ON THE DESIGN OF MONITORS WITH PRIORITY CONDITIONS

David E. Boddy
School of Engineering and Computer Science
Oakland University
Rochester Michigan 48063

ABSTRACT

Hoare[1] introduced the monitor as a tool for structuring the design of concurrent systems such as operating systems. He roposed the use of "priority conditions" to facillitate certain types of scheduling. However in his proposal there are no provisions for allowing a "customer" of the monitor to inquire as to the status of a condition. Such inquiries as "What is the highest priority process waiting on condition X?" or "How many processes are waiting on condition X with highest priority?" are not supported in Hoare's design. This paper investigates the implementation of priority conditions for monitors under UCSD Pascal and proposes two such status queries which are both useful and efficient. It is shown that the implementation of Hoare's "alarmclock" monitor is made simpler and more efficient through the use of these queries.

INTRODUCTION

In [1] is presented a technique for implementing monitors in UCSD Pascal using "Units" and semaphores. However the discussion in [1] did not go into the question of the implementation of priority conditions. This paper presents a technique for implementing priority conditions as priority-ordered lists of semaphores. Furthermore certain queries on priority conditions are introduced which simplify certain scheduling applications.

Monitors are used for scheduling concurrent activities such as access to a shared resource. Figure 1 is a simple monitor for enforcing exclusive access to resource R. It is assumed that the resource R is only accessible through this monitor. Within the monitor the condition variable C guards this access. When a process, seeking access to resource R, executes the procedure **WAITC**(C) the process will be delayed until no other process has "rights" to R. A process releases its rights to use R by executing the **SIGNALC**(C) procedure.

Priority conditions within monitors facilitate more complex scheduling. If two processes are waiting on a priority condition, then when the condition is signalled, that process having greater priority is released. However, if both processes have the same priority then (in the present implementation) the first process to wait is released. An obvious application of monitors with priority conditions would be in the scheduling of processes according to priority, as is commonly done in operating systems for the "ready queue" of processes waiting to be dispatched.

IMPLEMENTING PRIORITY CONDITIONS

The basic concept of a priority condition implies that processes waiting on that condition are ordered by priority. It is not clear whether processes waiting with the same priority should be serviced on a first-come, first-served basis, though this is frequently the case. Certainly priority conditions could be implemented in the operating system by means similar to those used to implement semaphores. But we are concerned here with implementing monitors under a system (UCSD) which does not directly support priority conditions, only semaphores. It should be clear that a priority condition queue can be implemented via an ordered list of semaphores, one semaphore for each active priority. All processes waiting with a given priority wait in the queue of the associated semaphore.

Figure 2 gives the data structure used to implement a priority condition. Each node of the list consists of

a) A priority (integer), with larger numbers representing greater priority.
b) A semaphore on which all processes having the above priority wait.
c) A count of the processes waiting on the semaphore.

The waiting-count field is used to enable the system to delete a node which is no longer in use (i.e., a semaphore on which no proceses are waiting). [Note that if this were done by the system, the semaphore counter could be used for this purpose by allowing it to run negative.]

The operations over priority conditions, namely **PWAITC** and **SIGNALC**, are implemented as follows. The **PWAITC** implies a search of the list to find either the desired priority upon which to wait or to find where to insert a new node with the desired priority. The **SIGNALC** operation simply signals the semaphore at the head of the list (highest priority) and decrements its waiting-counter. If this counter is reduced to zero, the node is then deleted. Figure 3 is a listing of the "Monitor Toolbox" which implements these priority condition operations. (Refer to [2] for a discussion of the Monitor Toolbox.) Note that the (non-prioritized) **WAITC** is realized as a wait on a priority condition, but with lowest priority. Normally the user would not mix priority and non-priority waits on a given condition, though it is permitted.

