PATH PASCAL USER MANUAL

Robert B. Kolstad & Roy H. Campbell
Deptartment of Computer Science
University of Illinois
Urbana, Illinois 61801

1 Introduction.

Experimental Path Pascal was designed to investigate the benefits and problems that arise when Path Expressions are combined with a language to provide a system programming tool. Instead of altering the Pascal language extensively, a minimal number of features was added such that Pascal programs still compile and execute. The language can be used for instruction or construction of example system programs. This manual describes the Path Pascal features and the implementation on the Cyber and PDP-11.

Path Expressions were introduced as a technique for specifying process synchronization by [Campbell & Habermann, 74], and further discussed by [Habermann, 75], [Lauer & Campbell, 75], [Flon & Habermann, 76], [Andler, 79] and [Campbell, 77]. Variations of the Path Expression idea have been proposed by [ONERA CERT, 77] and notations that are similar to paths that model system behavior have been developed independently by [Shaw, 77] and [Riddle, 76]. A specification language has also been designed [Lauer & Shields, 78] based upon the use of a Path Expression notation.

Path Pascal is based on the P4 subset of Pascal [Ammann, et al., 76] (see Appendix F for a summary of the P4 subset). The Path Pascal compiler is written in Pascal P4 and accepts any Pascal P4 program that does not use Path Pascal reserved words as identifiers. Pascal was augmented with an encapsulation mechanism (see chapter 2), Open Path Expressions [Campbell, 77] (see chapter 3), and a process mechanism (see chapter 4). Open Paths are integrated with the encapsulation mechanism to enforce a strict discipline upon the programmer to describe shared data objects. All access to encapsulated data is performed by operations synchronized by Open Paths. A process invoking such operations may execute the operation only if permitted by the Open Path Expression associated with the shared data object.

The following chapters describe Path Pascal in more detail. Motivations for the design of Path Pascal are discussed further in [Miller, 78], [Campbell & Kolstad, 79a], [Campbell & Kolstad, 79b], [Campbell & Kolstad, 80], [Horton & Campbell, 80], and [Kolstad & Campbell, 80]. A description of Pascal can be found in the Pascal Report [Jensen & Wirth, 75]. The additional Path Pascal syntax is listed in Appendix A. Appendix B contains error messages, control options and constants for the Path Pascal P code interpreter. Appendix C describes the semantics of Open Path Expressions in terms of P and V operations. Appendix D contains several sample programs. Appendix E describes the changes that have been made to the intermediate code (P-Code) for the additional Path Pascal constructs. Appendix F summarizes the differences between Path Pascal and Pascal P4.

2 Data Encapsulation.

2.1 Introduction to Objects.

Encapsulating data and definitions of operations on that data ensures that only intended accesses and transformations are made to an information structure. The addition of a synchronization mechanism to data encapsulation allows protection from asynchronous access. In Path Pascal, an encapsulation mechanism called an object specifies access, transformation, and synchronization. An object's data and code are accessible to other parts of the Pascal programs only by explicit declaration of entry types and entry operations. Objects are implemented as an extension of the Pascal structured type facility.

2.2 Object Declaration.

Each object begins with the declarator object, then specifies the synchronization for the object via a Path Expression (see chapter 3), followed by const declarations if needed, type declarations if needed, var declarations if needed, the routines of the object (routines can be an initialization block, procedures, functions, processes, or exported procedures, processes, and functions) in appropriate order for scope consideration, and finally an end token. The const, type, var, and routine specifications are expressed as in standard Pascal and have the same actions.

The object defines a block which follows the scope rules of standard Pascal; though exported procedures, functions, processes, and types have the additional attribute of appearing as defined in the scope containing the object. Only exported procedures, processes, and functions are available to enclosing scopes for examination and manipulation of encapsulated data.

Object types may be declared with explicit names in a type statement or implicitly (along with instantiation) using the var statement. Object names defined as types may be used to declare any number of object instantiations in var statements. Once instantiated, each object has its own copies of storage, the object's operations, and synchronization information.

Objects may be nested within structures or within other objects. Recursive object instantiations, however, are flagged as errors during compilation.

