Modula-2 Process Facilities

D. A. Sewry

Department of Computer Science
Rhodes University
Grahamstown, South Africa

Introduction

Considerable interest has been expressed in recent years on the subject of concurrent programming in high level languages, and several languages have been developed or extended to allow such facilities:

Ada	[Uni81], [You83]
Clang	[Cha84]
Concurrent Euclid	[Hol83]
Concurrent Pascal	[Bri75]
CSP	[Hoa78]
Edison	[Bri82]
Extended Pascal-S	[Ben81], [Cha82]
Modula	[Wir77]
Modula-2	[Wir83]
Pascal-Plus	[Wel79], [Wel80]
UCSD Pascal	[Sof81]

In this paper we wish to explore the features offered by Modula-2 (Wirth, 1979). Modula-2 is a most interesting language in that concurrency as such is not supplied as part of the language. A coroutine construction is supplied, in terms of which quasi-concurrency may be implemented at the user's fancy, using routines in a "module", which then become part of a library. In his book Wirth suggests one such simple implementation, and this has been included with several widely used systems:

i) Volition Systems' Apple implementation running under the Apple Pascal System
ii) Volition Systems' SAGE IV and IBM PC implementation running under the UCSD Pascal II.0 System
iii) University of Hamburg's VAX-11 implementation.

In the discussion which follows we shall point out some difficulties with this system, and suggest some enhancements, as they have been programmed for the SAGE IV microcomputer using Volition Systems' Modula-2.

Wirth's Processes Module

The module Processes offers the following facilities:

```
PROCEDURE StartProcess(P:PROC; n:CARDINAL)
   (* start a process P with workspace size of n *)
```

```
PROCEDURE SEND(VAR S:SIGNAL)
   (* reactivate a process WAITing on S *)

PROCEDURE WAIT(VAR S:SIGNAL)
   (* WAIT for some process to SEND S *)

PROCEDURE Awaited(S:SIGNAL):BOOLEAN
   (* true if any processes WAITing on S *)

PROCEDURE Init(VAR S:SIGNAL)
   (* initialise S *)
```

It is important to note that only parameterless procedures can be designated as processes. This seems to be an unfortunate, and an unavoidable restriction in the base language.

Each initiated process is represented by a ProcessDescriptor in a process (schedule) ring. A ProcessDescriptor is defined by:

```
TYPE SIGNAL = POINTER TO ProcessDescriptor;
     ProcessDescriptor =
       RECORD
         next:  SIGNAL;  (* ring *)
         queue: SIGNAL;  (* queue of waiting processes*)
         cor:   PROCESS; (* process variable *)
         ready: BOOLEAN  (* active or waiting *)
       END;
```

When a process is started, by a call to StartProcess, a descriptor of the process and a workspace for its associated coroutine are set up. The descriptor is inserted in a circular list (ring) containing all process descriptors created so far. By traversing the list any process can be reached. Within this ring are subsidiary queues to handle signals.

SEND(S) takes a process off the queue pointed to by S, reinstates it as ready-to-run, adjusts S to point to the next process waiting on the signal, and finally transfers control from the sending process to the reinstated process.

WAIT(S) places the current process at the end of the queue for S and then searches for the next ready-to-run process in the ring. Deadlock occurs if there is no ready-to-run process, but otherwise control passes to the process so found.

Process Termination

The first difficulty with Wirth's system is that it does not clarify the action to be taken when a process terminates.

In many implementations, as soon as any process or the main program terminates, the entire program terminates.

Since the process concept and the underlying coroutine concept may appear somewhat different to users (there is no reason why they should associate

one in terms of the other), a way to ensure conformity is to require that all processes (and the parent program) execute, as their last statement, a call to a standard procedure (say, procedure StopProcess) which will remove from the schedule ring the ProcessDescriptor representing that process or program (in effect destroying all evidence of its existence). In addition to removing the ProcessDescriptor, StopProcess also searches the schedule ring for the next process which is ready-to-run and transfers control to it.

The COBEGIN...COEND Construct

In several discussions on concurrent programming the COBEGIN...COEND construct is used as a means of specifying which processes one wishes to execute concurrently.

eg:

```
COBEGIN

  A ;
  B ;
  C ;

COEND
```

in this instance A,B and C must be executed concurrently.