The question arises as to whether the user should be allowed to examine any features of the priority queue. For example, should there be a function that returns the length of the queue? This particular query would seem to have two disadvantages: (1) it presumes some knowledge on the part of the user as to the internal structure of a priority queue (to interpret the meaning of "length"); and (2) the realization of the function might require a traversal of the list (if represented in linked form) which could be costly. A conservative approach would be to disallow any query which is either costly or discloses internal structure. But any query which could be answered only by examining the contents of the first node in the list would be efficient. Furthermore, disclosing only the highest priority value in the list and the number of processes waiting with this priority does not seem to require knowledge of the representation of the queue to interpret these values. Thus, in contrast to Hoare's design[1], it is proposed that the following queries be implemented:

a) **maxpri** (condition_variable) returns the greatest priority among all processes waiting on the condition.

b) **waiting** (condition_variable) returns the number of processes waiting (with maximum priority) on the condition variable.

These functions are implemented in the Monitor Toolbox shown in Figure 3. Note that in this implementation, **waiting** also applies to non-priority conditions. This can simplify monitors which do not use priority conditions and meets the need filled by the function **empty**(condition) which is defined for monitors under **Euclid**[4].

APPLICATIONS OF PRIORITY CONDITIONS

Figure 4 illustrates the use of priority conditions in the design of a monitor for scheduling disk accesses using the "scan" algorithm. This design is a modification of the algorithm presented in [3]. In this version the RELEASE algorithm is simpler due to the use of an array of condition variables indexed by the current direction.

Hoare[1] presented an "Alarmclock" algorithm to illustrate the application of priority conditions in the design of monitors. In his algorithm a process can delay itself until a desired time by waiting on a condition with priority based on the time at which it is to awake. Each "tick" of the clock signals this condition to awaken the highest priority condition. But the awakened process must then go back to sleep again if it is not yet time to awake. Hoare argues that this is a relatively minor source of inefficiency. Figure 5 presents an algorithm that does not suffer from this kind of inefficiency. By checking the maximum priority among processes in the queue, this algorithm will only **signal** the condition if it is indeed time to awaken the next process. The while loop in the **"tick"** procedure **signals** the alarm condition exactly as many times as there are processes due to be awakened **at the current time.** The Alarmclock algorithm of Figure 5 is presented to illustrate the fact that the inquiry **maxpri** (condition) is useful and can contribute to the efficiency of certain monitors.

Figure 6 gives a program which employs the scan and alarmclock monitors in a simulation of a number of processes randomly accessing a disk. Each process performs the following:

```
repeat
     delay a random time period
     request a random cylinder on the disk
     delay a random time period
     release the cylinder
forever
```

The results of this simulation show the "elevator" like scheduling of disk accesses imposed by the scan algorithm.

CONCLUSIONS

This paper has presented a technique for implementing monitors with priority conditions in UCSD Pascal. A priority condition is implemented as a linked list of semaphores, where each semaphore has an associated priority. It is argued that the user should be able to access certain features of a priority condition variable, provided such access is efficient and does not disclose internal implementation details. It is demonstrated that access to the both the maximum priority of processes waiting on a condition and the number of processes waiting with maximum priority are useful.

ACKNOWLEDGMENT

The author is indebted to Bill Halchin for his suggestion regarding the inclusion of the "gate" type in the Monitor Toolbox interface. By means of different variables of type "gate", several instances of a given monitor can be executed concurrently.

REFERENCES

[1] C.A.R. Hoare, Monitors: an operating system structuring concept. **CACM** V17 #10, October 1974.

[2] D. E. Boddy, Implementing Data Abstractions and Monitors in UCSD Pascal, **SIGPLAN Notices,** V18 #5, May 1983.

