CONCURRENCY IN MODULA-2

Thesis
Submitted By

DAVID ANDREW SEWRY

In Fulfilment
Of The Requirements
For The Degree

MASTER OF SCIENCE

Rhodes University
June, 1984

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and gratitude to Professor P. D. Terry for being so willing and helpful throughout the duration of this project and for conducting that arduous task of reading the rough drafts of this thesis.

Thanks also to Alan - my critic and eternal listener, who always provided that necessary amount of motivation and inspiration needed to see me through the bad patches I experienced.

My final thanks must go to my parents for their consistent support and encouragement over the past years.

I would also like to acknowledge the financial assistance of Rhodes University, the Council for Scientific and Industrial Research, and Sasol.

CONTENTS

INTRODUCTION

1. Introduction

A concurrent program is one in which a number of processes are considered to be active simultaneously. It is possible to think of a process as being a separate sequential program executing independently of other processes, although perhaps communicating with them at desired points.

The concurrent program, as a whole, can be executed in one of two ways:

i) in true concurrent manner, with each process executing on a dedicated processor

ii) in a quasi-concurrent manner, where a single processor's time is multiplexed between the processes.

There are two motivations for the study of concurrency in programming languages:

i) concurrent programming facilities can be exploited in systems where one has more than one processor. As technology improves, machines having multiple processors will proliferate

ii) concurrent programming facilities may allow programs to be structured as independent, but co-operating, processes which can then be implemented on a single processor system. This structure may be more natural to the programmer then the traditional sequential structures. An example is provided by Conway's

problem [Ben82].

Clearly, by their very nature, traditional sequential-type languages (Fortran, Basic, Cobol and earlier versions of Pascal) prove inadequate for the purposes of concurrent programming without considerable extension (which some manufacturers have provided, rendering their compilers non standard-conforming). The general convenience of high level languages provides strong motivation for their development for real time programming.

Modula-2 [Wir83] is but one of a number of such recently developed languages, designed not only to fulfil a "sequential" role but also to offer facilities for concurrent programming. Developed by Niklaus Wirth in 1979 as a successor to Pascal and Modula, it is intended to serve under the banner of a general-purpose systems-implementation language.

This thesis investigates concurrency in Modula-2 and takes the following form:

i) an analysis of the concurrent facilities offered
ii) problems and difficulties associated with these facilities
iii) improvements and enhancements, including the feasibility of using Modula-2 to simulate constructs found in other languages, such as the Hoare monitor [Hoa74] and the Ada rendezvous [Uni81].

Each section concludes with an appraisal of the work conducted in that section.

The final section consists of a critical assessment of those Modula-2 language constructs and facilities provided for the implementation of concurrency and a brief look at concurrency in Modula, Modula-2's predecessor.

CONCURRENT LANGUAGES

2. Concurrent Languages

2.1 Languages

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

Ada		[Uni81]
Clang		[Cha84]
Concurrent Euclid		[Hol83]
Concurrent Pascal		[Bri75]
CSP		[Hoa78]
Edison	[Bri81],	[Bri82]
Extended Pascal-S	[Ben81],	[Cha82]
Modula		[Wir77]
Modula-2		[Wir83]
Pascal-Plus	[Wel79],	[Wel80]
UCSD Pascal IV.0		[Sof81]

In addition to being able to execute multiple processes simultaneously, most of these languages have presented facilities for inter-process communication, as well as constructs which will ensure mutual exclusion of processes to critical regions.

Various constructs have been defined for inter-process communication, each employed by one or more of these languages:

Semaphores -
Clang, Extended Pascal-S, UCSD Pascal IV.0

Condition Variables -
Concurrent Euclid, Concurrent Pascal, Modula, Modula-2, Pascal-Plus

Conditional Critical Regions -
Edison

Monitors -
Clang, Concurrent Euclid, Concurrent Pascal, Modula, Modula-2, Pascal-Plus

Rendezvous -
Ada, CSP

2.2 Implementations

When including concurrent programming facilities in a language, a designer has two options:

i) include concurrency as an intrinsic feature of the language implementation

ii) include a separately compilable library module which will provide concurrency.

It has been the policy of many designers to employ the first option. This method is, however, not without disadvantages:

i) it requires some underlying "run-time system" to schedule processes for execution and (on a single processor system) to simulate concurrent execution by switching the processor between processes. This run-time system will be resident in memory whether the user invokes concurrent execution or not. This is, of course, no problem for the "concurrent" programmer, but clearly an unnecessary overhead for the programmer not the slightest bit interested in concurrency.

ii) it may not provide for any flexibility in scheduling policy. The user is typically presented with a fixed,

unchangeable policy and any job requiring a specific strategy cannot be accommodated.

The library module approach solves both of these problems. Concurrency remains available for use but does not affect the system if it is not used. Specific forms of process scheduling can be implemented by merely creating an appropriate library module.

This alternative approach to the implementation of concurrency has not attracted widespread attention. Compared to previous implementations of concurrency, an implementation of this nature might well be construed as unusual.

2.3 Modula-2

The designer of Modula-2 has adopted the library module method and has included, to a greater or lesser degree, all the constructs necessary for concurrent programming.

A suggested library module has been made available in the standard reference work [Wir83] and has enabled the author to conduct a detailed examination of concurrency in Modula-2, manipulate certain aspects, include modifications and introduce entirely different constructs for concurrency.

Modula-2 is very much like Pascal, its predecessor, and should present few problems to the accomplished Pascal programmer. Only minor differences can be detected in both the data and control structures. However, a number of new concepts have been

introduced:

a) modules: a program can be divided into a number of modules, each containing a section to describe the interface between the module and the rest of the program. This provides a powerful form of data abstraction.
b) clearly defined standards for separate compilation
c) clearly defined standards for module libraries which facilitate efficient program development
d) standard utility modules
e) low-level machine access
f) coroutines and interrupts: to facilitate concurrent programming
g) procedure variables

Modula-2 has thus extended Pascal in both directions: upwards to encapsulate systems design (modules, separate compilation) and downwards (low-level machine access, coroutines and interrupts) [Wir83], [Vol83].

MODULA-2 PROCESS FACILITIES

3. Modula-2 Process Facilities

3.1 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 IV.0	[Sof81]

In this section 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. What is supplied is an implementation of the coroutine construction, 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 implementations of the language:

i) Apple - running under the Apple Pascal System

ii) SAGE IV - running under the UCSD Pascal II.0 System

iii) VAX-11 and PDP-11 system

In the discussion which follows we shall point out some difficulties with this system, and suggest some enhancements.

3.2 Wirth's Processes Module

The module Processes offers the following facilities:

```
PROCEDURE StartProcess(P:PROC; n:CARDINAL)
PROCEDURE SEND(VAR S:SIGNAL)
PROCEDURE WAIT(VAR S:SIGNAL)
PROCEDURE Awaited(S:SIGNAL):BOOLEAN
PROCEDURE Init(VAR S:SIGNAL)
```

where PROC is a parameterless procedure and
SIGNAL is a pointer to a process descriptor

A call StartProcess(P,n) starts the execution of a process P which is expressed as a procedure of the same name, and allocates a workspace area of size n words for allocation of variables.

Communication between processes occurs in two ways:

a) shared variables
b) signals (semaphores)

Synchronisation of processes is achieved by use of signals. Operations that can be performed on signals (s) are:

a) SEND - reactivate a process waiting for s
b) WAIT - wait for some process to SEND s
c) Awaited - true if at least 1 process is waiting for s
d) Init - compulsory initialisation

The integrity of shared variables is guaranteed by placing them in a monitor. A monitor is a module which guarantees mutual exclusion of processes and, hence, integrity of its local data.

By specifying a priority in a module's heading it is designated a monitor, eg:

```
MODULE ModName[1];
```

At this point it will suffice to know that such a construct will set up a monitor and provide the necessary protection. (The significance of the priority value selected and its effect on the IOTRANSFER command will be discussed later. See subsection 3.6 "Degree of Concurrency")

A silly example of the use of these facilities follows:

```
MODULE PRODCONS;

FROM Processes IMPORT SIGNAL,Init,WAIT,SEND,StartProcess;
FROM SomeWhere IMPORT ProduceItem, PlaceInStore,
                      ConsumeItem, RemoveFromStore;

VAR Canstore,Cantake : SIGNAL;
    Instore : BOOLEAN;

 PROCEDURE Produce;
 VAR I : CARDINAL;
 BEGIN
  FOR I:=1 TO 2 DO
   ProduceItem;
   IF Instore THEN WAIT(Canstore) END;
   PlaceInStore;
   Instore:=TRUE;
   SEND(Cantake)
  END
 END Produce;
```

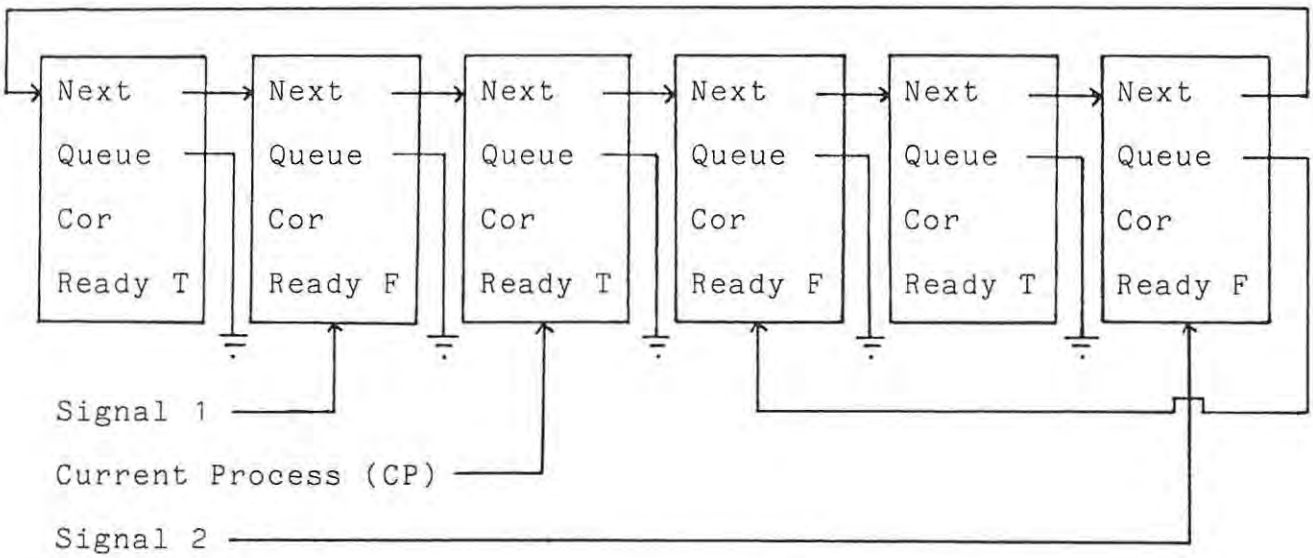
```
 PROCEDURE Consume;
 VAR I : CARDINAL;
 BEGIN
  FOR I:=1 TO 2 DO
   IF NOT Instore THEN WAIT(Cantake) END;
   RemoveFromStore;
   ConsumeItem;
   Instore:=FALSE;
   SEND(Canstore)
  END
 END Consumer;

BEGIN
 Init(Cantake); Init(Canstore);
 Instore:=FALSE;
 StartProcess(Produce, 100);
 StartProcess(Consumer, 100);
END PRODCONS.
```

The suggested way in which these facilities are to be implemented is as follows:

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

VAR CP: SIGNAL; (* pointer to current process *)
```

When a process is started 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 we have subsidiary queues to represent signals.

For example:



where we have one process waiting on Signal 1, and two processes waiting on Signal 2. Ready indicates whether the process is WAITing on a SIGNAL or not. Depending on the state of Ready the process may or may not be activated. The main program is also considered to be a process. Its Processdescriptor is set up and inserted into the ring as part of the initialisation code for the module Processes.

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.

The operations that are performed on the schedule ring include:

i) simple insertion of ProcessDescriptors (StartProcess)

ii) direct access of particular ProcessDescriptors (SEND)

iii) additions to subsidiary queues, (contained within the main ring, representing SIGNAL's) and a simple forward search of the ring to find a ready-to-run process (WAIT)

When a process calls procedure WAIT its ProcessDescriptor is not removed from the ring. It is merely added to one of the subsidiary queues contained within the main ring. This obviates the necessity to reinsert the ProcessDescriptor when the process is reactivated.

That concludes the discussion of process facilities as suggested by Wirth. Source code can be found in Appendix A, or alternatively [Wir83].

3.3 Process Termination

The first difficulty with this system is that it does not clarify the action to be taken when a process terminates. (For example, in the program above the Producer may terminate before the Consumer has had a chance to take the last item). In Wirth's report [Wir83] no mention is made of this point at all; various implementations have acted in different ways:

For example:

i) Report on the Programming Language - Modula-2 [Wir83]

No mention is made of coroutine termination.

ii) Programming in Modula-2 [Wir83]

"coroutines are started by an explicit transfer; they must be terminated by such a transfer." Where control is transferred to upon termination is not stated. To other unfinished coroutines? What happens when the last coroutine terminates?

iii) Manual for the SAGE IV implementation of Modula-2 [Vol83]

"a program is terminated if any coroutine reaches the end of its procedure body"

iv) Manual for the VAX-11 implementation of Modula-2 [Ham83]

The only reference to this problem is found as a fatal run-time diagnostic message:

"End of Coroutine"

v) Extension of Pascal by Coroutines and its Application to Quasi-parallel Programming and Simulation [Kri80]

"according to coroutine logic, a coroutine should not execute the anonymous end-and-return statement of the underlying procedure, the effect of which is undefined (a runtime error occurs)"

To this end they have introduced a procedure ENDCOR which must be called at the end of a coroutine to indicate that it has terminated execution. The procedure

has the following form:

Procedure ENDCOR (CIDN:CORID)

where CIDN is the identifier of a coroutine to which control will be transferred. The final ENDCOR call will transfer control to the main program which is then at liberty to terminate.

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 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 process descriptor it will also search the schedule ring for the next process which is ready-to-run.

The code for such a procedure might look as follows:

```
PROCEDURE StopProcess;
VAR SØ,NEXTJOB : SIGNAL;
    NOTFOUND : BOOLEAN;
BEGIN
 IF CP = CP^.next
   THEN (* last process therefore end of program *)
        HALT
 END;
 SØ:=CP; NEXTJOB:=CP; NOTFOUND:=TRUE;
 REPEAT
  CP:=CP^.next;
  IF ((NOTFOUND) AND (CP^.READY))
    THEN NEXTJOB:=CP;
         NOTFOUND:=FALSE
  END
 UNTIL CP^.next = SØ;
```

```
    CP^.next:=S0^.next;
    IF NOTFOUND
      THEN (* no ready-to-run processes: DEADLOCK *)
           HALT
      ELSE CP:=NEXTJOB;
           TRANSFER(S0^.cor, CP^.cor)
    END
  END StopProcess;
```

The process descriptor to be removed from the ring is pointed to by a global variable CP (of type SIGNAL). The StopProcess procedure will traverse the ring to find the process immediately "behind" the current process and also the process which can be activated next. If there is only 1 process descriptor left in the ring the program as a whole can be terminated. Otherwise we remove the current process descriptor and activate the next ready process. Naturally it is possible for no processes to be ready-to-run (they are all WAITing on some signal), in which case "deadlock" has occurred.

All processes are now in a position to complete execution and do not terminate as soon as one process terminates.

(Note that it is also necessary to have a call to StopProcess at the end of the main program - remember it is merely considered to be another process)

3.4 The COBEGIN...COEND Construct

In several discussions on concurrent programming the CEBEGIN...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 usind 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 that semicolons in general 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, and the code follows as:

The COBEGIN and COEND procedures follow:

```
PROCEDURE COBEGIN;
BEGIN
 ALLOCATE(CP,TSIZE(ProcessDescriptor));
 WITH CP^ DO
  next:=CP; ready:=true; queue:=NIL
 END;
```

```
 Main:=CP
END COBEGIN;

PROCEDURE COEND;
BEGIN
 StopProcess
END COEND;
```

and

```
PROCEDURE StopProcess;
VAR SØ,NEXTJOB: SIGNAL;
    NOTFOUND: BOOLEAN;
BEGIN
 IF CP = CP^.next
   THEN TRANSFER(CP^.cor, Main^.cor)
   ELSE SØ:=CP; NEXTJOB:=CP; NOTFOUND:=TRUE;
        REPEAT
         NEXTJOB:=NEXTJOB^.next;
         IF ((NOTFOUND) AND (NEXTJOB^.ready))
           THEN CP:=NEXTJOB;
                NOTFOUND:=FALSE
         END
        UNTIL NEXTJOB^.next = SØ;
        IF NOTFOUND
          THEN (* DEADLOCK *)
               HALT
          ELSE NEXTJOB^.next:=SØ^.next;
               TRANSFER(SØ^.cor, CP^.cor)
        END
 END
END StopProcess;
```

where Main (of type SIGNAL) is a pointer to the main program and Current is a pointer to the currently executing process.

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

The COBEGIN...COEND construct does allow the user to have the following trivial case:

```
.
.
COBEGIN;
COEND;
.
.
```

Unfortunately the above implementation produces a run-time error if no processes are launched between the two procedure calls. The problem arises from an attempt to transfer control to the main program using the .cor field of the main program's ProcessDescriptor, which has yet to be assigned anything (TRANSFER(CP^.cor, Main^.cor)). This is normally done when the first StartProcess is executed - the TRANSFER statement does the assignment.

The solution is to launch a dummy process from within COBEGIN. Its only statement will be a call to StopProcess that will, amongst other tasks, assign the correct value to Main^.cor.

```
PROCEDURE Dummy;
BEGIN
 StopProcess
END Dummy;

PROCEDURE COBEGIN;
BEGIN
 .
 .
 StartProcess(Dummy, 400)
End COBEGIN;
```

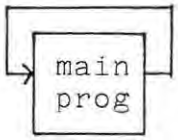
An implementation of the COBEGIN . . . COEND construct can be found in Appendix C.

3.5 Process Priorities

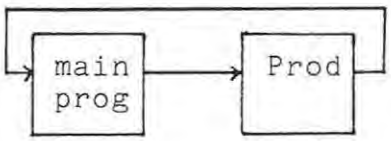
Up until now the schedule ring has been very simple: ProcessDescriptors are linked into it just next to the "launching" process in every case. In the example that was given (Producer/Consumer) the order of the ring was not an issue. In fact, the user has no way of affecting the order of

the ring other than by a complex arrangement of calls to StartProcess, SEND and WAIT.

In the Producer/Consumer program the ring is constructed as follows:

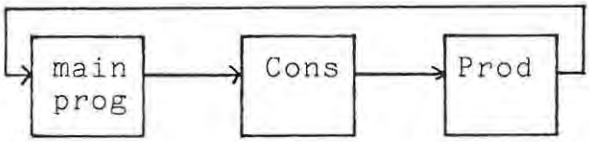


initialisation code of Module Processes



StartProcess(Producer, 100)

the WAIT(Canstore) in Produce transfers control back to the main program



StartProcess(Consumer, 100)

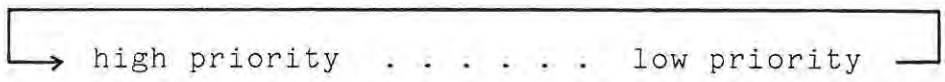
The scheduling policy has also been simple: at any process switch simply select the next ready-to-run process around the ring. (Notice that at present this switching only occurs as a result of an explicit WAIT or SEND or StartProcess call). The system does not really allow for any form of scheduling policy based on priority, time slicing etc.

Were a priority to be assigned to each process, scheduling

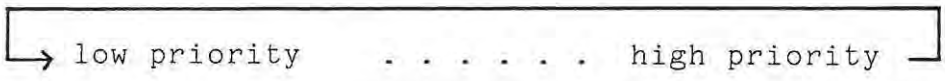
policies could be developed. Such policies might favour higher priority processes with either (i) greater amounts of processor time or (ii) more immediate attention on any process switch.

If at the point of initiation each process is assigned a priority, the ring can be arranged in either descending or ascending order of priority, eg:

i) descending



ii) ascending



The descending order is preferred as it would appear to be the most natural.

If we set aside for the moment the problem of scheduling strategies and concentrate on how the ring is extended and contracted by StartProcess and StopProcess we see that an extra field in the ProcessDescriptor is required:

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

The call to StartProcess now includes a parameter to indicate the priority of the process:

```
PROCEDURE StartProcess(P:PROC; n:CARDINAL; PRIORITY:INTEGER);
VAR SØ,S1,PREVIOUS : SIGNAL;
    wsp : ADDRESS;
BEGIN
 SØ:=CP; S1:=TOPPROC;
 ALLOCATE(CP,TSIZE(ProcessDescriptor));
 ALLOCATE(wsp,n);
 IF S1^.prior < PRIORITY
   THEN REPEAT
         S1:=S1^.next
        UNTIL S1^.next = TOPPROC;
        PREVIOUS:=S1;
        TOPPROC:=CP
   ELSE PREVIOUS:=S1;
        S1:=S1^.next;
        WHILE ((S1^.prior >= PRIORITY) AND (S1 <> TOPPROC))
         DO PREVIOUS:=S1;
            S1:=S1^.next
        END
 END;
 WITH CP^ DO
  next:=PREVIOUS^.next;
  PREVIOUS^.next:=CP;
  ready:=TRUE;
  prior:=PRIORITY;
  queue:=NIL
 END;
 NEWPROCESS(P, wsp, n, CP^.cor);
 TRANSFER(SØ^.cor, CP^.cor)
END StartProcess;
```

where TOPPROC (of type SIGNAL) is a pointer to the process that currently has the highest priority.

(We are assuming that low values of PRIORITY indicate low priorities)

If the process with the highest priority terminates it will be necessary to update TOPPROC to point to the process that now has the highest priority. The change is needed in StopProcess:

```
PROCEDURE StopProcess;
.
.
BEGIN
 .
 .
 IF SØ = TOPPROC
   THEN TOPPROC:=TOPPROC^.next
 END;
 .
 .
END StopProcess;
```

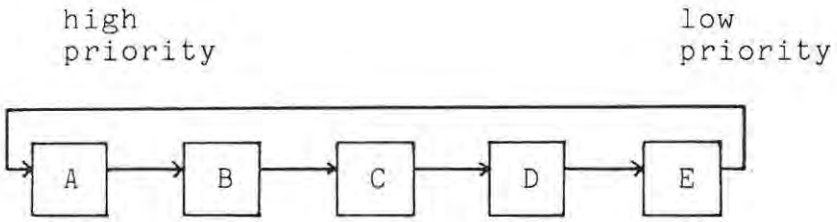
since the schedule ring is ordered, we merely move to the next element.

Two methods of selecting the next process to activate come to mind:

a) starting from the current process and moving to its "right" select the next ready-to-run process, ie. allocate each process some processor time working from high priority processes to low priority processes.

   Unfortunately this method will provide no better scheduling than we had before, notwithstanding the fact that the ring is now ordered. The reason is that both StartProcess and SEND transfer control to the process signalled and WAIT still works its way down the ring.

   High priority processes are not necessarily given any preference. Suppose five processes are currently represented in the ring:



B and C could alternatively WAIT and SEND to one another and the other processes would never execute.

b) always select the process with the highest priority to activate next. This method could result in a very low priority process being indefinitely postponed.

To employ a) no further changes are needed at this stage to the procedures developed.

For the second method we need all the changes required by the first in addition to some others.

Since this method has to do with continual preference being given to the process with the highest priority, a closer look needs to be taken of all the circumstances that could precipitate a context switch. Context switches can occur at the following places:

i) PROCEDURE StartProcess

ii) PROCEDURE StopProcess

iii) PROCEDURE WAIT

iv) PROCEDURE SEND

Procedure StartProcess has always transferred control to the newly initiated process after it has been created. If this were

allowed to continue we could effectively initiate a very low priority process which would execute, maybe to completion, regardless of any high priority processes which might already have been created. So instead of transferring control to the newly initiated process, the ring is searched for the ready process with the highest priority, to which control is then transferred.

The new procedure StartProcess:

```
PROCEDURE StartProcess(P:Proc; n:CARDINAL; PRIORITY:INTEGER);
.
.
BEGIN
 .
 .
 S1:=TOPPROC;
 WHILE (NOT S1^.ready) DO
  S1:=S1^.next
 END;
 CP:=S1;
 TRANSFER(SØ^.cor, CP^.cor)
END StartProcess;
```

There will always be at least one process that is ready - the newly initiated process. Unfortunately it is not possible to transfer control to the process pointed to by TOPPROC immediately - it may be WAITing on a signal.

Similar changes are required in both StopProcess and WAIT. Again we do not wish to merely transfer control to the next ready process, but rather to the ready process with the highest priority.

