Implementation of Basic Primitives for Concurrent Programming in Pascal

Elmqvist, Hilding; Mattsson, Sven Erik

1981

Document Version:
Publisher's PDF, also known as Version of record

CODEN:LUTFD2/(TFRT-7231)/1-023/(1981)

A REAL-TIME KERNEL FOR PASCAL

HILDING ELMQVIST
SVEN ERIK MATTSSON

DEPARTMENT OF AUTOMATIC CONTROL
LUND INSTITUTE OF TECHNOLOGY

AUGUST 1981

LUND INSTITUTE OF TECHNOLOGY
DEPARTMENT OF AUTOMATIC CONTROL
Box 725
S 220 07 Lund 7 Sweden

Document name
REPORT

Date of issue
August 1981

Document number
CODEN:LUTFD2/(TFRT-7231)/1-023/(1981)

Author(s)
Hilding Elmqvist
Sven Erik Mattsson

Supervisor

Sponsoring organization
Swedish Board for Technical Development
STU-80-3962

Title and subtitle
A Real-time Kernel for Pascal

Abstract
A real-time kernel written in Pascal is described. It uses a small assembler written nucleus to handle concurrent processes. The expressive power of Pascal together with the kernel is comparable to Concurrent Pascal. Interrupts can also be handled in Pascal.

Key words
Concurrent programming, Real-time programming, Concurrent processes, Pascal, Concurrent Pascal

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
23

Recipient's notes

Security classification

INTRODUCTION

The real-time kernel described here was developed mainly for education. We have previously used Concurrent Pascal [1], [2]. However, we also wanted to be able to show all the details of how concurrency is achieved. The kernel is written in Pascal, but relies on an assembler written nucleus [3] for process creation, transfer between processes and handling of interrupts.

The kernel supports concurrent programming in a version of Pascal for LSI-11, called OMSI-Pascal [5]. The kernel implements semaphores for mutual exclusion and events for other synchronization. A Pascal program using the kernel could be structured in a similar way to a Concurrent Pascal program. A monitor in Concurrent Pascal corresponds to a record and some associated reentrant procedures. The concept queue corresponds to event. The kernel also offers the possibility to program I/O handlers. However, since the concurrent program is compiled with the standard Pascal compiler, there is much less security than in Concurrent Pascal.

THE PRIMITIVES OF THE KERNEL

Process declaration, creation, scheduling and termination

A process is declared as a parameterless procedure, which in the following will be refered as process-procedure. A process-procedure may be declared at any level. If parameters to the process are wanted, a procedure with parameters corresponding to a process type with parameters is declared. A small process-procedure without parameters is then declared for each process instance. It just calls the procedure with appropriate actual parameters.

A process instance may be created at any place, where the procedure could be called in the ordinary way, by calling the procedure 'createprocess'.

```
procedure createprocess(procedure proced;
                        memreq: unsignedinteger);
```

proced - the process-procedure describing the process
memreq - memory requirements (in bytes) for stack and heap

An estimate of the memory requirements for stack and heap must be given as 'memreq'. Overflow is detected by the run-time library.

The processes are scheduled according to their priorities. A high priority corresponds to a small number. A process having a priority number less or equal to zero will have the

interrupts disabled when it is executing. A process starts with its priority number equal to one. The priority of a process can be changed dynamically by a call to 'setpriority'.

```
procedure setpriority(priority: integer);

  priority - the new priority of the process
```

The priority number must be less than 'maxpriority', which is a predefined constant (See Appendix 1). A priority number greater or equal to 'maxpriority' causes the termination of the process and is the normal way to terminate a process. No memory is released. A process-procedure must not pass its own end.

The scope rules are the same as for ordinary procedures. The use of locally declared process-procedure demands great care. In order to not destroy local variables and the addressing mechanism for intermediate level variables, a process must not leave a procedure, if the actual invocation of the procedure has active process instances of locally defined process-procedures.

The main program must be converted to a process by a call to 'initkernel' before other processes can be created.

```
procedure initkernel(memreq: unsignedinteger);

  memreq - memory requirement (in bytes) for
           stack and heap (global variables excluded)
```

The procedure 'initkernel' also creates two other processes: one for handling of the clock and one "idle process".

Communication and synchronization between processes

Communication of data between processes are done using variables which are accessible to the process-procedures according to scope rules, via formal variable parameters or via pointers. Global variables in the outermost scope are frequently used for communication.