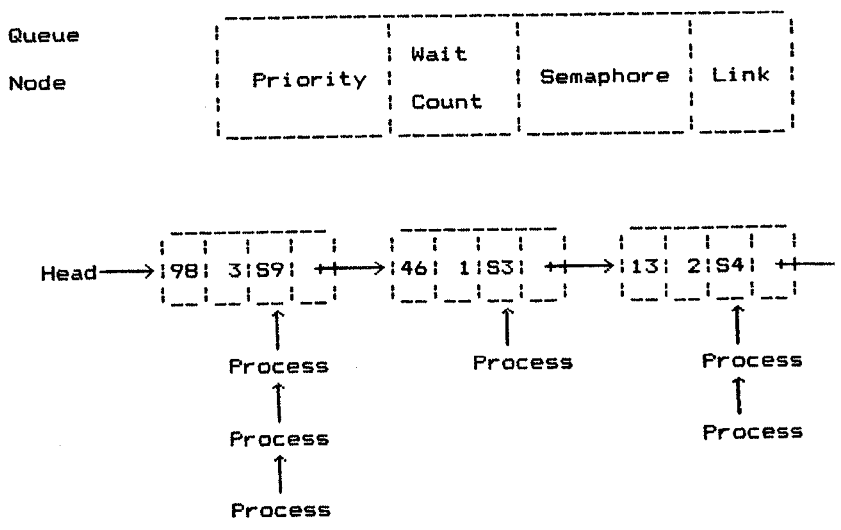
[3] R. C. Holt, G. S. Graham, E. D. Lazowska and M. A. Scott, **STRUCTURED CONCURRENT PROGRAMMING WITH OPERATING SYSTEM APPLICATIONS**, Addison-Wesley 1978.

[4] R. C. Holt, **CONCURRENT EUCLID, THE UNIX SYSTEM, AND TUNIS**, Addison-Wesley 1983 .

{   Figure 1. Simple Monitor for Exclusive Access

```
unit AccessControlMonitor;

{ Enforces mutually exclusive access to some resource }

   interface  uses MonitorToolBox;

           procedure request    (var fence: gate);
           procedure release    (var fence: gate);
           procedure initAccess (var fence: gate);

   implementation

        var
             in_use:  boolean;
             free:    Condition;

        procedure request {var fence: gate};
           begin
                EnterMonitor (fence);
                if in_use then waitC (fence, free);
                in_use:= true;
                ExitMonitor (fence);
           end;

        procedure release {var fence: gate};
           begin
                EnterMonitor (fence);
                in_use:= false;
                signalC (fence, free);
                ExitMonitor (fence);
           end;

        procedure initAccess {var fence: gate};
           begin
                create (fence);
                initC  (fence, free);
                in_use:= false;
           end;

   end {AccessControlMonitor}.
```

