro holds the full path to the *globusrun* program which is needed for submitting to the GLOBUS universe.

SHADOW_GLOBUS This macro holds the full path to the *condor_shadow.globus* program which is needed for submitting to the GLOBUS universe.

DEFAULT_IO_BUFFER_SIZE Condor keeps a buffer of recently-used data for each file an application opens. This macro specifies the default maximum number of bytes to be buffered for each open file at the executing machine. The *condor_status* `buffer_size` command will override this default. If this macro is undefined, a default size of 512 KB will be used.

DEFAULT_IO_BUFFER_BLOCK_SIZE When buffering is enabled, Condor will attempt to consolidate small read and write operations into large blocks. This macro specifies the default block size Condor will use. The *condor_status* `buffer_block_size` command will override this default. If this macro is undefined, a default size of 32 KB will be used.

SUBMIT_SKIP_FILECHECK If True, *condor_submit* behaves as if the **-d** command-line option is used. This tells *condor_submit* to disable file permission checks when submitting a job. This can significantly decrease the amount of time required to submit a large group of jobs. The default value is False.

3.3.14 condor_preen Configuration File Entries

These macros affect *condor_preen*.

PREEN_ADMIN This macro sets the e-mail address where *condor_preen* will send e-mail (if it is configured to send email at all... see the entry for `PREEN`). Defaults to `$(CONDOR_ADMIN)`.

VALID_SPOOL_FILES This macro contains a (comma or space separated) list of files that *condor_preen* considers valid files to find in the `$(SPOOL)` directory. Defaults to all the files that are valid. A change to the `$(HISTORY)` macro requires a change to this macro as well.

VALID_LOG_FILES This macro contains a (comma or space separated) list of files that *condor_preen* considers valid files to find in the `$(LOG)` directory. Defaults to all the files that are valid. A change to the names of any of the log files above requires a change to this macro as well. In addition, the defaults for the `$(SUBSYS_ADDRESS_FILE)` are listed here, so a change to those requires a change this entry as well.

3.3.15 condor_collector Configuration File Entries

These macros affect the *condor_collector*.

CLASSAD_LIFETIME This macro determines how long a ClassAd can remain in the collector before it is discarded as stale information. The ClassAds sent to the collector might also have an attribute that says how long the lifetime should be for that specific ad. If that attribute is present, the collector will either use it or the `$(CLASSAD_LIFETIME)`, whichever is greater. The macro is defined in terms of seconds, and defaults to 900 (15 minutes).

MASTER_CHECK_INTERVAL This macro defines how often the collector should check for machines that have ClassAds from some daemons, but not from the *condor_master* (*orphaned daemons*) and send e-mail about it. It is defined in seconds and defaults to 10800 (3 hours).

CLIENT_TIMEOUT Network timeout when talking to daemons that are sending an update. It is defined in seconds and defaults to 30.

QUERY_TIMEOUT Network timeout when talking to anyone doing a query. It is defined in seconds and defaults to 60.

CONDOR_DEVELOPERS Condor will send e-mail once per week to this address with the output of the *condor_status* command, which lists how many machines are in the pool and how many are running jobs. Use the default value of condor-admin@cs.wisc.edu and the weekly status message will be sent to the Condor Team at University of Wisconsin-Madison, the developers of Condor. The Condor Team uses these weekly status messages in order to have some idea as to how many Condor pools exist in the world. We appreciate getting the reports, as this is one way we can convince funding agencies that Condor is being used in the real world. If you do not wish this information to be sent to the Condor Team, set the value to NONE which disables this feature, or put in some other address that you want the weekly status report sent to.

COLLECTOR_NAME This macro is used to specify a short description of your pool. It should be about 20 characters long. For example, the name of the UW-Madison Computer Science Condor Pool is "UW-Madison CS".

CONDOR_DEVELOPERS_COLLECTOR By default, every pool sends periodic updates to a central *condor_collector* at UW-Madison with basic information about the status of your pool. This includes only the number of total machines, the number of jobs submitted, the number of machines running jobs, the hostname of your central manager, and the `$(COLLECTOR_NAME)` specified above. These updates help the Condor Team see how Condor is being used around the world. By default, they will be sent to condor.cs.wisc.edu. If you don't want these updates to be sent from your pool, set this macro to NONE.

COLLECTOR_SOCKET_BUFSIZE This specifies the buffer size, in bytes, reserved for *condor_collector* network sockets. The default is 1024000, or a one megabyte buffer. This is a healthy size, even for a large pool. The larger this value, the less likely the *condor_collector* will have stale information about the pool due to dropping update packets. If your pool is small or your central manager has very little RAM, considering setting this parameter to a lower value (perhaps 256000 or 128000).

KEEP_POOL_HISTORY This boolean macro is used to decide if the collector will write out statistical information about the pool to history files. The default is FALSE. The location, size and frequency of history logging is controlled by the other macros.

POOL_HISTORY_DIR This macro sets the name of the directory where the history files reside (if history logging is enabled). The default is the `SPOOL` directory.

POOL_HISTORY_MAX_STORAGE This macro sets the maximum combined size of the history files. When the size of the history files is close to this limit, the oldest information will be discarded. Thus, the larger this parameter's value is, the larger the time range for which history will be available. The default value is 10000000 (10 Mbytes).

POOL_HISTORY_SAMPLING_INTERVAL This macro sets the interval, in seconds, between samples for history logging purposes. When a sample is taken, the collector goes through the information it holds, and summarizes it. The information is written to the history file once for each 4 samples. The default (and recommended) value is 60 seconds. Setting this macro's value too low will increase the load on the collector, while setting it to high will produce less precise statistical information.

COLLECTOR_DEBUG This macro (and other macros related to debug logging in the collector) is described in section 3.3.3 as SUBSYS_DEBUG .

3.3.16 condor_negotiator Configuration File Entries

These macros affect the *condor_negotiator*.

NEGOTIATOR_INTERVAL Sets how often the negotiator starts a negotiation cycle. It is defined in seconds and defaults to 300 (5 minutes).

NEGOTIATOR_TIMEOUT Sets the timeout that the negotiator uses on its network connections to the schedds and startds. It is defined in seconds and defaults to 30.

PRIORITY_HALFLIFE This macro defines the half-life of the user priorities. See section 2.7.2 on User Priorities for details. It is defined in seconds and defaults to 86400 (1 day).

DEFAULT_PRIO_FACTOR This macro sets the priority factor for local users. See section 2.7.2 on User Priorities for details. Defaults to 1.

NICE_USER_PRIO_FACTOR This macro sets the priority factor for nice users. See section 2.7.2 on User Priorities for details. Defaults to 10000000.

REMOTE_PRIO_FACTOR This macro defines the priority factor for remote users (users who who do not belong to the accountant's local domain - see below). See section 2.7.2 on User Priorities for details. Defaults to 10000.

ACCOUNTANT_LOCAL_DOMAIN This macro is used to decide if a user is local or remote. A user is considered to be in the local domain if the UID_DOMAIN matches the value of this macro. Usually, this macro is set to the local UID_DOMAIN. If it is not defined, all users are considered local.

MAX_ACCOUNTANT_DATABASE_SIZE This macro defines the maximum size (in bytes) that the accountant database log file can reach before it is truncated (which re-writes the file in a more compact format). If, after truncating, the file is larger than one half the maximum size specified with this macro, the maximum size will be automatically expanded. The default is 1 megabyte (1000000).

NEGOTIATOR_SOCKET_CACHE_SIZE This macro defines the maximum number of sockets that the negotiator keeps in its open socket cache. Caching open sockets makes the negotiation protocol more efficient by eliminating the need for socket connection establishment for each negotiation cycle. The default is currently 16. To be effective, this parameter should be set to a value greater than the number of schedds submitting jobs to the negotiator at any time.

PREEMPTION_REQUIREMENTS The negotiator will not preempt a job running on a given machine unless the PREEMPTION_REQUIREMENTS expression evaluates to TRUE and the owner of the idle job has a better priority than the owner of the running job. This expression defaults to TRUE.

PREEMPTION_RANK This expression is used to rank machines that the job ranks the same. For example, if the job has no preference, it is usually preferable to preempt a job with a small ImageSize instead of a job with a large ImageSize. The default is to rank all preemptable matches the same. However, the negotiator will always prefer to match the job with an idle machine over a preemptable machine, if the job has no preference between them.

NEGOTIATOR_DEBUG This macro (and other settings related to debug logging in the negotiator) is described in section 3.3.3 as SUBSYS_DEBUG .

3.3.17 condor_eventd Configuration File Entries

These macros affect the Condor Event daemon. See section 3.4.4 on page 122 for an introduction. The eventd is not included in the main Condor binary distribution or installation procedure. It can be installed as a contrib module.

EVENT_LIST List of macros which define events to be managed by the event daemon.

EVENTD_CAPACITY_INFO Configures the bandwidth limits used when scheduling job checkpoint transfers before SHUTDOWN events. The EVENTD_CAPACITY_INFO file has the same format as the NETWORK_CAPACITY_INFO file, described in section 3.10.10.

EVENTD_ROUTING_INFO Configures the network routing information used when scheduling job checkpoint transfers before SHUTDOWN events. The EVENTD_ROUTING_INFO file has the same format as the NETWORK_ROUTING_INFO file, described in section 3.10.10.

EVENTD_INTERVAL The number of seconds between collector queries to determine pool state. The default is 15 minutes (300 seconds).

EVENTD_MAX_PREPARATION The number of minutes before a scheduled event when the eventd should start periodically querying the collector. If 0 (default), the eventd always polls.

EVENTD_SHUTDOWN_SLOW_START_INTERVAL The number of seconds between each machine startup after a shutdown event. The default is 0.

EVENTD_SHUTDOWN_CLEANUP_INTERVAL The number of seconds between each check for old shutdown configurations in the pool. The default is one hour (3600 seconds).

3.4 Installing Contrib Modules

This section describes how to install various *contrib modules* in the Condor system. Some of these modules are separate, optional pieces, not included in the main distribution of Condor. Examples are

the checkpoint server and DAGMan. Others are integral parts of Condor taken from the development series that have certain features users might want to install. Examples are the new SMP-aware *condor_startd* and the CondorView collector. Both of these modules come with Condor version 6.1 and later versions. However, these separate modules may be installed, maintaining most of the stable release, while not switching over to using the development binaries.

3.4.1 Installing the CondorView Client Contrib Module

The CondorView Client Contrib module is used to automatically generate World Wide Web pages to display usage statistics of a Condor pool. Included in the module is a shell script which invokes the *condor_stats* command to retrieve pool usage statistics from the CondorView server and generate HTML pages from the results. Also included is a Java applet which graphically visualizes Condor usage information. Users can interact with the applet to customize the visualization and to zoom in to a specific time frame. Figure 3.2 on page 115 is a screen shot of a web page created by CondorView. To get a further feel for what pages generated by CondorView look like, view the statistics for the University of Wisconsin-Madison pool by visiting the URL http://www.cs.wisc.edu/condor and clicking on Condor View.

After unpacking and installing the CondorView Client, a script named *make_stats* can be invoked to create HTML pages displaying Condor usage for the past hour, day, week, or month. By using the Unix *cron* facility to periodically execute *make_stats*, Condor pool usage statistics can be kept up to date automatically. This simple model allows the CondorView Client to be easily installed; no Web server CGI interface is needed.

Step-by-Step Installation of the CondorView Client

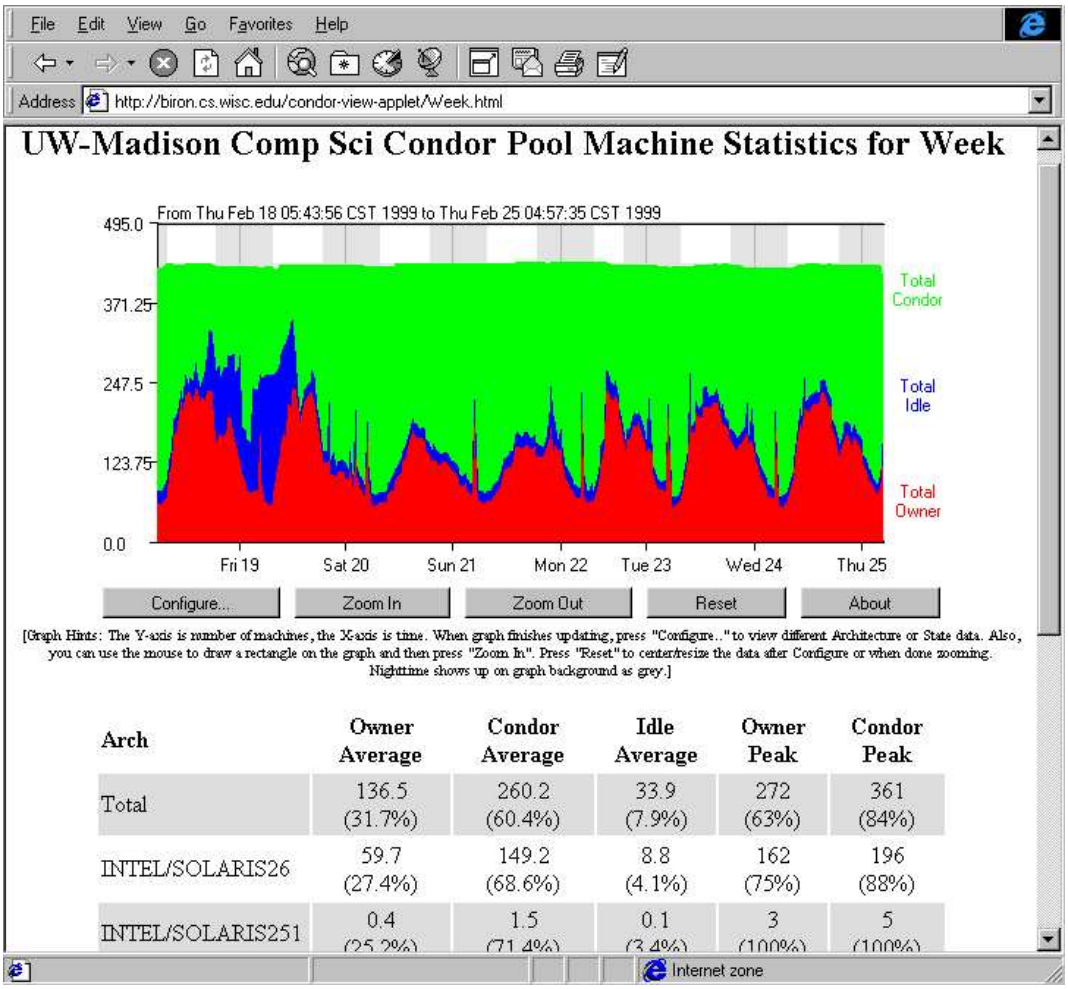
1. Make certain that the CondorView Server is configured. Section 3.10.6 describes configuration of the server. The server logs information on disk in order to provide a persistent, historical database of pool statistics. The CondorView Client makes queries over the network to this database. The *condor_collector* included with version 6.2.x and 6.1.x Condor includes this database support. To activate the persistent database logging, add the following entries to the configuration file on the central manager:

   ```
   POOL_HISTORY_DIR = /full/path/to/directory/to/store/historical/data
   KEEP_POOL_HISTORY = True
   ```

   For full details on these and other condor_collector configuration file entries, see section 3.3.15 on page 110.

2. Create a directory where CondorView is to place the HTML files. This directory should be one published by a web server, so that HTML files which exist in this directory can be accessed using a web browser. This directory is referred to as the `VIEWDIR` directory.

3. Unpack or untar the CondorView Client Contrib module into the directory `VIEWDIR`. This creates several files and subdirectories.



Arch	Owner Average	Condor Average	Idle Average	Owner Peak	Condor Peak
Total	136.5 (31.7%)	260.2 (60.4%)	33.9 (7.9%)	272 (63%)	361 (84%)
INTEL/SOLARIS26	59.7 (27.4%)	149.2 (68.6%)	8.8 (4.1%)	162 (75%)	196 (88%)
INTEL/SOLARIS251	0.4 (25.2%)	1.5 (71.4%)	0.1 (3.4%)	3 (100%)	5 (100%)

Figure 3.2: Screenshot of CondorView Client

4. Edit the *make_stats*script. At the beginning of the file are six parameters to customize. The parameters are

   ORGNAME A brief name that identifies an organization. An example is “Univ of Wisconsin”. Do not use any slashes in the name or other special regular-expression characters. Avoid characters \ ^ $.

   CONDORADMIN The e-mail address of the Condor administrator at your site. This e-mail address will appear at the bottom of the web pages.

   VIEWDIR The full pathname (*not* a relative path) to the `VIEWDIR` directory set by installation step 2. It is the directory that contains the *make_stats* script.

   STATSDIR The full pathname of the directory which contains the *condor_stats* binary. The

condor_stats program is included in the <release_dir>/bin directory with Condor version 6.1 and above; for Condor version 6.0x, the *condor_stats* program can be found in the CondorView Server Contrib module. The value for STATSDIR is added to the PATH parameter by default; see below.

PATH A list of subdirectories, separated by colons, where the *make_stats* script can find the *awk*, *bc*, *sed*, *date*, and *condor_stats* programs. If *perl* is installed, the path should also include the directory where *perl* is installed. The following default works on most systems:

```
PATH=/bin:/usr/bin:$STATSDIR:/usr/local/bin
```

5. To create all of the initial HTML files, type

   ```
   ./make_stats setup
   ```

   Open the file index.html to verify that things look good.

6. Add the *make_stats* program to *cron*. Running *make_stats* in step 5 created a cronentries file. This cronentries file is ready to be processed by the Unix *crontab* command. The *crontab* manual page contains details about the *crontab* command and the *cron* daemon. Look at the cronentries file; by default, it will run *make_stats hour* every 15 minutes, *make_stats day* once an hour, *make_stats week* twice per day, and *make_stats month* once per day. These are reasonable defaults. You can add these commands to cron on any system that can access the $(VIEWDIR) and $(STATSDIR) directories, even on a system that does not have Condor installed. The commands do not need to run as user root; in fact, they should probably not run as root. These commands can run as any user that has read/write access to the VIEWDIR. To add these commands to cron, enter :

   ```
   crontab cronentries
   ```

7. Point the web browser at the VIEWDIR directory, and to complete the installation.

3.4.2 Installing a Checkpoint Server

The Checkpoint Server maintains a repository for checkpoint files. Using checkpoint servers reduces the disk requirements of submitting machines in the pool, since the submitting machines no longer need to store checkpoint files locally. Checkpoint server machines should have a large amount of disk space available, and they should have a fast connection to machines in the Condor pool.

If your spool directories are on a network file system, then checkpoint files will make two trips over the network: one between the submitting machine and the execution machine, and a second between the submitting machine and the network file server. If you install a checkpoint server and configure it to use the server's local disk, the checkpoint will travel only once over the network, between the execution machine and the checkpoint server. You may also obtain checkpointing network performance benefits by using multiple checkpoint servers, as discussed below.

NOTE: It is a good idea to pick very stable machines for your checkpoint servers. If individual checkpoint servers crash, the Condor system will continue to operate, although poorly. While the Condor system will recover from a checkpoint server crash as best it can, there are two problems that can (and will) occur:

1. A checkpoint cannot be sent to a checkpoint server that is not functioning. Jobs will keep trying to contact the checkpoint server, backing off exponentially in the time they wait between attempts. Normally, jobs only have a limited time to checkpoint before they are kicked off the machine. So, if the server is down for a long period of time, chances are that a lot of work will be lost by jobs being killed without writing a checkpoint.
2. If a checkpoint is not available from the checkpoint server, a job cannot be retrieved, and it will either have to be restarted from the beginning, or the job will wait for the server to come back online. This behavior is controlled with the `MAX_DISCARDED_RUN_TIME` parameter in the config file (see section 3.3.6 on page 96 for details). This parameter represents the maximum amount of CPU time you are willing to discard by starting a job over from scratch if the checkpoint server is not responding to requests.

Preparing to Install a Checkpoint Server

The location of checkpoints changes upon the installation of a checkpoint server. A configuration change would cause currently queued jobs with checkpoints to not be able to find their checkpoints. This results in the jobs with checkpoints remaining indefinitely queued (never running) due to the lack of finding their checkpoints. It is therefore best to either remove jobs from the queues or let them complete before installing a checkpoint server. It is advisable to shut your pool down before doing any maintenance on your checkpoint server. See section 3.9 on page 175 for details on shutting down your pool.

A graduated installation of the checkpoint server may be accomplished by configuring submit machines as their queues empty.

Installing the Checkpoint Server Module

To install a checkpoint server, download the appropriate binary contrib module for the platform(s) on which your server will run. Uncompress and untar the file to result in a directory that contains a `README`, `ckpt_server.tar`, and so on. The file `ckpt_server.tar` acts much like the `release.tar` file from a main release. This archive contains the files:

```
sbin/condor_ckpt_server
sbin/condor_cleanckpts
etc/examples/condor_config.local.ckpt.server
```

These new files are not found in the main release, so you can safely untar the archive directly into your existing release directory. `condor_ckpt_server` is the checkpoint server binary. `condor_cleanckpts` is a script that can be periodically run to remove stale checkpoint files from

your server. The checkpoint server normally cleans all old files itself. However, in certain error situations, stale files can be left that are no longer needed. You may set up a cron job that calls *condor_cleanckpts* every week or so to automate the cleaning up of any stale files. The example configuration file give with the module is described below.

After unpacking the module, there are three steps to complete. Each is discussed in its own section:

1. Configure the checkpoint server.
2. Start the checkpoint server.
3. Configure your pool to use the checkpoint server.

Configuring a Checkpoint Server

Place settings in the local configuration file of the checkpoint server. The file `etc/examples/condor_config.local.ckpt.server` contains the needed settings. Insert these into the local configuration file of your checkpoint server machine.

The `CKPT_SERVER_DIR` must be customized. The `CKPT_SERVER_DIR` attribute defines where your checkpoint files are to be located. It is better if this is on a very fast local file system (preferably a RAID). The speed of this file system will have a direct impact on the speed at which your checkpoint files can be retrieved from the remote machines.

The other optional settings are:

DAEMON_LIST (Described in section 3.3.7). To have the checkpoint server managed by the *condor_master*, the `DAEMON_LIST` entry must have `MASTER` and `CKPT_SERVER`. Add `STARTD` if you want to allow jobs to run on your checkpoint server. Similarly, add `SCHEDD` if you would like to submit jobs from your checkpoint server.

The rest of these settings are the checkpoint server-specific versions of the Condor logging entries, as described in section 3.3.3 on page 89.

CKPT_SERVER_LOG The `CKPT_SERVER_LOG` is where the checkpoint server log is placed.

MAX_CKPT_SERVER_LOG Sets the maximum size of the checkpoint server log before it is saved and the log file restarted.

CKPT_SERVER_DEBUG Regulates the amount of information printed in the log file. Currently, the only debug level supported is `D_ALWAYS`.

Start the Checkpoint Server

To start the newly configured checkpoint server, restart Condor on that host to enable the *condor_master* to notice the new configuration. Do this by sending a *condor_restart* command from any

machine with administrator access to your pool. See section 3.7.5 on page 166 for full details about IP/host-based security in Condor.

Configuring your Pool to Use the Checkpoint Server

After the checkpoint server is running, you change a few settings in your configuration files to let your pool know about your new server:

USE_CKPT_SERVER This parameter should be set to TRUE (the default).

CKPT_SERVER_HOST This parameter should be set to the full hostname of the machine that is now running your checkpoint server.

It is most convenient to set these parameters in your global configuration file, so they affect all submission machines. However, you may configure each submission machine separately (using local configuration files) if you do not want all of your submission machines to start using the checkpoint server at one time. If USE_CKPT_SERVER is set to FALSE, the submission machine will not use a checkpoint server.

Once these settings are in place, send a *condor_reconfig* to all machines in your pool so the changes take effect. This is described in section 3.9.2 on page 178.

Configuring your Pool to Use Multiple Checkpoint Servers

It is possible to configure a Condor pool to use multiple checkpoint servers. The deployment of checkpoint servers across the network improves checkpointing performance. In this case, Condor machines are configured to checkpoint to the *nearest* checkpoint server. There are two main performance benefits to deploying multiple checkpoint servers:

- Checkpoint-related network traffic is localized by intelligent placement of checkpoint servers.
- Faster checkpointing implies that jobs spend less time checkpointing, more time doing useful work, jobs have a better chance of checkpointing successfully before returning a machine to its owner, and workstation owners see Condor jobs leave their machines quicker.

Once you have multiple checkpoint servers running in your pool, the following configuration changes are required to make them active.

First, USE_CKPT_SERVER should be set to TRUE (the default) on all submitting machines where Condor jobs should use a checkpoint server. Additionally, STARTER_CHOOSES_CKPT_SERVER should be set to TRUE (the default) on these submitting machines. When TRUE, this parameter specifies that the checkpoint server specified by the machine running the job should be used instead of the checkpoint server specified by the submitting machine. See section 3.3.6 on page 96 for more details. This allows the job to use the checkpoint

server closest to the machine on which it is running, instead of the server closest to the submitting machine. For convenience, set these parameters in the global configuration file.

Second, set CKPT_SERVER_HOST on each machine. As described, this is set to the full hostname of the checkpoint server machine. In the case of multiple checkpoint servers, set this in the local configuraton file. It is the hostname of the nearest server to the machine.

Third, send a *condor_reconfig* to all machines in the pool so the changes take effect. This is described in section 3.9.2 on page 178.

After completing these three steps, the jobs in your pool will send checkpoints to the nearest checkpoint server. On restart, a job will remember where its checkpoint was stored and get it from the appropriate server. After a job successfully writes a checkpoint to a new server, it will remove any previous checkpoints left on other servers.

NOTE: If the configured checkpoint server is unavailable, the job will keep trying to contact that server as described above. It will not use alternate checkpoint servers. This may change in future versions of Condor.

Checkpoint Server Domains

The configuration described in the previous section ensures that jobs will always write checkpoints to their nearest checkpoint server. In some circumstances, it is also useful to configure Condor to localize checkpoint read transfers, which occur when the job restarts from its last checkpoint on a new machine. To localize these transfers, we want to schedule the job on a machine which is near the checkpoint server on which the job's checkpoint is stored.

We can say that all of the machines configured to use checkpoint server "A" are in "checkpoint server domain A." To localize checkpoint transfers, we want jobs which run on machines in a given checkpoint server domain to continue running on machines in that domain, transferring checkpoint files in a single local area of the network. There are two possible configurations which specify what a job should do when there are no available machines in its checkpoint server domain:

- The job can remain idle until a workstation in its checkpoint server domain becomes available.
- The job can try to immediately begin executing on a machine in another checkpoint server domain. In this case, the job transfers to a new checkpoint server domain.

These two configurations are described below.

The first step in implementing checkpoint server domains is to include the name of the nearest checkpoint server in the machine ClassAd, so this information can be used in job scheduling decisions. To do this, add the following configuration to each machine:

```
CkptServer = "$(CKPT_SERVER_HOST)"
STARTD_EXPRS = $(STARTD_EXPRS), CkptServer
```

For convenience, we suggest that you set these parameters in the global config file. Note that this example assumes that STARTD_EXPRS is defined previously in your configuration. If not, then you should use the following configuration instead:

```
CkptServer = "$(CKPT_SERVER_HOST)"
STARTD_EXPRS = CkptServer
```

Now, all machine ClassAds will include a CkptServer attribute, which is the name of the checkpoint server closest to this machine. So, the CkptServer attribute defines the checkpoint server domain of each machine.

To restrict jobs to one checkpoint server domain, we need to modify the jobs' Requirements expression as follows:

```
Requirements = ((LastCkptServer == TARGET.CkptServer) || (LastCkpt-
Server =?= UNDEFINED))
```

This Requirements expression uses the LastCkptServer attribute in the job's ClassAd, which specifies where the job last wrote a checkpoint, and the CkptServer attribute in the machine ClassAd, which specifies the checkpoint server domain. If the job has not written a checkpoint yet, the LastCkptServer attribute will be UNDEFINED, and the job will be able to execute in any checkpoint server domain. However, once the job performs a checkpoint, LastCkptServer will be defined and the job will be restricted to the checkpoint server domain where it started running.

If instead we want to allow jobs to transfer to other checkpoint server domains when there are no available machines in the current checkpoint server domain, we need to modify the jobs' Rank expression as follows:

```
Rank = ((LastCkptServer == TARGET.CkptServer) || (LastCkpt-
Server =?= UNDEFINED))
```

This Rank expression will evaluate to 1 for machines in the job's checkpoint server domain and 0 for other machines. So, the job will prefer to run on machines in its checkpoint server domain, but if no such machines are available, the job will run in a new checkpoint server domain.

You can automatically append the checkpoint server domain Requirements or Rank expressions to all STANDARD universe jobs submitted in your pool using APPEND_REQ_STANDARD or APPEND_RANK_STANDARD . See section 3.3.13 on page 109 for more details.

3.4.3 Installing PVM Support in Condor

To install the PVM contrib module, you must first download the appropriate binary module for whatever platform(s) you plan to use for Condor-PVM. You can find all of the Condor binary modules at http://www.cs.wisc.edu/condor/downloads.

NOTE: The PVM contrib module version must match with your installed Condor version.

Once you have downloaded each module, uncompressed and untarred it, you will be left with a directory that contains a `pvm.tar`, `README` and so on. The `pvm.tar` acts much like the `release.tar` file for a main release. It contains all the binaries and supporting files you would install in your release directory to enable Condor-PVM:

```
sbin/condor_pvmd
sbin/condor_pvmgs
sbin/condor_shadow.pvm
sbin/condor_starter.pvm
```

You must install these files in the release directory for the platform they were built for. Since these files do not exist in a main release, you can safely untar the `pvm.tar` directly into the appropriate release directory. You do not need to worry about shutting down Condor, moving files out of the way, and so on. Once the `pvm.tar` file has been untarred into the release directory, you are done installing the PVM contrib module. You will now be able to submit PVM jobs to your Condor pool.

For complete documentation on using PVM in Condor, see the section 2.9 on page 41 entitled "Parallel Applications in Condor: Condor-PVM".

3.4.4 Installing the Condor Event Daemon

The event daemon is an administrative tool for scheduling events in a Condor pool. Every `EVENTD_INTERVAL` , for each defined event, the event daemon (eventd) computes an estimate of the time required to complete or prepare for the event. If the time required is less than the time between the next interval and the start of the event, the event daemon activates the event.

Currently, this daemon supports `SHUTDOWN` events, which place machines in the owner state during scheduled times. The eventd causes machines to vacate jobs in an orderly fashion in anticipation of `SHUTDOWN` events. Scheduling this improves performance, because the machines do not all attempt to checkpoint their jobs at the same time. Instead, the eventd schedules checkpoint transfers according to bandwidth limits defined in its configuration files.

When a `SHUTDOWN` event is activated, the eventd contacts all startd daemons that match constraints given in the configuration file, and instructs them to shut down. In response to this instruction, the startd on any machine not running a job will immediately transition to the owner state. Any machine currently running a job will continue to run the job, but will not start any new job. The eventd then sends a vacate command to each startd that is currently running a job. Once the job is vacated, the startd transitions to the owner state and remains in this state for the duration of the `SHUTDOWN` event.

The *condor_eventd* must run on a machine with `STARTD ADMINISTRATOR` and `CONFIG` access to your pool. See section 3.7.5 on page 166 for full details about IP/host-based security in Condor.

Installing the Event Daemon

First, download the *condor_eventd* contrib module. Uncompress and untar the file, to have a directory that contains a `eventd.tar`. The `eventd.tar` acts much like the `release.tar` file from a main release. This archive contains the files:

```
sbin/condor_eventd
etc/examples/condor_config.local.eventd
```

These are all new files, not found in the main release, so you can safely untar the archive directly into your existing release directory. The file `condor_eventd` is the eventd binary. The example configuration file is described below.

Configuring the Event Daemon

The file `etc/examples/condor_config.local.eventd` contains an example configuration. To define events, first set the `EVENT_LIST` macro. This macro contains a list of macro names which define the individual events. The definition of individual events depends on the type of the event. The format for `SHUTDOWN` events is

```
SHUTDOWN DAY TIME DURATION CONSTRAINT
```

`TIME` and `DURATION` are specified in an hours:minutes format. `DAY` is a string of days, where `M` = Monday, `T` = Tuesday, `W` = Wednesday, `R` = Thursday, `F` = Friday, `S` = Saturday, and `U` = Sunday. For example, `MTWRFSU` would specify that the event occurs daily, `MTWRF` would specify that the event occurs only on weekdays, and `SU` would specificy that the event occurs only on weekends.

Two options can be specified to change the default behavior of `SHUTDOWN` events. If `_RUNTIME` is appended to the `SHUTDOWN` event specification, the startd shutdown configurations will not be persistent. If a machine reboots or a startd is restarted, the startd will no longer be "shutdown" and may transition out of the owner state. This is useful for reboot events, where the startd should leave the shutdown state when the machine reboots. If `_STANDARD` is appended to the `SHUTDOWN` event specification, the eventd will only consider standard universe jobs. It will vacate only standard universe jobs and configure machines to run only non-standard universe jobs during the shutdown event. This is also useful for reboot events, where there is no point vacating vanilla or pvm jobs before the machine is rebooted because they are unable to checkpoint. Reboot events are usually listed as `SHUTDOWN_RUNTIME_STANDARD`.

The following is an example event daemon configuration:

```
EVENT_LIST = TestEvent, TestEvent2
TestEvent = SHUTDOWN_RUNTIME MTWRFSU 2:00 1:00 TestEventConstraint
TestEvent2 = SHUTDOWN MTWRF 14:00 0:30 TestEventConstraint2
TestEventConstraint = (Arch == "INTEL")
TestEventConstraint2 = (True)
```

In this example, the TestEvent is a SHUTDOWN_RUNTIME type event, which specifies that all machines whose startd ads match the constraint Arch == "INTEL" should be shutdown for one hour (or until the startd restarts) starting at 2:00 every Wednesday. TestEvent2 is a SHUTDOWN type event, which specifies that all machines should be shutdown for 30 minutes starting at 14:00 every Friday.

The bandwidth limits used in the eventd's schedule are specified in the file indicated by the EVENTD_CAPACITY_INFO parameter, and any network routing information required to implement those limits is specified in the file indicated by the EVENTD_ROUTING_INFO parameter. The EVENTD_CAPACITY_INFO file has the same format as the NETWORK_CAPACITY_INFO file, described in section 3.10.10. The EVENTD_ROUTING_INFO file has the same format as the NETWORK_ROUTING_INFO file, described in section 3.10.10.

Note that the DAEMON_LIST macro (described in section 3.3.7) is defined in the section of settings you may want to customize. If you want the event daemon managed by the *condor_master*, the DAEMON_LIST entry must contain both MASTER and EVENTD. Verify that this macro is set to run the correct daemons on this machine. By default, the list also includes SCHEDD and STARTD.

See section 3.3.17 on page 113 for a description of optional event daemon parameters.

Starting the Event Daemon

To start an event daemon once it is configured to run on a given machine, restart Condor on that given machine to enable the *condor_master* to notice the new configuration. Send a *condor_restart* command from any machine with administrator access to your pool. See section 3.7.5 on page 166 for full details about IP/host-based security in Condor.

3.4.5 Installing the Condor-G contrib module

To install the Condor-G contrib module, please see the complete instructions given in section 5.3.2 on page 231.

3.5 User Priorities in the Condor System

Condor uses priorities to determine machine allocation for jobs. This section details the priorities.

For accounting purposes, each user is identified by username@uid_domain. Each user is assigned a priority value even if submitting jobs from different machines in the same domain, or even submit from multiple machines in the different domains.

The numerical priority value assigned to a user is inversely related to the *goodness* of the priority. A user with a numerical priority of 5 gets more resources than a user with a numerical priority of 50. There are two priority values assigned to Condor users:

- Real User Priority (RUP), which measures resource usage of the user.
- Effective User Priority (EUP), which determines the number of resources the user can get.

This section describes these two priorities and how they affect resource allocations in Condor. Documentation on configuring and controlling priorities may be found in section 3.3.16.

3.5.1 Real User Priority (RUP)

A user's RUP measures the resource usage of the user through time. Every user begins with a RUP of one half (0.5), and at steady state, the RUP of a user equilibrates to the number of resources used by that user. Therefore, if a specific user continuously uses exactly ten resources for a long period of time, the RUP of that user stabilizes at ten.

However, if the user decreases the number of resources used, the RUP gets better. The rate at which the priority value decays can be set by the macro `PRIORITY_HALFLIFE` , a time period defined in seconds. Intuitively, if the `PRIORITY_HALFLIFE` in a pool is set to 86400 (one day), and if a user whose RUP was 10 removes all his jobs, the user's RUP would be 5 one day later, 2.5 two days later, and so on.

3.5.2 Effective User Priority (EUP)

The effective user priority (EUP) of a user is used to determine how many resources that user may receive. The EUP is linearly related to the RUP by a *priority factor* which may be defined on a per-user basis. Unless otherwise configured, the priority factor for all users is 1.0, and so the EUP is the same as the the RUP. However, if desired, the priority factors of specific users (such as remote submitters) can be increased so that others are served preferentially.

The number of resources that a user may receive is inversely related to the ratio between the EUPs of submitting users. Therefore user A with EUP=5 will receive twice as many resources as user B with EUP=10 and four times as many resources as user C with EUP=20. However, if A does not use the full number of allocated resources, the available resources are repartitioned and distributed among remaining users according to the inverse ratio rule.

Condor supplies mechanisms to directly support two policies in which EUP may be useful:

Nice users A job may be submitted with the parameter `nice_user` set to TRUE in the submit command file. A nice user job gets its RUP boosted by the `NICE_USER_PRIO_FACTOR` priority factor specified in the configuration file, leading to a (usually very large) EUP. This corresponds to a low priority for resources. These jobs are therefore equivalent to Unix background jobs, which use resources not used by other Condor users.

Remote Users The flocking feature of Condor (see section 3.10.7) allows the *condor_schedd* to submit to more than one pool. In addition, the submit-only feature allows a user to run a *condor_schedd* that is submitting jobs into another pool. In such situations, submitters from

other domains can submit to the local pool. It is often desirable to have Condor treat local users preferentially over these remote users. If configured, Condor will boost the RUPs of remote users by REMOTE_PRIO_FACTOR specified in the configuration file, thereby lowering their priority for resources.

The priority boost factors for individual users can be set with the **setfactor** option of *condor_userprio*. Details may be found in the *condor_submit* manual page on page 418.

3.5.3 Priorities and Preemption

Priorities are used to ensure that users get their fair share of resources. The priority values are used at allocation time. In addition, Condor preempts machines (by performing a checkpoint and vacate) and reallocates them to maintain priority standing.

To ensure that preemptions do not lead to *thrashing*, a PREEMPTION_REQUIREMENTS expression is defined to specify the conditions that must be met for a preemption to occur. It is usually defined to deny preemption if a current running job has been running for a relatively short period of time. This effectively limits the number of preemptions per resource per time interval.

3.5.4 Priority Calculation

This section may be skipped if the reader so feels, but for the curious, here is Condor's priority calculation algorithm.

The RUP of a user u at time t, $\pi_r(u,t)$, is calculated every time interval δt using the formula

$$\pi_r(u,t) = \beta \times \pi(u, t-\delta t) + (1-\beta) \times \rho(u,t)$$

where $\rho(u,t)$ is the number of resources used by user u at time t, and $\beta = 0.5^{\delta t/h}$. h is the half life period set by PRIORITY_HALFLIFE .

The EUP of user u at time t, $\pi_e(u,t)$ is calculated by

$$\pi_e(u,t) = \pi_r(u,t) \times f(u,t)$$

where $f(u,t)$ is the priority boost factor for user u at time t.

As mentioned previously, the RUP calculation is designed so that at steady state, each user's RUP stabilizes at the number of resources used by that user. The definition of β ensures that the calculation of $\pi_r(u,t)$ can be calculated over non-uniform time intervals δt without affecting the calculation. The time interval δt varies due to events internal to the system, but Condor guarantees that unless the central manager machine is down, no matches will be unaccounted for due to this variance.

3.6 Configuring The Startd Policy

This section describes the configuration of the *condor_startd* to implement the desired policy for when remote jobs should start, be suspended, (possibly) resumed, vacate (with a checkpoint) or be killed (no checkpoint). This policy is the heart of Condor's balancing act between the needs and wishes of resource owners (machine owners) and resource users (people submitting their jobs to Condor). Please read this section carefully if you plan to change any of the settings described here, as a wrong setting can have a severe impact on either the owners of machines in your pool (they may ask to be removed from the pool entirely) or the users of your pool (they may stop using Condor).

Before we get into the details, there are a few things to note:

- Much of this section refers to ClassAd expressions. You probably want to read through section 4.1 on ClassAd expressions before continuing with this.
- If you are primarily familiar with the version 6.0 policy expressions and what they do, read section 3.6.10 on page 149. This section explains the differences between the version 6.0 policy expressions and later versions.
- If you are defining the policy for an SMP machine (a multi-CPU machine), also read section 3.10.8 for specific information on configuring the *condor_startd* for SMP machines. Each *virtual machine* represented by the *condor_startd* on an SMP machine has its own *state* and *activity* (as described below). In the future, each virtual machine will be able to have its own individual policy expressions defined. Within this manual section, the word "machine" refers to an individual virtual machine within an SMP machine.

To define your policy, set expressions in the configuration file (see section 3.3 on Configuring Condor for an introduction to Condor's configuration files). The expressions are evaluated in the context of the machine's ClassAd and a job ClassAd. The expressions can therefore reference attributes from either ClassAd. Listed in this section are both the attributes that are included in the machine's ClassAd and the attributes that are included in a job ClassAd. The `START` expression is explained. It describes the conditions that must be met for a machine to start a job. The `RANK` expression is described. It allows the specification of the kinds of jobs a machine prefers to run. A final discussion details how the *condor_startd* daemon works. Included are the machine *states* and *activities*, to give an idea of what is possible in policy decisions. Two example policy settings are presented.

3.6.1 Startd ClassAd Attributes

The *condor_startd* daemon represents the machine on which it is running to the Condor pool. The daemon publishes characteristics about the machine in the machine's ClassAd to aid matchmaking with resource requests. The values of these attributes may be listed by using the command: *condor_status -l hostname*. On an SMP machine, the *condor_startd* will break the machine up and advertise it as separate virtual machines, each with its own name and ClassAd. The attributes themselves and what they represent are described below:

Activity : String which describes Condor job activity on the machine. Can have one of the following values:

"Idle" : There is no job activity

"Busy" : A job is busy running

"Suspended" : A job is currently suspended

"Vacating" : A job is currently checkpointing

"Killing" : A job is currently being killed

"Benchmarking" : The startd is running benchmarks

Arch : String with the architecture of the machine. Typically one of the following:

"INTEL" : Intel x86 CPU (Pentium, Xeon, etc).

"ALPHA" : Digital Alpha CPU

"SGI" : Silicon Graphics MIPS CPU

"SUN4u" : Sun UltraSparc CPU

"SUN4x" : A Sun Sparc CPU other than an UltraSparc, i.e. sun4m or sun4c CPU found in older Sparc workstations such as the Sparc 10, Sparc 20, IPC, IPX, etc.

"HPPA1" : Hewlett Packard PA-RISC 1.x CPU (i.e. PA-RISC 7000 series CPU) based workstation

"HPPA2" : Hewlett Packard PA-RISC 2.x CPU (i.e. PA-RISC 8000 series CPU) based workstation

ClockDay : The day of the week, where 0 = Sunday, 1 = Monday, . . ., 6 = Saturday.

ClockMin : The number of minutes passed since midnight.

CondorLoadAvg : The portion of the load average generated by Condor (either from remote jobs or running benchmarks).

ConsoleIdle : The number of seconds since activity on the system console keyboard or console mouse has last been detected.

Cpus : Number of CPUs in this machine, i.e. 1 = single CPU machine, 2 = dual CPUs, etc.

CurrentRank : A float which represents this machine owner's affinity for running the Condor job which it is currently hosting. If not currently hosting a Condor job, `CurrentRank` is -1.0.

Disk : The amount of disk space on this machine available for the job in kbytes (e.g. 23000 = 23 megabytes). Specifically, this is the amount of disk space available in the directory specified in the Condor configuration files by the `EXECUTE` macro, minus any space reserved with the `RESERVED_DISK` macro.

EnteredCurrentActivity : Time at which the machine entered the current Activity (see `Activity` entry above). On all platforms (including NT), this is measured in the number of seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

FileSystemDomain : A "domain" name configured by the Condor administrator which describes a cluster of machines which all access the same, uniformly-mounted, networked file systems usually via NFS or AFS. This is useful for Vanilla universe jobs which require remote file access.

KeyboardIdle : The number of seconds since activity on any keyboard or mouse associated with this machine has last been detected. Unlike `ConsoleIdle`, `KeyboardIdle` also takes activity on pseudo-terminals into account (i.e. virtual "keyboard" activity from telnet and rlogin sessions as well). Note that `KeyboardIdle` will always be equal to or less than `ConsoleIdle`.

KFlops : Relative floating point performance as determined via a Linpack benchmark.

LastHeardFrom : Time when the Condor central manager last received a status update from this machine. Expressed as seconds since the epoch (integer value). Note: This attribute is only inserted by the central manager once it receives the ClassAd. It is not present in the *condor_startd* copy of the ClassAd. Therefore, you could not use this attribute in defining *condor_startd* expressions (and you would not want to).

LoadAvg : A floating point number with the machine's current load average.

Machine : A string with the machine's fully qualified hostname.

Memory : The amount of RAM in megabytes.

Mips : Relative integer performance as determined via a Dhrystone benchmark.

MyType : The ClassAd type; always set to the literal string `"Machine"`.

Name : The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate virtual machines, each with with a unique name. These names will be of the form "vm#@full.hostname", for example, "vm1@vulture.cs.wisc.edu", which signifies virtual machine 1 from vulture.cs.wisc.edu.

OpSys : String describing the operating system running on this machine. For Condor Version 6.4.0 typically one of the following:

"HPUX10" : for HPUX 10.20

"IRIX6" : for IRIX 6.2, 6.3, or 6.4

"IRIX65" : for IRIX 6.5

"LINUX" : for LINUX 2.0.x or LINUX 2.2.x kernel systems

"OSF1" : for Digital Unix 4.x

"SOLARIS251"

"SOLARIS26"

"SOLARIS27"

"SOLARIS28"

"WINNT40" : for Windows NT 4.0

Requirements : A boolean, which when evaluated within the context of the machine ClassAd and a job ClassAd, must evaluate to TRUE before Condor will allow the job to use this machine.

StartdIpAddr : String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd.

State : String which publishes the machine's Condor state. Can be:

"Owner" : The machine owner is using the machine, and it is unavailable to Condor.

"Unclaimed" : The machine is available to run Condor jobs, but a good match is either not available or not yet found.

"Matched" : The Condor central manager has found a good match for this resource, but a Condor scheduler has not yet claimed it.

"Claimed" : The machine is claimed by a remote *condor_schedd* and is probably running a job.

"Preempting" : A Condor job is being preempted (possibly via checkpointing) in order to clear the machine for either a higher priority job or because the machine owner wants the machine back.

TargetType : Describes what type of ClassAd to match with. Always set to the string literal "Job", because machine ClassAds always want to be matched with jobs, and vice-versa.

UidDomain : a domain name configured by the Condor administrator which describes a cluster of machines which all have the same `passwd` file entries, and therefore all have the same logins.

VirtualMemory : The amount of currently available virtual memory (swap space) expressed in kbytes.

In addition, there are a few attributes that are automatically inserted into the machine ClassAd whenever a resource is in the Claimed state:

ClientMachine : The hostname of the machine that has claimed this resource

CurrentRank : The value of the `RANK` expression when evaluated against the ClassAd of the "current" job using this machine. If the resource has been claimed but no job is running, the "current" job ClassAd is the one that was used when claiming the resource. If a job is currently running, that job's ClassAd is the one that was used when claiming the resource. If a job is currently running, that job's ClassAd is the "current" one. If the resource is between jobs, the ClassAd of the last job that was run is used for `CurrentRank`.

RemoteOwner : The name of the user who originally claimed this resource.

RemoteUser : The name of the user who is currently using this resource. In general, this will always be the same as the `RemoteOwner`, but in some cases, a resource can be claimed by one entity that hands off the resource to another entity which uses it. In that case, `RemoteUser` would hold the name of the entity currently using the resource, while `RemoteOwner` would hold the name of the entity that claimed the resource.

Finally, there are a few attributes that are only inserted into the machine ClassAd if a job is currently executing. If the resource is claimed but no job are running, none of these attributes will be defined.

JobId : The job's identifier (for example,

```
152.3
```

), like you would see in *condor_q* on the submitting machine.

JobStart : The timestamp of when the job began executing.

LastPeriodicCheckpoint : If the job has performed a periodic checkpoint, this attribute will be defined and will hold the timestamp of when the last periodic checkpoint was begun. If the job has yet to perform a periodic checkpoint, or cannot checkpoint at all, the `LastPeriodicCheckpoint` attribute will not be defined.

3.6.2 Job ClassAd Attributes

CkptArch : String describing the architecture of the machine where this job last checkpointed. If the job has never checkpointed, this attribute is UNDEFINED.

CkptOpSys : String describing the operating system of the machine where this job last checkpointed. If the job has never checkpointed, this attribute is UNDEFINED.

ClusterId : Integer cluster identifier for this job. A "cluster" is a group of jobs that were submitted together. Each job has its own unique job identifier within the cluser, but shares a common cluster identifier.

CumulativeSuspensionTime : A running total of the number of seconds the job has spent in suspension for the life of the job.

ExecutableSize : Size of the executable in kbytes.

ExitBySignal : When a user job exits, this is true if the job exited via a signal, and false otherwise. Currently this is available for use only in Standard and Vanilla universe jobs.

ExitCode : When a user job exits by means other than a signal, this is the exit return code of the user job. Currently this is available for use only in Standard and Vanilla universe jobs.

ExitSignal : When a user job exits by means of an unhandled signal, this attribute takes on the numeric value of the signal. Currently this is available for use only in Standard and Vanilla universe jobs.

ImageSize : Estimate of the memory image size of the job in kbytes. The initial estimate may be specified in the job submit file. Otherwise, the initial value is equal to the size of the executable. When the job checkpoints, the `ImageSize` attribute is set to the size of the checkpoint file (since the checkpoint file contains the job's memory image).

JobPrio : Integer priority for this job, set by *condor_submit* or *condor_prio*. The default value is 0. The higher the number, the worse the priority.

JobStartDate : Time at which the job first began running. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

JobStatus : Integer which indicates the current status of the job, where 1 = Idle, 2 = Running, 3 = Removed, 4 = Completed, and 5 = Held.

JobUniverse : Integer which indicates the job universe, where 1 = Standard, 4 = PVM, 5 = Vanilla, and 7 = Scheduler.

LastCkptServer : Hostname of the last checkpoint server used by this job. When a pool is using multiple checkpoint servers, this tells the job where to find its checkpoint file.

LastCkptTime : Time at which the job last performed a successful checkpoint. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

LastSuspensionTime : Time at which the job last performed a successful suspension. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

LastVacateTime : Time at which the job was last evicted from a remote workstation. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

NumCkpts : A count of the number of checkpoints written by this job during its lifetime.

NumRestarts : A count of the number of restarts from a checkpoint attempted by this job during its lifetime.

NiceUser : Boolean value which indicates whether this is a nice-user job.

Owner : String describing the user who submitted this job.

ProcId : Integer process identifier for this job. In a cluster of many jobs, each job will have the same ClusterId but will have a unique ProcId.

TotalSuspensions : A count of the number of times this job has been suspended during its lifetime.

QDate : Time at which the job was submitted to the job queue. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

3.6.3 The `START` expression

The most important expression to the *condor_startd* is the `START` expression. This expression describes the conditions that must be met for a machine to run a job. This expression can reference attributes in the machine's ClassAd (such as `KeyboardIdle` and `LoadAvg`) or attributes in a job ClassAd (such as `Owner`, `Imagesize`, and `Cmd`, the name of the executable the job will run). The value after `START` expression evaluation plays a crucial role in determining the state and activity of a machine.

The Requirements expression is used for matching machines with jobs. The *condor_startd* defines the Requirements expression by using the START expression. In situations where a machine wants to make itself unavailable for further matches, the Requirements expression is set to FALSE. When the START expression locally evaluates to TRUE, the machine advertises the Requirements expression as TRUE and does not publish the START expression.

Normally, the expressions in the machine ClassAd are evaluated against certain request ClassAds in the *condor_negotiator* to see if there is a match, or against whatever request ClassAd currently has claimed the machine. However, by locally evaluating an expression, the machine only evaluates the expression against its own ClassAd. If an expression cannot be locally evaluated (because it references other expressions that are only found in a request ad, such as Owner or Imagesize), the expression is (usually) undefined. See section 4.1 for specifics on how undefined terms are handled in ClassAd expression evaluation.

NOTE: If you have machines with lots of real memory and swap space so the only scarce resource is CPU time, you could use JOB_RENICE_INCREMENT (see section 3.3.12 on condor_starter Configuration File Macros for details) so that Condor starts jobs on your machine with low priority. Then, set up your machines with:

```
START = True
SUSPEND = False
PREEMPT = False
KILL = False
```

In this way, Condor jobs always run and never be kicked off. However, because they would run with "nice priority", interactive response on your machines will not suffer. You probably would not notice Condor was running the jobs, assuming you had enough free memory for the Condor jobs that there was little swapping.

3.6.4 The RANK expression

A machine may be configured to prefer certain jobs over others using the RANK expression. It is an expression, like any other in a machine ClassAd. It can reference any attribute found in either the machine ClassAd or a request ad (normally, in fact, it references things in the request ad). The most common use of this expression is likely to configure a machine to prefer to run jobs from the owner of that machine, or by extension, a group of machines to prefer jobs from the owners of those machines.

For example, imagine there is a small research group with 4 machines called tenorsax, piano, bass, and drums. These machines are owned by the 4 users coltrane, tyner, garrison, and jones, respectively.

Assume that there is a large Condor pool in your department, but you spent a lot of money on really fast machines for your group. You want to implement a policy that gives priority on your machines to anyone in your group. To achieve this, set the RANK expression on your machines to

reference the `Owner` attribute and prefer requests where that attribute matches one of the people in your group as in

```
RANK = Owner == "coltrane" || Owner == "tyner" \
       || Owner == "garrison" || Owner == "jones"
```

The `RANK` expression is evaluated as a floating point number. However, like in C, boolean expressions evaluate to either 1 or 0 depending on if they are TRUE or FALSE. So, if this expression evaluated to 1 (because the remote job was owned by one of the preferred users), it would be a larger value than any other user (for whom the expression would evaluate to 0).

A more complex `RANK` expression has the same basic set up, where anyone from your group has priority on your machines. Its difference is that the machine owner has better priority on their own machine. To set this up for Jimmy Garrison, place the following entry in Jimmy Garrison's local configuration file `bass.local`:

```
RANK = (Owner == "coltrane") + (Owner == "tyner") \
       + ((Owner == "garrison") * 10) + (Owner == "jones")
```

NOTE: The parentheses in this expression are important, because "+" operator has higher default precedence than "==".

The use of "+" instead of "| |" allows us to distinguish which terms matched and which ones didn't. If anyone not in the John Coltrane quartet was running a job on the machine called bass, the `RANK` would evaluate numerically to 0, since none of the boolean terms evaluates to 1, and 0+0+0+0 still equals 0.

Suppose Elvin Jones submits a job. His job would match this machine (assuming the `START` was True for him at that time) and the `RANK` would numerically evaluate to 1. Therefore, Elvin would preempt the Condor job currently running. Assume that later Jimmy submits a job. The `RANK` evaluates to 10, since the boolean that matches Jimmy gets multiplied by 10. Jimmy would preempt Elvin, and Jimmy's job would run on Jimmy's machine.

The `RANK` expression is not required to reference the `Owner` of the jobs. Perhaps there is one machine with an enormous amount of memory, and others with not much at all. You can configure your large-memory machine to prefer to run jobs with larger memory requirements:

```
RANK = ImageSize
```

That's all there is to it. The bigger the job, the more this machine wants to run it. It is an altruistic preference, always servicing the largest of jobs, no matter who submitted them. A little less altruistic is John's `RANK` that prefers his jobs over those with the largest `Imagesize`:

```
RANK = (Owner == "coltrane" * 1000000000000) + Imagesize
```

This `RANK` breaks if a job is submitted with an image size of more 10^{12} Kbytes. However, with that size, this `RANK` expression preferring that job would not be Condors only problem!

3.6.5 Machine States

A machine is assigned a *state* by Condor. The state depends on whether or not the machine is available to run Condor jobs, and if so, what point in the negotiations has been reached. The possible states are

Owner The machine is being used by the machine owner, and/or is not available to run Condor jobs. When the machine first starts up, it begins in this state.

Unclaimed The machine is available to run Condor jobs, but it is not currently doing so.

Matched The machine is available to run jobs, and it has been matched by the negotiator with a specific schedd. That schedd just has not yet claimed this machine. In this state, the machine is unavailable for further matches.

Claimed The machine has been claimed by a schedd.

Preempting The machine was claimed by a schedd, but is now preempting that claim for one of the following reasons.

1. the owner of the machine came back
2. another user with higher priority has jobs waiting to run
3. another request that this resource would rather serve was found

Figure 3.3 shows the states and the possible transitions between the states.

3.6.6 Machine Activities

Within some machine states, *activities* of the machine are defined. The state has meaning regardless of activity. Differences between activities are significant. Therefore, a "state/activity" pair describes a machine. The following list describes all the possible state/activity pairs.

- Owner

  Idle This is the only activity for Owner state. As far as Condor is concerned the machine is Idle, since it is not doing anything for Condor.

- Unclaimed

  Idle This is the normal activity of Unclaimed machines. The machine is still Idle in that the machine owner is willing to let Condor jobs run, but Condor is not using the machine for anything.

  Benchmarking The machine is running benchmarks to determine the speed on this machine. This activity only occurs in the Unclaimed state. How often the activity occurs is determined by the `RunBenchmarks` expression.

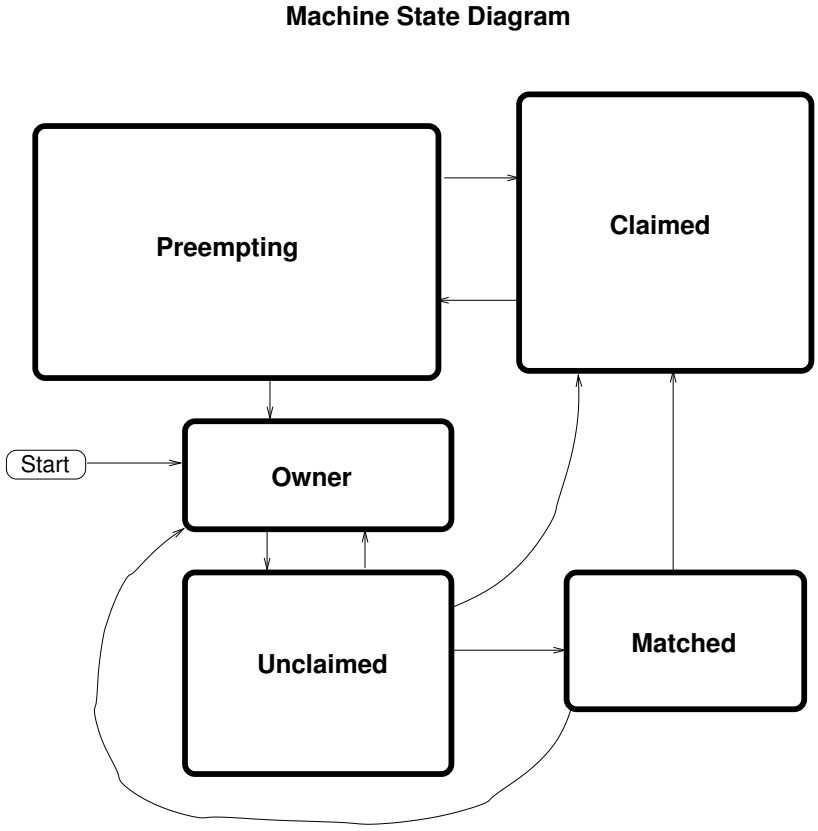


Figure 3.3: Machine States

- Matched

  Idle When Matched, the machine is still Idle to Condor.

- Claimed

  Idle In this activity, the machine has been claimed, but the schedd that claimed it has yet to *activate* the claim by requesting a *condor_starter* to be spawned to service a job.

  Busy Once a *condor_starter* has been started and the claim is active, the machine moves to the Busy activity to signify that it is doing something as far as Condor is concerned.

  Suspended If the job is suspended by Condor, the machine goes into the Suspended activity. The match between the schedd and machine has not been broken (the claim is still valid), but the job is not making any progress and Condor is no longer generating a load on the machine.

- Preempting The preempting state is used for evicting a Condor job from a given machine. When the machine enters the Preempting state, it checks the `WANT_VACATE` expression to determine its activity.

Vacating In the Vacating activity, the job that was running is in the process of checkpointing. As soon as the checkpoint process completes, the machine moves into either the Owner state or the Claimed state, depending on the reason for its preemption.

Killing Killing means that the machine has requested the running job to exit the machine immediately, without checkpointing.

Figure 3.4 on page 137 gives the overall view of all machine states and activities and shows the possible transitions from one to another within the Condor system. Each transition is labeled with a number on the diagram, and transition numbers referred to in this manual will be **bold**.

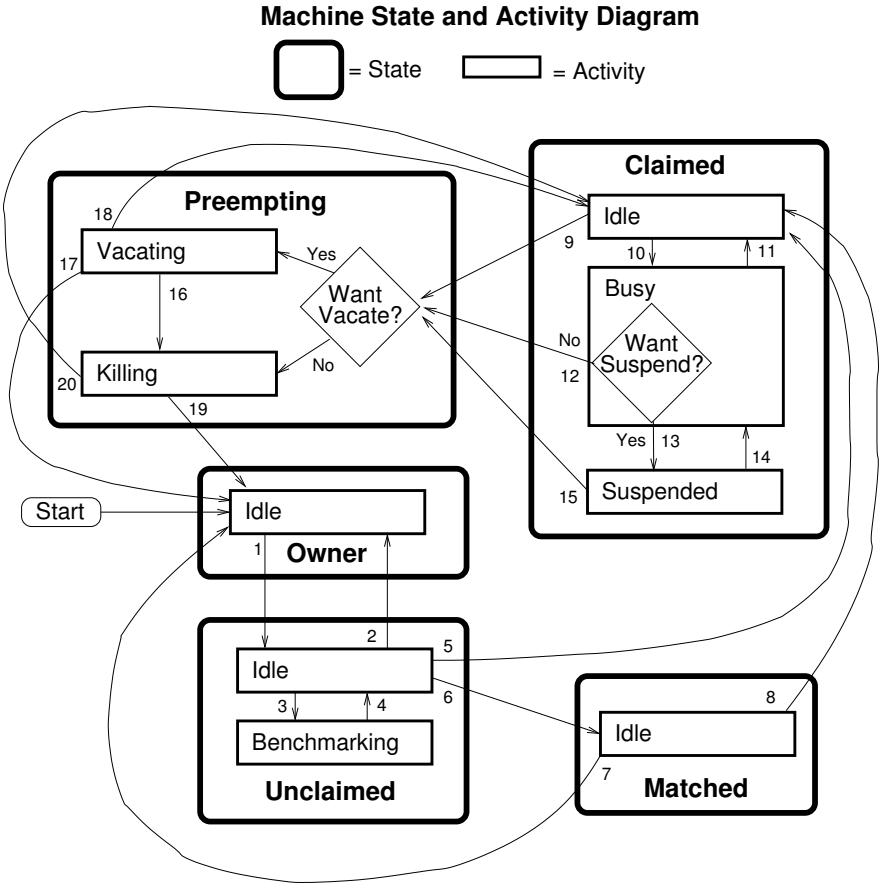


Figure 3.4: Machine States and Activities

Various expressions are used to determine when and if many of these state and activity transitions occur. Other transitions are initiated by parts of the Condor protocol (such as when the *condor_negotiator* matches a machine with a schedd). The following section describes the conditions that lead to the various state and activity transitions.

3.6.7 State and Activity Transitions

This section traces through all possible state and activity transitions within a machine and describes the conditions under which each one occurs. Whenever a transition occurs, Condor records when the machine entered its new activity and/or new state. These times are often used to write expressions that determine when further transitions occurred. For example, enter the Killing activity if a machine has been in the Vacating activity longer than a specified amount of time.

Owner State

When the startd is first spawned, the machine it represents enters the Owner state. The machine will remain in this state as long as the `START` expression locally evaluates to FALSE. If the `START` locally evaluates to TRUE or cannot be locally evaluated (it evaluates to UNDEFINED, transition **1** occurs and the machine enters the Unclaimed state.

As long as the `START` expression evaluates locally to FALSE, there is no possible request in the Condor system that could match it. The machine is unavailable to Condor and stays in the Owner state. For example, if the `START` expression is

```
START = KeyboardIdle > 15 * $(MINUTE) && Owner == "coltrane"
```

and if `KeyboardIdle` is 34 seconds, then the machine would remain in the Owner state. Owner is undefined, and `anything && FALSE` is FALSE.

If, however, the `START` expression is

```
START = KeyboardIdle > 15 * $(MINUTE) || Owner == "coltrane"
```

and `KeyboardIdle` is 34 seconds, then the machine leaves the Owner state and becomes Unclaimed. This is because `FALSE || UNDEFINED` is UNDEFINED. So, while this machine is not available to just anybody, if user coltrane has jobs submitted, the machine is willing to run them. Any other user's jobs have to wait until `KeyboardIdle` exceeds 15 minutes. However, since coltrane might claim this resource, but has not yet, the machine goes to the Unclaimed state.

While in the Owner state, the startd polls the status of the machine every `UPDATE_INTERVAL` to see if anything has changed that would lead it to a different state. This minimizes the impact on the Owner while the Owner is using the machine. Frequently waking up, computing load averages, checking the access times on files, computing free swap space take time, and there is nothing time critical that the startd needs to be sure to notice as soon as it happens. If the `START` expression evaluates to TRUE and five minutes pass before the startd notices, that's a drop in the bucket of high-throughput computing.

The machine can only transition to the Unclaimed state from the Owner state. It only does so when the `START` expression no longer locally evaluates to FALSE. In general, if the `START` expression locally evaluates to FALSE at any time, the machine will either transition directly to the

Owner state or to the Preempting state on its way to the Owner state, if there is a job running that needs preempting.

Unclaimed State

While in the Unclaimed state, if the `START` expression locally evaluates to FALSE, the machine returns to the Owner state by transition **2**.

When in the Unclaimed state, the `RunBenchmarks` expression is relevant. If `RunBenchmarks` evaluates to TRUE while the machine is in the Unclaimed state, then the machine will transition from the Idle activity to the Benchmarking activity (transition **3**) and perform benchmarks to determine `MIPS` and `KFLOPS`. When the benchmarks complete, the machine returns to the Idle activity (transition **4**).

The startd automatically inserts an attribute, `LastBenchmark`, whenever it runs benchmarks, so commonly `RunBenchmarks` is defined in terms of this attribute, for example:

```
BenchmarkTimer = (CurrentTime - LastBenchmark)
RunBenchmarks = $(BenchmarkTimer) >= (4 * $(HOUR))
```

Here, a macro, `BenchmarkTimer` is defined to help write the expression. This macro holds the time since the last benchmark, so when this time exceeds 4 hours, we run the benchmarks again. The startd keeps a weighted average of these benchmarking results to try to get the most accurate numbers possible. This is why it is desirable for the startd to run them more than once in its lifetime.

NOTE: `LastBenchmark` is initialized to 0 before benchmarks have ever been run. So, if you want the startd to run benchmarks as soon as the machine is Unclaimed (if it hasn't done so already), include a term for `LastBenchmark` as in the example above.

NOTE: If `RunBenchmarks` is defined and set to something other than FALSE, the startd will automatically run one set of benchmarks when it first starts up. To disable benchmarks, both at startup and at any time thereafter, set `RunBenchmarks` to FALSE or comment it out of the configuration file.