It is of interest to see how closely this idea can be implemented in Modula-2. Processes will still have to be initiated using StartProcess, and the closest we can hope to achieve will on the lines of:

```
BEGIN
 COBEGIN;
  StartProcess(A, 400);
  StartProcess(B, 400);
  StartProcess(C, 400);
 COEND
END MainProg.
```

COBEGIN and COEND have to be implemented as procedures, not "reserved words" which means, in general, that semicolons must follow both words.

Procedure COBEGIN can be made responsible for what was previously the initialisation code of module Processes. COEND can be merely a call to a revised version of StopProcess.

Further to the problem of process termination, it is still be necessary to call StopProcess as the last statement in any process.

Since it is possible to have the following trivial case:

```
.

COBEGIN;
COEND;
.
```

it is necessary to launch a dummy process from within procedure COBEGIN. Its only statement will be a call to procedure StartProcess which will, amongst other tasks, assign the correct value to the main program's process variable to allow control to be returned to the main program.

In Appendix A we present a revised implementation of the module Processes which show how these improvements could be effected.

Process Priorities

In Wirth's scheme the schedule ring is very simple: ProcessDescriptors are linked into it just next to the "launching" process in every case. The scheduling policy is also simple: at any process switch one simply selects the next ready-to-run process around the ring. (In Wirth's scheme switching only occurs as a result of an explicit WAIT, SEND, or StartProcess call). The system does not really allow for any form of scheduling policy based on priority, time slicing etc.

Were a priority to be assigned to each process, scheduling policies could be developed. Such policies might favour higher priority processes with either (i) greater amounts of processor time or (ii) more immediate attention on any process switch.

One such policy could be developed along the following lines: each time a process is initiated a priority can be assigned to it and used as a means of determining which process should be activated when confronted with a choice at a process switch. For example, activate the process with the highest priority (as is the case in the implementation given in the appendices).

Procedure StartProcess is augmented as follows:

```
PROCEDURE StartProcess(P:PROC; n:CARDINAL; PRIORITY:INTEGER);
```

The schedule ring now reflects ProcessDescriptors representing processes in ascending order of priority.

Degree of Concurrency

It will be noted that in the implementation discussed so far, context switching can only occur when an executing process either initiates another process, or executes a WAIT, SEND, or StopProcess operation. This can severely limit the degree of "concurrency" which can be achieved. (In an extreme case one obtains essentially none at all.)

To enhance the degree of concurrency one might either:

a) introduce time shared process scheduling as an intrinsic feature of the implementation, with implementation details hidden from the user, or
b) introduce the possibility of user defined scheduling, based on an accessible low level scheme such as a clock interrupt.

The latter possibility is totally in accordance with the design philosophy of Modula-2, whereas the former is in complete contradiction to it.

In an attempt to increase the degree of concurrency, a computer's internal clock can often be used to generate an interrupt driven process switch.

Since Modula-2 provides a low level procedure, IOTRANSFER, to handle interrupts, it can be used to trap clock interrupts. Having trapped the interrupt, processor control can be transferred to a Reschedule procedure which selects the next process to activate and then generates a process switch.

Timeslicing

A simple implementation of procedure Reschedule would merely select the next ready-to-run process in the schedule ring at a clock interrupt, whereas the process with the highest priority would always be selected when a choice is involved at all other process switches.

It is not much use altering Reschedule to activate the process with the highest priority at clock interrupts. This will, to a large degree, negate the effect of the clock. In general, the processor will currently be executing the process with the highest priority - with the exception of SEND the processor is always directed to the process with the highest priority. Consequently, when a clock interrupt occurs, there is a good chance that control will be passed back to the selfsame process which is currently being executed. A context switch will only occur when this process executes either a SEND or WAIT or StopProcess - a situation much the same as it was before the clock was introduced.

An alternative solution to merely selecting the next ready-to-run process is to allow each process a maximum number of timeslices determined by its priority, eg. a process with a priority of four will be allowed a maximum of four contiguous timeslices.