```
PROCEDURE StopProcess;
.
.
BEGIN
 .
 .
 NEXTJOB:=TOPPROC;
 WHILE ((NOT NEXTJOB^.ready) AND
        (NEXTJOB^.next <> TOPPROC)) DO
  NEXTJOB:=NEXTJOB^.next
 END;
 IF NEXTJOB^.ready
   THEN CP:=NEXTJOB;
        TRANSFER(SØ^.cor, CP^.cor)
   ELSE (* DEADLOCK *)
        HALT
 END
END StopProcess;

PROCEDURE WAIT(VAR S:SIGNAL);
.
.
BEGIN
 .
 .
 S1:=TOPPROC;
 WHILE ((NOT S1^.ready) AND (S1^.next <> TOPPROC)) DO
  S1:=S1^.next
 END;
 IF S1^.ready
   THEN CP:=S1;
        TRANSFER(SØ^.cor, CP^.cor)
   ELSE (* DEADLOCK *)
        HALT
 END
END WAIT;
```

In both instances it is possible that there may not be any ready processes, in which case DEADLOCK has arisen.

The SEND(S) concept enjoys various possible semantic definitions:

a) transfer control from the signalling to the signalled process - the signaller relinquishes control totally.
b) transfer control from the signalling to the signalled process and place the signaller on a top priority queue to

be activated as soon as the signalled process terminates or WAIT's - control is temporarily relinquished but recalled at the earliest convenience.

c) note that the signalled process is now ready-to-run but allow the signaller to maintain control. When the signaller process terminates or WAIT's, transfer control to the previously signalled process - control is maintained for as long as possible. There is, however, the possibility that the conditions that gave rise to the SEND(S) are no longer true by the time the previously signallled process gains control.

This implementation employs method a). If one looks at the definition of SEND(S) at the beginning of this report, control is transferred to the process at the head of the queue pointed to by S or, if the queue is empty (no processes WAITing on S), the SENDing process maintains control. Since the process which must be activated is given by the top element of the queue, no search of the ring is required to determine the ready-to-run process with the highest priority. Consequently no changes are needed in procedure SEND.

The call to StartProcess will now take the following form:

```
eg: StartProcess(SomeProcedure, 400, 5)
```

Two interesting points follow:

a) now that control is preferentially transferred to the process with the highest priority there exists a

temptation to assume that this selfsame process will be the first to terminate. Consider the situation where a low priority process is forced to WAIT, after which a higher priority process is initiated. It then executes a SEND which awakens that low priority process which might then terminate. Hence the reason why the changes made to StopProcess in the first method still remain in force.

b) since the main program is considered as just another process, what priority should it be assigned? It would seem reasonable to assume that most, if not all, processes will be initiated by the main program, (although nothing stops a process initiating a process) and that the sooner they are initiated the better, so that the main program should be assigned a fairly high priority to enable it to complete execution soonest. This would seem to be in accordance with the whole idea of concurrency - all processes initiated and executing simultaneously or as close to that ideal as possible. Naturally this policy should not be totally inflexible. If for some reason a process must start execution straight away, the priority of the main program could be lowered to allow this to happen. The priority of the main program is set as part of the initialisation code of module Processes.

3.6 Degree of Concurrency

In the implementation discussed previously, context switching can only occur when an executing process either initiates

another process, or executes a WAIT or SEND operation. This can severely limit the degree of "concurrency" which can be achieved. (In an extreme case one obtains essentially none at all.)

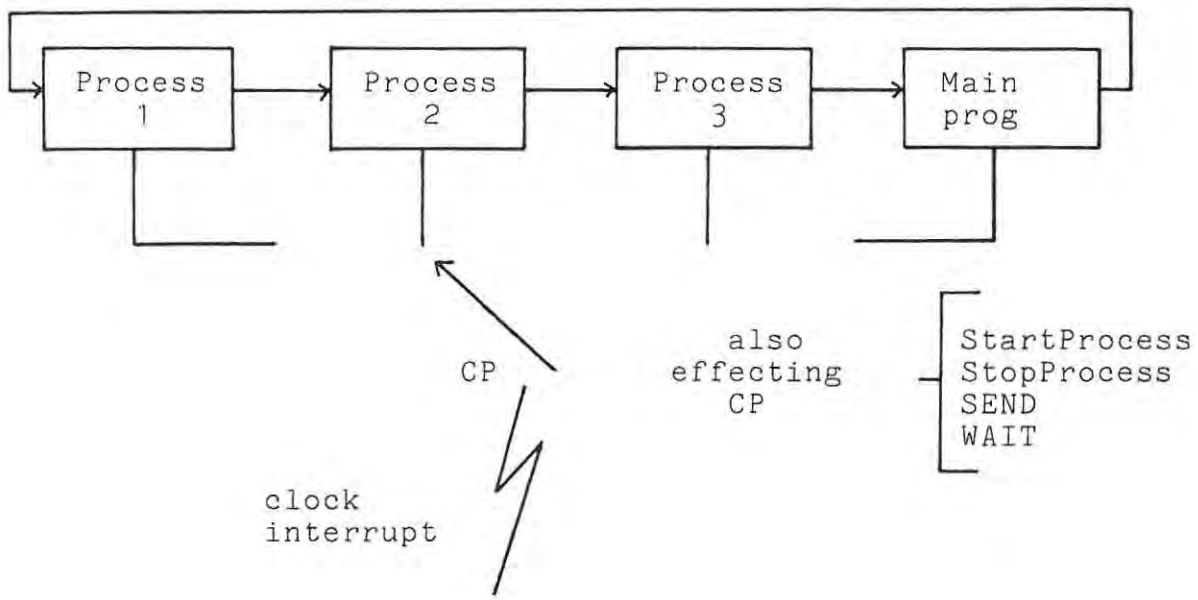
To enhance the degree of concurrency one might either:

a) introduce time shared process scheduling as an intrinsic feature of the implementation, with implementation details hidden from the user
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 what follows we describe how a clock interrupt system could be added to the system developed so far. (With the introduction of interrupts a certain amount of machine dependent information becomes unavoidable: particular details here are for the Volition systems implementation of Modula-2 on the SAGE IV microcomputer. See Appendix F)

Diagrammatically, the module should provide a context switch in the following instances:



The reader has already been introduced to the TRANSFER command which takes the form:

TRANSFER(A, B)

which will transfer control from process A to process B. If one wishes to return control to process A from process B, it must be done explicitly with the command:

TRANSFER(B, A)

Suppose one wished to transfer control from process A to process B and upon occurrence of an interrupt, return control back to process A. Since the interrupt is by definition an unscheduled event, the above command will not do. There is no way of predicting the occurrence of an interrupt to be able to place the relevant TRANSFER to effect the necessary switch in

control.

A further low level procedure, IOTRANSFER is provided to handle interrupts. An IOTRANSFER procedure call takes the form

IOTRANSFER(A, B, InterruptVector)

Execution of this statement causes control to be transferred from A to B, but upon occurrence of an interrupt via (a machine dependent) InterruptVector, control will be transferred back to process A (A is a process identifier for the process in which the IOTRANSFER statement occurs).

So, to produce some form of scheduling the following loop can be set up:

```
LOOP
 IOTRANSFER(CLK, Userprocess, Clock1Vector);
 Reschedule
END
```

where control will be transferred from process CLK (where this loop will be found) to process Userprocess, which would execute until the clock interrupted via Clock1Vector. Reschedule would then select the next process to activate, assign it to userprocess and the loop would start up again.

A simple algorithm for Reschedule might be (remembering that the currently executing process is pointed to by CP):

```
PROCEDURE Reschedule;
BEGIN
 CP^.cor:=Userprocess;
 REPEAT
  CP:=CP^.next
 UNTIL CP^.ready;
 Userprocess:=CP^.cor
END Reschedule;
```

Finally, because the interrupt has been introduced certain critical data, namely the schedule ring, must be protected. In short, any procedure that tampers with the schedule ring must be placed in a monitor (MODULE SYNCHRO[4]). The overall structure of module Processes now has the following form:

```
IMPLEMENTATION MODULE Processes;
.
.
 MODULE SYNCHRO[4];
   PROCEDURE StartProcess
   PROCEDURE StopProcess
   PROCEDURE SEND
   PROCEDURE WAIT
   PROCEDURE Awaited
   PROCEDURE Init
   PROCEDURE Reschedule

PROCEDURE CLOCK
PROCEDURE CLKinit  (* Start the clock *)
PROCEDURE CLKterm  (* Stop the clock  *)
```

The choice of the priority attached to module SYNCHRO, namely 4, is important. The reader should note that the clock is assigned a priority of 3 (see Appendix G). The effect of choosing a lower priority will ensure that the processor is not interrupted by the clock while it is busy within module SYNCHRO. If the clock does interrupt whilst a process is busy executing within module SYNCHRO, the first interrupt will be queued, but any further clock interrupts will be ignored (this is a feature of the implementation of the language).

The processor will, however, be interrupted if an IOTRANSFER, with a priority in excess of that which has been assigned to module SYNCHRO, is executed. Great care should thus be taken when choosing the various device and module priorities.

The question of user process priorities now becomes important. As it stands, when either StartProcess or StopProcess or WAIT is executed, the Processes module will always activate the process with the highest priority. However, Reschedule will work its way around the ring.

It is not much use altering Reschedule to activate the process with the highest priority. 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.

The alternatives are:

a) to leave it as it is. This is quite acceptable and will provide a good distribution of processor time as well as a certain amount of favouritism in process choice.
b) to allow each process a maximum number of timeslices determined by its priority, eg. a process with a priority of 4 will be allowed a maximum of 4 contiguous timeslices.

The second alternative requires a decision to be made regarding timeslice manipulation:

What happens to the timeslice total when a process, busy using up its allotted timeslices, executes either a StartProcess or SEND or WAIT and is then subsequently activated? Will it be required to carry on with the rest of its timeslice from when it previously stopped or should it be allocated a fresh timeslice ? In the interests of ensuring that all processes, low and high priority, make reasonable progress, the policy has been adopted of not assigning a fresh timeslice. Continually reinstating the timeslice total would to a certain degree destroy the timeslice concept - a process could end up monopolising the processor for a significantly longer time than it ought to.

Adjustments are needed in:

i) the declaration of ProcessDescriptor

ii) PROCEDURE StartProcess

iii) PROCEDURE Reschedule

A field is needed to indicate how many timeslices remain for a particular process:

```
TYPE ProcessDescriptor =
       RECORD
         .
         .
         slice:INTEGER;
         .
       END;
```

Procedure StartProcess must initialise the number of timeslices

the process is allotted:

```
PROCEDURE StartProcess(P:PROC; n:CARDINAL; PRIORITY:INTEGER);
.
BEGIN
 .
 WITH CP^ DO
  .
  slice:=prior;
  .
 END;
.
END StartProcess;
```

Procedure Reschedule will manage the timeslices:

```
PROCEDURE Reschedule;
BEGIN
 CP^.cor:=Userprocess;
 CP^.slice:=CP^.slice - 1;
 IF CP^.slice = 0
   THEN CP^.slice:=CP^.prior;
        REPEAT
         CP:=CP^.next
        UNTIL CP^.ready;
        Userprocess:=CP^.cor
 END
END Reschedule;
```

The reader should note that no timeslice management has been included in the procedures SEND and WAIT. Consider the following situation:

A process executes either a SEND or a WAIT. It has by then used up a portion of its timeslice, a fact which may not have been noted. The newly activated process continues (timewise) from where the previous process left off and will only be allowed to execute for part of a timeslice. In effect, the one process has received a free portion and the other only part of a portion for which it has had to pay. However, at some stage the first process will be activated and second process will do the

activating and the roles may be reversed. The author believes that in the final analysis most processes will receive most of their allotted timeslices.

3.7 In Retrospect

The reader has now been introduced to quite a number of implementations of the module Processes and it may prove useful to trace their development:

1. Process facilities as provided by Wirth (Appendix A)
2. Modified to provide process termination facilities (Appendix B)
3. Modified to include process priorities (Appendix D)
4. Modified to present an alternative process scheduling strategy (always activate the process with the highest priority when the opportunity arises) (Appendix E)
5. Modified to include interrupt-driven process switching (Appendix G)
6. Modified to provide a timeslice mechanism (Appendix H)

3.8 Other Constructs

Modula-2 is capable of supporting many different constructs for process synchronisation, not only the one presented by Wirth. One such alternative was proposed by J. Hoppe.

In his article, "A Simple Nucleus Written in Modula-2: A Case Study" [Hop80], Hoppe presents a system nucleus which makes provision for:

a) process creation
b) process synchronisation
c) interrupt handling (using preemptive scheduling)

Process synchronisation is achieved by a "mailbox" through which messages are passed.

The operations defined on a mailbox are:

```
SendMsg(MailBox, msg) -

     if some process is waiting for a message
       then place the message in the MailBox;
            place the waiting process at the head of
              the queue of ready-to-run processes;
            transfer control to the process at the
              head of the ready-to-run queue, ie. the
              previously waiting process (process switch)
       else place the message in the MailBox;
            continue execution