The programmer must ensure mutual exclusion by the use of semaphores or by temporarily disabling interrupts. A semaphore has an associated non-negative integer. It can only be operated by the following three procedures:

```
procedure initsemaphore(var sem: semaphore;
                        initval: integer);

procedure wait(sem: semaphore);

procedure signal(sem: semaphore);
```

```
sem       - the semaphore
initval   - initial value for semaphore
```

A semaphore must be initialized by a call to 'initsemaphore'. The effect of 'signal' is to increase the value of the semaphore with one, the increase being regarded as an indivisible operation. The effect of 'wait' is to decrease the value of the semaphore by one as soon as the result would be non-negative. A call of 'wait' implies a potential delay. The queue of waiting processes is ordered according to the priorities of the processes.

Synchronization between processes, such as waiting for a condition on a shared variable, is done by using the concept introduced in [6]. There are three operations on events.

```
procedure initevent(var e: event; sem: semaphore);

procedure await(e: event);

procedure cause(e: event);
```

```
e       - the event variable
sem     - the associated semaphore for mutual exclusion
```

An event must be initialized by a call to 'initevent', which associates it with the semaphore for mutual exclusion. A call of 'await' delays the process and makes an implicit signal to the associated semaphore. A call to 'cause' by another process moves all delayed processes to the queue of the associated semaphore.

Clock handling

The procedure 'waittime' makes the calling process wait a specified time interval.

```
procedure waittime(time: integer);
```

The time unit is ticks (20 ms). The prefix used when compiling user programs contains declarations of the three constants: tick = 1, sec = 50 and min = 3000. This makes it possible to express the time interval as for example waittime(2*min + 10*sec). Note that the maximum time interval allowed is 10 min.

Interrupt handling

The procedure 'waitio' makes the calling process wait for a specified interrupt. Only one process at a time can wait for a certain interrupt and this must be guaranteed by the user.

```
procedure waitio(vecaddr: unsignedinteger;
                 var statusreg: unsignedinteger);
```

The procedure sets the enable interrupt bit in the specified status register before suspending the running process. Someother process is then scheduled for execution.

When the interrupt occurs the enable bit in the status register is reset. The process waiting for the interrupt is indicated as ready for execution and the process having the highest priority is resumed.

The LSI-11 has memory mapped I/O. OMSI-Pascal allows manipulation of the device buffers and status registers as ordinary variables by allowing specification of addresses in the variable declaration (e.g. var printerbuffer origin 177566B: char). In standard Pascal this can be done by using a record with variants to convert an integer to a pointer.

Program organization

The user of the kernel should at compile time prefix his program with the file listed in Appendix 1.

Since the programmer has to ensure mutual exclusion by himself it is important to organize the program in a way that aids in using the semaphores in a correct way. A natural solution is to collect the data, the semaphore for mutual exclusion and the event variables in a record.

```
type data = record
  mutex: semaphore;
  cond: event;
  ...
  end;

var data1:data;
```

Operations on the data are then conveniently done using a with-statement.

```
with data1 do
  begin
  wait(mutex);
  while ... do await(cond);
  ...
  signal(mutex);
  end;
```

Ordinary Pascal procedures are reentrant. It means that it is possible to construct a set of procedures that operate on the shared data and which are the only way the data are operated on. This is the idea behind the monitor concept [1]. OMSI-Pascal allows type- , variable- and

procedure-declarations to be mixed. This makes it possible to collect the record-declaration and the procedures together.

RELATION BETWEEN THE KERNEL AND CONCURRENT PASCAL

Monitors

A monitor which handles a ring buffer is choosen to illustrate the relationship between the kernel and Concurrent Pascal. The monitor written in Concurrent Pascal is shown below.

```
const size = 100;

type buffer = monitor;

  var charbuff: array [1..size] of char;
      n, inp, outp: integer;
      sender, receiver: queue;

  procedure entry send(ch: char);
    begin
    if n = size then delay(sender);
    charbuff[inp] := ch;
    inp := (inp mod size) + 1;
    n := n + 1;
    continue(receiver);
    end;

  procedure entry receive(var ch: char);
    begin
    if n = 0 then delay(receiver);
    ch := charbuff[outp];
    outp := (outp mod size) + 1;
    n := n - 1;
    continue(sender);
    end;

  begin
  n := 0;
  inp := 1;
  outp := 1;
  end;
```