Pointers to objects are declared in var statements similar to declarations of pointers to other data types. Dynamic instantiations may be created by executing the standard procedure named 'new' with a pointer argument. Pointers to objects permit the construction of encapsulated and recursive data structures.

2.3 Operations.

Functions, processes, and procedures whose names are exported from an object are known as "operations". They are differentiated from

internal procedures, processes, and functions by prefixing their declaration by the token entry. Operations, like all routines within an object, can invoke other operations and routines within the object (as long as scope considerations are satisfied). Synchronization is applied as usual for invoked operations.

Operations within an object are invoked as standard procedures. Outside the object, however, the name of the object's instantiation (or a dereferenced pointer to the object's instantiation) and a period must precede the name of the operation to be invoked. Operations may be invoked recursively, even though a deadlock might eventually result.

2.4 Exported Types.

An object may export names of types in addition to names of functions, processes, and procedures. Variables may then be declared to be of such types, though no examination of the internal structure or representation of the type is possible. A type to be exported is declared with the word entry between the "=" and normal type declaration. Simple types may not be exported. Variables of exported types may be defined both inside and outside an object, and passed as arguments into and out of objects, but may not be examined or manipulated outside the object, since their structure is unknown.

2.5 Path Declaration.

The object's Path Expression specifies the synchronization constraints of the object's operations. Each operation's name must be mentioned at least once in the Path Expression. Chapter 3 discusses Path Expressions in detail.

2.6 Initialization Block.

The initialization part of an object is an optional block of code which is executed upon instantiation of the object. Labels, constants, types, variables, and routines may be declared within an initialization block. Standard scope rules apply. An initialization block may appear anywhere within an object's routine declarations.

An initialization block is composed of the token init followed by a semicolon and the tokens begin and end surrounding the block (of declarations and code) to be executed when the object is created. The use within an init block of variables and routines global to the object is discouraged. The init blocks of object variables nested within other objects are executed before the blocks of the surrounding objects.

2.7 Implementation Details.

Assignments between variables containing objects are not permitted. Object variables or structured variables containing objects are always passed as reference parameters to routines.

2.8 Examples.

The example below shows the declaration of a typical object type, its instantiation, and two invocations:

```
const   nbuf = 5;
type
    bufrange = 1..5;          (* 5 = nbuf *)
    ring = object

        path nbuf:(1:(put); 1:(get)) end;

        var buffer: array[bufrange] of char;
            inp, outp: bufrange;

        entry procedure put(x: char);
            begin
                inp := (inp mod nbuf) + 1;
                buffer[inp] := x
            end;

        entry function get: char;
            begin
                outp := (outp mod nbuf) + 1;
                get := buffer[outp]
            end;

        init; begin
            inp := nbuf;
            outp := nbuf
        end;
    end;

var buf: ring;
    c: char;

begin
    buf.put('a');
    c := buf.get
end.
```

The initialization block sets the pointers to appropriate values for standard ring buffering. The operation 'put' is called to deposit characters within the buffer, 'get' retrieves them. The Path Expression eliminates need for any further synchronization specification.

2.9 Syntax.

Backus Naur Form for each of the new specifications is shown below:

```
obj_type        ::=  "object" <path_decl_part>
                     <const_defn_part>
                     <obj_typdef_pt>
                     <var_decl_part>
                     <operation_part> "end"

obj_typdef_pt   ::=  <obj_type_defn> { ";"
                     <obj_type_defn> } |
                     <empty>

obj_type_defn   ::=  <type_defn> |
                     <ident> "=" "entry" <type>
```

```
operation_part  ::= { <routine> ";" } |
                    { <routine> ";" } "init" ";"
                    <block> { <routine> ";" }

routine         ::= <pp_or_f_decl> |
                    <opn_decl>

pp_or_f_decl    ::= <proc_decl> |
                    <func_decl> |
                    <procs_decl> |
                    <intrp_decl>

opn_decl        ::= "entry" <pp_or_f_decl>
```

3 Path Expressions.

3.1 Introduction to Path Expressions.

An Open Path Expression specifies the synchronization constraints for a possibly concurrent set of process, procedure, and function executions within objects. This static description allows code to be written without any explicit reference to synchronization primitives. Each object contains one Path Expression to specify the allowed orders of sequential and concurrent execution of the object's entry operations. Since only the entry operations can be accessed from outside the object, the information structure can be completely protected from unsafe sequences.