From the Unclaimed state, the machine can go to two other possible states: Matched or Claimed/Idle. Once the *condor_negotiator* matches an Unclaimed machine with a requester at a given schedd, the negotiator sends a command to both parties, notifying them of the match. If the schedd receives that notification and initiates the claiming procedure with the machine before the negotiator's message gets to the machine, the Match state is skipped, and the machine goes directly to the Claimed/Idle state (transition **5**). However, normally the machine will enter the Matched state (transition **6**), even if it is only for a brief period of time.

Matched State

The Matched state is not very interesting to Condor. Noteworthy in this state is that the machine lies about its `START` expression while in this state and says that `Requirements` are false to prevent being matched again before it has been claimed. Also interesting is that the startd starts a timer to make sure it does not stay in the Matched state too long. The timer is set with the `MATCH_TIMEOUT` configuration file macro. It is specified in seconds and defaults to 300 (5 minutes). If the schedd that was matched with this machine does not claim it within this period of time, the machine gives up, and goes back into the Owner state via transition **7**. It will probably leave the Owner state right away for the Unclaimed state again and wait for another match.

At any time while the machine is in the Matched state, if the `START` expression locally evaluates to FALSE, the machine enters the Owner state directly (transition **7**).

If the schedd that was matched with the machine claims it before the `MATCH_TIMEOUT` expires, the machine goes into the Claimed/Idle state (transition **8**).

Claimed State

The Claimed state is certainly the most complex state. It has the most possible activities and the most expressions that determine its next activities. In addition, the *condor_checkpoint* and *condor_vacate* commands affect the machine when it is in the Claimed state. In general, there are two sets of expressions that might take effect. They depend on the universe of the request: standard or vanilla. The standard universe expressions are the normal expressions. For example:

```
        WANT_SUSPEND            = True
        WANT_VACATE             = $(Activation-
Timer) > 10 * $(MINUTE)
        SUSPEND                 = $(KeyboardBusy) || $(CPUBusy)
        ...
```

The vanilla expressions have the string"_VANILLA" appended to their names. For example:

```
        WANT_SUSPEND_VANILLA    = True
        WANT_VACATE_VANILLA     = True
        SUSPEND_VANILLA         = $(KeyboardBusy) || $(CPUBusy)
        ...
```

Without specific vanilla versions, the normal versions will be used for all jobs, including vanilla jobs. In this manual, the normal expressions are referenced. The difference exists for the the resource owner that might want the machine to behave differently for vanilla jobs, since they cannot checkpoint. For example, owners may want vanilla jobs to remain suspended for longer than standard jobs.

While Claimed, the POLLING_INTERVAL takes effect, and the startd polls the machine much more frequently to evaluate its state.

If the machine owner starts typing on the console again, it is best to notice this as soon as possible to be able to start doing whatever the machine owner wants at that point. For SMP machines, if any virtual machine is in the Claimed state, the startd polls the machine frequently. If already polling one virtual machine, it does not cost much to evaluate the state of all the virtual machines at the same time.

In general, when the startd is going to take a job off a machine (usually because of activity on the machine that signifies that the owner is using the machine again), the startd will go through successive levels of getting the job out of the way. The first and least costly to the job is suspending it. This works for both standard and vanilla jobs. If suspending the job for a short while does not satisfy the machine owner (the owner is still using the machine after a specific period of time), the startd moves on to vacating the job. Vacating a job involves performing a checkpoint so that the work already completed is not lost. If even that does not satisfy the machine owner (usually because it is taking too long and the owner wants their machine back *now*), the final, most drastic stage is reached: killing. Killing is a quick death to the job, without a checkpoint. For vanilla jobs, vacating and killing are equivalent, although a vanilla job can request to have a specific *softkill signal* sent to it at vacate time so that the job itself can perform application-specific checkpointing.

The WANT_SUSPEND expression determines if the machine will evaluate the SUSPEND expression to consider entering the Suspended activity. The WANT_VACATE expression determines what happens when the machine enters the Preempting state. It will go to the Vacating activity or directly to Killing. If one or both of these expressions evaluates to FALSE, the machine will skip that stage of getting rid of the job and proceed directly to the more drastic stages.

When the machine first enters the Claimed state, it goes to the Idle activity. From there, it has two options. It can enter the Preempting state via transition **9** (if a *condor_vacate* arrives, or if the START expression locally evaluates to FALSE), or it can enter the Busy activity (transition **10**) if the schedd that has claimed the machine decides to activate the claim and start a job.

From Claimed/Busy, the machine can transition to three other state/activity pairs. The startd evaluates the WANT_SUSPEND expression to decide which other expressions to evaluate. If WANT_SUSPEND is TRUE, then the startd evaluates the SUSPEND expression. If SUSPEND is FALSE, then the startd will evaluate the PREEMPT expression and skip the Suspended activity entirely. By transition, the possible state/activity destinations from Claimed/Busy:

Claimed/Idle If the starter that is serving a given job exits (for example because the jobs completes), the machine will go to Claimed/Idle (transition **11**).

Preempting If WANT_SUSPEND is FALSE and the PREEMPT expression is TRUE, the machine enters the Preempting state (transition **12**). The other reason the machine would go from Claimed/Busy to Preempting is if the *condor_negotiator* matched the machine with a "better" match. This better match could either be from the machine's perspective using the RANK Expression above, or it could be from the negotiator's perspective due to a job with a higher user priority. In this case, WANT_VACATE is assumed to be TRUE, and the machine transitions to Preempting/Vacating.

Claimed/Suspended If both the WANT_SUSPEND and SUSPEND expressions evaluate to TRUE, the machine suspends the job (transition **13**).

If a *condor_checkpoint* command arrives, or the PeriodicCheckpoint expression evaluates to TRUE, there is no state change. The startd has no way of knowing when this process completes, so periodic checkpointing can not be another state. Periodic checkpointing remains in the Claimed/Busy state and appears as a running job.

From the Claimed/Suspended state, the following transitions may occur:

Claimed/Busy If the CONTINUE expression evaluates to TRUE, the machine resumes the job and enters the Claimed/Busy state (transition **14**).

Preempting If the PREEMPT expression is TRUE, the machine will enter the Preempting state (transition **15**).

Preempting State

The Preempting state is less complex than the Claimed state. There are two activities. Depending on the value of WANT_VACATE, a machine will be in the Vacating activity (if TRUE) or the Killing activity (if FALSE).

While in the Preempting state (regardless of activity) the machine advertises its Requirements expression as FALSE to signify that it is not available for further matches, either because it is about to transition to the Owner state, or because it has already been matched with one preempting match, and further preempting matches are disallowed until the machine has been claimed by the new match.

The main function of the Preempting state is to get rid of the starter associated with the resource. If the *condor_starter* associated with a given claim exits while the machine is still in the Vacating activity, then the job successfully completed its checkpoint.

If the machine is in the Vacating activity, it keeps evaluating the KILL expression. As soon as this expression evaluates to TRUE, the machine enters the Killing activity (transition **16**).

When the starter exits, or if there was no starter running when the machine enters the Preempting state (transition **9**), the other purpose of the Preempting state is completed: notifying the schedd that had claimed this machine that the claim is broken.

At this point, the machine enters either the Owner state by transition **17** (if the job was preempted because the machine owner came back) or the Claimed/Idle state by transition **18** (if the job was preempted because a better match was found). The machine enters the Killing activity, and it starts a timer, the length of which is defined by the KILLING_TIMEOUT macro. This macro is defined in seconds and defaults to 30. If this timer expires and the machine is still in the Killing activity, something has gone seriously wrong with the *condor_starter* and the startd tries to vacate the job immediately by sending SIGKILL to all of the *condor_starter*'s children, and then to the *condor_starter* itself.

Once the starter is gone and the schedd that had claimed the machine is notified that the claim is broken, the machine will either enter the Owner state by transition **19** (if the job was preempted because the machine owner came back) or the Claimed/Idle state by transition **20** (if the job was preempted because a better match was found).

3.6.8 State/Activity Transition Expression Summary

This section is a summary of the information from the previous sections. It serves as a quick reference.

START When TRUE, the machine is willing to spawn a remote Condor job.

RunBenchmarks While in the Unclaimed state, the machine will run benchmarks whenever TRUE.

MATCH_TIMEOUT If the machine has been in the Matched state longer than this value, it will transition to the Owner state.

WANT_SUSPEND If TRUE, the machine evaluates the SUSPEND expression to see if it should transition to the Suspended activity. If FALSE, the machine look at the PREEMPT expression.

SUSPEND If WANT_SUSPEND is TRUE, and the machine is in the Claimed/Busy state, it enters the Suspended activity if SUSPEND is TRUE.

CONTINUE If the machine is in the Claimed/Suspended state, it enter the Busy activity if CONTINUE is TRUE.

PREEMPT If the machine is either in the Claimed/Suspended activity, or is in the Claimed/Busy activity and WANT_SUSPEND is FALSE, the machine enters the Preempting state whenever PREEMPT is TRUE.

WANT_VACATE This is checked only when the PREEMPT expression is TRUE and the machine enters the Preempting state. If WANT_VACATE is TRUE, the machine enters the Vacating activity. If it is FALSE, the machine will proceed directly to the Killing activity.

KILL If the machine is the Preempting/Vacating state, it enters Preempting/Killing whenever KILL is TRUE.

KILLING_TIMEOUT If the machine is in the Preempting/Killing state for longer than KILLING_TIMEOUT seconds, the startd sends a SIGKILL to the *condor_starter* and all its children to try to kill the job as quickly as possible.

PERIODIC_CHECKPOINT If the machine is in the Claimed/Busy state and PERIODIC_CHECKPOINT is TRUE, the user's job begins a periodic checkpoint.

RANK If this expression evaluates to a higher number for a pending resource request than it does for the current request, the machine preempts the current request (enters the Preempting/Vacating state). When the preemption is complete, the machine enters the Claimed/Idle state with the new resource request claiming it.

3.6.9 Policy Settings

This section describes the default configuration policy and then provides examples of extensions to these policies.

Default Policy Settings

These settings are the default as shipped with Condor. They have been used for many years with no problems. The vanilla expressions are identical to the regular ones. (They are not listed here. If not defined, the standard expressions are used for vanilla jobs as well).

The following are macros to help write the expressions clearly.

StateTimer Amount of time in the current state.

ActivityTimer Amount of time in the current activity.

ActivationTimer Amount of time the job has been running on this machine.

LastCkpt Amount of time since the last periodic checkpoint.

NonCondorLoadAvg The difference between the system load and the Condor load (the load generated by everything but Condor).

BackgroundLoad Amount of background load permitted on the machine and still start a Condor job.

HighLoad If the `$(NonCondorLoadAvg)` goes over this, the CPU is considered too busy, and eviction of the Condor job should start.

StartIdleTime Amount of time the keyboard must to be idle before Condor will start a job.

ContinueIdleTime Amount of time the keyboard must to be idle before resumption of a suspended job.

MaxSuspendTime Amount of time a job may be suspended before more drastic measures are taken.

MaxVacateTime Amount of time a job may be checkpointing before we give up kill it outright.

KeyboardBusy A boolean expression that evaluates to TRUE when the keyboard is being used.

CPUIdle A boolean expression that evaluates to TRUE when the CPU is idle.

CPUBusy A boolean expression that evaluates to TRUE when the CPU is busy.

MachineBusy The CPU or the Keyboard is busy.

CPUIsBusy A boolean value set to the same value as `CPUBusy` .

CPUBusyTime The value 0 if CPUBusy is False; the time in seconds since CPUBusy became True.

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)
ActivationTimer = (CurrentTime - JobStart)
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)
MaxVacateTime           = 10 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
ConsoleBusy             = (ConsoleIdle  < $(MINUTE))
CPUIdle                 = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPUBusy                 = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy         = ($(KeyboardBusy) == False)
MachineBusy             = ($(CPUBusy) || $(KeyboardBusy)
```

Macros are defined to want to suspend jobs (instead of killing them) in the case of jobs that use little memory, when the keyboard is not being used, and for vanilla universe and PVM universe jobs. We want to gracefully vacate jobs which have been running for more than 10 minutes or are vanilla universe or PVM universe jobs.

```
WANT_SUSPEND       = ( $(SmallJob) || $(KeyboardNotBusy) \
                     || $(IsPVM) || $(IsVanilla) )
WANT_VACATE        = ( $(ActivationTimer) > 10 * $(MINUTE) \
                     || $(IsPVM) || $(IsVanilla) )
```

Finally, definitions of the actual expressions. Start a job if the keyboard has been idle long enough and the load average is low enough OR the machine is currently running a Condor job. Note that Condor would only run one job at a time. It just may prefer to run a different job, as defined by the machine rank or user priorities.

```
START        = ( (KeyboardIdle > $(StartIdleTime)) \
                 && ( $(CPUIdle) || \
                      (State != "Unclaimed" && State != "Owner")) )
```

Suspend a job if the keyboard has been touched. Alternatively, suspend if the CPU has been busy for more than two minutes and the job has been running for more than 90 seconds.

```
SUSPEND         = ( $(KeyboardBusy) || \
                  ( (CpuBusyTime > 2 * $(MINUTE)) \
                     && $(ActivationTimer) > 90 ) )
```

Continue a suspended job if the CPU is idle, the Keyboard has been idle for long enough, and the job has been suspended more than 10 seconds.

```
CONTINUE        = ( $(CPUIdle) && ($(ActivityTimer) > 10) \
                  && (KeyboardIdle > $(ContinueIdleTime)) )
```

There are two conditions that signal preemption. The first condition is if the job is suspended, but it has been suspended too long. The second condition is if suspension is not desired and the machine is busy.

```
PREEMPT         = ( ((Activity == "Suspended") && \
                    ($(ActivityTimer) > $(MaxSuspendTime))) \
                    || (SUSPEND && (WANT_SUSPEND == False)) )
```

Kill jobs that take too long leaving gracefully.

```
KILL            = $(ActivityTimer) > $(MaxVacateTime)
```

Finally, specify periodic checkpointing. For jobs smaller than 60 Mbytes, do a periodic checkpoint every 6 hours. For larger jobs, only checkpoint every 12 hours.

```
PERIODIC_CHECKPOINT     = ( (ImageSize < 60000) && \
                            ($(LastCkpt) > (6 * $(HOUR))) ) || \
                          ( $(LastCkpt) > (12 * $(HOUR)) )
```

At UW-Madison, we have a fast network. We simplify our expression considerably to

```
PERIODIC_CHECKPOINT     = $(LastCkpt) > (3 * $(HOUR))
```

For reference, the entire set of policy settings are included once more without comments:

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
```

```
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)
ActivationTimer = (CurrentTime - JobStart)
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)
MaxVacateTime           = 10 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
ConsoleBusy             = (ConsoleIdle  < $(MINUTE))
CPUIdle                 = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPUBusy                 = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy         = ($(KeyboardBusy) == False)
MachineBusy             = ($(CPUBusy) || $(KeyboardBusy)

WANT_SUSPEND       = ( $(SmallJob) || $(KeyboardNotBusy) \
                     || $(IsPVM) || $(IsVanilla) )
WANT_VACATE        = ( $(ActivationTimer) > 10 * $(MINUTE) \
                     || $(IsPVM) || $(IsVanilla) )
START        = ( (KeyboardIdle > $(StartIdleTime)) \
                && ( $(CPUIdle) || \
                     (State != "Unclaimed" && State != "Owner")) )
SUSPEND         = ( $(KeyboardBusy) || \
                  ( (CpuBusyTime > 2 * $(MINUTE)) \
                    && $(ActivationTimer) > 90 ) )
CONTINUE        = ( $(CPUIdle) && ($(ActivityTimer) > 10) \
                  && (KeyboardIdle > $(ContinueIdleTime)) )
PREEMPT         = ( ((Activity == "Suspended") && \
                    ($(ActivityTimer) > $(MaxSuspendTime))) \
                    || (SUSPEND && (WANT_SUSPEND == False)) )
KILL            = $(ActivityTimer) > $(MaxVacateTime)
PERIODIC_CHECKPOINT     = ( (ImageSize < 60000) && \
                            ($(LastCkpt) > (6 * $(HOUR))) ) || \
                          ( $(LastCkpt) > (12 * $(HOUR)) )
```

Policy Examples

This example shows how the default macros can be used to set up a machine for testing. Suppose we want the machine to behave normally, except if user coltrane submits a job. In that case, we want that job to start regardless of what is happening on the machine. We do not want the job suspended, vacated or killed. This is reasonable if we know coltrane is submitting very short running programs

testing purposes. The jobs should be executed right away. This works with any machine (or the whole pool, for that matter) by adding the following 5 expressions to the existing configuration:

```
START      = ($(START)) || Owner == "coltrane"
SUSPEND    = ($(SUSPEND)) && Owner != "coltrane"
CONTINUE   = $(CONTINUE)
PREEMPT    = ($(PREEMPT)) && Owner != "coltrane"
KILL       = $(KILL)
```

Notice that there is nothing special in either the `CONTINUE` or `KILL` expressions. If Coltrane's jobs never suspend, they never look at `CONTINE`. Similarly, if they never preempt, they never look at `KILL`.

Condor can be configured to only run jobs at certain times of the day. In general, we discourage configuring a system like this, since you can often get lots of good cycles out of machines, even when their owners say "I'm always using my machine during the day." However, if you submit mostly vanilla jobs or other jobs that cannot checkpoint, it might be a good idea to only allow the jobs to run when you know the machines will be idle and when they will not be interrupted.

To configure this kind of policy, you should use the `ClockMin` and `ClockDay` attributes, defined in section 3.6.1 on "Startd ClassAd Attributes". These are special attributes which are automatically inserted by the *condor_startd* into its ClassAd, so you can always reference them in your policy expressions. `ClockMin` defines the number of minutes that have passed since midnight. For example, 8:00am is 8 hours after midnight, or 8 * 60 minutes, or 480. 5:00pm is 17 hours after midnight, or 17 * 60, or 1020. `ClockDay` defines the day of the week, Sunday = 0, Monday = 1, and so on.

To make the policy expressions easy to read, we recommend using macros to define the time periods when you want jobs to run or not run. For example, assume regular "work hours" at your site are from 8:00am until 5:00pm, Monday through Friday:

```
WorkHours = ( (ClockMin >= 480 && ClockMin < 1020) && \
              (ClockDay > 0 && ClockDay < 6) )
AfterHours = ( (ClockMin < 480 || ClockMin >= 1020) || \
               (ClockDay == 0 || ClockDay == 6) )
```

Of course, you can fine-tune these settings by changing the definition of `AfterHours` and `WorkHours` for your site.

Assuming you are using the default policy expressions discussed above, there are only a few minor changes required to force Condor jobs to stay off of your machines during work hours:

```
# Only start jobs after hours.
START = $(AfterHours) && $(CPUIdle) && KeyboardI-
dle > $(StartIdleTime)
```

```
# Consider the machine busy during work hours, or if the key-
board or
# CPU are busy.
MachineBusy = ( $(WorkHours) || $(CPUBusy) || $(KeyboardBusy) )
```

By default, the MachineBusy macro is used to define the SUSPEND and PREEMPT expressions. If you have changed these expressions at your site, you will need to add $(WorkHours) to your SUSPEND and PREEMPT expressions as appropriate.

Depending on your site, you might also want to avoid suspending jobs during work hours, so that in the morning, if a job is running, it will be immediately preempted, instead of being suspended for some length of time:

```
WANT_SUSPEND = $(AfterHours)
```

3.6.10 Differences from the Version 6.0 Policy Settings

This section describes how the current policy expressions differ from the policy expressions in previous versions of Condor. If you have never used Condor version 6.0 or earlier, or you never looked closely at the policy settings, skip this section.

In summary, there is no longer a VACATE expression, and the KILL expression is not evaluated while a machine is claimed. There is a PREEMPT expression which describes the conditions when a machine will move from the Claimed state to the Preempting state. Once a machine is transitioning into the Preempting state, the WANT_VACATE expression controls whether the job should be vacated with a checkpoint or directly killed. The KILL expression determines the transition from Preempting/Vacating to Preempting/Killing.

In previous versions of Condor, the KILL expression handled three distinct cases (the transitions from Claimed/Busy, Claimed/Suspended and Preempting/Vacating), and the VACATE expression handled two cases (the transitions from Claimed/Busy and Claimed/Suspended). In the current version of Condor, PREEMPT handles the same two cases as the previous VACATE expression, but the KILL expression handles one case. Very complex policies can now be specified using all of the default expressions, only tuning the WANT_VACATE and WANT_SUSPEND expressions. In previous versions, heavy use of the WANT_* expressions caused a complex KILL expression.

3.7 Security In Condor

This section describes various aspects of security within Condor.

3.7.1 UIDs in Condor

On a Unix system, UIDs (User IDentification numbers) form part of an operating system's tools for maintaining access control. Each executing program has a UID, a unique identifier of a user executing the program. This is also called the real UID. A common situation has one user executing the program owned by another user. Many system commands work this way, with a user (corresponding to a person) executing a program belonging to (owned by) root. Since the program may require privileges that root has which the user does not have, a special bit in the program's protection specification (a setuid bit) allows the program to run with the UID of the program's owner, instead of the user that executes the program. This UID of the program's owner is called an effective UID.

Condor works most smoothly when its daemons run as root. The daemons then have the ability to switch their effective UIDs at will. When the daemons run as root, they normally leave their effective UID and GID (Group IDentification) to be those of user and group condor. This allows access to the log files without changing the ownership of the log files. It also allows access to these files when the user condor's home directory resides on an NFS server. root can not normally access NFS files.

On a machine where a job is submitted, the *condor_schedd* daemon changes its effective UID to root such that it has the capability to start up a *condor_shadow* daemon for the job. Before a *condor_shadow* daemon is created, the *condor_schedd* daemon switches back to root, so that it can start up the *condor_shadow* daemon with the (real) UID of the user who submitted the job. Since the *condor_shadow* runs as the owner of the job, all remote system calls are performed under the owner's UID and GID. This ensures that as the job executes, it can access only files that its owner could access if the job were running locally, without Condor. On the machine where the job executes, the job runs as user nobody, to help ensure that the job cannot access local resources or do harm.

What if Condor is not run as root?

Condor can also function on all platforms by starting up as user condor. Since user condor does not have the ability to switch UID or GID, all daemons run with both the UID and GID belonging to user condor. The *condor_shadow* daemon and the job's executable also run as user condor. This has the effect that the job can access only the files and directories that are accessible to the user condor on the machine where the job was submitted. Owners of jobs must make their input readable to the user condor. A job's output must be placed in a directory that is writable by the user condor as well. In practice, this means creating world-writable directories for output from Condor jobs. This creates a potential security risk, in that any user on the machine where the job is submitted can alter the data, remove it, or do other undesirable things. It is acceptable in an environment where users can trust other users.

On platforms where root access is not needed, Condor can even function without a UID or GID of the user condor. A directory to act as the condor home directory is still required, containing the configuration files, spool, execute and log directories. This home directory is not technically the home directory of any user. In this case, a user condor may or may not even exist, but the directory is still referred to as the condor home directory. If the user condor does not exist, use

the CONDOR_CONFIG environment variable such that all Condor daemons and tools can find their configuration file (which in turn defines the locations of other needed files and directories), or place a configuration file in `/etc/condor/condor_config`. The Condor daemons can then be started up by whatever UID and GID has access to the local `condor` directory. Normally, users without root access who wish to use Condor on their machines create a `condor` home directory somewhere within their own accounts and start up the daemons (to run with the UID of the user). As in the case where the daemons run as user condor, there is no ability to switch UIDs or GIDs. The daemons run as the UID and GID of the user who started them. On a machine where jobs are submitted, the *condor_shadow* daemons all run as this same user. However, if other users on the machine are using Condor in this environment, the *condor_shadow* daemons for these other users' jobs execute with the UID of the user who started the daemons. This is a security risk, since the Condor job of the other user has access to all the files and directories of the user who started the daemons. Some installations have this level of trust, but others do not. Where this level of trust does not exist, it is best to set up a condor account and group, or to have each user start up their own Personal Condor submit installation.

When a machine is an execution site for a Condor job, the Condor job executes with the UID of the user who started the *condor_startd* daemon. This is also potentially a security risk, which is why we do not recommend starting up the execution site daemons as a regular user. Use either root or a user (such as the user condor) that exists only to run Condor jobs.

What directory does a job run in?

Any executing process has a notion of its current working directory (cwd), the directory that acts as the base for all file system access. There are two sides to any Condor job: the submit side and the execution side. This implies that there are two cwds. On the submit side, the owner's cwd sets a default cwd as a job is submitted. The cwd can be changed with a command in the submit description file. Since many jobs can be submitted at the same time, the commands are flexible in order to set the cwd individually for each job if desired. This submit side cwd remains for the entire life of a job. The submit side cwd is also used as the cwd of the *condor_shadow* daemon. Since file system access for the job goes through the *condor_shadow* daemon, all accesses behave as if they were executing without Condor.

There is also a cwd associated with the Condor job on the execution machine. It is set to the `execute` subdirectory of Condor's home directory. This directory is world-writable, since a Condor job usually runs as user nobody. Normally, the executable would never access this directory, since all I/O system calls are passed back to the *condor_shadow* daemon on the submit machine. However, in the event that the job that creates a core dump, the cwd on the execute machine needs to be accessible by the job so that it can write the core file. The core file is moved back to the submit machine, and the *condor_shadow* daemon is informed. The *condor_shadow* daemon sends e-mail to the job owner announcing the crash and providing a pointer to the core file, then residing in the submit side cwd.

3.7.2 Running Condor as Non-Root

While we strongly recommend starting up the Condor daemons as root, we understand that it is not always possible to do so. The main problems appear if you have one Condor installation shared by many users on a single machine, or if you are setting up machines to only execute Condor jobs. If you are setting up a submit-only installation for a single user, then there is no need for (or benefit from) running as root.

What follows are the effects on the various parts of Condor of running both with and without root access.

condor_startd If you're setting up a machine to run Condor jobs and don't start the *condor_startd* as root, you're basically relying on the goodwill of your Condor users to agree to the policy you configure the startd to enforce as far as starting, suspending, vacating and killing Condor jobs under certain conditions. If you run as root, however, you can enforce these policies regardless of malicious users. By running as root, the Condor daemons run with a different UID than the Condor job that gets started (since the user's job is started as either the UID of the user who submitted it, or as user "nobody", depending on the `UID_DOMAIN` settings). Therefore, the Condor job cannot do anything to the Condor daemons. If you don't start the daemons as root, all processes started by Condor, including the end user's job, run with the same UID (since you can't switch UIDs unless you're root). Therefore, a user's job could just kill the *condor_startd* and *condor_starter* as soon as it starts up and by doing so, avoid getting suspended or vacated when a user comes back to the machine. This is nice for the user, since they get unlimited access to the machine, but awful for the machine owner or administrator. If you trust the users submitting jobs to Condor, this might not be a concern. However, to ensure that the policy you choose is effectively enforced by Condor, the *condor_startd* should be started as root.

In addition, some system information cannot be obtained without root access on some platforms (such as load average on IRIX). As a result, when running without root access, the *condor_startd* has to call other programs (for example, "uptime") to get this information. This is much less efficient than getting the information directly from the kernel (which is what we do if we're running as root). On Linux and Solaris, we can get this information directly without root access, so this is not a concern on those platforms.

If you can't have all of Condor running as root, at least consider whether you can install the *condor_startd* as setuid root. That would solve both of these problems. If you can't do that, you could also install it as a setgid sys or kmem program (depending on whatever group has read access to `/dev/kmem` on your system) and that would at least solve the system information problem.

condor_schedd The biggest problem running the schedd without root access is that the *condor_shadow* processes which it spawns are stuck with the same UID the *condor_schedd* has. This means that users submitting their jobs have to go out of their way to grant write access to user or group condor (or whoever the schedd is running as) for any files or directories their jobs write or create. Similarly, read access must be granted to their input files.

You might consider installing *condor_submit* as a setgid condor program so that at least

the `stdout`, `stderr` and `UserLog` files get created with the right permissions. If *condor_submit* is a setgid program, it will automatically set it's umask to 002, so that creates group-writable files. This way, the simple case of a job that just writes to `stdout` and `stderr` will work. If users have programs that open their own files, they'll have to know to set the right permissions on the directories they submit from.

condor_master The *condor_master* is what spawns the *condor_startd* and *condor_schedd*, so if want them both running as root, you should have the master run as root. This happens automatically if you start the master from your boot scripts.

condor_negotiator

condor_collector There is no need to have either of these daemons running as root.

condor_kbdd On platforms that need the *condor_kbdd* (Digital Unix and IRIX) the *condor_kbdd* has to run as root. If it is started as any other user, it will not work. You might consider installing this program as a setuid root binary if you can't run the *condor_master* as root. Without the *condor_kbdd*, the startd has no way to monitor mouse activity at all, and the only keyboard activity it will notice is activity on ttys (such as xterms, remote logins, etc).

3.7.3 Security Configuration

Condor provides support for strong authentication, encryption, integrity assurance, as well as authorization. Most of these security features are not visible to the user (one who submits jobs). They are enabled by site administrators through the use of configuration macros. This section describes the authentication, encryption, integrity assurance, as well as authorization configuration macros provided by Condor.

Authentication provides an assurance of the identity of one of the communicating parties. Mutual authentication provides an assurance of the identities of both of the communicating parties. Encoding information such that its contents is not easily decipherable by outsiders is called encryption. The integrity of a message is assured when any form of tampering with the message can be detected. With integrity support, nothing in the message can be added, deleted, or modified without being detected.

When Condor is installed, default configuration settings use no authentication, encryption, or integrity checks, nor are authorization checks provided. This allows newer versions of Condor with security features to work or interact with previous versions without security support. An administrator must modify the configuration settings to enable the security features.

Inside Condor, daemons need to communicate with each other; furthermore, various tools provided by Condor may also require communication with Condor daemons. All these communications can be made more secure through the proper usage of authentication, encryption, and integrity checks. Authorization can be used to protect resources in a Condor pool.

When a daemon receives a request, it uses the client's security configuration information together with its own configuration settings to decide upon the security aspects of the communication. This

can be considered a negotiation between the client and the daemon. The daemon replies to the client with a set of reconciled policies that controls the communication, including authentication, encryption, and integrity algorithms.

If the daemon determines that authentication is required, then the client must follow the chosen authentication protocol. After the required authentication, the client can send its request to the daemon. The daemon identifies the access level required for the specific request, and it checks the configuration settings to determine if the client has the required access level. If the client has the required access level, permission is granted, and the request is serviced.

Access Level Descriptions

Authorization is granted based on specified access levels. Access levels are granted for users by configuration settings. The following describes the various access levels provided by Condor.

READ This access level access can obtain or read information about Condor. Examples that require only `READ` access are viewing the status of the pool, checking the job queue(s), or viewing user permissions. `READ` access does not allow any changes, and it does not allow job submission.

WRITE This access level is required to send (write) information to Condor. Note that `WRITE` access does not include `READ` access. They are separate access levels. Job submission requires `WRITE` access.

ADMINISTRATOR This access level has additional Condor administrator rights to the pool. It includes the ability to change user priorities (with the command *condor_userprio -set*), and the ability to turn Condor on and off (as with the commands *condor_on* and *condor_off*).

CONFIG This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, this level of access can change any configuration parameters of a Condor pool, except those specified in the `condor_config.root` configuration file.

DAEMON This access level is only used by Condor daemons for internal exchange of requests. An example is the message sent from the *condor_startd* daemon to the *condor_schedd* daemon in order to claim a resource. In general, this level of access should be granted to all Condor daemons, implying that this level of access should be granted to the id under which the Condor daemons are run.

OWNER This level of access is required for commands that the owner of a machine (any local user) should be able to use, in addition to the Condor administrators. An example that requires the `OWNER` access level is the *condor_vacate* command. The command causes the *condor_startd* daemon to vacate any Condor job currently running on a machine. The owner of that machine should be able to cause the removal of a job running on the machine.

NEGOTIATOR This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the

pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run.

Security Macro Names and Values

The configuration macro names follow a pattern. Each of the names starts with the string `SEC_`. This string is followed by a string that describes an access level. The levels are

```
DEFAULT
READ
WRITE
ADMIN
DAEMON
CONFIG
OWNER
NEGOTIATOR
CLIENT
```

Both `DEFAULT` and `CLIENT` from this list are not access levels. The `DEFAULT` is used to define all levels of access for a specific configuration variable when individual levels are not specified. The `CLIENT` is used to define the client's requirements and preferences in a secure communication.

Still within the name of a configuration macro, the access level is followed by another underscore character and then a string describing the communication type. The communication types are

```
AUTHENTICATION
ENCRYPTION
INTEGRITY
```

Two examples of the complete macro names are `SEC_ADMIN_AUTHENTICATION` and `SEC_DEFAULT_INTEGRITY`.

Each configuration variable would be defined with one of four predefined values. The values are

```
REQUIRED
PREFERRED
OPTIONAL
NEVER
```

For example, a line in a daemon's configuration file to require all interactions to be encrypted is

```
SEC_DEFAULT_ENCRYPTION = REQUIRED
```

A second example from a configuration file specifies that all requests (from a client) that would require a `WRITE` access level be authenticated is

```
SEC_WRITE_AUTHENTICATION = REQUIRED
```

A daemon uses both the client's security configuration together with its own configuration to choose the communication setting for authentication, encryption, or integrity check. The following table defines whether or not (Yes or No) a communication setting will be used, or if the setting cannot work (Fail) due to a mismatch in the configuration settings.

client	daemon	Yes/No/Fail
REQUIRED	REQUIRED	Yes
REQUIRED	PREFERRED	Yes
REQUIRED	OPTIONAL	Yes
REQUIRED	NEVER	Fail
PREFERRED	REQUIRED	Yes
PREFERRED	PREFERRED	Yes
PREFERRED	OPTIONAL	Yes
PREFERRED	NEVER	No
OPTIONAL	REQUIRED	Yes
OPTIONAL	PREFERRED	Yes
OPTIONAL	OPTIONAL	No
OPTIONAL	NEVER	No
NEVER	REQUIRED	Fail
NEVER	PREFERRED	No
NEVER	OPTIONAL	No
NEVER	NEVER	No

Authentication

Authentication provides an assurance of an identity. Through configuration macros, both the client and the daemon can specify whether authentication is required.

The client uses one of two macros to configure authentication:

```
SEC_DEFAULT_AUTHENTICATION
SEC_CLIENT_AUTHENTICATION
```

For the daemon, there are eight macros to configure authentication:

```
SEC_DEFAULT_AUTHENTICATION
SEC_READ_AUTHENTICATION
SEC_WRITE_AUTHENTICATION
SEC_ADMIN_AUTHENTICATION
SEC_DAEMON_AUTHENTICATION
SEC_CONFIG_AUTHENTICATION
SEC_OWNER_AUTHENTICATION
SEC_NEGOTIATOR_AUTHENTICATION
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_WRITE_AUTHENTICATION = REQUIRED
```

signifies that the daemon must authenticate the client for any communication that requires the `WRITE` access level. If the daemon's configuration contains

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

and does not contain any other security configuration for `AUTHENTICATION`, then this default defines the daemon's needs for authentication over all access levels. Where a specific macro is present, its value takes precedence over any default given.

If authentication is to be done, then the communicating parties must negotiate a mutually acceptable method of authentication to be used. A list of acceptable methods may be provided by the client, using the macros

```
SEC_DEFAULT_AUTHENTICATION_METHODS
SEC_CLIENT_AUTHENTICATION_METHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

```
SEC_DEFAULT_AUTHENTICATION_METHODS
SEC_READ_AUTHENTICATION_METHODS
SEC_WRITE_AUTHENTICATION_METHODS
SEC_ADMIN_AUTHENTICATION_METHODS
SEC_DAEMON_AUTHENTICATION_METHODS
SEC_CONFIG_AUTHENTICATION_METHODS
SEC_OWNER_AUTHENTICATION_METHODS
SEC_NEGOTIATOR_AUTHENTICATION_METHODS
```

The methods are given as a comma-separated list of acceptable values. These variables list the authentication methods that are available to be used. The ordering of the list gives preference; the first item in the list indicates the highest preference. The values will be

```
KERBEROS
FS
GSS_AUTHENTICATION
CLAIMTOBE
ANONYMOUS
NTSSPI
```

As an example, the macro

```
SEC_DEFAULT_AUTHENTICATION_METHODS = KERBEROS, GSS_AUTHENTICATION
```

indicates that either Kerberos or X.509 authentication may be used, but Kerberos is preferred over X.509.

If Kerberos is used for authentication, then the optional configuration variable `KERBEROS_MAP_FILE` may be used to define a path to an administrator-maintained file that contains Kerberos domain (called a realm) to Condor UID domain mapping. The configuration syntax is

```
KERBEROS_MAP_FILE = /path/to/etc/condor.kmap
```

Lines within the map file have the syntax

```
KerberosRealm = UIDDomain
```

If no map file is specified, then Condor assumes that the Kerberos realm is the same as the Condor UID domain. Here are two lines from a map file to use as an example:

```
CS.WISC.EDU   = cs.wisc.edu
ENGR.WISC.EDU = ee.wisc.edu
```

Encryption

Encryption provides privacy support between two communicating parties. Through configuration macros, both the client and the daemon can specify whether encryption is required for further communication.

The client uses one of two macros to enable or disable encryption:

```
SEC_DEFAULT_ENCRYPTION
SEC_CLIENT_ENCRYPTION
```

For the daemon, there are eight macros to enable or disable encryption:

```
SEC_DEFAULT_ENCRYPTION
SEC_READ_ENCRYPTION
SEC_WRITE_ENCRYPTION
SEC_ADMIN_ENCRYPTION
SEC_DAEMON_ENCRYPTION
SEC_CONFIG_ENCRYPTION
SEC_OWNER_ENCRYPTION
SEC_NEGOTIATOR_ENCRYPTION
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_DAEMON_ENCRYPTION = REQUIRED
```

signifies that any daemon to daemon communication must be encrypted. If a daemon's configuration contains

```
SEC_DEFAULT_ENCRYPTION = REQUIRED
```

and does not contain any other security configuration for ENCRYPTION, then this default defines the daemon's needs for encryption over all access levels. Where a specific macro is present, its value takes precedence over any default given.

If encryption is to be done, then the communicating parties must find (negotiate) a mutually acceptable method of encryption to be used. A list of acceptable methods may be provided by the client, using the macros

```
SEC_DEFAULT_CRYPTO_METHODS
SEC_CLIENT_CRYPTO_METHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

```
SEC_DEFAULT_CRYPTO_METHODS
SEC_READ_CRYPTO_METHODS
SEC_WRITE_CRYPTO_METHODS
SEC_ADMIN_CRYPTO_METHODS
SEC_DAEMON_CRYPTO_METHODS
SEC_CONFIG_CRYPTO_METHODS
SEC_OWNER_CRYPTO_METHODS
SEC_NEGOTIATOR_CRYPTO_METHODS
```

The methods are given as a comma-separated list of acceptable values. These variables list the encryption methods that are available to be used. The ordering of the list gives preference; the first item in the list indicates the highest preference. Possible values are

```
3DES
BLOWFISH
```

Integrity Checks

An integrity check assures that the messages between communicating parties have not been tampered with. Any change, such as addition, modification, or deletion can be detected. Through configuration macros, both the client and the daemon can specify whether an integrity check is required of further communication.

The client uses one of two macros to enable or disable an integrity check:

```
SEC_DEFAULT_INTEGRITY
SEC_CLIENT_INTEGRITY
```

For the daemon, there are eight macros to enable or disable an integrity check:

```
SEC_DEFAULT_INTEGRITY
SEC_READ_INTEGRITY
SEC_WRITE_INTEGRITY
SEC_ADMIN_INTEGRITY
SEC_DAEMON_INTEGRITY
SEC_CONFIG_INTEGRITY
SEC_OWNER_INTEGRITY
SEC_NEGOTIATOR_INTEGRITY
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_DAEMON_INTEGRITY = REQUIRED
```

signifies that any daemon to daemon communication must have its integrity assured. If a daemon's configuration contains

```
SEC_DEFAULT_INTEGRITY = REQUIRED
```

and does not contain any other security configuration for `INTEGRITY`, then this default defines the daemon's needs for integrity checks over all access levels. Where a specific macro is present, its value takes precedence over any default given.

There is currently only one method used for integrity checking: a signed MD5 checksum. Its use is implied whenever integrity checks occur. If more methods are implemented, then there will be further macros to allow both the client and the daemon to specify which methods are acceptable.

Example of Daemon-Side Security Configuration

A configuration file is provided when Condor is installed. No security features are enabled within the configuration as distributed. Included as comments within the configuration file is an example suggesting settings that enable security features. Here is that example of the daemon-side portion.

```
SEC_DEFAULT_AUTHENTICATION=REQUIRED
SEC_DEFAULT_ENCRYPTION=REQUIRED
SEC_DEFAULT_INTEGRITY=REQUIRED

SEC_DEFAULT_AUTHENTICATION_METHODS = KERBEROS
SEC_DEFAULT_CRYPTO_METHODS = 3DES

KERBEROS_MAP_FILE = /path/to/etc/condor.kmap
```

This set of configuration macros forces all security features to be used at all times. All communication is authenticated (using Kerberos), and all communication is both encrypted (using triple DES) and has its integrity checked to make sure that messages are not modified or corrupted.

Note that this example configuration requires that all Condor daemons be version 6.3.3 or later, since previous versions will not have the ability to do secure communication.

3.7.4 Authentication

Authentication provides an assurance of an identity. Condor supports authenticated communications by using any one of several supported methods. The details of Condor's requirements for using X.509 authentication and Kerberos authentication are given below.

X.509 Authentication

The Globus GSI (Grid Security Infrastructure) protocol provides an avenue for Condor to do PKI-based authentication using X.509 certificates.

Several steps are required to enable this authentication.

X.509 Certificates PKI-based authentication between Condor daemons requires a valid X.509 certificate from a trusted CA (Certification Authority). The certificate may be a single host certificate, and all Condor daemons on the same machine may share the same certificate. The directory where the certificate resides, along with other related files such as one containing the private key, are utilized by Condor when doing authentication.

Condor locates the daemon's certificate through the use of a configuration macro. Note that the certificate can be shared by all Condor daemons running on a machine. The certificate can also be copied to other machines, where local copies are necessary. The certificates must be protected by access rights to files, since the password file is not encrypted.

Condor users (those who submit jobs) can also use the X.509-based authentication. These users must also have valid X.509 certificates signed by a trusted CA. For use when submitting jobs, the certificate is used to generate a proxy: a combination of certificate and password that is valid only for a specific (and usually short) time period. A proxy is used so that the user does not need to be present (to give a password) when a job is executed.

Condor Configuration To use X.509 certificates for authentication, the Condor configuration file must have the following configuration macros set. This setting is an example.

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
SEC_DEFAULT_AUTHENTICATION_METHODS = GSS_AUTHENTICATION
X509_DIRECTORY = /path/to/daemon/certificatedirectory
CONDOR_GATEKEEPER = /C=US/O=Condor/O=University of Wisconsin
/OU=Computer Sciences Department/CN=condor@cs.wisc.edu
```

The SEC_DEFAULT_AUTHENTICATION macro specifies that authentication is required for all communications. This single macro covers all communications, but could be replaced with a set of macros that require authentication for only specific communications. See section 3.7.3 for details on the more specific configuration variables used for authentication.

The GSS_AUTHENTICATION method specifies the use of the X.509 protocol. If this is the only method listed, then X.509 will be the only protocol used. If other methods are acceptable, then placing this method first within the list will cause Condor to give preference to this method over others.

The macro X509_DIRECTORY must be specified, so that Condor knows where to look for the daemon's certificate. This path may be a directory or a shared file system such as AFS. Alternatively, this path name can point to local copies of the certificate stored in a local file system.

The macro CONDOR_GATEKEEPER configuration macro provides daemons with a distinguished name to use for X.509 authentication. This name is specified with the following format

```
CONDOR_GATEKEEPER = /C=?/O=?/O=?/OU=?/CN=<daemon_name@domain>
```

A complete example that has the question marks filled in and the daemon's user name filled in is given in the example above.

Condor will also need a way to map an X.509 distinguished name to a Condor user id. This is done in an administrator-maintained file called an X.509 map file, mapping from X509 Distinguished Name (DN) to Condor user id. It is similar to a Globus Gridmap file. Entries (lines) in the file each contain two items. The first item in an entry is the distinguished name given in an X.509 certificate. The second item is the Condor user id, given as a fully qualified name. Note that the two items in an entry are separated by tab character(s), not spaces. An example of two entries in an X.509 map file:

```
V 020406001927Z 01 unknown /C=US/O=Condor/O=University
of Wisconsin/ OU=Computer Sciences Department/CN=condor@cs.wisc.edu
condor@cs.wisc.edu
V 020406002049Z 02 unknown /C=US/O=Condor/O=University
of Wisconsin/OU=Computer Sciences Department/CN=jsmith@cs.wisc.edu
jsmith@cs.wisc.edu
```

Condor expects the map file to be in the location given by

```
$(X509_DIRECTORY)/certdir/index.txt
```

If the map file is not in this location, a symbolic link may be used to point to the actual map file.

User environment The Condor user (one who submits jobs) must create a proxy prior to authentication. The user creates a proxy using the program *grid-proxy-init*. This program needs to know the location of the user's certificate directory. This is done by setting an environment variable, called `X509_CERT_DIR`. For example, if the X509 directory is `/test/myX509`, then `X509_CERT_DIR` should be set to `/test/myX509/certdir`.

After running *grid-proxy-init*, an environment variable called `X509_USER_PROXY` may be set to give the location where the proxy is stored. By default, *grid-proxy-init* will place the proxy in the `/tmp` directory with the file name being determined by the format:

```
/tmp/x509_uXXXX
```

The specific file name is given by substituting the `XXXX` characters with the UID of the user. The `X509_USER_PROXY` environment variable is optional, since Condor looks in the default location if the environment variable is not set.

Kerberos Authentication

This section has not yet been written

3.7.5 Authorization

Authorization protects resource usage by granting or denying access requests made to the resources. It defines who is allowed to do what.

Authorization is defined in terms of users. An initial implementation provided authorization based on hosts (machines), while the current implementation relies on user-based authorization. Section 3.7.5 on Setting Up IP/Host-Based Security in Condor describes the previous implementation. This IP/Host-Based security still exists, and it can be used, but significantly stronger and more flexible security can be achieved with the newer authorization based on fully qualified user names.

User-based Authorization

Unlike authentication, encryption, and integrity checks, which can be configured by both client and server, authorization is used only by a server. The authorization portion of the security of a Condor pool is based on a set of configuration macros. The macros list which user/daemon will be authorized to issue what request given a specific access level.

These configuration macros define a set of users that will be allowed to (or denied from) carrying out various Condor commands. Each access level may have its own list of authorized users. A complete list of the authorization macros:

```
ALLOW_READ
ALLOW_WRITE
ALLOW_ADMINISTRATOR
ALLOW_CONFIG
ALLOW_DAEMON
ALLOW_OWNER
ALLOW_NEGOTIATOR
DENY_READ
DENY_WRITE
DENY_ADMINISTRATOR
DENY_CONFIG
DENY_DAEMON
DENY_OWNER
DENY_NEGOTIATOR
```

Each macro is defined by a comma-separated list of fully qualified users. Each fully qualified user is described using the following format:

```
username@domain/hostname
```

The information to the left of the slash character describes a user within a domain. The information to the right of the slash character describes a machine from which the user would be issuing a command. An example is

```
zmiller@cs.wisc.edu/bird.cs.wisc.edu
```

Within the format, wildcard characters (the asterisk, *) are allowed. The use of wildcards is limited to one wildcard on either side of the slash character. For example,

```
*@cs.wisc.edu/bird.cs.wisc.edu
```

refers to any user that comes from `cs.wisc.edu`, where the command is originating from the machine `bird.cs.wisc.edu`. Another example,

```
zmiller@cs.wisc.edu/*.cs.wisc.edu
```

refers to commands coming from any machine within the `cs.wisc.edu` domain, and issued by `zmiller`. A third example,

```
*@cs.wisc.edu/*
```

refers to commands coming from any user within the cs.wisc.edu domain where the command is issued from any machine.

This flexible set of configuration macros could used to define conflicting authorization. Therefore, the following protocol defines the precedence of the configuration macros.

1. DENY_* macros take precedence over ALLOW_* macros where there is a conflict. This implies that if a specific user is both denied and granted authorization, the conflict is resolved by denying access.
2. If macros are omitted, the default behavior is to grant authorization for every user.

Example of Authorization Security Configuration

An example of the configuration variables for the user-side authorization is derived from the necessary access levels as described in Section 3.7.3.

```
ALLOW_READ              = *@cs.wisc.edu/*
ALLOW_WRITE             = *@cs.wisc.edu/*.cs.wisc.edu
ALLOW_ADMINISTRATOR     = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_NEGOTIATOR        = condor@cs.wisc.edu/$(NEGOTIATOR_HOST)
ALLOW_CONFIG            = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_DAEMON            = condor@cs.wisc.edu/*.cs.wisc.edu
```

This example configuration authorizes any user in the cs.wisc.edu domain to carry out a request that requires the READ access level from any machine. Any user in the cs.wisc.edu domain may carry out a request that requires the WRITE access level from any machine in the cs.wisc.edu domain. Only the user called condor-admin may carry out a request that requires the ADMINISTRATOR access level from any machine in the cs.wisc.edu domain. Only the negotiator daemon, running as condor on the machine defined by the NEGOTIATOR_HOST macro is authorized with the NEGOTIATOR access level. The administrator, logged into any machine within the cs.wisc.edu domain is authorized at the CONFIG access level. And, the last line of the example presumes that there is a user called condor, and that the daemons have all been started up as this user. It authorizes only programs (which will be the daemons) running as condor to carry out requests that require the DAEMON access level, where the commands originate from any machine in the cs.wisc.edu domain.

In the local configuration file for each host, the host's owner should be authorized as the owner of the machine. An example of the entry in the local configuration file:

```
ALLOW_OWNER             = username@cs.wisc.edu/hostname.cs.wisc.edu
```

In this example the owner has a login of username, and the machine's name is represented by hostname.

Setting Up IP/Host-Based Security in Condor

This section describes the mechanisms for setting up Condor's host-based security. This is now an outdated form of implementing security at the level of machine access. It remains available and documented for purposes of backward compatibility.

The host-based security allows control over what machines can join a Condor pool, what machines can find out information about your pool, and what machines within your pool can perform administrative commands. By default, Condor is configured to allow anyone to view or join your pool. You probably want to change that.

This section discusses how the host-based security works inside Condor. It lists the different levels of access and what parts of Condor use which levels. There is a description of how to configure your pool to grant (or deny) certain levels of access to various machines. Configuration examples and the settings of configuration variables using the *condor_config_val* command complete this section.

Inside the Condor daemons or tools that use DaemonCore (see section 3.8 for details), most things are accomplished by sending commands to another Condor daemon. These commands are formed from an integer to specify which command, followed by any optional information that the protocol requires at that point (such as a ClassAd, capability string, etc). When the daemons start up, they register which commands they are willing to accept, what to do with arriving commands, and the access level required for that command. When a command arrives, Condor identifies the access level required, and checks the IP address of the sender to be sure it passes the various allow/deny settings in the configuration file for the given access level. If permission is granted, the command continues. If not, the command is aborted.

As expected, settings for the access levels in the global configuration file affect all the machines in the pool. Settings in a local configuration file only affect the specific machine. The settings for a given machine determine what other hosts can send commands to that machine. So, if machine foo is to be given administrator access on machine bar, place foo in bar's configuration file access list (not the other way around).

The following are the various access levels that commands within Condor can be registered with:

READ Machines with `READ` access can read information from Condor. For example, they can view the status of the pool, see the job queue(s) or view user permissions. `READ` access does not allow a machine to change anything, and it does not allow job submission. A machine listed with `READ` permission cannot join a Condor pool; the machine can only view information about the pool.

WRITE Machines with `WRITE` access can write information to Condor. Most notably, a machine can join a pool by sending ClassAd updates to the central manager. The machine can talk to the other machines in a pool in order to submit or run jobs. In addition, any machine with `WRITE` access can request the *condor_startd* daemon to perform a periodic checkpoint on a currently executing job. After a periodic checkpoint, the job will continue to execute, and the machine will still be claimed by whatever *condor_schedd* daemon had claimed it. This allows users on the machines where they submitted their jobs to use the *condor_checkpoint* command

to get their jobs to periodically checkpoint, even if the users do not have an account on the machine where the jobs execute.

IMPORTANT: For a machine to join a Condor pool, the machine must have both `WRITE` permission **AND** `READ` permission. `WRITE` permission is not enough.

`ADMINISTRATOR` Machines with `ADMINISTRATOR` access have additional Condor administrator rights to the pool. This includes the ability to change user priorities (with the command `userprio -set`), and the ability to turn Condor on and off (with the command `off <machine>`). Typically, very few machines are in this list, perhaps only the workstations where the Condor administrators or system administrators work, or perhaps only the pool's central manager.

IMPORTANT: This access is given to a machine, and it applies to an entire pool. So, `ADMINISTRATOR` access for a given machine provides **ANY USER** on that machine `ADMINISTRATOR` rights (including users who can run Condor jobs on that machine). Therefore, grant `ADMINISTRATOR` access carefully.

`OWNER` This level of access is required for commands that the owner of a machine (any local user) should be able to use, in addition to the Condor administrators. For example, the *condor_vacate* command causes the *condor_startd* daemon to vacate any running Condor job. It requires `OWNER` permission, so that any user logged into a local machine can issue a *condor_vacate* command.

`NEGOTIATOR` This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run.

`CONFIG` This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, machines with this level of access are able to change any configuration parameters, except those specified in the `condor_config.root` configuration file. Therefore, granting this level of host-wide access requires extreme caution. By default, `CONFIG` access is denied for all hosts.

Starting with version 6.3.2, Condor provides a mechanism for more fine-grained control over the configuration settings that can be modified remotely with *condor_config_val*. Please see section **??** below on Host Security for *condor_config_val*.

Host-based security access permissions are specified in configuration files.

`ADMINISTRATOR` and `NEGOTIATOR` access default to the central manager machine. `OWNER` access defaults to the local machine, as well as any machines given with `ADMINISTRATOR` access. `CONFIG` access is not granted to any machine as its default. These defaults work well, and should not be changed without a compelling reason. If machines other than the default are to have to have `OWNER` access, they probably should also have `ADMINISTRATOR` access. By granting machines `ADMINISTRATOR` access, they will automatically have `OWNER` access, given how `OWNER` access is set within the configuration.

The default access configuration is

```
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
HOSTALLOW_READ = *
HOSTALLOW_WRITE = *
HOSTALLOW_NEGOTIATOR = $(NEGOTIATOR_HOST)
HOSTALLOW_NEGOTIATOR_SCHEDD = $(NEGOTIATOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
HOSTALLOW_WRITE_COLLECTOR = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_WRITE_STARTD    = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_READ_COLLECTOR  = $(HOSTALLOW_READ), $(FLOCK_FROM)
HOSTALLOW_READ_STARTD     = $(HOSTALLOW_READ), $(FLOCK_FROM)
```

For each access level, an ALLOW or a DENY may be added.

- If you have an ALLOW, it means "only allow these machines". No ALLOW means allow anyone.
- If you have a DENY, it means "deny these machines". No DENY means to deny nobody.
- If you have both an ALLOW and a DENY, it means allow the machines listed in ALLOW except for the machines listed in DENY.
- Exclusively for the `CONFIG` access, no ALLOW means allow no one. Note that this is different than the other ALLOW configurations. It is different to enable more stringent security where older configurations are used, since older configuration files would not have a `CONFIG` configuration entry.

Multiple machine entries in the configuration files may be separated by either a space or a comma. The machines may be listed by

- Individual host names - for example: condor.cs.wisc.edu
- Individual IP address - for example: 128.105.67.29
- IP subnets (use a trailing "*") - for example: 144.105.*, 128.105.67.*
- Host names with a wildcard "*" character (only one "*" is allowed per name) - for example: *.cs.wisc.edu, sol*.cs.wisc.edu

To resolve an entry that falls into both allow and deny: individual machines have a higher order of precedence than wildcard entries, and host names with a wildcard have a higher order of precedence than IP subnets. Otherwise, DENY has a higher order of precedence than ALLOW. (this is how most people would intuitively expect it to work).

In addition, the above access levels may be specified on a per-daemon basis, instead of machine-wide for all daemons. Do this with the subsystem string (described in section 3.3.1 on Subsystem

Names), which is one of: STARTD, SCHEDD, MASTER, NEGOTIATOR, or COLLECTOR. For example, to grant different read access for the *condor_schedd*:

```
HOSTALLOW_READ_SCHEDD = <list of machines>
```

The following is a list of registered commands that daemons will accept. The list is ordered by daemon. For each daemon, the commands are grouped by the access level required for a daemon to accept the command from a given machine.

ALL DAEMONS:

WRITE The command sent as a result of *condor_reconfig* to reconfigure a daemon.

ADMINISTRATOR The command sent as a result of `reconfig -full` to perform a full reconfiguration on a daemon.

STARTD:

WRITE All commands that relate to a *condor_schedd* daemon claiming a machine, starting jobs there, or stopping those jobs.

The command that *condor_checkpoint* sends to periodically checkpoint all running jobs.

READ The command that *condor_preen* sends to request the current state of the *condor_startd* daemon.

OWNER The command that *condor_vacate* sends to cause any running jobs to stop running.

NEGOTIATOR The command that the *condor_negotiator* daemon sends to match a machine's *condor_startd* daemon with a given *condor_schedd* daemon.

NEGOTIATOR:

WRITE The command that initiates a new negotiation cycle. It is sent by the *condor_schedd* when new jobs are submitted or a *condor_reschedule* command is issued.

READ The command that can retrieve the current state of user priorities in the pool (sent by the *condor_userprio* command).

ADMINISTRATOR The command that can set the current values of user priorities (sent as a result of the `userprio -set` command).

COLLECTOR:

WRITE All commands that update the *condor_collector* daemon with new ClassAds.

READ All commands that query the *condor_collector* daemon for ClassAds.

SCHEDD:

NEGOTIATOR The command that the *condor_negotiator* sends to begin negotiating with this *condor_schedd* to match its jobs with available *condor_startds*.

WRITE The command which *condor_reschedule* sends to the *condor_schedd* to get it to update the *condor_collector* with a current ClassAd and begin a negotiation cycle.

The commands that a *condor_startd* sends to the *condor_schedd* when it must vacate its jobs and release the *condor_schedd's* claim.

The commands which write information into the job queue (such as *condor_submit* and *condor_hold*). Note that for most commands which attempt to write to the job queue, Condor will perform an additional user-level authentication step. This additional user-level authentication prevents, for example, an ordinary user from removing a different user's jobs.

READ The command from any tool to view the status of the job queue.

MASTER: All commands are registered with `ADMINISTRATOR` access:

restart : Master restarts itself (and all its children)

off : Master shuts down all its children

off -master : Master shuts down all its children and exits

on : Master spawns all the daemons it is configured to spawn

This section provides examples of configuration settings. Notice that `ADMINISTRATOR` access is only granted through a HOSTALLOW setting to explicitly grant access to a small number of machines. We recommend this.

- Let any machine join your pool. Only the central manager has administrative access (this is the default that ships with Condor)

```
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA to join or view the pool. The central manager is the only machine with `ADMINISTRATOR` access.

```
HOSTALLOW_READ = *.ncsa.uiuc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA and the U of I Math department join the pool, EXCEPT do **not** allow lab machines to do so. Also, do not allow the 177.55 subnet (perhaps this is the dial-in subnet). Allow anyone to view pool statistics. The machine named bigcheese administers the pool (not the central manager).

```
HOSTALLOW_WRITE = *.ncsa.uiuc.edu, *.math.uiuc.edu
HOSTDENY_WRITE = lab-*.edu, *.lab.uiuc.edu, 177.55.*
HOSTALLOW_ADMINISTRATOR = bigcheese.ncsa.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA and UW-Madison's CS department to view the pool. Only NCSA machines and the machine raven.cs.wisc.edu can join the pool. (Note: the machine raven has the read access it needs through the wildcard setting in `HOSTALLOW_READ`). This example also shows how to use "\" to continue a long list of machines onto multiple lines, making it more readable (this works for all configuration file entries, not just host access entries)

```
HOSTALLOW_READ = *.ncsa.uiuc.edu, *.cs.wisc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu, raven.cs.wisc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                          biggercheese.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Allow anyone except the military to view the status of the pool, but only let machines at NCSA view the job queues. Only NCSA machines can join the pool. The central manager, bigcheese, and biggercheese can perform most administrative functions. However, only biggercheese can update user priorities.

```
HOSTDENY_READ = *.mil
HOSTALLOW_READ_SCHEDD = *.ncsa.uiuc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                          biggercheese.uiuc.edu
HOSTALLOW_ADMINISTRATOR_NEGOTIATOR = biggercheese.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

A new security feature introduced in Condor version 6.3.2 enables more fine-grained control over the configuration settings that can be modified remotely with the *condor_config_val* command. The manual page for *condor_config_val* on page 331 details how to use *condor_config_val* to modify configuration settings remotely. Since certain configuration attributes can have a large impact on the functioning of the Condor system and the security of the machines in a Condor pool, it is important to restrict the ability to change attributes remotely.

For each security access level described, the Condor administrator can define which configuration settings a host at that access level is allowed to change. Optionally, the administrator can define separate lists of settable attributes for each Condor daemon, or the administrator can define one list that is used by all daemons.

For each command that requests a change in configuration setting, Condor searches all the different possible security access levels to see which, if any, the request satisfies. (Some hosts can qualify for multiple access levels. For example, any host with ADMINISTRATOR permission probably has WRITE permission also). Within the qualified access level, Condor searches for the list of attributes that may be modified. If the request is covered by the list, the request will be granted. If not covered, the request will be refused.

The default configuration shipped with Condor is exceedingly restrictive. Condor users or administrators cannot set configuration values from remote hosts with *condor_config_val*. Enabling this feature requires a change to the settings in the configuration file. Use this security feature carefully. Grant access only for attributes which you need to be able to modify in this manner, and grant access only at the most restrictive security level possible.

The most secure use of this feature allows Condor users to set attributes in the configuration file which are not used by Condor directly. These are custom attributes published by various Condor daemons with the SUBSYS_EXPRS setting described in section 3.3.4 on page 93. It is secure to grant access only to modify attributes that are used by Condor to publish information. Granting access to modify settings used to control the behavior of Condor is not secure. The goal is to ensure no one can use the power to change configuration attributes to compromise the security of your Condor pool.

The control lists are defined by configuration settings that contain SETTABLE_ATTRS in their name. The name of the control lists have the following form:

```
SUBSYS_SETTABLE_ATTRS_PERMISSION-LEVEL
```

The two parts of this name that can vary are PERMISSION-LEVEL and the SUBSYS. The PERMISSON-LEVEL can be any of the security access levels described earlier in this section. Examples include WRITE, OWNER, and CONFIG.

The SUBSYS is an optional portion of the name. It can be used to define separate rules for which configuration attributes can be set for each kind of Condor daemon (for example, STARTD, SCHEDD, MASTER). There are many configuration settings that can be defined differently for each daemon that use this SUBSYS naming convention. See section 3.3.1 on page 84 for a list. If there is no daemon-specific value for a given daemon, Condor will look for SETTABLE_ATTRS_PERMISSION-LEVEL.

Each control list is defined by a comma-separated list of attribute names which should be allowed to be modified. The lists can contain wildcards characters ('*').

Some examples of valid definitions of control lists with explanations:

- SETTABLE_ATTRS_CONFIG = *

  Grant unlimited access to modify configuration attributes to any request that came from a machine in the CONFIG access level. This was the default behavior before Condor version 6.3.2.

- `SETTABLE_ATTRS_ADMINISTRATOR = *_DEBUG, MAX_*_LOG`

  Grant access to change any configuration setting that ended with "_DEBUG" (for example, `STARTD_DEBUG`) and any attribute that matched "MAX_*_LOG" (for example, `MAX_SCHEDD_LOG`) to any host with `ADMINISTRATOR` access.

- `STARTD_SETTABLE_ATTRS_OWNER = HasDataSet`

  Allows any request to modify the `HasDataSet` attribute that came from a host with `OWNER` access. By default, `OWNER` covers any request originating from the local host, plus any machines listed in the `ADMINISTRATOR` level. Therefore, any Condor job would qualify for OWNER access to the machine where it is running. So, this setting would allow any process running on a given host, including a Condor job, to modify the `HasDataSet` variable for that host. `HasDataSet` is not used by Condor, it is an invented attribute included in the `STARTD_EXPRS` setting in order for this example to make sense.

3.8 DaemonCore

This section is a brief description of *DaemonCore*. DaemonCore is a library that is shared among most of the Condor daemons which provides common functionality. Currently, the following daemons use DaemonCore:

- *condor_master*
- *condor_startd*
- *condor_schedd*
- *condor_collector*
- *condor_negotiator*
- *condor_kbdd*

Most of DaemonCore's details are not interesting for administrators. However, DaemonCore does provide a uniform interface for the daemons to various Unix signals, and provides a common set of command-line options that can be used to start up each daemon.

3.8.1 DaemonCore and Unix signals

One of the most visible features DaemonCore provides for administrators is that all daemons which use it behave the same way on certain Unix signals. The signals and the behavior DaemonCore provides are listed below:

SIGHUP Causes the daemon to reconfigure itself.

SIGTERM Causes the daemon to gracefully shutdown.

SIGQUIT Causes the daemon to quickly shutdown.

Exactly what "gracefully" and "quickly" means varies from daemon to daemon. For daemons with little or no state (the kbdd, collector and negotiator) there's no difference and both signals result in the daemon shutting itself down basically right away. For the master, graceful shutdown just means it asks all of its children to perform their own graceful shutdown methods, while fast shutdown means it asks its children to perform their own fast shutdown methods. In both cases, the master only exits once all its children have exited. In the startd, if the machine is not claimed and running a job, both result in an immediate exit. However, if the startd is running a job, graceful shutdown results in that job being checkpointed, while fast shutdown does not. In the schedd, if there are no jobs currently running (i.e. no *condor_shadow* processes), both signals result in an immediate exit. With jobs running, however, graceful shutdown means that the schedd asks each shadow to gracefully vacate whatever job it is serving, while fast shutdown results in a hard kill of every shadow with no chance of checkpointing.

For all daemons, "reconfigure" just means that the daemon re-reads its config file(s) and any settings that have changed take effect. For example, changing the level of debugging output, the value of timers that determine how often daemons perform certain actions, the paths to the binaries you want the *condor_master* to spawn, etc. See section 3.3 on page 82, "Configuring Condor" for full details on what settings are in the config files and what they do.

3.8.2 DaemonCore and Command-line Arguments

The other visible feature that DaemonCore provides to administrators is a common set of command-line arguments that all daemons understand. The arguments and what they do are described below:

-b Causes the daemon to start up in the background. When a DaemonCore process starts up with this option, disassociates itself from the terminal and forks itself so that it runs in the background. This is the default behavior for Condor daemons, and what you get if you specify no options at all.

-f Causes the daemon to start up in the foreground. Instead of forking, the daemon just runs in the foreground.

NOTE: when the *condor_master* starts up daemons, it does so with the -f option since it has already forked a process for the new daemon. That is why you will see -f in the argument list of all Condor daemons that the master spawns.

-c filename Causes the daemon to use the specified filename (you must use a full path) as its global config file. This overrides the `CONDOR_CONFIG` environment variable, and the regular locations that Condor checks for its config file: the condor user's home directory and `/etc/condor/condor_config`.

-p port Causes the daemon to bind to the specified port for its *command socket*. The master uses this option to make sure the *condor_collector* and *condor_negotiator* start up on the well-known ports that the rest of Condor depends on them using.

-t Causes the daemon to print out its error message to `stderr` instead of its specified log file. This option forces the -f option described above.

-v Causes the daemon to print out version information and exit.

-l directory Overrides the value of `LOG` as specified in your config files. Primarily, this option would be used with the *condor_kbdd* when it needs to run as the individual user logged into the machine, instead of running as root. Regular users would not normally have permission to write files into Condor's log directory. Using this option, they can override the value of `LOG` and have the *condor_kbdd* write its log file into a directory that the user has permission to write to.

-a string Whatever string you specify is automatically appended (with a ".") to the filename of the log for this daemon, as specified in your config file.

-pidfile filename Causes the daemon to write out its PID, or process id number, to the specified file. This file can be used to help shutdown the daemon without searching through the output of the "ps" command.