The corresponding Pascal program and the use of the kernel primitives is shown below.

```
const size = 100;

type buffer =
     record
       guard: semaphore;
       change: event;
       charbuff: array [1..size] of char;
       n, inp, outp: integer;
     end;

procedure send(var buff: buffer; ch: char);
  begin
  with buff do
    begin
    wait(guard);
    while n = size do await(change);
    charbuff[inp] := ch;
    inp := (inp mod size) + 1;
    n := n + 1;
    cause(change);
    signal(guard);
    end;
  end;

procedure receive(var buff: buffer; var ch: char);
  begin
  with buff do
    begin
    wait(guard);
    while n = 0 do await(change);
    ch := charbuff[outp];
    outp := (outp mod size) + 1;
    n := n - 1;
    cause(change);
    signal(guard);
    end;
  end;

procedure initbuffer(var buff: buffer);
  begin
  with buff do
    begin
    initsemaphore(guard, 1);
    initevent(change, guard);
    n := 0;
    inp := 1;
    outp := 1;
    end;
  end;
```

When using the kernel it is necessary to explicitly declare a semaphore as guard and to call wait and signal. This is done implicitly in Concurrent Pascal. One advantage with

Concurrent Pascal is that the compiler ensures mutual exclusion.

An event variable corresponds to a variable of the standard type 'queue' in Concurrent Pascal, with the following major differences: only one process at a time can be delayed in a queue variable and 'continue' implies an implicit return from the entry procedure.

The buffer record is included as a formal parameter to the procedures send and receive. Furthermore, a with-statement is used to get access to the fields of the record. In the case of Concurrent Pascal, this is done implicitly. A procedure of a certain monitor in Concurrent Pascal (e.g. outbuffer) is called using dot-notation:

```
outbuffer.send(ch);
```

When using Pascal, the monitor (record-variable) is given as an ordinary argument to the procedure.

```
send(outbuffer, ch);
```

If only one monitor of a certain type is required then it is possible to implicitly make the procedures operate on it as a nonlocal variable.

Processes

Processes in Concurrent Pascal are allowed to have formal parameters of monitor types to indicate "access rights" for the process. In the case of Pascal with the kernel this corresponds to using formal variable parameters of the record types corresponding to the monitors. However, a process-procedure is not allowed to have parameters. This means that an interface procedure has to be declared for each process instance with different actual arguments.

The use of process types is demonstrated by an example. A consumer process is receiving characters from a buffer monitor. The description of this situation is first shown for Concurrent Pascal.

```
type consumer = process(buff: buffer);
  var ch: char;
  begin
    cycle
    buff.receive(ch);
    ...
    end
  end
end;

var outbuffer: buffer;
```

```
    cons: consumer;

...

init outbuffer,
     cons(outbuffer);
```

The corresponding description when using Pascal together with the kernel is shown below.

```
var outbuffer: buffer;

procedure consumer(var buff: buffer);
  var ch: char;
  begin
  while true do
    begin
    receive(buff, ch);
    ...
    end
  end;

procedure cons;
  begin
  consumer(outbuffer);
  end;

...

initbuffer(outbuffer);
createprocess(cons, ...);
```

Input - Output

Input - output is handled by the standard procedure 'io' in Concurrent Pascal:

```
procedure io(var data: datatype; var param: ioparam;
             device: iodevice);
```

The kernel allows programming of input - output my means of the procedure 'waitio' and by using variables at fixed addresses to access the device buffers etc. A procedure similar to 'io' could therefore be programmed in Pascal.

EXAMPLE

A complete example that uses all of the primitives of the kernel is now given. The program writes on two terminals. It thus has two drivers (driver1, driver2) of the same type (driver). The drivers are connected to two ring buffers

(printer1, printer2) of the type described previously (buffer).