Normally, the order of invocation of procedures is unknown until the invocation occurs since processes can execute asynchronously. Path Expressions allow three distinct kinds of constraints to be specified: sequencing (denoted by ';'), resource restriction (denoted by 'n:( )'), and resource derestriction (denoted by '[ ]'). Each of these can be combined with the other forms to provide complex synchronization constraints and several constraints can be contained in a single Path Expression. These forms are described with examples below.

A Path with no synchronization information consists of a comma separated list of operation names surrounded by path and end. The Path below:

path name1, name2, name3 end

imposes no restriction on the order of invocation of the operations and no restriction on the number of concurrent executions of 'name1', 'name2', and 'name3'.

The sequencing mechanism imposes an order on procedure executions. The order is specified by a semi-colon separated list. In the example below:

path first; second; third end

one execution of operation 'first' must complete before each execution of 'second' may begin, and one execution of 'second' must complete before each execution of 'third' can begin. Of course, the execution of a 'third' or 'second' in no way inhibits the initiation of 'first'; several operations may be executing concurrently.

Limited resources (e.g., line printers) occasionally make it desirable to limit the number of concurrent executions of an operation. The resource restriction specification allows concurrent execution of operations to proceed until the restriction limit is reached. Restrictions are denoted by surrounding the expression to be restricted by parentheses and preceding it with the integer restriction limit and a colon. The restriction below:

path 2:(ttyhandler) end

allows only two invocations of 'ttyhandler' to proceed concurrently. Any invocation of 'ttyhandler' will wait until less than two executions are active before it begins execution. The number preceding the colon in a restrictor can be thought of as the number of resources for which the operation competes. A critical section, in which only a single resource is to be shared, is easily specified. In the example below:

path 1:(routine1, routine2, routine3) end

only one of the three operations can be active at a time. Restrictors may be positive integers or positive constants.

For some applications it is convenient to process all calls to an operation once that operation's execution has begun. Such a situation might occur when a large spooler is brought into memory to process I/O requests. The specifier denoting "derestriction" of a list of operations is shown by surrounding the list in square brackets. The path below:

path setup; [spooler] end

requires 'setup' to be executed before each sequence of calls to 'spooler', but once 'spooler' has begun execution, its invocations proceed to execution until all executions have terminated. Afterwards, 'setup' must again complete before any 'spooler' can proceed.

Each of the forms above (without path and end) can be considered to be a subexpression of a Path. Subexpressions may be combined (with the optional use of parentheses for clarity) in the formats above to yield complex paths. Normally, the sequencing operator (";") has higher precedence than the alternation operator (","). An operation name may be repeated within a path in which case the synchronization constraints for each occurrence of the operation are applied in the order from left to right.

3.2 Examples of Open Paths.

1. path a end;

   Routine 'a' can execute at any time, and any number of 'a's can execute concurrently. No synchronization is specified.

2. path a, b, c end;

   Routines 'a', 'b', and 'c' can execute at any time. Any number of each one can execute concurrently. No synchronization is specified.

3. path a; b end;

   Routine 'a' can be executed at any time, but 'b' can only begin if the number of 'b's that have begun execution is less than the number of 'a's that have completed.

4. path 1:(a) end;

   Routine 'a' must be executed sequentially (only one 'a' active at a time).

5. path 2:(a) end;

   At most two executions of routine 'a' can proceed concurrently.

6. path 1:(a), b end;

   Multiple invocations of routine 'a' proceed in sequential execution. No restriction is placed on routine 'b'.

7. path 1:(a), 1:(b) end;

   Both 'a' and 'b' are critical sections. A maximum of one 'a' and one 'b' can execute concurrently.

8. path 6:(5:(a), 4:(b)) end;

   As many as five invocations of 'a' and four of 'b' can proceed concurrently as long as the limit of six total executions is not exceeded.

9. path 5:(a; b) end;

   No more than five executions of routine 'a' and routine 'b' can be proceeding concurrently. Each execution of 'b' must be preceded by an execution completion of 'a'.