Since daemons run with their current working directory set to the value of `LOG` , if you don't specify a full path (with a "/" to begin), the file will be left in the log directory. If you leave your pidfile in your log directory, you will want to add whatever filename you use to the `VALID_LOG_FILES` parameter, described in section 3.3.14 on page 110, so that *condor_preen* does not remove it.

-k filename Causes the daemon to read out a pid from the specified filename and send a SIGTERM to that process. The daemon that you start up with "-k" will wait until the daemon it is trying to kill has exited.

-r minutes Causes the daemon to set a timer, upon expiration of which, sends itself a SIGTERM for graceful shutdown.

3.9 Managing your Condor Pool

There are a number of administrative tools Condor provides to help you manage your pool. The following sections describe various tasks you might wish to perform on your pool and explains how to most efficiently do them.

All of the commands described in this section must be run from a machine listed in the `HOST_ALLOW_ADMINISTRATOR` setting in your config files, so that the IP/host-based security allows the administrator commands to be serviced. See section 3.7.5 on page 166 for full details about IP/host-based security in Condor.

3.9.1 Shutting Down and Restarting your Condor Pool

There are a couple of situations where you might want to shutdown and restart your entire Condor pool. In particular, when you want to install new binaries, it is generally best to make sure no jobs are running, shutdown Condor, and then install the new daemons.

Shutting Down your Condor Pool

The best way to shutdown your pool is to take advantage of the remote administration capabilities of the *condor_master*. The first step is to save the IP address and port of the *condor_master* daemon on all of your machines to a file, so that even if you shutdown your *condor_collector*, you can still send administrator commands to your different machines. You do this with the following command:

```
% condor_status -master -format "%s\n" MasterI-
pAddr > addresses
```

The first step to shutting down your pool is to shutdown any currently running jobs and give them a chance to checkpoint. Depending on the size of your pool, your network infrastructure, and the image-size of the standard jobs running in your pool, you may want to make this a slow process, only vacating one host at a time. You can either shutdown hosts that have jobs submitted (in which case all the jobs from that host will try to checkpoint simultaneously), or you can shutdown individual hosts that are running jobs. To shutdown a host, simply send:

```
% condor_off hostname
```

where "hostname" is the name of the host you want to shutdown. This will only work so long as your *condor_collector* is still running. Once you have shutdown Condor on your central manager, you will have to rely on the `addresses` file you just created.

If all the running jobs are checkpointed and stopped, or if you're not worried about the network load put in effect by shutting down everything at once, it is safe to turn off all daemons on all machines in your pool. You can do this with one command, so long as you run it from a blessed administrator machine:

```
% condor_off `cat addresses`
```

where `addresses` is the file where you saved your master addresses. *condor_off* will shutdown all the daemons, but leave the *condor_master* running, so that you can send a *condor_on* in the future.

Once all of the Condor daemons (except the *condor_master*) on each host is turned off, you're done. You are now safe to install new binaries, move your checkpoint server to another host, or any other task that requires the pool to be shutdown to successfully complete.

NOTE: If you are planning to install a new *condor_master* binary, be sure to read the following section for special considerations with this somewhat delicate task.

Installing a New condor_master

If you are going to be installing a new *condor_master* binary, there are a few other steps you should take. If the *condor_master* restarts, it will have a new port it is listening on, so your `addresses` file will be stale information. Moreover, when the master restarts, it doesn't know that you sent it a *condor_off* in its past life, and will just start up all the daemons it's configured to spawn unless you explicitly tell it otherwise.

If you just want your pool to completely restart itself whenever the master notices its new binary, neither of these issues are of any concern and you can skip this (and the next) section. Just be sure installing the new master binary is the last thing you install, and once you put the new binary in place, the pool will restart itself over the next 5 minutes (whenever all the masters notice the new binary, which they each check for once every 5 minutes by default).

However, if you want to have absolute control over when the rest of the daemons restart, you must take a few steps.

1. Put the following setting in your global config file:

   ```
   START_DAEMONS = False
   ```

   This will make sure that when the master restarts itself that it doesn't also start up the rest of its daemons.

2. Install your new *condor_master* binary.

3. Start up Condor on your central manager machine. You will have to do this manually by logging into the machine and sending commands locally. First, send a *condor_restart* to make sure you've got the new master, then send a *condor_on* to start up the other daemons (including, most importantly, the *condor_collector*).

4. Wait 5 minutes, such that all the masters have a chance to notice the new binary, restart themselves, and send an update with their new address. Make sure that:

   ```
   % condor_status -master
   ```

   lists all the machines in your pool.

5. Remove the special setting from your global config file.

6. Recreate your `addresses` file as described above:

   ```
   % condor_status -master -format "%s\n" MasterIpAddr > addresses
   ```

Once the new master is in place, and you're ready to start up your pool again, you can restart your whole pool by simply following the steps in the next section.

Restarting your Condor Pool

Once you are done performing whatever tasks you need to perform and you're ready to restart your pool, you simply have to send a *condor_on* to all the *condor_master* daemons on each host. You can do this with one command, so long as you run it from a blessed administrator machine:

```
% condor_on `cat addresses`
```

That's it. All your daemons should now be restarted, and your pool will be back on its way.

3.9.2 Reconfiguring Your Condor Pool

If you change a global config file setting and want to have all your machines start to use the new setting, you must send a *condor_reconfig* command to each host. You can do this with one command, so long as you run it from a blessed administrator machine:

```
% condor_reconfig `condor_status -master`
```

NOTE: If your global config file is not shared among all your machines (using a shared filesystem), you will need to make the change to each copy of your global config file before sending the *condor_reconfig*.

3.10 Setting up Condor for Special Environments

The following sections describe how to set up Condor for use in special environments or configurations. See section 3.4 on page 113 for installation instructions on the various Contrib modules that can be optionally downloaded and installed.

3.10.1 Using Condor with AFS

If you are using AFS at your site, be sure to read section 3.3.5 on "Shared Filesystem Config Files Entries" for details on configuring your machines to interact with and use shared filesystems, AFS in particular.

Condor does not currently have a way to authenticate itself to AFS. This is true of the Condor daemons that would like to authenticate as AFS user Condor, and the *condor_shadow*, which would like to authenticate as the user who submitted the job it is serving. Since neither of these things can happen yet, there are a number of special things people who use AFS with Condor must do. Some of this must be done by the administrator(s) installing Condor. Some of this must be done by Condor users who submit jobs.

AFS and Condor for Administrators

The most important thing is that since the Condor daemons can't authenticate to AFS, the LOCAL_DIR (and it's subdirectories like "log" and "spool") for each machine must be either writable to unauthenticated users, or must not be on AFS. The first option is a **VERY** bad security hole so you should **NOT** have your local directory on AFS. If you've got NFS installed as well and want to have your LOCAL_DIR for each machine on a shared file system, use NFS. Otherwise, you should put the LOCAL_DIR on a local partition on each machine in your pool. This means that you should run *condor_install* to install your release directory and configure your pool, setting the LOCAL_DIR parameter to some local partition. When that's complete, log into each machine in your pool and run *condor_init* to set up the local Condor directory.

The RELEASE_DIR, which holds all the Condor binaries, libraries and scripts can and probably should be on AFS. None of the Condor daemons need to write to these files, they just need to read them. So, you just have to make your RELEASE_DIR world readable and Condor will work just fine. This makes it easier to upgrade your binaries at a later date, means that your users can find the Condor tools in a consistent location on all the machines in your pool, and that you can have the Condor config files in a centralized location. This is what we do at UW-Madison's CS department Condor pool and it works quite well.

Finally, you might want to setup some special AFS groups to help your users deal with Condor and AFS better (you'll want to read the section below anyway, since you're probably going to have to explain this stuff to your users). Basically, if you can, create an AFS group that contains all unauthenticated users but that is restricted to a given host or subnet. You're supposed to be able to make these host-based ACLs with AFS, but we've had some trouble getting that working here at UW-Madison. What we have instead is a special group for all machines in our department. So, the users here just have to make their output directories on AFS writable to any process running on any of our machines, instead of any process on any machine with AFS on the Internet.

AFS and Condor for Users

The *condor_shadow* process runs on the machine where you submitted your Condor jobs and performs all file system access for your jobs. Because this process isn't authenticated to AFS as the user who submitted the job, it will not normally be able to write any output. So, when you submit jobs, any directories where your job will be creating output files will need to be world writable (to non-authenticated AFS users). In addition, if your program writes to stdout or stderr, or you're using a user log for your jobs, those files will need to be in a directory that's world-writable.

Any input for your job, either the file you specify as input in your submit file, or any files your program opens explicitly, needs to be world-readable.

Some sites may have special AFS groups set up that can make this unauthenticated access to your files less scary. For example, there's supposed to be a way with AFS to grant access to any unauthenticated process on a given host. That way, you only have to grant write access to unauthenticated processes on your submit machine, instead of any unauthenticated process on the Internet. Similarly, unauthenticated read access could be granted only to processes running your submit ma-

chine. Ask your AFS administrators about the existence of such AFS groups and details of how to use them.

The other solution to this problem is to just not use AFS at all. If you have disk space on your submit machine in a partition that is not on AFS, you can submit your jobs from there. While the *condor_shadow* is not authenticated to AFS, it does run with the effective UID of the user who submitted the jobs. So, on a local (or NFS) file system, the *condor_shadow* will be able to access your files normally, and you won't have to grant any special permissions to anyone other than yourself. If the Condor daemons are not started as root however, the shadow will not be able to run with your effective UID, and you'll have a similar problem as you would with files on AFS. See the section on "Running Condor as Non-Root" for details.

3.10.2 Configuring Condor for Multiple Platforms

Beginning with Condor version 6.0.1, you can use a single, global config file for all platforms in your Condor pool, with only platform-specific settings placed in separate files. This greatly simplifies administration of a heterogeneous pool by allowing you to change platform-independent, global settings in one place, instead of separately for each platform. This is made possible by the `LOCAL_CONFIG_FILE` parameter being treated by Condor as a list of files, instead of a single file. Of course, this will only help you if you are using a shared filesystem for the machines in your pool, so that multiple machines can actually share a single set of configuration files.

If you have multiple platforms, you should put all platform-independent settings (the vast majority) into your regular condor_config file, which would be shared by all platforms. This global file would be the one that is found with the `CONDOR_CONFIG` environment variable, user condor's home directory, or `/etc/condor/condor_config`.

You would then set the `LOCAL_CONFIG_FILE` parameter from that global config file to specify both a platform-specific config file and optionally, a local, machine-specific config file (this parameter is described in section 3.3.2 on "Condor-wide Config File Entries").

The order in which you specify files in the `LOCAL_CONFIG_FILE` parameter is important, because settings in files at the beginning of the list are overridden if the same settings occur in files later in the list. So, if you specify the platform-specific file and then the machine-specific file, settings in the machine-specific file would override those in the platform-specific file (which is probably what you want).

Specifying a Platform-Specific Config File

To specify the platform-specific file, you could simply use the `ARCH` and `OPSYS` parameters which are defined automatically by Condor. For example, if you had Intel Linux machines, Sparc Solaris 2.6 machines, and SGIs running IRIX 6.x, you might have files named:

```
condor_config.INTEL.LINUX
condor_config.SUN4x.SOLARIS26
```

```
condor_config.SGI.IRIX6
```

Then, assuming these three files were in the directory held in the ETC macro, and you were using machine-specific config files in the same directory, named by each machine's hostname, your LOCAL_CONFIG_FILE parameter would be set to:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.$(ARCH).$(OPSYS), \
                    $(ETC)/$(HOSTNAME).local
```

Alternatively, if you are using AFS, you can use an "@sys link" to specify the platform-specific config file and let AFS resolve this link differently on different systems. For example, perhaps you have a soft linked named "condor_config.platform" that points to "condor_config.@sys". In this case, your files might be named:

```
condor_config.i386_linux2
condor_config.sun4x_56
condor_config.sgi_64
condor_config.platform -> condor_config.@sys
```

and your LOCAL_CONFIG_FILE parameter would be set to:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.platform, \
                    $(ETC)/$(HOSTNAME).local
```

Platform-Specific Config File Settings

The only settings that are truly platform-specific are:

RELEASE_DIR Full path to where you have installed your Condor binaries. While the config files may be shared among different platforms, the binaries certainly cannot. Therefore, you must still maintain separate release directories for each platform in your pool. See section 3.3.2 on "Condor-wide Config File Entries" for details.

MAIL The full path to your mail program. See section 3.3.2 on "Condor-wide Config File Entries" for details.

CONSOLE_DEVICES Which devices in /dev should be treated as "console devices". See section 3.3.8 on "condor_startd Config File Entries" for details.

DAEMON_LIST Which daemons the *condor_master* should start up. The only reason this setting is platform-specific is because on Alphas running Digital Unix and SGIs running IRIX, you must use the *condor_kbdd*, which is not needed on other platforms. See section 3.3.7 on "condor_master Config File Entries" for details.

Reasonable defaults for all of these settings will be found in the default config files inside a given platform's binary distribution (except the RELEASE_DIR , since it is up to you where you want to install your Condor binaries and libraries). If you have multiple platforms, simply take one of the condor_config files you get from either running *condor_install* or from the <release_dir>/etc/examples/condor_config.generic file, take these settings out and save them into a platform-specific file, and install the resulting platform-independent file as your global config file. Then, find the same settings from the config files for any other platforms you are setting up and put them in their own platform specific files. Finally, set your LOCAL_CONFIG_FILE parameter to point to the appropriate platform-specific file, as described above.

Not even all of these settings are necessarily going to be different. For example, if you have installed a mail program that understands the "-s" option in /usr/local/bin/mail on all your platforms, you could just set MAIL to that in your global file and not define it anywhere else. If you've only got Digital Unix and IRIX machines, the DAEMON_LIST will be the same for each, so there's no reason not to put that in the global config file (or, if you have no IRIX or Digital Unix machines, DAEMON_LIST won't have to be platform-specific either).

Other Uses for Platform-Specific Config Files

It is certainly possible that you might want other settings to be platform-specific as well. Perhaps you want a different startd policy for one of your platforms. Maybe different people should get the email about problems with different platforms. There's nothing hard-coded about any of this. What you decide should be shared and what should not is entirely up to you and how you lay out your config files.

Since the LOCAL_CONFIG_FILE parameter can be an arbitrary list of files, you can even break up your global, platform-independent settings into separate files. In fact, your global config file might only contain a definition for LOCAL_CONFIG_FILE , and all other settings would be handled in separate files.

You might want to give different people permission to change different Condor settings. For example, if you wanted some user to be able to change certain settings, but nothing else, you could specify those settings in a file which was early in the LOCAL_CONFIG_FILE list, give that user write permission on that file, then include all the other files after that one. That way, if the user was trying to change settings she/he shouldn't, they would simply be overridden.

As you can see, this mechanism is quite flexible and powerful. If you have very specific configuration needs, they can probably be met by using file permissions, the LOCAL_CONFIG_FILE setting, and your imagination.

3.10.3 Full Installation of condor_compile

In order to take advantage of two major Condor features: checkpointing and remote system calls, users of the Condor system need to relink their binaries. Programs that are not relinked for Condor can run in Condor's "vanilla" universe just fine, however, they cannot checkpoint and migrate, or

run on machines without a shared filesystem.

To relink your programs with Condor, we provide a special tool, *condor_compile*. As installed by default, *condor_compile* works with the following commands: *gcc*, *g++*, *g77*, *cc*, *acc*, *c89*, *CC*, *f77*, *fort77*, *ld*. On Solaris and Digital Unix, *f90* is also supported. See the *condor_compile*(1) man page for details on using *condor_compile*.

However, you can make *condor_compile* work transparently with all commands on your system whatsoever, including *make*.

The basic idea here is to replace the system linker (*ld*) with the Condor linker. Then, when a program is to be linked, the condor linker figures out whether this binary will be for Condor, or for a normal binary. If it is to be a normal compile, the old *ld* is called. If this binary is to be linked for condor, the script performs the necessary operations in order to prepare a binary that can be used with condor. In order to differentiate between normal builds and condor builds, the user simply places *condor_compile* before their build command, which sets the appropriate environment variable that lets the condor linker script know it needs to do its magic.

In order to perform this full installation of *condor_compile*, the following steps need to be taken:

1. Rename the system linker from ld to ld.real.
2. Copy the condor linker to the location of the previous ld.
3. Set the owner of the linker to root.
4. Set the permissions on the new linker to 755.

The actual commands that you must execute depend upon the system that you are on. The location of the system linker (*ld*), is as follows:

Operating System	Location of ld (ld-path)
Linux	/usr/bin
Solaris 2.X	/usr/ccs/bin
OSF/1 (Digital Unix)	/usr/lib/cmplrs/cc

On these platforms, issue the following commands (as root), where *ld-path* is replaced by the path to your system's *ld*.

```
mv /[ld-path]/ld /[ld-path]/ld.real
cp /usr/local/condor/lib/ld /[ld-path]/ld
chown root /[ld-path]/ld
chmod 755 /[ld-path]/ld
```

On IRIX, things are more complicated in that there are multiple *ld* binaries that need to be moved, and symbolic links need to be made in order to convince the linker to work, since it looks at the name of it's own binary in order to figure out what to do.

```
mv /usr/lib/ld /usr/lib/ld.real
mv /usr/lib/uld /usr/lib/uld.real
cp /usr/local/condor/lib/ld /usr/lib/ld
ln /usr/lib/ld /usr/lib/uld
chown root /usr/lib/ld /usr/lib/uld
chmod 755 /usr/lib/ld /usr/lib/uld
mkdir /usr/lib/condor
chown root /usr/lib/condor
chmod 755 /usr/lib/condor
ln -s /usr/lib/uld.real /usr/lib/condor/uld
ln -s /usr/lib/uld.real /usr/lib/condor/old_ld
```

If you remove Condor from your system latter on, linking will continue to work, since the condor linker will always default to compiling normal binaries and simply call the real ld. In the interest of simplicity, it is recommended that you reverse the above changes by moving your ld.real linker back to it's former position as ld, overwriting the condor linker. On IRIX, you need to do this for both linkers, and you will probably want to remove the symbolic links as well.

NOTE: If you ever upgrade your operating system after performing a full installation of *condor_compile*, you will probably have to re-do all the steps outlined above. Generally speaking, new versions or patches of an operating system might replace the system ld binary, which would undo the full installation of *condor_compile*.

3.10.4 Installing the *condor_kbdd*

The condor keyboard daemon (*condor_kbdd*) monitors X events on machines where the operating system does not provide a way of monitoring the idle time of the keyboard or mouse. In particular, this is necessary on Digital Unix machines and IRIX machines.

NOTE: If you are running on Solaris, Linux, or HP/UX, you do not need to use the keyboard daemon.

Although great measures have been taken to make this daemon as robust as possible, the X window system was not designed to facilitate such a need, and thus is less then optimal on machines where many users log in and out on the console frequently.

In order to work with X authority, the system by which X authorizes processes to connect to X servers, the condor keyboard daemon needs to run with super user privileges. Currently, the daemon assumes that X uses the `HOME` environment variable in order to locate a file named `.Xauthority`, which contains keys necessary to connect to an X server. The keyboard daemon attempts to set this environment variable to various users home directories in order to gain a connection to the X server and monitor events. This may fail to work on your system, if you are using a non-standard approach. If the keyboard daemon is not allowed to attach to the X server, the state of a machine may be incorrectly set to idle when a user is, in fact, using the machine.

In some environments, the keyboard daemon will not be able to connect to the X server because

the user currently logged into the system keeps their authentication token for using the X server in a place that no local user on the current machine can get to. This may be the case if you are running AFS and have the user's X authority file in an AFS home directory. There may also be cases where you cannot run the daemon with super user privileges because of political reasons, but you would still like to be able to monitor X activity. In these cases, you will need to change your XDM configuration in order to start up the keyboard daemon with the permissions of the currently logging in user. Although your situation may differ, if you are running X11R6.3, you will probably want to edit the files in /usr/X11R6/lib/X11/xdm. The Xsession file should have the keyboard daemon startup at the end, and the Xreset file should have the keyboard daemon shutdown. As of patch level 4 of Condor version 6.0, the keyboard daemon has some additional command line options to facilitate this. The -l option can be used to write the daemons log file to a place where the user running the daemon has permission to write a file. We recommend something akin to $HOME/.kbdd.log since this is a place where every user can write and won't get in the way. The -pidfile and -k options allow for easy shutdown of the daemon by storing the process id in a file. You will need to add lines to your XDM config that look something like this:

```
condor_kbdd -l $HOME/.kbdd.log -pidfile $HOME/.kbdd.pid
```

This will start the keyboard daemon as the user who is currently logging in and write the log to a file in the directory $HOME/.kbdd.log/. Also, this will save the process id of the daemon to /.kbdd.pid, so that when the user logs out, XDM can simply do a:

```
condor_kbdd -k $HOME/.kbdd.pid
```

This will shutdown the process recorded in /.kbdd.pid and exit.

To see how well the keyboard daemon is working on your system, review the log for the daemon and look for successful connections to the X server. If you see none, you may have a situation where the keyboard daemon is unable to connect to your machines X server. If this happens, please send mail to condor-admin@cs.wisc.edu and let us know about your situation.

3.10.5 Installing a Checkpoint Server

The Checkpoint Server maintains a repository for checkpoint files. Using checkpoint servers reduces the disk requirements of submitting machines in the pool, since the submitting machines no longer need to store checkpoint files locally. Checkpoint server machines should have a large amount of disk space available, and they should have a fast connection to machines in the Condor pool.

If your spool directories are on a network file system, then checkpoint files will make two trips over the network: one between the submitting machine and the execution machine, and a second between the submitting machine and the network file server. If you install a checkpoint server and configure it to use the server's local disk, the checkpoint will travel only once over the network, between the execution machine and the checkpoint server. You may also obtain checkpointing network performance benefits by using multiple checkpoint servers, as discussed below.

NOTE: It is a good idea to pick very stable machines for your checkpoint servers. If individual checkpoint servers crash, the Condor system will continue to operate, although poorly. While the Condor system will recover from a checkpoint server crash as best it can, there are two problems that can (and will) occur:

1. A checkpoint cannot be sent to a checkpoint server that is not functioning. Jobs will keep trying to contact the checkpoint server, backing off exponentially in the time they wait between attempts. Normally, jobs only have a limited time to checkpoint before they are kicked off the machine. So, if the server is down for a long period of time, chances are that a lot of work will be lost by jobs being killed without writing a checkpoint.
2. If a checkpoint is not available from the checkpoint server, a job cannot be retrieved, and it will either have to be restarted from the beginning, or the job will wait for the server to come back online. This behavior is controlled with the `MAX_DISCARDED_RUN_TIME` parameter in the config file (see section 3.3.6 on page 96 for details). This parameter represents the maximum amount of CPU time you are willing to discard by starting a job over from scratch if the checkpoint server is not responding to requests.

Preparing to Install a Checkpoint Server

The location of checkpoints changes upon the installation of a checkpoint server. A configuration change would cause currently queued jobs with checkpoints to not be able to find their checkpoints. This results in the jobs with checkpoints remaining indefinitely queued (never running) due to the lack of finding their checkpoints. It is therefore best to either remove jobs from the queues or let them complete before installing a checkpoint server. It is advisable to shut your pool down before doing any maintenance on your checkpoint server. See section 3.9 on page 175 for details on shutting down your pool.

A graduated installation of the checkpoint server may be accomplished by configuring submit machines as their queues empty.

Installing the Checkpoint Server Module

To install a checkpoint server, download the appropriate binary contrib module for the platform(s) on which your server will run. Uncompress and untar the file to result in a directory that contains a `README`, `ckpt_server.tar`, and so on. The file `ckpt_server.tar` acts much like the `release.tar` file from a main release. This archive contains the files:

```
sbin/condor_ckpt_server
sbin/condor_cleanckpts
etc/examples/condor_config.local.ckpt.server
```

These new files are not found in the main release, so you can safely untar the archive directly into your existing release directory. `condor_ckpt_server` is the checkpoint server binary. `condor_cleanckpts` is a script that can be periodically run to remove stale checkpoint files from

your server. The checkpoint server normally cleans all old files itself. However, in certain error situations, stale files can be left that are no longer needed. You may set up a cron job that calls *condor_cleanckpts* every week or so to automate the cleaning up of any stale files. The example configuration file give with the module is described below.

After unpacking the module, there are three steps to complete. Each is discussed in its own section:

1. Configure the checkpoint server.
2. Start the checkpoint server.
3. Configure your pool to use the checkpoint server.

Configuring a Checkpoint Server

Place settings in the local configuration file of the checkpoint server. The file `etc/examples/condor_config.local.ckpt.server` contains the needed settings. Insert these into the local configuration file of your checkpoint server machine.

The `CKPT_SERVER_DIR` must be customized. The `CKPT_SERVER_DIR` attribute defines where your checkpoint files are to be located. It is better if this is on a very fast local file system (preferably a RAID). The speed of this file system will have a direct impact on the speed at which your checkpoint files can be retrieved from the remote machines.

The other optional settings are:

DAEMON_LIST (Described in section 3.3.7). To have the checkpoint server managed by the *condor_master*, the `DAEMON_LIST` entry must have `MASTER` and `CKPT_SERVER`. Add `STARTD` if you want to allow jobs to run on your checkpoint server. Similarly, add `SCHEDD` if you would like to submit jobs from your checkpoint server.

The rest of these settings are the checkpoint server-specific versions of the Condor logging entries, as described in section 3.3.3 on page 89.

CKPT_SERVER_LOG The `CKPT_SERVER_LOG` is where the checkpoint server log is placed.

MAX_CKPT_SERVER_LOG Sets the maximum size of the checkpoint server log before it is saved and the log file restarted.

CKPT_SERVER_DEBUG Regulates the amount of information printed in the log file. Currently, the only debug level supported is `D_ALWAYS`.

Start the Checkpoint Server

To start the newly configured checkpoint server, restart Condor on that host to enable the *condor_master* to notice the new configuration. Do this by sending a *condor_restart* command from any

machine with administrator access to your pool. See section 3.7.5 on page 166 for full details about IP/host-based security in Condor.

Configuring your Pool to Use the Checkpoint Server

After the checkpoint server is running, you change a few settings in your configuration files to let your pool know about your new server:

USE_CKPT_SERVER This parameter should be set to TRUE (the default).

CKPT_SERVER_HOST This parameter should be set to the full hostname of the machine that is now running your checkpoint server.

It is most convenient to set these parameters in your global configuration file, so they affect all submission machines. However, you may configure each submission machine separately (using local configuration files) if you do not want all of your submission machines to start using the checkpoint server at one time. If USE_CKPT_SERVER is set to FALSE, the submission machine will not use a checkpoint server.

Once these settings are in place, send a *condor_reconfig* to all machines in your pool so the changes take effect. This is described in section 3.9.2 on page 178.

Configuring your Pool to Use Multiple Checkpoint Servers

It is possible to configure a Condor pool to use multiple checkpoint servers. The deployment of checkpoint servers across the network improves checkpointing performance. In this case, Condor machines are configured to checkpoint to the *nearest* checkpoint server. There are two main performance benefits to deploying multiple checkpoint servers:

- Checkpoint-related network traffic is localized by intelligent placement of checkpoint servers.
- Faster checkpointing implies that jobs spend less time checkpointing, more time doing useful work, jobs have a better chance of checkpointing successfully before returning a machine to its owner, and workstation owners see Condor jobs leave their machines quicker.

Once you have multiple checkpoint servers running in your pool, the following configuration changes are required to make them active.

First, USE_CKPT_SERVER should be set to TRUE (the default) on all submitting machines where Condor jobs should use a checkpoint server. Additionally, STARTER_CHOOSES_CKPT_SERVER should be set to TRUE (the default) on these submitting machines. When TRUE, this parameter specifies that the checkpoint server specified by the machine running the job should be used instead of the checkpoint server specified by the submitting machine. See section 3.3.6 on page 96 for more details. This allows the job to use the checkpoint

server closest to the machine on which it is running, instead of the server closest to the submitting machine. For convenience, set these parameters in the global configuration file.

Second, set CKPT_SERVER_HOST on each machine. As described, this is set to the full hostname of the checkpoint server machine. In the case of multiple checkpoint servers, set this in the local configuraton file. It is the hostname of the nearest server to the machine.

Third, send a *condor_reconfig* to all machines in the pool so the changes take effect. This is described in section 3.9.2 on page 178.

After completing these three steps, the jobs in your pool will send checkpoints to the nearest checkpoint server. On restart, a job will remember where its checkpoint was stored and get it from the appropriate server. After a job successfully writes a checkpoint to a new server, it will remove any previous checkpoints left on other servers.

NOTE: If the configured checkpoint server is unavailable, the job will keep trying to contact that server as described above. It will not use alternate checkpoint servers. This may change in future versions of Condor.

Checkpoint Server Domains

The configuration described in the previous section ensures that jobs will always write checkpoints to their nearest checkpoint server. In some circumstances, it is also useful to configure Condor to localize checkpoint read transfers, which occur when the job restarts from its last checkpoint on a new machine. To localize these transfers, we want to schedule the job on a machine which is near the checkpoint server on which the job's checkpoint is stored.

We can say that all of the machines configured to use checkpoint server "A" are in "checkpoint server domain A." To localize checkpoint transfers, we want jobs which run on machines in a given checkpoint server domain to continue running on machines in that domain, transferring checkpoint files in a single local area of the network. There are two possible configurations which specify what a job should do when there are no available machines in its checkpoint server domain:

- The job can remain idle until a workstation in its checkpoint server domain becomes available.
- The job can try to immediately begin executing on a machine in another checkpoint server domain. In this case, the job transfers to a new checkpoint server domain.

These two configurations are described below.

The first step in implementing checkpoint server domains is to include the name of the nearest checkpoint server in the machine ClassAd, so this information can be used in job scheduling decisions. To do this, add the following configuration to each machine:

```
CkptServer = "$(CKPT_SERVER_HOST)"
STARTD_EXPRS = $(STARTD_EXPRS), CkptServer
```

For convenience, we suggest that you set these parameters in the global config file. Note that this example assumes that STARTD_EXPRS is defined previously in your configuration. If not, then you should use the following configuration instead:

```
CkptServer = "$(CKPT_SERVER_HOST)"
STARTD_EXPRS = CkptServer
```

Now, all machine ClassAds will include a CkptServer attribute, which is the name of the checkpoint server closest to this machine. So, the CkptServer attribute defines the checkpoint server domain of each machine.

To restrict jobs to one checkpoint server domain, we need to modify the jobs' Requirements expression as follows:

```
Requirements = ((LastCkptServer == TARGET.CkptServer) || (LastCkpt-
Server =?= UNDEFINED))
```

This Requirements expression uses the LastCkptServer attribute in the job's ClassAd, which specifies where the job last wrote a checkpoint, and the CkptServer attribute in the machine ClassAd, which specifies the checkpoint server domain. If the job has not written a checkpoint yet, the LastCkptServer attribute will be UNDEFINED, and the job will be able to execute in any checkpoint server domain. However, once the job performs a checkpoint, LastCkptServer will be defined and the job will be restricted to the checkpoint server domain where it started running.

If instead we want to allow jobs to transfer to other checkpoint server domains when there are no available machines in the current checkpoint server domain, we need to modify the jobs' Rank expression as follows:

```
Rank = ((LastCkptServer == TARGET.CkptServer) || (LastCkpt-
Server =?= UNDEFINED))
```

This Rank expression will evaluate to 1 for machines in the job's checkpoint server domain and 0 for other machines. So, the job will prefer to run on machines in its checkpoint server domain, but if no such machines are available, the job will run in a new checkpoint server domain.

You can automatically append the checkpoint server domain Requirements or Rank expressions to all STANDARD universe jobs submitted in your pool using APPEND_REQ_STANDARD or APPEND_RANK_STANDARD . See section 3.3.13 on page 109 for more details.

3.10.6 Configuring The CondorView Server

The CondorView server is an alternate use of the *condor_collector* that logs information on disk, providing a persistent, historical database of pool state. This includes machine state, as well as the

state of jobs submitted by users. Historical information logging can be turned on or off, so you can install the CondorView collector without using up disk space for historical information if you don't want it.

The CondorView collector is a *condor_collector* that has been specially configured *and running on a different machine from the main condor_collector*. The pool must be configured to send updates to both the normal collector and the CondorView collector. Unfortunately, installing the CondorView collector on a separate host generates more network traffic (from all the duplicate updates that are sent from each machine in your pool to both collectors).

Configuring a Machine to be a CondorView Server

Before you configure the CondorView collector (as described in the following sections), you have to add a few settings to the *local* configuration file of the chosen machine(that is not the main *condor_collector* machine) to enable historical data collection. These settings are described in detail in the Condor Version 6.1 Administrator's Manual, in the section "condor_collector Config File Entries". A short explanation of the entries you must customize is provided below.

POOL_HISTORY_DIR This is the directory where historical data will be stored. This directory must be writable by whatever user the CondorView collector is running as (usually the user condor). There is a configurable limit to the maximum space required for all the files created by the CondorView server called (`POOL_HISTORY_MAX_STORAGE`).

NOTE: This directory should be separate and different from the `spool` or `log` directories already set up for Condor. There are a few problems putting these files into either of those directories.

KEEP_POOL_HISTORY This is a boolean value that determines if the CondorView collector should store the historical information. It is false by default, which is why you must specify it as true in your local configuration file to enable data collection.

Once these settings are in place in the local configuration file for your CondorView server host, you must to create the directory you specified in `POOL_HISTORY_DIR` and make it writable by the user your CondorView collector is running as. This is the same user that owns the `CollectorLog` file in your `log` directory. The user is usually condor.

After you've configured the CondorView attributes, you must configure Condor to automatically start and then begin reporting to the CondorView server. You do this by adding `COLLECTOR` to the `DAEMON_LIST` on this machine and defining what `COLLECTOR` means. For example:

```
COLLECTOR = $(SBIN)/condor_collector
DAEMON_LIST = MASTER, STARTD, SCHEDD, COLLECTOR
```

For this change to take effect, you must re-start the *condor_master* on this host (which you can do with the *condor_restart* command, if you run the command from a machine with administrator

access to your pool. (See section 3.7.5 on page 166 for full details of IP/host-based security in Condor).

As a last step, you tell all the machines in your pool to start sending updates to both collectors. You do this by specifying the following setting in your global configuration file:

```
CONDOR_VIEW_HOST = full.hostname
```

where `full.hostname` is the full hostname of the machine where you are running your CondorView collector.

Once this setting is in place, send a *condor_reconfig* to all machines in your pool so the changes take effect. This is described in section 3.9.2 on page 178.

3.10.7 Flocking: Configuring a Schedd to Submit to Multiple Pools

The *condor_schedd* may be configured to submit jobs to more than one pool—this is known as flocking. If Condor pool A can send jobs to Condor pool B, then we say that A flocks to B. Flocking can be one way, such as A flocking to B, or it can be set up in both directions.

To configure flocking, you normally need to set just two configuration variables. Assume you have the situation where pool A is flocking to pool B. In pool A, set `FLOCK_TO` to the host name of the central manager of pool B. You could set a list of host names, if you were flocking to multiple pools. In pool B, set `FLOCK_FROM` to the names of all the hosts from pool A that might flock to pool B. If you don't wish to list all of the hosts, you can use a wildcard to allow multiple hosts. For example, you could say use "*.cs.wisc.edu" to allow all hosts from the cs.wisc.edu domain.

If you wish to also allow flocking from pool B to pool A, you can simply set up flocking in the other direction.

When you flock to another pool, you will not attempt to flock a particular job unless you cannot currently run it in your pool. Jobs that are run in another pool can only be standard universe jobs, and they are run as user "nobody".

`$(HOSTALLOW_NEGOTIATOR_SCHEDD)` (see section 3.3.4) must also be configured to allow negotiators from all of the `$(FLOCK_NEGOTIATOR_HOSTS)` to contact the schedd. Please make sure the `$(NEGOTIATOR_HOST)` is first in the `$(HOSTALLOW_NEGOTIATOR_SCHEDD)` list. This is the default configuration, so it will be correct if you haven't modified it.

3.10.8 Configuring The Startd for SMP Machines

This section describes how to configure the *condor_startd* for SMP (Symmetric Multi-Processor) machines. Beginning with Condor version 6.1, machines with more than one CPU can be configured to run more than one job at a time. As always, owners of the resources have great flexibility in defining the policy under which multiple jobs may run, suspend, vacate, etc.

How Shared Resources are Represented to Condor

The way SMP machines are represented to the Condor system is that the shared resources are broken up into individual *virtual machines* ("VM") that can be matched with and claimed by users. Each virtual machine is represented by an individual "ClassAd" (see the ClassAd reference, section 4.1, for details). In this way, a single SMP machine will appear to the Condor system as a collection of separate virtual machines. So for example, if you had an SMP machine named "vulture.cs.wisc.edu", it would appear to Condor as multiple machines, named "vm1@vulture.cs.wisc.edu", "vm2@vulture.cs.wisc.edu", and so on.

You can configure how you want the *condor_startd* to break up the shared system resources into the different virtual machines. All shared system resources (like RAM, disk space, swap space, etc) can either be divided evenly among all the virtual machines, with each CPU getting its own virtual machine, or you can define your own *virtual machine types*, so that resources can be unevenly partitioned. The following section gives details on how to configure Condor to divide the resources on an SMP machine into seperate virtual machines.

Dividing System Resources in SMP Machines

This section describes the settings that allow you to define your own virtual machine types and to control how many virtual machines of each type are reported to Condor.

There are two main ways to go about dividing an SMP machine:

Define your own virtual machine types. By defining your own types, you can specify what fraction of shared system resources (CPU, RAM, swap space and disk space) go to each virtual machine. Once you define your own types, you can control how many of each type are being reported at any given time.

Evenly divide all resources. If you do not define your own types, the *condor_startd* will automatically partition your machine into virtual machines for you. It will do so by giving each VM a single CPU, and evenly dividing all shared resources among each CPU. With this default partitioning, you only have to specify how many VMs are reported at a time. By default, all VMs are reported to Condor.

Begining with Condor version 6.1.6, the number of each type being reported can be changed at run-time, by issuing a simple reconfig to the *condor_startd* (sending a SIGHUP or using *condor_reconfig*). However, the definitions for the types themselves cannot be changed with a reconfig. If you change any VM type definitions, you must use "condor_restart -startd" for that change to take effect.

Defining Virtual Machine Types

To define your own virtual machine types, you simply add config file parameters that list how much of each system resource you want in the given VM type. You do this with settings of the form VIRTUAL_MACHINE_TYPE_<N> . The <N> is to be replaced with an integer, for example, VIRTUAL_MACHINE_TYPE_1, which specifies the virtual machine type you're defining. You will use this number later to configure how many VMs of this type you want to advertise.

A type describes what share of the total system resources a given virtual machine has available to it.

The type can be defined in a number of ways:

- A simple fraction, such as "1/4"
- A simple percentage, such as "25%"
- A comma-separated list of attributes, and a percentage, fraction, or value for each one.

If you just specify a fraction or percentage, that share of the total system resources, including the number of cpus, will be used for each virtual machine of this type. However, if you specify the comma-seperated list, you can fine-tune the amounts for specific attributes.

Some attributes, such as the number of CPUs and total amount of RAM in the machine, do not change (unless the machine is turned off and more chips are added to it). For these two attributes, you can specify either absolute values, or percentages of the total available amount. For example, in a machine with 128 megs of RAM, you could specify any of the following to get the same effect: "mem=64", "mem=1/2", or "mem=50%". Other resources are dynamic, such as disk space and swap space. For these, you must specify the percentage or fraction of the total value that is alloted to each VM, instead of specifying absolute values. As the total values of these resources change on your machine, each VM will take its fraction of the total and report that as its available amount.

All attribute names are case insensitive when defining VM types. You can use as much or as little of each word as you'd like. The attributes you can tune are:

- cpus
- ram
- disk (must specify with a fraction or percentage)
- swap (must specify with a fraction or percentage)

In addition, the following names are equivalent: "ram" = "memory" and "swap" = "virtualmemory".

Assume the host as 4 CPUs and 256 megs of RAM. Here are some example VM type definitions, all of which are valid. Types 1-3 are all equivalent with each other, as are types 4-6

VIRTUAL_MACHINE_TYPE_1 = cpus=2, ram=128, swap=25%, disk=1/2

VIRTUAL_MACHINE_TYPE_2 = cpus=1/2, memory=128, virt=25%, disk=50%

VIRTUAL_MACHINE_TYPE_3 = c=1/2, m=50%, v=1/4, disk=1/2

VIRTUAL_MACHINE_TYPE_4 = c=25%, m=64, v=1/4, d=25%

VIRTUAL_MACHINE_TYPE_5 = 25%

VIRTUAL_MACHINE_TYPE_6 = 1/4

Configuring the Number of Virtual Machines Reported

If you are not defining your own VM types, all you have to configure is how many of the evenly divided VMs you want reported to Condor. You do this by setting the NUM_VIRTUAL_MACHINES parameter. You just supply the number of machines you want reported. If you do not define this yourself, Condor will advertise all the CPUs in your machines by default.

If you define your own types, things are slightly more complicated. Now, you must specify how many virtual machines of each type should be reported. You do this with settings of the form NUM_VIRTUAL_MACHINES_TYPE_<N> . The <N> is to be replaced with an actual number, for example, NUM_VIRTUAL_MACHINES_TYPE_1.

Configuring Startd Policy for SMP Machines

NOTE: Be sure you have read and understand section 3.6 on "Configuring The Startd Policy" before you proceed with this section.

Each virtual machine from an SMP is treated as an independent machine, with its own view of its machine state. For now, a single set of policy expressions is in place for all virtual machines simultaneously. Eventually, you will be able to explicitly specify separate policies for each one. However, since you do have control over each virtual machine's view of its own state, you can effectively have separate policies for each resource.

For example, you can configure how many of the virtual machines "notice" console or tty activity on the SMP as a whole. Ones that aren't configured to notice any activity will report ConsoleIdle and KeyboardIdle times from when the startd was started, (plus a configurable number of seconds). So, you can setup a 4 CPU machine with all the default startd policy settings and with the keyboard and console "connected" to only one virtual machine. Assuming there isn't too much load average (see section 3.10.8 below on "Load Average for SMP Machines"), only one virtual machine will suspend or vacate its job when the owner starts typing at their machine again. The rest of the virtual machines could be matched with jobs and leave them running, even while the user was interactively using the machine.

Or, if you wish, you can configure all virtual machines to notice all tty and console activity. In this case, if a machine owner came back to her machine, all the currently running jobs would suspend or preempt (depending on your policy expressions), all at the same time.

All of this is controlled with the config file parameters listed below. These settings are fully described in section 3.3.8 on page 100 which lists all the configuration file settings for the *condor_startd*.

- VIRTUAL_MACHINES_CONNECTED_TO_CONSOLE
- VIRTUAL_MACHINES_CONNECTED_TO_KEYBOARD
- DISCONNECTED_KEYBOARD_IDLE_BOOST

Load Average for SMP Machines

Most operating systems define the load average for an SMP machine as the total load on all CPUs. For example, if you have a 4 CPU machine with 3 CPU-bound processes running at the same time, the load would be 3.0 In Condor, we maintain this view of the total load average and publish it in all resource ClassAds as TotalLoadAvg.

However, we also define the "per-CPU" load average for SMP machines. In this way, the model that each node on an SMP is a virtual machine, totally separate from the other nodes, can be maintained. All of the default, single-CPU policy expressions can be used directly on SMP machines, without modification, since the LoadAvg and CondorLoadAvg attributes are the per-virtual machine versions, not the total, SMP-wide versions.

The per-CPU load average on SMP machines is a number we basically invented. There is no system call you can use to ask your operating system for this value. Here's how it works:

We already compute the load average generated by Condor on each virtual machine. We do this by close monitoring of all processes spawned by any of the Condor daemons, even ones that are orphaned and then inherited by init. This *Condor load average* per virtual machine is reported as CondorLoadAvg in all resource ClassAds, and the total Condor load average for the entire machine is reported as TotalCondorLoadAvg. We also have the total, system-wide load average for the entire machine (reported as TotalLoadAvg). Basically, we walk through all the virtual machines and assign out portions of the total load average to each one. First, we assign out the known Condor load average to each node that is generating any. If there's any load average left in the total system load, that's considered *owner load*. Any virtual machines we already think are in the Owner state (like ones that have keyboard activity, etc), are the first to get assigned this owner load. We hand out owner load in increments of at most 1.0, so generally speaking, no virtual machine has a load average above 1.0. If we run out of total load average before we run out of virtual machines, all the remaining machines think they have no load average at all. If, instead, we run out of virtual machines and we still have owner load left, we start assigning that load to Condor nodes, too, creating individual nodes with a load average higher than 1.0.

Debug logging in the SMP Startd

This section describes how the startd handles its debug messages for SMP machines. In general, a given log message will either be something that is machine-wide (like reporting the total system

load average), or it will be specific to a given virtual machine. Any log entrees specific to a virtual machine will have an extra header printed out in the entry: `vm#:`. So, for example, here's the output about system resources that are being gathered (with `D_FULLDEBUG` and `D_LOAD` turned on) on a 2 CPU machine with no Condor activity, and the keyboard connected to both virtual machines:

```
11/25 18:15 Swap space: 131064
11/25 18:15 number of kbytes available for (/home/condor/execute): 1345063
11/25 18:15 Looking up RESERVED_DISK parameter
11/25 18:15 Reserving 5120 kbytes for file system
11/25 18:15 Disk space: 1339943
11/25 18:15 Load avg: 0.340000 0.800000 1.170000
11/25 18:15 Idle Time: user= 0 , console= 4 seconds
11/25 18:15 SystemLoad: 0.340   TotalCondorLoad: 0.000  TotalOwn-
erLoad: 0.340
11/25 18:15 vm1: Idle time: Keyboard: 0        Console: 4
11/25 18:15 vm1: SystemLoad: 0.340  CondorLoad: 0.000  Owner-
Load: 0.340
11/25 18:15 vm2: Idle time: Keyboard: 0        Console: 4
11/25 18:15 vm2: SystemLoad: 0.000  CondorLoad: 0.000  Owner-
Load: 0.000
11/25 18:15 vm1: State: Owner          Activity: Idle
11/25 18:15 vm2: State: Owner          Activity: Idle
```

If, on the other hand, this machine only had one virtual machine connected to the keyboard and console, and the other vm was running a job, it might look something like this:

```
11/25 18:19 Load avg: 1.250000 0.910000 1.090000
11/25 18:19 Idle Time: user= 0 , console= 0 seconds
11/25 18:19 SystemLoad: 1.250   TotalCondorLoad: 0.996  TotalOwn-
erLoad: 0.254
11/25 18:19 vm1: Idle time: Keyboard: 0        Console: 0
11/25 18:19 vm1: SystemLoad: 0.254  CondorLoad: 0.000  Owner-
Load: 0.254
11/25 18:19 vm2: Idle time: Keyboard: 1496     Console: 1496
11/25 18:19 vm2: SystemLoad: 0.996  CondorLoad: 0.996  Owner-
Load: 0.000
11/25 18:19 vm1: State: Owner          Activity: Idle
11/25 18:19 vm2: State: Claimed        Activity: Busy
```

As you can see, shared system resources are printed without the header (like total swap space), which VM-specific messages (like the load average or state of each VM,) get the special header appended.

3.10.9 Configuring Condor for Machines With Multiple Network Interfaces

Beginning with Condor version 6.1.5, Condor can run on machines with multiple network interfaces. Here are some common scenarios that you might encounter and how you go about solving them.

Central Manager with Two or More NICs

Often users of Condor wish to set up "compute farms" where there is one machine with two network interface cards(one for the public internet, and one for the private net). It is convenient to set up the "head" node as a central manager in most cases and so here are the instructions required to do so.

Setting up the central manager on a machine with more than one NIC can be a little confusing because there are a few external variables that could make the process difficult. One of the biggest mistakes in getting this to work is that either one of the separate interfaces is not active, or the host/domain names associated with the interfaces are incorrectly configured.

Given that the interfaces are up and functioning, and they have good host/domain names associated with them here is how to configure Condor:

In this example, **farm-server.farm.org** maps to the private interface.

On your central manager's global(to the cluster) config file:
`CONDOR_HOST` = **farm-server.farm.org**

On your central manager's local configuration file:
`NETWORK_INTERFACE` = ip address of **farm-server.farm.org**
`NEGOTIATOR` = `$(SBIN)`/condor_negotiator
`COLLECTOR` = `$(SBIN)`/condor_collector
`DAEMON_LIST` = `MASTER`, `COLLECTOR`, `NEGOTIATOR`, `SCHEDD`, `STARTD`

If your central manager and farm machines are all NT, then you only have vanilla universe and it will work now. However, if you have this setup for UNIX, then at this point, standard universe jobs should be able to function in the pool, but if you did not configure the `UID_DOMAIN` macro to be homogeneous across the farm machines, the standard universe jobs will run as **nobody** on the farm machines.

In order to get vanilla jobs and file server load balancing for standard universe jobs working(under unix), you need to do some more work both in the cluster you have put together and in Condor to make everything work. First, you need a file server(which could also be the central manager) to serve files to all of the farm machines. This could be NFS or AFS, it does not really matter to Condor. The mount point of the directories you wish your users to use must be the same across all of the farm machines. Now, configure `UID_DOMAIN` and `FILESYSTEM_DOMAIN` to be homogeneous across the farm machines and the central manager. Now, you will have to inform Condor that an NFS or AFS filesystem exists and that is done in this manner. In the global(to the farm) configuration file:

```
# If you have NFS
```

```
USE_NFS = True
# If you have AFS
HAS_AFS = True
USE_AFS = True
# if you want both NFS and AFS, then enable both sets above
```

Now, if you've set up your cluster so that it is possible for a machine name to never have a domain name(for example: you've placed a machine name but no fully qualified domain name in `/etc/hosts`), you must configure `DEFAULT_DOMAIN_NAME` to be the domain that you wish to be added on to the end of your hostname.

A Client Machine with Multiple Interfaces

If you have a client machine with two or more NICs, then there might be a specific network interface with which you desire a client machine to communicate with the rest of the Condor pool. In this case, in the local configuration file for that machine, place:
`NETWORK_INTERFACE` = ip address of interface desired

A Checkpoint Server on a Machine with Multiple NICs

If your Checkpoint Server is on a machine with multiple interfaces, the only way to get things to work is if your different interfaces have different hostnames associated with them, and you set `CKPT_SERVER_HOST` to the hostname that corresponds with the IP address you want to use in the global configuration file for your pool. You will still need to specify `NETWORK_INTERFACE` in the local config file for your Checkpoint Server.

3.10.10 Allocating Bandwidth

An experimental mechanism for allocating bandwidth for checkpointing and remote system calls has been introduced in Condor version 6.3.0. This mechanism enables the *condor_negotiator* to limit job placements and preemptions to within configured bandwidth limits. If a bandwidth limit is reached for a host or network subnet, the *condor_negotiator* won't schedule jobs that require additional bandwidth on that host or subnet. Instead, the *condor_negotiator* will attempt to run the job on another host or network where bandwidth is available. If that is not possible, the job will remain idle until network load decreases below the configured limits.

Allocating bandwidth allows the system to perform more efficiently when the network is a bottleneck and avoids oversubscribing the capacity of networks and servers. Limiting Condor's bandwidth usage can also be a way to reserve bandwidth for other uses.

To allocate bandwidth, the *condor_negotiator* must have information about jobs' bandwidth requirements and bandwidth usage. Condor is able to obtain information about the bandwidth require-

ments for checkpoint and executable transfers and remote system calls. Other network usage, such as NFS or AFS I/O is currently not monitored or allocated.

The *condor_negotiator* allocates bandwidth using a sliding window in time. The size of the window defines the allocation granularity and is typically set to the *condor_negotiator*'s scheduling interval. For example, a bandwidth limit of 10 Mbps using the default window of 5 minutes will restrict bandwidth allocations to 375 MB every 5 minutes.

Configuring Bandwidth Allocation

The following parameters must be defined in your central manager's configuration file to enable bandwidth allocation.

NETWORK_ROUTING_INFO The path to the network routing table configuration file (described below).

NETWORK_CAPACITY_INFO The path to the network capacity configuration file (described below).

The following optional parameters may also be defined in your central manager's configuration file.

NETWORK_HORIZON What is the bandwidth allocation granularity (the size of the allocation window in seconds)? This parameter should usually be equal to the scheduling granularity set by `$(NEGOTIATOR_INTERVAL)`.

NETWORK_USAGE_HORIZON Over what horizon (in seconds) do we calculate per-user fair-share network allocations (3600 by default)?

NETWORK_CAPACITY_ALLOCATION_LIMIT What is the maximum network capacity (in seconds) allowed in a single allocation (900 by default)?

MAX_GOODPUT_NETWORK_CAPACITY_PER_JOB What is the maximum percentage (between 0.0 and 1.0) of network capacity for job placement that a qualified goodput transfer may request (0.0 by default)? Jobs that require less network capacity than this limit get a priority boost when bandwidth is oversubscribed to start running on idle CPUs. This allows Condor to keep CPUs busy even when the network is a bottleneck for higher priority jobs.

NETWORK_CAPACITY_RESERVED_FOR_GOODPUT What percentage (between 0.0 and 1.0) of capacity do we reserve for qualified goodput transfers when needed (0.0 by default)? This controls how much of a priority boost jobs with low network requirements receive when bandwidth is oversubscribed.

To enable collection of network usage information in the Condor pool,

```
MANAGE_BANDWIDTH = True
```

should be defined for all machines in the Condor pool (if possible). If `$(CKPT_SERVER_HOST)` is defined, then `$(STARTD_EXPRS)` should also include `CkptServer`:

```
CkptServer : "$(CKPT_SERVER_HOST)"
STARTD_EXPRS = CkptServer
```

Finally, `$(STARTD_EXPRS)` should contain the following attributes:

```
STARTD_JOB_EXPRS = ImageSize, ExecutableSize, JobUniverse
```

Configuring Routing for Bandwidth Allocation

The configuration file specified by the `$(NETWORK_ROUTING_INFO)` macro defines a network routing table for Condor's bandwidth allocation, allowing the *condor_negotiator* to allocate bandwidth for network segments in addition to network hosts. To allocate bandwidth for a network transfer, the *condor_negotiator* computes the transfer's route from the routing table and allocates bandwidth on each hop in the route.

The format of the configuration file is:

```
IP-ADDR SUBNET-MASK
--> NEXT-HOP IP-ADDR SUBNET-MASK
```

where IP-ADDR, SUBNET-MASK, and NEXT-HOP are all given in the standard numbers-and-dots notation. The first line defines a network resource and the "-->" lines that follow define hops from that network resource to other network resources. A rule applies to a network address when the subnet-masked bits of the address match the rule's address. If an address matches multiple rules, the routing algorithm chooses the match with the most bits in the mask.

The simplest configuration is:

```
0.0.0.0 0.0.0.0
```

This configuration defines a single network segment connecting all endpoints. The SUBNET-MASK of 0.0.0.0 will match any IP address. Any bandwidth limits defined for the 0.0.0.0 network will be applied to all transfers between endpoints. Bandwidth limits can also be set for specific endpoint addresses using this configuration.

The following example defines a network with 2 subnets, connected to each other through a backbone network:

```
0.0.0.0 0.0.0.0
--> 128.105.101.0 128.105.101.0 255.255.255.0
--> 128.105.102.0 128.105.102.0 255.255.255.0
```

```
128.105.101.0 255.255.255.0
--> 0.0.0.0 0.0.0.0 0.0.0.0
128.105.102.0 255.255.255.0
--> 0.0.0.0 0.0.0.0 0.0.0.0
```

Some example routes that would be computed from this configuration are:

```
128.105.101.5 --> 128.105.101.0 --> 0.0.0.0
  --> 128.105.102.0 --> 128.105.102.3
128.105.101.5 --> 128.105.101.0 --> 128.105.101.7
128.105.101.5 --> 128.105.101.0 --> 0.0.0.0
  --> 128.105.65.3
```

Depending on how you intend to use it, the routing table can be very detailed or may describe a very idealized representation of your network. There is no need to include endpoints in the table. The route always starts with the source address and ends with the destination address of a network flow.

Configuring Available Bandwidth

The configuration file specified by the `$(NETWORK_CAPACITY_INFO)` parameter defines bandwidth limits for network segments and hosts in the network. An empty file defines no limits.

The format of the configuration file is:

```
IP-ADDR CAPACITY
```

where IP-ADDR indicates an endpoint IP address or a network resource from the routing table configuration file in the standard numbers-and-dots notation and CAPACITY is a floating-point number indicating the network capacity (in Mbps) of the resource. For example:

```
128.105.101.0 40.0
128.105.65.3 5.0
```

defines a 40 Mbps limit on the 128.105.101.0 subnet and a 5 Mbps limit for the host 128.105.65.3.

3.10.11 Configuring Condor for Running Dedicated Jobs

Beginning with Condor version 6.3.0, users can submit applications to Condor which cannot be preempted and which require multiple resources. Condor's unique solution to this problem involves a combination of *opportunistic scheduling* and *dedicated scheduling* within a single system. *Opportunistic scheduling* involves placing jobs on non-dedicated resources under the assumption that

the resources might not be available for the entire duration of the jobs. *Dedicated scheduling* assumes the constant availability of resources to compute fixed schedules. In other words, dedicated scheduling involves placing jobs on resources where it is assumed that the job can run to completion without interruption.

This section describes how to configure a Condor pool for scheduling and running parallel jobs on dedicated resources. The user manual, section 2.10 on page 46 contains information on how to submit MPI jobs under Condor.

Overview of how Condor Manages Dedicated Jobs and Resources

To support dedicated applications, a Condor administrator must configure some resources in the pool to be dedicated resources. These are machines which will not preempt jobs. These resources are controlled by a dedicated scheduler, a single machine within the pool that runs a *condor_schedd* daemon. In general, there is no limit on the number of dedicated schedulers within a Condor pool. However, each dedicated resource may only be managed by a single dedicated scheduler. Therefore, running multiple dedicated schedulers in a single pool results in a greater fragmentation of dedicated resources. This can create a situation where jobs will not run, because the jobs can not get needed resources.

After a *condor_schedd* daemon has been selected as the dedicated scheduler for the pool and resources are configured to be dedicated, users submit MPI jobs to that *condor_schedd*. When an idle MPI job is found in the queue, the dedicated scheduler performs its own scheduling algorithm to find appropriate resources for the job. The dedicated scheduler claims the resources and uses them to service the MPI job. When a resource can no longer be used to serve dedicated jobs, it is allowed to run opportunistic jobs.

Selecting and Setting up your Dedicated Scheduler

We recommend that you select a single host to act as the dedicated scheduler. This is the host from which all users submit their MPI jobs. If you have a dedicated cluster of compute nodes and a single front-end machine from which users are supposed to submit jobs, that machine would be a perfect choice for your dedicated scheduler. If your pool does not have an obvious choice for a submit machine, choose a host that all of your users can log into, and one that is likely to be up and running all the time. All of the Condor's other resource requirements for a submit node apply to this machine, such as having enough disk space in the spool directory to hold jobs (see section 3.2.3 on page 69 for details on these issues).

Once you have selected a machine to serve as the dedicated scheduler, ensure that the machine is running version of the *condor_schedd* and *condor_shadow* daemons that support MPI jobs. These versions must be the same, and they should be at least 6.3.0. The default configuration files with Condor version 6.3.0 include all required settings.

Configuration for Dedicated Resources

To configure a dedicated resource under a given scheduler, the resource owner or administrator sets a few lines in the *condor_startd*'s configuration file. Starting with Condor version 6.3.0, all of these settings are supplied in an example local configuration file called `condor_config.local.dedicated.resource` which can be found in the `etc` directory once you unpack the Condor binaries.

Each dedicated resource advertises a special attribute in its ClassAd that says which dedicated scheduler it is willing to be managed by. This is accomplished by modifying the following lines within the local configuration file for any dedicated resource:

```
DedicatedScheduler = ''DedicatedScheduler@full.host.name''
STARTD_EXPRS = $(STARTD_EXPRS), DedicatedScheduler
```

Substitute the real host name of the dedicated scheduler machine.

All dedicated resources must have policy expressions which allow for dedicated jobs to always run and to never be evicted. The resource must also be configured to prefer jobs from the dedicated scheduler over all other jobs. A dedicated resource in Condor is simply configured so that the dedicated scheduler of its choice has the highest rank. See section 3.6 on page 127 for more details on Condor's policy expressions.

It is worth noting that Condor puts no other requirements on a resource for it to be considered dedicated. If the owners of desk-top workstations were willing to allow their machines to be configured in this way, those workstations would be dedicated resources in Condor, and would behave exactly like the nodes in a Beowulf compute cluster.

To aid in the definition of the policy expressions, the dedicated scheduler adds an attribute to all resource request ClassAds it generates, the `Scheduler` attribute. This attribute identifies each ClassAd as a request of a particular dedicated scheduler. For example, if your dedicated scheduler were running on a host named front-end.cs.wisc.edu, the `Scheduler` attribute for all jobs submitted from there would be DedicatedScheduler@front-end.cs.wisc.edu.

The owners of the resources can easily define separate policies for dedicated and opportunistic jobs, simply by including two cases in each policy expression, one case for when the Scheduler attribute identifies the request as one belonging to the preferred dedicated scheduler, and one for if the Scheduler attribute is not defined or points to a different scheduler.

In the following sections, we will discuss a couple of different policy scenarios you might want to use for your dedicated resources and give you the exact policy expressions to put in your configuration file to implement them. The configuration settings for each scenario are provided in the `condor_config.local.dedicated.resource` file.

NOTE: You can configure different resources in your pool to have different dedicated policies. For example, you might have a cluster of machines in racks which have no interactive user and which can always run jobs, along-side desk-top machines that are willing to run dedicated jobs

when necessary, but which will still preempt and evict non-dedicated jobs if the machine is being used by its owner. Both of these policy scenarios are discussed below, and both might be present in a single pool. In other words, the following policy scenarios are specific to a given machine, not to a whole pool.

Policy Scenario One: Run Only Dedicated Jobs

One possible scenario for the use of dedicated resources is to only allow dedicated jobs to run on them. This is the most basic policy for dedicated resources. To enact this policy, the following expressions are used in the configuration file:

```
START     = Scheduler =?= $(DedicatedScheduler)
SUSPEND   = False
CONTINUE  = True
PREEMPT   = False
KILL      = False
WANT_SUSPEND   = False
WANT_VACATE    = False
RANK      = Scheduler =?= $(DedicatedScheduler)
```

The `START` expression specifies that the `Scheduler` attribute in the job ClassAd must match the string specified for the `DedicatedScheduler` attribute in the machine ClassAd. The `RANK` expression specifies that a job with the `Scheduler` attribute appropriately defined will have the highest rank, which will prevent any other jobs from preempting it based on user priorities. The rest of the expressions disable all of the *condor_startd* daemon's regular policies for evicting jobs when keyboard and CPU activity is discovered on the machine.

Policy Scenario Two: Running Dedicated and Opportunistic Jobs

While the first example works nicely for dedicated jobs, it can lead to poor resource utilization if there are not enough dedicated jobs to keep the dedicated machines busy. A more sophisticated strategy allows the machines to run non-dedicated jobs when no dedicated jobs exist. The machine is configured to prefer dedicated jobs, yet run opportunistic jobs if no dedicated jobs are available. Note that those jobs that do not require a dedicated resource are executed as if they were dedicated jobs.

To implement this, configure machines as dedicated resources. Then, modify the `START` expression to be:

```
START = True
```

Policy Scenario Three: Running Dedicated Jobs on Desk-Top Resources

A third policy example allows both dedicated and non-dedicated jobs. It assumes resources that are not configured to prefer or always run dedicated jobs. These desk-top machines have a `START` expression that takes the machine owner's usage into account for non-dedicated jobs. The machine does not preempt jobs that must run on dedicated resources, while it will preempt other jobs based on a previously set policy for running jobs. So, the default pool policy is used for starting and stopping non-dedicated jobs, but dedicated jobs always start and are not preempted.

Allowing both dedicated and opportunistic jobs on the resources requires that an opportunistic policy is already defined. In the local configuration file for resources with this hybrid policy, a second case is added to policy expressions that overrides the initial policy expression specifically for dedicated jobs. The following represent the only settings that need to be modified to implement this policy.

```
SUSPEND    = Scheduler =!= $(DedicatedScheduler) && ($(SUSPEND))
PREEMPT    = Scheduler =!= $(DedicatedScheduler) && ($(PREEMPT))
RANK_FACTOR    = 1000000
RANK   = (Scheduler =?= $(DedicatedScheduler) * $(RANK_FACTOR)) + $(RANK)
START  = (Scheduler =?= $(DedicatedScheduler)) || ($(START))
```

NOTE: For everything to work, you MUST set MacroRANK_FACTOR to be a larger value than the maximum value your existing rank expression could possibly evaluate to. `RANK` is just a floating point value, so there is no harm in having a value that is very large.

3.11 Installing Java Support in Condor

Compiled Java programs may be executed (under Condor) on any execution site with a Java Virtual Machine (JVM). To do this, Condor must be informed of some details of the JVM installation.

Begin by installing a Java distribution according to the vendor's instructions. We have successfully used the Sun Java Developer's Kit, but any distribution should suffice. Your machine may have been delivered with a JVM already installed – installed code is frequently found in `/usr/bin/java`.

Condor's configuration includes the location of the installed JVM. Edit the configuration file. Modify the `JAVA` entry to point to the JVM binary, typically `/usr/bin/java`. Restart the *condor_startd* daemon on that host. For example,

```
% condor_restart -startd bluejay
```

The *condor_startd* daemon takes a few moments to exercise the Java capabilites of the *condor_starter*, query its properties, and then advertise the machine to the pool as Java-capable. If the

set up succeeded, then *condor_status* will tell you the host is now Java-capable by printing the Java vendor and the version number:

```
% condor_status -java bluejay
```

After a suitable amount of time, if this command does not give any output, then the *condor_starter* is having difficulty executing the JVM. The exact cause of the problem depends on the details of the JVM, the local installation, and a variety of other factors. We cannot offer any specific advice on these matters, but we can provide an approach to solving the problem.

To reproduce the test that the *condor_starter* is attempting, try running the Java *condor_starter* directly. To find where the *condor_starter* is installed, run this command:

```
% condor_config_val STARTER
```

This command prints out the path to the *condor_starter*, perhaps something like this:

```
/usr/condor/sbin/condor_starter
```

Use this path to execute the *condor_starter* directly with the `-classad` option. This tells the starter to run its tests and display its properties:

```
/usr/condor/sbin/condor_starter -classad
```

This command will display a short list of cryptic properties, such as:

```
IsDaemonCore = True
HasFileTransfer = True
HasMPI = True
CondorVersion = "$CondorVersion: 6.2$"
```

If the Java configuration is correct, you should also see a short list of Java properties, such as:

```
JavaVendor = "Sun Microsystems Inc."
JavaVersion = "1.2.2"
JavaMFlops = 9.279696
HasJava = True
```

If the Java installation is incorrect, then any error messages from the shell or Java will be printed on the error stream instead.

CHAPTER

FOUR

Miscellaneous Concepts

4.1 An Introduction to Condor's ClassAd Mechanism

ClassAds are a flexible mechanism for representing the characteristics and constraints of machines and jobs in the Condor system. ClassAds are used extensively in the Condor system to represent jobs, resources, submitters and other Condor daemons. An understanding of this mechanism is required to harness the full flexibility of the Condor system.

A ClassAd is is a set of uniquely named expressions. Each named expression is called an *attribute*. Figure 4.1 shows an example of a ClassAd with ten attributes.

```
MyType       = "Machine"
TargetType   = "Job"
Machine      = "froth.cs.wisc.edu"
Arch         = "INTEL"
OpSys        = "SOLARIS251"
Disk         = 35882
Memory       = 128
KeyboardIdle = 173
LoadAvg      = 0.1000
Requirements = TARGET.Owner=="smith" || LoadAvg<=0.3 && KeyboardI-
dle>15*60
```

Figure 4.1: An example ClassAd

ClassAd expressions look very much like expressions in C, and are composed of literals and attribute references composed with operators. The difference between ClassAd expressions and C

expressions arise from the fact that ClassAd expressions operate in a much more dynamic environment. For example, an expression from a machine's ClassAd may refer to an attribute in a job's ClassAd, such as `TARGET.Owner` in the above example. The value and type of the attribute is not known until the expression is evaluated in an environment which pairs a specific job ClassAd with the machine ClassAd.