WaitMsg(MailBox, msg) -

     if there are no messages in the MailBox
       then remove process from the ready-to-run
            queue and place the process on the
            MailBox queue;
```

transfer control to the process at the
head of the ready-to-run queue (process
switch)
else take the message from the MailBox;
continue execution

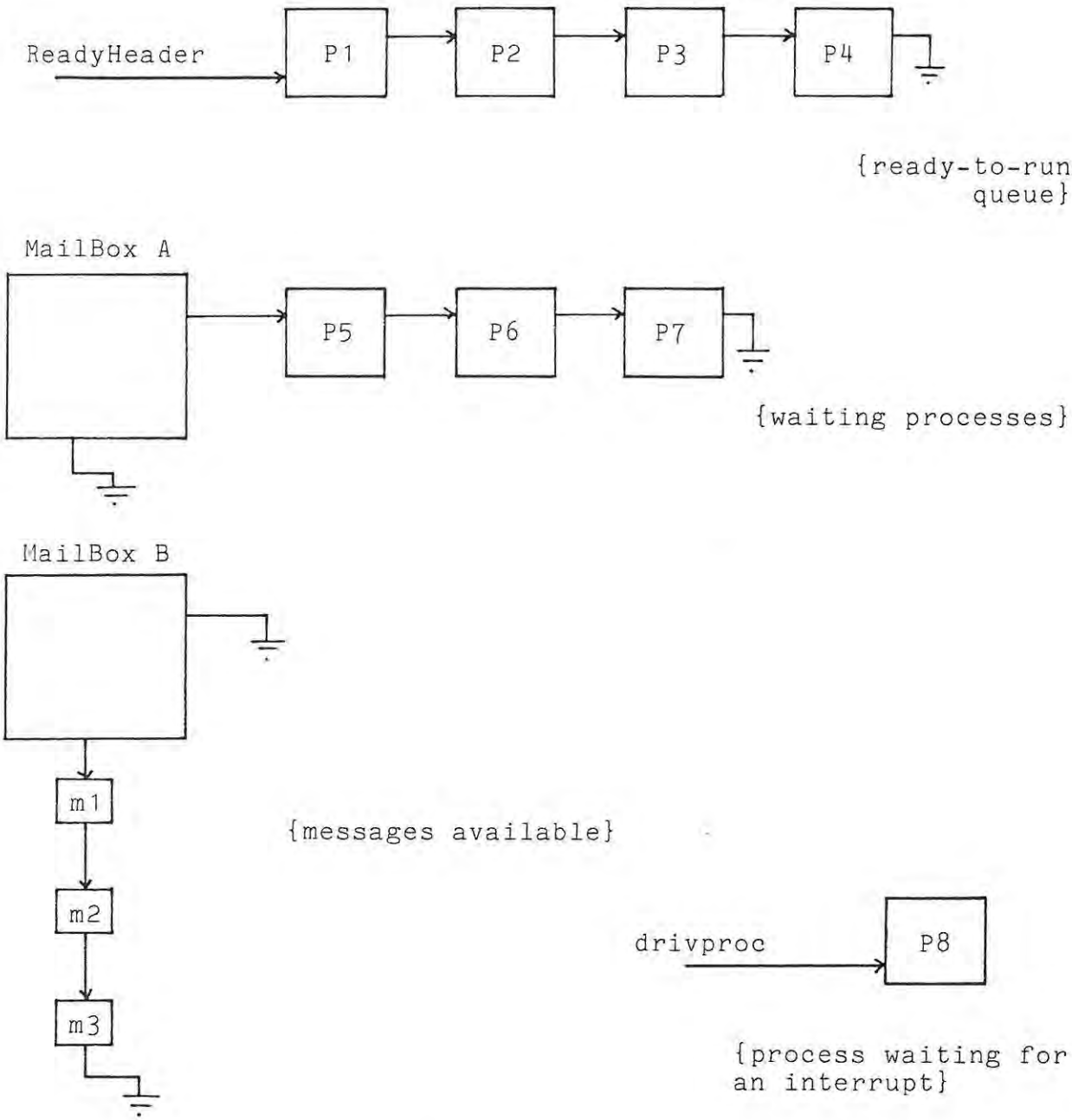
Interrupt handling is controlled by the WaitIO procedure:

WaitIO(vector) -

remove the process from the ready-to-run queue;
transfer control to the process at the head of
ready-to-run queue (process switch)

when the interrupt occurs, control is automatically
transferred back to the process waiting for the
interrupt (process switch);
the process is placed at the head of ready-to-run
queue;
the process continues execution

Diagrammatically, Hoppe's system looks as follows:



A listing of Hoppe's module Nucleus accompanied his article and was implemented by the author.

The following observations and comparison with Wirth's process facilities (as provided by Wirth [Wir83] and as modified by the

author - see Appendix G) should be noted:

a) Process Descriptors can be found in any one of a number of places:

   i) in the ready-to-run queue
   ii) waiting on a MailBox queue
   iii) waiting for an interrupt.

   The idea of a central process ring (presented by Wirth) seems more acceptable. This wide distribution of process descriptors leads to an excessive amount of process descriptor movement.

b) The SendMsg command always deposits a message in the MailBox, regardless of whether there are any processes waiting for messages or not.

   It is interesting to note that the majority of other synchronisation primitives act differently, ie. if no processes are waiting for the message (or signal or condition to become true), the message is disregarded and not queued.

   This strategy of remembering messages is very similar to the pending semaphore of CHILL [Bra82]. Unfortunately, this method is not without problems. It is possible that by the time a message is taken from the mailbox the conditions that gave rise to the message are no longer true.

   Consequently, careful use of the synchronisation primitives

is necessitated.

c) Most system nuclei include some form of process switching over and above that provided by some of the calls to WaitMsg and SendMsg, and all the calls to WaitIO, eg. an interrupt driven process switch, to increase the system's degree of concurrency

   Hoppe's nucleus does not include any form of interrupt driven process switching.

3.9 Conclusion

Initially a user is provided with a little less than the bare minimum required to achieve something which resembles, albeit somewhat remote, concurrency (I say 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 and by no means wonderful but

using what is given as a basis for development, they prove not to be beyond redemption.

MODULA-2
AND THE
MONITOR CONCEPT

4. Modula-2 and the Monitor Concept

4.1 Introduction

The monitor concept [Hoa74] is one approach towards ensuring a reliable concurrent programming environment. Although the processes constituting a concurrent program may declare individual data areas, a frequent occurrence is the declaration of a common data area to be accessed by several processes. Since the processes execute independently, it is possible that more than one process will attempt to access this shared data area simultaneously, with chaotic results. Clearly a form of manager is required to ensure an orderly access to this shared data area. One such manager is known as a monitor, one which will encapsulate the shared data area and the procedures that will act on this data.

In short, the monitor will provide mutual exclusion of processes to a set of procedures that act on the shared data and consequently ensure the integrity of that data.

Several recent languages have provided the monitor facility in their basic design:

Clang	[Cha84]
Concurrent Euclid	[Hol83]
Concurrent Pascal	[Bri75]
Modula	[Wir77]
Modula-2	[Wir83]
Pascal-Plus	[Wel79], [Wel80]

The language Modula-2 provides for a monitor in a limited environment.

It is the intention of this section to discuss the monitor as it has been provided by Wirth and to suggest an implementation of the monitor concept, as developed by Hoare [Hoa74], to suit a "real" environment, ie. an environment where, for example, interrupt driven process switching occurs.

A listing of this implementation (specifically for the SAGE IV microcomputer) is given in Appendix I.

A similar effort to provide monitors in UCSD Pascal, making use of so-called UCSD "units", has been made by Boddy [Bod83] and extended in [Bod84]. In UCSD, however, the basic primitive is the semaphore as opposed to the signal, from which the condition variable is derived, in Modula-2.

4.2 Wirth's Monitor

In Modula-2 a monitor is defined by specifying a priority value, a cardinal number, in a module's heading, eg:

```
MODULE SomeMonitor[4];

IMPORT ...;
EXPORT ...;

{monitor procedures}

BEGIN
 .
 .
END SomeMonitor;
```

At this stage it will suffice to know that a priority value causes a module to become a monitor.

The monitor's task is the provision of mutual exclusion of processes to a set of procedures that act on the shared data, ie. it must ensure that only one process accesses the data at a time.

A closer look at Wirth's process facilities (see Appendix A, or alternatively [Wir83]) will reveal that a process monopolises the processor until such time that it wishes to relinquish the processor - it can never be interrupted. The process will volunteer the processor of its own volition at a process switch, and only then can another process begin/continue executing. (A process switch occurs when a) a new process is initiated, or b) a SEND or WAIT is executed). The synchronisation primitives themselves ensure mutual exclusivity.

Consequently, a process wishing to enter the monitor can do so without fear of finding it occupied. A process is only given the opportunity to execute, let alone enter a monitor, when the previously executing process effects a process switch, which by definition means it has released the processor and exclusivity on any monitor it might have held.

However, the occurrence of a machine level interrupt does have the power to interrupt an executing process. It is for this reason that the module priority is used and is so important. The execution of a monitor procedure can only be interrupted by the occurrence of an interrupt having a priority in excess of that assigned to the particular module when designating it a

monitor.
Choosing a sufficiently high module priority precludes the interruption of the execution of any monitor procedure. In this instance, the module priority ensures mutual exclusivity.

4.3 "Real" Concurrency

In a "real" concurrent environment, processes can attempt to enter the monitor whilst it is occupied. Should this be the case, Wirth's monitor facility will have to be modified.

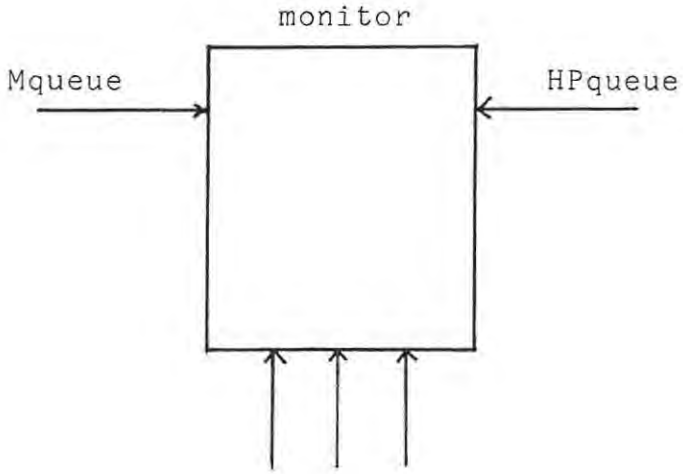
In an attempt to provide such "real" concurrency, the SAGE IV internal clock can be used to generate an interrupt driven process switch: each process can be allocated an amount of processor time, after which the clock will interrupt and the next ready-to-run process will be activated (see section 3 "Modula-2 Process Facilities" and Appendix F).

The introduction of such a facility will necessitate the addition of a built-in queue: if a process attempts to enter an already occupied monitor it must be placed on this queue.

It may also be necessary for a process to wait within the monitor until a particular condition becomes true. For this, the system must be able to deal with various user-defined "conditions" [Hoa74]. A further queue, a high priority queue, is defined, on which processes must wait after having signalling a process waiting on a condition queue. When the monitor is subsequently vacated, processes waiting on this queue are given first option to enter. (The existence of such a

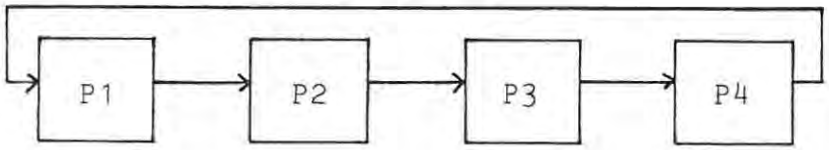
queue is justified on the grounds that a process signals waiting processes, not for reasons of necessity, but rather as a polite gesture. Should the signaller lose processor control as a result of this gesture, it is fitting that it should receive preferential treatment later on.)

Diagrammatically, the system looks as follows:



WaitCondition queues

Process ring (initiated processes)



where a) Mqueue = monitor queue -
processes waiting to enter the monitor

HPqueue = high priority queue -
processes waiting to re-enter the monitor after having signalled a

process waiting on a WaitCondition queue. A process at the head of the HPqueue is given first option to enter the monitor when it becomes vacant

WaitCondition queues = WaitCondition queue - processes waiting on various user-defined condition queues

b) In the absence of user-definable priorities, all queues must operate on a FIFO (first-in first-out) basis.

c) A process in the process ring can be in one of three states:

   i) waiting on one of the three queues mentioned above
   ii) ready-to-run but not executing
   iii) executing

Processes that have completed execution are removed from the process ring.

4.4 The Monitor

Procedures introduced:

```
PROCEDURE StartProcess(P:PROC; N:CARDINAL);
PROCEDURE StopProcess;
PROCEDURE Init(VAR S:SIGNAL);
PROCEDURE WaitCondition(VAR S:SIGNAL);
PROCEDURE SendCondition(VAR S:SIGNAL);
PROCEDURE GainEx;
PROCEDURE ReleaseEx;
```

The way in which these are employed in a concurrent program follows the skeleton program below:

```
FROM HMonitor IMPORT StartProcess, StopProcess,
                     Init, SIGNAL,
                     GainEx, ReleaseEx,
                     WaitCondition, SendCondition;

   MODULE SomeMonitor;

    IMPORT Init, SIGNAL,
           GainEx, ReleaseEx,
           WaitCondition, SendCondition;

    EXPORT A, . . ;  {monitor procedures made available}

    PROCEDURE A(Value: CARDINAL);
    BEGIN
     GainEx;
        .
        .
     ReleaseEx
    END A;

    {other monitor procedures each of the same form,
     ie. with GainEx and ReleaseEx}

   BEGIN
    {monitor initialisation code}
   END SomeMonitor;

PROCEDURE SomeProcess;
VAR Num: CARDINAL;
BEGIN
 .
 A(Num);
 .
 StopProcess
END SomeProcess;

{other procedures to be designated processes}

BEGIN
 StartProcess(SomeProcess, 400);
 .
 .
 StopProcess
END SomeModule.
```

NOTE: Although it is possible to assign a priority to Module SomeMonitor, the user is discouraged from

doing so. If a priority is assigned to this module, it must be lower than the priority assigned to the clock (the clock is assigned a priority of 3. See Appendix F). Should the priority be higher, interrupts which would otherwise have precipitated a process switch, will be lost and the degree of concurrency reduced.

A solution to the traditional Producer-Consumer problem making use of this monitor construct will now be presented.