10. path 1:([a], [b]) end;

    Routines 'a' and 'b' operate in mutual exclusion. Either is authorized to proceed as long as requests for its execution exist. When the executing routine's request list is exhausted, either routine may start again.

3.3 Syntax.

The BNF syntax for Open Paths is shown below:

```
path_decl   ::= "path" <list> "end"

list        ::= <sequence> { "," <sequence> }

sequence    ::= <item> { ";" <item> }

item        ::= <bound> ":" "(" <list> ")" |
                "[" <list> "]" |
                "(" <list> ")" |
                <ident>

bound       ::= <unsgnd_int> |
                <const>
```

4 Processes.

A process is a program structuring unit which has an independent execution sequence associated with it. Processes can interact and are coordinated by performing operations on shared variables. In Path Pascal, the declaration of a process is separated from its activation. A process may be declared in any block and activations of the process may be created from any body of code with scope that includes the declaration.

Processes are declared in a manner similar to standard Pascal procedures. They may possess parameters (passed by value or by reference) and may also have a size attribute. The optional size attribute is an estimate of the process's storage requirements.

4.1 Instantiation.

An instance of a process is dynamically created by invoking the process name in the same manner as a procedure invocation. The creating process need not wait for the created process to terminate and continues its own execution. Each process created is allocated a run-time heap and stack from the heap of the process which is performing the creation. The number of words allocated is optionally specified by the size attribute. No mechanism is provided to abnormally terminate a process; termination occurs only when the end of a process's code body is reached.

4.2 Process Storage Considerations.

Processes may themselves spawn processes. The storage from any process is acquired from the heap of its parent. It is occasionally desirable to specify a larger of smaller heap for a process than that of the default. This is done by inserting the storage requirement in words between the name of the process and the arguments (if any). An example is:

```
process bigun [500] (arg: int);
```

A process's storage is not automatically released when a process terminates. Although mark and release may be used for storage management, this use is discouraged.

4.3 Process Lifetimes.

The lifetime of a block which contains a process declaration is at least as long as the lifetime of any activation of that process. If an attempt is made to exit a block which contains a process declaration for which there is an existing activation, the exit will be delayed until that process completes.

4.4 Parameter Restriction.

The scope of an actual parameter which is passed by reference to a process must contain scope of the process's declaration (hence storage for the parameter will exist as long as the process does).

4.5 Priorities.

One of two static priority schemes can be associated with processes in order to provide rudimentary control over process scheduling. In the first scheme, all processes have the same priority. In the other, priority is determined by the static nesting level of a process's declaration, with processes declared at the outermost levels having the highest priority. Within a given priority level, a process is selected for execution by a first-in first-out scheduler. The second priority scheme is selected by default, but equal priorities can be chosen by specifying the 'np' option on the interpreter command card.

4.6 Simulated Time.

A process can be delayed for a fixed time interval by calling the procedure 'delay'. Its integer argument specifies how long the process is to be delayed. The number of simulated time units which have elapsed since execution began can be obtained from the parameterless integer function 'time'.

4.7 Interrupt Processes.

Interrupt processes are used in Path Pascal to program input and output devices. The doio statement is used only within interrupt processes and suspends process execution while input or output is being performed.

An interrupt process is declared by preceding a normal process declaration by the token interrupt and succeeding it with the priority and interrupt vector to be assigned both enclosed within square brackets.

A sample output driver for a PDP-11 exemplifies interrupt processes and is shown below:

```
interrupt process print[priority = 4;
                        vector = #64] (buf: buffer);
var
  i: integer;
  pts[#177564]:bits;  (* printer status word *)
  ptb[#177566]:char;  (* printer buffer word *)

begin
  i := 0;
  repeat
    i := i + 1;
    pts := [6];      (* enable printer interrupt *)
    ptb := buf[i];   (* send char to printer *)
    doio;        (* wait for interrupt *)
    pts := pts - [6]; (* disable interupt *)
  until ((i >= linesize) or (buf[i] = cr))
end;
```

Absolute memory locations can be allocated via an extension of the var mechanism which allows easy access to I/O devices on machines with architectures similar to that of the PDP-11. The name of the variable to be allocated is succeeded by the location to be assigned enclosed in square brackets. This location may be expressed in octal if it is preceded by a '#' token.