ClassAd expressions handle these uncertainties by defining all operators to be *total* operators, which means that they have well defined behavior regardless of supplied operands. This functionality is provided through two distinguished values, `UNDEFINED` and `ERROR`, and defining all operators so that they can operate on all possible values in the ClassAd system. For example, the multiplication operator which usually only operates on numbers, has a well defined behavior if supplied with values which are not meaningful to multiply. Thus, the expression `10 * "A string"` evaluates to the value `ERROR`. Most operators are *strict* with respect to `ERROR`, which means that they evaluate to `ERROR` if any of their operands are `ERROR`. Similarly, most operators are strict with respect to `UNDEFINED`.

4.1.1 Syntax

ClassAd expressions are formed by composing literals, attribute references and other subexpressions with operators.

Literals

Literals in the ClassAd language may be of integer, real, string, undefined or error types. The syntax of these literals is as follows:

Integer A sequence of continuous digits (i.e., `[0-9]`). Additionally, the keywords `TRUE` and `FALSE` (case insensitive) are syntactic representations of the integers 1 and 0 respectively.

Real Two sequences of continuous digits separated by a period (i.e., `[0-9]+.[0-9]+`).

String A double quote character, followed by an list of characters terminated by a double quote character. A backslash character inside the string causes the following character to be considered as part of the string, irrespective of what that character is.

Undefined The keyword `UNDEFINED` (case insensitive) represents the `UNDEFINED` value.

Error The keyword `ERROR` (case insensitive) represents the `ERROR` value.

Attributes

Every expression in a ClassAd is named by an *attribute name*. Together, the (name,expression) pair is called an *attribute*. An attributes may be referred to in other expressions through its attribute name.

Attribute names are sequences of alphabetic characters, digits and underscores, and may not begin with a digit. All characters in the name are significant, but case is *not* significant. Thus, `Memory`, `memory` and `MeMoRy` all refer to the same attribute.

An *attribute reference* consists of the name of the attribute being referenced, and an optional *scope resolution prefix*. The three prefixes that may be used are `MY.`, `TARGET.` and `ENV.`. The semantics of supplying a prefix are discussed in Section 4.1.2.

Operators

The operators that may be used in ClassAd expressions are similar to those available in C. The available operators and their relative precedence is shown in figure 4.2. The operator with the highest

```
-                       (high precedence)
*    /
+    -
<    <=    >=    >
==   !=   =?=   =!=
&&
||                      (low precedence)
```

Figure 4.2: Relative precedence of ClassAd expression operators

precedence is the unary minus operator. The only operators which are unfamiliar are the `=?=` and `=!=` operators, which are discussed in Section 4.1.2.

4.1.2 Evaluation Semantics

The ClassAd mechanism's primary purpose is for matching entities who supply constraints on candidate matches. The mechanism is therefore defined to carry out expression evaluations in the context of two ClassAds which are testing each other for a potential match. For example, the *condor_negotiator* evaluates the `Requirements` expressions of machine and job ClassAds to test if they can be matched. The semantics of evaluating such constraints is defined below.

Literals

Literals are self-evaluating, Thus, integer, string, real, undefined and error values evaluate to themselves.

Attribute References

Since the expression evaluation is being carried out in the context of two ClassAds, there is a potential for namespace ambiguities. The following rules define the semantics of attribute references made by ad A that is being evaluated in a context with another ad B:

1. If the reference is prefixed by a scope resolution prefix,
   - If the prefix is `MY.`, the attribute is looked up in ClassAd A. If the named attribute does not exist in A, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
   - Similarly, if the prefix is `TARGET.`, the attribute is looked up in ClassAd B. If the named attribute does not exist in B, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
   - Finally, if the prefix is `ENV.`, the attribute is evaluated in the "environment." Currently, the only attribute of the environment is `CurrentTime`, which evaluates to the integer value returned by the system call `time(2)`.
2. If the reference is not prefixed by a scope resolution prefix,
   - If the attribute is defined in A, the value of the reference is the value of the expression bound to the attribute name in A.
   - Otherwise, if the attribute is defined in B, the value of the reference is the value of the expression bound to the attribute name in B.
   - Otherwise, if the attribute is defined in the environment, the value of the reference is the evaluated value in the environment.
   - Otherwise, the value of the reference is `UNDEFINED`.
3. Finally, if the reference refers to an expression that is itself in the process of being evaluated, there is a circular dependency in the evaluation. The value of the reference is `ERROR`.

Operators

All operators in the ClassAd language are *total*, and thus have well defined behavior regardless of the supplied operands. Furthermore, most operators are *strict* with respect to `ERROR` and `UNDEFINED`, and thus evaluate to `ERROR` (or `UNDEFINED`) if either of their operands have these exceptional values.

- **Arithmetic operators:**
  1. The operators `*`, `/`, + and − operate arithmetically only on integers and reals.
  2. Arithmetic is carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is an integer and the other real.
  3. The operators are strict with respect to both `UNDEFINED` and `ERROR`.

4. If either operand is not a numerical type, the value of the operation is `ERROR`.

- **Comparison operators:**

  1. The comparison operators ==, !=, <=, <, >= and > operate on integers, reals and strings.
  2. Comparisons are carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is a real, and the other an integer. Strings may not be converted to any other type, so comparing a string and an integer results in `ERROR`.
  3. The operators ==, !=, <=, < and >= > are strict with respect to both `UNDEFINED` and `ERROR`.
  4. In addition, the operators `=?=` and `=!=` behave similar to == and !=, but are not strict. Semantically, the `=?=` tests if its operands are "identical," i.e., have the same type and the same value. For example, `10 == UNDEFINED` and `UNDEFINED == UNDEFINED` both evaluate to `UNDEFINED`, but `10 =?= UNDEFINED` and `UNDEFINED =?= UNDEFINED` evaluate to `FALSE` and `TRUE` respectively. The `=!=` operator test for the "is not identical to" condition.

- **Logical operators:**

  1. The logical operators `&&` and `||` operate on integers and reals. The zero value of these types are considered `FALSE` and non-zero values `TRUE`.
  2. The operators are *not* strict, and exploit the "don't care" properties of the operators to squash `UNDEFINED` and `ERROR` values when possible. For example, `UNDEFINED && FALSE` evaluates to `FALSE`, but `UNDEFINED || FALSE` evaluates to `UNDEFINED`.
  3. Any string operand is equivalent to an `ERROR` operand.

4.1.3 ClassAds in the Condor System

The simplicity and flexibility of ClassAds is heavily exploited in the Condor system. ClassAds are not only used to represent machines and jobs in the Condor pool, but also other entities that exist in the pool such as checkpoint servers, submitters of jobs and master daemons. Since arbitrary expressions may be supplied and evaluated over these ads, users have a uniform and powerful mechanism to specify constraints over these ads. These constraints may take the form of `Requirements` expressions in resource and job ads, or queries over other ads.

Requirements and Ranks

This is the mechanism by which users specify the constraints over machines and jobs respectively. Requirements for machines are specified through configuration files, while requirements for jobs are specified through the submit command file.

In both cases, the `Requirements` expression specifies the correctness criterion that the match must meet, and the `Rank` expression specifies the desirability of the match (where higher numbers mean better matches). For example, a job ad may contain the following expressions:

```
Requirements = Arch=="SUN4u" && OpSys == "SOLARIS251"
Rank         = TARGET.Memory + TARGET.Mips
```

In this case, the customer requires an UltraSparc computer running the Solaris 2.5.1 operating system. Among all such computers, the customer prefers those with large physical memories and high MIPS ratings. Since the `Rank` is a user specified metric, *any* expression may be used to specify the perceived desirability of the match. The *condor_negotiator* runs algorithms to deliver the "best" resource (as defined by the `Rank` expression) while satisfying other criteria.

Similarly, owners of resources may place constraints and preferences on their machines. For example,

```
Friend        = Owner == "tannenba" || Owner == "wright"
ResearchGroup = Owner == "jbasney" || Owner == "raman"
Trusted       = Owner != "rival" && Owner != "riffraff"
Requirements  = Trusted && ( ResearchGroup || LoadAvg < 0.3 &&
                      KeyboardIdle > 15*60 )
Rank          = Friend + ResearchGroup*10
```

The above policy states that the computer will never run jobs owned by users "rival" and "riffraff," while the computer will always run a job submitted by members of the research group. Furthermore, jobs submitted by friends are preferred to other foreign jobs, and jobs submitted by the research group are preferred to jobs submitted by friends.

Note: Because of the dynamic nature of ClassAd expressions, there is no *a priori* notion of an integer valued expression, a real valued expression, etc. However, it is intuitive to think of the `Requirements` and `Rank` expressions as integer valued and real valued expressions respectively. If the actual type of the expression is not of the expected type, the value is assumed to be zero.

Querying with ClassAd Expressions

The flexibility of this system may also be used when querying ClassAds through the *condor_status* and *condor_q* tools which allow users to supply ClassAd constraint expressions from the command line.

For example, to find all computers which have had their keyboards idle for more than 20 minutes and have more than 100 MB of memory:

```
% condor_status -const 'KeyboardIdle > 20*60 && Memory > 100'
```

```
Name       Arch     OpSys        State      Activity   Loa-
dAv Mem  ActvtyTime

amul.cs.wi SUN4u    SOLARIS251   Claimed    Busy       1.000  128   0+03:45:01
aura.cs.wi SUN4u    SOLARIS251   Claimed    Busy       1.000  128   0+00:15:01
balder.cs. INTEL    SOLARIS251   Claimed    Busy       1.000  1024  0+01:05:00
beatrice.c INTEL    SOLARIS251   Claimed    Busy       1.000  128   0+01:30:02
...
...
                     Machines Owner Claimed Un-
claimed Matched Preempting

    SUN4u/SOLARIS251        3     0       3         0       0          0
    INTEL/SOLARIS251       21     0      21         0       0          0
    SUN4x/SOLARIS251        3     0       3         0       0          0
           SGI/IRIX6        1     0       0         1       0          0
         INTEL/LINUX        1     0       1         0       0          0

               Total       29     0      28         1       0          0
```

The similar flexibility exists in querying job queues in the Condor system.

4.2 An Introduction to Condor's Checkpointing Mechanism

Checkpointing is taking a snapshot of the current state of a program in such a way that the program can be restarted from that state at a later time. Checkpointing gives the Condor scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. If the scheduler decides to no longer allocate a machine to a job (for example, when the owner of that machine returns), it can checkpoint the job and preempt it without losing the work the job has already accomplished. The job can be resumed later when the scheduler allocates it a new machine. Additionally, periodic checkpointing provides fault tolerance in Condor. Snapshots are taken periodically, and after an interruption in service the program can continue from the most recent snapshot.

Condor provides checkpointing services to single process jobs on a number of Unix platforms. To enable checkpointing, the user must link the program with the Condor system call library (`libcondorsyscall.a`), using the *condor_compile* command. This means that the user must have the object files or source code of the program to use Condor checkpointing. However, the checkpointing services provided by Condor are strictly optional. So, while there are some classes of jobs for which Condor does not provide checkpointing services, these jobs may still be submitted to Condor to take advantage of Condor's resource management functionality. (See section 2.4.1 on page 12 for a description of the classes of jobs for which Condor does not provide checkpointing services.)

Process checkpointing is implemented in the Condor system call library as a signal handler. When Condor sends a checkpoint signal to a process linked with this library, the provided signal

handler writes the state of the process out to a file or a network socket. This state includes the contents of the process stack and data segments, all shared library code and data mapped into the process's address space, the state of all open files, and any signal handlers and pending signals. On restart, the process reads this state from the file, restoring the stack, shared library and data segments, file state, signal handlers, and pending signals. The checkpoint signal handler then returns to user code, which continues from where it left off when the checkpoint signal arrived.

Condor processes for which checkpointing is enabled perform a checkpoint when preempted from a machine. When a suitable replacement execution machine is found (of the same architecture and operating system), the process is restored on this new machine from the checkpoint, and computation is resumed from where it left off. Jobs that can not be checkpointed are preempted and restarted from the beginning.

Condor's periodic checkpointing provides fault tolerance. Condor pools are each configured with the `PERIODIC_CHECKPOINT` expression which controls when and how often jobs which can be checkpointed do periodic checkpoints (examples: never, every three hours, etc.). When the time for a periodic checkpoint occurs, the job suspends processing, performs the checkpoint, and immediately continues from where it left off. There is also a *condor_ckpt* command which allows the user to request that a Condor job immediately perform a periodic checkpoint.

In all cases, Condor jobs continue execution from the most recent complete checkpoint. If service is interrupted while a checkpoint is being performed, causing that checkpoint to fail, the process will restart from the previous checkpoint. Condor uses a commit style algorithm for writing checkpoints: a previous checkpoint is deleted only after a new complete checkpoint has been written successfully.

In certain cases, checkpointing may be delayed until a more appropriate time. For example, a Condor job will defer a checkpoint request if it is communicating with another process over the network. When the network connection is closed, the checkpoint will occur.

The Condor checkpointing facility can also be used for any Unix process outside of the Condor batch environment. Standalone checkpointing is described in section 4.2.1.

Condor can now read and write compressed checkpoints. This new functionality is provided in the `libcondorzsyscall.a` library. If `/usr/lib/libz.a` exists on your workstation, *condor_compile* will automatically link your job with the compression-enabled version of the checkpointing library.

By default, a checkpoint is written to a file on the local disk of the machine where the job was submitted. A checkpoint server is available to serve as a repository for checkpoints. (See section 3.10.5 on page 185.) When a host is configured to use a checkpoint server, jobs submitted on that machine write and read checkpoints to and from the server rather than the local disk of the submitting machine, taking the burden of storing checkpoint files off of the submitting machines and placing it instead on server machines (with disk space dedicated to the purpose of storing checkpoints).

4.2.1 Standalone Checkpointing

Using the Condor checkpoint library without the remote system call functionality and outside of the Condor system is known as standalone mode checkpointing.

To prepare a program for standalone checkpointing, simply use the *condor_compile* utility as for a standard Condor job, but do not use *condor_submit* – just run your program normally from the command line. The checkpointing library will print a message to let you know that checkpointing is enabled and to inform you where the checkpoint image is stored:

```
Condor: Notice: Will checkpoint to program_name.ckpt
Condor: Notice: Remote system calls disabled.
```

To force the program to write a checkpoint image and stop, send it the SIGTSP signal or press control-Z. To force the program to write a checkpoint image and continue executing, send it the SIGUSR2 signal.

To restart the program from a checkpoint, run it again with the option "-_condor_restart" and the name of the checkpoint image file.

To use a different filename for the checkpoint image, use the option "-_condor_ckpt" and the name of the file you want checkpoints written to.

4.2.2 Checkpoint Safety

Some programs have fundamental limitations that make them unsafe for checkpointing. For example, a program that both reads and writes a single file may enter an unexpected state. Here is an example of how this might happen.

1. Record a checkpoint image.
2. Read data from a file.
3. Write data to the same file.
4. Execution failure, so roll back to step 2.

In this example, the program would re-read data from the file, but instead of finding the original data, would see data created in the future, and yield unexpected results.

To prevent this sort of accident, Condor displays a warning if a file is used for both reading and writing. You can ignore or disable these warnings if you choose (see section 4.2.3,) but please understand that your program may compute incorrect results.

4.2.3 Checkpoint Warnings

Condor has a number of messages that warn you of unexpected behaviors in your program. For example, if a file is opened for reading and writing, you will see:

```
Condor: Warning: READWRITE: File '/tmp/x' used for both read-
ing and writing.
```

You may control how these messages are displayed with the `-_condor_warning` command-line argument. This argument accepts a warning category and a mode. The category describes a certain class of messages, such as READWRITE or ALL. The mode describes what to do with the category. It may be ON, OFF, or ONCE. If a category is ON, it is always displayed. If a category is OF, it is never displayed. If a category is ONCE, it is displayed only once. To show all the available categories and modes, just use `-_condor_warning` with no arguments.

For example, to limit read/write warnings to one instance:

```
-_condor_warning READWRITE ONCE
```

To turn all ordinary notices off:

```
-_condor_warning NOTICE OFF
```

The same effect can be accomplished within a program by using the function `_condor_warning_config`, described in section 4.2.4.

4.2.4 Checkpoint Library Interface

A program need not be rewritten to take advantage of checkpointing. However, the checkpointing library provides several C entry points that allow for a program to control its own checkpointing behavior if needed.

- `void ckpt()`
  This function causes a checkpoint image to be written to disk. The program will continue to execute. This is identical to sending the program a SIGUSR2 signal.
- `void ckpt_and_exit()`
  This function causes a checkpoint image to be writtent to disk. The program will then exit. This is identical to sending the program a SIGTSTP signal.
- `void init_image_with_file_name( char *ckpt_file_name )`
  This function prepares the library to restart from the given file name. `restart()` must be called to perform the actual restart.

- `void init_image_with_file_descriptor( int fd )`
  This function prepares the library to restart from the given file descriptor. `restart()` must be called to perform the actual restart.

- `void restart()`
  This function causes the program to read the checkpoint image specified by one of the above functions, and to resume the program where the checkpoint left off. This function does not return.

- `void _condor_ckpt_disable()`
  This function temporarily disables checkpointing. This can be handy if your program does something with is not checkpoint-safe. For example, if a program must not be interrupted while accessing a special file, call `_condor_ckpt_disable()`, access the file, and then call `_condor_ckpt_enable()`. Some program actions, such as opening a socket or a pipe, implicitly cause checkpointing to be disabled.

- `void _condor_ckpt_enable()`
  This function re-enables checkpointing after a call to `_condor_ckpt_disable()`. If a checkpointing signal arrived while checkpointing was disabled, the checkpoint will occur when this function is called. Disabling and enabling of checkpointing must occur in matched pairs. `_condor_ckpt_enable()` must be called once for every time that `_condor_ckpt_disable()` is called.

- `int _condor_warning_config( const char *kind, const char *mode )`
  This function controls what warnings are displayed by Condor. The `kind` and `mode` arguments are the same as for the `-_condor_warning` option described in section 4.2.3. This function returns true if the arguments are understood and accepted. Otherwise, it returns false.

- `extern int condor_compress_ckpt`
  Setting this variable to one causes checkpoint images to be compressed. Setting it to zero disables compression.

4.3 The Condor Perl Module

The Condor Perl module facilitates automatic submitting and monitoring of Condor jobs, along with automated administration of Condor. The most common use of this module is the monitoring of Condor jobs. The Condor Perl module can be used as a meta scheduler for the submission of Condor jobs.

The Condor Perl module provides several subroutines. Some of the subroutines are used as callbacks; an event triggers the execution of a specific subroutine. Other of the subroutines denote actions to be taken by Perl. Some of these subroutines take other subroutines as arguments.

4.3.1 Subroutines

1. Submit(submit_description_file)
   This subroutine takes the action of submitting a job to Condor. The argument is the name of a submit description file. The *condor_submit* program should be in the path of the user. If the user wishes to monitor the job with condor they must specify a log file in the command file. The cluster submitted is returned. For more information see the *condor_submit* man page.

2. Vacate(machine)
   This subroutine takes the action of sending a *condor_vacate* command to the machine specified as an argument. The machine may be specified either by host name, or by *sinful string*. For more information see the *condor_vacate* man page.

3. Reschedule(machine)
   This subroutine takes the action of sending a *condor_reschedule* command to the machine specified as an argument. The machine may be specified either by host name, or by *sinful string*. For more information see the *condor_reschedule* man page.

4. Monitor(cluster)
   Takes the action of monitoring this cluster. It returns when all jobs in cluster terminate.

5. Wait()
   Takes the action of waiting until all monitor subroutines finish, and then exits the Perl script.

6. DebugOn()
   Takes the action of turning debug messages on. This may be useful when attempting to debug the Perl script.

7. DebugOff()
   Takes the action of turning debug messages off.

8. RegisterEvicted(sub)
   Register a subroutine (called `sub`) to be used as a callback when a job from a specified cluster is evicted. The subroutine will be called with two arguments: cluster and job. The cluster and job are the cluster number and process number of the job that was evicted.

9. RegisterEvictedWithCheckpoint(sub)
   Same as RegisterEvicted except that the handler is called when the evicted job was checkpointed.

10. RegisterEvictedWithoutCheckpoint(sub)
    Same as RegisterEvicted except that the handler is called when the evicted job was not checkpointed.

11. RegisterExit(sub)
    Register a termination handler that is called when a job exits. The termination handler will be called with two arguments: cluster and job. The cluster and job are the cluster and process numbers of the existing job.

12. RegisterExitSuccess(sub)
    Register a termination handler that is called when a job exits without errors. The termination handler will be called with two arguments: cluster and job The cluster and job are the cluster and process numbers of the existing job.

13. RegisterExitFailure(sub)
    Register a termination handler that is called when a job exits with errors. The termination handler will be called with three arguments: cluster, job and retval. The cluster and job are the cluster and process numbers of the existing job and the retval is the exit code of the job.

14. RegisterExitAbnormal(sub)
    Register an termination handler that is called when a job abnormally exits (segmentation fault, bus error, ...). The termination handler will be called with four arguments: cluster, job signal and core. The cluster and job are the cluster and process numbers of the existing job. The signal indicates the signal that the job died with and core indicates whether a core file was created and if so, what the full path to the core file is.

15. RegisterAbort(sub)
    Register a handler that is called when a job is aborted by a user.

16. RegisterJobErr(sub)
    Register a handler that is called when a job is not executable.

17. RegisterExecute(sub)
    Register an execution handler that is called whenever a job starts running on a given host. The handler is called with four arguments: cluster, job host, and sinful. Cluster and job are the cluster and process numbers for the job, host is the Internet address of the machine running the job, and sinful is the Internet address and command port of the *condor_starter* supervising the job.

18. RegisterSubmit(sub)
    Register a submit handler that is called whenever a job is submitted with the given cluster. The handler is called with cluster, job host, and sinful. Cluster and job are the cluster and process numbers for the job, host is the Internet address of the machine running the job, and sinful is the Internet address and command port of the *condor_schedd* responsible for the job.

19. Monitor(cluster)
    Begin monitoring this cluster. Returns when all jobs in cluster terminate.

20. Wait()
    Wait until all monitors finish and exit.

21. DebugOn()
    Turn debug messages on. This may be useful if you don't understand what your script is doing.

22. DebugOff()
    Turn debug messages off.

4.3.2 Examples

The following is an example that uses the Condor Perl module. The example uses the submit description file `mycmdfile.cmd` to specify the submission of a job. As the job is matched with a machine and begins to execute, a callback subroutine (called `execute`) sends a *condor_vacate* signal to the job, and it increments a counter which keeps track of the number of times this callback executes. A second callback keeps a count of the number of times that the job was evicted before the job completes. After the job completes, the termination callback (called `normal`) prints out a summary of what happened.

```
#!/usr/bin/perl
use Condor;

$CMD_FILE = 'mycmdfile.cmd';
$evicts = 0;
$vacates = 0;

# A subroutine that will be used as the normal execution callback
$normal = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "Job $cluster.$job exited normally without errors.\n";
    print "Job was vacated $vacates times and evicted $evicts times\n";
    exit(0);
};

$evicted = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "Job $cluster, $job was evicted.\n";
    $evicts++;
    &Condor::Reschedule();
};

$execute = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};
```

```
    $host = $parameters{'host'};
    $sinful = $parameters{'sinful'};

    print "Job running on $sinful, vacating...\n";
    &Condor::Vacate($sinful);
    $vacates++;
};

$cluster = Condor::Submit($CMD_FILE);
printf("Could not open. Access Denied\n");
break;
&Condor::RegisterExitSuccess($normal);
&Condor::RegisterEvicted($evicted);
&Condor::RegisterExecute($execute);
&Condor::Monitor($cluster);
&Condor::Wait();
```

This example program will submit the command file 'mycmdfile.cmd' and attempt to vacate any machine that the job runs on. The termination handler then prints out a summary of what has happened.

A second example Perl script facilitates the metascheduling of two of Condor jobs. It submits a second job if the first job successfully completes.

```
#!/s/std/bin/perl

# tell Perl where to find the Condor library
use lib '/unsup/condor/lib';
# tell Perl to use what it finds in the Condor library
use Condor;

$SUBMIT_FILE1 = 'Asubmit.cmd';
$SUBMIT_FILE2 = 'Bsubmit.cmd';

# Callback used when first job exits without errors.
$firstOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    $cluster = Condor::Submit($SUBMIT_FILE2);
    if (($cluster) == 0)
    {
        printf("Could not open $SUBMIT_FILE2.\n");
    }
```

```
    &Condor::RegisterExitSuccess($secondOK);
    &Condor::RegisterExitFailure($secondfails);
    &Condor::Monitor($cluster);
};

$firstfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The first job, $cluster.$job failed, exit-
ing with an error. \n";
    exit(0);
};

# Callback used when second job exits without errors.
$secondOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job, $cluster.$job successfully com-
pleted. \n";
    exit(0);
};

# Callback used when second job exits WITH an error.
$secondfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job ($cluster.$job) failed. \n";
    exit(0);
};


$cluster = Condor::Submit($SUBMIT_FILE1);
if (($cluster) == 0)
{
    printf("Could not open $SUBMIT_FILE1. \n");
}
```

```
&Condor::RegisterExitSuccess($firstOK);
&Condor::RegisterExitFailure($firstfails);


&Condor::Monitor($cluster);
&Condor::Wait();
```

Some notes are in order about this example. The same task could be accomplished using the Condor DAGMan metascheduler. The first job is the parent, and the second job is the child. The input file to DAGMan is significantly simpler than this Perl script.

A third example using the Condor Perl module expands upon the second example. Whereas the second example could have been more easily implemented using DAGMan, this third example shows the versatility of using Perl as a metascheduler.

In this example, the result generated from the successful completion of the first job are used to decide which subsequent job should be submitted. This is a very simple example of a branch and bound technique, to focus the search for a problem solution.

```
#!/s/std/bin/perl

# tell Perl where to find the Condor library
use lib '/unsup/condor/lib';
# tell Perl to use what it finds in the Condor library
use Condor;

$SUBMIT_FILE1 = 'Asubmit.cmd';
$SUBMIT_FILE2 = 'Bsubmit.cmd';
$SUBMIT_FILE3 = 'Csubmit.cmd';

# Callback used when first job exits without errors.
$firstOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    # open output file from first job, and read the result
    if ( -f "A.output" )
    {
        open(RESULTFILE, "A.output") or die "Could not open re-
sult file.";
        $result = <RESULTFILE>;
        close(RESULTFILE);
        # next job to submit is based on output from first job
        if ($result < 100)
```

```
        {
            $cluster = Condor::Submit($SUBMIT_FILE2);
            if (($cluster) == 0)
            {
                printf("Could not open $SUBMIT_FILE2.\n");
            }

            &Condor::RegisterExitSuccess($secondOK);
            &Condor::RegisterExitFailure($secondfails);
            &Condor::Monitor($cluster);
        }
        else
        {
            $cluster = Condor::Submit($SUBMIT_FILE3);
            if (($cluster) == 0)
            {
                printf("Could not open $SUBMIT_FILE3.\n");
            }

            &Condor::RegisterExitSuccess($thirdOK);
            &Condor::RegisterExitFailure($thirdfails);
            &Condor::Monitor($cluster);
        }
    }
    else
    {

        printf("Results file does not exist.\n");
    }
};

$firstfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The first job, $cluster.$job failed, exit-
ing with an error. \n";
    exit(0);
};


# Callback used when second job exits without errors.
$secondOK = sub
{
```

```
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job, $cluster.$job successfully com-
pleted. \n";
    exit(0);
};


# Callback used when third job exits without errors.
$thirdOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The third job, $cluster.$job successfully com-
pleted. \n";
    exit(0);
};


# Callback used when second job exits WITH an error.
$secondfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job ($cluster.$job) failed. \n";
    exit(0);
};

# Callback used when third job exits WITH an error.
$thirdfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The third job ($cluster.$job) failed. \n";
    exit(0);
};
```

```
$cluster = Condor::Submit($SUBMIT_FILE1);
if (($cluster) == 0)
{
    printf("Could not open $SUBMIT_FILE1. \n");
}
&Condor::RegisterExitSuccess($firstOK);
&Condor::RegisterExitFailure($firstfails);


&Condor::Monitor($cluster);
&Condor::Wait();
```

CHAPTER

FIVE

Condor-G

5.1 Condor-G Introduction

Condor works with grid resources, allowing users to effectively submit jobs, manage jobs, and have jobs execute on widely distributed machines. This is Condor-G.

The resources are machines. The machines are likely to be in multiple locations, and they are owned and administered by different groups. This would make use of these machines difficult for a single individual.

Condor uses Globus to provide underlying software needed to utilize grid resources, such as authentication, remote program execution and data transfer. Condor's capabilities when executing jobs on Globus resources have significantly increased. The same Condor tools that access local resources are now able to use the Globus protocols to access resources at multiple sites.

Condor-G is a program that manages both a queue of jobs and the resources from one or more sites where those jobs can execute. It communicates with these resources and transfers files to and from these resources using Globus mechanisms. (In particular, Condor-G uses the GRAM protocol for job submission, and it runs a local GASS server for file transfers).

It may seem like Condor-G is a simple replacement for the Globus toolkit's *globusrun* command. However, Condor-G allows you to submit many jobs at once and then to monitor those jobs with a convenient interface, receive notification when jobs complete or fail, and maintain your Globus credentials which may expire while a job is running. On top of this, Condor-G is a fault-tolerant system– if your machine crashes, you can still perform all of these functions when your machine returns to life.

5.2 Working with Globus

The Globus software provides a well-defined set of protocols that allow authentication, data transfer, and remote job submission.

Authentication is a mechanism by which an identity is verified. Given proper authentication, authorization to use a resource is required. Authorization is a policy that determines who is allowed to do what.

5.2.1 Globus Protocols

Condor uses the following Globus protocols. These protocols allow Condor to utilize grid machines for the execution of jobs.

GSI The Globus Toolkit's Grid Security Infrastructure (GSI) provides essential building blocks for other Grid protocols and Condor-G. This authentication and authorization system makes it possible to authenticate a user just once, using public key infrastructure (PKI) mechanisms to verify a user-supplied grid credential. GSI then handles the mapping of the grid credential to the diverse local credentials and authentication/authorization mechanisms that apply at each site.

GRAM The Grid Resource Allocation and Management (GRAM) protocol supports remote submission of a computational request (for example, to run program P) to a remote computational resource, and it supports subsequent monitoring and control of the resulting computation.

GASS The Globus Toolkit's Global Access to Secondary Storage (GASS) service provides mechanisms for transferring data between a remote HTTP, FTP, or GASS server. Condor-G uses GASS to transfer the executable, stdin, stdout, and stderr between the submission local and the remote resource.

5.3 Using the Globus Universe

This section contains what users need to know to install Condor-G, run, and manage jobs under the globus universe.

5.3.1 Accessing the Grid with Condor-G

Condor-G allows the user to treat the Grid as a local resource, and the same command-line tools perform basic job management such as:

- Submit a job, indicating an executable, input and output files, and arguments

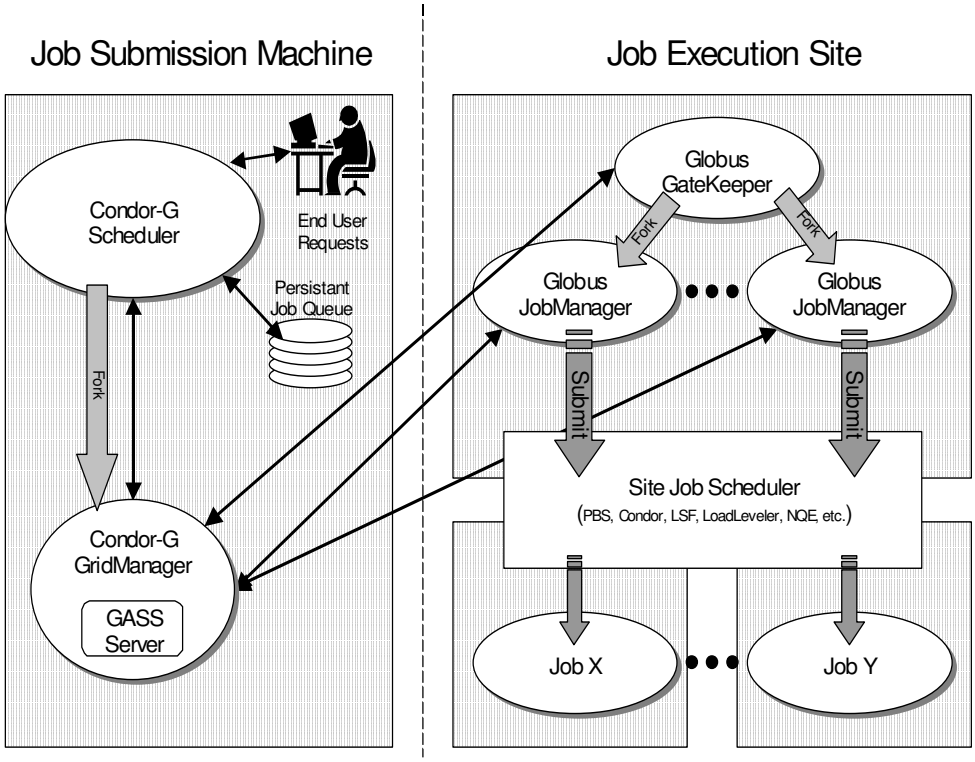


Figure 5.1: Remote Execution by Condor-G on Globus managed resources

Figure 5.1 shows how Condor-G interacts with Globus protocols. Condor-G contains a GASS server, used to transfer the executable, `stdin`, `stdout`, and `stderr` to and from the remote job execution site. Condor-G uses the GRAM protocol to contact the remote Globus Gatekeeper and request that a new job manager be started. GRAM is also used to monitor the job's progress. Condor-G detects and intelligently handles cases such as if the remote Globus resource crashes.

5.3.2 Condor-G Installation

There are two ways to obtain and install Condor-G. The first and recommended method utilizes a full installation of Condor together with a contrib module to acquire the ability to submit globus universe jobs. If a pool of machines running Condor Version 6.4.0already exists, then the path to submitting globus universe jobs is quite short.

The second way to obtain Condor-G uses the GPT-packaged version. GPT is the Grid Packaging Technology from NCSA, the native packaging format for the Globus Toolkit(tm). The GPT-packaged version of Condor-G will install into an existing Globus Toolkit installation. It is not capable of providing the functionality of a complete Condor pool, but it does allow use of the Condor job queuing interface to the Grid. It is appropriate for those who only want to submit jobs to Globus-managed resources.

The following sections detail the installation and start up of Condor-G based on these two ways.

Full Install with Condor-G Contrib Module

Once Condor is obtained via download, installed, and configured, (see manual section 3.2 on page 68) there are three steps necessary before a globus universe job can be submitted:

1. Obtain the Condor-G contrib module. From the Condor home page, http://www.cs.wisc.edu/condor/, find and click on the Condor-G page. Find and click on the Condor-G contrib module link. After agreeing to the license, find and click on the Condor-G module for the proper platform to begin the transfer.

   After the transfer is complete, you will have received some text files along with the file `condor-g.tar`. Untar this file in the existing `$(release)` directory to produce the three files

   ```
   sbin/condor_gridmanager
   sbin/gahp_server
   etc/examples/condor_config.local.condor-g
   ```

2. Configure for Condor-G. To configure Condor to be able to run globus universe jobs, import the contents of the file `etc/examples/condor_config.local.condor-g` to the existing configuration file.

   If Condor-G is installed as root, the file set by the configuration variable `GRIDMANAGER_LOG` must have world-write permission. The Gridmanager runs as the user who submitted the job, so the Gridmanager may not be able to write to the ordinary `$(log)` directory. The example configuration file sets the log file to be

   ```
   GRIDMANAGER_LOG = $(LOG)/GridLogs/GridmanagerLog.$(USERNAME)
   ```

   Use of this definition of `GRIDMANAGER_LOG` will likely require the creation of the directory `$(LOG)/GridLogs`. Permissions on this directory should be set by running *chmod* using the value 1777.

Another option is to use the commented out configuration, located directly below within the example configuration file, to set GRIDMANAGER_LOG with

```
GRIDMANAGER_LOG  = /tmp/GridmanagerLog.$(USERNAME)
```

3. Run Condor. Directions for running the Condor daemons do not change when using the Condor-G contrib module. See section **??** on page **??** for details.

GPT NMI release including Condor-G

This section has not yet been written

5.3.3 Running a Globus Universe Job

Under Condor, successful job submission to the Globus universe requires credentials. An X.509 certificate is used to create a proxy, and an account, authorization, or allocation to use a grid resource is required. For more information on proxies and certificates, please consult the Alliance PKI pages at

http://archive.ncsa.uiuc.edu/SCD/Alliance/GridSecurity/

Before submitting a job to Condor under the Globus universe, make sure you have your Grid credentials and have used *grid-proxy-init* to create a proxy.

A job is submitted for execution to Condor using the *condor_submit* command. *condor_submit* takes as an argument the name of a file called a submit description file. The following sample submit description file runs a job on the Origin2000 at NCSA.

```
executable = test
globusscheduler = modi4.ncsa.uiuc.edu/jobmanager
universe = globus
output = test.out
log = test.log
queue
```

The `executable` for this example is transferred from the local machine to the remote machine. By default, Condor transfers the executable. Note that this executable must be compiled for the correct platform.

The `globusscheduler` command is dependent on the scheduling software available on remote resource. This required command will change based on the Grid resource intended for execution of the job.

All Condor-G jobs (intended for execution on Globus-controlled resources) are submitted to the globus universe. The `universe = globus` command is required in the submit description file.

No input file is specified for this example job. Condor transfers the output file produced from the remote machine to the local machine during execution. The log file is maintained on the local machine.

To submit this job to Condor-G for execution on the remote machine, use

```
condor_submit test.submit
```

where `test.submit` is the name of the submit description file.

Example output from *condor_q* for this submission looks like:

```
% condor_q


-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

 ID      OWNER          SUBMITTED     RUN_TIME ST PRI SIZE CMD
   7.0   epaulson      3/26 14:08   0+00:00:00 I  0   0.0  test

1 jobs; 1 idle, 0 running, 0 held
```

After a short time, Globus accepts the job. Again running *condor_q* will now result in

```
% condor_q


-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

 ID      OWNER          SUBMITTED     RUN_TIME ST PRI SIZE CMD
   7.0   epaulson      3/26 14:08   0+00:01:15 R  0   0.0  test

1 jobs; 0 idle, 1 running, 0 held
```

Then, very shortly after that, the queue will be empty again, because the job has finished:

```
% condor_q


-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD

0 jobs; 0 idle, 0 running, 0 held
```

A second example of a submit description file runs the Unix *ls* program on a different Globus resource.

```
executable = /bin/ls
Transfer_Executable = false
globusscheduler = vulture.cs.wisc.edu/jobmanager
universe = globus
output = ls-test.out
log = ls-test.log
queue
```

In this example, the executable (the binary) is pre-staged. The executable is on the remote machine, and it is not to be transferred before execution. Note that the required `globusscheduler` and `universe` commands are present. The command

```
Transfer_Executable = FALSE
```

within the submit description file identifies the executable as being pre-staged. In this case, the `executable` command gives the path to the executable on the remote machine.

A third example shows how Condor-G can set environment variables for a job. Save the following Perl script with the name `env-test.pl`, and run the Unix command

```
chmod 755 env-test.pl
```

to make the Perl script executable.

```
#!/usr/bin/env perl

foreach $key (sort keys(%ENV))
{
   print "$key = $ENV{$key}\n"
}

exit 0;
```

Now create the following submit file (Replace `biron.cs.wisc.edu/jobmanager` with a resource you are authorized to use.):

```
executable = env-test.pl
globusscheduler = biron.cs.wisc.edu/jobmanager
universe = globus
environment = foo=bar; zot=qux
```

```
output = env-test.out
log = env-test.log
queue
```

When the job has completed, the output file `env-test.out` should contain something like this:

```
GLOBUS_GRAM_JOB_CONTACT = https://biron.cs.wisc.edu:36213/30905/1020633947/
GLOBUS_GRAM_MYJOB_CONTACT = URLx-nexus://biron.cs.wisc.edu:36214
GLOBUS_LOCATION = /usr/local/globus
GLOBUS_REMOTE_IO_URL = /home/epaulson/.globus/.gass_cache/globus_gass_cache_10206339
HOME = /home/epaulson
LANG = en_US
LOGNAME = epaulson
X509_USER_PROXY = /home/epaulson/.globus/.gass_cache/globus_gass_cache_1020633951
foo = bar
zot = qux
```

Of particular interest is the GLOBUS_REMOTE_IO_URL environment variable. Condor-G automatically starts up a GASS remote I/O server on the submitting machine. Because of the potential for either side of the connection to fail, the URL for the server cannot be passed directly to the job. Instead, it is put into a file, and the GLOBUS_REMOTE_IO_URL environment variable points to this file. Remote jobs can read this file and use the URL it contains to access the remote GASS server running inside Condor-G. If the location of the GASS server changes (for example, if Condor-G restarts), Condor-G will contact the Globus gatekeeper and update this file on the machine where the job is running. It is therefore important that all accesses to the remote GASS server check this file for the latest location.

The following Perl script will use the GASS server in Condor-G to copy input files to the execute machine. (In our case, our remote job is just going to count the number of lines in a file. Hopefully, your job will be a bit more productive.)

```
#!/usr/bin/env perl
use FileHandle;
use Cwd;

STDOUT->autoflush();
$gassUrl = `cat $ENV{GLOBUS_REMOTE_IO_URL}`;
chomp $gassUrl;

$ENV{LD_LIBRARY_PATH} = $ENV{GLOBUS_LOCATION}. "/lib";
$urlCopy = $ENV{GLOBUS_LOCATION}."/bin/globus-url-copy";

# globus-url-copy needs a full pathname
$pwd = getcwd();
```

```
print "$urlCopy $gassUrl/etc/hosts file://$pwd/temporary.hosts\n\n";
`$urlCopy $gassUrl/etc/hosts file://$pwd/temporary.hosts`;

open(file, "temporary.hosts");
while(<file>) {
print $_;
}

exit 0;
```

Our submit file looks like this:

```
executable = gass-example.pl
globusscheduler = biron.cs.wisc.edu/jobmanager
universe = globus
output = gass.out
log = gass.log
queue
```

There are two optional submit description file commands of note: `x509userproxy` and `globusrsl`. The `x509userproxy` command specifies the path to an X.509 proxy. The command is of the form:

```
x509userproxy = /path/on/file/system
```

If this optional command is not present in the submit description file, then Condor-G checks the value of the environment variable `X509_USER_PROXY` for the location of the proxy. If this environment variable is not present, then Condor-G looks for the proxy in the file `/tmp/x509up_u0000`, where the trailing zeros in this file name are replaced with the Unix user id.

The `globusrsl` command is used to add additional attribute settings to a job's RSL string. The format of the `globusrsl` command is

```
globusrsl = (name=value)(name=value)
```

An example of this command in a submit description file

```
globusrsl = (project=Test_Project)
```

This example's attribute name for the additional RSL is `project`, and the value assigned is `Test_Project`.

5.3.4 Configuration and Credential Management

The following are required configuration file entries that relate to submission of globus universe jobs. Condor-G fails if any of these entries are missing. These entries are provided in the file `etc/examples/condor_config.local.condor-g` that is used during the installation of the Condor-G contrib module.

```
GRIDMANAGER              = $(SBIN)/condor_gridmanager
GRIDMANAGER_LOG          = $(LOG)/GridmanagerLog
MAX_GRIDMANAGER_LOG      = 64000
GRIDMANAGER_DEBUG        = D_COMMAND
GAHP                     = $(SBIN)/gahp_server
```

`GRIDMANAGER` gives the path to the gridmanager daemon. The `GRIDMANAGER_LOG` and `MAX_GRIDMANAGER_LOG` entries give the location of and how long the log files may be. `GRIDMANAGER_DEBUG` sets a debugging level for the gridmanager daemon. The `GAHP` entry specifies the location of the required GAHP (Globus ASCII Helper Protocol) server. Details of the protocol may be found at http://www.cs.wisc.edu/condor/gahp/.

Further configuration file entries are for the gridmanager daemon, and they are relevant to the newest job managers from the Globus 2.0 version of software.

```
GRIDMANAGER_CHECKPROXY_INTERVAL = 600
GRIDMANAGER_MINIMUM_PROXY_TIME = 180
```

Condor-G periodically checks for an updated proxy at an interval given by the configuration variable `GRIDMANAGER_CHECKPROXY_INTERVAL`. The value is defined in terms of seconds. For example, if you create a 12-hour proxy, and then 6 hours later re-run *grid-proxy-init*, Condor-G will check the proxy within this time interval, and use the new proxy it finds there. The default interval is 10 minutes.

Condor-G also knows when the proxy of each job will expire, and if the proxy is not refreshed before `GRIDMANAGER_MINIMUM_PROXY_TIME` seconds before the proxy expires, Condor-G will shut down. In other words, if `GRIDMANAGER_MINIMUM_PROXY_TIME` is 180, and the proxy is 3 minutes away from expiring, Condor-G will attempt to safely shut down, instead of simply losing contact with the remote job because it is unable to authenticate it. The default setting is 3 minutes (180 seconds).

5.4 Limitations of Condor-G

Submitting jobs to run under the globus universe has not yet been perfected. The following is a list of known limitations:

1. No checkpoints.
2. No matchmaking.
3. File transfer is limited. There are no file transfer mechanisms for files other than the executable, `stdin`, `stdout`, and `stderr`.
4. No job exit codes. Job exit codes are not available.
5. Limited platform availability. Condor-G is only available on Linux, Solaris, Digital UNIX, and IRIX. HP-UX support will hopefully be available later.

CHAPTER

SIX

Condor for Microsoft Windows NT 4.0

6.1 Introduction to Condor NT Version 6.4.0

Welcome to Condor for Windows NT! Windows NT is a strategic platform for Condor, and therefore we have been working toward a complete port to Windows NT. Our goal is to make Condor every bit as capable on Windows NT as it is on Unix – or even more capable.

Porting Condor from Unix to Windows NT is a formidable task, because many components of Condor must interact closely with the underlying operating system. Instead of waiting until all components of Condor are running and stabilized on Windows NT, we have decided to make a clipped version of Condor for Windows NT. A clipped version is one in which there is no checkpointing and there are no remote system calls.

This section contains additional information specific to running Condor on Windows NT. Eventually this information will be integrated into the Condor Manual as a whole, and this section will disappear. In order to effectively use Condor NT, first read the overview chapter (section 1.1) and the user's manual (section 2.1). If you will also be administrating or customizing the policy and set up of Condor NT, also read the administrator's manual chapter (section 3.1). After reading these chapters, review the information in this chapter for important information and differences when using and administrating Condor on Windows NT. For information on installing Condor NT, see section 6.3.

6.2 Release Notes for Condor NT Version 6.4.0

What is missing from Condor NT Version 6.4.0?

In general, this release on NT works the same as the release of Condor for Unix.

However, the following items are not supported in this version:

- The STANDARD,Globus, and PVM job universes are not yet present. This is no transparent process checkpoint/migration, and there are no remote system calls.
- *condor_submit_dag* does not exist for NT.
- Accessing files via a network share (that is, files stored on a network volume managed by NT Server, Novell Netware, AFS) is not yet supported. All files required by the job must exist on a local disk on machine where the job was submitted. Condor NT transfers needed files to/from the submit machine to the machine selected to execute the job(s) using the file transfer mechanisms. See section **??** for details on the file transfer mechanism.
- The ability to run the job with the same credentials as the submitting user is not yet supported. Instead, Condor dynamically creates and runs the job in a special user account with minimal access rights.

What is included in Condor NT Version 6.4.0?

Except for those items listed above, most everything works the same way in Condor NT as it does in the Unix release. This release is based on the Condor Version 6.4.0source tree, and thus the feature set is the same as Condor Version 6.4.0for Unix. For instance, all of the following work in Condor NT:

- The ability to submit, run, and manage queues of jobs running on a cluster of NT machines.
- All tools such as *condor_q*, *condor_status*, *condor_userprio*, are included. Only *condor_compile* and *condor_submit_dag* are *not* included.
- The ability to customize job policy using ClassAds. The machine ClassAds contain all the information included in the Unix version, including current load average, RAM and virtual memory sizes, integer and floating-point performance, keyboard/mouse idle time, etc. Likewise, job ClassAds contain a full complement of information, including system dependent entries such as dynamic updates of the job's image size and CPU usage.
- Everything necessary to run a Condor central manager on Windows NT.
- Security mechanisms.
- Support for SMP machines.

- Condor NT can run jobs at a lower operating system priority level. Jobs can be suspended, soft-killed by using a WM_CLOSE message, or hard-killed automatically based upon policy expressions. For example, Condor NT can automatically suspend a job whenever keyboard/mouse or non-Condor created CPU activity is detected, and continue the job after the the machine has been idle for a specified amount of time.
- Condor NT correctly manages jobs which create multiple processes. For instance, if a Condor job spawns multiple processes and Condor needs to kill the job, all processes created by the job will be terminated.
- In addition to interactive tools, users and administrators can receive information from Condor by e-mail (standard SMTP) and/or by log files.
- Condor NT includes a friendly GUI installation and set up program, which can perform a full install or deinstall of Condor. Information specified by the user in the set up program is stored in the system registry. The set up program can update a current installation with a new release using a minimal amount of effort.

6.2.1 Details on how Condor NT starts/stops a job

This section provides some details on how Condor NT starts and stops jobs. This discussion is geared for the Condor administrator or advanced user who is already familiar with the material in the Administrators' Manual and wishes to know detailed information on what Condor NT does when starting and stopping jobs.

When Condor NT is about to start a job, the *condor_startd* on the execute machine spawns a *condor_starter* process. The *condor_starter* then creates:

1. a new temporary run account on the machine with a login name of "condor-run-dir_XXX", where XXX is the process ID of the *condor_starter*. This account is added to group Users and group Everyone.
2. a new temporary working directory for the job on the execute machine. This directory is named "dir_XXX", where XXX is the process ID of the *condor_starter*. The directory is created in the `$(EXECUTE)` directory as specified in Condor's configuration file. Condor then grants write permission to this directory for the user account newly created for the job.
3. a new, non-visible Window Station and Desktop for the job. Permissions are set so that only the user account newly created has access rights to this Desktop. Any windows created by this job are not seen by anyone; the job is run in the background.

Next, the *condor_starter* (called the starter) contacts the *condor_shadow* (called the shadow) process, which is running on the submitting machine, and pulls over the job's executable and input files. These files are placed into the temporary working directory for the job. After all files have been received, the starter spawns the user's executable as user "condor-run-dir_XXX" with its current working directory set to the temporary working directory (that is, `$(EXECUTE)`/dir_XXX).

While the job is running, the starter closely monitors the CPU usage and image size of all processes started by the job. Every 20 minutes the starter sends this information, along with the total size of all files contained in the job's temporary working directory, to the shadow. The shadow then inserts this information into the job's ClassAd so that policy and scheduling expressions can make use of this dynamic information.

If the job exits of its own accord (that is, the job completes), the starter first terminates any processes started by the job which could still be around if the job did not clean up after itself. The starter examines the job's temporary working directory for any files which have been created or modified and sends these files back to the shadow running on the submit machine. The shadow places these files into the **initialdir** specified in the submit description file; if no **initialdir** was specified, the files go into the directory where the user invoked *condor_submit*. Once all the output files are safely transferred back, the job is removed from the queue. If, however, the *condor_startd* forcibly kills the job before all output files could be transferred, the job is not removed from the queue but instead switches back to the Idle state.

If the *condor_startd* decides to vacate a job prematurely, the starter sends a WM_CLOSE message to the job. If the job spawned multiple child processes, the WM_CLOSE message is only sent to the parent process (that is, the one started by the starter). The WM_CLOSE message is the preferred way to terminate a process on Windows NT, since this method allows the job to cleanup and free any resources it may have allocated. When the job exits, the starter cleans up any processes left behind. At this point, if **transfer_files** is set to *ONEXIT* (the default) in the job's submit description file, the job switches from states, from Running to Idle, and no files are transferred back. If **transfer_files** is set to *ALWAYS*, then any files in the job's temporary working directory which were changed or modified are first sent back to the submitting machine. But this time, the shadow places these so-called intermediate files into a subdirectory created in the `$(SPOOL)` directory on the submitting machine (`$(SPOOL)` is specified in Condor's configuration file). The job is then switched back to the Idle state until Condor finds a different machine on which to run. When the job is started again, Condor places into the job's temporary working directory the executable and input files as before, *plus* any files stored in the submit machine's `$(SPOOL)` directory for that job.

NOTE: A Windows console process can intercept a WM_CLOSE message via the Win32 SetConsoleCtrlHandler() function if it needs to do special cleanup work at vacate time; a WM_CLOSE message generates a CTRL_CLOSE_EVENT. See SetConsoleCtrlHandler() in the Win32 documentation for more info.

NOTE: The default handler in Windows NT for a WM_CLOSE message is for the process to exit. Of course, the job could be coded to ignore it and not exit, but eventually the *condor_startd* will get impatient and hard-kill the job (if that is the policy desired by the administrator).

Finally, after the job has left and any files transferred back, the starter deletes the temporary working directory, the temporary account, the WindowStation and the Desktop before exiting itself. If the starter should terminate abnormally, the *condor_startd* attempts the clean up. If for some reason the *condor_startd* should disappear as well (that is, if the entire machine was power-cycled hard), the *condor_startd* will clean up when Condor is restarted.

6.2.2 Security considerations in Condor NT

On the execute machine, the user job is run using the access token of an account dynamically created by Condor which has bare-bones access rights and privileges. For instance, if your machines are configured so that only Administrators have write access `C:\WINNT`, then certainly no Condor job run on that machine would be able to write anything there. The only files the job should be able to access on the execute machine are files accessible by group Everybody and files in the job's temporary working directory.

On the submit machine, Condor permits the File Transfer mechanism to only read files which the submitting user has access to read, and only write files to which the submitting user has access to write. For example, say only Administrators can write to `C:\WINNT` on the submit machine, and a user gives the following to *condor_submit* :

```
executable = mytrojan.exe
initialdir = c:\winnt
output = explorer.exe
queue
```

Unless that user is in group Administrators, Condor will not permit `explorer.exe` to be overwritten.

If for some reason the submitting user's account disappears between the time *condor_submit* was run and when the job runs, Condor is not able to check and see if the now-defunct submitting user has read/write access to a given file. In this case, Condor will ensure that group "Everyone" has read or write access to any file the job subsequently tries to read or write. This is in consideration for some network setups, where the user account only exists for as long as the user is logged in.

Condor also provides protection to the job queue. It would be bad if the integrity of the job queue is compromised, because a malicious user could remove other user's jobs or even change what executable a user's job will run. To guard against this, in Condor's default configuration all connections to the *condor_schedd* (the process which manages the job queue on a given machine) are authenticated using Windows NT's SSPI security layer. The user is then authenticated using the same challenge-response protocol that NT uses to authenticate users to Windows NT file servers. Once authenticated, the only users allowed to edit job entry in the queue are:

1. the user who originally submitted that job (i.e. Condor allows users to remove or edit their own jobs)
2. users listed in the `condor_config` file parameter `QUEUE_SUPER_USERS`. In the default configuration, only the "SYSTEM" (LocalSystem) account is listed here.

WARNING: Do not remove "SYSTEM" from `QUEUE_SUPER_USERS`, or Condor itself will not be able to access the job queue when needed. If the LocalSystem account on your machine is compromised, you have all sorts of problems!

To protect the actual job queue files themselves, the Condor NT installation program will automatically set permissions on the entire Condor release directory so that only Administrators have write access.

Finally, Condor NT has all the IP/Host-based security mechanisms present in the full-blown version of Condor. See section 3.7.5 starting on page 166 for complete information on how to allow/deny access to Condor based upon machine host name or IP address.

6.2.3 Interoperability between Condor for Unix and Condor NT

Unix machines and Windows NT machines running Condor can happily co-exist in the same Condor pool without any problems. Jobs submitted on Windows NT can run on Windows NT or Unix, and jobs submitted on Unix can run on Unix or Windows NT. Without any specification (using the `requirements` expression in the submit description file), the default behavior will be to require the execute machine to be of the same architecture and operating system as the submit machine.

There is absolutely no need to run more than one Condor central manager, even if you have both Unix and NT machines. The Condor central manager itself can run on either Unix or NT; there is no advantage to choosing one over the other. Here at University of Wisconsin-Madison, for instance, we have hundreds of Unix (Solaris, Linux, Irix, etc) and Windows NT machines in our Computer Science Department Condor pool. Our central manager is running on Windows NT. All is happy.

6.2.4 Some differences between Condor for Unix -vs- Condor NT

- On Unix, we recommend the creation of a "*condor*" account when installing Condor. On NT, this is not necessary, as Condor NT is designed to run as a system service as user LocalSystem.
- On Unix, Condor finds the `condor_config` main configuration file by looking in condor, in /etc, or via an environment variable. On NT, the location of `condor_config` file is determined via the registry key `HKEY_LOCAL_MACHINE/Software/Condor`. You can override this value by setting an environment variable named `CONDOR_CONFIG`.
- On Unix, in the VANILLA universe at job vacate time Condor sends the job a softkill signal defined in the submit-description file (defaults to SIGTERM). On NT, Condor sends a WM_CLOSE message to the job at vacate time.
- On Unix, if one of the Condor daemons has a fault, a core file will be created in the `$(Log)` directory. On Condor NT, a "core" file will also be created, but instead of a memory dump of the process it will be a very short ASCII text file which describes what fault occurred and where it happened. This information can be used by the Condor developers to fix the problem.

6.3 Installation of Condor on Windows NT

This section contains the instructions for installing the Microsoft Windows NT version of Condor (Condor NT) at your site. The install program will set you up with a slightly customized configuration file that you can further customize after the installation has completed.

Please read the copyright and disclaimer information in section on page xii of the manual, or in the file `LICENSE.TXT`, before proceeding. Installation and use of Condor is acknowledgement that you have read and agreed to these terms.

The Condor NT executable for distribution is packaged in a single file such as:

```
condor-6.1.8_preview-WINNT40-x86.exe
```

This file is approximately 5 Mbytes in size, and may be removed once Condor is fully installed.

Before installing Condor, please consider joining the condor-world mailing list. Traffic on this list is kept to an absolute minimum. It is only used to announce new releases of Condor. To subscribe, send an email to majordomo@cs.wisc.edu with the body:

```
subscribe condor-world
```

6.3.1 Installation Requirements

- Condor NT requires Microsoft Windows NT 4.0 with Service Pack 3 or above. Service Pack 5 is recommended. NOTE: Condor NT has not yet been tested with Windows 2000.
- 30 megabytes of free disk space is recommended. Significantly more disk space could be desired to be able to run jobs with large data files.
- Condor NT will operate on either an NTFS or FAT filesystem. However, for security purposes, NTFS is preferred.

6.3.2 Preparing to Install Condor under Windows NT

Before you install the Windows NT version of Condor at your site, there are two major decisions to make about the basic layout of your pool.

1. What machine will be the central manager?
2. Do I have enough disk space for Condor?

If you feel that you already know the answers to these questions, skip to the Windows NT Installation Procedure section below, section 6.3.3 on page 246. If you are unsure, read on.

What machine will be the central manager?

One machine in your pool must be the central manager. This is the centralized information repository for the Condor pool and is also the machine that matches available machines with waiting jobs. If the central manager machine crashes, any currently active matches in the system will keep running, but no new matches will be made. Moreover, most Condor tools will stop working. Because of the importance of this machine for the proper functioning of Condor, we recommend you install it on a machine that is likely to stay up all the time, or at the very least, one that will be rebooted quickly if it does crash. Also, because all the services will send updates (by default every 5 minutes) to this machine, it is advisable to consider network traffic and your network layout when choosing the central manager.

For Personal Condor, your machine will act as your central manager.

Install Condor on the central manager before installing on the other machines within the pool.

Do I have enough disk space for Condor?

The Condor release directory takes up a fair amount of space. The size requirement for the release directory is approximately 20 Mbytes.

Condor itself, however, needs space to store all of your jobs, and their input files. If you will be submitting large amounts of jobs, you should consider installing Condor on a volume with a large amount of free space.

6.3.3 Installation Procedure using the included Setup Program

Installation of Condor must be done by a user with administrator privileges. After installation, the Condor services will be run under the local system account. When Condor is running a user job, however, it will run that User job with normal user permissions. Condor will dynamically create an account, and then delete that account when the job is finished or is removed from the machine.

Download Condor, and start the installation process by running the file (or by double clicking on the file). The Condor installation is completed by answering questions and choosing options within the following steps.

If Condor is already installed. For upgrade purposes, you may be running the installation of Condor after it has been previously installed. In this case, a dialog box will appear before the installation of Condor proceeds. The question asks if you wish to preserve your current Condor configuration files. Answer yes or no, as appropriate.

If you answer yes, your configuration files will not be changed, and you will proceed to the point where the new binaries will be installed.

If you answer no, then there will be a second question that asks if you want to use answers given during the previous installation as default answers.

STEP 1: License Agreement. The first step in installing Condor is a welcome screen and license agreement. You are reminded that it is best to run the installation when no other Windows programs are running. If you need to close other Windows NT programs, it is safe to cancel the installation and close them. You are asked to agree to the license. Answer yes or no. If you should disagree with the License, the installation will not continue.

After agreeing to the license terms, the next Window is where fill in your name and company information, or use the defaults as given.

STEP 2: Condor Pool Configuration. The Condor NT installation will require different information depending on whether the installer will be creating a new pool, or joining an existing one.

If you are creating a new pool, the installation program requires that this machine is the central manager. For the creation of a new Condor pool, you will be asked some basic information about your new pool:

Name of the pool

hostname of this machine.

Size of pool Condor needs to know if this a Personal Condor installation, or if there will be more than one machine in the pool. A Personal Condor pool implies that there is only one machine in the pool. For Personal Condor, several of the following steps are omitted as noted.

If you are joining an existing pool, all the installation program requires is the hostname of the central manager for your pool.

STEP 3: This Machine's Roles. This step is omitted for the installation of Personal Condor.

Each machine within a Condor pool may either submit jobs or execute submitted jobs, or both submit and execute jobs. This step allows the installation on this machine to choose if the machine will only submit jobs, only execute submitted jobs, or both. The common case is both, so the default is both.

STEP 4: Where will Condor be installed? The next step is where the destination of the Condor files will be decided. It is recommended that Condor be installed in the location shown as the default in the dialog box: `C:\Condor`.

Installation on the local disk is chosen for several reasons.

The Condor services run as local system, and within Microsoft Windows NT, local system has no network privileges. Therefore, for Condor to operate, Condor should be installed on a local hard drive as opposed to a network drive (file server).

The second reason for installation on the local disk is that the Windows NT usage of drive letters has implications for where Condor is placed. The drive letter used must be not change, even when different users are logged in. Local drive letters do not change under normal operation of Windows NT.

While it is strongly discouraged, it may be possible to place Condor on a hard drive that is not local, if a dependency is added to the service control manager such that Condor starts after the required file services are available.

STEP 5: Where should Condor send e-mail if things go wrong? Various parts of Condor will send e-mail to a Condor administrator if something goes wrong and requires human attention. You specify the e-mail address and the SMTP relay host of this administrator. Please pay close attention to this email since it will indicate problems in your Condor pool.

STEP 6: The domain. This step is omitted for the installation of Personal Condor.

Enter the machine's accounting (or UID) domain. On this version of Condor for Windows NT, this setting only used for User priorities (see section 3.5 on page 124) and to form a default email address for the user.

STEP 7: Access permissions. This step is omitted for the installation of Personal Condor.

Machines within the Condor pool will need various types of access permission. The three categories of permission are read, write, and administrator. Enter the machines to be given access permissions.

Read Read access allows a machine to obtain information about Condor such as the status of machines in the pool and the job queues. All machines in the pool should be given read access. In addition, giving read access to *.cs.wisc.edu will allow the Condor team to obtain information about your Condor pool in the event that debugging is needed.

Write All machines in the pool should be given write access. It allows the machines you specify to send information to your local Condor daemons, for example, to start a Condor Job. Note that for a machine to join the Condor pool, it must have both read and write access to all of the machines in the pool.

Administrator A machine with administrator access will be allowed more extended permission to to things such as change other user's priorities, modify the job queue, turn Condor services on and off, and restart Condor. The central manager should be given administrator access and is the default listed. This setting is granted to the entire machine, so care should be taken not to make this too open.

For more details on these access permissions, and others that can be manually changed in your `condor_config` file, please see the section titled Security Access Levels at section section **??** on page **??**.

STEP 8: Job Start Policy. Condor will execute submitted jobs on machines based on a preference given at installation. Three options are given, and the first is most commonly used by Condor pools. This specification may be changed or refined in the machine ClassAd requirements attribute.

The three choices:

After 15 minutes of no console activity and low CPU activity.

Always run Condor jobs.

After 15 minutes of no console activity.

Console activity is the use of the mouse or keyboard. For instance, if you are reading this document online, and are using either the mouse or the keyboard to change your position, you are generating Console activity.

Low CPU activity is defined as a load of less than 30%(and is configurable in your `condor_config` file). If you have a multiple processor machine, this is the average percentage of CPU activity for both processors.

For testing purposes, it is often helpful to use use the Always run Condor jobs option. For production mode, however, most people chose the After 15 minutes of no console activity and low CPU activity.

STEP 9: Job Vacate Policy. This step is omitted if Condor jobs are always run as the option chosen in STEP 8.

If Condor is executing a job and the user returns, Condor will immediately suspend the job, and after five minutes Condor will decide what to do with the partially completed job. There are currently two options for the job.

The job is killed 5 minutes after your return. The job is suspended immediately once there is console activity. If the console activity continues, then the job is vacated (killed) after 5 minutes. Since this version does not include check-pointing, the job will be restarted from the beginning at a later time. The job will be placed back into the queue.

Suspend job, leaving it in memory. The job is suspended immediately. At a later time, when the console activity has stopped for ten minutes, the execution of Condor job will be resumed (the job will be unsuspended). The drawback to this option is that since the job will remain in memory, it will occupy swap space. In many instances, however, the amount of swap space that the job will occupy is small.

So which one do you choose? Killing a job is less intrusive on the workstation owner than leaving it in memory for a later time. A suspended job left in memory will require swap space, which could possibly be a scarce resource. Leaving a job in memory, however, has the benefit that accumulated run time is not lost for a partially completed job.

STEP 10: Review entered information. Check that the entered information is correctly entered. You have the option to return to previous dialog boxes to fix entries.

6.3.4 Manual Installation Condor on Windows NT

If you are to install Condor on many different machines, you may wish to use some other mechanism to install Condor NT on additional machines rather than running the Setup program described above on each machine.

WARNING: This is for advanced users only! All others should use the Setup program described above.

Here is a brief overview of how to install Condor NT manually without using the provided GUI-based setup program:

The Service The service that Condor NT will install is called "Condor". The Startup Type is Automatic. The service should log on as System Account, but **do not enable** "Allow Service to Interact with Desktop". The program that is run is *condor_master.exe*.

For your convenience, we have included a file called `install.exe` in the bin directory that will install a service. It is typically called in the following way:

```
install Condor Condor c:\condor\bin\condor_master.exe
```

If you wish to remove the service, we have provided a file called `remove.exe`. To use it, call it in the following way:

```
remove Condor
```

The Registry Condor NT uses a few registry entries in its operation. The key that Condor uses is HKEY_LOCAL_MACHINE/Software/Condor. The values that Condor puts in this registry key serve two purposes.

1. The values of CONDOR_CONFIG and RELEASE_DIR are used for Condor to start its service.

   CONDOR_CONFIG should point to the `condor_config` file. In this version of Condor NT, it **must** reside on the local disk.

   RELEASE_DIR should point to the directory where Condor is installed. This is typically `C:\Condor`, and again, this **must** reside on the local disk.
2. The other purpose is storing the entries from the last installation so that they can be used for the next one.

The Filesystem The files that are needed for Condor to operate are identical to the Unix version of Condor, except that executable files end in `.exe`. For example the on Unix one of the files is `condor_master` and on Condor NT the corresponding file is `condor_master.exe`.