Figure 2. Structure of the Priority Queue



```
{         Figure 3.  The Monitor Toolbox Unit                     }

  unit Monitor_Toolbox; {version 6 }
  interface
     type
        priority = 0..maxint; {maximum priority = maxint}
        semQue =  record  { a semaphore waiting queue, with counter}
                    waiting: integer;
                    sem: semaphore;
                  end;
         gate = ^gate_record;
         gate_record = record {to control access to monitored data}
                          main: semaphore;
                          reentry: semQue;
                        end;
         Pque_cell = record  {a node in a priority queue}
                    pryority: priority;
                    waiting: integer;
                    sem: semaphore;
                    next: ^Pque_cell;
                 end;
        Condition = ^Pque_cell; {a priority queue}

     procedure Create       (var fence: gate);
     procedure EnterMonitor (var fence: gate);
     procedure ExitMonitor  (var fence: gate);
     procedure signalC  (var fence: gate; var c: Condition);
     procedure waitC    (var fence: gate; var c: Condition);
     procedure initC    (var fence: gate; var c: Condition);
     procedure PwaitC   (var fence: gate; var c: Condition;  pri: priority);
     function  maxpri   (first_cell: Condition): integer ;
     function  waitingC (first_cell: Condition): integer;

  implementation

     procedure EnterMonitor (var fence: gate);
     { seek to enter via the main gate }
        begin   wait (fence^.main)   end;

     procedure ReenterMonitor (var fence: gate);
      { seek to enter via the reentry gate.  Called by a signaller}
        begin
           with fence^.reentry  do begin
              waiting:= waiting + 1;
              wait(sem); {wait for signal from proc. exiting monitor}
              waiting:= waiting - 1;
              end;
        end;

     procedure ExitMonitor (var fence: gate);
        begin
           with fence^  do
              if (reentry.waiting > 0) then
                    signal( reentry.sem)
              else signal( main );
        end {Exit Monitor};


     procedure cleanup( var c: Condition);
     {delete unused queue node, if any}
        var temp: Condition;
        begin
           if c^.waiting = 0 then begin
              temp:= c;
              c:= c^.next;
              dispose(temp);
              end;
        end {cleanup};
```

```
procedure PwaitC (var fence: gate; var c : Condition; pri: priority);
{priority wait on a condition variable}
   var curser, trailer, temp: Condition;
        found, done: boolean;
   begin
      trailer:= nil;  curser:= c;  found:= false;
      repeat
         if curser = nil then done:= true {end of list}
         else if pri = curser^.pryority  then begin
                  found:= true; {found cell to wait on}
                  temp:= curser; {wait on this cell}
                  done:= true; {exit loop}
                  end
              else if pri > curser^.pryority then
                     done:= true   {insert new cell before curser}
                 else begin {advance curser}
                        trailer:= curser;
                        curser:= curser^.next
                      end;
        until done;
        if not found then begin {create  a new cell}
           new(temp);
           with temp^ do begin {initialize and link}
              pryority:= pri;  seminit(temp^.sem,0);
              waiting:= 0;
              next:= curser; {link temp before curser}
              end;
           if trailer = nil then {temp is new first cell}
              c:= temp
           else  {link temp after trailer}
                 trailer^.next:= temp
           end; { create new cell}
        temp^.waiting:= temp^.waiting + 1;
        ExitMonitor (fence);
           wait(temp^.sem); { wait on temp (whether new or old)}
        {...Signaller lets me back into monitor}
        temp^.waiting:= temp^.waiting - 1;
        cleanup(c);
     end {PwaitC};

  procedure waitC {var fence: gate; var c: Condition};
  {non-priority wait on a condition variable}
     begin
        PwaitC (fence, c, 0);{wait with least priority}
     end {waitC};

  procedure signalC { var fence: gate; var c: Condition};
  {wake up a highest-priority process waiting on a condition variable}
     begin
        if c <> nil then begin
           signal( c^.sem);
           ReenterMonitor (fence);
           end;
     end {signalC};


   function maxpri { first_cell: Condition};
   {returns greatest priority of any process waiting on the condition or
    else a negative integer if no process is waiting}
      begin
         if first_cell = nil then maxpri:= -1  {a non-priority value}
         else maxpri:= first_cell^.pryority;
      end;

   function waitingC   { first_cell: Condition; pri: priority };
   {returns no. of processes waiting with maximum priority}
      begin  if first_cell = nil then waitingC:= 0
              else waitingC:= first_cell^.waiting
      end;

   procedure initC{ var c: Condition};
   {initialize a condition variable (to empty queue)}
         begin  c:= nil  end;


   procedure create {var fence: gate};
   { allocate and initialize fence data }
      begin
         new (fence);
         with fence^ do begin
            seminit (main ,1);
            seminit (reentry.sem, 0);
            reentry.waiting:= 0;
            end;
      end {init};

   begin
   end {Monitor_Toolbox unit}.
```