The bracketed parameters specify the priority of the process and the location of its interrupt vector. In the example above, the vector is stored at location octal 64 (decimal 52) and the priority of the process is 4. (On the PDP-11, the priority of the processor is set to the priority of the process it is running. Interrupts from devices can only affect the process when the process priority is less than the priority of the interrupting device. Other processes normally run with a processor priority of 0.)

Interrupt processes are created in exactly the same manner as other processes. Running duplicate interrupt processes or terminating an interrupt process while an interrupt is pending is discouraged.

4.8 Syntax.

The syntax extensions for interrupt processes are shown below:

```
procs_decl      ::=  <procs_hdg> <block>

procs_hdg       ::=  "process" <ident> <size_part>
                     ";" |
                     "process" <ident> <size_part>
                     "(" <formal_parm_sec> { ";"
                     <formal_parm_sec> } ")" ";"

size_part       ::=  "[" <unsgn_int> "]" |
                     <empty>

intrpt_decl     ::=  <intrpt_procs_hd> <block>

intrpt_procs_hd ::=  "interrupt" "process" <ident>
                     <intrpt_parms> ";" |
                     "interrupt" "process" <ident>
                     <intrpt_parms> "("
                     <formal_parm_scn> { ";"
                     <formal_parm_scn> } ")"

intrpt_parms    ::=  "[" "priority" "=" <unsgn_int>
                     ";" "vector" "=" <addr> "]"
```

```
addr        ::= "#" <unsgn_int> |
                <unsgn_int>
```

5 Summary.

The Path Pascal programming language is an extension of Pascal P4 which includes concurrent processes, processes for controlling I/O devices, Path Expressions, and objects. The Path Pascal compiler is written in Pascal P4 and is self-compiling. An intermediate code (an extended P-Code) is produced by the Path Pascal compiler and can either be executed interpretively or assembled into machine instructions. The language can be used to simulate systems, as an educational tool, or to construct system and real-time programs.

6 References.

[Ammann, et al., 76] Ammann, U., K. Nori, and C. Jacobi, "The Portable Pascal Compiler," Institut Fuer Informatik, EIDG, Technische Hochschule CH-8096, Zurich, 1976.

[Andler, 79] Andler, Sten, "Predicate Path Expressions," 6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, Tex., pp. 226-236, 1979.

[Campbell & Habermann, 74] Campbell, R. H., and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," Lecture Notes in Computer Science (Editors G. Goos and J. Hartmanis), Vol. 16, pp. 89-102, Springer-Verlag, 1974.

[Campbell, 76] Campbell, R. H., "Path Expressions: A technique for specifying process synchronization," Ph.D. Thesis, The University of Newcastle upon Tyne, August, 1976; Also, Department of Computer Science Technical Report, University of Illinois at Urbana-Champaign, UIUCDCS-R-77-863, May, 1977.

[Campbell & Kolstad, 79a] Campbell, R. H. and R. B. Kolstad, "Path Expressions in Pascal," Fourth International Conference on Software Engineering, Munich, September 17-19, 1979.

[Campbell & Kolstad, 79b] Campbell, R. H. and R. B. Kolstad, "Practical Applications of Path Expressions to Systems Programming," ACM79, Detroit, 1979.

[Campbell & Kolstad, 80] Campbell, R. H. and R. B. Kolstad, "A Practical Implementation of Path Pascal," Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-80-1008, 1980.

[Dahl, et al., 68] Dahl, O. J., B. Myhrhaug, and K. Nygaard, "The Simula 67 Common Base Language," Norwegian Computer Center, Oslo, 1968.

[Flon & Habermann, 76] Flon, L. and A. N. Habermann, "Towards the Construction of Verifiable Software Systems," SIGPLAN Notices Vol. 8, No. 2, March, 1976.

[Habermann, 75] Habermann, A. N., "Path Expressions," Department of Computer Science Technical Report, Carnegie-Mellon University, June, 1975.

[Habermann, 76] Habermann, A. N., Introduction to Operating System Design, Science Research Associates, p. 89, 1976.