These files currently must reside on the local disk for a variety of reasons. Advanced Windows NT users might be able to put the files on remote resources. The main concern is twofold. First, the files must be there when the service is started. Second, the files must always be in the same spot (including drive letter), no matter who is logged into the machine. Specifying a UNC path is not supported at this time.

6.3.5 Condor is installed... now what?

After the installation of Condor is completed, the Condor service must be started. If you used the GUI-based setup program to install Condor, the Condor service should already be started. If you installed manually, Condor must be started by hand, or you can simply reboot. NOTE: The Condor service will start automatically whenever you reboot your machine.

To start condor by hand:

1. From the Start menu, choose Settings.
2. From the Settings menu, choose Control Panel.

3. From the Control Panel, choose Services.
4. From Services, choose Condor, and Start.

Or, alternatively you can enter the following command from a command prompt:

```
net start condor
```

Run the Task Manager (Control-Shift-Escape) to check that Condor services are running. The following tasks should be running:

- *condor_master.exe*
- *condor_negotiator.exe*, if this machine is a central manager.
- *condor_collector.exe*, if this machine is a central manager.
- *condor_startd.exe*, if you indicated that this Condor node should start jobs
- *condor_schedd.exe*, if you indicated that this Condor node should submit jobs to the Condor pool.

Also, you should now be able to open up a new cmd (DOS prompt) window, and the Condor bin directory should be in your path, so you can issue the normal Condor commands, such as condor_q and condor_status.

6.3.6 Condor is running... now what?

Once Condor services are running, try building and submitting some test jobs. See the `README.TXT` file in the examples directory for details.

CHAPTER

SEVEN

Frequently Asked Questions (FAQ)

This is where you can find quick answers to some commonly asked questions about Condor.

7.1 Obtaining & Installing Condor

7.1.1 Where can I download Condor?

Condor can be downloaded from http://www.cs.wisc.edu/condor/downloads (Madison, Wisconsin, USA) or http://www.bo.infn.it/condor-mirror/downloads (a mirror site at the Istituto Nazionale di Fisica Nucleare in Bologna, Italy).

7.1.2 When I click to download Condor, it sends me back to the downloads page!

If you are trying to download Condor through a web proxy, try disabling it. Our web site uses the "referring page" as you navigate through our download menus in order to give you the right version of Condor, but sometimes proxies block this information from reaching our web site.

7.1.3 What platforms do you support?

See Section 1.5, on page 5.

7.1.4 Do you distribute source code?

At this time we do **not** distribute source code publicly, but instead consider requests on a case-by-case basis. If you need the source code, please email us at condor-admin@cs.wisc.edu explaining why, and we'll get back to you.

7.1.5 How do I upgrade the Unix machines in my pool from 6.2.x to 6.4.x?

This series of steps explains how to upgrade a pool of machines from running Condor version 6.2.x to version 6.4.x. Read through the entire set of directions before following them.

Briefly, the steps are to download the new version in order to replace your current binaries with the new binaries. Condor will notice that there are new binaries, since it checks for this every few minutes. The next time it checks, the new binaries will be used.

Step 1: (Optional) Place test jobs in queue This optional first step safeguards jobs currently in the queue when you upgrade. By completing this extra step, you will not lose any partially completed jobs, even if something goes wrong with your upgrade.

Manufacture test jobs that utilize each universe you use in your Condor pool. Submit each job, and put the job in the hold state, using *condor_hold*.

Step 2: Place all jobs on hold Place all jobs into the hold state while replacing binaries.

Step 3: Download Condor Version 6.4.0 To ensure that both new and current binaries are within the same volume, make a new directory within your current release directory where Version 6.4.0will go. Unix commands will be of the form

```
cd <release-dir>
mkdir new
cd new
```

Locate the correct Version 6.4.0binary, and download into this `new` directory.

Do *not* install the downloaded version. Do uncompress and then untar the downloaded version. Further untar the release directory (called `release.tar`). This will create the directories

```
bin
etc
include
sbin
lib
man
```

From this list of created directories, `bin`, `include`, `sbin`, and `lib` will be used to replace current directories.

Step 4: Configuration files The downloaded Version 6.4.0configuration file will have extra, new suggestions for configuration macro settings, to go with new features in Condor. These extra configuration macros are not be required in order to run Version 6.4.0.

Make a backup copy of the current configuration, to safeguard backing out of the upgrade, if something goes wrong.

Work through the new Version 6.4.0example configuration file to see if there is anything useful and merge with your site-specific (current) configuration file.

Step 5: Replace release directories For each of the directories that is to be replaced, move the current one aside, and put the new one in its place. The Unix commands to do this will be of the form

```
cd <release-dir>

mv bin bin.v62
mv new/bin bin

mv include include.v62
mv new/include include

mv sbin sbin.v62
mv new/sbin sbin

mv lib lib.v62
mv new/lib lib
```

Do this series of directory moves at one sitting, especially avoiding a long time lag between the moves relating to the `sbin` directory. Condor imposes a delay by design, but it does not idly wait for the new binaries to be in place.

Step 6: Observe propagation of new binaries Use *condor_status* to observe the propagation of the upgrade through the pool. As the machines notice and use the new binaries, their version number will change. Complete propagation should occur in five to ten minutes.

The command

```
condor_status -format "%s" Machine -format " %s\n" CondorVersion
```

gives a single line of information about each machine in the pool, containing only the machine name and version of Condor it is running.

Step 7: (Optional) Release test jobs Release the test jobs that were placed into the hold state in Step 1. If these test jobs complete successfully, then the upgrade is successful. If these test jobs fail (possibly by leaving the queue before finishing), then the upgrade is unsuccessful. If unsuccessful, back out of the upgrade by replacing the new configuration file with the backup copy and moving the Version 6.2.x release directories back to their previous location. Also send e-mail to condor-admin@cs.wisc.edu, explaining the situation and we'll help you work through it.

Step 8: Release all jobs Release all jobs in the queue, but running *condor_release*.

Step 9: (Optional) Install manual pages The `man` directory is new with Condor version 6.4. It contains manual pages. Note that installation of manual pages is optional; the manual pages are available in the manual, in section **??**.

To install the manual pages, move the `man` directory from `<release-dir>/new` to the desired location. Add the path name to this directory to the `MANPATH`.

7.1.6 What is "Personal Condor"?

Personal Condor is a term used to describe a specific style of Condor installation suited for individual users who do not have their own pool of machines, but want to submit Condor jobs to run elsewhere.

A Personal Condor is essentially a one-machine, self-contained Condor pool which can use "flocking" to access resources in other Condor pools. See Section 3.10.7, on page 192 for more information on flocking.

7.1.7 What do I do now? My installation of Condor does not work.

What to do to get Condor running properly depends on what sort of error occurs. One common error category are communication errors. Condor daemon log files report a failure to bind. Or, the errors in the various log files may be of the form:

```
(date and time) Error sending update to collector(s)
(date and time) Can't send end_of_message
(date and time) Error sending UDP update to the collector

(date and time) failed to update central manager

(date and time) Can't send EOM to the collector
```

This problem can also be observed by running *condor_status*. It will give a message of the form:

```
Error:  Could not fetch ads --- error communication error
```

To solve this problem, understand that Condor uses the first network interface it sees on the machine. Since machines often have more than one interface, this problem usually implies that the wrong network interface is being used.

It is incorrect to use the localhost network interface. This has IP address 127.0.0.1 on all machines. To check if this incorrect IP address is being used, look at the contents of the CollectorLog file on the pool's your central manager right after it is started. The contents will be of the form:

```
5/25 15:39:33 ******************************************************
5/25 15:39:33 ** condor_collector (CONDOR_COLLECTOR) STARTING UP
5/25 15:39:33 ** $CondorVersion: 6.2.0 Mar 16 2001 $
5/25 15:39:33 ** $CondorPlatform: INTEL-LINUX-GLIBC21 $
5/25 15:39:33 ** PID = 18658
5/25 15:39:33 ******************************************************
5/25 15:39:33 DaemonCore: Command Socket at <128.105.101.15:9618>
```

The last line tells the IP address and port the collector has bound to and is listening on. If the IP address is 127.0.0.1, then Condor is definitely using the wrong network interface.

There are two solutions to this problem. One solution changes the order of the network interfaces. The preferred solution sets which network interface Condor should use by adding the following parameter to the local Condor configuration file:

```
NETWORK_INTERFACE = machine-ip-address
```

Where `machine-ip-address` is the IP address of the interface you wish Condor to use.

7.2 Setting up Condor

7.2.1 How do I set up a central manager on a machine with multiple network interfaces?

Please see section 3.10.9 on page 198.

7.2.2 How do I get more than one job to run on my SMP machine?

Condor will automatically recognize a SMP machine and advertise each CPU of the machine separately. For more details, see section 3.10.8 on page 192.

7.2.3 How do I set up my machines so that only certain users's jobs will run on them?

Restrictions on what jobs will run on a given resource can be easily specified in the resource's Requirements statement.

To specify that a given machine should only run certain users's jobs, for example, you could add the following Requirements entry to the machine's Condor configuration file:

```
Requirements = (RemoteUser == "userfoo@baz.edu" || Remo-
teUser == "userbar@baz.edu" )
```

To configure multiple machines to do so, simply create a common configuration file containing this requirement for them to share.

7.2.4 How do I configure Condor to run my jobs only on machines that have the right packages installed?

This is a two-step process. First, you need to tell the machines to report that they have special software installed, and second, you need to tell the jobs to require machines that have that software.

To tell the machines to report the presence of special software, first add a parameter to their configuration files like so:

```
HAS_MY_SOFTWARE = True
```

And then, if there are already STARTD_EXPRS defined in that file, add HAS_MY_SOFTWARE to them, or, if not, add the line:

```
STARTD_EXPRS = HAS_MY_SOFTWARE, $(STARTD_EXPRS)
```

NOTE: For these changes to take effect, each *condor_startd* you update needs to be reconfigured with *condor_reconfig* -startd.

Next, to tell your jobs to only run on machines that have this software, add a requirements statement to their submit files like so:

```
Requirements = (HAS_MY_SOFTWARE =?= True)
```

NOTE: Be sure to use =?= instead of == so that if a machine doesn't have the HAS_MY_SOFTWARE parameter defined, the job's Requirements expression will not evaluate to "undefined", preventing it from running anywhere!

7.2.5 How do I configure Condor to only run jobs at night?

A commonly requested policy for running batch jobs is to only allow them to run at night, or at other pre-specified times of the day. Condor allows you to configure this policy with the use of the `ClockMin` and `ClockDay` *condor_startd* attributes. A complete example of how to use these attributes for this kind of policy is discussed in subsection **??** on page **??**, "A Policy for Only Running Jobs at Certain Times of the Day", inside section 3.6 on "Configuring The Startd Policy"

7.2.6 Why will the *condor_master* not run when a local configuration file is missing?

If a LOCAL_CONFIG_FILE is specified in the global configuration file, but the specified file does not exist, the *condor_master* will not start up, and it prints a variation of the following example message.

```
ERROR: Can't read config file /mnt/condor/hosts/bagel/condor_config.local
```

This is not a bug; it is a feature! Condor has always worked this way on purpose. There is a potentially large security hole if Condor is configured to read from a file that does not exist. By creating that file, a malicious user could change all sorts of Condor settings. This is an easy way to gain root access to a machine, where the daemons are running as root.

The intent is that if you've set up your global configuration file to read from a local configuration file, and the local file is not there, then something is wrong. It is better for the *condor_master* to exit right away and log an error message than to start up.

If the *condor_master* continued with the local configuration file missing, either A) someone could breach security or B) you will have potentially important configuration information missing. Consider the example where the local configiguration file was on an NFS partition and the server was down. There would be all sorts of really important stuff in the local configuration file, and Condor might do bad things if it started without those settings.

If supplied it with an empty file, the *condor_master* works fine.

7.3 Running Condor Jobs

7.3.1 I'm at the University of Wisconsin-Madison Computer Science Dept., and I am having problems!

Please see the web page http://www.cs.wisc.edu/condor/uwcs. As it explains, your home directory is in AFS, which by default has access control restrictions which can prevent Condor jobs from running properly. The above URL will explain how to solve the problem.

7.3.2 I'm getting a lot of email from Condor. Can I just delete it all?

Generally you shouldn't ignore **all** of the mail Condor sends, but you can reduce the amount you get by telling Condor that you don't want to be notified every time a job successfully completes, only when a job experiences an error. To do this, include a line in your submit file like the following:

```
Notification = Error
```

See the Notification parameter in the *condor_q* man page on page 400 of this manual for more information.

7.3.3 Why will my vanilla jobs only run on the machine where I submitted them from?

Check the following:

1. Did you submit the job from a local filesystem that other computers can't access?

   See Section 3.3.5, on page 93.

2. Did you set a special requirements expression for vanilla jobs that's preventing them from running but not other jobs?

   See Section 3.3.5, on page 93.

3. Is Condor running as a non-root user?

   See Section 3.7.2, on page 152.

7.3.4 My job starts but exits right away with signal 9.

This can occur when the machine your job is running on is missing a shared library required by your program. One solution is to install the shared library on all machines the job may execute on. Another, easier, solution is to try to re-link your program statically so it contains all the routines it needs.

7.3.5 Why aren't any or all of my jobs running?

Problems like the following are often reported to us:

```
> I have submitted 100 jobs to my pool, and only 18 appear to be
> running, but there are plenty of machines available.
What should I
> do to investigate the reason why this happens?
```

Start by following these steps to understand the problem:

1. Run *condor_q* -analyze and see what it says.

2. Look at the User Log file (whatever you specified as "log = XXX" in the submit file).

   See if the jobs are starting to run but then exiting right away, or if they never even start.

3. Look at the SchedLog on the submit machine after it negotiates for this user. If a user doesn't have enough priority to get more machines the SchedLog will contain a message like "lost priority, no more jobs".
4. If jobs are successfully being matched with machines, they still might be dying when they try to execute due to file permission problems or the like. Check the ShadowLog on the submit machine for warnings or errors.
5. Look at the NegotiatorLog during the negotiation for the user. Look for messages about priority, "no more machines", or similar.

7.3.6 Can I submit my standard universe SPARC Solaris 2.6 jobs and have them run on a SPARC Solaris 2.7 machine?

No. You may only use binary compatibility between SPARC Solaris 2.5.1 and SPARC Solaris 2.6 and between SPARC Solaris 2.7 and SPARC Solaris 2.8, but not between SPARC Solaris 2.6 and SPARC Solaris 2.7. We may implement support for this feature in a future release of Condor.

7.3.7 Why do my vanilla jobs keep cycling between suspended and unsuspended?

This is a load sampling error that Condor performs when starting a many process vanilla job with heavy initial load. Condor mistakenly decides that the load on the machine has gotten too high while the job is in the initialization phase and kicks the job off the machine.

What is needed is a way for Condor to check to see if the load of the machine has been high over a certain period of time. There is a startd attribute, `CpuBusyTime` that can be used for this purpose. This macro returns the time `$(CpuBusy)` (usually defined in the default config file) has been true. `$(CpuBusy)` is defined in terms of non-Condor load.

To take advantage of this macro, you can use it in your `SUSPEND` macro. Here is an example:

```
SUSPEND = (CpuBusyTime > 3 * $(MINUTE)) && ((CurrentTime - Job-
Start) > 90)
```

The above policy says to only suspend the job if the cpu has been busy with non-Condor load at least three minutes and it has been at least 90 seconds since the start of the job.

7.3.8 Why might my job be preempted (evicted)?

There are four circumstances under which Condor may evict a job. They are controlled by different expressions.

Reason number 1 is the user priority: controlled by the PREEMPTION_REQUIREMENTS expression in the configuration file. If there is a job from a higher priority user sitting idle, the *condor_negotiator* daemon may evict a currently running job submitted from a lower priority user if PREEMPTION_REQUIREMENTS is True. For more on user priorities, see section 2.7 and section 3.5.

Reason number 2 is the owner (machine) policy: controlled by the PREEMPT expression in the configuration file. When a job is running and the PREEMPT expression evaluates to True, the *condor_startd* will evict the job. The PREEMPT expression should reflect the requirements under which the machine owner will not permit a job to continue to run. For example, a policy to evict a currently running job when a key is hit or when it is the 9:00am work arrival time, would be expressed in the PREEMPT expression and enforced by the *condor_startd*. For more on the PREEMPT expression, see section 3.6.

Reason number 3 is the owner (machine) preference: controlled by the RANK expression in the configuration file (sometimes called the startd rank or machine rank). The RANK expression is evaluated as a floating point number. When one job is running, a second idle job that evaluates to a higher RANK value tells the *condor_startd* to prefer the second job over the first. Therefore, the *condor_startd* will evict the first job so that it can start running the second (preferred) job. For more on RANK, see section 3.6.

Reason number 4 is if Condor is to be shutdown: on a machine that is currently running a job. Condor evicts the currently running job before proceding with the shutdown.

7.4 Condor on Windows NT / Windows 2000

7.4.1 Will Condor work on a network of mixed Unix and NT machines?

You can have a Condor pool that consists of both Unix and NT machines.

Your central manager can be either Windows NT or Unix. For example, even if you had a pool consisting strictly of Unix machines, you could use an NT box for your central manager, and vice versa.

You can submit jobs destined to run on Windows NT from either an NT machine **or** a Unix machine. However, at this point in time you cannot submit jobs destined to run on Unix from NT. We do plan on adding this functionality, however.

So, in summary:

1. A single Condor pool can consist of both Windows NT and Unix machines.
2. It does not matter at all if your Central Manager is Unix or NT.
3. Unix machines can submit jobs to run on other Unix or Windows NT machines.
4. Windows NT machines can only submit jobs which will run on Windows NT machines.

7.4.2 When I run *condor_status* I get a communication error, or the Condor daemon log files report a failure to bind.

Condor uses the first network interface it sees on your machine. This problem usually means you have an extra, inactive network interface (such as a RAS dialup interface) defined before to your regular network interface.

To solve this problem, either change the order of your network interfaces in the Control Panel, or explicity set which network interface Condor should use by adding the following parameter to your Condor config file:

```
NETWORK_INTERFACE = ip-address
```

Where "ip-address" is the IP address of the interface you wish Condor to use.

7.4.3 My job starts but exits right away with status 128.

This can occur when the machine your job is running on is missing a DLL (Dynamically Linked Library) required by your program. The solution is to find the DLL file the program needs and put it in the TRANSFER_INPUT_FILES list in the job's submit file.

To find out what DLLs your program depends on, right-click the program in Explorer, choose Quickview, and look under "Import List".

7.4.4 Why does the startd crash on CondorNT with the error "caInsert: Can't insert CpuBusy into target classad."?

This is a common problem with European installations on Windows. The problem is Condor expects all decimal points to be the period character (.), but the Windows locale defines them as the comma character(,). This will be fixed in the next version of Condor for NT, however we have users who have fixed the problem by changing the following registry value to a period instead of a comma: `HKEY_USERS\.DEFAULT\Control Panel\International\sDecimal`

7.4.5 How can I access network files with Condor on NT?

Features to allow Condor NT to work well with a network file server are coming very soon. However, there are a couple of work-arounds which you can do immediately with the current version of Condor NT in order to access a file server.

The heart of the problem is that on the execute machine, Condor creates a "temporary" user which will run the job... and your file server has never heard of this user before. So the workaround is to either

- A: access the file server as a different user via a net use command with a login and password
- B: access the file server as guest
- C: access the file server with a ”NULL” descriptor
- D: use the contrib module from the folks at Bristol University

All of these workarounds have disadvantages, but they may be able to hold you until our code to support shared file servers in Condor is officially released.

Here are the three methods in more detail:

METHOD A - access the file server as a different user via a net use command with a login and password

Example: you want to copy a file off of a server before running it....

```
@echo off
net use \\myserver\someshare MYPASSWORD /USER:MYLOGIN
copy \\myserver\someshare\my-program.exe
my-program.exe
```

The idea here is to simply authenticate to the file server with a different login than the temporary Condor login. This is easy with the ”net use” command as shown above. Of course, the obvious disadvantage is this user’s password is stored and transferred as cleartext.

METHOD B - access the file server as guest

Example: you want to copy a file off of a server before running it as GUEST

```
@echo off
net use \\myserver\someshare
copy \\myserver\someshare\my-program.exe
my-program.exe
```

In this example, you’d contact the server MYSERVER as the Condor temporary user. However, if you have the GUEST account enabled on MYSERVER, you will be authenticated to the server as user ”GUEST”. If your file permissions (ACLs) are setup so that either user GUEST (or group EVERYONE) has access the share ”someshare” and the directories/files that live there, you can use this method. The downside of this method is you need to enable the GUEST account on your file server. WARNING: This should be done *with extreme caution* and only if your file server is well protected behind a firewall that blocks SMB traffic.

METHOD C - access the file server with a ”NULL” descriptor

One more option is to use NULL Security Descriptors. In this way, you can specify which shares are accessible by NULL Descriptor by adding them to your registry. You can then use the batch file wrapper like:

```
net use z: \\myserver\someshare /USER:""
z:\my-program.exe
```

so long as 'someshare' is in the list of allowed NULL session shares. To edit this list, run regedit.exe and navigate to the key:

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
     CurrentControlSet\
       Services\
         LanmanServer\
           Parameters\
             NullSessionShares
```

and edit it. unfortunately it is a binary value, so you'll then need to type in the hex ascii codes to spell out your share. each share is separated by a null (0x00) and the last in the list is terminated with two nulls.

although a little more difficult to set up, this method of sharing is a relatively safe way to have one quasi-public share without opening the whole guest account. you can control specifically which shares can be accessed or not via the registry value mentioned above.

METHOD D - access with the contrib module from Bristol

Another option: some hardcore Condor users at Bristol University developed their own module for starting jobs under Condor NT to access file servers. It involves storing submitting user's passwords on a centralized server. Below I have included the README from this contrib module, which will soon appear on our website within a week or two. If you want it before that, let me know, and I could email it to you.

Here is the README from the Bristol Condor NT contrib module:

```
README
Compilation Instructions
Build the projects in the following order

CondorCredSvc
CondorAuthSvc
Crun
Carun
AfsEncrypt
RegisterService
DeleteService
Only the first 3 need to be built in order. This just makes sure that the
RPC stubs are correctly rebuilt if required. The last 2 are only helper
```

```
applications to install/remove the services. All projects are Vi-
sual Studio
6 projects. The nmakefiles have been exported for each. Only the project
for Carun should need to be modified to change the loca-
tion of the AFS
libraries if needed.

Details
CondorCredSvc
CondorCredSvc is a simple RPC service that serves the do-
main account
credentials. It reads the account name and password from the reg-
istry of
the machine it's running on. At the moment these de-
tails are stored in
clear text under the key

HKEY_LOCAL_MACHINE\Software\Condor\CredService

The account name and password are held in REG_SZ values "Ac-
count" and
"Password" respectively. In addition there is an op-
tional REG_SZ value
"Port" which holds the clear text port number (e.g. "1234"). If this value
is not present the service defaults to using port 3654.

At the moment there is no attempt to encrypt the user-
name/password when it
is sent over the wire - but this should be reasonably straightfor-
ward to
change. This service can sit on any machine so keeping the reg-
istry entries
secure ought to be fine. Certainly the ACL on the key could be set to only
allow administrators and SYSTEM access.

CondorAuthSvc and Crun
These two programs do the hard work of getting the job authenti-
cated and
running in the right place. CondorAuthSvc actually han-
dles the process
creation while Crun deals with getting the winstation/desktop/working
directory and grabbing the console output from the job so that Con-
dor's
output handling mechanisms still work as advertised. Proba-
bly the easiest
way to see how the two interact is to run through the job cre-
```

```
ation process:

The first thing to realize is that condor it-
self only runs Crun.exe. Crun
treats its command line parameters as the program to re-
ally run. e.g. "Crun
\\mymachine\myshare\myjob.exe" actually causes
\\mymachine\myshare\myjob.exe to be executed in the con-
text of the domain
account served by CondorCredSvc. This is how it works:

When Crun starts up it gets its window station and desktop -
these are the
ones created by condor. It also gets its current directory -
again already
created by condor. It then makes sure that SYSTEM has permis-
sion to modify
the DACL on the window station, desktop and direc-
tory. Next it creates a
shared memory section and copies its environment vari-
able block into it.
Then, so that it can get hold of STDOUT and STDERR from the job it makes
two named pipes on the machine it's running on and at-
taches a thread to
each which just prints out anything that comes in on the pipe to the
appropriate stream. These pipes currently have a NULL DACL, but only one
instance of each is allowed so there shouldn't be any is-
sues involving
malicious people putting garbage into them. The shared memory sec-
tion and
both named pipes are tagged with the ID of Crun's pro-
cess in case we're on
a multi-processor machine that might be running more than one job. Crun
then makes an RPC call to CondorAuthSvc to actu-
ally start the job, passing
the names of the window station, desktop, exe-
cutable to run, current
directory, pipes and shared memory section (it only at-
tempts to call
CondorAuthSvc on the same machine as it is run-
ning on). If the jobs starts
successfully it gets the process ID back from the RPC call and then just
waits for the new process to finish before clos-
ing the pipes and exiting.
Technically, it does this by synchronizing on a handle to the pro-
cess and
```

```
waiting for it to exit. CondorAuthSvc sets the ACL on the pro-
cess to allow
EVERYONE  to synchronize on it.

[ Technical note: Crun adds "C:\WINNT\SYSTEM32\CMD.EXE /C" to the start of
the command line. This is because the process is cre-
ated with the network
context of the caller i.e. LOCALSYSTEM. Prepend-
ing cmd.exe gets round any
unexpected "Access Denied" errors. ]

If Crun gets a WM_CLOSE (CTRL_CLOSE_EVENT) while the job is run-
ning it
attempts to stop the job, again with an RPC call to Condo-
rAuthSvc passing
the job's process ID.

CondorAuthSvc runs as a service under the LOCALSYSTEM ac-
count and does the
work of starting the job. By default it listens on port 3655, but this can
be changed by setting the optional REG_SZ value "Port" un-
der the registry key

HKEY_LOCAL_MACHINE\Software\Condor\AuthService

(Crun also checks this registry key when attempting to contact
CondorAuthSvc.) When it gets the RPC to start a job Condo-
rAuthSvc first
connects to the pipes for STDOUT and STDERR to prevent any-
one else sending
data to them. It also opens the shared memory sec-
tion with the environment
stored by Crun.  It then makes an RPC call to Condor-
CredSvc (to get the
name and password of the domain account) which is most likely run-
ning on
another system. The location information is stored in the reg-
istry under
the key

HKEY_LOCAL_MACHINE\Software\Condor\CredService

The name of the machine running CondorCredSvc must be held in the REG_SZ
value "Host". This should be the fully qualified do-
main name of the
machine. You can also specify the optional "Port" REG_SZ value in case you
```

```
are running CondorCredSvc on a different port.

Once the domain account credentials have been received the ac-
count is
logged on through a call to LogonUser. The DACLs on the win-
dow station,
desktop and current directory are then modified to allow the do-
main account
access to them and the job is started in that window sta-
tion and desktop
with a call to CreateProcessAsUser. The starting direc-
tory is set to the
same as sent by Crun, STDOUT and STDERR handles are set to the named pipes
and the environment sent by Crun is used. Condo-
rAuthSvc also starts a
thread which waits on the new process handle until it termi-
nates to close
the named pipes. If the process starts correctly the pro-
cess ID is returned
to Crun.

If Crun requests that the job be stopped (again via RPC), CondorAuthSvc
loops over all windows on the window station and desktop speci-
fied until it
finds the one associated with the required pro-
cess ID. It then sends that
window a WM_CLOSE message, so any termination han-
dling built in to the job
should work correctly.

[Security Note: CondorAuthSvc currently makes no attempt to ver-
ify the
origin of the call starting the job. This is, in princi-
pal, a bad thing
since if the format of the RPC call is known it could let any-
one start a
job on the machine in the context of the domain user. If sensi-
ble security
practices have been followed and the ACLs on sensitive sys-
tem directories
(such as C:\WINNT) do not allow write access to any-
one other than trusted
users the problem should not be too serious.]

Carun and AFSEncrypt
Carun and AFSEncrypt are a couple of utilities to al-
```

```
low jobs to access AFS
without any special recompliation. AFSEncrypt encrypts an AFS
username/password into a file (called .afs.xxx) using a simple XOR
algorithm. It's not a particularly secure way to do it, but it's sim-
ple and
self-inverse. Carun reads this file and gets an AFS token be-
fore running
whatever job is on its command line as a child pro-
cess. It waits on the
process handle and a 24 hour timer. If the timer ex-
pires first it briefly
suspends the primary thread of the child process and at-
tempts to get a new
AFS token before restarting the job, the idea be-
ing that the job should
have uninterrupted access to AFS if it runs for more than 25 hours (the
default token lifetime). As a security measure, the AFS creden-
tials are
cached by Carun in memory and the .afs.xxx file deleted as soon as the
username/password have been read for the first time.

Carun needs the machine to be running either the IBM AFS client or the
OpenAFS client to work. It also needs the client li-
braries if you want to
rebuild it.

For example, if you wanted to get a list of your AFS tokens un-
der Condor
you would run the following:

Crun \\mymachine\myshare\Carun tokens.exe

Running a job
To run a job using this mechanism specify the follow-
ing in your job
submission (assuming Crun is in C:\CondorAuth):

Executable= c:\CondorAuth\Crun.exe
Arguments = \\mymachine\myshare\carun.exe
\\anothermachine\anothershare\myjob.exe
Transfer_Input_Files = .afs.xxx

along with your usual settings.

Installation
A basic installation script for use with the Inno Setup installation
```

```
package compiler can be found in the Install folder.
```

7.4.6 Does Condor run under Windows 2000?

Condor does run under Windows 2000 Professional and Server.

There will be problems if you have more than 2 Gigabytes of RAM or swap space.

A Personal Condor installation will *not* work.

7.5 Troubleshooting

7.5.1 What happens if the central manager crashes?

If the central manager crashes, jobs that are already running will continue to run unaffected. Queued jobs will remain in the queue unharmed, but can not begin running until the central manager is restarted and begins matchmaking again. Nothing special needs to be done after the central manager is brought back online.

7.5.2 When I ssh/telnet to a machine to check particulars of how Condor is doing something, it is always vacating or unclaimed when I know a job had been running there!

Depending on how your policy is set up, Condor will track *any* tty on the machine for the purpose of determining if a job is to be vacated or suspended on the machine. It could be the case that after you ssh there, Condor notices activity on the tty allocated to your connection and then vacates the job.

7.6 Other questions

7.6.1 Is Condor Y2K-compliant?

Yes. Internally, Condor uses the standard UNIX time representation (the number of seconds since 1/1/1970) and is not affected by the Y2K bug. In addition, the Condor tools now correctly display the four-digit year in their output.

The output of Condor tools from some older versions (pre-6.2) may display years incorrectly, but their internal representation is still correct and their display bugs do not affect the operation of Condor.

7.6.2 Is there a Condor mailing-list?

Yes. We run an extremely low traffic mailing list solely to announce new versions of Condor. To subscribe, email majordomo@cs.wisc.edu with a message body of:

```
subscribe condor-world
```

7.6.3 Do you support Globus?

Yes, we support a variety of interactions with Globus software, including running Condor jobs on Globus-managed resources. At this time, however, we have not released this software publicly. If you are interested in using Condor with Globus, please send email to condor-admin@cs.wisc.edu and we can provide you with more information.

7.6.4 What is the process for upgrading from 6.0 to 6.2?

The usual process for upgrading is to copy the original 6.0 binaries to <name>.old versions and then copy the 6.2 binaries over the 6.0 binaries. The *condor_master* will automatically(in about 5 minutes) notice that new binaries have been installed and exec them. Please be sure to upgrade the config files to match the 6.2 version and pay attention to copying over your site's specific policy to the new config file. Some macros and their meanings have changed and need to be addressed in the upgrade, you may find them in section 3.6.10 of the manual.

One special note: You must relink your 6.0 standard universe jobs with the 6.2 supplied libraries. Backwards compatibility to 6.0 from 6.2 had been intentionally broken because of an evolution of features that the standard universe now supports.

7.6.5 My question isn't in the FAQ!

If you have any questions that are not listed in this FAQ, try looking through the rest of the manual. If you still can't find an answer, feel free to contact us at condor-admin@cs.wisc.edu.

Note that Condor's free email support is provided on a best-effort basis, and at times we may not be able to provide a timely response. If guaranteed support is important to you, please inquire about our paid support services.

CHAPTER

EIGHT

Condor Version History

8.1 Introduction to Condor Versions

This chapter provides descriptions of what features have been added or bugs fixed for each version of Condor. The first section describes the Condor version numbering scheme, what the numbers mean, and what the different *release series* are. The rest of the sections each describe a specific release series, and all the Condor versions found in that series.

8.1.1 Condor Version Number Scheme

Starting with version 6.0.1, Condor adopted a new, hopefully easy to understand version numbering scheme. It reflects the fact that Condor is both a production system and a research project. The numbering scheme was primarily taken from the Linux kernel's version numbering, so if you are familiar with that, it should seem quite natural.

There will usually be two Condor versions available at any given time, the *stable* version, and the *development* version. Gone are the days of "patch level 3", "beta2", or any other random words in the version string. All versions of Condor now have exactly three numbers, seperated by "."

- The first number represents the major version number, and will change very infrequently.
- *The thing that determines whether a version of Condor is "stable" or "development" is the second digit. Even numbers represent stable versions, while odd numbers represent development versions.*
- The final digit represents the minor version number, which defines a particular version in a given release series.

8.1.2 The Stable Release Series

People expecting the stable, production Condor system should download the stable version, denoted with an even number in the second digit of the version string. Most people are encouraged to use this version. We will only offer our paid support for versions of Condor from the stable release series.

On the stable series, new minor version releases will only be made for bug fixes and to support new platforms. No new features will be added to the stable series. People are encouraged to install new stable versions of Condor when they appear, since they probably fix bugs you care about. Hopefully, there won't be many minor version releases for any given stable series.

8.1.3 The Development Release Series

Only people who are interested in the latest research, new features that haven't been fully tested, etc, should download the development version, denoted with an odd number in the second digit of the version string. We will make a best effort to ensure that the development series will work, but we make no guarantees.

On the development series, new minor version releases will probably happen frequently. People should not feel compelled to install new minor versions unless they know they want features or bug fixes from the newer development version.

Most sites will probably never want to install a development version of Condor for any reason. Only if you know what you are doing (and like pain), or were explicitly instructed to do so by someone on the Condor Team, should you install a development version at your site.

NOTE: Different releases within a development series cannot be installed side-by-side within the same pool. For example, the protocols used by version 6.1.6 are not compatible with the protocols used in version 6.1.5. When you upgrade to a new development release, make certain you upgrade all machines in your pool to the same version.

After the feature set of the development series is satisfactory to the Condor Team, we will put a code freeze in place, and from that point forward, only bug fixes will be made to that development series. When we have fully tested this version, we will release a new stable series, resetting the minor version number, and start work on a new development release from there.

8.2 Stable Release Series 6.4

This is the stable release series of Condor. New features will be added and tested in the 6.5 development series. The details of each version are described below.

8.2.1 Version 6.4.3

New Features:

- None.

Bugs Fixed:

- When the NIS environment was being used, the *condor_starter* daemon would produce heavy amounts of NIS traffic. This has been fixed.
- Binary characters in the `StarterLog` file and a possible segmentation fault have been fixed.
- Fixed *select*(2) in the standard universe on our Linux ports.
- Fixed a small bug in *condor_q* that was displaying the wrong username for "niceuser" jobs.
- Fixed a bug where, in the standard universe, you could not open a file with spaces in it.

Known Bugs:

- You may not open a file in the standard uiniverse with a colon ":" in its name.

8.2.2 Version 6.4.2

New Features:

- None.

Bugs Fixed:

- None.

Known Bugs:

- None.

8.2.3 Version 6.4.1

New Features:

- None.

Bugs Fixed:

- Users are now allowed to answer "none" when prompted by the installer to provide a Java JVM path. This avoids an endless loop and leaves the Java abilities of Condor unconfigured.

Known Bugs:

- None.

8.2.4 Version 6.4.0

New Features:

-
-

Bugs Fixed:

-
-

Known Bugs:

-
-

8.3 Development Release Series 6.3

This is the second development release series of Condor.

It contains numerous enhancements over the 6.2 stable series. For example:

- Support for Kerberos and X.509 authentication.
- Support for transfering files needed by jobs (for all universes except standard and PVM)
- Support for MPICH jobs.
- Support for JAVA jobs.

- Condor DAGMan is dramatically more reliable and efficient, and offers a number of new features.

The 6.3 series has many other improvements over the 6.2 series, and may be available on newer platforms. The new features, bugs fixed, and known bugs of each version are described below in detail.

8.3.1 Version 6.3.4

New Features:

- If a job universe is not specified in a submit description file, *condor_submit* will check the config file for `DEFAULT_UNIVERSE` instead of always choosing the standard universe.
- The `D_SECONDS` debug flag is deprecated. Seconds are now always included in logfiles.
- For each daemon listed in `DAEMON_LIST` , you can now control the environment variables of the daemon with a config file setting of the form `DAEMONNAME_ENVIRONMENT` , where `DAEMONNAME` is the name of a daemon listed in `DAEMON_LIST` . For more information, see section 3.3.7.

Bugs Fixed:

- Fixed a bug in the new starter where if the submit file set no arguments, the job would receive one argument of zero length.

Known Bugs:

- None.

8.3.2 Version 6.3.3

New Features:

- Added support for Kerberos and X.509 authentication in Condor.
- Added the ability for vanilla jobs on Unix to use Condor's file transfer mechanism so that you don't have to rely on a shared file system.
- Added support for MPICH jobs on Windows NT and 2000.
- Added support for the JAVA universe.

- When you use *condor_hold* and *condor_release*, you now see an entry about the event in the UserLog file for the job.
- Whenever a job is removed, put on hold, or released (either by a Condor user or by the Condor system itself), there is a "reason" attribute placed in the job ad and written to the UserLog file. If a job is held, `HoldReason` will be set. If a job is released, `ReleaseReason` will be set. If a job is removed, `RemoveReason` will be set. In addition, whenever a job's status changes, `EnteredCurrentStatus` will contain the epoch time when the change took place.
- The error messages you get from *condor_rm*, *condor_hold* and *condor_release* have all been updated to be more specific and accurate.
- Condor users can now specify a policy for when their jobs should leave the queue or be put on hold. They can specify expressions that are evaluated periodically, and whenever the job exits. This policy can be used to ensure that the job remains in the queue and is re-run until it exits with a certain exit code, that the job should be put on hold if a certain condition is true, and so on. If any of these policy expressions result in the job being removed from the queue or put on hold, the UserLog entry for the event includes a string describing why the action was taken.
- Changed the way Condor finds the various *condor_shadow* and *condor_starter* binaries you have installed on your machine. Now, you can specify a `SHADOW_LIST` and a `STARTER_LIST` . These are treated much like the `DAEMON_LIST` setting, they specify a list of attribute names, each of which point to the actual binary you want to use. On startup, Condor will check these lists, make sure all the binaries specified exist, and find out what abilities each program provides. This information is used during matchmaking to ensure that a job which requires a certain ability (like having a new enough version of Condor to support transfering files on Unix) can find a resource that provides that ability.
- Added new security feature to offer fine-grained control over what configuration values can be modified by *condor_config_val* using *-set* and related options. Pool administrators can now define lists of attributes that can be set by hosts that authenticate to the various permission levels of Condor's host based security (for example, `WRITE`, `ADMINISTRATOR`, etc). These lists are defined by attributes with names like `SETTABLE_ATTRS_CONFIG` and `STARTD_SETTABLE_ATTRS_OWNER` . For more information about host-based security in Condor, see section 3.7.5 on page 166. For more information about how to configure the new settings, see the same section of the manual. In particular, see section **??** on page **??**.
- Greatly improved the handling of the "soft kill signal" you can specify for your job. This signal is now stored as a signal name, not an integer, so that it works across different platforms. Also, fixed some bugs where the signal numbers were getting translated incorrectly in some circumstances.
- Added the *-full* option to *condor_reconfig*. The *-full* option causes the Condor daemon to clear its cache of DNS information and some other expensive operations. So, the regular *condor_reconfig* is now more light-weight, and can be used more frequently without undue overhead on the Condor daemons. The default *condor_reconfig* has also been changed so that it will work from any host with `WRITE` permission in your pool, instead of requiring `ADMINISTRATOR` access.

- Added the EMAIL_DOMAIN config file setting. This allows Condor administrators to define a default domain where Condor should send email if whatever UID_DOMAIN is set to would yield invalid email addresses. For more information, see section **??** on page **??**.
- Added support for RedHat 7.2.
- When printing out the UserLog, we now only log a new event for "Image size of job updated" when the new value is different than the existing value.

Bugs Fixed:

- Fixed a bug in Condor-PVM where it was possible that a machine would be placed into the virtual machine, but then ignored by Condor for the purposes of scheduling tasks there.
- Under Solaris, the checkpointing libraries could segfault while determining the page size of the machine. This has been fixed.
- In a heavily loaded submit machine, the *condor_schedd* would time out authentication checks with its shadows. This would cause the shadows to exit believing the *condor_schedd* had died placing jobs into the idle state and the *condor_schedd* to exhibit poor performance. This timeout problem has been corrected.
- Removed use of the bfd libary in the Condor Linux distribution. This will make the dynamic versions of the Condor executables have a higher chance of being usable when RedHat upgrades.
- When you specify "STARTD_HAS_BAD_UTMP = True" in the config files on a linux machine with a 2.4+ kernel, the *condor_startd* would report an error stating some of the tty entries in /dev. This would result in incorrect tty activity sampling causing jobs to not be migrated or incorrectly started on a resource. This has now been corrected.
- When you specify "GenEnv = True" in a *condor_submit* file, your environment is no longer restricted to 10KB.
- The three-digit event numbers which begin each job event in the userlog were incorrect for some events in Condor 6.3.0 and 6.3.1. Specifically, ULOG_JOB_SUSPENDED, ULOG_JOB_UNSUSPENDED, ULOG_JOB_HELD, ULOG_JOB_RELEASED, ULOG_GENERIC, and ULOG_JOB_ABORTED had incorrect event numbers. This has now been corrected.

  NOTE: This means userlog-parsing code written for Condor 6.3.0 or 6.3.1 development releases may not work reliably with userlogs generated by other versions of Condor, and visa-versa. Userlog events will remain compatible between all stable releases of Condor, however, and with post-6.3.1 releases in this development series.
- The *condor_run* script now correctly exits when it sees a job aborted event, instead of hanging, waiting for a termination event.

- Until now, when a DAG node's Condor job failed, the node failed, regardless of whether its POST script succeeded or failed. This was a bug, because it prevented users from using POST scripts to evaluate jobs with non-zero exit codes and deem them successful anyway. This has now been fixed – a node's success is equal to its POST script's success – but the change may affect existing DAGs which rely on the old, broken behavior. Users utilizing POST scripts must now be sure to pass the POST script the job's return value, and return it again, if they do not wish to alter it; otherwise failed jobs will be masked by ignorant POST scripts which always succeed.

Known Bugs:

- The HP-UX Vendor C++ CFront compiler does not work with *condor_compile* if exception handling is enabled with +eh.
- The HP-UX Vendor aCC compiler does not work at all with Condor.

8.3.3 Version 6.3.2

Version 6.3.2 of Condor was only released as a version of "Condor-G". This version of Condor-G is not widely deployed. However, to avoid confusion, the Condor developers did not want to release a full Condor distribution with the same version number.

Version 6.3.1

New Features:

- Added support for an `x509proxy` option in *condor_submit*. There is now a seperate *condor_GridManager* for each user and proxy pair. This will be detailed in a future release of Condor.
- More Condor DAGMan improvements and bug fixes:
  - Added a [*-dag*] flag to *condor_q* to more succinctly display dags and their ownership.
  - Added a new event to the Condor userlog at the completion of a POST script. This allows DAGMan, during recovery, to know which POST scripts have finished succesfully, so it no longer has to re-run them all to make sure.
  - Implemented separate *-MaxPre* and *-MaxPost* options to limit the number of simultaneously running PRE and POST scripts. The *-MaxScripts* option is still available, and is equivalent to setting both *-MaxPre* and *-MaxPost* to the same value.
  - Added support for a new "Retry" parameter in the DAG file, which instructs DAGMan to automatically retry a node a configurable number of times if its PRE Script, Job, or POST Script fail for any reason.

  - Added timestamps to all DAGMan log messages.
  - Fixed a bug whereby DAGMan would clean up its lock file without creating a rescue file when killed with SIGTERM.
  - DAGMan no longer aborts the DAG if it encounters executable error or job aborted events in the userlog, but rather marks the corresponding DAG nodes as "failed" so the rest of the DAG can continue.
  - Fixed a bug whereby DAGMan could crash if it saw userlog events for jobs it didn't submit.

- Added port restriction capabilities to Condor so you can specify a range of ports to use for the communication between Condor Daemons.
- To improve performance: if there's no `HISTORY` file specified, don't connect back to the schedd to report your exit info on successful compeletion, since the schedd is simply going to discard that info anyway.
- Added the macro `SECONDARY_COLLECTOR_LIST` to tell the master to send classads to an additional list of collectors so you can do administration commands when the primary collector is down.
- When a job checkpoints it askes the shadow whether or not it should and if so where. This fixes some flocking bugs and increases performance of the pool.
- Added match rejection diagnostics in *condor_q* [*-analyze*] to give more information on why a particular job hasn't started up yet.
- Added [*–vms*] argument to *condor_glidein* that enables the control of how many virtual machines to start up on the target platform.
- Added capability to the config file language to retrieve environment variables while being processed.
- Added capability to make default user user priority factor configurable with the `DEFAULT_PRIORITY_FACTOR` macro in the config files.
- Added full support for RedHat 7.1 and the gcc 2.96 compiler. However, the standard universe binaries must still be statically linked.
- When jobs are suspended or unsuspended, an event is now written into the user job log.
- Added [*-a*] flag to *condor_submit* to add/override attributes specified in the submit file.
- Under Unix, added the ability for a submittor of a job to describe when and how a job is allowed/not allowed to leave the queue. For example, if a job has only run for 5 minutes, but it was supposed to have run an hour minimum, then do not let the job leave the queue but restart it instead.
- New environment variable available CONDOR_SCRATCH_DIR available in a standard or vanilla job's environment that denotes temporary space the job can use that will be cleaned up automatically when the job leaves from the machine.

- Not exactly a new feature, but some internal parts of Condor had been fixed up to try and improve the memory footprint of a few of our daemons.

Bugs Fixed:

- Fixed a bug where *condor_q* would produce wildly inaccurate run time reports of jobs in the queue.
- Fixed it so that if the condor scheduler fails to notify the administrator through email, just print a warning and do not except.
- Fixed a bug where *condor_submit* would incorrectly create the user log file.
- Fixed a bug where a job queue sorted by date with *condor_q* would be displayed in descending instead of ascending order.
- Fixed and improved error handling when *condor_submit* fails.
- Numerous fixes in the Condor User Log System.
- Fixed a bug where when Condor inspects its on disk job queue log, it would do it with case sensitivity. Now there is no case sensitivity.
- Fixed a bug in *condor_glidein* where it have trouble figuring out the architecture of a minimally installed HP-UX machine.
- Fixed it so that email to the user has the word "condor" capitalized in the subject.
- Fixed a situation where when a user has multiple schedulers submitting to the same pool, the Negotiator would starve some of the schedulers.
- Added a feature whereby if a transfer of an executable from a submission machine to an execute machine fails, Condor will retry a configurable numbers of times denotated by the `EXEC_TRANSFER_ATTEMPTS` macro. This macro defaults to three if left undefined. This macro exists only for the Unix port of Condor.
- Fixed a bug where if a schedd had too many rejected clusters during a match phase, it would "except" and have to be restarted by the master.

Known Bugs:

- The HP-UX Vendor C++ CFront compiler does not work with *condor_compile* if exception handling is enabled with +eh.
- The HP-UX Vendor aCC compiler does not work at all with Condor.

Version 6.3.0

New Features:

- Added support for running MPICH jobs under Condor.

Many Condor DAGMan improvements and bug fixes:

- PRE and POST scripts now run asynchronously, rather than synchronously as in the past. As a result, DAGMan now supports a *-MaxScripts* option to limit the number of simultaneously running PRE and POST scripts.
- Whether or not POST scripts are always executed after failed jobs is now configurable with the *-NoPostFail* argument.
- Added a *-r* flag to *condor_submit_dag* to submit DAGMan to a remote *condor_schedd*.
- Made the arguments to *condor_submit_dag* case-insensitive.
- Fixed a variety of bugs in DAGMan's event handling, so DAGMan should no longer hang indefinitely after failed jobs, or mistake one job's userlog events for those of another.
- DAGMan's error handling and logging output have been substantially clarified and improved. For example, DAGMan now prints a list of failed jobs when it exits, rather than just saying "some jobs failed".
- Jobs submitted by a *condor_dagman* job now have `DAGManJobId` and `DAGNodeName` in the job classad.
- Fixed a *condor_submit_dag* bug preventing the submission of DAGMan Rescue files.
- Improved the handling of userlog errors (less crashing, more coping).
- Fixed a bug when recovering from the userlog after a crash or reboot.
- Fixed bugs in the handling of *-MaxJobs*.
- Added a *-a line* argument to *condor_submit* to add a line to the submit file before processing (overriding the submit file).
- Added a *-dag* flag to *condor_q* to format and sort DAG jobs sensibly under their DAGMan master job.

Known Bugs:

- *condor_kbdd* doesn't work properly under Compaq Tru64 5.1, and as a result, resources may not leave the "Unclaimed" state regardless of keyboard or pty activity. Compaq Tru64 5.0a and earlier do work properly.

8.4 Stable Release Series 6.2

This is the second stable release series of Condor. All of the new features developed in the 6.1 series are now considered stable, supported features of Condor. New releases of 6.2.0 should happen infrequently and will only include bug fixes and support for new platforms. New features will be added and tested in the 6.3 development series. The details of each version are described below.

8.4.1 Version 6.2.2

New Features:

- Man pages are now included in the Condor distribution for UNIX.
- Condor administrators can now specify `CONDOR_SUPPORT_EMAIL` in their config file. This address is included at the bottom of all email Condor sends out. Previously, the `CONDOR_ADMIN` setting was used for this, but at many sites, the address where the Condor daemons should send email about administrative problems is not the same address that users should use for technical support. If your site has different addresses for these two things, you can now specify them properly.
- There is a new macro automatically defined by Condor for use in your config files: `IP_ADDRESS` . If you refer to `$(IP_ADDRESS)`, it will be replaced with the ASCII string version of the local host's IP address.

Bugs Fixed:

- Fixed a bug with the scheduler universe where if one or more of the stderr, stdout, or stdin files of a scheduler universe job couldn't be opened, the *condor_schedd* would leak file descriptors.
- Fixed the default startd policy expressions that are shipped with Condor. When you upgrade Condor, if you keep your old `condor_config` file and do not use the new one we ship with the Condor binaries, we highly recommend you open the `<release_dir>/etc/condor/config.generic` file and see what's changed. Carefully read part 3, see what's different from your existing policy expressions, and make the relevent changes to your own expressions. Here is a summary of the improvements made:
  - The policy expressions now use `CpuBusyTime` to avoid problems with faulty values for `CondorLoadAvg`. For certain kinds of jobs (mostly vanilla jobs), the old policy could cause the jobs to cycle between the running and suspended states.
  - The policy macros that refered to `JobUniverse` and `ImageSize` now use "TARGET." to ensure we're refering to the right ClassAd to find those attributes. Our old policy used to make important decisions based on the universe and image size of the job it was planning to run, but instead, it was using the values from the job that was currently running.

  - Added verbose comments to explain the policy in English.

- If *condor_compile* can not find the *condor_config_val* program, *condor_compile* can not work. In previous versions, *condor_compile* would try to perform the compilation and fail in an unclear way with cryptic error messages. Now, *condor_compile* detects this case, prints out a verbose error message, and exits.
- The `FILESYSTEM_DOMAIN` and `UID_DOMAIN` settings were not being automatically append to the requirements of a Vanilla-universe job, possibly causing it to run on machines where it will not successfully run. This has been fixed.
- The getdirentries call was not being trapped, causing a few applications to fail running inside of Condor.
- When the NT shadow and the Unix shadow were used in conjunction with each other during the submission of heterogeneous jobs, they conflicted in where they should store their Condor internal files. This would cause extremely long hangs where the job was listed as running but no job actually started. This has been fixed and now you can mix the NT shadow and Unix shadow together just fine during heterogeneous submits.
- Numerous additions/clarifications to sections of the manual to bring the manual up to date with the source base.
- Fixed a bug where if you set `MAX_JOBS_RUNNING` to zero in the config files, the schedd would fail with a floating point error.
- PVM support for Solaris 2.8 was mistakenly turned off in 6.2.1, it has been turned back on again.
- Added exit status of Condor tools and daemons to the manual.
- Fixed a bug in the schedd where it would segfault if it manipulated class ads in a certain way.
- Fixed `stat()` and `lstat()` to ask the shadow where the file was that it was trying to locate instead of assuming it was always going to be remote I/O.
- Fixed a bug where Condor would incorrectly inform you that it didn't have read/write permission on a file located in NFS when you actually did have permission to read/write it.
- Removed mention of *condor_master_off* since it was deprecated a long time ago and is not available anymore.
- Removed mention of *condor_reconfig_schedd* since it was deprecated a long time ago and is not available anymore.
- Fixed a bug where the schedd would occasionally EXCEPT with an "`ATTR_JOB_STATUS` not found" error soon after a *condor_rm* command had been invoked.
- Fixed a bug in the Negotiator and the Schedd where it would segfault during reading of a corrupted log file.

- Fixed a bug where we used to EXCEPT if we couldn't email the administrator of the pool. We now do not EXCEPT anymore.
- In the *condor_startd*, messages were being printed into the log file about load average and idle time computations whenever D_FULLDEBUG was included in the STARTD_DEBUG config file setting. Now, to see the periodic messages printing out the current load averages, you must include D_LOAD in STARTD_DEBUG , and to see the messages about idle time, you must include D_IDLE.
- In most of the Condor tools (*condor_on*, *condor_off*, *condor_restart*, *condor_vacate*, *condor_checkpoint*, *condor_reconfig*, *condor_reschedule*), if you used **-pool** to specify which pool to query to find specific daemons, but did not specify any daemons or machines, the tool would just act on the local host. Now, if you specify **-pool** and do not list any machines, these Condor tools will print an error message and exit.
- If you specified a file in the CONDOR_CONFIG environment variable, all Condor daemons and tools used to silently exit. Now, they print an error message about it and exit. Also, if there were any problems with the file you specified in the CONDOR_CONFIG environment variable, Condor used to try other options, namely the "condor" user's home directory and /etc/condor_config. Now, if you specify something in CONDOR_CONFIG and Condor cannot read it, Condor will print an error and exit, instead of searching the other locations.

Known Bugs:

- *condor_compile* on DIGITAL UNIX 4.0 with the Vendor cc compiler now passes a different option to specify the path to find the Condor libraries. We believe that this new option is correct for all versions of the compiler, but do not have enough testing options to confirm this.
- Condor does not work on international version of Windows 2000. It has only been tested with the US version.

8.4.2 Version 6.2.1

New Features:

- The *condor_userlog* command is now available on Windows NT.
- Jobs run in stand-alone checkpointing mode can now take a -_condor_nowarn argument, which silences the warnings from the system call library when you perform a checkpoint-unsafe action, such as opening a file for reading and writing.

Bugs Fixed:

- When using heterogeneous specifications of an executable between NT and UNIX, Condor could get confused if the vanilla job had run on an NT machine, vacated without being completed, and then restarted as a standard universe job on UNIX. The job would be labeled as running in the queue, but not perform any work. This has been fixed.

- The entries in the `environment` option in a submit file now correctly override the variables brought in from the `getenv` option on Windows NT. In previous version of CondorNT, the job would get an environment with the variable defined multiple times. This bug did not affect UNIX versions of Condor.
- Some service packs of Windows NT had bugs that prevented Condor from determining the file permissions on input and output files. 6.2.1 uses a different set of API's to determine the permissions and works properly across all service packs
- In versions of Condor previous to 6.2.0, the registry would slowly grow on Windows NT and sometimes become corrupted. This was fixed in 6.2.0, but if a previously-corrupted registry was detected Condor aborted. In 6.2.1, this has been turned into a warning, as it doesn't need to be a fatal error.
- Fixed a memory-corruption bug in the *condor_collector*
- PVM resources in Condor were unable to have more than one `@` symbol in a name.
- The `TRANSFER_FILES` is now set to `ON_EXIT` on UNIX by default for the vanilla universe. Previously, users submitting from UNIX to NT needed to explicitly enable it or include the executable in the list of input files for the job to run.
- If `TRANSFER_FILES` was set to `TRUE` files created during the job's run would be transfered whenever the job was vacated and transfered to the next machine the job ran on, but would not be transfered back to the submit machine when the job finally exited for the last time.
- Determining the current working directory was broken in stand-alone checkpointing.
- A job's standard output and standard error can now go to the same file.
- When the `START_HAS_BAD_UTMP` is set to TRUE, the *condor_startd* now detects activity on the

  `/dev/pts`

  devices.
- The *condor_negotiator* in 6.2.0 could incorrectly reject a job that should have been successfully matched if it previously rejected a job. If the same jobs were sent to the *condor_negotiator* in a different order, the match that should succeed would. In 6.2.1, the order is no longer important, and previous rejections will not prevent future matches.
- The getdents, getdirents, and statfs system calls now work correctly in cross-platform submissions.
- *condor_compile* is better able to detect which version of Linux it is running on and which flags it should pass to the linker. This should help Condor users on non-Red Hat distributions.
- Fixed a bug in the *condor_startd* that would cause the daemon to crash if you set the `POLLING_INTERVAL` macro to a value greater than 60.

- In *condor_q*, dash-arguments (e.g., -pool, -run, etc.) were being parsed incorrectly such that the same arguments specified without a dash would be interpreted as if the dash were present, making it impossible to specify "pool" or "globus" or "run" as an owner argument.
- Fixed bug in *condor_submit* that would cause certain submit file directives to be silently ignored if you used the wrong attribute name. Now, all submit file attributes can use the same names you see in the job ClassAd (what you'd see with *condor_q* **-long**. For example, you can now use "CoreSize = 0" or "core_size = 0" in your submit file, and either one would be recognized.
- A static limit on the number of clusters the *condor_schedd* would accept from the *condor_negotiator* was removed.
- On Windows NT, if a job's log file was in a non-existent location, both the *condor_submit* and the *condor_schedd* would crash.
- Encounting unsupported system calls could cause Condor to corrupt the signal state of the job.
- Fixed some of the error messages in *condor_submit* so that they are all consistently formatted.
- Fixed a bug in the Linux standard universe where *calloc*(2) would not return zero filled memory.
- *condor_rm*, *condor_hold* and *condor_release* will now return a non-zero exit status on failure, and only return 0 on success. Previously, they always returned status 0.
- If a user accidentally put

  ```
  notify_user = false
  ```

  in their submit file, Condor used to treat that as a valid entry. Now, *condor_submit* prints out a warning in this case, telling the user that they probably want to use

  ```
  notification = never
  ```

  instead.

Known Bugs:

- It may be possible to checkpoint with an open socket on IRIX 6.2. On restart, the job will abort and go back into the queue.

Version 6.2.0

New Features Over the 6.0 Release Series

- Support for running multiple jobs on SMP (Symmetric Multi-Processor) machines.

New Features Over the Last Development Series: 6.1.17

- If `CkptArch` isn't specified in the job submission file's `Requirements` attribute, then automatically add this expression:

  ```
  CkptRequirements = ((CkptArch == Arch) || (CkptArch =?= UNDE-
  FINED)) &&
  ((CkptOpSys == OpSys) || (CkptOpSys =?= UNDEFINED))
  ```

  to the `Requirements` expression. This allows for users who specify a heterogeneous submission to not have to worry about having their checkpoints incorrectly starting up on architectures for which they were not designed to run.

- The `APPEND_REQ_<universe>` config file entries now get appended to the beginning of the expressions before Condor adds internal default expressions. This allows the sysadmin to override any default policy that Condor enforces.

- There is now a single `APPEND_REQUIREMENTS` attribute that will get appended to all universe's `Requirements` expressions unless a specific `APPEND_REQ_STANDARD` or `APPEND_REQ_VANILLA` expression is defined.

- Increased certain networking parameters to help alleviate the *condor_shadow*'s inability to contact the *condor_schedd* during heavy load of the system.

- Added a *condor_glidein* man page to the manual.

- Some of the log messages in the *condor_startd* were modified to be more clear and to provide more information.

- Added a new attribute to the *condor_startd* ClassAd when the machine is claimed, `RemoteOwner`.

Bugs fixed since 6.1.17

- On NT, the Registry would increase in size while Condor was servicing jobs. This has been fixed.

- Added `utmpx` support for Solaris 2.8 to fix a problem where `KeyBoardIdle` wasn't being set correctly.

- When doing a *condor_hold* under NT, the job was removed instead of held. This has been fixed.

- When using the *-master* argument to*condor_restart*, the *condor_master* used to exit instead of restarting. Now, the *condor_master* correctly restarts itself in this case.

Known Bugs:

- `STARTD_HAS_BAD_UTMP` does not work if set to True on Solaris 2.8. However, since `utmpx` support is enabled, you shouldn't need to do this normally.
- *condor_kbdd* doesn't work properly under Compaq Tru64 5.1, and as a result, resources may not leave the "Unclaimed" state regardless of keyboard or pty activity. Compaq Tru64 5.0a and earlier do work properly.

8.5 Development Release Series 6.1

This was the first development release series. It contains numerous enhancements over the 6.0 stable series. For example:

- Support for running multiple jobs on SMP machines
- Enhanced functionality for pool administrators
- Support for PVM, MPI and Globus jobs
- Support for *Flocking* jobs across different Condor pools

The 6.1 series has many other improvements over the 6.0 series, and is available on more platforms. The new features, bugs fixed, and known bugs of each version are described below in detail.

8.5.1 Version 6.1.17

This version is the 6.2.0 "release candidate". It was publically released in Feburary of 2001, and it will be released as 6.2.0 once it is considered "stable" by heavy testing at the UW-Madison Computer Science Department Condor pool.

New Features:

- Hostnames in the HOSTALLOW and HOSTDENY entries are now case-insensitive.
- It is now possible to submit NT jobs from a UNIX machine.
- The NT release of Condor now supports a USE_VISIBLE_DESKTOP parameter. If true, Condor will allow the job to create windows on the desktop of the execute machine and interact with the job. This is particularly useful for debugging why an application will not run under Condor.
- The *condor_startd* contains support for the new MPI dedicated scheduler that will appear in the 6.3 development series. This will allow you to use your 6.2 Condor pool with the new scheduler.

- Added a **mixedcase** option to *condor_config_val* to allow for overriding the default of lower-casing all the config names
- Added a pid_snapshot_interval option to the config file to control how often the *condor_startd* should examine the running process family. It defaults to 50 seconds.

Bugs Fixed:

- Fixed a bug with the *condor_schedd* reaching the MAX_JOBS_RUNNING mark and properly calculating Scheduler Universe jobs for preemption.
- Fixed a bug in the *condor_schedd* loosing track of *condor_startd*s in the initial claiming phase. This bug affected all platforms, but was most likely to manifest on Solaris 2.6
- CPU Time can be greater than wall clock time in Multi-threaded apps, so do not consider it an error in the UserLog.
- *condor_restart* **-master** now works correctly.
- Fixed a rare condition in the *condor_startd* that could corrupt memory and result in a signal 11 (SIGSEGV, or segmentation violation).
- Fixed a bug that would cause the "execute event" to not be logged to the UserLog if the binary for the job resided on AFS.
- Fixed a race-condition in Condor's PVM support on SMP machines (introduced in version 6.1.16) that caused PVM tasks to be associated with the wrong daemon.
- Better handling of checkpointing on large-memory Linux machines.
- Fixed random occasions of job completion email not being sent.
- It is no longer possible to use *condor_user_prio* to set a priority of less than 1.
- Fixed a bug in the job completion email statistics. Run Time was being underreported when the job completed after doing a periodic checkpoint.
- Fixed a bug that caused CondorLoadAvg to get stuck at 0.0 on Linux when the system clock was adjusted.
- Fixed a *condor_submit* bug that caused all machine_count commands after the first queue statement to be ignored for PVM jobs.
- PVM tasks now run as the user when appropriate instead of always running under the UNIX "nobody" account.
- Fixed support for the PVM group server.
- PVM uses an environment variable to communicate with it's children instead of a file in /tmp. This file previously could become overwritten by mulitple PVM jobs.

- *condor_stats* now lives in the "bin" directory instead of "sbin".

Known Bugs:

- The *condor_negotiator* can crash if the Accountantnew.log file becomes corrupted. This most often occurs if the Central Manager runs out of diskspace.

8.5.2 Version 6.1.16

New Features:

- Condor now supports multiple pvmds per user on a machine. Users can now submit more than one PVM job at a time, PVM tasks can now run on the submission machine, and multiple PVM tasks can run on SMP machines. *condor_submit* no longer inserts default job requirements to restrict PVM jobs to one pvmd per user on a machine. This new functionality requires the *condor_pvmd* included in this (and future) Condor releases. If you set "PVM_OLD_PVMD = True" in the Condor configuration file, *condor_submit* will insert the default PVM job requirements as it did in previous releases. You must set this if you don't upgrade your *condor_pvmd* binary or if your jobs flock with pools that user an older *condor_pvmd*.
- The NT release of Condor no longer contains debugging information. This drastically reduces the size of the binaries you must install.

Bugs Fixed:

- The configuration files shipped with version 6.1.15 contained a number of errors relating to host-based security, the configuration of the central manager, and a few other things. These errors have all been corrected.
- Fixed a memory management bug in the *condor_schedd* that could cause it to crash under certain circumstances when machines were taken away from the schedd's control.
- Fixed a potential memory leak in a library used by the *condor_startd* and *condor_master* that could leak memory while Condor jobs were executing.
- Fixed a bug in the NT version of Condor that would result in faulty reporting of the load average.
- The *condor_shadow.pvm* should now correctly return core files when a task or *condor_pvmd* crashes.
- This release fixes a memory error introduced in version 6.1.15 that could crash the *condor_shadow.pvm*.

- Some *condor_pvmd* binaries in previous releases included debugging code we added that could cause the *condor_pvmd* to crash. This release includes new *condor_pvmd* binaries for all platforms with the problematic debugging code removed.
- Fixed a bug in the **-unset** options to *condor_config_val* that was introduced in version 6.1.15. Both **-unset** and **-runset** work correctly, now.

Known Bugs:

- None.

8.5.3 Version 6.1.15

New Features:

- In the job submit description file passed to *condor_submit*, a new style of macro (with two dollar-signs) can reference attributes from the machine ClassAd. This new style macro can be used in the job's `Executable`, `Arguments`, or `Environment` settings in the submit description file. For example, if you have both Linux and Solaris machines in your pool, the following submit description file will run either foo.INTEL.LINUX or foo.SUN4u.SOLARIS27 as appropiate, and will pass in the amount of memory available on that machine on the command line:

```
executable = foo.$$(Arch).$$(Opsys)
arguments = $$(Memory)
queue
```

- The `CONFIG` security access level now controls the modification of daemon configurations using *condor_config_val*. For more information about security access levels, see section **??** on page **??**.
- The `DC_DAEMON_LIST` macro now indicates to the *condor_master* which processes in the `DAEMON_LIST` use Condor's DaemonCore inter-process communication mechanisms. This allows the *condor_master* to monitor both processes developed with or without the Condor DaemonCore library.
- The new `NEGOTIATE_ALL_JOBS_IN_CLUSTER` macro can be use to configure the *condor_schedd* to not assume (for efficiency) that if one job in a cluster can't be scheduled, then no other jobs in the cluster can be scheduled. If `NEGOTIATE_ALL_JOBS_IN_CLUSTER` is set to True, the *condor_schedd* will now always try to schedule each individual job in a cluster.
- The *condor_schedd* now automatically adds any machine it is matched with to its HOSTALLOW_WRITE list. This simplifies setting up a machine for flocking, since the submitting user doesn't have to know all the machines where the job might execute, they only have to know what central manager they wish to flock to. Submitting users must trust a central manager they report to, so this doesn't impact security in any way.

- Some static limits relating to the number of jobs which can be simultaneously started by the *condor_schedd* has been removed.
- The default Condor config file(s) which are installed by the installation program have been re-organized for greater clarity and simplicity.

Bugs Fixed:

- In the STANDARD Universe, jobs submitted to Condor could segfault if they opened multiple files with the same name. Usually this bug was exposed when users would submit jobs without specifying a file for either stdout or stderr; in this case, both would default to `/dev/null`, and this could trigger the problem.
- The Linux 2.2.14 kernel, which is used by default with RedHat 6.2, has a serious bug can cause the machine to lock up when the same socket is used for repeated connection attempts. Thus, previous versions of Condor could cause the 2.2.14 kernel to hang (lots of other applications could do this as well). The Condor Team recommends that you upgrade your kernel to 2.2.16 or later. However, in v6.1.15 of Condor, a patch was added to the Condor networking layer so that Condor would not trigger this Linux kernel bug.
- If no email address was specified when the job was submitted with *condor_submit*, completion email was being sent to user@submit-machine-hostname. This is not the correct behavior. Now email goes by default to user@uid-domain, where uid-domain is defined by the `UID_DOMAIN` setting in the config file.
- The *condor_master* can now correctly shutdown and restart the *condor_checkpoint_server*.
- Email sent when a SCHEDULER Universe job compeltes now has the correct From: header.
- In the STANDARD universe, jobs which call sigsuspend() will now receive the correct return value.
- Abnormal error conditions, such as the hard disk on the submit machine filling up, are much less likely to result in a job disappearing from the queue.
- The *condor_checkpoint_server* now correctly reconfigures when a *condor_reconfig* command is received by the *condor_master*.
- Fixed a bug with how the *condor_schedd* associates jobs with machines (claimed resources) which would, under certain circumstances, cause some jobs to remain idle until other jobs in the queue complete or are preempted.
- A number of PVM universe bugs are fixed in this release. Bugs in how the *condor_shadow.pvm* exited, which caused jobs to hang at exit or to run multiple times, have been fixed. The *condor_shadow.pvm* no longer exits if there is a problem starting up PVM on one remote host. The *condor_starter.pvm* now ignores the periodic checkpoint command from the startd. Previously, it would vacate the job when it received the periodic checkpoint command. A number of bugs with how the *condor_starter.pvm* handled asynchronous events, which caused it to take a long time to clean up an exited PVM task, have been fixed. The *condor_schedd*

now sets the status correctly on multi-class PVM jobs and removes them from the job queue correctly on exit. *condor_submit* no longer ignores the machine_count command for PVM jobs. And, a problem which caused pvm_exit() to hang was diagnosed: PVM tasks which call pvm_catchout() to catch the output of child tasks should be sure to call it again with a NULL argument to disable output collection before calling pvm_exit().

- The change introduced in 6.1.13 to the *condor_shadow* regarding when it logged the execute event to the user log produced situations where the shadow could log other events (like the shadow exception event) before the execute event was logged. Now, the *condor_shadow* will always log an execute event before it logs any other events. The timing is still improved over 6.1.12 and older versions, with the execute event getting logged after the bulk of the job initialization has finished, right before the job will actually start executing. However, you will no longer see user logs that contain a "shadow exception" or "job evicted" message without a "job executing" event, first.
- `stat()` and varient calls now go through the file table to get the correct logical size and access times of buffered files. Before, `stat()` used to return zero size on a buffered file that had not yet been synced to disk.

Known Bugs:

- On IRIX 6.2, C++ programs compiled with GNU C++ (g++) 2.7.2 and linked with the Condor libraries (using *condor_compile*) will not execute the constructors for any global objects. There is a work-around for this bug, so if this is a problem for you, please send email to condor-admin@cs.wisc.edu.
- In HP-UX 10.20, *condor_compile* will not work correctly with HP's C++ compiler. The jobs might link, but they will produce incorrect output, or die with a signal such as SIGSEGV during restart after a checkpoint/vacate cycle. However, the GNU C/C++ and the HP C compilers work just fine.
- The `getrusage()` call does not work always as expected in STANDARD Universe jobs. If your program uses `getrusage()`, it could decrease incorrectly by a second across a checkpoint and restart. In addition, the time it takes Condor to restart from a checkpoint is included in the usage times reported by `getrusage()`, and it probably should not be.

8.5.4 Version 6.1.14

New Features:

- Initial supported added for RedHat Linux 6.2 (i.e. glibc 2.1.3).

Bugs Fixed:

- In version 6.1.13, periodic checkpoints would not occur (see the Known Bugs section for v6.1.13 listed below). This bug, which only impacts v6.1.13, has been fixed.

Known Bugs:

- The `getrusage()` call does not work properly inside "standard" jobs. If your program uses `getrusage()`, it will not report correct values across a checkpoint and restart. If your program relies on proper reporting from `getrusage()`, you should either use version 6.0.3 or 6.1.10.
- While Condor now supports many networking calls such as `socket()` and `connect()`, (see the description below of this new feature added in 6.1.11), on Linux, we cannot at this time support `gethostbyname()` and a number of other database lookup calls. The reason is that on Linux, these calls are implemented by bringing in a shared library that defines them, based on whether the machine is using DNS, NIS, or some other database method. Condor does not support the way in which the C library tries to explicitly bring in these shared libraries and use them. There are a number of possible solutions to this problem, but the Condor developers are not yet agreed on the best one, so this limitation might not be resolved by 6.1.14.
- In HP-UX 10.20, *condor_compile* will not work correctly with HP's C++ compiler. The jobs might link, but they will produce incorrect output, or die with a signal such as SIGSEGV during restart after a checkpoint/vacate cycle. However, the GNU C/C++ and the HP C compilers work just fine.
- When a program linked with the Condor libraries (using *condor_compile*) is writing output to a file, `stat()`–and variant calls, will return zero for the size of the file if the program has not yet read from the file or flushed the file descriptors. This is a side effect of the file buffering code in Condor and will be corrected to the expected semantic.
- On IRIX 6.2, C++ programs compiled with GNU C++ (g++) 2.7.2 and linked with the Condor libraries (using *condor_compile*) will not execute the constructors for any global objects. There is a work-around for this bug, so if this is a problem for you, please send email to condor-admin@cs.wisc.edu.

8.5.5 Version 6.1.13

New Features:

- Added `DEFAULT_IO_BUFFER_SIZE` and `DEFAULT_IO_BUFFER_BLOCK_SIZE` to config parameters to allow the administrator to set the default file buffer sizes for user jobs in *condor_submit*.
- There is no longer any difference in the configuration file syntax between "macros" (which were specified with an "=" sign) and "expressions" (which were specified with a ":" sign).

Now, all config file entries are treated and referenced as macros. You can use either "=" or ":" and they will work the same way. There is no longer any problem with forward-referencing macros (referencing macros you haven't yet defined), so long as they are eventually defined in your config files (even if the forward reference is to a macro defined in another config file, like the local config file, for example).

- *condor_vacate* now supports a **-fast** option that forces Condor to hard-kill the job(s) immediately, instead of waiting for them to checkpoint and gracefully shutdown.
- *condor_userlog* now displays times in days+hours:minutes format instead of total hours or total minutes.
- The *condor_run* command provides a simple front-end to *condor_submit* for submitting a shell command-line as a vanilla universe job.
- Solaris 2.7 SPARC, 2.7 INTEL have been added to the list of ports that now support remote system calls and checkpointing.
- Any mail being sent from Condor now shows up as having been sent from the designated Condor Account, instead of root or "Super User".
- The *condor_submit* "hold" command may be used to submit jobs to the queue in the hold state. Held jobs will not run until released with *condor_release*.
- It is now possible to use checkpoint servers in remote pools when flocking even if the local pool doesn't use a checkpoint server. This is now the default behavior (see the next item).
- `USE_CKPT_SERVER` now defaults to True if a checkpoint server is available. It is usually more efficient to use a checkpoint server near the execution site instead of storing the checkpoint back to the submission machine, especially when flocking.
- All Condor tools that used to expect just a hostname or address (*condor_checkpoint*, *condor_off*, *condor_on*, *condor_restart*, *condor_reconfig*, *condor_reschedule*, *condor_vacate*) to specify what machine to effect, can now take an optional **-name** or **-addr** in front of each target. This provides consistancy with other Condor tools that require the **-name** or **-addr** options. For all of the above mentioned tools, you can still just provide hostnames or addresses, the new flags are not required.
- Added **-pool** and **-addr** options to *condor_rm*, *condor_hold* and *condor_release*.
- When you start up the *condor_master* or *condor_schedd* as any user other than "root" or "condor" on Unix, or "SYSTEM" on NT, the daemon will have a default `Name` attribute that includes both the username of the user who the daemon is running as and the full hostname of the machine where it is running.
- Clarified our Linux platform support. We now officially support the RedHat 5.2 and 6.x distributions, and although other Linux distributions (especially those with similar libc versions) may work, they are not tested or supported.

- The schedd now periodically updates the run-time counters in the job queue for running jobs, so if the schedd crashes, the counters will remain relatively up-to-date. This is controlled by the `WALL_CLOCK_CKPT_INTERVAL` parameter.
- The *condor_shadow* now logs the "job executing" event in the user log after the binary has been successfully transfered, so that the events appear closer to the actual time the job starts running. This can create some somewhat unexpected log files. If something goes wrong with the job's initialization, you might see an "evicted" event before you see an "executing" event.

Bugs Fixed:

- Fixed how we internally handle file names for user jobs. This fixes a nasty bug due to changing directories between checkpoints.
- Fixed a bug in our handling of the `Arguments` macro in the command file for a job. If the arguments were extremely long, or there were an extreme number of them, they would get corrupted when the job was spawned.
- Fixed DAGMan. It had not worked at all in the previous release.
- Fixed a nasty bug under Linux where file seeks did not work correctly when buffering was enabled.
- Fixed a bug where *condor_shadow* would crash while sending job completion e-mail forcing a job to restart multiple times and the user to get multiple completion messages.
- Fixed a long standing bug where Fortran 90 would occasionally truncate its output files to random sizes and fill them with zeros.
- Fixed a bug where `close()` did not propogate its return value back to the user job correctly.
- If a SIGTERM was delivered to a *condor_shadow*, it used to remove the job it was running from the job queue, as if *condor_rm* had been used. This could have caused jobs to leave the queue unexpectedly. Now, the *condor_shadow* ignores SIGTERM (since the *condor_schedd* knows how to gracefully shutdown all the shadows when it gets a SIGTERM), so jobs should no longer leave the queue prematurely. In addition, on a SIGQUIT, the shadow now does a fast shutdown, just like the rest of the Condor daemons.
- Fixed a number of bugs which caused checkpoint restarts to fail on some releases of Irix 6.5 (for example, when migrating from a mips4 to a mips3 CPU or when migrating between machines with different pagesizes).
- Fixed a bug in the implementation of the `stat()` family of remote system calls on Irix 6.5 which caused file opens in Fortran programs to sometimes fail.
- Fixed a number of problems with the statistics reported in the job completion email and by *condor_q* **-goodput**, including the number of checkpoints and total network usage. Correct values will now be computed for all new jobs.

- Changes in `USE_CKPT_SERVER` and `CKPT_SERVER_HOST` no longer cause problems for jobs in the queue which have already checkpointed.
- Many of the Condor administration tools had a bug where they would suffer a segmentation violation if you specified a **-pool** option and did not specify a hostname. This case now results in an error message instead.
- Fixed a bug where the *condor_schedd* could die with a segmentation violation if there was an error mapping an IP address into a hostname.
- Fixed a bug where resetting the time in a large negative direction caused the *condor_negotiator* to have a floating point error on some platforms.
- Fixed *condor_q*'s output so that certain arguments are not ignored.
- Fixed a bug in *condor_q* where issuing a **-global** with a fairly restrictive **-constraint** argument would cause garbage to be printed to the terminal sometimes.
- Fixed a bug which caused jobs to exit without completing a checkpoint when preempted in the middle of a periodic checkpoint. Now, the jobs will complete their periodic checkpoint in this case before exiting.

Known Bugs:

- Periodic checkpoints do not occur. Normally, when the config file attribute `PERIODIC_CHECKPOINT` evaluates to True, Condor performs a periodic checkpoint of the running job. This bug has been fixed in v6.1.14. NOTE: there is a work-around to permit periodic checkpoints to occur in v6.1.13: include the attribute name "PERIODIC_CHECKPOINT" to the attributes listed in the `STARTD_EXPRS` entry in the config file.
- The `getrusage()` call does not work properly inside "standard" jobs. If your program uses `getrusage()`, it will not report correct values across a checkpoint and restart. If your program relies on proper reporting from `getrusage()`, you should either use version 6.0.3 or 6.1.10.
- While Condor now supports many networking calls such as `socket()` and `connect()`, (see the description below of this new feature added in 6.1.11), on Linux, we cannot at this time support `gethostbyname()` and a number of other database lookup calls. The reason is that on Linux, these calls are implemented by bringing in a shared library that defines them, based on whether the machine is using DNS, NIS, or some other database method. Condor does not support the way in which the C library tries to explicitly bring in these shared libraries and use them. There are a number of possible solutions to this problem, but the Condor developers are not yet agreed on the best one, so this limitation might not be resolved by 6.1.14.
- In HP-UX 10.20, *condor_compile* will not work correctly with HP's C++ compiler. The jobs might link, but they will produce incorrect output, or die with a signal such as SIGSEGV during restart after a checkpoint/vacate cycle. However, the GNU C/C++ and the HP C compilers work just fine.

- When writing output to a file, `stat()`–and variant calls, will return zero for the size of the file if the program has not yet read from the file or flushed the file descriptors, This is a side effect of the file buffering code in Condor and will be corrected to the expected semantic.
- On IRIX 6.2, C++ programs compiled with GNU C++ (g++) 2.7.2 and linked with the Condor libraries (using *condor_compile*) will not execute the constructors for any global objects. There is a work-around for this bug, so if this is a problem for you, please send email to condor-admin@cs.wisc.edu.

8.5.6 Version 6.1.12

Version 6.1.12 fixes a number of bugs from version 6.1.11. If you linked your "standard" jobs with version 6.1.11, you should upgrade to 6.1.12 and re-link your jobs (using *condor_compile*) as soon as possible.

New Features:

- None.

Bugs Fixed:

- A number of system calls that were not being trapped by the Condor libraries in version 6.1.11 are now being caught and sent back to the submit machine. Not having these functions being executed as remote system calls prevented a number of programs from working, in particular Fortran programs, and many programs on IRIX and Solaris platforms.
- Sometimes submitted jobs report back as having no owner and have **-????-** in the status line for the job. This has been fixed.
- *condor_q* **-io** has been fixed in this release.

Known Bugs:

- The `getrusage()` call does not work properly inside "standard" jobs. If your program uses `getrusage()`, it will not report correct values across a checkpoint and restart. If your program relies on proper reporting from `getrusage()`, you should either use version 6.0.3 or 6.1.10.
- While Condor now supports many networking calls such as `socket()` and `connect()`, (see the description below of this new feature added in 6.1.11), on Linux, we cannot at this time support `gethostbyname()` and a number of other database lookup calls. The reason is that on Linux, these calls are implemented by bringing in a shared library that defines them, based on whether the machine is using DNS, NIS, or some other database method. Condor does not support the way in which the C library tries to explicitly bring in these shared libraries

and use them. There are a number of possible solutions to this problem, but the Condor developers are not yet agreed on the best one, so this limitation might not be resolved by 6.1.13.

- In HP-UX 10.20, *condor_compile* will not work correctly with HP's C++ compiler. The jobs might link, but they will produce incorrect output, or die with a signal such as SIGSEGV during restart after a checkpoint/vacate cycle. However, the GNU C/C++ and the HP C compilers work just fine.
- When writing output to a file, `stat()`–and variant calls, will return zero for the size of the file if the program has not yet read from the file or flushed the file descriptors, This is a side effect of the file buffering code in Condor and will be corrected to the expected semantic.
- On IRIX 6.2, C++ programs compiled with GNU C++ (g++) 2.7.2 and linked with the Condor libraries (using *condor_compile*) will not execute the constructors for any global objects. There is a work-around for this bug, so if this is a problem for you, please send email to condor-admin@cs.wisc.edu.
- The **-format** option in *condor_q* has no effect when querying remote machines with the **-n** option.
- *condor_dagman* does not work at all in this release. The behaviour of its failure is to exit immediately with a success and to not perform any work. It will be fixed in the next release of Condor.

8.5.7 Version 6.1.11

New Features:

- *condor_status* outputs information for held jobs instead of MaxRunningJobs when supplied with **-schedd** or **-submitter**.
- *condor_userprio* now prints 4 digit years (for Y2K compiance). If you give a two digit date, it also will assume that 1/1/00 is 1/1/2000 and not 1/1/1900.
- IRIX 6.5 has been added to the list of ports that now support remote system calls and checkpointing.
- *condor_q* has been fixed to be faster and much more memory efficient. This is much more obvious when getting the queue from *condor_schedd*'s that have more than 1000 jobs.
- Added support for support for socket() and pipe() in standard jobs. Both sockets and pipes are created on the executing machine. Checkpointing is deferred anytime a socket or pipe is open.
- Added limited support for select() and poll() in standard jobs. Both calls will work only on files opened locally.

- Added limited support for fcntl() and ioctl() in standard jobs. Both calls will be performed remotely if the control-number is understood and the third argument is an integer.
- Replaced buffer implementation in standard jobs. The new buffer code reads and writes variable sized chunks. It will never issue a read to satisfy a write. Buffering is enabled by default.
- Added extensive feedback on I/O performance in the user's email.
- Added **-io** option to *condor_q* to show I/O statistics.
- Removed libckpt.a and libzckpt.a. To build for standalone checkpointing, just do a regular *condor_compile*. No -standalone option is necessary.
- The checkpointing library now only re-opens files when they are actually used. If files or other needed resources cannot be found at restart time, the checkpointer will fail with a verbose error.
- The `RemoteHost` and `LastRemoteHost` attributes in the job classad now contain hostnames instead IP address and port numbers. The **-run** option of older versions of *condor_q* is not compatible with this change.
- Condor will now automatically check for compatibility between the version of the Condor libraries you have linked into a standard job (using *condor_compile*) and the version of the *condor_shadow* installed on your submit machine. If they are incompatible, the *condor_shadow* will now put your job on hold. Unless you set "Notification = Never" in your submit file, Condor will also send you email explaining what went wrong and what you can do about it.
- All Condor daemons and tools now have a `CondorPlatform` string, which shows which platform a given set of Condor binaries was built for. In all places that you used to see `CondorVersion`, you will now see both `CondorVersion` and `CondorPlatform`, such as in each daemon's ClassAd, in the output to a **-version** option (if supported), and when running *ident* on a given Condor binary. This string can help identify situations where you are running the wrong version of the Condor binaries for a given platform (for example, running binaries built for Solaris 2.5.1 on a Solaris 2.6 machine).
- Added commented-out settings in the default `condor_config` file we ship for various SMP-specific settings in the *condor_startd*. Be sure to read section 3.10.8 on "Configuring the Startd for SMP Machine" on page 192 for details about using these settings.
- *condor_rm*, *condor_hold*, and *condor_release* all support **-help** and **-version** options now.

Bugs Fixed:

- A race condition which could cause the *condor_shadow* to not exit when its job was removed has been fixed. This bug would cause jobs that had been removed with *condor_rm* to remain in the queue marked as status "X" for a long time. In addition, Condor would not shutdown quickly on hosts that had hit this race condition, since the *condor_schedd* wouldn't exit until all of its *condor_shadow* children had exited.

- A signal race condition during restart of a Condor job has been fixed.
- In a Condor linked job, `getdomainname()` is now supported.
- IRIX 6.5 can give negative time reports for how long a process has been running. We account for that now in our statistics about usage times.
- The *condor_status* memory error introduced in version 6.1.10 has been fixed.
- The `DAEMON_LIST` configuration setting is now case insensitive.
- Fixed a bug where the *condor_schedd*, under rare circumstances, cause another schedd's jobs not to be matched.
- The free disk space is now properly computed on Digital Unix. This fixed problems where the `Disk` attribute in the condor_startd classad reported incorrect values.
- The config file parser now detects incremental macro definitions correctly (see section 3.3.1 on page 82). Previously, when a macro (or expression) being defined was a substring of a macro (or expression) being referenced in its definition, the reference would be erroneously marked as an incremental definition and expanded immediately. The parser now verifies that the entire strings match.

Known Bugs:

- The output for condor_q **-io** is incorrect and will likely show zeroes for all values. A fixed version will appear in the next release.

8.5.8 Version 6.1.10

New Features:

- *condor_q* now accepts `-format` parameters like *condor_status*
- *condor_rm*, *condor_hold* and *condor_release* accept `-constraint` parameters like *condor_status*
- *condor_status* now sorts displayed totals by the first column. (This feature introduced a bug in *condor_status*. See "Known Bugs" below.)
- Condor version 6.1.10 introduces "clipped" support for Sparc Solaris version 2.7. This version does not support checkpointing or remote system calls. Full support for Solaris 2.7 will be released soon.
- Introduced code to enable Linux to use the standard C library's I/O buffering again, instead of relying on the Condor I/O buffering code (which is still in beta testing).

Bugs Fixed:

- The bug in checkpointing introduced in version 6.1.9 has been fixed. Checkpointing will now work on all platforms, as it always used to. Any jobs linked with the 6.1.9 Condor libraries will need to be relinked with *condor_compile* once version 6.1.10 has been installed at your site.

Known Bugs:

- The `CondorLoadAvg` attribute in the *condor_startd* has some problems in the way it is computed. The CondorLoadAvg is somewhat inaccurate for the first minute a job starts running, and for the first minute after it completes. Also, the computation of CondorLoadAvg is very wrong on NT. All of this will be fixed in a future version.
- A memory error may cause *condor_status* to die with SIGSEGV (segmentation violation) when displaying totals or cause incorrect totals to be displayed. This will be fixed in version 6.1.11.

8.5.9 Version 6.1.9

New Features:

- Added full support for Linux 2.0.x and 2.2.x kernels using libc5, glibc20 and glibc21. This includes support for RedHat 6.x, Debian 2.x and other popular Linux distributions. Whereas the Linux machines had once been fragmented across libc5 and GNU libc, they have now been reunified. This means there is no longer any need for the "LINUX-GLIBC" OpSys setting in your pool: all machines will now show up as "LINUX". Part of this reunification process was the removal of dynamically linked user jobs on Linux. *condor_compile* now forces static linking of your Standard Universe Condor jobs. Also, please use *condor_compile* on the same machine on which you compiled your object files.
- Added *condor_qedit* utility to allow users to modify job attributes after submission. See the new manual page on page 366.
- Added **-runforminutes** *o*ption to daemonCore to have the daemon gracefully shut down after the given number of minutes.
- Added support for statfs(2) and fstatfs(2) in user jobs. We support only the fields *f_bsize, f_blocks, f_bfree, f_bavail, f_files, f_ffree* from the structure statfs. This is still in the experimental stage.
- Added the **-direct** option to *condor_status*. After you give **-direct**, you supply a hostname, and *condor_status* will query the *condor_startd* on the specified host and display information directly from there, instead of querying the *condor_collector*. See the manual page on page 395 for details.

- Users can now define NUM_CPUS to override the automatic computation of the number of CPUs in your machine. Using this config setting can cause unexpected results, and is not recommended. This feature is only provided for sites that specifically want this behavior and know what they are doing.
- The **-set** and **-rset** options to *condor_config_val* have been changed to allow administrators to set both macros and expressions. Previously, *condor_config_val* assumed you wanted to set expressions. Now, these two options each take a single argument, the string containing exactly what you would put into the config file, so you can specify you want to create a macro by including an "=" sign, or an expression by including a ":". See section 3.3.1 on page 82 for details on macros vs. expressions. See the *condor_config_val* man page on page **??** for details on *condor_config_val*.
- If the directory you specified for LOCK (which holds lock files used by Condor) doesn't exist, Condor will now try to create that directory for you instead of giving up right away.
- If you change the COLLECTOR_HOST setting and reconfig the *condor_startd*, the startd will "invalidate" its ClassAds at the old collector before it starts reporting to the new one.

Bugs Fixed:

- Fixed a major bug dealing with the group access a Condor job is started with. Now, Condor jobs are started with all the groups the job's owner is in, not just their default group. This also fixes a security hole where user jobs could be started up in access groups they didn't belong to.
- Fixed a bug where there was a needless limitation on the number of open file descriptors a user job could have.
- Fixed a standalone checkpointing bug where we weren't blocking signals in critical sections and causing file table corruption at checkpoint time.
- Fixed a linker bug on Digital Unix 4.0 concerning fortran where the linker would fail on __uname and __sigsuspend.
- Fixed a bug in *condor_shadow* that would send incorrect job completion email under Linux.
- Fixed a bug in the remote system call of fchdir() that caused a garbage file descriptor to be used in Standard Universe jobs.
- Fixed a bug in the *condor_shadow* which was causing *condor_q* **-goodput** to display incorrect values for some jobs.
- Fixed some minor bugs and made some minor enhancements in the *condor_install* script. The bugs included a typo in one of the questions asked, and incorrect handling for the answers of a few different questions. Also, if DNS is misconfigured on your system, *condor_install* will try a few ways to find your fully qualified hostname, and if it still can't determine the correct hostname, it will prompt the user for it. In addition, we now avoid one installation step in cases were it is not needed.

- Fixed a rare race condition that could delay the completion of large clusters of short running jobs.
- Added more checking to the various arguments that might be passed to *condor_status*, so that in the case of bad input, *condor_status* will print an error message and exit, instead of performing a segmentation fault. Also, when you use the **-sort** option, *condor_status* will only display ClassAds where the attributes you use to sort are defined.
- Fixed a bug in the handling of the config files created by using the **-set** or **-rset** options to *condor_config_val*. Previously, if you manually deleted the files that were created, you could cause the affected Condor daemon to have a segmentation fault. Now, the daemons simply exit with a fatal error but still have a chance to clean up.
- Fixed a bug in the **-negotiator** option for most Condor tools that was causing it to get the wrong address.
- Fixed a couple of bugs in the *condor_master* that could cause improper shutdowns. There were cases during shutdown where we would restart a daemon (because we previously noticed a new executable, for example). Now, once you begin a shutdown, the *condor_master* will not restart anything. Also, fixed a rare bug that could cause the *condor_master* to stop checking the timestamps on a daemon.
- Fixed a minor bug in the **-owner** option to *condor_config_val* that was causing *condor_init* not to work.
- Fixed a bug where the *condor_startd*, while it was already shutting down, was allowing certain actions to succeed that should have failed. For example, it allowed itself to be matched with a user looking for available machines, or to begin a new PVM task.

Known Bugs:

- The `CondorLoadAvg` attribute in the *condor_startd* has some problems in the way it is computed. The CondorLoadAvg is somewhat inaccurate for the first minute a job starts running, and for the first minute after it completes. Also, the computation of CondorLoadAvg is very wrong on NT. All of this will be fixed in a future version.
- There is a serious bug in checkpointing when using Condor's I/O buffering for "standard" jobs. By default, Linux uses Condor buffering in version 6.1.9 for all standard jobs. The bug prevents checkpointing from working more than once. This renders the *condor_vacate* and *condor_checkpoint* commands useless, and jobs will just be killed without a checkpoint when machine owners come back to their machines.

8.5.10 Version 6.1.8

- Added *file_remaps* as command in the job submit file given to STANDARD universe jobs. A Job can now specify that it would like to have files be remapped from one file to another. In addition you can specify that files should be read from the local machine by specifing them. See the *condor_submit* manual page on page 395 for more details.

- Added *buffer_size* and *buffer_block_size* so that STANDARD universe jobs can specify that they wish to have I/O buffering turned on. Without buffering, all I/O requests in the STANDARD universe are sent back over the network to be executed on the submit machine. With buffering, read ahead, write behind, and seek batch buffering is performed to minimize network traffic and latency. By default, jobs do not specify buffering, however, for many situations buffering can drastically increase throughput. See the *condor_submit* manual page on page 395 for more details.
- The *condor_schedd* is much more memory efficient handling clusters with hundreds/thousands of jobs. If you submit large clusters, your submit machine will only use a fraction of the amount of RAM it used to require. NOTE: The memory savings will only be realized for new clusters submitted after the upgrade to v6.1.8 – clusters which previously existed in the queue at upgrade time will still use the same amount of RAM in the *condor_schedd*.
- Submitting jobs, especially submitting large clusters containing many jobs, is much faster.
- Added a **-goodput** option to *condor_q*, which displays statistics about the execution efficiency of STANDARD universe jobs.
- Added FS_REMOTE method of user authentication to possible values of the configuration option `AUTHENTICATION_METHODS` to fix problems with using the **-r** remote scheduler option of *condor_submit*. Additionally, the user authentication protocol has changed, so previous versions of Condor programs cannot co-exist with this new protocol.
- Added a new utility and documentation for *condor_glidein* which uses Globus resources to extend your local pool to use remote Globus machines as part of your Condor pool.
- Fixed more bugs in the handling of the stat() system call and its relatives on Linux with glibc. This was causing problems mainly with Fortran I/O, though other I/O related problems on glibc Linux will probably be solved now.
- Fixed a bug in various Condor tools (*condor_status*, *condor_user_prio*, *condor_config_val*, and *condor_stats*) that would cause them to seg fault on bad input to the **-pool** option.
- Fixed a bug with the **-rset** option to *condor_config_val* which could crash the Condor daemon whose configuration was being changed.
- Added *allow_startup_script* command to the job submit description file which is given to *condor_submit*. This allows the submission of a startup script to the STANDARD universe. See
- Fixed a bug in the *condor_schedd* where it would get into an infinite loop if the persistant log of the job queue got corrupted. The *condor_schedd* now correctly handles corrupted log files.
- The full release tar file now contains a `dagman` subdirectory in the `examples` directory. This subdirectory includes an example DAGMan job, including a README (in both ASCII and HTML), a Makefile, and so on.
- Condor will now insert an environment variable, `CONDOR_VM`, into the environment of the user job. This variable specifies which SMP "virtual machine" the job was started on. It will

equal either vm1, vm2, vm3, . . ., depending upon which virtual machine was matched. On a non-SMP machine, CONDOR_VM will always be set to vm1.

- Fixed some timing bugs introduced in v6.1.6 which could occur when Condor tries to simultaneously start a large number of jobs submitted from a single machine.
- Fixed bugs when Condor is told to gracefully shutdown; Condor no longer starts up new jobs when shutting down. Also, the *condor_schedd* progressively checkpoints running jobs during a graceful shutdown instead of trying to vacate all the job simultaneously. The rate at which the shutdown occurs is controlled by the JOB_START_DELAY configuration parameter (see page 105).
- Fixed a bug which could cause the *condor_master* process to exit if the Condor daemons have been hung for a while by the operating system (if, for instance, the LOG directory was placed on an NFS volume and the NFS server is down for an extended period).
- Previously, removing a large number of jobs with *condor_rm* would result in the *condor_schedd* being unresponsive for a period of time (perhaps leading to timeouts when running *condor_q*). The *condor_schedd* has been improved to multitask the removal of jobs while servicing new requests.
- Added new configuration parameter COLLECTOR_SOCKET_BUFSIZE which controls the size of TCP/IP buffers used by the *condor_collector*. For more info, see section refparam:CollectorSocketBufsize on page pagerefparam:CollectorSocketBufsize.
- Fixed a bug with the **-analyze** option to *condor_q*: in some cases, the RANK expression would not be evaluated correctly. This could cause the output from **-analyze** to be in error.
- When running on a multi-CPU (SMP) Hewlett-Packard machine, fixed bugs computing the system load average.
- Fixed bug in *condor_q* which could cause the RUN_TIME reported to be temporarily incorrect when jobs first start running.
- The *condor_startd* no longer rapidly sends multiple ClassAds one right after another to the Central Manager when its state/activity is in rapid transition. Also, on SMP machines, the *condor_startd* will only send updates for 4 nodes per second (to avoid overflowing the central manager when reporting the state of a very large SMP machine with dozens of CPUs).
- Reading a parameter with *condor_config_val* is now allowed from any machine with Host-IP READ permission. Previsouly, you needed ADMINISTRATOR permission. Of course, setting a parameter still requires ADMINISTRATOR permission.
- Worked around a bug in the StreamTokenizer Java class from Sun that we use in the CondorView client Java applet. The bug would cause errors if usernames or hostnames in your pool contained "-" or "_" characters. The CondorView applet now gets around this and properly displays all data, including entries with the "bad" characters.

8.5.11 Version 6.1.7

NOTE: Version 6.1.7 only adds support for platforms not supported in 6.1.6. There are no bug fixes, so there are no binaries released for any other platforms. You do not need 6.1.7 unless you are using one of the two platforms we released binaries for.

- Added "clipped" support for Alpha Linux machines running the 2.0.X kernel and glibc 2.0.X (such as RedHat 5.X). We do not yet support checkpointing and remote system calls on this platform, but we can start "vanilla" jobs. See section 2.4.1 on page 11 for details on vanilla vs. standard jobs.
- Re-added support for Intel Linux machines running the 2.0.X Linux kernel, glibc 2.0.X, using the GNU C compiler (gcc/g++ 2.7.X) or the EGCS compilers (versions 1.0.X, 1.1.1 and 1.1.2). This includes RedHat 5.X, and Debian 2.0. **RedHat 6.0 and Debian 2.1 are not yet supported, since they use glibc 2.1.X and the 2.2.X Linux kernel.** Future versions of Condor will support all combinations of kernels, compilers and versions of libc.

8.5.12 Version 6.1.6

- Added *file_remaps* as command in the job submit file given to *condor_submit*. This allows the user to explicitly specify where to find a given file (e.g. either on the submit or execute machine), as well as remap file access to a different filename altogether.
- Changed the way that *condor_master* spawns daemons and *condor_preen* which allows you to specify command line arguments for any of them, though a `SUBSYS_ARGS` setting. Previously, when you specified `PREEN` , you added the command line arguments directly to that setting, but that caused some problems, and only worked for *condor_preen*. **Once you upgrade to version 6.1.6, if you continue to use your old `condor_config` files, you must change the `PREEN` setting to remove any arguments you have defined and place those arguments into a separate config setting, `PREEN_ARGS` .** See section 3.3.7, "condor_master Config File Entries", on page 97 for more details.
- Fixed a very serious bug in the Condor library linked in with *condor_compile* to create standard jobs that was causing checkpointing to fail in many cases. Any jobs that were linked with the 6.1.5 Condor libraries should probably be removed, re-linked, and re-submitted.
- Fixed a bug in *condor_userprio* that was introduced in version 6.1.5 that was preventing it from finding the address of the *condor_negotiator* for your pool.
- Fixed a bug in *condor_stats* that was introduced in version 6.1.5 that was preventing it from finding the address of the *condor_collector* for your pool.
- Fixed a bug in the way the **-pool** option was handled by many Condor tools that was introduced in version 6.1.5.
- *condor_q* now displays job *allocation time* by default, instead of displaying CPU time. Job allocation time, or RUN_TIME, is the amount of wall-clock time the job has spent running.

Unlike CPU time information which is only updated when a job is checkpointed, the allocation time displayed by *condor_q* is continuously updated, even for vanilla universe jobs. By default, the allocation time displayed will be the total time across all runs of the job. The new **-currentrun** option to *condor_q* can be used to display the allocation time for solely the current run of the job. Additionally, the **-cputime** option can be used to view job CPU times as in earlier versions of Condor.

- *condor_q* will display an error message if there is a timeout fetching the job queue listing from a condor_schedd. Previously, *condor_q* would simply list the queue as empty upon a communication error.
- The condor_schedd daemon has been updated to verify all queue access requests via Condor's IP/Host-Based Security mechanism (see section 3.7.5).
- Fixed a bug on platforms which require the *condor_kbdd* (currently Digital Unix and IRIX). This bug could have allowed Condor to start a job within the first five minutes after the Condor daemons had been started, even if there is a user typing on the keyboard.
- *condor_release* now gives an error message if the user tries to release a job which either does not exist or is not in the hold state.
- Added a new config file parameter, `USER_JOB_WRAPPER` , which allows administrators to specify a file to act as a "wrapper" script around all jobs started by Condor. See inside section 3.3.12, on page 108, for more details.
- *condor_dagman* now permits the backslash character ("\") to be used as a line-continuation character for DAG Input Files, just like the condor_config files.
- The Condor version string is now included in all Condor libraries. You can now run *ident* on any program linked with *condor_compile* to view which version of the Condor libraries you linked with. In addition, the format of the version string changed in 6.1.6. Now, the identifier used is "CondorVersion" instead of "Version" to prevent any potential ambiguity. Also, the format of the date changed slightly.
- The SMP startd can now handle dynamic reconfiguration of the number of each type of virtual machine being reported. This allows you, during the normal running of the startd, to increase or decrease the number of CPUs that Condor is using. If you reconfigure the startd to use less CPUs than it currently has under its control, it will first remove CPUs that have no Condor jobs running on them. If more CPUs need to be evicted, the startd will checkpoint jobs and evict them in reverse rank order (using the startd's `Rank` expression). So, the lower the value of the rank, the more likely a job will be kicked off.
- The SMP startd contrib module's *condor_starter* no longer makes a call that was causing warning messages about "ERROR: Unknown System Call (-58) - system call not supported by Condor" when used with the 6.0.X *condor_shadow*. This was a harmless call, but removing the call prevents the error message.
- The SMP contrib module now includes the *condor_checkpoint* and *condor_vacate* programs, which allow you to vacate or checkpoint jobs on individual CPUs on the SMP, instead of checkpointing or vacating everything. You can now use "condor_vacate vm1@hostname" to

just vacate the first virtual machine, or "condor_vacate hostname" to vacate all virtual machines.

- Added support for SMP Digital Unix (Alpha) machines.
- Fixed a bug that was causing an overflow in the computation of free disk and swap space on Digital Unix (Alpha) machines.
- The *condor_startd* and *condor_schedd* now can "invalidate" their classads from the collector. So, when a daemon is shut down, or a machine is reconfigured to advertise fewer virtual machines, those changes will be instantly visible with *condor_status*, instead of having to wait 15 minutes for the stale classads to time-out.
- The *condor_schedd* no longer forks a child process (a "schedd agent") to claim available *condor_startd*s. You should no longer see multiple condor_schedd processes running on your machine after a negotiation cycle. This is now accomplished in a non-blocking manner within the *condor_schedd* itself.
- The startd now adds an `VirtualMachineID` attribute to each virtual machine classad it advertises. This is just an integer, starting at 1, and increasing for every different virtual machine the startd is representing. On regular hosts, this is the only ID you will ever see. On SMP hosts, you will see the ID climb up to the number of different virtual machines reported. This ID can be used to help write more complex policy expressions on SMP hosts, and to easily identify which hosts in your pool are in fact SMP machines.
- Modified the output for *condor_q* -run for scheduler and PVM universe jobs. The host where the scheduler universe job is running is now displayed correctly. For PVM jobs, a count of the current number of hosts where the job is running is displayed.
- Fixed the *condor_startd* so that it no longer prints lots of ProcAPI errors to the log file when it is being run as non-root.
- `FS_PATHNAME` and `VOS_PATHNAME` are no longer used. AFS support now works similar to NFS support, via the `FILESYSTEM_DOMAIN` macro.
- Fixed a minor bug in the `Condor.pm` perl module that was causing it to be case-sensitive when parsing the Condor submit file. Now, the perl module is properly case-insensitive, as indicated in the documentation.

8.5.13 Version 6.1.5

- Fixed a nasty bug in *condor_preen* that would cause it to remove files it shouldn't remove if the *condor_schedd* and/or *condor_startd* were down at the time *condor_preen* ran. This was causing jobs to mysteriously disappear from the job queue.
- Added preliminary support to Condor for running on machines with multiple network interfaces. On such machines, users can specify the IP address Condor should use in the `NETWORK_INTERFACE` config file parameter on each host. In addition, if the pool's central manager is on such a machine, users should set the `CM_IP_ADDR` parameter to the ip address you wish to use on that machine. See section 3.10.9 on page 198 for more details.

- The support for multiple network interfaces introduced bugs in *condor_userprio*, *condor_stats*, CondorPVM, and the **-pool** option to many Condor tools. All of these will be fixed in version 6.1.6.
- Fixed a bug in the remote system call library that was preventing certain Fortran operations from working correctly on Linux.
- The Linux binaries for GLIBC we now distribute are compiled on a RedHat 5.2 machine. If you're using this version of RedHat, you might have better luck with the dynamically linked version of Condor than previous releases of Condor. Sites using other GLIBC Linux distributions should continue to use the statically linked version of Condor.
- Fixed a bug in the *condor_shadow* that could cause it to die with signal 11 (segmentation violation) under certain rare circumstances.
- Fixed a bug in the *condor_schedd* that could cause it to die with signal 11 (segmentation violation) under certain rare circumstances.
- Fixed a bug in the *condor_negotiator* that could cause it to die with signal 8 (floating point exception) on Digital Unix machines.
- The following shadow parameters have been added to control checkpointing: `COMPRESS_PERIODIC_CKPT` , `COMPRESS_VACATE_CKPT` , `PERIODIC_MEMORY_SYNC` , `SLOW_CKPT_SPEED` . See section 3.3.10 on page 107 for more details. In addition, the shadow now honors the `CkptWanted` flag in a job classad, and if it is set to "False", the job will never checkpoint.
- Fixed a bug in the *condor_startd* that could cause it to report negative values for the CondorLoadAvg on rare occasions.
- Fixed a bug in the *condor_startd* that could cause it to die with a fatal exception in situations where the act of getting claimed by a remote schedd failed for some reason. This resulted in the *condor_startd* exiting on rare occasions with a message in its log file to the effect of `ERROR ``Match timed out but not in matched state''.`
- Fixed a bug in the *condor_schedd* that under rare circumstances could cause a job to be left in the "Running" state even after the *condor_shadow* for that job had exited.
- Fixed a bug in the *condor_schedd* and various tools that prevented remote read-only access to the job queue from working. So, for example, `condor_q -name foo`, if run on any machine other than foo, wouldn't display any jobs from foo's queue. This fix re-enables the following options to *condor_q* to work: **submitter**, **name**, **global**, etc.
- Changed the *condor_schedd* so that when starting jobs, it always sorts on the cluster number, in addition to the date the jobs were enqueued and the process number within clusters, so that if many clusters were submitted at the same time, the jobs are started in order.
- Fixed a bug in *condor_compile* that was modifying the `PATH` environment variable by adding things to the front of it. This would potentially cause jobs to be compiled and linked with a different version of a compiler than they thought they were getting.

- Minor change in the way the *condor_startd* handles the D_LOAD and D_KEYBOARD debug flags. Now, each one, when set, will only display every UPDATE_INTERVAL , regardless of the startd state. If you wish to see the values for keyboard activity or load average every POLLING_INTERVAL , you must enable D_FULLDEBUG.

8.5.14 Version 6.1.4

- Fixed a bug in the socket communication library used by Condor that was causing daemons and tools to die on some platforms (notably, Digital Unix) with signal 8, SIGFPE (floating point exception).
- Fixed a bug in the usage message of many Condor tools that mentioned a **-all** option that isn't yet supported. This option will be supported in future versions of Condor.
- Fixed a bug in the filesystem authentication code used to authenticate operations on the job queue that left empty temporary files in /tmp. These files are now properly removed after they are used.
- Fixed a minor bug in the totals *condor_status* displays when you use the **ckptsrvr** option.
- Fixed a minor syntax error in the *condor_install* script that would cause warnings.
- the Condor.pm Perl module is now included in the lib directory of the main release directory.

8.5.15 Version 6.1.3

NOTE: There are a lot of new, unstable features in 6.1.3. PLEASE do not install all of 6.1.3 on a production pool. Almost all of the bug fixes in 6.1.3 are in the *condor_startd* or *condor_starter*, so, unless you really know what you're doing, we recommend you just upgrade SMP-Startd contrib module, not the entire 6.1.3 release.

- Owners can now specify how the SMP-Startd partitions the system resources into the different types and numbers of virtual machines, specifying the number of CPUs, megs of RAM, megs of swap space, etc., in each. Previously, each virtual machine reported to Condor from an SMP machine always had one CPU, and all shared system resources were evenly divided among the virtual machines.
- Fixed a bug in the reporting of virtual memory and disk space on SMP machines where each virtual machine represented was advertising the total in the system for itself, instead of its own share. Now, both the totals, and the virtual machine-specific values are advertised.
- Fixed a bug in the *condor_starter* when it was trying to suspend jobs. While we always killed all of the processes when we were trying to vacate, if a vanilla job forked, the starter would sometimes not suspend some of the children processes. In addition, we could sometimes miss a standard universe job for suspending as well. This is all fixed.

- Fixed a bug in the SMP-Startd's load average computation that could cause processes spawned by Condor to not be associated w/ the Condor load average. This would cause the startd to over-estimate the owner's load average, and under-estimate the Condor load, which would cause a cycle of suspending and resuming a Condor job, instead of just letting it run.
- Fixed a bug in the SMP-Startd's load average computation that could cause certain rare exceptions to be treated as fatal, when in fact, the Startd could recover from them.
- Fixed a bug in the computation of the total physical memory on some platforms that was resulting in an overflow on machines with lots of ram (over 1 gigabyte).
- Fixed some bugs that could cause *condor_starter* processes to be left as zombies underneath the *condor_startd* under very rare conditions.
- For sites using AFS, if there are problems in the *condor_startd* computing the AFS cell of the machine it's running on, the startd will exit with an error message at start-up time.
- Fixed a minor bug in *condor_install* that would lead to a syntax error in your config file given a certain set of installation options.
- Added the **-maxjobs** option to the *condor_submit_dag* script that can be used to specify the maximum number of jobs Condor will run from a DAG at any given time. Also, *condor_submit_dag* automatically creates a "rescue DAG". See section 2.12 on page 52 for details on DAGMan.
- Fixed bug in ClassAd printing when you tried to display an integer or float attribute that didn't exist in the given ClassAd. This could show up in *condor_status*, *condor_q*, *condor_history*, etc.
- Various commands sent to the Condor daemons now have separate debug levels associated with them. For example, commands such as "keep-alives", and the command sent from the *condor_kbdd* to the *condor_startd* are only seen in the various log files if `D_FULLDEBUG` is turned on, instead of `D_COMMAND`, which the default and now enabled for all daemons on all platforms by default. Administrators retaining their old configuration when upgrading to this version are encouraged to enable `D_COMMAND` in the `SCHEDD_DEBUG` setting. In addition, for IRIX and Digital Unix machines, it should be enabled in the `STARTD_DEBUG` setting as well. See section 3.3.3 on page 89 for details on debug levels in Condor.
- New debug levels added to Condor:
  - `D_NETWORK`, used by various daemons in Condor to report various network statistics about the Condor daemons.
  - `D_PROCFAMILY`, used to report information about various families of processes that are monitored by Condor. For example, this is used in the *condor_startd* when monitoring the family of processes spawned by a given user job for the purposes of computing the Condor load average.
  - `D_KEYBOARD`, used by the *condor_startd* to print out statistics about remote tty and console idle times in the *condor_startd*. This information used to be logged at `D_FULLDEBUG`, along with everything else, so now, you can see just the idle times, and/or have the information stored to a separate file.

- Added a **-run** option to *condor_q*, which displays information for running jobs, including the remote host where each job is running.
- Macros can now be incrementally defined. See section 3.3.1 on page 82 for more details.
- *condor_config_val* can now be used to set configuration variables. See the man page on page 331 for more details.
- The job log file now contains a record of network activity. The evict, terminate, and shadow exception events indicate the number of bytes sent and received by the job for the specific run. The terminate event additionally indicates totals for the life of the job.
- `STARTER_CHOOSES_CKPT_SERVER` now defaults to true. See section 3.3.6 on page 96 for more details.
- The infrastructure for authentication within Condor has been overhauled, allowing for much greater flexibility in supporting new forms of authentication in the future. This means that the 6.1.3 schedd and queue management tools (like *condor_q*, *condor_submit*, *condor_rm* and so on) are incompatible with previous versions of Condor.
- Many of the Condor administration tools have been improved to allow you to specify the "subsystem" you want them to effect. For example, you can now use "condor_reconfig -startd" to just have the startd reconfigure itself. Similarly, condor_off, condor_on and condor_restart can now all work on a single daemon, instead of machine-wide. See the man pages (section 9 on page 322) or run any command with **-help** for details. NOTE: The usage message in 6.1.3 incorrectly reports **-all** as a valid option.
- Fixed a bug in the Condor tools that could cause a segmentation violation in certain rare error conditions.

8.5.16 Version 6.1.2

- Fixed some bugs in the *condor_install* script. Also, enhanced *condor_install* to customize the path to perl in various perl scripts used by Condor.
- Fixed a problem with our build environment that left some files out of the `release.tar` files in the binary releases on some platforms.
- *condor_dagman*, "DAGMan" (see section 2.12 on page 52 for details) is now included in the development release by default.
- Fixed a bug in the computation of the total physical memory in HPUX machines that was resulting in an overflow on machines with lots of ram (over 1 gigabyte). Also, if you define "MEMORY" in your config file, that value will override whatever value Condor computes for your machine.
- Fixed a bug in *condor_starter.pvm*, the PVM version of the Condor starter (available as an optional "Contrib module"), when you disabled `STARTER_LOCAL_LOGGING` . Now, having this set to "False" will properly place debug messages from *condor_starter.pvm* into the

ShadowLog file of the machine that submitted the job (as opposed to the StarterLog file on the machine executing the job).

8.5.17 Version 6.1.1

- Fixed a bug in the *condor_startd* where we compute the load average caused by Condor that was causing us to get the wrong values. This could cause a cycle of continuous job suspends and job resumes.
- Beginning with this version, any jobs linked with the Condor checkpoint libraries will use the zlib compression code (used by gzip and others) to compress periodic checkpoints before they are written to the network. These compressed checkpoints are uncompressed at startup time. This saves network bandwidth, disk space, as well as time (if the network is the bottleneck to checkpointing, which it usually is). In future versions of Condor, all checkpoints will probably be compressed, but at this time, it is only used for periodic checkpoints. Note, you have to relink your jobs with the *condor_compile* command to have this feature enabled. Old jobs (not relinked) will continue to run just fine, they just won't be compressed.
- *condor_status* now has better support for displaying checkpoint server ClassAds.
- More contrib modules from the development series are now available, such as the checkpoint server, PVM support, and the CondorView server.
- Fixed some minor bugs in the UserLog code that were causing problems for DAGMan in exceptional error cases.
- Fixed an obscure bug in the logging code when D_PRIV was enabled that could result in incorrect file permissions on log files.

8.5.18 Version 6.1.0

- Support has been added to the condor_startd to run multiple jobs on SMP machines. See section 3.10.8 on page 192 for details about setting up and configuring SMP support.
- The expressions that control the condor_startd policy for vacating, jobs has been simplified. See section 3.6 on page 127 for complete details on the new policy expressions, and section 3.6.10 on page 149 for an explanation of what's different from the version 6.0 expressions.
- We now perform better tracking of processes spawned by Condor. If children die and are inherited by init, we still know they belong to Condor. This allows us to better ensure we don't leave processes lying around when we need to get off a machine, and enables us to have a much more accurate computation of the load average generated by Condor (the CondorLoadAvg as reported by the *condor_startd*).
- The condor_collector now can store historical information about your pool state. This information can be queried with the *condor_stats* program (see the man page on page 386), which is used by the *condor_view* Java GUI, which is available as a separate contrib module.

- Condor jobs can now be put in a "hold" state with the *condor_hold* command. Such jobs remain in the job queue (and can be viewed with *condor_q*), but there will not be any negotiation to find machines for them. If a job is having a temporary problem (like the permissions are wrong on files it needs to access), the job can be put on hold until the problem can be solved. Jobs put on hold can be released with the *condor_release* command.
- condor_userprio now has the notion of *user factors* as a way to create different groups of users in different priority levels. See section 3.5 on page 124 for details. This includes the ability to specify a local priority domain, and all users from other domains get a much worse priority.
- Usage statistics by user is now available from condor_userprio. See the man page on page 418 for details.
- The condor_schedd has been enhanced to enable "flocking", where it seeks matches with machines in multiple pools if its requests cannot be serviced in the local pool. See section 3.10.7 on page 192 for more details.
- The condor_schedd has been enhanced to enable condor_q and other interactive tools better response time.
- The condor_schedd has also been enhanced to allow it to check the permissions of the files you specify for input, output, error and so on. If the schedd doesn't have the required access rights to the files, the jobs will not be submitted, and *condor_submit* will print an error message.
- When you perform a *condor_rm* command, and the job you removed was using a "user log", the remove event is now recorded into the log.
- Two new attributes have been added to the job classad when it begins executing: `RemoteHost` and `LastRemoteHost`. These attributes list the IP address and port of the startd that is either currently running the job, or the last startd to run the job (if it's run on more than one machine). This information helps users track their job's execution more closely, and allows administrators to troubleshoot problems more effectively.
- The performance of checkpointing was increased by using larger buffers for the network I/O used to get the checkpoint file on and off the remote executing host (this helps for all pools, with or without checkpoint servers).

8.6 Stable Release Series 6.0

6.0 is the first version of Condor with *ClassAds*. It contains many other fundamental enhancements over version 5. It is also the first official stable release series, with a development series (6.1) simultaneously available.

8.6.1 Version 6.0.3

- Fixed a bug that was causing the hostname of the submit machine that claimed a given execute machine to be incorrectly reported by the *condor_startd* at sites using NIS.

- Fixed a bug in the *condor_startd*'s benchmarking code that could cause a floating point exception (SIGFPE, signal 8) on very, very fast machines, such as newer Alphas.
- Fixed an obscure bug in *condor_submit* that could happen when you set a requirements expression that references the "Memory" attribute. The bug only showed up with certain formations of the requirement expression.

8.6.2 Version 6.0.2

- Fixed a bug in the `fcntl()` call for Solaris 2.6 that was causing problems with file I/O inside Fortran jobs.
- Fixed a bug in the way the `DEFAULT_DOMAIN_NAME` parameter was handled so that this feature now works properly.
- Fixed a bug in how the `SOFT_UID_DOMAIN` config file parameter was used in the *condor_starter*. This feature is also documented in the manual now (see section 3.3.5 on page 94).
- You can now set the RunBenchmarks expression to "False" and the *condor_startd* will never run benchmarks, not even at startup time.
- Fixed a bug in `getwd()` and `getcwd()` for sites that use the NFS automounter. his bug was only present if user programs tried to call `chdir()` themselves. Now, this is supported.
- Fixed a bug in the way we were computing the available virtual memory on HPUX 10.20 machines.
- Fixed a bug in *condor_q* -analyze so it will correctly identify more situations where a job won't run.
- Fixed a bug in *condor_status* -format so that if the requested attribute isn't available for a given machine, the format string (including spaces, tabs, newlines, etc) is still printed, just the value for the requested attribute will be an empty string.
- Fixed a bug in the *condor_schedd* that was causing *condor_history* to not print out the first ClassAd attribute of all jobs that have completed
- Fixed a bug in *condor_q* that would cause a segmentation fault if the argument list was too long.

8.6.3 Version 6.0.1

- Fixed bugs in the `getuid()`), `getgid()`, `geteuid()`, and `getegid()` system calls.
- Multiple config files are now supported as a list specified via the `LOCAL_CONFIG_FILE` variable.
- `ARCH` and `OPSYS` are now automatically determined on all machines (including HPUX 10 and Solaris).

- Machines running IRIX now correctly suspend vanilla jobs.
- *condor_submit* doesn't allow root to submit jobs.
- The *condor_startd* now notices if you have changed `COLLECTOR_HOST` on reconfig.
- Physical memory is now correctly reported on Digital Unix when daemons are not running as root.
- New `$(SUBSYSTEM)` macro in configuration files that changes based on which daemon is reading the file (i.e. STARTD, SCHEDD, etc.) See section 3.3.1, "Condor Subsystem Names" on page 84 for a complete list of the subsystem names used in Condor.
- Port to HP-UX 10.20.
- `getrusage()` is now a supported system call. This system call will allow you to get resource usage about the entire history of your condor job.
- Condor is now fully supported on Solaris 2.6 machines (both Sparc and Intel).
- Condor now works on Linux machines with the GNU C library. This includes machines running RedHat 5.x and Debian 2.0. In addition, there seems to be a bug in RedHat that was causing the output from *condor_config_val* to not appear when used in scripts (like *condor_compile*). We put in explicit calls to flush the I/O buffers before *condor_config_val* exits, which seems to solve the problem.
- Hooks have been added to the checkpointing library to help support the checkpointing of PVM jobs.
- Condor jobs can now send signals to themselves when running in the standard universe. You do this just as you normally would:

  ```
  kill( getpid(), signal_number )
  ```

  Trying to send a signal to any other process will result in `kill()` returning -1.
- Support for NIS has been improved on Digital Unix and IRIX.
- Fixed a bug that would cause the negotiator on IRIX machines to never match jobs with available machines.

8.6.4 Version 6.0 pl4

NOTE: Back in the bad old days, we used this evil "patch level" version number scheme, with versions like "6.0pl4". This has all gone away in the current versions of Condor.

- Fixed a bug that could cause a segmentation violation in the *condor_schedd* under rare conditions when a *condor_shadow* exited.

- Fixed a bug that was preventing any core files that user jobs submitted to Condor might create from being transferred back to the submit machine for inspection by the user who submitted them.
- Fixed a bug that would cause some Condor daemons to go into an infinite loop if the "ps" command output duplicate entries. This only happens on certain platforms, and even then, only under rare conditions. However, the bug has been fixed and Condor now handles this case properly.
- Fixed a bug in the *condor_shadow* that would cause a segmentation violation if there was a problem writing to the user log file specified by "log = filename" in the submit file used with *condor_submit*.
- Added new command line arguments for the Condor daemons to support saving the PID (process id) of the given daemon to a file, sending a signal to the PID specified in a given file, and overriding what directory is used for logging for a given daemon. These are primarily for use with the *condor_kbdd* when it needs to be started by XDM for the user logged onto the console, instead of running as root. See section 3.10.4 on "Installing the *condor_kbdd*" on page 184 for details.
- Added support for the `CREATE_CORE_FILES` config file parameter. If this setting is defined, Condor will override whatever limits you have set and in the case of a fatal error, will either create core files or not depending on the value you specify ("true" or "false").
- Most Condor tools (*condor_on*, *condor_off*, *condor_master_off*, *condor_restart*, *condor_vacate*, *condor_checkpoint*, *condor_reconfig*, *condor_reconfig_schedd*, *condor_reschedule*) can now take the IP address and port you want to send the command to directly on the command line, instead of only accepting hostnames. This IP/port must be passed in a special format used in Condor (which you will see in the daemon's log files, etc). It is of the form: <ip.address:port>. For example: <123.456.789.123:4567>.

8.6.5 Version 6.0 pl3

- Fixed a bug that would cause a segmentation violation if a machine was not configured with a full hostname as either the official hostname or as any of the hostname aliases.
- If your host information does not include a fully qualified hostname anywhere, you can specify a domain in the `DEFAULT_DOMAIN_NAME` parameter in your global config file which will be appended to your hostname whenever Condor needs to use a fully qualified name.
- All Condor daemons and most tools now support a "-version" option that displays the version information and exits.
- The *condor_install* script now prompts for a short description of your pool, which it stores in your central manager's local config file as `COLLECTOR_NAME` . This description is used to display the name of your pool when sending information to the Condor developers.

- When the *condor_shadow* process starts up, if it is configured to use a checkpoint server and it cannot connect to the server, the shadow will check the `MAX_DISCARDED_RUN_TIME` parameter. If the job in question has accumulated more CPU minutes than this parameter, the *condor_shadow* will keep trying to connect to the checkpoint server until it is successful. Otherwise, the *condor_shadow* will just start the job over from scratch immediately.
- If Condor is configured to use a checkpoint server, it will only use the checkpoint server. Previously, if there was a problem connecting to the checkpoint server, Condor would fall back to using the submit machine to store checkpoints. However, this caused problems with local disks filling up on machines without much disk space.
- Fixed a rare race condition that could cause a segmentation violation if a Condor daemon or tool opened a socket to a daemon and then closed it right away.
- All TCP sockets in Condor now have the "keep alive" socket option enabled. This allows Condor daemons to notice if their peer goes away in a hard crash.
- Fixed a bug that could cause the *condor_schedd* to kill jobs without a checkpoint during its graceful shutdown method under certain conditions.
- The *condor_schedd* now supports the `MAX_SHADOW_EXCEPTIONS` parameter. If the *condor_shadow* processes for a given match die due to a fatal error (an exception) more than this number of times, the *condor_schedd* will now relinquish that match and stop trying to spawn *condor_shadow* processes for it.
- The "-master" option to *condor_status* now displays the `Name` attribute of all *condor_master* daemons in your pool, as opposed to the `Machine` attribute. This helps for pools that have submit-only machines joining them, for example.

8.6.6 Version 6.0 pl2

- In patch level 1, code was added to more accurately find the full hostname of the local machine. Part of this code relied on the resolver, which on many platforms is a dynamic library. On Solaris, this library has needed many security patches and the installation of Solaris on our development machines produced binaries that are incompatible with sites that haven't applied all the security patches. So, the code in Condor that relies on this library was simply removed for Solaris.
- Version information is now built into Condor. You can see the `CondorVersion` attribute in every daemon's ClassAd. You can also run the UNIX command "ident" on any Condor binary to see the version.
- Fixed a bug in the "remote submit" mode of *condor_submit*. The remote submit wasn't connecting to the specified schedd, but was instead trying to connect to the local schedd.
- Fixed a bug in the *condor_schedd* that could cause it to exit with an error due to its log file being locked improperly under certain rare circumstances.

8.6.7 Version 6.0 pl1

- *condor_kbdd* bug patched: On Silicon Graphics and DEC Alpha ports, if your X11 server is using Xauthority user authentication, and the *condor_kbdd* was unable to read the user's `.Xauthority` file for some reason, the *condor_kbdd* would fall into an infinite loop.
- When using a Condor Checkpoint Server, the protocol between the Checkpoint Server and the *condor_schedd* has been made more robust for a faulty network connection. Specifically, this improves reliability when submitting jobs across the Internet and using a remote Checkpoint Server.
- Fixed a bug concerning `MAX_JOBS_RUNNING` : The parameter `MAX_JOBS_RUNNING` in the config file controls the maximum number of simultaneous *condor_shadow* processes allowed on your submission machine. The bug was the number of shadow processes could, under certain conditions, exceed the number specified by `MAX_JOBS_RUNNING` .
- Added new parameter `JOB_RENICE_INCREMENT` that can be specified in the config file. This parameter specifies the UNIX nice level that the *condor_starter* will start the user job. It works just like the *renice*(1) command in UNIX. Can be any integer between 1 and 19; a value of 19 is the lowest possible priority.
- Improved response time for *condor_userprio*.
- Fixed a bug that caused periodic checkpoints to happen more often than specified.
- Fixed some bugs in the installation procedure for certain environments that weren't handled properly, and made the documentation for the installation procedure more clear.
- Fixed a bug on IRIX that could allow vanilla jobs to be started as root under certain conditions. This was caused by the non-standard uid that user "nobody" has on IRIX. Thanks to Chris Lindsey at NCSA for help discovering this bug.
- On machines where the `/etc/hosts` file is misconfigured to list just the hostname first, then the full hostname as an alias, Condor now correctly finds the full hostname anyway.
- The local config file and local root config file are now only found by the files listed in the `LOCAL_CONFIG_FILE` and `LOCAL_ROOT_CONFIG_FILE` parameters in the global config files. Previously, `/etc/condor` and user condor's home directory (c̃ondor) were searched as well. This could cause problems with submit-only installations of Condor at a site that already had Condor installed.

8.6.8 Version 6.0 pl0

- Initial Version 6.0 release.

CHAPTER

NINE

Command Reference Manual (man pages)

condor_advertise

Send a classad to the collector daemon

Synopsis

condor_advertise [**-help** | **-version**] [**-pool** *centralmanagerhostname*] [**-debug**] *update-command classad-filename*

Description

condor_advertise sends a classad to the collector daemon on the central manager machine. The classad is contained in a file, which is specified by the second required argument. Which daemon's classad to update is specified by the first required argument. The *update-command* may be one of the following strings:

UPDATE_STARTD_AD

UPDATE_SCHEDD_AD

UPDATE_MASTER_AD

UPDATE_GATEWAY_AD

UPDATE_CKPT_SRVR_AD

UPDATE_SUBMITTOR_AD

UPDATE_COLLECTOR_AD

UPDATE_LICENSE_AD

UPDATE_STORAGE_AD

Options

-help Display usage information

-version Display version information

-pool ***centralmanagerhostname*** Specify a pool by giving the central manager's hostname. The default is the `COLLECTOR_HOST` specified in the configuration file.

-debug Print debugging information as the command executes.

General Remarks

The job and machine classads are regularly updated. Therefore, the result of *condor_advertise* is likely to be overwritten in a very short time. It is unlikely that either Condor users (those who submit jobs) or administrators will ever have a use for this command. If it is desired to update or set a classad attribute, the *condor_config_val* command is the proper command to use.

Exit Status

condor_advertise will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_checkpoint

send a checkpoint command to jobs running on specified hosts

Synopsis

condor_checkpoint [**-help** | **-version**]

condor_checkpoint [**-name hostname** | **hostname** | **-addr "<a.b.c.d:port>"** | **"<a.b.c.d:port>"**] . . .

condor_checkpoint **-pool centralmanagerhostname**
-name hostname | **hostname** | **-addr "<a.b.c.d:port>"** | **"<a.b.c.d:port>"** . . .

Description

condor_checkpoint sends a checkpoint command to a set of machines within a single pool. This causes the startd daemon on each of the specified machines to take a checkpoint of any running job that is executing under the standard universe. The job is temporarily stopped, a checkpoint is taken, and then the job continues. If no machine is specified, then the command is sent to the machine that issued the *condor_checkpoint* command.

The command sent is a periodic checkpoint. The job will take a checkpoint, but then the job will immediately continue running after the checkpoint is completed. *condor_vacate*, on the other hand, will result in the job exiting (vacating) after it produces a checkpoint.

If the job being checkpointed is running under the standard universe, the job produces a checkpoint and then continues running on the same machine. If the job is running under another universe, or if there is currently no Condor job running on that host, then *condor_checkpoint* has no effect.

There is generally no need for the user or administrator to explicitly run *condor_checkpoint*. Taking checkpoints of running Condor jobs is handled automatically following the policies stated in the configuration files.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>" Send the command to a machine located at **"<a.b.c.d:port>"**

Exit Status

condor_checkpoint will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To send a *condor_checkpoint* command to two named machines:

```
% condor_checkpoint  robin cardinal
```

To send the *condor_checkpoint* command to a machine within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command sends the command to a the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_checkpoint -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized

without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_compile

create a relinked executable for submission to the Standard Universe

Synopsis

condor_compile *cc* | *CC* | *gcc* | *f77* | *g++* | *ld* | *make* | ...

Description

Use *condor_compile* to relink a program with the Condor libraries for submission into Condor's Standard Universe. The Condor libraries provide the program with additional support, such as the capability to checkpoint, which is required in Condor's Standard Universe mode of operation. *condor_compile* requires access to the source or object code of the program to be submitted; if source or object code for the program is not available (i.e. only an executable binary, or if it is a shell script), then the program must submitted into Condor's Vanilla Universe. See the reference page for *condor_submit* and/or consult the "Condor Users and Administrators Manual" for further information.

To use *condor_compile*, simply enter "condor_compile" followed by whatever you would normally enter to compile or link your application. Any resulting executables will have the Condor libraries linked in. For example:

```
condor_compile cc -O -o myprogram.condor file1.c file2.c ...
```

will produce a binary "myprogram.condor" which is relinked for Condor, capable of checkpoint/migration/remote-system-calls, and ready to submit to the Standard Universe.

If the Condor administrator has opted to fully install *condor_compile*, then *condor_compile* can be followed by practically any command or program, including make or shell-script programs. For example, the following would all work:

```
condor_compile make

condor_compile make install

condor_compile f77 -O mysolver.f

condor_compile /bin/csh compile-me-shellscript
```

If the Condor administrator has opted to only do a partial install of *condor_compile*, the you are restricted to following *condor_compile* with one of these programs:

```
        cc (the system C compiler)

        acc (ANSI C compiler, on Sun systems)

        c89 (POSIX compliant C compiler, on some systems)

        CC (the system C++ compiler)

        f77 (the system FORTRAN compiler)

        gcc (the GNU C compiler)

        g++ (the GNU C++ compiler)

        g77 (the GNU FORTRAN compiler)

        ld (the system linker)
f90 (the system FORTRAN 90 compiler)
```

NOTE: If you use explicitly call "ld" when you normally create your binary, simply use:

```
        condor_compile ld <ld arguments and options>
```

instead.

NOTE: f90 (FORTRAN 90) is only supported on Solaris and Digital Unix.

Exit Status

condor_compile is a script that executes specified compilers and/or linkers. If an error is encountered before calling these other programs, *condor_compile* will exit with a status value of 1 (one). Otherwise, the exit status will be that given by the executed program.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized

without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_config_val

Query or set a given Condor configuration variable

Synopsis

condor_config_val [*options*] **variable** [**variable** . . .]

condor_config_val [*options*] **-set** *string* [**string** . . .]

condor_config_val [*options*] **-rset** *string* [**string** . . .]

condor_config_val [*options*] **-unset** *variable* [**variable** . . .]

condor_config_val [*options*] **-runset** *variable* [**variable** . . .]

condor_config_val [*options*] **-tilde**

condor_config_val [*options*] **-owner**

Description

condor_config_val can be used to quickly see what the current condor configuration is on any given machine. Given a list of variables, *condor_config_val* will report what each of these variables is currently set to. If a given variable is not defined, *condor_config_val* will halt on that variable, and report that it is not defined. By default, *condor_config_val* looks in the local machine's configuration files in order to evaluate the variables.

condor_config_val can also be used to quickly set configuration variables for a specific daemon on a given machine. Each daemon remembers settings made by *condor_config_val*. The configuration file is not modified by this command. Persistent settings remain when the daemon is restarted. Runtime settings are lost when the daemon is restarted. In general, modifying a host's configuration with *condor_config_val* requires the `CONFIG` access level, which is disabled on all hosts by default. See section **??** on page **??** for more details. Begining with Condor version 6.3.2, administrators have more fine-grained control over which access levels can modify which settings. See section **??** on page **??** for more details.

NOTE: The changes will not take effect until you perform a *condor_reconfig*.

NOTE: It is generally wise to test a new configuration on a single machine to ensure you have no syntax or other errors in the configuration before you reconfigure many machines. Having bad syntax or invalid configuration settings is a fatal error for Condor daemons, and they will exit. Far better to discover such a problem on a single machine than to cause all the Condor daemons in your pool to exit.

Options

-name ***daemon_name*** Query the specified daemon for its configuration.

-pool ***hostname*** Use the given central manager to find daemons.

-address <***ip:port***> Connect to the given ip/port.

-master | **-schedd** | **-startd** | **-collector** | **-negotiator** The daemon to query (if not specified, master is default).

-set ***string*** Set a persistent config file entry. The string must be a single argument, so you should enclose it in double quotes. The string must be of the form "variable = value".

-rset ***string*** Set a runtime config file entry. See the description for **-set** for details about the string to use.

-unset ***variable*** Unset a persistent config file variable.

-runset ***variable*** Unset a runtime config file variable.

-tilde Return the path to the Condor home directory.

-owner Return the owner of the *condor_config_val* process.

variable . . . The variables to query.

Exit Status

condor_config_val will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To request the schedd daemon on host perdita to give the value of the `MAX_JOBS_RUNNING` configuration variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

To request the schedd daemon on host perdita to set the value of the MAX_JOBS_RUNNING configuration variable to the value 10.