Because the ring buffers can hold the entire strings, they appear almost at the same time on the terminals when the program is executed.

```
{Include the prefix of Appendix 1.}

{ Program twoterminals;}

{ Include monitor type buffer. }


{ PROCESS TYPE }

procedure driver(var outbuffer: buffer;
                 var PRBUFF: char;
                 var PRSTATUS: integer;
                     INTPR: unsignedinteger);
  var ch: char;
  begin
  while true do
    begin
    receive(outbuffer,ch);
    waitio(INTPR, PRSTATUS);
    PRBUFF := ch;
    end;
  end;


{ MONITORS }

var printer1, printer2: buffer;


{ PROCESSES }

procedure driver1;
  const INTPR1 = 64B;
  var PRBUFF1 origin 177566B: char;
      PRSTATUS1 origin 177564B: integer;
  begin
  driver(printer1, PRBUFF1, PRSTATUS1, INTPR1);
  end;

procedure driver2;
  const INTPR2 = 374B;
  var PRBUFF2 origin 175616B: char;
      PRSTATUS2 origin 175614B: integer;
  begin
  driver(printer2, PRBUFF2, PRSTATUS2, INTPR2);
  end;
```

```
const CR = 15B; LF = 12B;

var string: array[1..30] of char;
    i: integer;

  begin
  initkernel(1000);
  initbuffer(printer1);
  initbuffer(printer2);

  createprocess(driver1, 1000);
  createprocess(driver2, 1000);

  while true do
    begin
    string := 'The quick brown fox jumped    ';
    for i := 1 to 26 do send(printer1, string[i]);
    send(printer1, chr(CR));
    send(printer1, chr(LF));

    string := 'over the lazy dog''s back      ';
    for i:= 1 to 24 do send(printer2, string[i]);
    send(printer2, chr(CR));
    send(printer2, chr(LF));

    waitime(2*sec);
    end;
  end.
```

IMPLEMENTATION

In this section the implementation of the procedures defined above will be discussed. The aim is to use standard Pascal [4] as far as possible. As hardware facilities like registers must be manipulated, it is not possible to make the code completely portable. However, use of the nucleus primitives discussed in [3] makes it possible to isolate the computer dependent parts. The introduction of concurrent processes means that code, processor and storage are shared resources. The problem to handle and protect these resources will now be considered.

Shared_Routines

A routine (procedure or function) compiled with a Pascal compiler is reentrant, because Pascal allows recursive routines. This means that such a routine can be used by several processes at the same time. However, it is not sure that standard routines (e.g. sin, cos and write) in Pascal are reentrant. Many Pascal implementations allow assembly code in-line or separatly compiled routines and it is not sure that these are reentrant. In the following it will be

assumed that a routine, that cannot be used by several processes at the same time, is regarded as a common resource that has to be protected with a semaphore in the ordinary way.

Processor Management

The procedures defined above contain switches of the processor between the processes. The nucleus has primitives that can suspend and resume processes, but the kernel must decide which of the processes that should be running. With respect to the scheduling of the processor, the existing processes can be divided into the three groups running, ready and blocked. In the ready state the process competes for the processor.

A process can wait on a synchronizing signal (semaphore, event, point of time or interrupt) by calling 'wait', 'await', 'waittime' or 'waitio'. If the synchronizing signal has not arrived the process is transfered to the blocked state and the process becomes ready when the signal arrives. A synchronizing signal is sent when a process calls 'signal' or 'await' (makes an implicit signal on the associated semaphore), the real-time clock ticks or an interrupt occurs. When a process calls 'cause', the processes, which have awaited that event, are all transfered to wait on the synchronizing signal of the associated semaphore.

A running process is preempted and transfered to the ready state if a process having a higher priority becomes ready.

To be able to perform the transitions, the kernel must know the priority and the process variable (stackpointer, program counter etc, see [3]) of each process. It is convenient to store this information in a process record.

When a synchronizing signal is produced the kernel must find the waiting processes in an efficient way. A process can only wait on one sychronizing signal at a time. All the processes waiting on a particular signal are therefore conviniently organized as a doubly linked list of their records. The process record must then contain one forward and one backward pointer, which in Pascal can be written as

```
type processref = ^processrec;
     processrec =  record
       succ, pred: processref;
       proc: process;
       priority: integer;
       time: integer;
     end;
```

Every list has a head of the same type as the rest of the elements in the list in order to avoid special handling of

empty lists.

There is one list of process records associated with each semaphore ('waiting') and each event ('delayed'). All processes waiting a specified time are kept in a single list ('timequeue'). They are ordered according to increasing waiting times. The process record contains one field ('time') which contains the waiting time relative to the preceeding process in the time queue. The waiting time for the first process is relative to the current time. This is an efficient way of organizing the time queue because the clock process needs only to decrement the time-field of the first process at each clock tick. The relative waiting times are calculated by the procedure 'waittime'.

Only one process is allowed to wait on a specific interrupt. There is therefore no need for a list of process records. A reference to the process record of the waiting process is kept in a variable ('driver'), which is local to the procedure 'waitio'. The nucleus takes care of resuming the rigth process when the interrupt occurs.