[Horton & Campbell, 80] Horton, Kurt H. and Roy H. Campbell, "PDP-11 Path Pascal Implementation Manual," Technical Report, University of Illinois at Urbana-Champaign, to be published, 1980.

[Jensen & Wirth, 75] Jensen, K. and N. Wirth, Pascal User Manual and Report, Springer-Verlag, New York, 1975.

[Lauer & Campbell, 75] Lauer, P. E. and R. H. Campbell, "Formal Semantics of a Class of High Level Primitives for Co-ordinating Concurrent Processes," Acta Informatica, No. 5, pp. 297-332, 1975.

[Lauer & Shields, 78] Lauer, P. E. and M. W. Shields, "Abstract Specification of Resource Accessing Disciplines: Adequacy, Starvation, Priority and Interrupts," SIGPLAN Notices, Vol. 13, Number 12, pp. 41-59, 1978.

[Miller, 78] Miller, T. J., "An Implementation of Path Expressions in Pascal," M. S. Thesis, University of Illinois, Urbana, May, 1978.

[ONERA CERT, 78] "Parallelism, Control and Synchronization Expression in a Single Assignment Language," Sigplan Notices Vol. 13, No. 1, January, 1978.

[Riddle, 76] Riddle, W. E., "Software System Modelling and Analysis," RSSM/25, Tech. Report, Department of Computer and Communication Sciences, University of Michigan, July, 1976.

[Shaw, 77] Shaw, A. C., "Software Descriptions with Flow Expressions," IEEE TSE, Vol. 4, No. 3, p. 242-254, May, 1978.

[Wirth, 77] Wirth, N., "Modula: a Language for Modular Multiprogramming," Software-Practice and Experience, Vol. 7, pp. 3-84, 1977.

APPENDIX D
PROGRAMMING EXAMPLES

D.1 NETWORK

A small network simulation program patterned after [Brinch Hansen, 78] is presented below. The network is ring oriented and request-driven. Requests are sent from a processor through the network to a (probably foreign) processor, where a complementary process transmits a reply. This reply is then forwarded to the original processor. Each processor contains a single input link and a single output link. A request/response message pair circumnavigates the ring once in a normal request/respond cycle or twice if the processor attempts communication with itself. This program is presented only to compare and contrast different methods of synchronization specification, not as a solution to data transfer problems.

As presented, the program contains not only a network system, but also a simulation of the machines and physical lines. The program is somewhat shorter than Brinch Hansen's, and refers to synchronization only in the Path Expressions of the objects: semaphores (or conditions), monitors and queues are not required. The programmer can therefore simply invoke routines, knowledgeable of the fact that they are already synchronized correctly.