In the appendices, Reschedule is coded to manage this timeslice mechanism.

Implementations

In Appendix A we present an implementation of the COBEGIN ... COEND concept (including process termination).

In Appendix B we present an implementation of Wirth's module Processes [Wir83] extended to include process termination, process priorities, improved concurrency and timeslicing.

Conclusion

Initially a user is provided with a little less than the bare minimum required to achieve something which resembles, albeit somewhat remote, concurrency ("less" because a process-filled program will not even terminate correctly). However, with a little thought it is possible to

design one's own scheduler which not only solves the problem of process termination but which also provides some useful extensions.

Some might contend that this is precisely what Modula-2 is all about - building in an hierarchical fashion, from something very simple to something quite sophisticated. In this way the user provides himself with exactly that which he requires, no more, no less. It certainly affords the user the flexibility of choosing/creating the process facilities of his choice.

In conclusion, the suggested process facilities provided with the Modula-2 system are weak but using what is given as a basis for development, they can be extended to include some useful features.

Acknowledgements

I would like to thank Professor P. D. Terry for all his help and advice. I also acknowledge the financial support of the Council for Scientific and Industrial Research.

References

Ben81 Ben-Ari, M. Cheap Concurrent Programming. Software - Practice and Experience, 11, (12), 1261-1264 (1981).

Bri75 Brinch Hansen, Per. The Programming Language Concurrent Pascal. IEEE Transactions on Software Engineering, SE-1, (2), 199-207 (1975).

Bri82 Brinch Hansen, Per. Programming a Personal Computer. Prentice-Hall Inc., Englewood Cliffs, New Jersey. (1982).

Cha82 Chalmers, A.G. Pascal-S Mark1.HAC Compilers. B.Sc.(Hons) project, Dept. of Computer Science, Rhodes University, South Africa. (1982).

Cha84 Chalmers, A.G. The Monitor and Synchroniser Concepts in the programming language Clang. M.Sc. thesis, Dept. of Computer Science, Rhodes University, South Africa. (1984).

Hoa78 Hoare, C.A.R. Communicating Sequential Processes. Comm. ACM, 21, (8), 666-677 (1978).

Hol83 Holt, R.C. Concurrent Euclid, the UNIX system and TUNIS. Addison-Wesley, Reading, Massachusetts. (1983).

Sof81 UCSD Pascal 1V.Ø User Manual and Internal Architecture Guide. Softech Microsystems. (1981).

Uni81 United States Department of Defense. The Programming Language Ada -- Reference Manual. Lecture Notes in Computer Science, 1Ø6. Springer-Verlag, Berlin. (1981).

Wel79 Welsh, J. and Bustard, D.W. Pascal-Plus - Another Language for Modula Multiprogramming. Software - Practice and Experience, 9, (11), 947-957 (1979).

Wel80 Welsh, J. and McKeag, M. Structured System Programming. Prentice-Hall Inc., Englewood Cliffs, New Jersey. (1980).

Wir77 Wirth, N. Modula: A Language for Modula Multiprogramming. Software - Practice and Experience, 7, (1), 3-35 (1977).

Wir83 Wirth, N. Programming in Modula-2 and Report on the Programming Language Modula-2. Springer-Verlag, Berlin. (1983).

You83 Young, S.J. An Introduction to Ada. Ellis Horwood, Chichester, England. (1983).