```
MODULE ThatsLife;

FROM HMonitor IMPORT StartProcess, StopProcess,
                     Init, SIGNAL,
                     GainEx, ReleaseEx,
                     WaitCondition, SendCondition;

    (* MONITOR *) MODULE Warehouse;

    IMPORT Init, SIGNAL,
           GainEx, ReleaseEx,
           WaitCondition, SendCondition;

    EXPORT Store, Remove;

    VAR Canstore,Cantake: SIGNAL;
        Full: BOOLEAN;
        Buffer: CARDINAL;

    PROCEDURE Store(Article: CARDINAL);
    BEGIN
     GainEx;
       IF Full
         THEN WaitCondition(Canstore)
       END;
       Buffer:= Article;
       Full:= TRUE;
       SendCondition(Cantake);
     ReleaseEx
    END Store;
```

```
        PROCEDURE Remove(VAR Article: CARDINAL);
        BEGIN
         GainEx;
           IF NOT Full
             THEN WaitCondition(Cantake)
           END;
           Article:= Buffer;
           Full:= FALSE;
           SendCondition(Canstore);
         ReleaseEx
        END Remove;

        BEGIN
         Init(Canstore); Init(Cantake);
         Full:= FALSE
        END Warehouse (* monitor *);

    PROCEDURE Producer;
    VAR Item: CARDINAL;
        Count: CARDINAL;
    BEGIN
     FOR Count:= 1 TO 10 DO
      (* Produce item *)
      Store(Item)
     END;
     StopProcess
    END Producer;

    PROCEDURE Consumer;
    VAR Item: CARDINAL;
        Count: CARDINAL;
    BEGIN
     FOR Count:= 1 TO 10 DO
      Remove(Item)
      (* Consume item *)
     END;
     StopProcess
    END Consumer;

    BEGIN
     StartProcess(Producer, 400);
     StartProcess(Consumer, 400);
     StopProcess
    END ThatsLife.
```

Any parameterless procedure can be designated a process by a call to procedure **StartProcess**. This procedure takes two parameters:

a) the name of the procedure to be designated a process

b) a workspace area of size N words for allocation of local variables.

A process is terminated by a call to procedure **StopProcess**. The call should be made as the last executable statement in a process. Since the main program is also considered a process, it should execute as its last statement a call to procedure StopProcess.

All signals must be initialised by a call to procedure **Init**. It takes one parameter, the signal (S).

Condition queues are handled by procedures SendCondition and WaitCondition:

A call to procedure **SendCondition** will reactivate a process waiting for S. It takes one parameter, the signal (S).

A call to procedure **WaitCondition** will cause the waiting process to wait for some other process to SendCondition(S). It takes one parameter, the signal (S).

All monitor procedures that are to be exported for use must execute, as their first statement, a call to procedure **GainEx** and, as their last statement, a call to procedure **ReleaseEx**. These calls will have the effect of gaining and releasing monitor exclusivity.

Calls to the following procedures may only be made from procedures within the monitor:

a) GainEx
b) ReleaseEx
c) SendCondition
d) WaitCondition

Calls to these procedures outside the monitor will have unpredictable results. (Unfortunately, it is not possible to restrict these procedures to monitor procedures. They must be IMPORTed to the main program and then IMPORTed to the monitor module. This naturally makes them available to the entire concurrent program. This is a language restriction.)

Monitor procedures may call each other. Should this occur, the process will not attempt to gain, or, more dangerously, later release, monitor exclusivity a second time.

4.5 Underlying Mechanics

The monitor facility has been developed as part of a library module (MODULE HMonitor) which also provides for process initiation, termination, synchronisation and interrupt driven process switching.

The discussion will be limited to process synchronisation and monitor implementation.

Data structure modification:

The definition of ProcessDescriptor (see section entitled "Modula-2 Process Facilities") needs to be modified, by the

addition of another field, if monitor procedures are to be allowed to call each other:

```
TYPE   SIGNAL            = POINTER TO ProcessDescriptor;
       ProcessDescriptor = RECORD
                             next: SIGNAL;
                             queue: SIGNAL;
                             cor: PROCESS;
                             excl: INTEGER;
                             Ready: BOOLEAN
                           END;
```

A process which has already gained monitor exclusivity will have an excl value of one or more. Any further calls to procedure GainEx, (ie. a monitor procedure calling another monitor procedure) will merely increment the value of excl by one - the process will not seek monitor exclusivity again.

A process will only release monitor exclusivity when a call is made to procedure ReleaseEx with an excl value of one, otherwise the value of excl is decremented by one (ie. a monitor procedure has finished executing some other monitor procedure).

The semantics of some procedures:

GainEx:

```
 if the monitor is vacant
   then allow the process to enter the monitor
   else place the process on Mqueue;
        search the process ring for a ready-to-run process;
        if there is a ready-to-run process
          then transfer control to that process
```

```
            else Deadlock has occurred;
                 HALT

ReleaseEx:
 vacate the monitor;
 if HPqueue is not empty
   then allow  the top process on HPqueue to enter the monitor
   else if Mqueue is not empty
          then allow  the  top  process on Mqueue to enter the
               monitor
          else continue execution

WaitCondition(S):
 place the process on the relevant WaitCondition queue;
 if HPqueue is not empty
   then allow the top process on HPqueue to enter  the  monitor
   else if Mqueue is not empty
          then allow  the  top process on Mqueue to  enter  the
               monitor
          else search  the  process  ring  for  a  ready-to-run
               process;
               if there is a ready-to-run process
                 then transfer control to that process
                 else Deadlock has occurred;
                      HALT

SendCondition(S):
 if there are no processes waiting on condition S
   then allow the signalling process to continue execution
```

else place the signalling process on HPqueue;

transfer control to the top process on the S condition queue

The reader should note that this policy is but one of several that can be employed, for example:

a) transfer control from the signalling to the signalled process.

The signaller totally relinquishes control.

b) transfer control from the signalling to the signalled process and place the signaller on a high priority queue to be activated the moment the monitor becomes vacant.

Control is temporarily relinquished but recalled at the earliest convenience. (As used in this implementation of the monitor).

c) note that the signalled process is now ready-to-run (place the process in a pool of ready-to-run processes) but allow the signaller to maintain control. When the monitor subsequently becomes vacant, control is transferred either:

i) to the process that was previously denoted ready- to-run, or

ii) to a process chosen at random from the pool of ready-to-run processes

The signaller maintains control for as long as possible. There is, however, a chance that the

conditions that gave rise to a call to SendCondition are no longer true by the time the signalled process gains control.

The reader should note that the procedures SEND and WAIT, as defined in Wirth's module Processes (See Appendix A, or alternatively [Wir83]), are not made visible to a user. They are used to define the procedures GainEx, ReleaseEx, SendCondition and WaitCondition, which are then made visible to the user.

4.6 Limitations on the Current Implementation

i) Only one monitor module can be declared. Programs of a tutorial nature will not find this restriction a problem, but programs of a more complex nature will suffer from this imposition. Judicious introduction of parameters to the existing library procedures will, however, solve this problem.

   A similar extension in UCSD Pascal can be found in [Bod84].

ii) The success of the entire system hinges on the fact that the user remembers to start each monitor procedure with a call to procedure GainEx and to end each with a call to procedure ReleaseEx. The system does not ensure that this has been done and unpredictable results ensue if the user forgets.

4.7 Conclusion

The above implementation provides the user with a simple means of structuring a monitor.

Essentially the only difference between a module and a module to be designated a monitor is the addition of calls to procedures GainEx and ReleaseEx at the start and end of any procedure to be exported from the monitor.

The Modula-2 library facility has been found to be a most useful device in the implementation of the monitor.

MODULA-2
AND THE
ADA TASK

5. Modula-2 and the ADA Task

5.1 Introduction

Many languages developed in recent years have included some form of concurrent programming facility. Modula-2 is such a language but with an important difference: concurrency as such is not supplied as part of the language. What is supplied is an implementation of the coroutine construction, in terms of which quasi-concurrency may be implemented at the user's fancy using routines in a "module" which then becomes part of a library. This affords the user the flexibility of introducing any form of concurrent facility that might appeal to him.

In this section we examine the possibility of using Modula-2 to provide the concurrent facilities like those of Ada [Bar82], [Uni81], [Weg80], [You82], and [You83].

Apart from the challenge of the exercise, this provides useful insight into both the flexibility of Modula-2 and the deeper implications of the Ada constructions. This section describes the implementation and its use, and then tries to assess the ease and success of the implementation, and the insight gained by creating it.

(The full implementation is found in MODULE AdaTasks, a listing of which is included as Appendix J.)

5.2 Tasks

A sequential program is a program where statements are executed one at a time, one after the other. In many situations, however, a program may be subdivided into a number of smaller activities each executing independently of the other - a concurrent program. Various means have been found to represent these independent activities.

In Ada, a concurrent activity is known as a task. A program may consist of several of these independently executing tasks.

Ada example:

```
task Producer;
task body Producer is
begin
 .
 .
end Producer;

task Consumer;
task body Consumer is
begin
 .
 .
end Consumer;
```

in this instance, there are two tasks, Producer and Consumer, each executing independently of the other. (Since many current computing systems have a single processor, these tasks may not execute truly in parallel but rather have the processor's time shared amongst them)

Modula-2 implementation

Procedures introduced:

```
PROCEDURE StartTask(P:PROC; N:CARDINAL);
PROCEDURE EndTask;
```

In Modula-2 the basic facility provided for implementing quasi-concurrency is the coroutine. The closest syntax one can achieve to that in the Ada example above seems to be:

```
MODULE ThatsLife;

FROM AdaTasks IMPORT StartTask, EndTask;

PROCEDURE Producer;
BEGIN
 .
 .
 EndTask
END Producer;

PROCEDURE Consumer;
BEGIN
 .
 .
 EndTask
END Consumer;

BEGIN
 StartTask(Producer, 400);
 StartTask(Consumer, 400);
 EndTask
END ThatsLife.
```

Any procedure can be designated a task by calling procedure **StartTask**, which takes two parameters:

a) the name of the procedure to be designated a task

b) a workspace area of size N words (400 in the example above) for the allocation of variables.

As soon as the call to procedure StartTask is completed the newly created task can begin contending for processor time.

Each task must execute as its last statement a call to procedure **EndTask**. This informs the system that the task has completed execution. The main program is also designated a task

(this is done by the initialisation code of the module AdaTasks), necessitating the call to procedure EndTask found at the end of the main program.

Underlying mechanics

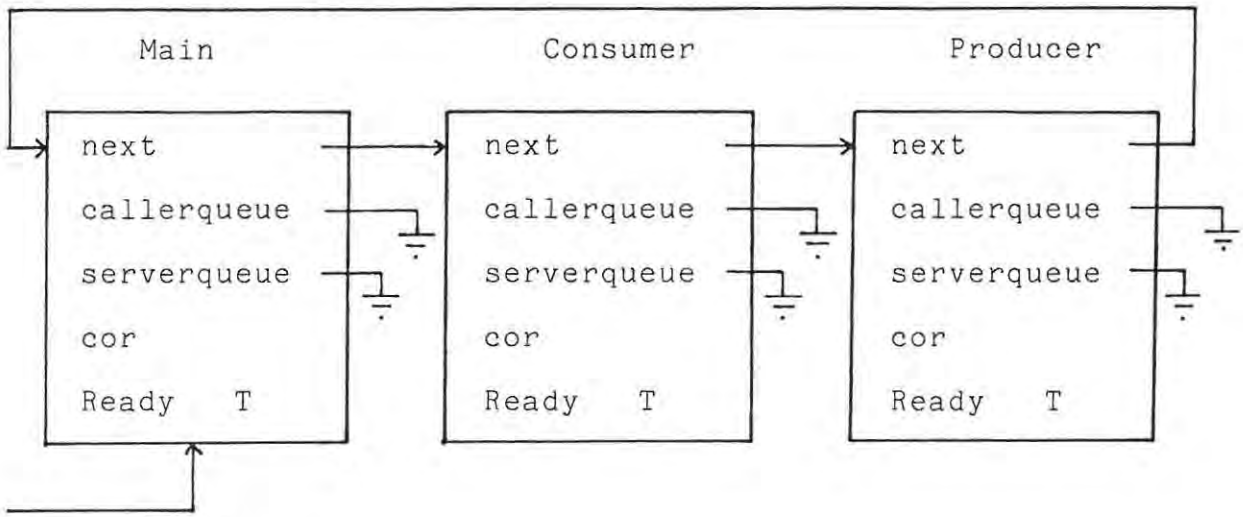
The Ada task has been implemented as a coroutine. A call to procedure StartTask will initiate a task by:

a) designating the relevant procedure as a coroutine
b) allocating a workspace area
c) allocating a TaskDescriptor to represent the task, and inserting it into the task ring.

Each task, including the main program, will be represented by a TaskDescriptor in the task ring.