```
% condor_config_val -name perdita -schedd -
set "MAX_JOBS_RUNNING = 10"
Successfully set configuration "MAX_JOBS_RUNNING = 10" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
% condor_reconfig -schedd perdita
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects the change implemented:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
10
```

To set the configuration variable MAX_JOBS_RUNNING back to what it was before the command to set it to 10:

```
% condor_config_val -name perdita -schedd -
unset MAX_JOBS_RUNNING
Successfully unset configuration "MAX_JOBS_RUNNING" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
% condor_reconfig -schedd perdita
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects that variable has gone back to is value before initial set of the variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_findhost

find machine(s) in the pool that can be used with minimal impact on currently running Condor jobs and best meet any specified constraints

Synopsis

condor_findhost [**-help**] [**-m**] [**-n** *num*] [**-c** *c_expr*] [**-r** *r_expr*] [**-p** *centralmanagerhostname*]

Description

condor_findhost searches a Condor pool of machines for the best machine or machines that will have the minimum impact on running Condor jobs if the machine or machines are taken out of the pool. The search may be limited to the machine or machines that match a set of constraints and rank expression.

condor_findhost returns a fully-qualified domain name for each machine. The search is limited (constrained) to a specific set of machines using the *-c* option. The search can use the *-r* option for rank, the criterion used for selecting a machine or machines from the constrained list.

Options

-help Display usage information and exit

-m Only search for entire machines. Virtual machines within an entire machine are not considered.

-n *num* Find and list up to *num* machines that fulfill the specification. *num* is an integer greater than zero.

-c *c_expr* Constrain the search to only consider machines that result from the evaluation of *c_expr*. *c_expr* is a ClassAd expression.

-r *r_expr* *r_expr* is the rank expression evaluated to use as a basis for machine selection. *r_expr* is a ClassAd expression.

-p *centralmanagerhostname* Specify the pool to be searched by giving the central manager's host name. Without this option, the current pool is searched.

General Remarks

condor_findhost is used to locate a machine within a pool that can be taken out of the pool with the least disturbance of the pool.

An administrator should set preemption requirements for the Condor pool. The expression

```
(Interactive =?= TRUE )
```

will let *condor_findhost* know that it can claim a machine even if Condor would not normally preempt a job running on that machine.

Exit Status

The exit status of *condor_findhost* is zero on success. If not able to identify as many machines as requested, it returns one more than the number of machines identified. For example, if 8 machines are requested, and *condor_findhost* only locates 6, the exit status will be 7. If not able to locate any machines, or an error is encountered, *condor_findhost* will return the value 1.

Examples

To find and list four machines, preferring those with the highest mips (on Drystone benchmark) rating:

```
condor_findhost -n 4 -r "mips"
```

To find and list 24 machines, considering only those where the `kflops` attribute is not defined:

```
condor_findhost -n 24 -c "kflops=?=undefined"
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team,

Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_glidein

add a Globus resource to a Condor pool

Synopsis

condor_glidein [**-help**] [**-basedir** *basedir*] [**-archdir** *dir*] [**-localdir** *dir*] [**-setuponly**] [**-runonly**] [**-scheduler** *name*] [**-queue** *name*] [**-project** *name*] [**-memory** *MBytes*] [**-count** *CPUcount*] [**-vms** *VMcount*] [**-idletime** *minutes*] [**-runtime** *minutes*] [**-anybody**] [**-admin** *address*] [**-genconfig**] [**-useconfig** *filename*] {**-contactfile** *filename* } | **Globus contact string**

Description

condor_glidein allows the temporary addition of a Globus resource to a local Condor pool. The addition is accomplished by installing and executing some of the Condor daemons on the Globus resource. A condor_shadow.globus job appears in the queue of the local Condor pool for each glidein request. To remove the Globus resource from the local Condor pool, use *condor_rm* to remove the condor_shadow.globus job from the job queue.

You must have an X.509 certificate and access to the Globus resource to use *condor_glidein*. The Globus software must also be installed.

Globus is a software system that provides uniform access to different high-performance computing resources. When specifying a machine to use with Globus, you provide a **Globus contact string**. Often, the contact string can be just the hostname of the machine. Sometimes, a more complicated contact string is required. For example, if a machine has multiple schedulers (ways to run a job), the contact string may need to specify which to use. See the Globus home page, www.globus.org for more information about Globus.

condor_glidein works in two steps: set up and execution. During set up, a configuration file and the Condor daemons master, startd and starter are installed on the Globus resource. Binaries for the correct architecture are copied from a central server. To obtain access to the server, send e-mail to condor-admin@cs.wisc.edu with your X.509 certificate name. Globus software version 1.1.3 does not yet include the Globus program *gsincftp*, the Globus secure version of *ftp*. *condor_glidein* needs this program. Install *gsincftp*, obtained from http://www.globus.org/datagrid/deliverables/gsiftp-tools.html. Set up need only be done once per machine and version of Condor. The execution step starts the Condor daemons running through the resource's Globus interface.

By default, all files placed on the remote machine are placed in `$(HOME)/Condor_glidein`. Each use of *condor_glidein* will generate spool and log files on the Globus resource. These files should be occasionally removed.

Options

-help Display brief usage information and exit

-basedir ***basedir*** Specifies the base directory on the Globus resource used for placing files. The default file is `$(HOME)/Condor_glidein` on the Globus resource.

-archdir ***dir*** Specifies the directory on the Globus resource for placement of the executables. The default value for **-archdir** , given according to version information on the Globus resource, is *basedir*`/<condor-version>-<Globus canonicalsystemname>` An example of the directory (without the base directory on the Globus resource) for Condor version 6.1.13 running on a Sun Sparc machine with Solaris 2.6 is `6.1.13-sparc-sun-solaris-2.6`

-localdir ***dir*** Specifies the directory on the Globus resource in which to create log and execution subdirectories needed by Condor. If limited disk quota in the home or base directory on the Globus resource is a problem, set **-localdir** to a large temporary space, such as `/tmp` or `/scratch`

-contactfile ***filename*** Allows the use of a file of Globus contact strings, rather than the single Globus contact string given in the command line. For each of the contacts listed in the file, the Globus resource is added to the local Condor pool.

-setuponly Performs only the placement of files on the Globus resource. This option cannot be run simultaneously with **-runonly**

-runonly Starts execution of the Condor daemons on the Globus resource. If any of the files are missing, exits with an error code. This option cannot be run simultaneously with **-setuponly**

-scheduler ***name*** Selects the Globus job scheduler type. Defaults to *fork*. NOTE: Contact strings which already contain the scheduler type will *not* be overridden by this option.

-queue ***name*** The argument *name* is a string which specifies which job queue is to be used for submission on the Globus resource.

-project ***name*** The argument *name* is a string which specifies which project is to be used for submission on the Globus resource.

-memory ***MBytes*** The maximum memory size to request from the Globus resource (in megabytes).

-count *CPUcount* Number of CPUs to request, default is 1.

-vms *VMcount* For machines with multiple CPUs, the CPUs maybe divided up into virtual machines. *VMcount* is the number of virtual machines that results. By default, Condor divides multiple-CPU resources such that each CPU is a virtual machine, each with an equal share of RAM, disk, and swap space. This option configures the number of virtual machines, so that multi-threaded jobs can run in a virtual machine with multiple CPUs. For example, if 4 CPUs are requested and **-vms** is not specified, Condor will divide the request up into 4 virtual machines with 1 CPU each. However, if **-vms** *2* is specified, Condor will divide the request up into 2 virtual machines with 2 CPUs each, and if **-vms** *1* is specified, Condor will put all 4 CPUs into one virtual machine.

-idletime *minutes* How long the Condor daemons on the Globus resource can remain idle before the resource reverts back to its former state of not being part of the local Condor pool. If the value is 0 (zero), the resource will not revert back to its former state. In this case, the Condor daemons will run until the *runtime* time expires, or they are killed by the resource or with *condor_rm*. The default value is 20 minutes.

-runtime *minutes* How long the Condor daemons on the Globus resource will run before shutting themselves down. This option is useful for resources with enforced maximum run times. Setting *runtime* to be a few minutes shorter than the allowable limit gives the daemons time to perform a graceful shutdown.

-anybody Sets the Condor `START` expression to TRUE to allow any user job which meets the job's requirements to run on the Globus resource added to the local Condor pool. Without this option, only jobs owned by the user executing *condor_glidein* can execute on the Globus resource. WARNING: Using this option may violate the usage policies of many institutions.

-admin *address* Where to send e-mail with problems. The defaults is the login of the user running *condor_glidein* at UID domain of the local Condor pool.

-genconfig This option creates a local copy of the configuration file used on the Globus resource. The file is called `condor_config.glidein`.

-useconfig *filename* Install *filename* as the configuration file on the Globus resource instead of the default configuration file during the set up phase.

Exit Status

condor_glidein will exit with a status value of 0 (zero) upon complete success. The script exits with non-zero values upon failure. The status value will be 1 (one) if *condor_glidein* encountered an error making a directory, was unable to copy a tar file, encountered an error in parsing the command line, or was not able to gather required information. The status value will be 2 (two) if there was an error in the remote set up. The status value will be 3 (three) if there was an error in remote submission. The status value will be -1 (negative one) if no resource was specified in the command line.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_history

View log of condor jobs completed to date

Synopsis

condor_history [**-help**] [**-l**] [**-f** *filename*] [*-constraint expr* | *cluster* | *cluster.process* | *owner*]

Description

condor_history displays a summary of all condor jobs listed in the specified history files. If no history files are specified (with the **-f** option), the local history file as specified in Condor's configuration file (condor/spool/history by default) is read. The default listing summarizes each job on a single line, and contains the following items:

ID The cluster/process id of the condor job.

OWNER The owner of the job.

SUBMITTED The month, day, hour, and minute the job was submitted to the queue.

CPU_USAGE Remote user CPU time accumulated by the job to date in days, hours, minutes, and seconds.

ST Completion status of the job (C = completed and X = removed).

COMPLETED The time the job was completed.

PRI User specified priority of the job, ranges from -20 to +20, with higher numbers corresponding to greater priority.

SIZE The virtual image size of the executable in megabytes.

CMD The name of the executable.

If a job ID (in the form of cluster_id or cluster_id.proc_id) or an owner is provided, output will be restricted to jobs with the specified IDs and/or submitted by the specified owner. The -constraint option can be used to display jobs that satisfy a specified boolean expression.

Options

-help Get a brief description of the supported options

-f *filename* Use the specified file instead of the default history file

-constraint *expr* Display jobs that satisfy the expression

-l Display job ads in long format

Exit Status

condor_history will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_hold

put jobs in the queue into the hold state

Synopsis

condor_hold [**-help** | **-version**] [**-pool centralmanagerhostname**] [**-name hostname** | **-addr "<a.b.c.d:port>"**] *cluster...* | *cluster.process...* | *user...*

condor_hold [**-help** | **-version**] [**-pool centralmanagerhostname**] [**-name hostname** | **-addr "<a.b.c.d:port>"**] *-all*

Description

condor_hold places one or more jobs from the Condor job queue in hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be held are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can place the job on hold.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine located at "<a.b.c.d:port>"

cluster Hold all jobs in the specified cluster

cluster.process Hold the specific job in the cluster

user Hold jobs belonging to specified user

-all Hold all the jobs in the queue

-constraint constraint Hold jobs matching specified constraint

See Also

condor_release (on page 373)

General Remarks

To put a PVM universe job on hold, you must put each "process" in the PVM job cluster on hold. (In the PVM universe, each PVM job is assigned its own cluster number, and each machine class is assigned a "process" number in the job's cluster.) Putting a subset of the machine classes for a PVM job on hold is not supported.

Exit Status

condor_hold will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_master

The master Condor Daemon

Synopsis

condor_master

Description

This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send email to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

See section 3.1.2 in Admin Manual for more information about *condor_master* and other Condor daemons.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_master_off

Shutdown Condor and the *condor_master*

Synopsis

condor_master_off [**-help**] [**-version**] [**hostname ...**]

Description

condor_master_off no longer exists.

General Remarks

condor_master_off no longer exists as a Condor command. Instead, use

```
condor_off -master
```

to accomplish this task.

See Also

See the *condor_off* manual page.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and

Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_off

Shutdown condor daemons

Synopsis

condor_off [**-help** | **-version**]

condor_off [**-graceful** | **-fast**] [**-name hostname** | **hostname** | **-addr "**<**a.b.c.d:port**>**"** | **"**<**a.b.c.d:port**>**"**] . . .
[**-master** | **-startd** | **-schedd** | **-collector** | **-negotiator** | **-kbdd**]

condor_off [**-graceful** | **-fast**] **-pool centralmanagerhostname**
-name hostname | **hostname** | **-addr "**<**a.b.c.d:port**>**"** | **"**<**a.b.c.d:port**>**"** . . .
[**-master** | **-startd** | **-schedd** | **-collector** | **-negotiator** | **-kbdd**]

Description

condor_off shuts down a set of the Condor daemons running on a set of one or more machines. It does this cleanly without a loss of work done by any jobs currently running on this machine, or jobs that are running on other machines that have been submitted from this machine.

The command *condor_off* without any arguments will shut down all daemons except *condor_master*. The *condor_master* can then handle both local and remote requests to restart the other Condor daemons if need be. To restart Condor running on a machine, see the *condor_on* command.

The command *condor_off -master* will shut down all daemons including the *condor_master*. The command *condor_off -otherdaemon* will shut down only the specified daemon.

Options

-help Display usage information

-version Display version information

-graceful Gracefully shutdown daemons (the default)

-fast Quickly shutdown daemons

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>" Send the command to a machine located at **"<a.b.c.d:port>"**

-master Send the command to the master (default)

-startd Send the command to the startd daemon

-schedd Send the command to the schedd daemon

-collector Send the command to the collector daemon

-kbdd Send the command to the kbdd daemon

Exit Status

condor_off will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To shut down all daemons (other than *condor_master*) on the local host:

```
% condor_off
```

To shut down only the *condor_collector* on three named machines:

```
% condor_off  cinnamon cloves vanilla -collector
```

To shut down daemons within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command shuts down all daemons except the *condor_master* on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_off  -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_on

Start up Condor daemons

Synopsis

condor_on **[-help | -version]**

condor_on **[-name hostname | hostname | -addr "<a.b.c.d:port>" | "<a.b.c.d:port>"] ...**
[-master | -startd | -schedd | -collector | -negotiator | -kbdd]

condor_on **-pool centralmanagerhostname**
-name hostname | hostname | -addr "<a.b.c.d:port>" | "<a.b.c.d:port>" ...
[-master | -startd | -schedd | -collector | -negotiator | -kbdd]

Description

condor_on starts up a set of the Condor daemons on a set of machines. This command assumes that the *condor_master* is already running on the machine. If this is not the case, *condor_on* will fail complaining that it cannot find the address of the master. The command *condor_on* with no arguments will tell the *condor_master* to start up the Condor daemons specified in the configuration variable `DAEMON_LIST`. If a daemon is specified as an argument, *condor_on* starts up only that daemon.

This command cannot be used to start up the *condor_master* daemon.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>" Send the command to a machine located at **"<a.b.c.d:port>"**

-master Send the command to the master (default)

-startd Send the command to the startd daemon

-schedd Send the command to the schedd daemon

-collector Send the command to the collector daemon

-kbdd Send the command to the kbdd daemon

Exit Status

condor_on will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To begin running all daemons (other than *condor_master*) given in the configuration variable `DAEMON_LIST` on the local host:

```
% condor_on
```

To start up only the *condor_negotiator* on two named machines:

```
% condor_on  robin cardinal -negotiator
```

To start up only a daemon within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command starts up only the *condor_schedd* daemon on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_on -pool condor.cae.wisc.edu -name cae17 -schedd
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_preen

remove extraneous files from Condor directories

Synopsis

condor_preen [**-mail**] [**-remove**] [**-verbose**]

Description

condor_preen examines the directories belonging to Condor, and removes extraneous files and directories which may be left over from Condor processes which terminated abnormally either due to internal errors or a system crash. The directories checked are the LOG, EXECUTE, and SPOOL directories as defined in the Condor configuration files. *condor_preen* is intended to be run as user root (or user condor) periodically as a backup method to ensure reasonable file system cleanliness in the face of errors. This is done automatically by default by the *condor_master*. It may also be explicitly invoked on an as needed basis.

When *condor_preen* cleans the SPOOL directory, it always leaves behind the files specified in the `VALID_SPOOL_FILES` list in your config file. For the log directory, the only files removed or reported are those listed in the `INVALID_LOG_FILES` list. The reason for this difference is that, in general, you want to leave all files in the LOG directory alone, with a few exceptions (namely, core files). *condor_preen* still works if you supply a `VALID_LOG_FILES` list instead, but this usage is depricated. There are new log files for different things introduced all the time, and you wouldn't want to have to keep updating the list of files to leave alone in the LOG directory. For example, the SMP startd can spawn an arbitrary number of *condor_starter* processes, each with its own log file. On the other hand, there are only a small, fixed number of files in the SPOOL directory that the *condor_schedd* needs to keep around, so it is easier to specify the files you want to keep instead of the ones you want to get rid of.

Options

-mail Send mail to the `PREEN_ADMIN` as defined in the Condor configuration files instead of writing to the standard output

-remove Remove the offending files and directories rather than just reporting on them

-verbose List all files found in the Condor directories, even those which are not considered extraneous

Exit Status

condor_preen will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_prio

change priority of jobs in the condor queue

Synopsis

condor_prio [**-p** *priority*] [**+** | **-** *value*] [**-n** *schedd_name*] *cluster* | *cluster.process* | *username* | *-a*

Description

condor_prio changes the priority of one or more jobs in the condor queue. If a cluster_id and a process_id are both specified, *condor_prio* attempts to change the priority of the specified process. If a cluster_id is specified without a process_id, *condor_prio* attempts to change priority for all processes belonging to the specified cluster. If a username is specified, *condor_prio* attempts to change priority of all jobs belonging to that user. If the -a flag is set, *condor_prio* attempts to change priority of all jobs in the condor queue. The user must specify a priority adjustment or new priority. If the -p option is specified, the priority of the job(s) are set to the next argument. The user can also adjust the priority by supplying a + or - immediately followed by a digit. The priority of a job ranges from -20 to +20, with higher numbers corresponding to greater priority. Only the owner of a job or the super user can change the priority for it.

The priority changed by *condor_prio* is only compared to the priority of other jobs owned by the same user and submitted from the same machine. See the "Condor Users and Administrators Manual" for further details on Condor's priority scheme.

Options

-p *priority* Set priority to the specified value

+ | **-** ***value*** Change priority by the specified value

-n *schedd_name* Change priority of jobs queued at the specified schedd

-a Change priority of all the jobs in the queue

Exit Status

condor_prio will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_q

Display information about jobs in queue

Synopsis

condor_q [**-help**] [**-global**] [**-submitter** *submitter*] [**-name** *name*] [**-pool** *hostname*] [**-analyze**] [**-run**] [**-goodput**] [**-io**] [**-dag**] [**-long**] [**-format** *formatter attribute*] [**-cputime**] [**-currentrun**] [{*cluster* | *cluster.process* | *owner* | ***-constraint*** *expression* . . .}]

Description

condor_q displays information about jobs in the Condor job queue. By default, *condor_q* queries the local job queue but this behavior may be modified by specifying:

- the **-global** option, which queries all job queues in the pool
- a schedd name with the **-name** option, which causes the queue of the named schedd to be queried
- a submitter with the **-submitter** option, which causes all queues of the named submitter to be queried

To restrict the display to jobs of interest, a list of zero or more restrictions may be supplied. Each restriction may be one of:

- a *cluster* and a *process* matches jobs which belong to the specified cluster and have the specified process number
- a *cluster* without a *process* matches all jobs belonging to the specified cluster
- a *owner* matches all jobs owned by the specified owner
- a **-constraint** *expression* which matches all jobs that satisfy the specified ClassAd expression. (See section 4.1 for a discussion of ClassAd expressions.)

If no *owner* restrictions are present in the list, the job matches the restriction list if it matches at least one restriction in the list. If *owner* restrictions are present, the job matches the list if it matches one of the *owner* restrictions *and* at least one non-owner restriction.

If the **-long** option is specified, *condor_q* displays a long description of the queried jobs by printing the entire job classad. The attributes of the job classad may be displayed by means of the **-format** option, which displays attributes with a `printf(3)` format. (Multiple **-format** options may be specified in the option list to display several attributes of the job.) If neither **-long** or **-format** are specified, *condor_q* displays a a one line summary of information as follows:

ID The cluster/process id of the condor job.

OWNER The owner of the job.

SUBMITTED The month, day, hour, and minute the job was submitted to the queue.

RUN_TIME Wall-clock time accumulated by the job to date in days, hours, minutes, and seconds.

ST Current status of the job. U = unexpanded (never been run), H = on hold, R = running, I = idle (waiting for a machine to execute on), C = completed, and X = removed.

PRI User specified priority of the job, ranges from -20 to +20, with higher numbers corresponding to greater priority.

SIZE The virtual image size of the executable in megabytes.

CMD The name of the executable.

If the **-dag** option is specified, the OWNER column is replaced with NODENAME for jobs started by Condor DAGMan.

NOTE: The **-dag** option has no effect on a pre-v6.3.0 Condor queue, because older *condor_schedd* daemons don't pass the necessary DAG information to their jobs.

If the **-run** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

HOST(S) The host where the job is running. For PVM jobs, a host count is displayed instead.

If the **-goodput** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

GOODPUT The percentage of RUN_TIME for this job which has been saved in a checkpoint. A low GOODPUT value indicates that the job is failing to checkpoint. If a job has not yet attempted a checkpoint, this column contains `[?????]`.

CPU_UTIL The ratio of CPU_TIME to RUN_TIME for checkpointed work. A low CPU_UTIL indicates that the job is not running efficiently, perhaps because it is I/O bound or because the job requires more memory than available on the remote workstations. If the job has not (yet) checkpointed, this column contains `[??????]`.

Mb/s The network usage of this job, in Megabits per second of run-time.

If the **-io** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

READ The total number of bytes the application has read from files and sockets.

WRITE The total number of bytes the application has written to files and sockets.

SEEK The total number of seek operations the application has performed on files.

XPUT The effective throughput (average bytes read and written per second) from the application's point of view.

BUFSIZE The maximum number of bytes to be buffered per file.

BLOCKSIZE The desired block size for large data transfers.

These fields are updated when a job checkpoints or completes. If a job has not yet checkpointed, this information is not available.

If the **-cputime** option is specified, the RUN_TIME column is replaced with:

CPU_TIME The remote CPU time accumulated by the job to date (which has been stored in a checkpoint) in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is *not* shown. If the job has not checkpointed, this column contains 0+00:00:00.)

The *-analyze* option may be used to determine why certain jobs are not running by performing an analysis on a per machine basis for each machine in the pool. The reasons may vary among failed constraints, insufficient priority, resource owner preferences and prevention of preemption by the `PREEMPTION_REQUIREMENTS` expression. If the *-long* option is specified along with the *-analyze* option, the reason for failure is displayed on a per machine basis.

Options

-help Get a brief description of the supported options

-global Get queues of all the submitters in the system

-submitter *submitter* List jobs of specific submitter from all the queues in the pool

-pool *hostname* Use hostname as the central manager to locate schedds. (The default is the `COLLECTOR_HOST` specified in the configuration file.

-analyze Perform an approximate analysis to determine how many resources are available to run the requested jobs

-run Get information about running jobs.

-goodput Display job goodput statistics.

-io Display job input/output summaries.

-dag Display DAG jobs under their DAGMan.

-name *name* Show only the job queue of the named schedd

-long Display job ads in long format

-format *fmt attr* Display attribute *attr* in format *fmt*. Attributes must be from the job ClassAd.

-cputime I
nstead of wall-clock allocation time (RUN_TIME), display remote CPU time accumulated by the job to date in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is *not* shown.)

-currentrun N
ormally, RUN_TIME contains all the time accumulated during the current run plus all previous runs. If this option is specified, RUN_TIME only displays the time accumulated so far on this current run.

Restriction list The restriction list may have zero or more items, each of which may be:

cluster match all jobs belonging to cluster

cluster.proc match all jobs belonging to cluster with a process number of *proc*

-constraint *expression* match all jobs which match the ClassAd expression constraint

A job matches the restriction list if it matches any restriction in the list Additionally, if *owner* restrictions are supplied, the job matches the list only if it also matches an *owner* restriction.

General Remarks

Although *-analyze* provides a very good first approximation, the analyzer cannot diagnose all possible situations because the analysis is based on instantaneous and local information. Therefore, there are some situations (such as when several submitters are contending for resources, or if the pool is rapidly changing state) which cannot be accurately diagnosed.

-goodput, *-cputime*, and *-io* are most useful for STANDARD universe jobs, since they rely on values computed when a job checkpoints.

Examples

The **-format** option provides a way to specify both the job attributes and formatting of those attributes. The format specification is given as in *printf*. There must be only one conversion specification per **-format** option. As an example, to list only Jane Doe's jobs in the queue, choosing to print and format only the owner of the job, the command line arguments for the job, and the process ID of the job:

```
%condor_q -submitter jdoe -format "%s" Owner -format " %s " Args -
format "ProcId = %d\n" ProcId
jdoe 16386 2800 ProcId = 0
jdoe 16386 3000 ProcId = 1
jdoe 16386 3200 ProcId = 2
jdoe 16386 3400 ProcId = 3
jdoe 16386 3600 ProcId = 4
jdoe 16386 4200 ProcId = 7
```

If only the cluster and process ID of Jane Doe's job's are to be listed, the following example works well.

```
%condor_q -submitter jdoe -format "%d ." ClusterId -
format " %d\n" ProcId
27 . 0
27 . 1
27 . 2
27 . 3
27 . 4
27 . 7
```

Exit Status

condor_q will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized

without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_qedit

modify job attributes

Synopsis

condor_qedit [**-n** *schedd-name*] {*cluster* | *cluster.proc* | *owner* | *-constraint constraint*} *attribute-name attribute-value . . .*

Description

condor_qedit modifies job attributes in the Condor job queue. The jobs are specified either by cluster number, cluster.proc job ID, owner, or by a ClassAd constraint expression. The attribute-value may be any ClassAd expression (integer, floating point number, string, expression).

Options

-n *schedd-name* Modify job attributes in the queue of the specified schedd

Examples

```
% condor_qedit -name perdita 1849.0 In '"myinput"'
Set attribute "In".
% condor_qedit jbasney NiceUser TRUE
Set attribute "NiceUser".
% condor_qedit -constraint 'JobUniverse == 1' Require-
ments '(Arch == "INTEL") && (OpSys == "SOLARIS26") && (Disk >= Ex-
ecutableSize) && (VirtualMemory >= ImageSize)'
Set attribute "Requirements".
```

General Remarks

You can view the list of attributes with their current values for a job with *condor_q* **-long**.

Strings must be specified with quotes (for example, '"String"').

If a job is currently running, modified attributes for that job will not take effect until the job restarts.

condor_qedit will not allow modification of the following attributes to ensure security and correctness: `Owner`, `ClusterId`, `ProcId`, `MyType`, `TargetType`, and `JobStatus`.

Please use *condor_hold* to place a job "on hold" and *condor_release* to release a held job, instead of attempting to modify `JobStatus` directly.

Exit Status

condor_qedit will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_reconfig

Reconfigure Condor daemons

Synopsis

condor_reconfig [**-help** | **-version**]

condor_reconfig [**-name hostname** | **hostname** | **-addr "<a.b.c.d:port>"** | **"<a.b.c.d:port>"**]
...
[**-master** | **-startd** | **-schedd** | **-collector** | **-negotiator** | **-kbdd**] [**-full**]

condor_reconfig **-pool centralmanagerhostname**
-name hostname | **hostname** | **-addr "<a.b.c.d:port>"** | **"<a.b.c.d:port>"** ...
[**-master** | **-startd** | **-schedd** | **-collector** | **-negotiator** | **-kbdd**] [**-full**]

Description

condor_reconfig reconfigures all of the condor daemons in accordance with the current status of the Condor configuration file(s). Once reconfiguration is complete, the daemons will behave according to the policies stated in the configuration file(s). The main exception is with the DAEMON_LIST variable, which will only be updated if the *condor_restart* command is used. There are a few other configuration settings that can only be changed if the Condor daemons are restarted. Whenever this is the case, it will be mentioned in section 3.3 on page 82 which lists all of the settings used to configure Condor. In general, *condor_reconfig* should be used when making changes to the configuration files, since it is faster and more efficient than restarting the daemons.

The command *condor_reconfig* with no arguments or with the **-master** argument specifying a daemon will cause the reconfiguration of the *condor_master* daemon and all the child processes of the *condor_master*.

Options

-help Display usage information

-version Display version information

-full Perform a full reconfiguration. In addition to re-reading the configuration files, a full reconfiguration will clear cached DNS information in the daemons. Use this option only when the DNS information needs to be reinitialized.

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>" Send the command to a machine located at **"<a.b.c.d:port>"**

Exit Status

condor_reconfig will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To reconfigure the *condor_master* and all its children on the local host:

```
% condor_reconfig
```

To reconfigure only the *condor_startd* on a named machine:

```
% condor_reconfig -name bluejay -startd
```

To reconfigure a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reconfigures the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_reconfig -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_reconfig_schedd

Reconfigure condor schedd

Synopsis

condor_reconfig_schedd [**-help**] [**-version**] [**hostname ...**]

Description

condor_reconfig_schedd no longer exists.

General Remarks

condor_reconfig_schedd no longer exists as a Condor command. Instead, use

```
condor_reconfig -schedd
```

to accomplish this task.

See Also

See the *condor_reconfig* manual page.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and

Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_release

release held jobs in the Condor queue

Synopsis

condor_release [**-help** | **-version**] [**-pool centralmanagerhostname**] [**-name hostname** | **-addr "<a.b.c.d:port>"**] *cluster...* | *cluster.process...* | *user...*

condor_release [**-help** | **-version**] [**-pool centralmanagerhostname**] [**-name hostname** | **-addr "<a.b.c.d:port>"**] *-all*

Description

condor_release releases one or more jobs from the Condor job queue that were previously placed in hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be released are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can release the job.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine located at "<a.b.c.d:port>"

cluster Release all jobs in the specified cluster

cluster.process Release the specific job in the cluster

user Release jobs belonging to specified user

-all Release all the jobs in the queue

See Also

condor_hold (on page 344)

General Remarks

When releasing a held PVM universe job, you must release the entire job cluster. (In the PVM universe, each PVM job is assigned its own cluster number, and each machine class is assigned a "process" number in the job's cluster.) Releasing a subset of the machine classes for a PVM job is not supported.

Exit Status

condor_release will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_reschedule

Update scheduling information to the central manager

Synopsis

condor_reschedule [**-help** | **-version**]

condor_reschedule [**-name hostname** | **hostname** | **-addr "**<**a.b.c.d:port**>**"** | **"**<**a.b.c.d:port**>**"**] . . .

condor_reschedule **-pool centralmanagerhostname**
-name hostname | **hostname** | **-addr "**<**a.b.c.d:port**>**"** | **"**<**a.b.c.d:port**>**"** . . .

Description

condor_reschedule updates the information about a set of machines' resources and jobs to the central manager. This command is used to force an update before viewing the current status of a machine. Viewing the status of a machine is done with the *condor_status* command. *condor_reschedule* also starts a new negotiation cycle between resource owners and resource providers on the central managers, so that jobs can be matched with machines right away. This can be useful in situations where the time between negotiation cycles is somewhat long, and an administrator wants to see if a job in the queue will get matched without waiting for the next negotiation cycle.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>" Send the command to a machine located at **"<a.b.c.d:port>"**

Exit Status

condor_reschedule will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To update the information on three named machines:

```
% condor_reschedule robin cardinal bluejay
```

To reschedule on a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reschedules the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_reschedule -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_restart

Restart the a set of Condor daemons

Synopsis

condor_restart [**-help** | **-version**]

condor_restart [**-name hostname** | **hostname** | **-addr "**<**a.b.c.d:port**>**"** | **"**<**a.b.c.d:port**>**"**] . . .
[**-master** | **-startd** | **-schedd** | **-collector** | **-negotiator** | **-kbdd**]

condor_restart **-pool centralmanagerhostname**
-name hostname | **hostname** | **-addr "**<**a.b.c.d:port**>**"** | **"**<**a.b.c.d:port**>**"** . . .
[**-master** | **-startd** | **-schedd** | **-collector** | **-negotiator** | **-kbdd**]

Description

condor_restart restarts a set of Condor daemon(s) on a set of machines. The daemon(s) will be put into a consistent state, killed, and then started anew.

If, for example, the *condor_master* needs to be restarted again with a fresh state, this is the command that should be used to do so. If the DAEMON_LIST variable in the configuration file has been changed, this command is used to restart the *condor_master* in order to see this change. The *condor_reconfigure* command cannot be used in the case where the `DAEMON_LIST` expression changes.

The command *condor_restart* with no arguments or the command *condor_restart -master* will safely shut down all running jobs and all submitted jobs from the machine(s) being restarted, then shut down all the child daemons of the *condor_master*, and then restart the *condor_master*. This, in turn, will allow the *condor_master* to start up other daemons as specified in the `DAEMON_LIST` configuration file entry.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>" Send the command to a machine located at **"<a.b.c.d:port>"**

-master Send the command to the master (default)

-startd Send the command to the startd daemon

-schedd Send the command to the schedd daemon

-collector Send the command to the collector daemon

-kbdd Send the command to the kbdd daemon

Exit Status

condor_restart will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To restart the *condor_master* and all its children on the local host:

```
% condor_restart
```

To restart only the *condor_startd* on a named machine:

```
% condor_restart -name bluejay -startd
```

To restart a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command restarts the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_restart -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_rm

remove jobs from the Condor queue

Synopsis

condor_rm [**-help** | **-version**] [**-pool centralmanagerhostname**] [**-name hostname** | **-addr "<a.b.c.d:port>"**] *cluster...* | *cluster.process...* | *user...*

condor_rm [**-help** | **-version**] [**-pool centralmanagerhostname**] [**-name hostname** | **-addr "<a.b.c.d:port>"**] *-all*

Description

condor_rm removes one or more jobs from the Condor job queue. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be removed are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can remove the job.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine located at "<a.b.c.d:port>"

cluster Remove all jobs in the specified cluster

cluster.process Remove the specific job in the cluster

user Remove jobs belonging to specified user

-all Remove all the jobs in the queue

General Remarks

When removing a PVM universe job, you should always remove the entire job cluster. (In the PVM universe, each PVM job is assigned its own cluster number, and each machine class is assigned a "process" number in the job's cluster.) Removing a subset of the machine classes for a PVM job is not supported.

Exit Status

condor_rm will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_run

Submit a shell command-line as a Condor job.

Synopsis

condor_run *"shell-cmd"*

Description

condor_run is a simple front-end to the *condor_submit* command for submitting a shell command-line as a vanilla universe Condor job. The *condor_run* command waits for the Condor job to complete, writes the job's output to the terminal, and exits with the exit status of the Condor job. No output will appear until the job completes. The shell command-line should be enclosed in quotes so it is passed directly to *condor_run* without modification by the invoking shell.

condor_run will not read any input from the terminal while the job executes. If the shell command-line requires input, you must explicitly redirect the input from a file to the command, as illustrated in the example.

You can specify where *condor_run* should execute the shell command-line with three environment variables:

CONDOR_ARCH Specifies the architecture of the execution machine (from the "Arch" field in the output of *condor_status*).

CONDOR_OPSYS Specifies the operating system of the execution machine (from the "OpSys" field in the output of *condor_status*).

CONDOR_REQUIREMENTS Specifies any additional requirements for the Condor job (as described in manual page for *condor_submit* on page 395). It is recommended that `CONDOR_REQUIREMENTS` always be enclosed in parenthesis.

If one or more of these environment variables is specified, the job is submitted with:

```
requirements = $CONDOR_REQUIREMENTS && Arch == $CONDOR_ARCH && \
               OpSys == $CONDOR_OPSYS
```

Otherwise, the job receives the default requirements expression, which requests a machine of the same architecture and operating system of the machine on which *condor_run* is executed.

All environment variables set when *condor_run* is executed will be included in the environment of the Condor job.

condor_run will remove the Condor job from the Condor queue and delete its temporary files if it is killed before the Condor job finishes.

Examples

condor_run can be used to compile jobs on architectures and operating systems to which the user doesn't have login access. For example:

```
$ setenv CONDOR_ARCH "SGI"
$ setenv CONDOR_OPSYS "IRIX65"
$ condor_run "f77 -O -o myprog myprog.f"
$ condor_run "make"
$ condor_run "condor_compile cc -o myprog.condor myprog.c"
```

Since *condor_run* does not read input from the terminal, you must explicitly redirect input from a file to the shell command. For example:

```
$ condor_run "myprog < input.dat > output.dat"
```

Files

condor_run creates the following temporary files in the user's working directory (replacing "pid" with *condor_run*'s process id):

.condor_run.pid This is the shell script containing the shell command-line which is submitted to Condor.

.condor_submit.pid This is the submit file passed to *condor_submit*.

.condor_log.pid This is the Condor log file monitored by *condor_run* to determine when the job exits.

.condor_out.pid This file contains the output of the Condor job (before it is copied to the terminal).

.condor_error.pid This file contains any error messages for the Condor job (before they are copied to the terminal).

The script removes these files when the job completes. However, if the script fails, it is possible that these files will remain in the user's working directory and the Condor job will remain in the queue.

General Remarks

condor_run is intended for submitting simple shell command-lines to Condor. It does not provide the full functionality of *condor_submit*. We have attempted to make *condor_run* as robust as possible, but it is possible that it will not correctly handle some possible *condor_submit* errors or system failures.

condor_run jobs have the same restrictions as other vanilla universe jobs. Specifically, the current working directory of the job must be accessible on the machine where the job runs. This typically means that the job must be submitted from a network file system such as NFS or AFS. Also, since Condor does not manage AFS credentials, permissions must be set to allow unauthenticated processes to access any AFS directories used by the Condor job.

All processes on the command-line will be executed on the machine where Condor runs the job. Condor will not distribute multiple processes of a command-line pipe across multiple machines.

condor_run will use the shell specified in the `SHELL` environment variable, if one exists. Otherwise, it will use */bin/sh*(t)o execute the shell command-line.

By default, *condor_run* expects perl to be installed in `/usr/bin/perl`. If perl is installed in another path, you can ask your Condor administrator to edit the path in the *condor_run* script or explicitly call perl from the command line:

```
$ perl [path-to-condor]/bin/condor_run "shell-cmd"
```

Exit Status

condor_run exits with a status value of 0 (zero) upon complete success. The exit status of *condor_run* will be non-zero upon failure. The exit status in the case of a single error due to a system call will be the error number (`errno`) of the failed call.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_stats

Display historical information about the Condor pool

Synopsis

condor_stats [**-f** *filename*] [**-orgformat**] [**-pool** *hostname*] [*query-type*] [**time-range**]

Description

condor_stats is a tool that is used to display historic information about a Condor pool. Based on the type of information requested (by specifying it using the command line arguments), a query is sent to the collector, and the information received is displayed using the standard ouptut. If the -f option is used the information will ne written to a file instead of the standard output. The -pool option can be used to get information from other pools, instead of the local (default) pool. Condor pool. The *condor_status* tool can be used to query resource information (single or by platform), submitter and user information, and checkpoint server information. When a time range is not specified, the query retrieves information for the last day. Otherwise, information can be retrieved for other time ranges such as the last specified number of hours, last week, last month, or a specified date range.

The information is diplayed in columns separated by tabs. The first column always reresents the time, as a percentage of the range of the query (for example, a value of 50 in the first column indicates that the information on that line corresponds to a time in the middle of the query time range). If the -orgformat option is used, the time is displayed as number of seconds since the beginning of 1970. The information in the rest of the columns depends on the query type.

The possible query types and the information they provide:

- **Single resource query** requested using the -resourcequery option and provides information about a single machine. The information displayed includes the keyboard idle time (in seconds), the load average, and the machine state.
- **Single resource list** requested using the -resourcelist option and provides a list of all the machines for which the collector has historic information in the query's time range.
- **Resource group query** requested using the -resgroupquery option and provides information about a group of machines (based on operating system and architecture). The information displayed includes number of machines in unclaimed state, matched state, claimed state, preempting state, owner state, and total number of machines.
- **Resource group list** requested using the -resgrouplist option and provides a list of all the group names for which the collector has historic information in the query's time range.

- **Submitter query** requested using the -userquery option and provides information about a submitter (a user submitting from a specific machine). The information displayed includes the number of running jobs and the number of idle jobs.
- **Submitter list** requested using the -userlist option and provides a list of all the submitters for which the collector has historic information in the query's time range.
- **User query** requested using the -usergroupquery option and provides information about a user (for all jobs submitted by that user, regardless of the machine they were submitted from). The information displayed includes the number of running jobs and the number of idle jobs.
- **User list** requested using the -usergrouplist option and provides a list of all the users for which the collector has historic information in the query's time range.
- **Checkpoint server query** requested using the -ckptquery option and provides information about a checkpoint server. The information displayed includes the number of bytes received (in Mb), bytes sent (Mb), average receive bandwidth (in Kb/s), and average send bandwidth (Kb/s).
- **Checkpoint server list** requested using the -ckptlist option and provides a list of all the checkpoint servers for which the collector has historic information in the query's time range.

One of the above query types must be specified on the command line. Note that logging of pool history must be enabled in the collector, otherwise no information will be available and the query will not be responded to.

Options

-f *filename* Write the information to a file instead of the standard output.

-pool *hostname* Contact the specified central manager instead of the local one.

-orgformat Display the information in the same format it is actually stored.

-lastday Get information for the last day.

-lastweek Get information for the last week.

-lastmonth Get information for the last month.

-lasthours *n* Get information for the n last hours.

-from *m d y* Get information for the time since the specified date.

-to *m d y* Get information for the time up to the specified date, instead of up to now.

-resourcequery *name* Perform a single resource query for the specified resource.

-resourcelist Get the list of resources.

-resgroupquery *name* Perform a resource group query for the specified group.

-resgrouplist Get the list of groups.

-userquery *name* Perform a submitter query for the specified submitter.

-userlist Get the list of submitters.

-usergroupquery *name* Perform a user query for the specified user.

-usergrouplist Get the list of users.

-ckptquery *name* Perform a checkpoint server query for the specified checkpoint server.

-ckptlist Get the list of checkpoint servers.

Exit Status

condor_stats will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_status

Display status of the Condor pool

Synopsis

condor_status [*help options*] [*query options*] [*display options*] [*custom options*] [*hostname . . .*]

Description

condor_status is a versatile tool that may be used to monitor and query the Condor pool. The *condor_status* tool can be used to query resource information, submitter information, checkpoint server information, and daemon master information. The specific query sent and the resulting information display is controlled by the query options supplied. Queries and display formats can also be customized.

The options that may be supplied to *condor_status* belong to five groups:

- **Help options** provide information about the *condor_status* tool.
- **Query options** control the content and presentation of status information.
- **Display options** control the display of the queried information.
- **Custom options** allow the user to customize query and display information.
- **Host options** specify specific machines to be queried

At any time, only one *help option*, one *query option* and one *custom option* may be specified. Any number of *custom* and *host options* may be specified.

Options

-help (Help option) Display usage information

-diagnose (Help option) Print out query ad without performing query

-avail (Query option) Query *condor_startd* ads and identify resources which are available

-claimed (Query option) Query *condor_startd* ads and print information about claimed resources

-ckptsrvr (Query option) Query *condor_ckpt_server* ads and display checkpoint server attributes

-direct *hostname* (Query option) Go directly to the given hostname to get the ads to display

-java (Query option) Display only Java-capable resources.

-master (Query option) Query *condor_master* ads and display daemon master attributes

-pool *hostname* Query the specified central manager. (*condor_status* queries `COLLECTOR_HOST` by default)

-schedd (Query option) Query *condor_schedd* ads and display attributes

-server (Query option) Query *condor_startd* ads and display resource attributes

-startd (Query option) Query *condor_startd* ads

-state (Query option) Query *condor_startd* ads and display resource state information

-submitters (Query option) Query ads sent by submitters and display important submitter attributes

-verbose (Display option) Display entire classads. Implies that totals will not be displayed.

-long (Display option) Display entire classads (same as **-verbose**)

-total (Display option) Display totals only

-constraint *const* (Custom option) Add constraint expression

-format *fmt attr* (Custom option) Register display format and attribute name. The *fmt* string has the same format as `printf(3)`, and *attr* is the name of the attribute that should be displayed in the specified format.

General Remarks

- The information obtained from *condor_startds* and *condor_schedds* may sometimes appear to be inconsistent. This is normal since startds and schedds update the Condor manager at different rates, and since there is a delay as information propagates through the network and the system.
- Note that the `ActivityTime` in the `Idle` state is *not* the amount of time that the machine has been idle. See the section on *condor_startd* states in the *Administrator's Manual* for more information.
- When using *condor_status* on a pool with SMP machines, you can either provide the hostname, in which case you will get back information about all virtual machines that are represented on that host, or you can list specific virtual machines by name. See the examples below for details.
- If you specify hostnames, without domains, Condor will automatically try to resolve those hostnames into fully qualified hostnames for you. This also works when specifying specific nodes of an SMP machine. In this case, everything after the "@" sign is treated as a hostname and that is what is resolved.
- You can use the **-direct** option in conjunction with almost any other set of options. However, at this time, the only daemon that will allow direct queries for its ad(s) is the *condor_startd*. So, the only options currently not supported with **-direct** are **-schedd** and **-master**. Most other options use startd ads for their information, so they work seamlessly with **-direct**. The only other restriction on **-direct** is that you may only use 1 **-direct** option at a time. If you want to query information directly from multiple hosts, you must run *condor_status* multiple times.
- Unless you use the local hostname with **-direct**, *condor_status* will still have to contact a collector to find the address where the specified daemon is listening. So, using a **-pool** option in conjunction with **-direct** just tells *condor_status* which collector to query to find the address of the daemon you want. The information actually displayed will still be retrieved directly from the daemon you specified as the argument to **-direct**.

Examples

Example 1 To view information from all nodes of an SMP machine, just use the hostname. For example, if you had a 4-CPU machine, named "vulture.cs.wisc.edu", here's what you might see:

```
% condor_status vulture

Name          OpSys       Arch   State      Activity   Loa-
dAv Mem   ActvtyTime

vm1@vulture.c SOLARIS26   INTEL  Owner      Idle       0.020  128   0+00:57:13
vm2@vulture.c SOLARIS26   INTEL  Claimed    Busy       1.006  128   0+01:16:03
```

```
vm3@vulture.c SOLARIS26   INTEL  Claimed    Busy       0.978  128   0+03:32:53
vm4@vulture.c SOLARIS26   INTEL  Claimed    Busy       1.001  128   0+02:21:07

                     Machines Owner Claimed Un-
claimed Matched Preempting

     INTEL/SOLARIS26        4     0       4         0       0          0

               Total        4     0       4         0       0          0
```

Example 2 To view information from a specific nodes of an SMP machine, specify the node directly. You do this by providing the name of the *virtual machine*. This has the form `vm#@hostname`. For example:

```
% condor_status vm2@vulture

Name          OpSys       Arch   State      Activity   Loa-
dAv Mem   ActvtyTime

vm2@vulture.c SOLARIS26   INTEL  Claimed    Busy       1.006  128   0+01:16:03

                     Machines Owner Claimed Un-
claimed Matched Preempting

     INTEL/SOLARIS26        1     0       1         0       0          0

               Total        1     0       1         0       0          0
```

Exit Status

condor_status will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team,

Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_submit

Queue jobs for execution under Condor

Synopsis

condor_submit [**–**] [**-v**] [**-n** *schedd_name*] [**-r** *schedd_name*] [**-d**] [**-a** *command* . . .][*submit description file*]

Description

condor_submit is the program for submitting jobs for execution under Condor. *condor_submit* requires a submit description file which contains commands to direct the queuing of jobs. One submit description file may contain specifications for the queuing of many Condor jobs at once. All jobs queued by a single invocation of *condor_submit* must share the same executable, and are referred to as a job cluster. It is advantageous to submit multiple jobs as a single cluster because:

- Only one copy of the checkpoint file is needed to represent all jobs in a cluster until they begin execution.
- There is much less overhead involved for Condor to start the next job in a cluster than for Condor to start a new cluster. This can make a big difference when submitting lots of short jobs.

SUBMIT DESCRIPTION FILE COMMANDS

Each submit description file describes one cluster of jobs to be placed in the Condor execution pool. All jobs in a cluster must share the same executable, but they may have different input and output files, and different program arguments. The submit description file is the only command-line argument to *condor_submit*.

The submit description file must contain one *executable* command and at least one *queue* command. All of the other commands have default actions.

The commands which can appear in the submit description file are:

executable = <**name**> The name of the executable file for this job cluster. Only one executable command may be present in a description file. If submitting into the Standard Universe, which is the default, then the named executable must have been re-linked with the Condor libraries (such as via the *condor_compile* command). If submitting into the Vanilla Universe, then the named executable need not be re-linked and can be any process which can run in the background (shell scripts work fine as well). If submitting into the Java Universe, then the argument must be a compiled `.class` file.

input = <**pathname**> Condor assumes that its jobs are long-running, and that the user will not wait at the terminal for their completion. Because of this, the standard files which normally access the terminal, (`stdin`, `stdout`, and `stderr`), must refer to files. Thus, the filename specified with **input** should contain any keyboard input the program requires (that is, this file becomes `stdin`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, input is ignored for submission to an NT machine.

output = <**pathname**> The **output** filename will capture any information the program would normally write to the screen (that is, this file becomes `stdout`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, output is ignored for submission to an NT machine. More than one job should not use the same output file, since this will cause one job to overwrite the output of another.

error = <**pathname**> The **error** filename will capture any error messages the program would normally write to the screen (that is, this file becomes `stderr`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, error messages are ignored for submission to an NT machine. More than one job should not use the same error file, since this will cause one job to overwrite the errors of another.

arguments = <**argument_list**> List of arguments to be supplied to the program on the command line. In the Java Universe, the first argument must be the main of the class containing `main`.

initialdir = <**directory-path**> Used to specify the current working directory for the Condor job. Should be a path to a preexisting directory. If not specified, *condor_submit* will automatically insert the user's current working directory at the time *condor_submit* was run as the value for **initialdir**.

transfer_input_files = < **file1, file2, file...** > Lists all the files to be transferred into the working directory for the job before the job is started. Separate multiple filenames with a comma. By default, the file specified in the **executable** command and any file specified in the **input** command (for example, `stdin`) are transferred.

To ensure that Condor picks a machine with enough local disk space for the job, add to the **Requirements** expression in the submit description file such that it includes the `Disk` attribute. A **requirements** example that specifies only the `Disk` attribute:

```
        # The machine which runs the job must have more than
# 10 Mbytes of free disk space.  The Disk attribute
        # is expressed in kilobytes; 10 Mbytes is 10000 Kbytes.
        requirements = Disk > 10000
```

Only the transfer of files is available; the transfer of subdirectories is not supported.

transfer_output_files = < **file1, file2, file...** > This command forms an explicit list of output files to be transferred back from the temporary working directory on the execute machine to the submit machine. Most of the time, there is no need to use this command. If **transfer_output_files** is not specified, Condor will automatically transfer back all files in the job's temporary working directory which have been modified or created by the job. This is usually the desired behavior. Explicitly listing output files is typically only done when the job creates

many files, and the user wants to keep a subset of those files. WARNING: Do not specify **transfer_output_files** in the submit description file unless there is a really good reason – it is best to let Condor figure things out by itself based upon what the job produces.

transfer_files = <**ONEXIT** | **ALWAYS** | **NEVER**> Setting **transfer_files** equal to *ONEXIT* will cause Condor to transfer the job's output files back to the submitting machine only when the job completes (exits). If not specified, *ONEXIT* is used as the default on NT machines. If not specified, *NEVER* is used as the default on Unix machines. Specifying *ALWAYS* forces Condor to transfer output files either when the job completes or when the job is preempted or killed prior to job completion. The *ALWAYS* option is intended for fault tolerant jobs which periodically save their own state and can restart where they left off. Any output files transferred back to the submit machine are automatically sent back out again as input files when the job restarts.

requirements = <**ClassAd Boolean Expression**> The requirements command is a boolean ClassAd expression which uses C-like operators. In order for any job in this cluster to run on a given machine, this requirements expression must evaluate to true on the given machine. For example, to require that whatever machine executes your program has a least 64 Meg of RAM and has a MIPS performance rating greater than 45, use:

```
requirements = Memory >= 64 && Mips > 45
```

Only one requirements command may be present in a submit description file. By default, *condor_submit* appends the following clauses to the requirements expression:

1. Arch and OpSys are set equal to the Arch and OpSys of the submit machine. In other words: unless you request otherwise, Condor will give your job machines with the same architecture and operating system version as the machine running *condor_submit*.
2. Disk >= DiskUsage. The `DiskUsage` attribute is initialized to the size of the executable plus the size of any files specified in a **transfer_input_files** command. It exists to ensure there is enough disk space on the target machine for Condor to copy over both the executable and needed input files. The `DiskUsage` attribute represents the maximum amount of total disk space required by the job in kilobytes. Condor automatically updates the `DiskUsage` attribute approximately every 20 minutes while the job runs with the amount of space being used by the job on the execute machine.
3. VirtualMemory >= ImageSize. To ensure the target machine has enough virtual memory to run your job.
4. If Universe is set to Vanilla, FileSystemDomain is set equal to the submit machine's FileSystemDomain.

You can view the requirements of a job which has already been submitted (along with everything else about the job ClassAd) with the command *condor_q -l*; see the command reference for *condor_q* on page 360. Also, see the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

rank = <**ClassAd Float Expression**> A ClassAd Floating-Point expression that states how to rank machines which have already met the requirements expression. Essentially, rank expresses preference. A higher numeric value equals better rank. Condor will give the job the machine with the highest rank. For example,

```
requirements = Memory > 60
rank = Memory
```

asks Condor to find all available machines with more than 60 megabytes of memory and give the job the one with the most amount of memory. See the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

universe = <**vanilla** | **standard** | **pvm** | **scheduler** | **globus** | **mpi** | **java**> Specifies which Condor Universe to use when running this job. The Condor Universe specifies a Condor execution environment. The standard Universe is the default (except where the configuration variable `DEFAULT_UNIVERSE` defines it otherwise), and tells Condor that this job has been re-linked via *condor_compile* with the Condor libraries and therefore supports checkpointing and remote system calls. The *vanilla* Universe is an execution environment for jobs which have not been linked with the Condor libraries. *Note:* Use the *vanilla* Universe to submit shell scripts to Condor. The *pvm* Universe is for a parallel job written with PVM 3.4. The *scheduler* is for a job that should act as a metascheduler. The *globus* universe uses the Globus GRAM API to contact the Globus resource specified and requests it run the job. The *mpi* universe is for running mpi jobs made with the MPICH package. The *java* Universe is for programs written to the Java Virtual Machine. See the Condor User's Manual for more information about using Universe.

on_exit_remove = <**ClassAd Boolean Expression**> This expression is checked when the job exits and if true, then it allows the job to leave the queue normally. If false, then the job is placed back into the Idle state. If the user job is a vanilla job then it restarts from the beginning. If the user job is a standard job, then it restarts from the last checkpoint.

For example: Suppose you have a job that occasionally segfaults but you know if you run it again on the same data, chances are it will finish successfully. This is how you would represent that with `on_exit_remove`(assuming the signal identifier for segmentation fault is 4):

```
on_exit_remove = (ExitBySignal == True) && (ExitSignal != 4)
```

The above expression will not let the job exit if it exited by a signal and that signal number was 4(representing segmentation fault). In any other case of the job exiting, it will leave the queue as it normally would have done.

If left unspecified, this will default to `True`.

periodic_$*$ expressions(defined elsewhere in this man page) take precedent over on_exit_$*$ expressions and a $*$_hold expression takes precedent over a $*$_remove expression.

This expression is available for the vanilla and java universes. It is additionally available, when submitted from a Unix machine, for the standard universe.

on_exit_hold = <**ClassAd Boolean Expression**> This expression is checked when the job exits and if true, places the job on hold. If false then nothing happens and the `on_exit_remove` expression is checked to determine if that needs to be applied.

For example: Suppose you have a job that you know will run for an hour minimum. If the job exits after less than an hour, you would like it to be placed on hold and notified by e-mail instead of being allowed to leave the queue.

```
on_exit_hold = (ServerStartTime - JobStartDate) < 3600
```

The above expression will place the job on hold if it exits for any reason before running for an hour. An e-mail will be sent to the user explaining that the job was placed on hold because this expression became true.

periodic_* expressions(defined elsewhere in this man page) take precedent over on_exit_* expressions and any *_hold expression takes precedent over a *_remove expression.

If left unspecified, this will default to `False`.

This expression is available for the vanilla and java universes. It is additionally available, when submitted from a Unix machine, for the standard universe.

periodic_remove = <**ClassAd Boolean Expression**> This expression is checked every 20 seconds(non-configurable, but might be in future) and if it becomes true, the job will leave the queue. `periodic_remove` takes precedent over `on_exit_remove` if the two describe conflicting states.

For example: Suppose you would like your job removed if the total suspension time of the job is more than half of the run time of the job.

```
periodic_remove = CumulativeSuspensionTime > (RemoteWallClock-
Time / 2.0)
```

The above expression will remove the job once the conditions have become true.

*Note:*Currently, this option will force a "terminate" event in the user log of the job and it will report a successful termination of the job. This reporting of this event will be changed in a future version of Condor to be a "job aborted event" with descriptions of who initiated the event. As it stands now, someone looking at the user log termination event wouldn't be able to tell the difference between a job that legitimately ended successfully and a job where the `periodic_remove` expression had become true.

periodic_* expressions(defined elsewhere in this man page) take precedent over on_exit_* expressions and any *_hold expression takes precedent over a *_remove expression.

If left unspecified, this will default to `False`.

This expression is available for the vanilla and java universes. It is additionally available, when submitted from a Unix machine, for the standard universe.

periodic_hold = <**ClassAd Boolean Expression**> This expression is checked every 20 seconds(non-configurable, but might be in future) and if it becomes true, the job will be placed on hold.

For example: Suppose you would like your job held if the total suspension time of the job is more than half of the total run time of the job.

```
periodic_hold = CumulativeSuspensionTime > (RemoteWallClock-
Time / 2.0)
```

The above expression will place the job on hold if it suspends longer than half the amount of time it has totally run. An e-mail will be sent to the user explaining that the job was placed on hold because this expression became true.

If left unspecified, this will default to `False`.

periodic_* expressions(defined elsewhere in this man page) take precedent over on_exit_* expressions and any *_hold expression takes precedent over a *_remove expression.

This expression is available for the vanilla and java universes. It is additionally available, when submitted from a Unix machine, for the standard universe.

priority = <**priority**> Condor job priorities range from -20 to +20, with 0 being the default. Jobs with higher numerical priority will run before jobs with lower numerical priority. Note that this priority is on a per user basis; setting the priority will determine the order in which your own jobs are executed, but will have no effect on whether or not your jobs will run ahead of another user's jobs.

notification = <**when**> Owners of Condor jobs are notified by email when certain events occur. If *when* is set to Always, the owner will be notified whenever the job is checkpointed, and when it completes. If *when* is set to Complete (the default), the owner will be notified when the job terminates. If *when* is set to Error, the owner will only be notified if the job terminates abnormally. If *when* is set to Never, the owner will not be mailed, regardless what happens to the job. The statistics included in the email are documented in section 2.6.5 on page 34.

notify_user = <**email-address**> Used to specify the email address to use when Condor sends email about a job. If not specified, Condor will default to using :

```
job-owner@UID_DOMAIN
```

where `UID_DOMAIN` is specified by the Condor site administrator. If `UID_DOMAIN` has not been specified, Condor will send the email to :

```
job-owner@submit-machine-name
```

copy_to_spool = <**True** | **False**> If **copy_to_spool** is set to *True*, then *condor_submit* will copy the executable to the local spool directory before running it on a remote host. Oftentimes this can be quite time consuming and unnecessary. By setting it to *False*, *condor_submit* will skip this step. Defaults to *True*.

getenv = <**True** | **False**> If **getenv** is set to *True*, then *condor_submit* will copy all of the user's current shell environment variables at the time of job submission into the job ClassAd. The job will therefore execute with the same set of environment variables that the user had at submit time. Defaults to *False*.

hold = <**True** | **False**> If **hold** is set to *True*, then the job will be submitted in the hold state. Jobs in the hold state will not run until released by *condor_release*.

environment = <**parameter_list**> List of environment variables of the form :

```
<parameter>=<value>
```

Multiple environment variables can be specified by separating them with a semicolon (" ; ") when submitting from a Unix platform. Multiple environment variables can be specified by separating them with a vertical bar (" — ") when submitting from an NT platform. These environment variables will be placed (as given) into the job's environment before execution. The length of all characters specified in the environment is currently limited to 10240 characters. Note that spaces are accepted, but rarely desired, characters within parameter names and values. Place spaces within the parameter list only if required.

log = <**pathname**> Use **log** to specify a filename where Condor will write a log file of what is happening with this job cluster. For example, Condor will log into this file when and where the job begins running, when the job is checkpointed and/or migrated, when the job completes, etc. Most users find specifying a **log** file to be very handy; its use is recommended. If no **log** entry is specified, Condor does not create a log for this cluster.

jar_files = <**file_list**> Specifies a list of additional JAR files to include when using the Java Universe. JAR files will be transferred along with the executable and automatically added to the classpath.

image_size = <**size**> This command tells Condor the maximum virtual image size to which you believe your program will grow during its execution. Condor will then execute your job only on machines which have enough resources, (such as virtual memory), to support executing your job. If you do not specify the image size of your job in the description file, Condor will automatically make a (reasonably accurate) estimate about its size and adjust this estimate as your program runs. If the image size of your job is underestimated, it may crash due to inability to acquire more address space, e.g. malloc() fails. If the image size is overestimated, Condor may have difficulty finding machines which have the required resources. *size* must be in kbytes, e.g. for an image size of 8 megabytes, use a *size* of 8000.

machine_count = <**min..max**> | <**max**> For the PVM universe, both *min* and *max* or just *max* may be defined. If **machine_count** is specified, Condor will not start the job until it can simultaneously supply the job with *min* machines. Condor will continue to try to provide up to *max* machines, but will not delay starting of the job to do so. If the job is started with fewer than *max* machines, the job will be notified via a usual PvmHostAdd notification as additional hosts come on line.

For the MPI universe, a single value (*max*) is required. It is neither a maximum or minimum, but the number of machines to be dedicated toward running the job.

coresize = <**size**> Should the user's program abort and produce a core file, **coresize** specifies the maximum size in bytes of the core file which the user wishes to keep. If **coresize** is not specified in the command file, the system's user resource limit "coredumpsize" is used (except on HP-UX).

nice_user = <**True** | **False**> Normally, when a machine becomes available to Condor, Condor decides which job to run based upon user and job priorities. Setting **nice_user** equal to *True* tells Condor not to use your regular user priority, but that this job should have last priority among all users and all jobs. So jobs submitted in this fashion run only on machines which no other non-nice_user job wants — a true "bottom-feeder" job! This is very handy if a user has some jobs they wish to run, but do not wish to use resources that could instead be used to run

other people's Condor jobs. Jobs submitted in this fashion have "nice-user." pre-appended in front of the owner name when viewed from *condor_q* or *condor_userprio*. The default value is **False**.

kill_sig = <signal-number> When Condor needs to kick a job off of a machine, it will send the job the signal specified by *signal-number*. *signal-number* needs to be an integer which represents a valid signal on the execution machine. For jobs submitted to the Standard Universe, the default value is the number for `SIGTSTP` which tells the Condor libraries to initiate a checkpoint of the process. For jobs submitted to the Vanilla Universe, the default is `SIGTERM` which is the standard way to terminate a program in UNIX.

compress_files = file1, file2, ... If your job attempts to access any of the files mentioned in this list, Condor will automatically compress them (if writing) or decompress them (if reading). The compress format is the same as used by GNU gzip.

The files given in this list may be simple filenames or complete paths and may include $*$ as a wildcard. For example, this list causes the file /tmp/data.gz, any file named event.gz, and any file ending in .gzip to be automatically compressed or decompressed as needed:

```
compress_files = /tmp/data.gz, event.gz, *.gzip
```

Due to the nature of the compression format, compressed files must only be accessed sequentially. Random access reading is allowed but is very slow, while random access writing is simply not possible. This restriction may be avoided by using both compress_files and fetch_files at the same time. When this is done, a file is kept in the decompressed state at the execution machine, but is compressed for transfer to its original location.

This option only applies to standard-universe jobs.

fetch_files = file1, file2, ... If your job attempts to access a file mentioned in this list, Condor will automatically copy the whole file to the executing machine, where it can be accessed quickly. When your job closes the file, it will be copied back to its original location. This list uses the same syntax as compress_files, shown above.

This option only applies to standard-universe jobs.

append_files = file1, file2, ... If your job attempts to access a file mentioned in this list, Condor will force all writes to that file to be appended to the end. Furthermore, condor_submit will not truncate it. This list uses the same syntax as compress_files, shown above.

This option may yield some surprising results. If several jobs attempt to write to the same file, their output may be intermixed. If a job is evicted from one or more machines during the course of its lifetime, such an output file might contain several copies of the results. This option should be only be used when you wish a certain file to be treated as a running log instead of a precise result.

This option only applies to standard-universe jobs.

local_files = file1, file2, ... If your job attempts to access a file mentioned in this list, Condor will cause it to be read or written at the execution machine. This is most useful for temporary files not used for input or output. This list uses the same syntax as compress_files, shown above.