Appendix A

```
DEFINITION MODULE COBI;
(* $SEG:=47; *)

(* NOTE: Ensure that the library module name is distinct *)

FROM SYSTEM IMPORT PROCESS;

EXPORT QUALIFIED StartProcess,StopProcess,SEND,WAIT,
                 COBEGIN,COEND,
                 Awaited,Init,
                 SIGNAL;


TYPE SIGNAL = POINTER TO ProcessDescriptor;
     ProcessDescriptor = RECORD
                          next: SIGNAL;
                          queue: SIGNAL;
                          cor: PROCESS;
                          ready: BOOLEAN
                         END;


PROCEDURE StartProcess(P:PROC; N:CARDINAL);
     (* Start a process P with workspace size of N *)

PROCEDURE StopProcess;
     (* Terminate the current process *)

PROCEDURE SEND(VAR S:SIGNAL);
     (* Reactivate a process WAITing on S *)

PROCEDURE WAIT(VAR S:SIGNAL);
     (* WAIT for some process to SEND S *)

PROCEDURE COBEGIN;
     (* Start concurrent execution *)

PROCEDURE COEND;
     (* Stop concurrent execution *)

(* Function *) PROCEDURE Awaited(S:SIGNAL): BOOLEAN;
     (* Any processes WAITing on S *)

PROCEDURE Init(VAR S:SIGNAL);
     (* Initialise S *)


END COBI.
```

```
IMPLEMENTATION MODULE COBI[1];

(*******************************************************
 *                                                     *
 *     MODULE COBI                                     *
 *                                                     *
 *     An implementation of the                        *
 *     COBEGIN . . . COEND construct making            *
 *     use of Wirth's Processes module as a            *
 *     basis, with process termination facilities      *
 *     added.                                          *
 *                                                     *
 *     Machine details: Volition Systems'              *
 *                       implementation of Modula-2    *
 *                     SAGE IV microcomputer           *
 *                                                     *
 *     Date: June, 1984          Author: D. A. Sewry   *
 *                                                     *
 *******************************************************)

FROM SYSTEM  IMPORT ADDRESS,TSIZE,PROCESS,NEWPROCESS,TRANSFER;
FROM Storage IMPORT ALLOCATE;

VAR CP,             (* pointer to current process *)
    MAIN: SIGNAL;   (* pointer to main program descriptor *)

PROCEDURE StartProcess(P:PROC; N:CARDINAL);
 (* Start a process P with workspace size of N *)
 VAR S0: SIGNAL;     (* temporary, launching process *)
     WSP: ADDRESS;   (* workspace *)
 BEGIN
  S0:=CP;
  ALLOCATE(CP, TSIZE(ProcessDescriptor));
  ALLOCATE(WSP, N);
  WITH CP^ DO  (* link descriptor into ring *)
   next:=S0^.next;
   S0^.next:=CP;
   ready:=TRUE;
   queue:=NIL;
  END;
  NEWPROCESS(P, WSP, N, CP^.cor);
  TRANSFER(S0^.cor, CP^.cor)
 END StartProcess;

PROCEDURE StopProcess;
 (* Terminate the current process *)
 VAR S0,NEXTJOB: SIGNAL;  (* temporary *)
     NOTFOUND: BOOLEAN;   (* ready process not found *)
 BEGIN
  IF CP = CP^.next
    THEN (* last process terminated - return to main program *)
         TRANSFER(CP^.cor, MAIN^.cor)
    ELSE S0:=CP; NEXTJOB:=CP; NOTFOUND:=TRUE;
         REPEAT  (* find next ready-to-run process *)
          NEXTJOB:=NEXTJOB^.next;
          IF ((NOTFOUND) AND (CP^.ready))
            THEN CP:=NEXTJOB;
                 NOTFOUND:=FALSE
          END
         UNTIL NEXTJOB^.next = S0;
         IF NOTFOUND
           THEN (* DEADLOCK *)
                HALT
           ELSE (* remove descriptor from ring *)
                NEXTJOB^.next:=S0^.next;
                TRANSFER(S0^.cor, CP^.cor)
         END
    END
 END StopProcess;

PROCEDURE SEND(VAR S:SIGNAL);
 (* Reactivate a process WAITing on S *)
 VAR S0: SIGNAL;  (* temporary, signalling process *)
 BEGIN
  IF S <> NIL
    THEN (* a process is WAITing *)
         S0:=CP;
         CP:=S;
         WITH CP^ DO  (* mark signalled process as active *)
          S:=queue; ready:=TRUE; queue:=NIL
         END;
         TRANSFER(S0^.cor, CP^.cor)
  END
 END SEND;

PROCEDURE WAIT(VAR S:SIGNAL);
 (* WAIT for some process to SEND S *)
 VAR S0,S1: SIGNAL;  (* temporary *)
 BEGIN
  IF S = NIL
    THEN (* first such process to WAIT *)
         S:=CP
    ELSE (* add to existing queue *).
         S0:=S;
         S1:=S0^.queue;
         WHILE S1 <> NIL DO  (* search for tail *)
          S0:=S1;
          S1:=S0^.queue
         END;
         S0^.queue:=CP
  END;
  S0:=CP;
  REPEAT CP:=CP^.next UNTIL CP^.ready;  (* find next process *)
  IF CP = S0 THEN HALT (* DEADLOCK *) END;
  S0^.ready:=FALSE;  (* deactivate process *)
  TRANSFER(S0^.cor, CP^.cor)
 END WAIT;

(* Function *) PROCEDURE Awaited(S:SIGNAL): BOOLEAN;
 (* any processes WAITing on S *)
 BEGIN
  RETURN S <> NIL
 END Awaited;

PROCEDURE Init(VAR S:SIGNAL);
 (* initialise S *)
 BEGIN
  S:=NIL
 END Init;

PROCEDURE Dummy;
 (* dummy process to allocate correct value to
    MAIN^.cor *)
 BEGIN
  StopProcess
 END Dummy;

PROCEDURE COBEGIN;
 (* start concurrent execution *)
 BEGIN
  ALLOCATE(CP, TSIZE(ProcessDescriptor));
  WITH CP^ DO  (* enter main program into ring *)
   next:=CP; ready:=TRUE; queue:=NIL
  END;
  MAIN:=CP;
  StartProcess(Dummy, 400)
 END COBEGIN;

PROCEDURE COEND;
 (* stop concurrent execution *)
 BEGIN
  StopProcess
 END COEND;

BEGIN  (* COBI *)
END COBI.
```