```
TYPE Task = POINTER TO TaskDescriptor;
     TaskDescriptor = RECORD
                        next: Task;
                        callerqueue: Task;
                        serverqueue: Task;
                        cor: PROCESS;
                        Ready: BOOLEAN
                      END;

VAR CT: Task;
```

After tasks Producer and Consumer have been initiated, the ring will look as follows:

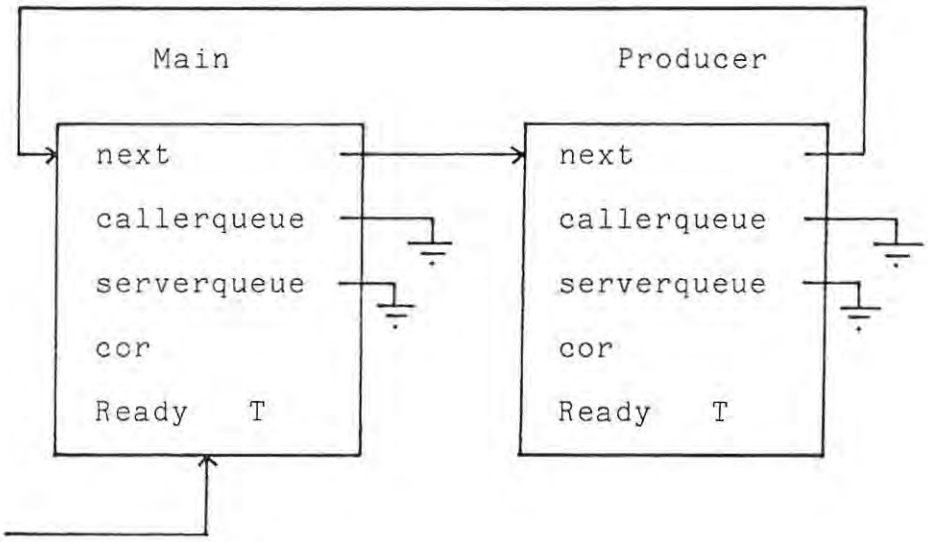


- next points to the next TaskDescriptor in the ring
- callerqueue and serverqueue will be used for rendezvous implementation (see subsection 5.3 "The Ada Rendezvous")
- cor is a system coroutine identifier
- Ready will indicate whether the task is active or can be activated (ready-to-run). It is mainly used to indicate whether a task is waiting for a rendezvous or is engaged in a rendezvous.
- CT points to the task that is currently executing, ie. receiving processor attention

The initialisation code of module AdaTasks will set up the main program as a task.

All tasks, including the main program, must execute as their last statement a call to procedure EndTask. This will have the effect of removing the relevant TaskDescriptor from the task

ring. The task will then have been "killed" and cannot be resumed unless a call is again made to procedure StartTask with appropriate parameters. Should the Consumer task complete execution, the task ring would look as follows:



After the TaskDescriptor has been removed, the next ready-to-run task in the task ring is activated.

The program as a whole is said to have completed execution once the last TaskDescriptor has been removed from the task ring. (When the rendezvous is introduced deadlock will also cause the program to halt.)

This is very similar to ideas found in Wirth [Wir83] and extended by the author (see section 3 "Modula-2 Process Facilities"). However, no mention has yet been made of any scheduling policy.

The scheduling policy relates to how the processor's time is

distributed amongst the various tasks.

The policy is quite simple: each task is given an equal amount of time and having used up its time, the next ready-to-run task is activated. In other words, the processor distributes its attention in a counter-clockwise manner around the ring. (This regular strategy will, however, be upset by the introduction of rendezvous.)

Most Modula-2 implementations provide for handling real-time interrupts. For example, a hardware clock may be set to interrupt periodically, after which the next ready-to-run task is selected and then activated. (For a full discussion regarding a particular clock implementation see Appendix F)

5.3 The Ada Rendezvous

In all but a very few circumstances, individual tasks in a concurrent program will wish to interact and co-operate with one another. One task may wish to transfer data to another task; it may be necessary for a task to wait for some action to occur before proceeding.

The Ada rendezvous [Uni81] provides a means by which tasks can do just this. As will be shown later, it serves as a combined solution to the problems of synchronisation and mutual exclusion.

The rendezvous defies terse description and may be adequately explained by way of an example.

Ada example:

```
task Producer;
task body Producer is
    C: CHARACTER;
begin
 loop
  Produce(C);
  Warehouse.PUT(C);
 end loop
end Producer;

task Consumer;
task body Consumer is
    C: CHARACTER;
begin
 loop
  Warehouse.GET(C);
  Consume(C);
 end loop
end Consumer;

task Warehouse;
    entry PUT(C: in CHARACTER);
    entry GET(C: out CHARACTER);
end Warehouse;

task body Warehouse is
    BUF: CHARACTER;
begin
 loop
  accept PUT(C: CHARACTER) do
    BUF:=C;
  end PUT;
  accept GET(C: CHARACTER) do
    C:=BUF;
  end GET;
 end loop
end Warehouse;
```

where i) Producer and Consumer are known as "callers"
ii) Warehouse is known as a "server"
iii) PUT and GET are "services" offered by Warehouse, known as "entries".

Instead of two tasks exchanging information by reading and writing to shared memory areas, they perform a rendezvous. One task, known as the caller (Producer or Consumer), will call an entry defined by another task, known as the server (Warehouse).

The server need not respond to the call instantly (in which case the caller must wait), but will eventually acknowledge the call by executing an accept clause. Likewise, the server can offer a service (by reaching an entry (PUT and GET)) but no callers need necessarily be requesting that service, in which case the server must wait until a caller does so. The accept statement defines what actions must be taken as a result of the call. As soon as the server starts executing the corresponding accept clause, the caller and server tasks are synchronised together. The rendezvous has begun. Whilst the accept clause is being executed, the caller just waits. Data can be transferred from the caller to the server, and vice versa, via parameters included in the entry call. As soon as the accept clause has completed execution, the rendezvous is broken and both the caller and server tasks continue execution independently.

Since the caller and server tasks are locked together before data is transferred, the tasks are synchronised. Since a server can only rendezvous with one caller at a time, mutual exclusion is guaranteed.

Modula-2 implementation

Procedures introduced:

```
PROCEDURE Entry(VAR R:Rendezvous);
PROCEDURE Call(VAR R:Rendezvous);
PROCEDURE Accept(VAR R:Rendezvous);
PROCEDURE EndAccept(VAR R:Rendezvous);
```

The closest syntax one can achieve to that in the Ada example

above seems to be:

```
MODULE ThatsLife;

FROM AdaTasks IMPORT StartTask, EndTask,
                     Rendezvous,
                     Entry, Call, Accept, EndAccept;
FROM SomeWhere IMPORT Produce,Consume,PlaceInStore,
                      RemoveFromStore;

VAR PUT,GET: Rendezvous;

PROCEDURE Producer;
BEGIN
 LOOP
  Produce;
  Call(PUT)
 END
 (* no call to EndTask because of loop *)
END Producer;

PROCEDURE Consumer;
BEGIN
 LOOP
  Call(GET);
  Consume
 END
 (* no call to EndTask because of loop *)
END Consumer;

PROCEDURE Warehouse;
BEGIN
 LOOP
  Accept(PUT);
    PlaceInStore;
  EndAccept(PUT);
  Accept(GET);
    RemoveFromStore;
  EndAccept(GET)
 END
 (* no call to EndTask because of loop *)
END Warehouse;

BEGIN
 Entry(PUT); Entry(GET);
 StartTask(Warehouse, 400);
 StartTask(Producer, 400);
 StartTask(Consumer, 400);
 EndTask
END ThatsLife.
```

Each entry (PUT and GET) must be declared to be of type

Rendezvous (the variable is sometimes referred to as a rendezvous variable in this report).

The reader will note that the Modula-2 implementation does not offer the facility to transfer data in the form of parameters during a rendezvous. Data cannot be transferred in the form of parameters of PUT and GET (since these are not "procedures").

Before a rendezvous variable can be used it must be initialised by making a call to procedure **Entry**, which takes one parameter, the rendezvous variable

A call to procedure **Call** will request a rendezvous with an entry. Call takes one parameter, a rendezvous variable indicating with which entry a rendezvous is requested

A call to procedure **Accept** indicates an offer of a particular service - the provision of a rendezvous. Accept takes one parameter, a rendezvous variable indicating the entry which will provide the rendezvous.

A rendezvous is terminated by a call to procedure **EndAccept**. The rendezvous will then be broken and both the caller and the server can continue to execute independently. EndAccept takes one parameter, a rendezvous variable indicating the entry which provided the rendezvous. Obviously it must match the parameter in the call to procedure Accept.

Thus, there are three components to setting up a rendezvous:

a) rendezvous variable initialisation (procedure Entry)

b) caller's request (procedure Call)

c) server's entry provision (procedures Accept and EndAccept)

The accept clause contains those statements which will be executed during the rendezvous. These statements are located between the calls to procedures Accept and EndAccept but may not include further calls to procedure Accept.

eg:

```
Accept(PUT);
  .
  .
  Accept(CHECK);
    .
    .
  EndAccept(CHECK);
  .
  .
EndAccept(PUT);
```

Nesting of calls to procedure Accept is not permitted (this is a Modula-2 implementation restriction, not an Ada restriction).

A server task may offer several entry points for the same entry.

eg:

```
Accept(PUT);
  .
  .
EndAccept(PUT);
.
.
Accept(PUT);
  .
  .
EndAccept(PUT);
```

There is no limit to the number of callers requesting the same service, but only one server can offer that service - a many-

to-one relationship.

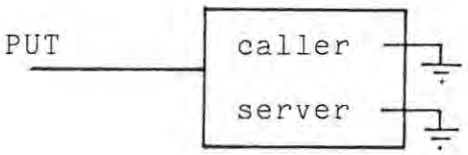
Underlying mechanics

The TaskDescriptor concept does not require any changes - they are still placed in a ring, and basic scheduling is still done by clock interrupts.

All rendezvous variables have the following components,