The program source is shown here:

```
const
  nmax  = 3;            (* three nodes *)
  cmax  = 6;            (* six channels *)
  bmax  = 3;            (* three buffers *)

    (* The constants above define the network
       configuration *)

type
  node    = 1..nmax;
  channel = 1..cmax;
  channelset = set of channel;
  item    = array[1..10] of char;

  message = record
    kind: (request, response);
    link: channel;
    contents: item
  end;

    (* 'item' is the logical atomic data packet
       sent between nodes.  A 'message' contains routing information and the
       'item'.  *)

    (* The 'line' simulates the physical line
       between machines.  Each machine references two different 'line's:  one for
       input, one for output.  *)

  line  = object        (* physical line *)

    path 1:(to_buslink; from_buslink) end;

      (* input must wait for output from elsewhere, only a single output can occur
         before an input *)

    var mesgbuffer: message;

    entry procedure to_buslink(m:message);
      begin
        delay(5);
        mesgbuffer := m
      end;

    entry procedure from_buslink(var m:message);
      begin
        m := mesgbuffer
      end;
end;                    (* line *)

    (* The 'machine' object contains all the
       attributes of a simulated machine.
       These include: 'buffer' operations for
       the physical line; 'inputs', which waits
       for data to be returned after a request
       has been sent; 'outputs', which sends
       the data after requested; 'reader', monitors traffic on line, routing messages
       forward or through request/response
       mechanism; 'writer', which copies messages from the output buffer to the physical line; 'go', forks the processes
       'reader'/'writer' as initialization; and
       finally 'receive' and 'send': the user
       accessible routines to use the network
       *)

machine = object

  path go end;
    (* no synchronization necessary for this
       initialization *)

  type
    buffer  = object        (* simple queue *)

      path bmax:(1:(bufput); 1:(bufget)) end;

    (* bmax outstanding requests (namely
       'bufput's) may exist, 'bufput's must
       precede 'bufget's.  *)

      var iobuffer: array[1..bmax] of message;
        inpp, outp: 1..bmax;

      entry procedure bufput(m:message);
        begin
          iobuffer[inpp] := m;
          inpp := (inpp mod bmax) + 1
        end;

      entry procedure bufget(var m:message);
        begin
          m := iobuffer[outp];
          outp := (outp mod bmax) + 1
        end;

      init; begin
        inpp := 1;
        outp := 1
      end;
    end;                (* buffer *)
    (* Only the Path Expression synchronizes
       the buffer code.  *)

    inputs = object     (* handle input *)

      path resp_rcvd; resp_wait end;

    (* 'resp_wait' will not continue until
```

```
    'resp_rcvd' is finished.  It then merely
    copies  the  message from the line moni-
    tor.  *)

    var mesgcontents: item;

    entry procedure resp_rcvd(cont: item);
      begin
        mesgcontents := cont
      end;

    entry procedure resp_wait
        (var cont: item);
      begin
        cont := mesgcontents
      end;
  end;                (* inputs *)

  outputs = object      (* handles output *)

    path rqst_rcvd; build_mesg end;

  (* 'build_mesg' may not be  executed  until
    'rqst_rcvd' is complete *)

    entry procedure build_mesg(c:channel;
        info:item; var mesg:message);
      begin
        mesg.kind := response;
        mesg.link := c;
        mesg.contents := info;
      end;

    entry procedure rqst_rcvd;
      begin   end;

  (* This procedure is empty as  no  code  is
    required,  only  a "signal" for the Path
    Expression to process.  *)

  end;                (* outputs *)

var buf: buffer;
  inp: array [channel] of inputs;
  out: array [channel] of outputs;
 (* logical channels are used for communica-
    tion.  Each machine has a different set
    of input and output channels.  *)

process reader(inpset, outset:channelset;
      inline:line);
    (* read messages from line *)
  var m: message;

  begin
    repeat
      inline.from_buslink(m);
        (* get message from line *)
      if (m.kind = response) and
          (m.link in inpset)
            (* response for me? *)
        then inp[m.link].resp_rcvd(m.contents)
      else
        if (m.kind = request) and
          (m.link in outset)
            (* request for me? *)
          then out[m.link].rqst_rcvd
        else
          buf.bufput(m)
            (* pass message on *)
    until false
  end;          (* reader process *)

  process writer(outline:link);
      (* put messages onto line *)
    var m: message;

    begin
      repeat
        buf.bufget(m);
        outline.to_buslink(m)
      until false
    end;                (* writerprocess *)

  entry procedure go(who: node;
    inpset, outset:channelset;
    inline, outline: line);
    begin
      reader(inpset, outset, inline);
      writer(outline)
    end;

   (* User called procedures: *)

  procedure   receive(c:channel; var v:item);

    var mesg: message;

    begin
      mesg.kind := request;
      mesg.link := c;
      buf.bufput(mesg);    (* request mesg *)
      inp[c].resp_wait(v)  (* grab response *)
    end;

  procedure   send(c: channel; info:item);

    var mesg: message;

    begin
      out[c].build_mesg(c, info, mesg);
        (* build mesg after reqst *)
      buf.bufput(mesg)    (* send mesg along *)
    end;

     (* Each machines's code would go here:   it
       would be invoked by go *)

  end;                  (* machine *)

     (* Finally, it is necessary to specify  the
       physical lines between the machines *)

var net: array [node] of machine;
  lines: array [node] of line;

begin
  net[1].go(1, [2,3], [1,4], lines[3], lines[2]);
  net[2].go(2, [1,6], [2,5], lines[1], lines[3]);
  net[3].go(3, [4,5], [3,6], lines[2], lines[1])
end.
```