The processes in the ready state are also kept in a list ('readyqueue') and a variable ('running') of type 'processref' keeps track of the running process.

The scheduling rules make it natural to order the elements in the ready queue and in the queues associated with semaphores according to their priorities. The order is unimportant in a queue associated with an event so its elements can be inserted at the end. If the running process remains in the ready queue the switches between the ready and running states will take shorter time.

Memory Management

The memory management that has to be performed by the kernel is very simple. The kernel needs only to keep track of the free memory. The initial start and end addresses of the free memory are returned by the nucleus after the call of initnucleus and, since no memory is released, the kernel needs only to update the free memory when a new process is created.

The Structure of the Kernel

The data structure of the kernel is a shared resource and mutual exclusion must be ensured. It is accomplished at this level by disabling interrupts.

The entry routines of the kernel must be visible from the user's program. The OMSI-Pascal compiler allows separately compiled modules with procedures and functions. A global

routine in a separately compiled module is visible from other compile modules, if the compiler switch E (External) is turned on {$E+} and hidden if it is turned off {$E-}. The user, who wants to use a separately compiled routine must define a procedure heading followed by the keyword external. The compiler cannot verify that this is correctly done. If the user is provided with a file containing these declarations, he can include it in his program to eliminate this source of errors.

With this solution it is only the entry procedures of the kernel that are visible, while the rest are hidden. Variables like running must be permanent and visible from the procedures of the kernel. The permanent variables of the kernel must consequently be global. Global variables in separate modules are mapped over the global variables in the main program. The mechanism is similar to the unnamed common section available in Fortran programs. This implies that the permanent variables must be declared in the prefix. In order to hide the structure, they are declared as an anonymous array (See Appendix 1).

SUMMARY AND DISCUSSION

This paper and [3] shows how it is possible to introduce concurrency in ordinary Pascal without changing the compiler or the support library. This approach is therefore suitable for education.

The resulting kernel is comparable to Concurrent Pascal or to Texas MPP (Microprocessor Pascal). A similar kernel with message passing mechanisms written in Modula-2 is given in [7].

Some extensions have been made to the kernel in order to make it easier to work with. A D/A converter is used for outputting an analog signal to be displayed on an oscilloscope and showing what process is currently running. It is also possible to take a snap-shot and generate a status report for the processes. A multi terminal handler has been written in Pascal. It has been connected to the read- and write-statements of Pascal by handling the software interrupts generated by the support library.

A set of message passing primitives has also been constructed as an alternative to the monitor approach.

ACKNOWLEDGEMENTS

The idea to use OMSI-Pascal together with a kernel for concurrent programming came from the authors and Professor Karl Johan Åström. It was tested in a course on modern languages for process control. This project was performed by

Tommy Essebo, Rolf Johansson, Matz Lenells and Lars Nielsen under supervision of the authors. It resulted in a kernel with in-line assembly code for context switches etc. [8].

The authors also want to thank Leif Andersson for many good ideas and stimulating discussions, and Per Hagander for all the valuable comments on the manuscript.

REFERENCES

1. P. Brinch Hansen, 'The Architecture of Concurrent Programs', Prentice Hall Inc., Englewood Cliffs, New Jersey, 1978.

2. S. E. Mattsson, 'Implementation of Concurrent Pascal on LSI-11', Software - Practice and Experience, Vol. 10, 205-217 (1980).

3. H. Elmqvist and S. E. Mattsson, 'Implementation of Basic Primitives for Concurrent Programming in Pascal', Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1981.

4. K. Jensen and N. Wirth, 'Pascal - User and Manual Report', Springer Verlag, Berlin, 1975.

5. OMSI PASCAL-1 Documentation Version 1.1, Oregon Minicomputer Software Inc., 2340 SW Canyon Road, Portland, Oregon 97201.

6. P. Brinch Hansen, 'Operating System Principles', Prentice Hall Inc., Englewood Cliffs, New Jersey, 1973.

7. J. Hoppe, 'A Simple Nucleus Written in Modula-2', Institut für Informatik, ETH, Zürich, March 1980.

8. T. Essebo, R. Johansson, M. Lenells and L. Nielsen, 'A Facility for Executing Concurrent Processes in Pascal', Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1980.
CODEN:LUTFD2/(TFRT-7194)/1-061/(1980).