Figure 4. The SCAN Monitor }

```
unit scan_monitor;
{ monitor for scheduling a disk using "SCAN" algorithm}

interface    uses MonitorToolbox ;

   const   max_cyl = 200; { cylinders numbered 0..200}

   procedure initDisk (var fence: gate);
   procedure acquire (var fence: gate;  cyl: integer);
   procedure release (var fence: gate);

implementation
   type
      direction = (up, down);
   var
      in_use: boolean; {records state of the disk drive}
      queue: array [direction] of condition; {Note array of conditions}
      current_direction: direction; { of disk head motion }
      current_cylinder: integer;

   procedure initDisk {var fence: gate};
   { allocate and initialize fence semaphores, and initialize conditions}
      begin
         create (fence);
         in_use:= false;
         current_direction:= up;
         initC (fence, queue[up]);
         initC (fence, queue[down]);
      end;

   procedure switch( var d: direction);  { change direction of motion}
      begin
         if d = up
         then d:= down
         else d:= up
      end;

   procedure acquire { var fence: gate; cyl: integer};
   {acquire access rights to a cylinder }
      begin
         EnterMonitor (fence);
         if  in_use then
            if (cyl < current_cylinder)
               or ((cyl = current_cylinder) and (current_direction = up))
            then PwaitC (fence, queue [down], cyl)
            else PwaitC (fence, queue [up], max_cyl-cyl);
         in_use:= true;
         current_cylinder:= cyl;
         ExitMonitor (fence);
      end {acquire};

   procedure release {fence};
   {release access rights to current cylinder}
      begin
         EnterMonitor (fence);
         in_use:= false;
         if 0 = waitingC (queue[current_direction])
         then switch (current_direction);
         signalC (fence, queue [current_direction]);
         ExitMonitor (fence);
      end {release};

   begin
   end {scan_monitor  monitor}.
```

```
      Figure 5.  The Alarmclock Monitor                        }

unit alarmclock;
interface   uses Monitor_Toolbox {version 6 };

   function time ( var clock: gate):integer;
   procedure tick (var clock: gate; print_time: boolean);
   procedure delay( var clock: gate; t: integer);
   procedure init (var clock: gate);

implementation
   var alarm: Condition;
       counter: integer   ; {count-down from maxint}

   function time { var clock: gate };
      begin
         EnterMonitor (clock);
         time:= maxint-counter;
         ExitMonitor (clock);
      end;

   procedure tick { var clock: gate; print_time: boolean};
      begin
         EnterMonitor (clock);
            if counter <= 0 then begin
               writeln('Timer runout.  Execution terminates.');
               exit(program);
               end
            else counter:= counter-1;
            if print_time then writeln('time =  ,maxint-counter);
            while counter <= maxpri (alarm)  do signalC (clock, alarm);
         ExitMonitor (clock);
      end;

   procedure delay { var clock: gate; t: integer};
      var setting: priority;
      begin
         if t>0 then begin
            EnterMonitor (clock);
                        setting:= counter-t;
                        PwaitC( clock, alarm, setting);
            ExitMonitor (clock);
         end;
      end;

      procedure init { var clock: gate };
         begin
            Create (clock);
            initC ( clock, alarm );
            counter:= maxint;
         end;
   begin
   end {alarmclock}.
```

```
{      Figure 6.  A Simulation Using the SCAN and ALARMCLOCK Monitors    }

       program prog4b; {disk-arm scheduling via "scan" alg.}

          uses MonitorToolbox,  scan_monitor, alarmclock;

          const
             stack_size = 500;
             fileSize = 20;
             proc_priority = 128;
          type
             file_index = 0..19;
             S_file = file of packed array[0..59] of char;
          var
             infile: S_file;
             pid: processid;
             seed: real;
             k: integer;
             clock, disk: gate;

          function rand( range: integer): integer;
             begin
                seed:= seed*31.415927;
                seed:= seed - trunc (seed); {delete integer part}
                rand:= 1 + trunc (seed * range);
             end;

          process P;
             var rec: file_index ;
             begin
                repeat
                   rec:= rand(fileSize);
                   acquire(disk, rec);
                   seek (infile, rec);
                   get(infile);
                   writeln('Read  record  ',rec,': ',infile^);
                   delay ( clock, rand( 5));
                   release (disk);
                   delay (clock, rand(5));
                until time (clock) >= 900;
            end {process P};

          begin {main prog}
             initClock (clock);
             initDisk (disk);
             seed:=  0.71123;
             reset(infile,'data4');
             for k:= 1 to  10 do
                start( P, pid, stack_size, proc_priority);
             repeat tick (clock, true) until time (clock) = 1000;
          end.
```