D.2 DINING PHILOSOPHERS

The well known problem of the dining philosophers involves a set of five philosophers whose activities in life are eating and thinking. Each philosopher thinks for a while, eats, thinks, eats and so on. The philosophers share a unique dining arrangement: though two utensils are required for a philosopher to eat, the five dining places are located around a circular table with only one

utensil on the right of each dining place. Therefore, the philosophers must share utensils. The problem involves the scheduling of the philosophers so that no philosopher attempts to begin eating when his utensils are not available. The Path Pascal solution to this problem is different from many in that no explicit queues are needed. Each philosopher is a process attempting to use the 'fork' objects. Paths synchronize access and prevent deadlocks from occurring. Note that only simple synchronization statements are given (e.g., only four philosophers eating at a time, only one using each fork). The rest of the program specifies the logic of thinking and eating.

```
const nphilosophers = 5;
      maxindex = 4;      (* nphilosophers - 1 *)

type  diner = 0..maxindex;

var   i: integer;
   table: object
      path maxindex:(starteating; stopeating) end;
      var   fork: array [diner] of
         object
            path 1:(pickup; putdown) end;
            entry procedure pickup;  begin end;
            entry procedure putdown;  begin end;
         end;

      entry procedure starteating(no: diner);
         begin
            fork[no].pickup;
            fork[(no+1) mod nphilosophers].pickup
         end;

      entry procedure stopeating(no: diner);
         begin
            fork[no].putdown;
            fork[(no+1) mod nphilosophers].putdown;
         end;
      end;      (* table *)


process philosopher(mynum: diner);
   begin
      repeat
         delay(ran(seed));
         table.starteating(mynum);
         delay(ran(seed));
         table.stopeating(mynum);
      until false;
   end;

begin
   for i:= 0 to maxindex do philosopher(i)
end.
```

D.3 BUFFER MANAGEMENT

A simple ring buffer implementation is shown below:

```
const   bufsize = 32;
        maxbuf = 31;

type    buffer = object          (* buffers i/o *)

    path bufsize: (1: (fill); 1: (empty)) end;

    type     bufrange = 0..maxbuf;
             bufarray = array[bufrange] of char;

    var inptr, outptr: bufrange;
        buf: bufarray;

    entry procedure fill(ch: char);
        begin
            buf[inptr] := ch;
            inptr := (inptr+1) mod bufsize
        end;

    entry procedure empty(var ch: char);
        begin
            ch := buf[outptr];
            outptr := (outptr+1) mod bufsize
        end;

    init; begin inptr := 0; outptr := 0 end
end;
```

Two routines are provided, 'fill' and 'empty'. Note that the routines are very terse: only information relating to the actually changing of pointers and data is presented. All synchronization and restriction information is described by the Path Expression, which assures mutual exclusion for each routine and places a maximum on the buffer size. Attempts to exceed the buffer size are not allowed to proceed until an element is removed from the buffer.

D.4 TERMINAL DRIVER

A simple driver for a full duplex terminal is shown below:

```
type    bits = set of 0..15;

var screenbuf, programbuf: buffer;

interrupt process kbd[vector = #60, priority = 4];

    var kybdst[#177560]: bits;
        kybddt[#177562]: bits;
        ch: bits;

    begin
        kybdst := [6];
        repeat
            doio;
            ch := kybddt - [7]; (* zap parity bit! *)
            screenbuf.fill(ch);
            programbuf.fill(ch);
        until false;
    end;
```

```
interrupt process scrn[vector = #64, priority = 4];

    var scnst[#177564]: bits;
        scndt[#177566]: bits;
        ch: bits;

    begin
        scnst := [6];
        repeat
            screenbuf.empty(ch);
            scndt := ch;
            doio;
        until false;
    end;
```

The two routines perform input and output respectively. Very little code is required once all specifications have been presented. The doio in each routine waits for its associated interrupt and then does a small amount of processing before enabling the next interrupt. The input routine fills a buffer named 'screenbuf', while the output routine empties it and displays it on the screen after copying the contents to 'programbuf'. These routines show the ease with which device drivers can be implemented in Path Pascal.

BIBLIOGRAPHY

[Brinch Hansen, 78] Brinch Hansen, P., "Network: A Multiprocessor Program," IEEE Trans. Software Eng., Vol. SE-4, No. 3, pp. 194-199, May, 1978.