APPENDIX 1 - User interface of the kernel

```
program concurrent;

const sizekerneldata = 5;
      maxpriority = 1000;
      tick = 1; sec = 50; min = 3000;


type unsignedinteger = 0..65535;
     semaphore = unsignedinteger;
     event = unsignedinteger;


procedure initkernel(memreq: unsignedinteger); external;
procedure createprocess(procedure proced;
                        memreq: unsignedinteger); external;
procedure setpriority(priority: integer); external;

procedure initsem(var sem: semaphore;
                  initval: integer); external;
procedure wait(sem: semaphore); external;
procedure signal(sem: semaphore); external;

procedure initevent(var e: event;
                    sem: semaphore); external;
procedure await(e: event); external;
procedure cause(e: event); external;

procedure waitio(vecaddr: unsignedinteger;
                 var statusreg: integer); external;

procedure waittime(t: integer); external;

var kerneldata: array [1..sizekerneldata] of unsignedinteger;
```

APPENDIX 2 - Listing of the kernel

```
{Real-Time Kernel for Pascal.}

{Use the nucleus prefix.}

const maxpriority = 1000;

type processref = ^processrec;
     semaphore = ^semaphorerec;
     event = ^eventrec;

     processrec = record
       succ, pred: processref;
       proc: process;
       priority: integer;
       time: integer;
     end;

     semaphorerec = record
       counter: integer;
       waiting: processref;
     end;

     eventrec = record
       reentry: semaphore;
       delayed: processref;
     end;

var running, readyqueue, timequeue: processref;
    freetop, freebase: unsignedinteger;

{-------------------------------------------------------------}
procedure put(p, q: processref);
{Inserts process record p before process record q in q's
 list.}
  begin
  p^.succ:=q;
  p^.pred:=q^.pred;
  q^.pred^.succ:=p;
  q^.pred:=p
  end;
{-------------------------------------------------------------}
procedure remove(p: processref);
{Removes processrecord p from its list.}
  begin
  with p^ do
    begin
    pred^.succ := succ;
    succ^.pred := pred
    end
  end;
{-------------------------------------------------------------}
```

APPENDIX 2 - Listing of the kernel

```
procedure putpriority(p, q: processref);
{Inserts processrecord p in queue q according to priority.}
  var p1: processref; pri: integer;
  begin
  pri := p^.priority;
  p1 := q^.succ;
  while (p1 <> q) and (pri >= p1^.priority) do p1:=p1^.succ;
  put(p, p1)
  end;
{-------------------------------------------------------------}
procedure setpriority(priority: integer); forward;
{-------------------------------------------------------------}
{process} procedure idleproc;
  begin
  setpriority(maxpriority - 1);
  while true do begin end;
  end;
{-------------------------------------------------------------}
{process} procedure clock;
  const clockint = 100B;
  var p: processref;
  begin
  setpriority(-maxpriority);
  remove(running);  {clock scheduled specially.}

  while true do
    begin
    running := readyqueue^.succ;
    ioresume(running^.proc, clockint);

    {Decrement wait time for first waiting process.}
    p := timequeue^.succ;
    if p <> timequeue then p^.time := p^.time-1;

    {Move all due processes to readyqueue.}
    while (p^.time = 0) and (p <> timequeue) do
      begin
      remove(p);
      putpriority(p,readyqueue);
      p := timequeue^.succ;
      end;
    end;
  end;
{-------------------------------------------------------------}
procedure schedule;
  begin
  if readyqueue^.succ <> running then
    begin
    running := readyqueue^.succ;
    resume(running^.proc);
    end;
  end;
{-------------------------------------------------------------}
```

APPENDIX 2 - Listing of the kernel

```
{$E+} {external}
procedure createprocess(procedure proced;
                        memreq: unsignedinteger);
  var child: processref;
  begin
  disableinterrupts;
  freetop := freetop - memreq;

  new(child);
  child^.priority := 1;  {Default priority.}
  putpriority(child, readyqueue);
  newprocess(proced, freetop, freetop+memreq, child^.proc);

  schedule;
  if running^.priority > 0 then enableinterrupts;
  end;
{-------------------------------------------------------}
procedure initkernel(memreq: unsignedinteger);
  const clockarea = 100; idlearea = 100;
  begin
  disableinterrupts;

  {Create readyqueue with running.}
  new(running);
  new(readyqueue);
  readyqueue^.succ := running;
  readyqueue^.pred := running;
  running^.succ := readyqueue;
  running^.pred := readyqueue;

  {Create empty time queue.}
  new(timequeue);
  timequeue^.succ := timequeue;
  timequeue^.pred := timequeue;

  running^.priority := 1;
  initnucleus(memreq, freebase, freetop, running^.proc);

  createprocess(clock, clockarea);
  createprocess(idleproc, idlearea);
  end;
{-------------------------------------------------------}
procedure setpriority {(priority: integer)};
  begin
  disableinterrupts;
  if priority > running^.priority then
    begin
    running^.priority := priority;
    remove(running);  {Reorder readyqueue.}
    putpriority(running, readyqueue);
    schedule;
    end
  else
    running^.priority := priority;
  if running^.priority > 0 then enableinterrupts;
  end;
{-------------------------------------------------------}
```