Appendix B

```
DEFINITION MODULE Processes;
(* $SEG:=47; *)

(* NOTE: Ensure that the library module name is distinct *)

FROM SYSTEM IMPORT PROCESS;

EXPORT QUALIFIED StartProcess,StopProcess,SEND,WAIT,
                 Awaited,Init,
                 SIGNAL;


TYPE SIGNAL = POINTER TO ProcessDescriptor;
     ProcessDescriptor = RECORD
                          next: SIGNAL;
                          queue: SIGNAL;
                          cor: PROCESS;
                          prior: INTEGER;
                          slice: INTEGER;
                          ready: BOOLEAN
                         END;


PROCEDURE StartProcess(P:PROC; N:CARDINAL; PRIORITY:INTEGER);
     (* Start a process P with workspace size of N and
        priority of PRIORITY *)

PROCEDURE StopProcess;
     (* Terminate the current process *)

PROCEDURE SEND(VAR S:SIGNAL);
     (* Reactivate a process WAITing on S *)

PROCEDURE WAIT(VAR S:SIGNAL);
     (* WAIT for some process to SEND S *)

(* Function *) PROCEDURE Awaited(S:SIGNAL): BOOLEAN;
     (* Any processes WAITing on S *)

PROCEDURE Init(VAR S:SIGNAL);
     (* Initialise S *)


END Processes.
```

```
IMPLEMENTATION MODULE Processes[1];

(*****************************************************
 *                                                   *
 *     MODULE Processes                              *
 *                                                   *
 *     Modified version of Wirth's Processes module. *
 *      Process termination added.                   *
 *      Priority assigned to processes. Always       *
 *       activate process with highest priority.     *
 *      Interrupt driven process switching added.    *
 *      Timeslice mechanism based on priority added. *
 *                                                   *
 *     Machine details: Volition Systems'            *
 *                       implementation of Modula-2  *
 *                      SAGE IV microcomputer        *
 *                                                   *
 *     Date: June, 1984          Author: D. A. Sewry *
 *                                                   *
 *****************************************************)

FROM SYSTEM   IMPORT ADDRESS,ADR,WORD,SIZE,TSIZE,PROCESS,
                     NEWPROCESS,TRANSFER,IOTRANSFER;
FROM Storage  IMPORT ALLOCATE;
FROM Program  IMPORT SetEnvelope,FirstCall;
FROM UnitIO   IMPORT UnitWrite;
FROM SYSTEM68 IMPORT ClearVector,SetPriority,Attach,Detach;

CONST Clock1Vector  = 9;    (* clock interrupt vector *)
      Event1Number  = 36;   (* clock event number *)
      ClockDevice   = 131;  (* clock device number *)
      ClockPriority = 3;    (* clock priority number *)
      StartClock    = 1;    (* clock start indicator *)
      StopClock     = 0;    (* clock stop indicator *)

VAR CLK,                            (* clock process *)