```
local_files = /tmp/*
```

This option only applies to standard-universe jobs.

file_remaps = < **" name = newname ; name2 = newname2 ... "**> Directs Condor to use a new filename in place of an old one. *name* describes a filename that your job may attempt to open, and *newname* describes the filename it should be replaced with. *newname* may include an optional leading access specifier, `local:` or `remote:`. If left unspecified, the default access specifier is `remote:`. Multiple remaps can be specified by separating each with a semicolon.

This option only applies to standard universe jobs.

If you wish to remap file names that contain equals signs or semicolons, these special characters may be escaped with a backslash.

Example One: Suppose that your job reads a file named `dataset.1`. To instruct Condor to force your job to read `other.dataset` instead, add this to the submit file:

```
file_remaps = "dataset.1=other.dataset"
```

Example Two: Suppose that your run many jobs which all read in the same large file, called `very.big`. If this file can be found in the same place on a local disk in every machine in the pool, (say `/bigdisk/bigfile`,) you can instruct Condor of this fact by remapping `very.big` to `/bigdisk/bigfile` and specifying that the file is to be read locally, which will be much faster than reading over the network.

```
file_remaps = "very.big = local:/bigdisk/bigfile"
```

Example Three: Several remaps can be applied at once by separating each with a semicolon.

```
file_remaps = "very.big = local:/bigdisk/bigfile ; dataset.1 = other.dataset'
```

buffer_files = < **" name = (size,block-size) ; name2 = (size,block-size) ... "** >

buffer_size = <**bytes-in-buffer**>

buffer_block_size = <**bytes-in-block**> Condor keeps a buffer of recently-used data for each file a job accesses. This buffer is used both to cache commonly-used data and to consolidate small reads and writes into larger operations that get better throughput. The default settings should produce reasonable results for most programs.

These options only apply to standard-universe jobs.

If needed, you may set the buffer controls individually for each file using the buffer_files option. For example, to set the buffer size to 1 Mbyte and the block size to 256 KBytes for the file `input.data`, use this command:

```
buffer_files = "input.data=(1000000,256000)"
```

Alternatively, you may use these two options to set the default sizes for all files used by your job:

```
buffer_size = 1000000
buffer_block_size = 256000
```

If you do not set these, Condor will use the values given by these two configuration file macros:

```
DEFAULT_IO_BUFFER_SIZE = 1000000
DEFAULT_IO_BUFFER_BLOCK_SIZE = 256000
```

Finally, if no other settings are present, Condor will use a buffer of 512 Kbytes and a block size of 32 Kbytes.

rendezvousdir = <**directory-path**> Used to specify the shared file system directory to be used for file system authentication when submitting to a remote scheduler. Should be a path to a preexisting directory.

x509directory = <**directory-path**> Used to specify the directory which contains the certificate, private key, and trusted certificate directory for GSS authentication. If this attribute is set, the environment variables X509_USER_KEY, X509_USER_CERT, and X509_CERT_DIR are exported with default values. See section 3.7.4 for more info.

x509userproxy = <**full-pathname**> Used to override the default pathname for X509 user certificates. The default location for X509 proxies is the `/tmp` directory, which is generally a local file system. Setting this value would allow Condor to access the proxy in a shared file system (for example, AFS). Condor will use the proxy specified in the submit description file first. If nothing is specified in the submit description file, it will use the environment variable X509_USER_CERT. If that variable is not present, it will search in the default location. See section 3.7.4 for more info.

globusscheduler = <**scheduler-name**> Used to specify the Globus resource to which the job should be submitted. More than one scheduler can be submitted to, simply place a *queue* command after each instance of **globusscheduler**. Each instance should be a valid Globus scheduler, using either the full Globus contact string or the host/scheduler format shown below:

Example: To submit to the LSF scheduler of the Globus gatekeeper on lego at Boston University:

```
GlobusScheduler = lego.bu.edu/jobmanager-lsf
queue
```

globusrsl = <**RSL-string**> Used to provide any additional Globus RSL string attributes which are not covered by regular submit description file parameters.

transfer_executable = <**True** | **False**> If **transfer_executable** is set to *false*, then Condors look for the executable on the remote machine, and does not transfer the executable over. This is useful if you have already pre-staged your executable and wish to have Condor behave more like rsh. Defaults to *True*. This option is only used in the Globus universe.

+<attribute> = <value> A line which begins with a '+' (plus) character instructs *condor_submit* to insert the following *attribute* into the job ClasssAd with the given *value*.

queue [number-of-procs] Places one or more copies of the job into the Condor queue. If desired, new **input**, **output**, **error**, **initialdir**, **arguments**, **nice_user**, **priority**, **kill_sig**, **coresize**, or **image_size** commands may be issued between **queue** commands. This is very handy when submitting multiple runs into one cluster with one submit description file; for example, by issuing an **initialdir** between each **queue** command, each run can work in its own subdirectory. The optional argument *number-of-procs* specifies how many times to submit the job to the queue, and it defaults to 1.

In addition to commands, the submit description file can contain macros and comments:

Macros Parameterless macros in the form of `$(macro_name)` may be inserted anywhere in Condor submit description files. Macros can be defined by lines in the form of

```
<macro_name> = <string>
```

Three pre-defined macros are supplied by the submit description file parser. The third of the pre-defined macros is only relevant to MPI universe jobs. The `$(Cluster)` macro supplies the number of the job cluster, and the `$(Process)` macro supplies the number of the job. These macros are intended to aid in the specification of input/output files, arguments, etc., for clusters with lots of jobs, and/or could be used to supply a Condor process with its own cluster and process numbers on the command line. The `$(Process)` macro should not be used for PVM jobs. The `$(Node)` macro is defined only for MPI universe jobs. It is a unique value assigned for the duration of the job that essentially identifies the machine on which a program is executing.

If the dollar sign ("$") is desired as a literal character, then use

```
$(DOLLAR)
```

In addition to the normal macro, there is also a special kind of macro called a *substitution macro* that allows you to substitute expressions defined on the resource machine itself (gotten after a match to the machine has been performed) into specific expressions in your submit description file. The special substitution macro is of the form:

```
$$(attribute)
```

The substitution macro may only be used in three expressions in the submit description file: `executable`, `environment`, and `arguments`. The most common use of this macro is for heterogeneous submission of an executable:

```
executable = povray.$$(opsys).$$(arch)
```

The `opsys` and `arch` attributes will be substituted at match time for any given resource. This will allow Condor to automatically choose the correct executable for the matched machine.

An extension to the syntax of the substitution macro provides an alternative string to use if the machine attribute within the substitution macro is undefined. The syntax appears as:

```
$$(attribute:string_if_attribute_undefined)
```

An example using this extended syntax provides a path name to a required input file. Since the file can be placed in different locations on different machines, the file's path name is given as an argument to the program.

```
argument = $$(input_file_path:/usr/foo)
```

On the machine, if the attribute `input_file_path` is not defined, then the path `/usr/foo` is used instead.

The environment macro, $ENV, allows the evaluation of an environment variable to be used in setting a submit description file command. The syntax used is

```
$ENV(variable)
```

An example submit description file command that uses this functionality evaluates the submitter's home directory in order to set the path and file name of a log file:

```
log = $ENV(HOME)/jobs/logfile
```

The environment variable is evaluated when the submit description file is processed.

Comments Blank lines and lines beginning with a pound sign ('#') character are ignored by the submit description file parser.

Options

– Accept the command file from stdin.

-v Verbose output - display the created job class-ad

-n *schedd_name* Submit to the specified schedd. This option is used when there is more than one schedd running on the submitting machine

-r *schedd_name* Submit to a remote schedd. The jobs will be submitted to the schedd on the specified remote host. On Unix systems, the Condor administrator for you site must override the default AUTHENTICATION_METHODS configuration setting to enable remote file system (FS_REMOTE) authentication.

-d Disable file permission checks.

-a *command* Augment the commands in the submit description file with the given command. This command will be considered to immediately precede the Queue command within the submit description file, and come after all other previous commands. The submit description file is not modified. Multiple commands are specified by using the **-a** option multiple times. Each new command is given in a separate **-a** option. Commands with spaces in them will need to be enclosed in double quote marks.

submit description file The pathname to the submit description file. If this optional argument is missing, then the commands are taken from standard input.

Exit Status

condor_submit will exit with a status value of 0 (zero) upon success, and a non-zero value upon failure.

Examples

- Submit Description File Example 1: This example queues three jobs for execution by Condor. The first will be given command line arguments of *15* and *2000*, and it will write its standard output to `foo.out1`. The second will be given command line arguments of *30* and *2000*, and it will write its standard output to `foo.out2`. Similarly the third will have arguments of *45* and *6000*, and it will use `foo.out3` for its standard output. Standard error output (if any) from all three programs will appear in `foo.error`.

```
####################
#
# submit description file
# Example 1: queuing multiple jobs with differing
# command line arguments and output files.
#
####################

Executable     = foo
Universe       = standard

Arguments      = 15 2000
Output  = foo.out1
Error   = foo.err1
Queue

Arguments      = 30 2000
Output  = foo.out2
```

```
Error   = foo.err2
Queue

Arguments      = 45 6000
Output  = foo.out3
Error   = foo.err3
Queue
```

- Submit Description File Example 2: This submit description file example queues 150 runs of program *foo* which must have been compiled and linked for Silicon Graphics workstations running IRIX 6.x. Condor will not attempt to run the processes on machines which have less than 32 Megabytes of physical memory, and it will run them on machines which have at least 64 Megabytes, if such machines are available. Stdin, stdout, and stderr will refer to `in.0`, `out.0`, and `err.0` for the first run of this program (process 0). Stdin, stdout, and stderr will refer to `in.1`, `out.1`, and `err.1` for process 1, and so forth. A log file containing entries about where and when Condor runs, takes checkpoints, and migrates processes in this cluster will be written into file `foo.log`.

```
####################
#
# Example 2: Show off some fancy features including
# use of pre-defined macros and logging.
#
####################

Executable     = foo
Universe       = standard
Requirements   = Memory >= 32 && Op-
Sys == "IRIX6" && Arch =="SGI"
Rank           = Memory >= 64
Image_Size     = 28 Meg

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log = foo.log

Queue 150
```

- Command Line example: The following command uses the **-a** option to add two commands before the job(s) is queued. A log file and an error log file are specified. The submit description file is unchanged.

```
condor_submit -a "log = out.log" -a "error = er-
ror.log" mysubmitfile
```

Note that each of the added commands is contained within quote marks because there are space characters within the command.

General Remarks

- For security reasons, Condor will refuse to run any jobs submitted by user root (UID = 0) or by a user whose default group is group wheel (GID = 0). Jobs submitted by user root or a user with a default group of wheel will appear to sit forever in the queue in an idle state.
- All pathnames specified in the submit description file must be less than 256 characters in length, and command line arguments must be less than 4096 characters in length; otherwise, *condor_submit* gives a warning message but the jobs will not execute properly.
- Somewhat understandably, behavior gets bizarre if the user makes the mistake of requesting multiple Condor jobs to write to the same file, and/or if the user alters any files that need to be accessed by a Condor job which is still in the queue. For example, the compressing of data or output files before a Condor job has completed is a common mistake.
- To disable checkpointing for Standard Universe jobs, include the line:

  ```
  +WantCheckpoint = False
  ```

  in the submit description file before the queue command(s).

See Also

Condor User Manual

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial

Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_submit_dag

Manage and queue jobs within a specified DAG for execution on remote machines

Synopsis

condor_submit_dag [**-no_submit**] [**-verbose**] [**-force**] [**-maxjobs** *NumberOfJobs*] [**-maxpre** *NumberOfPREscripts*] [**-maxpost** *NumberOfPOSTscripts*] [**-nopostfail**] [**-log** *LogFileName*] [**-notification** *value*] *DAGInputFile*

Description

condor_submit_dag is the program for submitting a DAG (directed acyclic graph) of jobs for execution under Condor. The program enforces the job dependencies defined in the *DAGInputFile*. The *DAGInputFile* contains commands to direct the submission of jobs implied by the nodes of a DAG to Condor. See the Condor User Manual, section 2.12 for a complete description.

Options

-no_submit Produce the Condor submit description file for DAGMan, but do not submit DAGMan as a Condor job.

-verbose Give verbose error messages.

-force Require *condor_submit_dag* to overwrite the files that it produces, if the files already exist.

-maxjobs *NumberOfJobs* Sets the maximum number of jobs within the DAG that will be submitted to Condor at one time. *NumberOfJobs* is a positive integer. If the option is omitted, the default number of jobs is unlimited.

-maxpre *NumberOfPREscripts* Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPREScripts* is a positive integer. If this option is omitted, the default number of PRE scripts is unlimited.

-maxpost *NumberOfPOSTscripts* Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPOSTScripts* is a positive integer. If this option is omitted, the default number of POST scripts is unlimited.

-nopostfail An option applied to all nodes within the DAG that prevents the POST script within a node from running in the case that the job within the node fails. Without this option, POST scripts always run when jobs fail.

-log *LogFileName* Forces *condor_submit_dag* to omit the check of Condor submit description files for nodes within the DAG to verify that they all use the same log file. The argument *LogFileName* is used as the single, common log file.

-notification *value* Sets the e-mail notification for DAGMan itself. This information will be used within the Condor submit description file for DAGMan. This file is produced by *condor_submit_dag*. See **notification** within the section of submit description file commands in the *condor_submit* manual page on page 395 for specification of *value*.

See Also

Condor User Manual

Exit Status

condor_submit_dag will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention:

Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_userlog

Display and summarize job statistics from job log files.

Synopsis

condor_userlog [**-help**] [**-total** | **-raw**] [**-debug**] [**-evict**] [**-j** *cluster* | *cluster.proc*] [**-all**] [**-hostname**] *logfile* . . .

Description

condor_userlog parses the information in job log files and displays summaries for each workstation allocation and for each job. See the manual page for *condor_submit* on page 395 for instructions for specifying that Condor write a log file for your jobs.

If **-total** is not specified, *condor_userlog* will first display a record for each workstation allocation, which includes the following information:

Job The cluster/process id of the Condor job.

Host The host where the job ran. By default, the host's IP address is displayed. If **-hostname** is specified, the hostname will be displayed instead.

Start Time The time (month/day hour:minute) when the job began running on the host.

Evict Time The time (month/day hour:minute) when the job was evicted from the host.

Wall Time The time (days+hours:minutes) for which this workstation was allocated to the job.

Good Time The allocated time (days+hours:min) which contributed to the completion of this job. If the job exited during the allocation, then this value will equal "Wall Time." If the job performed a checkpoint, then the value equals the work saved in the checkpoint during this allocation. If the job did not exit or perform a checkpoint during this allocation, the value will be 0+00:00. This value can be greater than 0 and less than "Wall Time" if the application completed a periodic checkpoint during the allocation but failed to checkpoint when evicted.

CPU Usage The CPU time (days+hours:min) which contributed to the completion of this job.

condor_userlog will then display summary statistics per host:

Host/Job The IP address or hostname for the host.

Wall Time The workstation time (days+hours:minutes) allocated by this host to the jobs specified in the query. By default, all jobs in the log are included in the query.

Good Time The time (days+hours:minutes) allocated on this host which contributed to the completion of the jobs specified in the query.

CPU Usage The CPU time (days+hours:minutes) obtained from this host which contributed to the completion of the jobs specified in the query.

Avg Alloc The average length of an allocation on this host (days+hours:minutes).

Avg Lost The average amount of work lost (days+hours:minutes) when a job was evicted from this host without successfully performing a checkpoint.

Goodput This percentage is computed as Good Time divided by Wall Time.

Util. This percentage is computed as CPU Usage divided by Good Time.

condor_userlog will then display summary statistics per job:

Host/Job The cluster/process id of the Condor job.

Wall Time The total workstation time (days+hours:minutes) allocated to this job.

Good Time The total time (days+hours:minutes) allocated to this job which contributed to the job's completion.

CPU Usage The total CPU time (days+hours:minutes) which contributed to this job's completion.

Avg Alloc The average length of a workstation allocation obtained by this job in minutes (days+hours:minutes).

Avg Lost The average amount of work lost (days+hours:minutes) when this job was evicted from a host without successfully performing a checkpoint.

Goodput This percentage is computed as Good Time divided by Wall Time.

Util. This percentage is computed as CPU Usage divided by Good Time.

Finally, *condor_userlog* will display a summary for all hosts and jobs.

Options

-help Get a brief description of the supported options

-total Only display job totals

-raw Display raw data only

-debug Debug mode

-j Select a specific cluster or cluster.proc

-evict Select only allocations which ended due to eviction

-all Select all clusters and all allocations

-hostname Display hostname instead of IP address

General Remarks

Since the Condor job log file format does not contain a year field in the timestamp, all entries are assumed to occur in the current year. Allocations which begin in one year and end in the next will be silently ignored.

Exit Status

condor_userlog will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention:

Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_userprio

Manage user priorities

Synopsis

condor_userprio [**-pool** *hostname*] [**-all**] [**-usage**] [**-setprio** *username value*] [**-setfactor** *username value*] [**-resetusage** *username*] [**-resetall**] [**-getreslist** *username*] [**-allusers**] [**-activefrom** *month day year*] [**-l**]

Description

condor_userprio with no arguments, lists the active users (see below) along with their priorities, in increasing priority order. The -all option can be used to display more detailed information about each user, which includes the following columns:

Effective Priority The effective priority value of the user, which is used to calculate the user's share when allocating resources. A lower value means a higher priority, and the minimum value (highest priority) is 0.5. The effective priority is calculated by multiplying the real priority by the priority factor.

Real Priority The value of the real priority of the user. This value follows the user's resource usage.

Priority Factor The system administrator can set this value for each user, thus controlling a user's effective priority relative to other users. This can be used to create different classes of users.

Res Used The number of resources currently used (e.g. the number of running jobs for that user).

Accumulated Usage The accumulated number of resource-hours used by the user since the usage start time.

Usage Start Time The time since when usage has been recorded for the user. This time is set when a user job runs for the first time. It is reset to the present time when the usage for the user is reset (with the -resetusage or -resetall options).

Last Usage Time The most recent time a resource usage has been recorded for the user.

The -usage option displays the username, accumulated usage, usage start time and last usage time for each user, sorted on accumulated usage.

The -setprio, -setfactor options are used to change a user's real priority and priority factor. The -resetusage and -resetall options are used to reset the accumulated usage for users. The usage start time is set to the current time when the accumulated usage is reset. These options require administrator privilages.

By default only users for whom usage was recorded in the last 24 hours or whose priority is greater than the minimum are listed. The -activefrom and -allusers options can be used to display users who had some usage since a specified date, or ever. The summary line for last usage time will show this date.

The -getreslist option is used to display the resources currently used by a user. The output includes the start time (the time the resource was allocated to the user), and the match time (how long has the resource been allocated to the user).

Note that when specifying usernames on the command line, the name must include the uid domain (e.g. user@uid-domain - exactly the same way usernames are listed by the userprio command).

The -pool option can be used to contact a different central-manager instead of the local one (the default).

Options

-pool ***hostname*** Contact the specified hostname instead of the local central manager. This can be used to check other pools.

-all Display detailed information about each user.

-usage Display usage information for each user.

-setprio ***username value*** Set the real priority of the specified user to the specified value.

-setfactor ***username value*** Set the priority factor of the specified user to the specified value.

-resetusage ***username*** Reset the accumulated usage of the specified user to zero.

-resetall Reset the accumulated usage of all the users to zero.

-getreslist ***username*** Display all the resources currently allocated to the specified user.

-allusers Display information for all the users who have some recorded accumulated usage.

-activefrom ***month day year*** Display information for users who have some recorded accumulated usage since the specified date.

-l Show the class-ad which was received from the central-manager in long format.

Exit Status

condor_userprio will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

condor_vacate

Vacate jobs that are running on the specified hosts

Synopsis

condor_vacate [**-help** | **-version**]

condor_vacate [**-graceful** | **-fast**] [**-name hostname** | **hostname** | **-addr "<a.b.c.d:port>"** | **"<a.b.c.d:port>"**] . . .

Description

condor_vacate causes Condor to checkpoint any running jobs on a set of machines and force the jobs to vacate the machine. The job(s) remains in the submitting machine's job queue.

Given the (default) **-graceful** option, a job running under the standard universe will first produce a checkpoint and then the job will be killed. Condor will then restart the job somewhere else, using the checkpoint to continue from where it left off. A job running under the vanilla universe is killed, and Condor restarts the job from the beginning somewhere else. *condor_vacate* has no effect on a machine with no Condor job currently running.

There is generally no need for the user or administrator to explicitly run *condor_vacate*. Condor takes care of jobs in this way automatically following the policies given in configuration files.

Options

-help Display usage information

-version Display version information

-graceful Inform the job to checkpoint, then soft-kill it.

-fast Hard-kill jobs instead of checkpointing them

-pool *centralmanagerhostname* Specify a pool by giving the central manager's hostname

-name *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by **hostname**

-addr *"<a.b.c.d:port>"* Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>" Send the command to a machine located at **"<a.b.c.d:port>"**

Exit Status

condor_vacate will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To send a *condor_vacate* command to two named machines:

```
% condor_vacate  robin cardinal
```

To send the *condor_vacate* command to a machine within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command sends the command to a the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_vacate -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2001 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and

Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.4.0 Manual* for additional notices.

INDEX

$ENV
    in configuration file, 83
    in submit description file, 406
condor_glidein, 50
_RUNTIME macro, 123
_STANDARD macro, 123

ACCOUNTANT_LOCAL_DOMAIN macro, 112
ActivationTimer macro, 144
activities and state figure, 137
activity
    of a machine, 135
    transitions, 138–143
    transitions summary, 143
ActivityTimer macro, 144
administrator
    e-mail to, 76
ADMINISTRATOR macro, 122
administrator's manual
    on Unix, 64–207
AFS
    interaction with, 61
AfterHours macro, 148
agents
    condor_shadow, 12
ALIVE_INTERVAL macro, 105
ALL_DEBUG macro, 91
ALLOW_* macros macro, 165
ALLOW_ADMIN_COMMANDS macro, 100
ALLOW_ADMINISTRATOR macro, 164
ALLOW_CONFIG macro, 164
ALLOW_DAEMON macro, 164
ALLOW_NEGOTIATOR macro, 164
ALLOW_OWNER macro, 164
ALLOW_READ macro, 164
ALLOW_REMOTE_SUBMIT macro, 106
ALLOW_WRITE macro, 164
ALTERNATE_STARTER_1 macro, 101
APPEND_PREF_STANDARD macro, 109
APPEND_PREF_VANILLA macro, 109
APPEND_RANK_STANDARD macro, 109, 121, 190
APPEND_RANK_VANILLA macro, 109
APPEND_REQ_<universe> macro, 288
APPEND_REQ_STANDARD macro, 109, 121, 190, 288
APPEND_REQ_VANILLA macro, 109, 288
APPEND_REQUIREMENTS macro, 288
APPEND_REQUREMENTS macro, 109
ARCH macro, 85, 180, 317
Arguments macro, 297
argv[0]
    Condor use of, 63
ATTR_JOB_STATUS macro, 284
authentication, 161–163
AUTHENTICATION_METHODS macro, 92, 306

BackgroundLoad macro, 144
batch system, 7
BIN macro, 86
buffer_block_size macro, 110
buffer_size macro, 110

central manager, 64, 65
    installation issues, 70
checkpoint, 2, 3, 12
    periodic, 3
checkpoint image, 12

checkpoint server, 65
configuration of, 118, 187
installation, 116–121, 185–190
multiple servers, 119, 188
Chirp, 39
Chirp.jar, 40
ChirpClient, 40
ChirpInputStream, 39
ChirpOutputStream, 39
CKPT_SERVER_DEBUG macro, 118, 187
CKPT_SERVER_DIR macro, 97, 118, 187
CKPT_SERVER_HOST macro, 96, 119, 120, 188, 189, 199, 201, 298
CKPT_SERVER_LOG macro, 118, 187
ClassAd, 2, 4, 8
attributes, 8
job, 8
job attributes, 22, 131
machine, 8
machine attributes, 18, 127
machine example, 9
ClassAd attribute
rank, 17
rank examples, 23
requirements, 17, 33
CLASSAD_LIFETIME macro, 110
CLIENT_TIMEOUT macro, 111
Cluster macro, 405
cluster macro, 56
CM_IP_ADDR macro, 89, 310
COLLECTOR_DEBUG macro, 112
COLLECTOR_HOST macro, 86, 318, 362, 391
COLLECTOR_NAME macro, 111, 319
COLLECTOR_SOCKET_BUFSIZE macro, 111, 307
COMPRESS_PERIODIC_CKPT macro, 107, 311
COMPRESS_VACATE_CKPT macro, 107, 311
Condor
availability, 5
binaries, 252
configuration, 82
contact information, 6
default policy, 144
distribution, 252–255
downloading, 252–255
FAQ, 252–271
flocking, 192
Frequently Asked Questions, 252–271
getting, 252–255
limitations, under UNIX, 4
mailing list, 271
mailing-list, 271
new versions, notification of, 271
overview, 1–4
Personal, 255
pool, 64
PVM applications, 41
resource allocation, 8
resource management, 2
source code, 253
universe, 11
Unix administrator, 70
user manual, 7–63
Y2K, 270
Condor commands
condor_advertise, 323
condor_checkpoint, 325
condor_compile, 61, 328
condor_config_val, 331
condor_findhost, 335
condor_glidein, 338
condor_history, 342
condor_hold, 344
condor_master, 347
condor_master_off, 348
condor_off, 350
condor_on, 353
condor_preen, 356
condor_prio, 32, 35, 358
condor_q, 11, 29, 33, 360
condor_qedit, 366
condor_reconfig, 368
condor_reconfig_schedd, 371
condor_release, 373
condor_reschedule, 375
condor_restart, 377
condor_rm, 11, 31, 380
condor_run, 382
condor_stats, 386
condor_status, 9, 11, 18, 29, 30, 390

condor_submit, 11, 15, 62, 232, 395
condor_submit_dag, 411
condor_userprio, 35, 418
condor_vacate, 421
Condor daemon
condor_shadow, 12, 62
condor_startd, 127
Condor-G, 228–238
contrib module, 124, 231
GASS, 229
GRAM, 229
GSI, 229
installation, 231
installation with Contrib module, 231
installation with GPT NMI release, 232
job submission, 232
Condor-PVM, 41
CONDOR_ADMIN macro, 87, 88, 98, 110
condor_advertise command, 323
condor_checkpoint command, 325
condor_ckpt_server, 67
condor_collector, 67
condor_compile command, 328
condor_config_val command, 331
CONDOR_DEVELOPERS macro, 111
CONDOR_DEVELOPERS_COLLECTOR macro, 111
condor_findhost command, 335
condor_glidein command, 338
condor_history command, 342
condor_hold command, 344
CONDOR_HOST macro, 86, 198
CONDOR_IDS
environment variable, 71, 88
condor_init script, 75
condor_install script, 75
condor_kbdd, 67
condor_master, 66
condor_master command, 347
condor_master_off command, 348
condor_negotiator, 67
condor_off command, 350
condor_on command, 353
condor_preen command, 356
condor_prio command, 358
condor_q command, 360
condor_qedit command, 366
condor_reconfig command, 368
condor_reconfig_schedd command, 371
condor_release command, 373
CONDOR_REQUIREMENTS macro, 382
condor_reschedule command, 375
condor_restart command, 377
condor_rm command, 380
condor_run command, 382
condor_schedd, 66
CONDOR_SCRATCH_DIR, 36
condor_shadow, 12, 30, 67
condor_startd, 66
condor_startd daemon, 127
condor_starter, 66
condor_stats command, 386
condor_status command, 390
condor_submit command, 395
condor_submit_dag, 52
condor_submit_dag command, 411
CONDOR_SUPPORT_EMAIL macro, 88
condor_userprio command, 418
condor_vacate command, 421
Condor_View
use of*crontab* program, 116
CONDOR_VIEW_HOST macro, 106
CONDOR_VM, 36
CONDORADMIN macro, 115
CondorView, 190
installation, 116, 191
CondorView Client
installation, 114
CONFIG macro, 122
configuration, 82
example, 133
for flocking, 192
RANK, 133
START expression, 127
startd policy, 127
configuration file
$ENV definition, 83
Condor-wide entries, 86
daemon logging entries, 89
macro definitions, 82
macros, 85
pre-defined macros, 84

subsystem names, 84
configuration files
location, 73, 78
configuration macro
ACCOUNTANT_LOCAL_DOMAIN, 112
ADMINISTRATOR, 122
ALIVE_INTERVAL, 105
ALLOW_* macros, 165
ALLOW_ADMIN_COMMANDS, 100
ALLOW_REMOTE_SUBMIT, 106
ALL_DEBUG, 91
ALTERNATE_STARTER_1, 101
APPEND_PREF_STANDARD, 109
APPEND_PREF_VANILLA, 109
APPEND_RANK_STANDARD, 109, 121, 190
APPEND_RANK_VANILLA, 109
APPEND_REQUIREMENTS, 288
APPEND_REQUREMENTS, 109
APPEND_REQ_<universe>, 288
APPEND_REQ_STANDARD, 109, 121, 190, 288
APPEND_REQ_VANILLA, 109, 288
ARCH, 85, 180, 317
ATTR_JOB_STATUS, 284
AUTHENTICATION_METHODS, 92, 306
ActivationTimer, 144
ActivityTimer, 144
AfterHours, 148
Arguments, 297
BIN, 86
BackgroundLoad, 144
CKPT_SERVER_DEBUG, 118, 187
CKPT_SERVER_DIR, 97, 118, 187
CKPT_SERVER_HOST, 96, 119, 120, 188, 189, 199, 201, 298
CKPT_SERVER_LOG, 118, 187
CLASSAD_LIFETIME, 110
CLIENT_TIMEOUT, 111
CM_IP_ADDR, 89, 310
COLLECTOR_DEBUG, 112
COLLECTOR_HOST, 86, 318, 362, 391
COLLECTOR_NAME, 111, 319
COLLECTOR_SOCKET_BUFSIZE, 111, 307
COMPRESS_PERIODIC_CKPT, 107, 311
COMPRESS_VACATE_CKPT, 107, 311
CONDORADMIN, 115
CONDOR_ADMIN, 87, 88, 98, 110
CONDOR_DEVELOPERS_COLLECTOR, 111
CONDOR_DEVELOPERS, 111
CONDOR_HOST, 86, 198
CONDOR_REQUIREMENTS, 382
CONDOR_SUPPORT_EMAIL, 88
CONDOR_VIEW_HOST, 106
CONFIG, 122
CONSOLE_DEVICES, 80, 101, 181
CPUBusyTime, 145
CPUBusy, 144, 145
CPUIdle, 144
CPUIsBusy, 144
CREATE_CORE_FILES, 89, 319
Cluster, 405
ContinueIdleTime, 144
DAEMONNAME_ENVIRONMENT, 98, 276
DAEMON_LIST, 75, 97–99, 118, 124, 181, 182, 187, 191, 276, 277, 292, 302
DC_DAEMON_LIST, 97, 292
DEFAULT_DOMAIN_NAME, 88, 199, 317, 319
DEFAULT_IO_BUFFER_BLOCK_SIZE, 110, 295
DEFAULT_IO_BUFFER_SIZE, 110, 295
DEFAULT_PRIORITY_FACTOR, 280
DEFAULT_PRIO_FACTOR, 112
DEFAULT_RANK_STANDARD, 109
DEFAULT_RANK_VANILLA, 109
DEFAULT_UNIVERSE, 109, 276, 398
DISCONNECTED_KEYBOARD_IDLE_BOOST, 102, 196
D_SECONDS, 276
EMAIL_DOMAIN, 89, 278
ETC, 181
EVENTD_CAPACITY_INFO, 113, 124
EVENTD_INTERVAL, 113, 122

EVENTD_MAX_PREPARATION, 113
EVENTD_ROUTING_INFO, 113, 124
EVENTD_SHUTDOWN_CLEANUP_INTERVAL, 113
EVENTD_SHUTDOWN_SLOW_START_INTERVAL, 113
EVENT_LIST, 113, 123
EXECUTE, 19, 87, 88, 128
EXEC_TRANSFER_ATTEMPTS, 108, 281
FILESYSTEM_DOMAIN, 77, 85, 94–96, 198, 310
FLOCK_COLLECTOR_HOSTS, 106
FLOCK_FROM, 192
FLOCK_NEGOTIATOR_HOSTS, 106, 192
FLOCK_TO, 192
FLOCK_VIEW_SERVERS, 106
FLOCK_VIEW_SERVER, 106
FS_PATHNAME, 310
FULL_HOSTNAME, 77, 84, 88, 89
GLOBUSRUN, 51, 109
GRIDMANAGER_LOG, 231
HAS_AFS, 95
HIGHPORT, 89
HISTORY, 88, 110, 280
HOSTALLOW. . ., 92
HOSTALLOW_READ, 171
HOSTALLOW_WRITE, 51
HOSTALLOW, 92
HOSTDENY, 92
HOSTNAME, 71, 78, 79, 84, 88
HOST_ALLOW_ADMINISTRATOR, 175
HasDataSet, 173
HighLoad, 144
INVALID_LOG_FILES, 356
IP_ADDRESS, 84, 283
JOB_RENICE_INCREMENT, 108, 133, 321
JOB_START_DELAY, 105, 307
KBDD, 75
KEEP_POOL_HISTORY, 111, 191
KERBEROS_MAP_FILE, 158
KILLING_TIMEOUT, 142, 143
KILL, 149
KeyboardBusy, 144
LIB, 86
LOCAL_CONFIG_FILE, 73, 82, 87, 88, 180–182, 258, 317, 321
LOCAL_DIR, 71, 74, 75, 86–88, 179
LOCAL_ROOT_CONFIG_FILE, 73, 83, 321
LOCK, 72, 75, 88, 90, 106, 107
LOG, 87, 90, 110, 175
LOWPORT, 89
LastCkpt, 144
MAIL, 88, 181, 182
MASTER_ADDRESS_FILE, 100
MASTER_BACKOFF_CEILING, 99, 100
MASTER_BACKOFF_FACTOR, 99
MASTER_CHECK_INTERVAL, 110
MASTER_CHECK_NEW_EXEC_INTERVAL, 99
MASTER_DEBUG, 100
MASTER_EXPRS, 100
MASTER_NEW_BINARY_DELAY, 99
MASTER_RECOVER_FACTOR, 99, 100
MASTER_UPDATE_INTERVAL, 99
MATCH_TIMEOUT, 140, 143
MAX_ACCOUNTANT_DATABASE_SIZE, 112
MAX_CKPT_SERVER_LOG, 118, 187
MAX_DISCARDED_RUN_TIME, 97, 117, 186, 320
MAX_GOODPUT_NETWORK_CAPACITY_PER_JOB, 200
MAX_JOBS_RUNNING, 30, 104, 284, 321
MAX_SCHEDD_LOG, 173
MAX_SHADOW_EXCEPTIONS, 104, 320
MAX_SUBSYS_LEVEL_LOG, 91
MAX_SUBSYS_LOG, 90, 91
MAX_VIRTUAL_MACHINE_TYPES, 103
MEMORY, 102
MachineBusy, 144, 149
MaxSuspendTime, 144
MaxVacateTime, 144
NEGOTIATE_ALL_JOBS_IN_CLUSTER, 107, 292
NEGOTIATOR_DEBUG, 113

NEGOTIATOR_HOST, 86, 106
NEGOTIATOR_INTERVAL, 112, 200
NEGOTIATOR_SOCKET_CACHE_SIZE, 112
NEGOTIATOR_TIMEOUT, 112
NETWORK_CAPACITY_ALLOCATION_LIMIT, 200
NETWORK_CAPACITY_INFO, 113, 124, 200, 202
NETWORK_CAPACITY_RESERVED_FOR_GOODPUT, 200
NETWORK_HORIZON, 200
NETWORK_INTERFACE, 88, 199, 310
NETWORK_ROUTING_INFO, 113, 124, 200, 201
NETWORK_USAGE_HORIZON, 200
NICE_USER_PRIO_FACTOR, 112, 125
NODE, 48
NUM_CPUS, 102, 304
NUM_VIRTUAL_MACHINES_TYPE_<N>, 195
NUM_VIRTUAL_MACHINES, 195
NUM_VIRUAL_MACHINES_TYPE_<N>, 103
NUM_VIRUAL_MACHINES, 103
Node, 405
NonCondorLoadAvg, 144
OBITUARY_LOG_LENGTH, 98
OPSYS, 85, 180, 317
ORGNAME, 115
PATH, 116
PERIODIC_CHECKPOINT, 298
PERIODIC_MEMORY_SYNC, 107, 311
POLLING_INTERVAL, 101, 141, 286, 312
POOL_HISTORY_DIR, 111, 191
POOL_HISTORY_MAX_STORAGE, 111, 191
POOL_HISTORY_SAMPLING_INTERVAL, 112
PREEMPTION_RANK, 113
PREEMPTION_REQUIREMENTS, 36, 113, 126, 362
PREEMPT, 149
PREEN_ADMIN, 110, 356
PREEN_ARGS, 98, 308
PREEN_INTERVAL, 98
PREEN, 98, 110, 308
PRIORITY_HALFLIFE, 35, 112, 125, 126
PUBLISH_OBITUARIES, 98
PVMD, 108
PVMGS, 108
Process, 405
QUERY_TIMEOUT, 111
QUEUE_CLEAN_INTERVAL, 105
QUEUE_SUPER_USERS, 106
RANK, 205, 206
RELEASE_DIR, 73, 74, 86, 179, 181, 182
REMOTE_PRIO_FACTOR, 112, 126
RESERVED_DISK, 19, 88, 128
RESERVED_MEMORY, 102
RESERVED_SWAP, 88, 104
RESERVE_AFS_CACHE, 95
Rank, 309
SBIN, 86, 97, 101, 104, 108
SCHEDD_ADDRESS_FILE, 106
SCHEDD_DEBUG, 91, 106, 313
SCHEDD_EXPRS, 106
SCHEDD_INTERVAL, 104
SCHEDD_LOCK, 106
SECONDARY_COLLECTOR_LIST, 100, 280
SETTABLE_ATTRS..., 92
SETTABLE_ATTRS_CONFIG, 277
SETTABLE_ATTRS_PERMISSION_LEVEL, 172
SETTABLE_ATTRS, 92, 172
SHADOW_DEBUG, 107
SHADOW_GLOBUS, 51, 109
SHADOW_LIST, 277
SHADOW_LOCK, 107
SHADOW_LOG, 34
SHADOW_PVM, 104
SHADOW_RENICE_INCREMENT, 105
SHADOW_SIZE_ESTIMATE, 105
SHADOW, 90, 104
SHELL, 384
SHUTDOWN_FAST_TIMEOUT, 99
SHUTDOWN_GRACEFUL_TIMEOUT, 92

SHUTDOWN_RUNTIME_STANDARD, 123
SHUTDOWN_RUNTIME, 124
SHUTDOWN, 113, 122–124
SLOW_CKPT_SPEED, 107, 311
SOFT_UID_DOMAIN, 77, 94, 317
SPOOL, 87, 97, 110
STARTD_ADDRESS_FILE, 102
STARTD_ARGS, 98
STARTD_AVAIL_CONFIDENCE, 103
STARTD_COMPUTE_AVAIL_STATS, 103, 104
STARTD_DEBUG, 101, 173, 285, 313
STARTD_EXPRS, 101, 102, 121, 173, 190, 201, 298
STARTD_HAS_BAD_UTMP, 101
STARTD_JOB_EXPRS, 101
STARTD_LOG, 89
STARTD_MAX_AVAIL_PERIOD_SAMPLES, 104
STARTD_SETTABLE_ATTRS_OWNER, 277
STARTD, 122
STARTER_CHOOSES_CKPT_SERVER, 96, 119, 188, 314
STARTER_DEBUG, 108
STARTER_LIST, 277
STARTER_LOCAL_LOGGING, 108, 314
STARTER, 101
START_DAEMONS, 99
START_HAS_BAD_UTMP, 286
START_MASTER, 98, 99
START, 205
STATSDIR, 115, 116
SUBMIT_EXPRS, 93
SUBMIT_SKIP_FILECHECK, 110
SUBSYSTEM, 85, 318
SUBSYS_ADDRESS_FILE, 92, 100, 102, 110
SUBSYS_ARGS, 98, 308
SUBSYS_DEBUG, 90, 100, 101, 107, 112, 113
SUBSYS_EXPRS, 93, 100, 101, 172
SUBSYS_LEVEL_LOG, 91
SUBSYS_LOCK, 90
SUBSYS_LOG, 89, 90
SUBSYS_SETTABLE_ATTRS, 92
SUBSYS, 97
SUSPEND, 260
StartIdleTime, 144
StateTimer, 144
TILDE, 71, 84, 86
TRUNC_SUBSYS_LEVEL_LOG_ON_OPEN, 91
TRUNC_SUBSYS_LOG_ON_OPEN, 90, 91
UID_DOMAIN, 77, 85, 89, 93–96, 152, 198, 278, 400
UPDATE_INTERVAL, 101, 138, 312
USER_JOB_WRAPPER, 108, 309
USE_AFS, 96
USE_CKPT_SERVER, 97, 119, 188, 296, 298
USE_NFS, 95, 96
VACATE, 149
VALID_LOG_FILES, 90, 106, 107, 110, 175, 356
VALID_SPOOL_FILES, 110, 356
VIEWDIR, 115, 116
VIRTUAL_MACHINES_CONNECTED_TO_CONSOLE, 102, 196
VIRTUAL_MACHINES_CONNECTED_TO_KEYBOARD, 102, 196
VIRTUAL_MACHINE_TYPE_<N>, 194
VIRUAL_MACHINE_TYPE_<N>, 103
VOS_PATHNAME, 310
WALL_CLOCK_CKPT_INTERVAL, 105, 297
WANT_*, 149
WANT_SUSPEND, 149
WANT_VACATE, 149
WorkHours, 148, 149
_RUNTIME, 123
_STANDARD, 123
buffer_block_size, 110
buffer_size, 110
cluster, 56
macro_name, 405
ALLOW_ADMINISTRATOR, 164
ALLOW_CONFIG, 164
ALLOW_DAEMON, 164
ALLOW_NEGOTIATOR, 164

ALLOW_OWNER, 164
ALLOW_READ, 164
ALLOW_WRITE, 164
DENY_ADMINISTRATOR, 164
DENY_CONFIG, 164
DENY_DAEMON, 164
DENY_NEGOTIATOR, 164
DENY_OWNER, 164
DENY_READ, 164
DENY_WRITE, 164
SEC_ADMIN_AUTHENTICATION_METHODS, 157
SEC_ADMIN_AUTHENTICATION, 156
SEC_ADMIN_CRYPTO_METHODS, 159
SEC_ADMIN_ENCRYPTION, 158
SEC_ADMIN_INTEGRITY, 160
SEC_CLIENT_AUTHENTICATION_METHODS, 157
SEC_CLIENT_AUTHENTICATION, 156
SEC_CLIENT_CRYPTO_METHODS, 159
SEC_CLIENT_ENCRYPTION, 158
SEC_CLIENT_INTEGRITY, 160
SEC_CONFIG_AUTHENTICATION_METHODS, 157
SEC_CONFIG_AUTHENTICATION, 156
SEC_CONFIG_CRYPTO_METHODS, 159
SEC_CONFIG_ENCRYPTION, 158
SEC_CONFIG_INTEGRITY, 160
SEC_DAEMON_AUTHENTICATION_METHODS, 157
SEC_DAEMON_AUTHENTICATION, 156
SEC_DAEMON_CRYPTO_METHODS, 159
SEC_DAEMON_ENCRYPTION, 158
SEC_DAEMON_INTEGRITY, 160
SEC_DEFAULT_AUTHENTICATION_METHODS, 157
SEC_DEFAULT_AUTHENTICATION, 156
SEC_DEFAULT_CRYPTO_METHODS, 159
SEC_DEFAULT_ENCRYPTION, 158
SEC_DEFAULT_INTEGRITY, 160
SEC_NEGOTIATOR_AUTHENTICATION_METHODS, 157
SEC_NEGOTIATOR_AUTHENTICATION, 156
SEC_NEGOTIATOR_CRYPTO_METHODS, 159
SEC_NEGOTIATOR_ENCRYPTION, 158
SEC_NEGOTIATOR_INTEGRITY, 160
SEC_OWNER_AUTHENTICATION_METHODS, 157
SEC_OWNER_AUTHENTICATION, 156
SEC_OWNER_CRYPTO_METHODS, 159
SEC_OWNER_ENCRYPTION, 158
SEC_OWNER_INTEGRITY, 160
SEC_READ_AUTHENTICATION_METHODS, 157
SEC_READ_AUTHENTICATION, 156
SEC_READ_CRYPTO_METHODS, 159
SEC_READ_ENCRYPTION, 158
SEC_READ_INTEGRITY, 160
SEC_WRITE_AUTHENTICATION_METHODS, 157
SEC_WRITE_AUTHENTICATION, 156
SEC_WRITE_CRYPTO_METHODS, 159
SEC_WRITE_ENCRYPTION, 158
SEC_WRITE_INTEGRITY, 160
Console activity, 248
CONSOLE_DEVICES macro, 80, 101, 181
ContinueIdleTime macro, 144
contrib module
    checkpoint server, 116, 185
    Condor-G, 124, 231
    event daemon, 122
    PVM, 121
CPU activity, 248
CPUBusy macro, 144, 145
CPUBusyTime macro, 145
CPUIdle macro, 144
CPUIsBusy macro, 144
crashes, 270
CREATE_CORE_FILES macro, 89, 319
crontab program, 116

D_SECONDS macro, 276
daemon
condor_ckpt_server, 67
condor_collector, 67
condor_kbdd, 67
condor_master, 66
condor_negotiator, 67
condor_schedd, 66
condor_shadow, 67
condor_startd, 66, 127
condor_starter, 66
eventd, 122
running as root, 62
DAEMON_LIST macro, 75, 97–99, 118, 124, 181, 182, 187, 191, 276, 277, 292, 302
DAEMONNAME_ENVIRONMENT macro, 98, 276
DAGMan, 52–59
describing dependencies, 55
example submit description file, 56
input file example, 52
Job Entry (names node of DAG), 53
job submission, 57
POST script, 54
PRE script, 54
rescue DAG, 58
submit description file with, 56
DC_DAEMON_LIST macro, 97, 292
decimal point
problems with, 262
dedicated machines, 202
dedicated scheduling, 203
DEFAULT_DOMAIN_NAME macro, 88, 199, 317, 319
DEFAULT_IO_BUFFER_BLOCK_SIZE macro, 110, 295
DEFAULT_IO_BUFFER_SIZE macro, 110, 295
DEFAULT_PRIO_FACTOR macro, 112
DEFAULT_PRIORITY_FACTOR macro, 280
DEFAULT_RANK_STANDARD macro, 109
DEFAULT_RANK_VANILLA macro, 109
DEFAULT_UNIVERSE macro, 109, 276, 398
DENY_ADMINISTRATOR macro, 164
DENY_CONFIG macro, 164
DENY_DAEMON macro, 164
DENY_NEGOTIATOR macro, 164
DENY_OWNER macro, 164
DENY_READ macro, 164
DENY_WRITE macro, 164
directed acyclic graph, 52
Directed Acyclic Graph Manager (DAGMan), 52
DISCONNECTED_KEYBOARD_IDLE_BOOST macro, 102, 196
disk space requirement
`execute` directory, 71
`log` directory, 71
`spool` directory, 71
all versions, 74
Condor files, 72
distributed ownership
of machines, 2
download, 68

effective user priority (EUP), 125
EMAIL_DOMAIN macro, 89, 278
environment variables, 36
CONDOR_IDS, 71, 88
in submit description file, 406
ETC macro, 181
event daemon, 122
example configuration, 123
EVENT_LIST macro, 113, 123
EVENTD_CAPACITY_INFO macro, 113, 124
EVENTD_INTERVAL macro, 113, 122
EVENTD_MAX_PREPARATION macro, 113
EVENTD_ROUTING_INFO macro, 113, 124
EVENTD_SHUTDOWN_CLEANUP_INTERVAL macro, 113
EVENTD_SHUTDOWN_SLOW_START_INTERVAL macro, 113
EXEC_TRANSFER_ATTEMPTS macro, 108, 281
execute machine, 65
EXECUTE macro, 19, 87, 88, 128
execution environment, 36
expression
RANK, 133
START, 132

FAQ, 252–271

file
locking, 4, 13
memory-mapped, 4, 13
read only, 4, 13
submit description, 15
write only, 4, 13
file system
AFS, 61
NFS, 62
FILESYSTEM_DOMAIN macro, 77, 85, 94–96, 198, 310
FLOCK_COLLECTOR_HOSTS macro, 106
FLOCK_FROM macro, 192
FLOCK_NEGOTIATOR_HOSTS macro, 106, 192
FLOCK_TO macro, 192
FLOCK_VIEW_SERVER macro, 106
FLOCK_VIEW_SERVERS macro, 106
flocking
definition of, 192
Frequently Asked Questions, 252–271
FS_PATHNAME macro, 310
FULL_HOSTNAME macro, 77, 84, 88, 89

Globus, 50, 271
GLOBUSRUN macro, 51, 109
GRIDMANAGER_LOG macro, 231

HAS_AFS macro, 95
HasDataSet macro, 173
heterogeneous pool
submitting a job to, 25
High-Performance Computing (HPC), 1
High-Throughput Computing (HTC), 1
HighLoad macro, 144
HIGHPORT macro, 89
HISTORY macro, 88, 110, 280
HOST_ALLOW_ADMINISTRATOR macro, 175
HOSTALLOW macro, 92
HOSTALLOW... macro, 92
HOSTALLOW_READ macro, 171
HOSTALLOW_WRITE macro, 51
HOSTDENY macro, 92
HOSTNAME macro, 71, 78, 79, 84, 88
HPC (High-Performance Computing), 1
HTC (High-Throughput Computing), 1

installation
checkpoint server, 116, 185
Condor-G contrib module, 124, 231
CondorView Client, 114
download, 68
Java, 206
PVM contrib module, 121
running as root, 70
scripts, 75
Windows NT, 245–251
INVALID_LOG_FILES macro, 356
IP_ADDRESS macro, 84, 283

Java, 12, 37, 206
Java Virtual Machine, 12, 37, 206
job
analysis, 32
batch ready, 10
completion, 34
dependencies, 52
exiting
signal 9 $Unix$, 259
status 128 NT, 262
heterogeneous submit, 25
multiple data sets, 2
not running, 32
preparation, 10
priority, 32, 35
submitting, 15
JOB_RENICE_INCREMENT macro, 108, 133, 321
JOB_START_DELAY macro, 105, 307
JVM, 12, 37, 206

KBDD macro, 75
KEEP_POOL_HISTORY macro, 111, 191
KERBEROS_MAP_FILE macro, 158
KeyboardBusy macro, 144
KILL macro, 149
KILLING_TIMEOUT macro, 142, 143

LastCkpt macro, 144
LIB macro, 86
linking
dynamic, 5, 13

static, 5, 13
LOCAL_CONFIG_FILE macro, 73, 82, 87, 88, 180–182, 258, 317, 321
LOCAL_DIR macro, 71, 74, 75, 86–88, 179
LOCAL_ROOT_CONFIG_FILE macro, 73, 83, 321
LOCK macro, 72, 75, 88, 90, 106, 107
LOG macro, 87, 90, 110, 175
LOWPORT macro, 89

machine
central manager, 65
checkpoint server, 65
execute, 65
owner, 64
submit, 65
machine activity, 135
Benchmarking, 135
Busy, 136
Idle, 135
Killing, 137
Suspended, 136
transitions, 138–143
transitions summary, 143
Unclaimed, 135
Vacating, 136
machine ClassAd, 9
machine state, 135
Claimed, 135
Matched, 135
Owner, 135
Preempting, 135
transitions, 138–143
transitions summary, 143
Unclaimed, 135
machine state and activities figure, 137
MachineBusy macro, 144, 149
macro
in configuration file, 82
in submit description file, 405
predefined, 48
subsystem names, 85
macro_name macro, 405
MAIL macro, 88, 181, 182
manual installation
Windows NT, 250
master-worker paradigm, 42
MASTER_ADDRESS_FILE macro, 100
MASTER_BACKOFF_CEILING macro, 99, 100
MASTER_BACKOFF_FACTOR macro, 99
MASTER_CHECK_INTERVAL macro, 110
MASTER_CHECK_NEW_EXEC_INTERVAL macro, 99
MASTER_DEBUG macro, 100
MASTER_EXPRS macro, 100
MASTER_NEW_BINARY_DELAY macro, 99
MASTER_RECOVER_FACTOR macro, 99, 100
MASTER_UPDATE_INTERVAL macro, 99
MATCH_TIMEOUT macro, 140, 143
matchmaking, 2
MAX_ACCOUNTANT_DATABASE_SIZE macro, 112
MAX_CKPT_SERVER_LOG macro, 118, 187
MAX_DISCARDED_RUN_TIME macro, 97, 117, 186, 320
MAX_GOODPUT_NETWORK_CAPACITY_PER_JOB macro, 200
MAX_JOBS_RUNNING macro, 30, 104, 284, 321
MAX_SCHEDD_LOG macro, 173
MAX_SHADOW_EXCEPTIONS macro, 104, 320
MAX_SUBSYS_LEVEL_LOG macro, 91
MAX_SUBSYS_LOG macro, 90, 91
MAX_VIRTUAL_MACHINE_TYPES macro, 103
MaxSuspendTime macro, 144
MaxVacateTime macro, 144
MEMORY macro, 102
migration, 2, 3
MPI, 46–50, 202
Condor configuration, 202

NEGOTIATE_ALL_JOBS_IN_CLUSTER macro, 107, 292
NEGOTIATOR_DEBUG macro, 113
NEGOTIATOR_HOST macro, 86, 106
NEGOTIATOR_INTERVAL macro, 112, 200
NEGOTIATOR_SOCKET_CACHE_SIZE macro, 112

NEGOTIATOR_TIMEOUT macro, 112
network, 4, 13
NETWORK_CAPACITY_ALLOCATION_LIMIT macro, 200
NETWORK_CAPACITY_INFO macro, 113, 124, 200, 202
NETWORK_CAPACITY_RESERVED_FOR_GOODPUT macro, 200
NETWORK_HORIZON macro, 200
NETWORK_INTERFACE macro, 88, 199, 310
NETWORK_ROUTING_INFO macro, 113, 124, 200, 201
NETWORK_USAGE_HORIZON macro, 200
NFS
  interaction with, 62
nice job, 36
NICE_USER_PRIO_FACTOR macro, 112, 125
NODE macro, 48
Node macro, 405
NonCondorLoadAvg macro, 144
NUM_CPUS macro, 102, 304
NUM_VIRTUAL_MACHINES macro, 195
NUM_VIRTUAL_MACHINES_TYPE_<N> macro, 195
NUM_VIRUAL_MACHINES macro, 103
NUM_VIRUAL_MACHINES_TYPE_<N> macro, 103

OBITUARY_LOG_LENGTH macro, 98
opportunistic scheduling, 202
OPSYS macro, 85, 180, 317
ORGNAME macro, 115
overview, 1–4

Parallel Virtual Machine (PVM), 41
PATH macro, 116
PERIODIC_CHECKPOINT macro, 298
PERIODIC_MEMORY_SYNC macro, 107, 311
Perl module, 218
  examples, 221
Personal Condor, 247, 255
policy
  at UW-Madison, 146
  default with Condor, 144
  time of day, 148
  version differences, 149
POLLING_INTERVAL macro, 101, 141, 286, 312
pool of machines, 64
POOL_HISTORY_DIR macro, 111, 191
POOL_HISTORY_MAX_STORAGE macro, 111, 191
POOL_HISTORY_SAMPLING_INTERVAL macro, 112
PREEMPT macro, 149
preemption
  of resources, 126
PREEMPTION_RANK macro, 113
PREEMPTION_REQUIREMENTS macro, 36, 113, 126, 362
PREEN macro, 98, 110, 308
PREEN_ADMIN macro, 110, 356
PREEN_ARGS macro, 98, 308
PREEN_INTERVAL macro, 98
priority
  in machine allocation, 124
  nice job, 36
  of a job, 32, 35
  of a user, 35
PRIORITY_HALFLIFE macro, 35, 112, 125, 126
Process macro, 405
PUBLISH_OBITUARIES macro, 98
PVM
  machine class, 43
  master-worker paradigm, 42
  PvmHostSuspend and PvmHostResume notifications, 43
  submit description file, 45
PVM (Parallel Virtual Machine), 41–46
PVM contrib module, 121
PVMD macro, 108
PVMGS macro, 108

QUERY_TIMEOUT macro, 111
QUEUE_CLEAN_INTERVAL macro, 105
QUEUE_SUPER_USERS macro, 106

rank attribute, 17
  examples, 23
RANK macro, 205, 206

Rank macro, 309
real user priority (RUP), 125
recovery from crashes, 270
RELEASE_DIR macro, 73, 74, 86, 179, 181, 182
remote system call, 2, 3, 12
    condor_shadow, 12, 30, 62
REMOTE_PRIO_FACTOR macro, 112, 126
requirements attribute, 17
RESERVE_AFS_CACHE macro, 95
RESERVED_DISK macro, 19, 88, 128
RESERVED_MEMORY macro, 102
RESERVED_SWAP macro, 88, 104
resource
    management, 2
    offer, 3
    owner, 64
    request, 3
running a job
    at certain times of day, 257
    on a different architecture, 25
    on only certain machines, 257
    only at night, 257
running multiple programs, 17

SBIN macro, 86, 97, 101, 104, 108
SCHEDD_ADDRESS_FILE macro, 106
SCHEDD_DEBUG macro, 91, 106, 313
SCHEDD_EXPRS macro, 106
SCHEDD_INTERVAL macro, 104
SCHEDD_LOCK macro, 106
scheduling
    dedicated, 47, 203
    opportunistic, 202
SDK
    Chirp, 39
SEC_ADMIN_AUTHENTICATION macro, 156
SEC_ADMIN_AUTHENTICATION_METHODS macro, 157
SEC_ADMIN_CRYPTO_METHODS macro, 159
SEC_ADMIN_ENCRYPTION macro, 158
SEC_ADMIN_INTEGRITY macro, 160
SEC_CLIENT_AUTHENTICATION macro, 156
SEC_CLIENT_AUTHENTICATION_METHODS macro, 157
SEC_CLIENT_CRYPTO_METHODS macro, 159
SEC_CLIENT_ENCRYPTION macro, 158
SEC_CLIENT_INTEGRITY macro, 160
SEC_CONFIG_AUTHENTICATION macro, 156
SEC_CONFIG_AUTHENTICATION_METHODS macro, 157
SEC_CONFIG_CRYPTO_METHODS macro, 159
SEC_CONFIG_ENCRYPTION macro, 158
SEC_CONFIG_INTEGRITY macro, 160
SEC_DAEMON_AUTHENTICATION macro, 156
SEC_DAEMON_AUTHENTICATION_METHODS macro, 157
SEC_DAEMON_CRYPTO_METHODS macro, 159
SEC_DAEMON_ENCRYPTION macro, 158
SEC_DAEMON_INTEGRITY macro, 160
SEC_DEFAULT_AUTHENTICATION macro, 156
SEC_DEFAULT_AUTHENTICATION_METHODS macro, 157
SEC_DEFAULT_CRYPTO_METHODS macro, 159
SEC_DEFAULT_ENCRYPTION macro, 158
SEC_DEFAULT_INTEGRITY macro, 160
SEC_NEGOTIATOR_AUTHENTICATION macro, 156
SEC_NEGOTIATOR_AUTHENTICATION_METHODS macro, 157
SEC_NEGOTIATOR_CRYPTO_METHODS macro, 159
SEC_NEGOTIATOR_ENCRYPTION macro, 158
SEC_NEGOTIATOR_INTEGRITY macro, 160
SEC_OWNER_AUTHENTICATION macro, 156
SEC_OWNER_AUTHENTICATION_METHODS macro, 157
SEC_OWNER_CRYPTO_METHODS macro, 159

SEC_OWNER_ENCRYPTION macro, 158
SEC_OWNER_INTEGRITY macro, 160
SEC_READ_AUTHENTICATION macro, 156
SEC_READ_AUTHENTICATION_METHODS macro, 157
SEC_READ_CRYPTO_METHODS macro, 159
SEC_READ_ENCRYPTION macro, 158
SEC_READ_INTEGRITY macro, 160
SEC_WRITE_AUTHENTICATION macro, 156
SEC_WRITE_AUTHENTICATION_METHODS macro, 157
SEC_WRITE_CRYPTO_METHODS macro, 159
SEC_WRITE_ENCRYPTION macro, 158
SEC_WRITE_INTEGRITY macro, 160
SECONDARY_COLLECTOR_LIST macro, 100, 280
security
  in Condor, 149–173
  access levels, 154
  authentication, 156, 161
  authorization, 163
  based on user authorization, 163
  encryption, 158
  host-based, 166
  integrity, 160
SETTABLE_ATTRS macro, 92, 172
SETTABLE_ATTRS... macro, 92
SETTABLE_ATTRS_CONFIG macro, 277
SETTABLE_ATTRS_PERMISSION-LEVEL macro, 172
shadow, 12
SHADOW macro, 90, 104
SHADOW_DEBUG macro, 107
SHADOW_GLOBUS macro, 51, 109
SHADOW_LIST macro, 277
SHADOW_LOCK macro, 107
SHADOW_LOG macro, 34
SHADOW_PVM macro, 104
SHADOW_RENICE_INCREMENT macro, 105
SHADOW_SIZE_ESTIMATE macro, 105
SHELL macro, 384
SHUTDOWN macro, 113, 122–124
SHUTDOWN_FAST_TIMEOUT macro, 99
SHUTDOWN_GRACEFUL_TIMEOUT macro, 92
SHUTDOWN_RUNTIME macro, 124
SHUTDOWN_RUNTIME_STANDARD macro, 123
signal, 4, 13
  SIGTSTP, 4, 13
  SIGUSR2, 4, 13
SLOW_CKPT_SPEED macro, 107, 311
SOFT_UID_DOMAIN macro, 77, 94, 317
Software Developer's Kit
  Chirp, 39
Solaris26, 260
Solaris27, 260
SPOOL macro, 87, 97, 110
START expression, 127
START macro, 205
START_DAEMONS macro, 99
START_HAS_BAD_UTMP macro, 286
START_MASTER macro, 98, 99
startd
  configuration, 127
STARTD macro, 122
STARTD_ADDRESS_FILE macro, 102
STARTD_ARGS macro, 98
STARTD_AVAIL_CONFIDENCE macro, 103
STARTD_COMPUTE_AVAIL_STATS macro, 103, 104
STARTD_DEBUG macro, 101, 173, 285, 313
STARTD_EXPRS macro, 101, 102, 121, 173, 190, 201, 298
STARTD_HAS_BAD_UTMP macro, 101
STARTD_JOB_EXPRS macro, 101
STARTD_LOG macro, 89
STARTD_MAX_AVAIL_PERIOD_SAMPLES macro, 104
STARTD_SETTABLE_ATTRS_OWNER macro, 277
STARTER macro, 101
STARTER_CHOOSES_CKPT_SERVER macro, 96, 119, 188, 314
STARTER_DEBUG macro, 108
STARTER_LIST macro, 277
STARTER_LOCAL_LOGGING macro, 108, 314
StartIdleTime macro, 144

state
of a machine, 135
transitions, 138–143
transitions summary, 143
state and activities figure, 137
StateTimer macro, 144
STATSDIR macro, 115, 116
status
of queued jobs, 29
submit description file, 15
contents of, 15
examples, 15–17
for PVM application, 45
globus universe, 232
submit machine, 65
submit-only installation, 79
SUBMIT_EXPRS macro, 93
SUBMIT_SKIP_FILECHECK macro, 110
substitution macro
in submit description file, 405
SUBSYS macro, 97
SUBSYS_ADDRESS_FILE macro, 92, 100, 102, 110
SUBSYS_ARGS macro, 98, 308
SUBSYS_DEBUG macro, 90, 100, 101, 107, 112, 113
SUBSYS_DEBUG macro levels
D_COMMAND, 90
D_DAEMONCORE, 90
D_FULLDEBUG, 90
D_JOB, 91
D_KEYBOARD, 91
D_LOAD, 91
D_MACHINE, 91
D_MATCH, 91
D_NETWORK, 91
D_PRIV, 90
D_SYSCALLS, 91
SUBSYS_EXPRS macro, 93, 100, 101, 172
SUBSYS_LEVEL_LOG macro, 91
SUBSYS_LOCK macro, 90
SUBSYS_LOG macro, 89, 90
SUBSYS_SETTABLE_ATTRS macro, 92
SUBSYSTEM macro, 85, 318
subsystem names, 85
SUSPEND macro, 260

thread
kernel-level, 4, 13
user-level, 4, 13
TILDE macro, 71, 84, 86
TRUNC_SUBSYS_LEVEL_LOG_ON_OPEN macro, 91
TRUNC_SUBSYS_LOG_ON_OPEN macro, 90, 91

UID
effective, 150
real, 150
UID_DOMAIN macro, 77, 85, 89, 93–96, 152, 198, 278, 400
UIDs in Condor, 150–151
universe, 11
Globus, 12, 14, 50, 229
Java, 12, 15
MPI, 12, 14
PVM, 12, 14
standard, 12
vanilla, 12, 14
Unix
alarm, 4, 13
exec, 4, 13
flock, 4, 13
fork, 4, 13
lockf, 4, 13
mmap, 4, 13
pipe, 4, 13
semaphore, 4, 13
shared memory, 4, 13
sleep, 4, 13
socket, 4, 13
system, 4, 13
timer, 4, 13
Unix administrator, 70
Unix administrator's manual, 64–207
Unix daemon
running as root, 62
Unix directory
execute, 71
lock, 72
log, 71
spool, 71
Unix installation

download, 68
scripts, 75
submit-only, 79
Unix user
condor, 71
root, 70
UPDATE_INTERVAL macro, 101, 138, 312
upgrade
version 6.2.x to version 6.4.x, 253
upgrading
6.0 to 6.2, 271
USE_AFS macro, 96
USE_CKPT_SERVER macro, 97, 119, 188, 296, 298
USE_NFS macro, 95, 96
user
priority, 35
user manual, 7–63
user priority, 124
effective (EUP), 125
real (RUP), 125
USER_JOB_WRAPPER macro, 108, 309

vacate, 60
VACATE macro, 149
VALID_LOG_FILES macro, 90, 106, 107, 110, 175, 356
VALID_SPOOL_FILES macro, 110, 356
vanilla jobs
cycling between suspended and unsuspended, 260
VIEWDIR macro, 115, 116
VIRTUAL_MACHINE_TYPE_<N> macro, 194
VIRTUAL_MACHINES_CONNECTED_TO_CONSOLE macro, 102, 196
VIRTUAL_MACHINES_CONNECTED_TO_KEYBOARD macro, 102, 196
VIRUAL_MACHINE_TYPE_<N> macro, 103
VOS_PATHNAME macro, 310

WALL_CLOCK_CKPT_INTERVAL macro, 105, 297
WANT_* macro, 149
WANT_SUSPEND macro, 149
WANT_VACATE macro, 149
Windows NT
Condor daemon names, 251
installation, 245–251
initial file size, 245
location of files, 247
Personal Condor, 247
preparation, 245
required disk space, 246
introduction, 239
manual install, 249
release notes, 240–244
starting the Condor service, 250–251
WorkHours macro, 148, 149

Y2K, 270