APPENDIX 2 - Listing of the kernel

```
procedure initsem(var sem: semaphore; initval: integer);
  begin
  new(sem);
  with sem^ do
    begin
    counter:=initval;
    new(waiting);  {Empty waiting queue.}
    waiting^.succ:=waiting;   waiting^.pred:=waiting;
    end;
  end;
{-------------------------------------------------------------}
procedure wait(sem: semaphore);
  begin
  disableinterrupts;
  with sem^ do
    begin
    if counter>0 then
      counter := counter-1
    else
      begin
      remove(running);
      putpriority(running, waiting);
      schedule
      end
    end;
  if running^.priority > 0 then enableinterrupts;
  end;
{-------------------------------------------------------------}
procedure signal(sem: semaphore);
  var p:processref;
  begin
  disableinterrupts;
  with sem^ do
    begin
    if waiting<>waiting^.succ then
      begin  {Put first waiting process in ready queue.}
      p := waiting^.succ;
      remove(p);
      putpriority(p, readyqueue);
      schedule
      end
    else
      counter:=counter+1
    end;
  if running^.priority > 0 then enableinterrupts;
  end;
{-------------------------------------------------------------}
```

APPENDIX 2 - Listing of the kernel

```
procedure initevent(var e: event; sem: semaphore);
  begin
  new(e);
  with e^ do
    begin
    reentry:=sem;
    new(delayed);  {Empty delayed queue.}
    delayed^.succ:=delayed;   delayed^.pred:=delayed;
    end;
  end;
{-------------------------------------------------------}
procedure await(e: event);
  var p: processref;
  begin
  disableinterrupts;
  remove(running);
  put(running, e^.delayed);
  {Signal associated semaphore.}
  with e^.reentry^ do
    begin
    if waiting <> waiting^.succ then
      begin
      p := waiting^.succ;
      remove(p);
      putpriority(p, readyqueue)
      end
    else
      counter := counter+1
    end;
  schedule;
  if running^.priority > 0 then enableinterrupts;
  end;
{-------------------------------------------------------}
procedure cause(e: event);
  var p: processref;
  begin
  disableinterrupts;
  with e^ do
    begin
    {Make all delayed processes wait for the associated
     semaphore.}
    while delayed <> delayed^.succ do
      begin
      p := delayed^.succ;
      remove(p);
      putpriority(p, reentry^.waiting);
      end;
    end;
  if running^.priority > 0 then enableinterrupts;
  end;
{-------------------------------------------------------}
```

APPENDIX 2 - Listing of the kernel

```
procedure waitio(vecaddr: integer;
                 var statusreg: unsignedinteger);
  const enable = 1008; disable = 0;
  var driver: processref;
  begin
  disableinterrupts;
  driver := running;

  {Save pointer to process record for calling process.}
  remove(running);
  running := readyqueue^.succ;  {Schedule}
  statusreg := enable;
  ioresume(running^.proc, vecaddr);
  statusreg := disable;

  running := driver;
  putpriority(driver, readyqueue);
  schedule;
  if running^.priority > 0 then enableinterrupts;
  end;
{-------------------------------------------------------}
procedure waittime(t: integer);
  var p: processref;
  begin
  disableinterrupts;
  remove(running);

  {Find position in time queue and compute wait time
   relative to preceeding process.}
  p := timequeue^.succ;
  while (t > p^.time) and (p <> timequeue) do
    begin
    t := t - p^.time;
    p := p^.succ;
    end;
  running^.time := t;
  put(running, p);

  {Modify relative wait time for next process.}
  if p <> timequeue then p^.time := p^.time - t;
  schedule;
  if running^.priority > 0 then enableinterrupts;
  end;
```