    Userprocess: PROCESS;           (* current user process *)
    CLKWSP: ARRAY [0..499] OF WORD; (* clock workspace *)
    T: ARRAY [0..1] OF CARDINAL;    (* timeslice duration value *)
```

```
MODULE SYNCHRO[4];
  (* non-interruptable module to protect process ring *)

IMPORT ADDRESS,TSIZE,PROCESS,NEWPROCESS,TRANSFER;
IMPORT ALLOCATE;
IMPORT SIGNAL,ProcessDescriptor;
IMPORT Userprocess;

EXPORT StartProcess,StopProcess,SEND,WAIT,Awaited,Init,
       CP,Reschedule;

VAR CP,                 (* pointer to current process *)
    TOPPROC: SIGNAL;    (* pointer to process with highest priority *)



PROCEDURE StartProcess(P:PROC; N:CARDINAL; PRIORITY:INTEGER);
 (* Start a process P with workspace size of N and
    priority of PRIORITY *)
 VAR S0,S1,PREVIOUS: SIGNAL;  (* temporary *)
     WSP: ADDRESS;  (* workspace *)
 BEGIN
  S0:=CP; S1:=TOPPROC;
  ALLOCATE(CP, TSIZE(ProcessDescriptor));
  ALLOCATE(WSP, N);
  IF S1^.prior < PRIORITY
    THEN (* new process has highest priority *)
         REPEAT
          S1:=S1^.next
         UNTIL S1^.next = TOPPROC;
         PREVIOUS:=S1;
         TOPPROC:=CP  (* amend top priority pointer *)
    ELSE (* find correct place to insert descriptor *)
         PREVIOUS:=S1;
         S1:=S1^.next;
         WHILE ((S1^.prior >= PRIORITY) AND (S1 <> TOPPROC)) DO
          PREVIOUS:=S1;
          S1:=S1^.next
         END
  END;
  WITH CP^ DO  (* link descriptor into ring *)
   next:=PREVIOUS^.next;
   PREVIOUS^.next:=CP;
   ready:=TRUE;
   prior:=PRIORITY;
   slice:=prior;
   queue:=NIL;
  END;
  NEWPROCESS(P, WSP, N, CP^.cor);
  S1:=TOPPROC;
  WHILE (NOT S1^.ready) DO  (* search for ready process *)
   S1:=S1^.next             (* with highest priority    *)
  END;
  CP:=S1;
  TRANSFER(S0^.cor, CP^.cor)
 END StartProcess;

PROCEDURE StopProcess;
 (* Terminate the current process *)
 VAR S0,NEXTJOB: SIGNAL;  (* temporary *)
 BEGIN
  IF CP = CP^.next
    THEN (* END OF PROGRAM *)
         HALT
  END;
  S0:=CP;
  REPEAT  (* find next ready-to-run process *)
   CP:=CP^.next;
  UNTIL CP^.next = S0;
  CP^.next:=S0^.next;  (* remove descriptor from ring *)
  IF S0 = TOPPROC
    THEN (* process with top priority terminated *)
         TOPPROC:=TOPPROC^.next
    END;
  NEXTJOB:=TOPPROC;
  (* find next ready-to-run process *)
  WHILE ((NOT NEXTJOB^.ready) AND (NEXTJOB^.next <> TOPPROC)) DO
   NEXTJOB:=NEXTJOB^.next
  END;
  IF NEXTJOB^.ready
    THEN CP:=NEXTJOB;
         TRANSFER(S0^.cor, CP^.cor)
    ELSE (* DEADLOCK *)
         HALT
  END
 END StopProcess;
```

```
PROCEDURE SEND(VAR S:SIGNAL);
 (* Reactivate a process WAITing on S *)
 VAR S0: SIGNAL;  (* temporary, signalling process *)
 BEGIN
  IF S <> NIL
    THEN (* a process is WAITing *)
         S0:=CP;
         CP:=S;
         WITH CP^ DO  (* mark signalled process as active *)
          S:=queue; ready:=TRUE; queue:=NIL
         END;
         TRANSFER(S0^.cor, CP^.cor)
  END
 END SEND;

PROCEDURE WAIT(VAR S:SIGNAL);
 (* WAIT for some process to SEND S *)
 VAR S0,S1: SIGNAL;  (* temporary *)
 BEGIN
  IF S = NIL
    THEN (* first such process to WAIT *)
         S:=CP
    ELSE (* add to existing queue *)
         S0:=S;
         S1:=S0^.queue;
         WHILE S1 <> NIL DO  (* search for tail *)
          S0:=S1;
          S1:=S0^.queue
         END;
         S0^.queue:=CP
  END;
  CP^.ready:=FALSE;  (* deactivate process *)
  S0:=CP;
  S1:=TOPPROC;
  (* find next ready-to-run process *)
  WHILE ((NOT S1^.ready) AND (S1^.next <> TOPPROC)) DO
   S1:=S1^.next
  END;
  IF S1^.ready
    THEN CP:=S1;
         TRANSFER(S0^.cor, CP^.cor)
    ELSE (* DEADLOCK *)
         HALT
  END
 END WAIT;

(* Function *) PROCEDURE Awaited(S:SIGNAL): BOOLEAN;
 (* Any process WAITing on S *)
 BEGIN
  RETURN S <> NIL
 END Awaited;

PROCEDURE Init(VAR S:SIGNAL);
 (* Initialise S *)
 BEGIN
  S:=NIL
 END Init;

PROCEDURE Reschedule;
 (* Interrupt driven process switch. Find next ready-to-run
    process in the process ring and activate it *)
 BEGIN
  CP^.cor:=Userprocess;
  CP^.slice:=CP^.slice - 1;  (* decrement timeslice *)
  IF CP^.slice = 0
    THEN (* timeslice complete *)
         CP^.slice:=CP^.prior;  (* new timeslice *)
         IF CP <> CP^.next
           THEN REPEAT  (* find next ready-to-run process *)
                  CP:=CP^.next
                UNTIL CP^.ready;
                Userprocess:=CP^.cor
           END
    END
 END Reschedule;

BEGIN  (* SYNCHRO *)
 ALLOCATE(CP, TSIZE(ProcessDescriptor));
 WITH CP^ DO  (* enter main program into ring *)
  next:=CP; ready:=TRUE; prior:=2; queue:=NIL;
 END;
 TOPPROC:=CP
END SYNCHRO;
```

```
PROCEDURE CLOCK;
 (* Interrupt driver *)
 BEGIN
  Attach(Clock1Vector);
  LOOP
   IOTRANSFER(CLK, Userprocess, Clock1Vector);
   Reschedule
  END
 END CLOCK;


PROCEDURE CLKinit;
 (* Start the SAGE IV clock
    See "SAGE II User's Manual
         section IV.4.6 System Clock Access" *)
 VAR P: CARDINAL;
 BEGIN
  P:=SetPriority(ClockPriority);
  NEWPROCESS(CLOCK, ADR(CLKWSP), SIZE(CLKWSP), CLK);
  P:=SetPriority(P);
  T[0]:=0;
  T[1]:=6400;  (* 100 millisecond timeslice *)
  TRANSFER(Userprocess, CLK);
  CP^.cor:=Userprocess;
  UnitWrite(ClockDevice, ADR(T), 0, 0, Event1Number,
                                       {StartClock})
 END CLKinit;

PROCEDURE CLKterm;
 (* Stop the SAGE IV clock
    See "SAGE II User's Manual
         section IV.4.6 System Clock Access" *)
 BEGIN
  UnitWrite(ClockDevice, NIL, 0, 0, Event1Number,
                                       {StopClock});
  Detach(Clock1Vector);
  ClearVector(NIL, NIL, Clock1Vector)
 END CLKterm;

BEGIN  (* Processes *)
 SetEnvelope(CLKinit, CLKterm, FirstCall)
END Processes.
```