```
TYPE Rendezvous = POINTER TO RendezvousDescriptor;
     RendezvousDescriptor = RECORD
                              caller: Task;
                              server: Task
                            END;
```

and are initialised by a call to procedure Entry. For example, Entry(PUT) sets up:



Each time a caller task requests a rendezvous (Call(PUT)) the following occurs:

a) Its TaskDescriptor is placed at the end of the queue pointed to by the caller. The resultant queue of TaskDescriptors is linked using the callerqueue field of the TaskDescriptor.
b) The ready field of the TaskDescriptor is set to false.
c) If the queue pointed to by server is empty, ie. no task is offering such a service, control is transferred to the next ready task in the ring.

If the queue pointed to by server is not empty, ie. a task is offering such a service, but the ready field of the TaskDescriptor pointed to by server is true, ie. the server is already engaged in a rendezvous with a caller, control is transferred to the next ready task in the ring.
If the queue pointed to by server is not empty, ie. a task is offering such a service, and the ready field of the TaskDescriptor pointed to by the server is false, ie. the server is not engaged in a rendezvous, the following occurs:

i) the ready field of the TaskDescriptor pointed to by the server is set to true
ii) control is transferred to the task pointed to by server, ie. a rendezvous is launched.

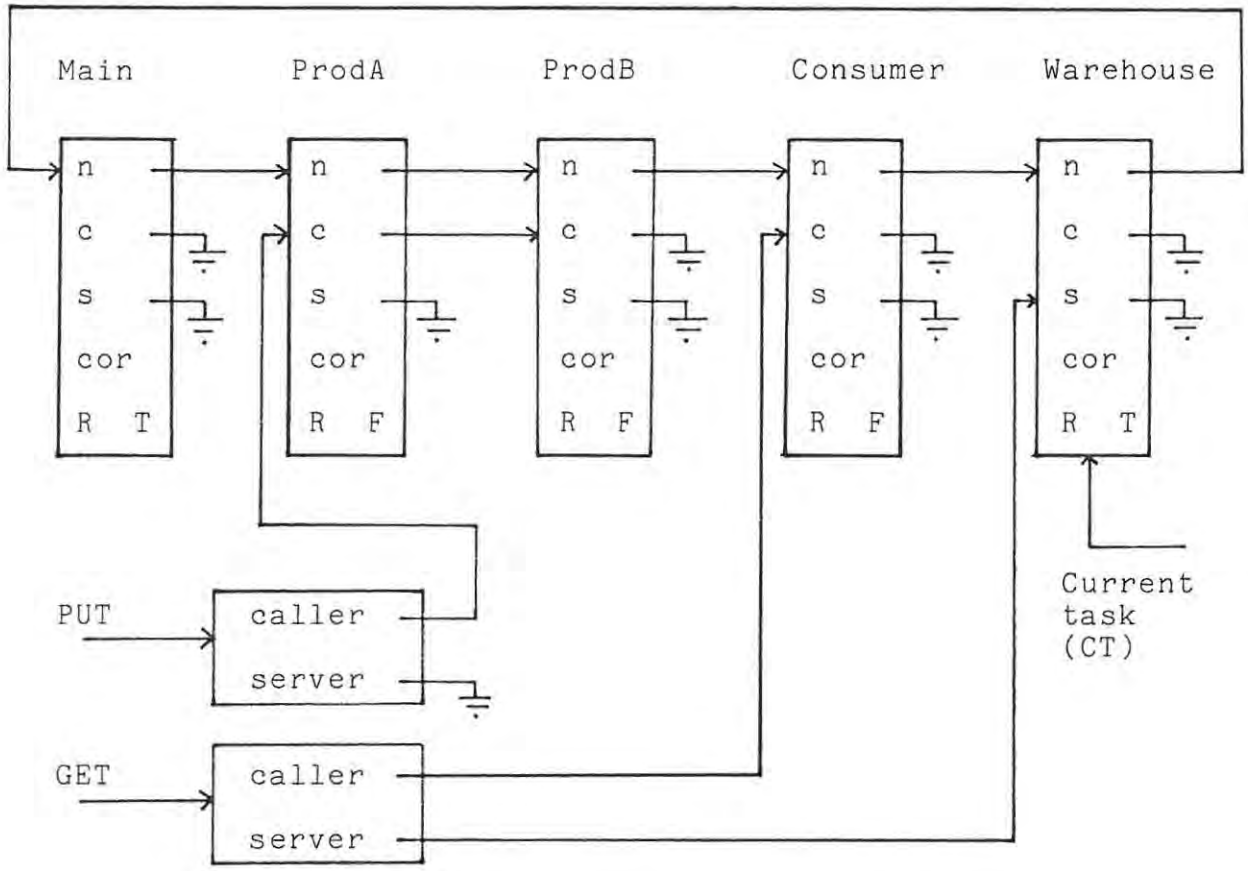
Each time a server task offers a service the following occurs:

a) Its TaskDescriptor is linked to the server queue
b) If the queue pointed to by caller is empty, ie. no task is requesting such a service, the ready field of the server's TaskDescriptor is set to false, and control is transferred to the next ready task in the ring.
If the queue pointed to by caller is not empty, ie. there is a task requesting such a service, the server task continues execution, ie. a rendezvous is launched.

Each time an attempt is made to transfer control to the next ready task in the ring, the possibility exists that no such task exists is present (perhaps all other tasks are waiting for

a rendezvous, or there are no other tasks). This situation is known as deadlock, and the program will halt if it develops.

A program consisting of two producers, a consumer and a warehouse could at some stage of its execution have the following TaskDescriptor structure:



where both ProducerA and ProducerB are waiting on entry PUT, Consumer is engaged in a rendezvous with Warehouse and Warehouse is the currently executing task.

The call to procedure EndAccept, to denote the completion of a rendezvous, has the effect of:

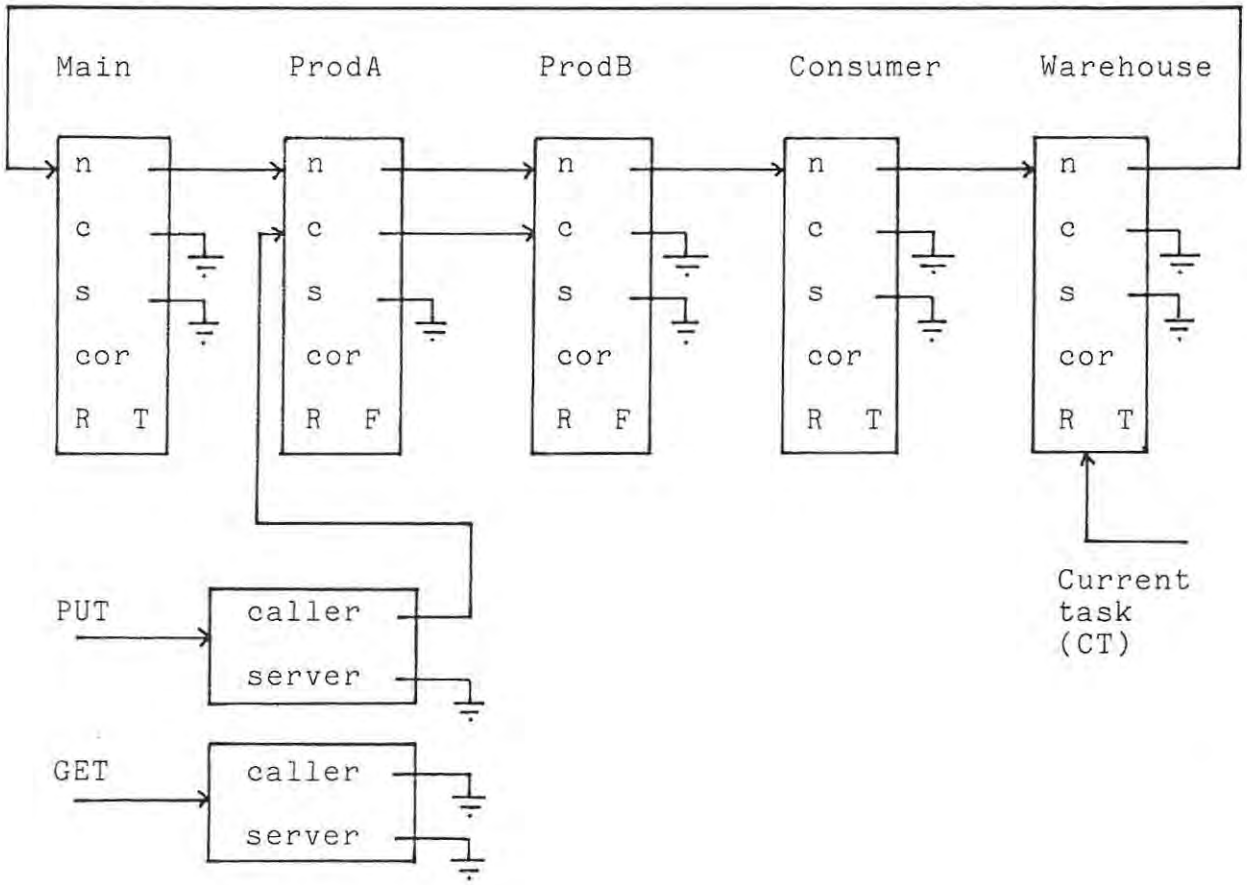
a) removing the top TaskDescriptor from the queues pointed

to by caller and server of the particular rendezvous variable

b) setting the ready field in the TaskDescriptor representing the caller task to true

c) allowing the server task to maintain processor control.

As soon as the rendezvous in the above example is complete, the TaskDescriptor structure will look as follows:



5.4 Selective Wait Rendezvous

In many examples the order in which accept clauses are executed is predetermined by some fixed order in a server task. In

fact, this may be a necessary condition for their successful operation. However, there are instances in which calls must be accepted in the order in which they occur, ie. the callers dictate the order in which particular rendezvous occur, not the server.

Consequently, a server must be capable of providing a set of possible rendezvous from which a caller can select in any order. The selective wait rendezvous is a mechanism whereby a server process can avoid executing an accept statement and thereby committting itself to waiting for a caller process to rendezvous until a caller is known to be actually waiting. The server will wait until a call is made to any one of the entries contained in the set and then (and only then) execute the appropriate accept statement.

Ada example:

```
task body ProtectedTime is
 TIMENOW: TIME;
begin
 loop
  select
   accept READ(T: out TIME) do
     T:=TIMENOW;
   end READ;
  or
   accept WRITE(T: in TIME) do
     TIMENOW:=T;
   end WRITE;
  end select;
 end loop;
end ProtectedTime;
```

When the select is encountered four possibilities exist:

a) no calls to either READ or WRITE are pending

b) a call to READ but not WRITE is pending

c) a call to WRITE but not READ is pending

d) calls to both READ and WRITE are pending

For case a), the task will wait until a call is made to either READ or WRITE.

For cases b) and c), the appropriate select alternative is chosen and executed.

For case d), one of the calls is selected (on a random basis) and the appropriate select alternative executed.

Modula-2 implementation

Procedures introduced:

```
PROCEDURE Select(VAR SR:SelectRend);
PROCEDURE SAccept(VAR SR:SelectRend; R:Rendezvous;
                  SProc: Proc);
PROCEDURE EndSelect(VAR SR:SelectRend;
                    VAR AProc:PROC);
```

The closest syntax one can achieve to that in the Ada example above seems to be:

```
FROM AdaTasks IMPORT StartTask, EndTask,
                     Rendezvous,
                     Entry, Call, Accept, EndAccept,
                     SelectRend,
                     Select, SAccept, EndSelect;
FROM SomeWhere IMPORT TIMENOW;

VAR RWTime: SelectRend;
    READ, WRITE: Rendezvous;
    SomeAction: PROC;
    T: TimeType;

PROCEDURE R;
BEGIN
  T:=TIMENOW;
  EndAccept(READ)
END R;
```

```
PROCEDURE W;
BEGIN
 TIMENOW:=T;
 EndAccept(WRITE)
END W;

PROCEDURE ProtectedTime;
BEGIN
 LOOP
  Select(RWTime);
    SAccept(RWTime, READ, R);
    SAccept(RWTime, WRITE, W);
  EndSelect(RWTime, SomeAction); SomeAction
 END
END ProtectedTime;
```

The data declarations of importance are:

RWTime - identifies the selective wait rendezvous

READ, WRITE - rendezvous variables

SomeAction - a parameterles procedure variable which will be assigned the procedure (either R or W, in this case) which will contain the accept statements that must be executed during the rendezvous.

Again, it is important for the reader to note that the Modula-2 implementation does not offer the facility to transfer data in the form of parameters during a rendezvous. Data cannot be transferred in the form of parameters of READ and WRITE (Modula-2 limitation).

Calls to procedures Select, EndSelect and SomeAction (SomeAction being a user defined procedure variable) form the framework of the selective wait rendezvous structure and are compulsory. The user may place as many calls to procedure SAccept in-between as he likes.

Procedure **Select** initialises the selective wait rendezvous.

Select takes one parameter, the variable which identifies the selective wait rendezvous

Procedure **SAccept** adds further entry calls to the set of possible rendezvous. SAccept takes three parameters:

a) the variable which identifies the selective wait rendezvous
b) the entry call (rendezvous variable)
c) a parameterless procedure name identifying a procedure whose statements will be executed during the rendezvous with the entry mentioned in b).

   Important: this procedure must execute as its last statement a call to procedure EndAccept. Its only parameter must be the rendezvous variable mentioned in b). If no action is to be taken during the rendezvous (so that it serves as a synchronising mechanism only), a procedure must still be set up with a lonely call to procedure EndAccept with the appropriate parameter.

Procedure **EndSelect** will select the entry call, from the possible set, for which a call is pending, and will note the associated procedure. EndSelect takes two parameters:

a) the variable which identifies the selective wait rendezvous
b) the user-declared parameterless procedure name which will be assigned the name of the procedure that must be executed during the pending rendezvous.

Finally, a call to the user defined parameterless procedure has the effect of executing the accept clause.

Thus, there are four components to a selective wait rendezvous:

a) rendezvous variable initialisation (procedure Entry)
b) caller's request (procedure Call)
   (even though a new procedure (procedure SAccept) offers a service, a caller still requests a rendezvous in the same way)
c) server's provision of entries (procedures Select, SAccept and EndSelect and the call to the associated procedure)
d) the various procedures containing the accept clauses to be executed during the rendezvous.

The accept clauses contained in the various procedures may not include calls to procedures Select, SAccept or EndSelect.

If the user does not include any select alternatives in the selective wait rendezvous:

```
Select(RWTime);
EndSelect(RWTime, SomeAction); SomeAction;
```

the program will halt with the fatal error message:

"Select alternatives missing"

Underlying mechanics

The basic idea is to construct a queue of all contending entry calls and then to scan the queue continually until an

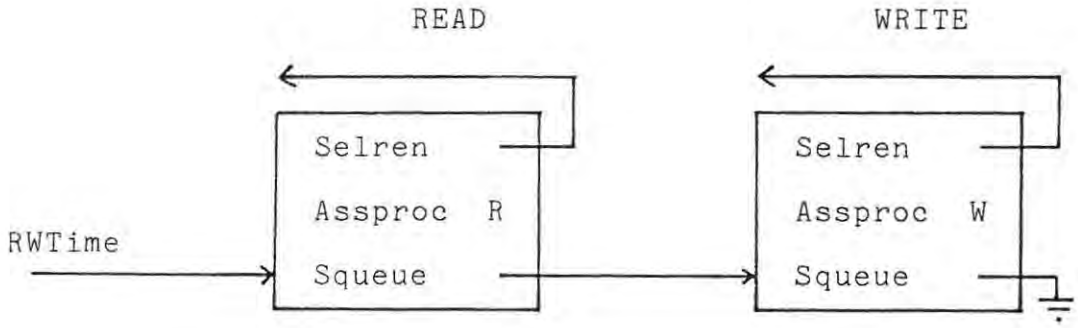
appropriate call is made, whereupon a rendezvous is set up.

The elements of the queue will take the following form:

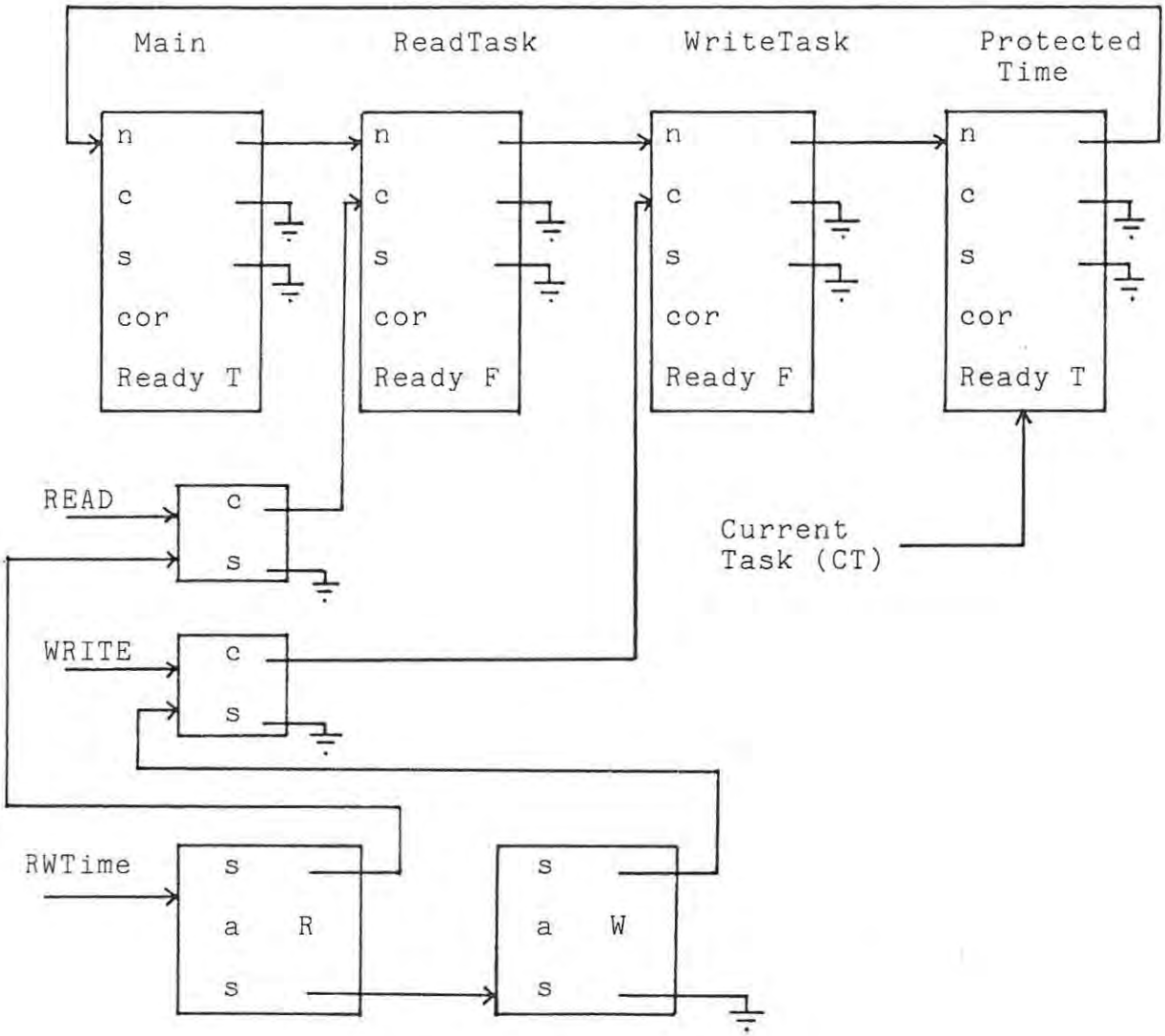
```
TYPE SelectRend = POINTER TO SelectRendDescriptor;
     SelectRendDescriptor = RECORD
                              selren: Rendezvous;
                              assproc: PROC;
                              squeue: SelectRend
                            END;
```

Procedure Select will initialise the selective wait rendezvous by setting the pointer to the queue of contending entry calls to NIL:

Procedure SAccept will add elements to the queue:



The Selren field will point to the appropriate rendezvous variable. The whole structure will look as follows:



where both ReadTask and WriteTask have made calls to READ and WRITE, respectively, and ProtectedTime has just completed constructing the selective wait rendezvous queue.

Procedure EndSelect will scan the queue examining each element in the following way:

a) follow the Selren pointer to the rendezvous variable

b) if the caller pointer is NIL

then move to the next element in the queue

```
            if the end of the queue is reached
              then start at the beginning again
            goto a) above
        else a call is pending
            note the contents of Assproc (associated
            procedure)
```

Only when a call is found to be pending will the server task be linked to the server queue of the particular rendezvous variable (as opposed to linking the server task as soon as it offers the service, as was done before).

The queue is then dismantled and each element is placed on another queue for re-use.

The procedure name, identifying the procedure whose statements will be executed during the rendezvous, is returned to the user's program via a parameter.

The reader will note that procedure EndSelect is located within a monitor within module AdaTasks. One might be tempted to think that the following code would suffice:

```
PROCEDURE EndSelect(VAR SR:SelectRend; VAR Aproc:PROC);
BEGIN
 WHILE Checking(SR, Aproc) DO
 END;
 {queue destruction}
END EndSelect;

where procedure Checking will scan the queue once and return
  true  if no calls are pending or
  false if it has found a caller requesting a service
```

However, it is very possible that no calls have been made to

any of the entries represented in the queue, in which case procedure Checking will be invoked again. Since procedure EndSelect is located within a monitor, the clock interrupt, providing the means by which a task switch is made, may be inhibited. As a result no task switching will occur and consequently no task will be able to request a rendezvous, leaving the program in a static state.

The solution is to include a call to procedure Listen (a Modula-2 system procedure) which will temporarily lower the priority of the monitor in which procedure EndSelect is located and enable the clock to interrupt the monitor and effect a task switch.

```
PROCEDURE EndSelect(VAR SR:SelectRend; VAR Aproc:PROC);
BEGIN
 WHILE Checking(SR, Aproc) DO
  LISTEN
 END;
 {queue destruction}
END EndSelect;
```

So, if there are no calls to any of the entries provided by the selective wait rendezvous when it is first scanned, the callers will subsequently have the opportunity of execution and possibly request a rendezvous.

5.5 Guards

A guarding condition is a condition which must be true before a particular service (entry) within a selective wait rendezvous can be offered.

Ada example:

```
select
  when not VEGETARIAN =>
    accept MEAT ....
      .
      .
    end MEAT;
or
    accept CARROTS ....
      .
      .
    end CARROTS;
or
  when MONEY > LOTS =>
    accept CAVIAR ....
      .
      .
    end CAVIAR;
end select;
```

Each time the selective wait rendezvous is encountered all the guarding conditions are evaluated. The procedure is then exactly the same as for an ordinary selective wait rendezvous but only those entry calls for which there was no condition or the condition was true are available for "selection". A guard condition is never re-evaluated. If a guard condition changes (the user's bank balance suddenly depletes) while waiting within a selective wait rendezvous, the set of contending entry calls is not amended (and CAVIAR stays on the menu).

Modula-2 implementation

The Modula-2 implementation has, to a large degree, skirted the issue of the Ada guard condition. It has been implemented in the form of an IF ... THEN ... ELSE ... END.

The closest syntax one can achieve to that in the Ada example above seems to be:

```
FROM AdaTasks IMPORT StartTask, EndTask,
                     Rendezvous,
                     Entry, Call, Accept, EndAccept,
                     SelectRend,
                     Select, SAccept, EndSelect;

VAR Shop: SelectRend;
    MEAT, CARROTS, CAVIAR: Rendezvous;
    BuyWhat: PROC;
    MONEY, LOTS: MoneyType;
    VEGETARIAN: BOOLEAN;

PROCEDURE Menu;
BEGIN
 LOOP
  Select(Shop);
    IF NOT VEGETARIAN
      THEN SAccept(Shop, MEAT, M)
    END;
    SAccept(Shop, CARROTS, C);
    IF MONEY > LOTS
      THEN SAccept(Shop, CAVIAR, Ca)
    END;
  EndSelect(Shop, BuyWhat); BuyWhat;
 END
END Menu;
```

with relevant procedures for M, C and Ca.

This structure nonetheless conforms with the guard rules mentioned above.

Underlying mechanics

No changes or additions to any of the existing constructs is required. The entry call is either added to the queue of contenders or it is not.

5.6 Conditional Rendezvous - ELSE

Lastly, the conditional form of the selective wait rendezvous is presented, in which provision is made for an ELSE part.

Ada example:

```
select
   accept CARROTS ....
     .
     .
   end CARROTS;
 or
   accept BEANS ....
     .
     .
   end BEANS;
 else null
end select;
```

If no calls are pending for any of the entries provided, the else part is executed instead of a rendezvous, ie. accept a call to either CARROTS or BEANS only if a call to either is pending, otherwise do nothing (null).

Modula-2 implementation

Procedure introduced:

```
PROCEDURE ElseAccept(VAR SR:SelectRend; Sproc:PROC);
```

The closest syntax one can achieve to that in the Ada example above seems to be:

```
FROM AdaTasks IMPORT StartTask, EndTask,
                     Rendezvous,
                     Entry, Call, Accept, EndAccept,
                     SelectRend,
                     Select, SAccept, EndSelect,
                     ElseAccept;

VAR Shop: SelectRend;
    CARROTS, BEANS: Rendezvous;
    BuyWhat: PROC;
```

```
PROCEDURE ShoppingList;
BEGIN
 LOOP
  Select(Shop);
    SAccept(Shop, CARROTS, BuyCarrots);
    SAccept(Shop, BEANS, BuyBeans);
    ElseAccept(Shop, BuyNothing);
  EndSelect(Shop, BuyWhat); BuyWhat;
 END
END ShoppingList;
```

with relevant procedures for BuyCarrots, BuyBeans and BuyNothing.

The call to procedure **ElseAccept** is very similar to a call to procedure SAccept, ie. both include a parameter to identify the selective wait rendezvous, and the name of a parameterless procedure to contain the accept clause. Since the ELSE option does not involve a rendezvous, there is no rendezvous variable parameter in a call to procedure ElseAccept.

The structure of the procedure to contain the accept clause is also different. Using the above example:

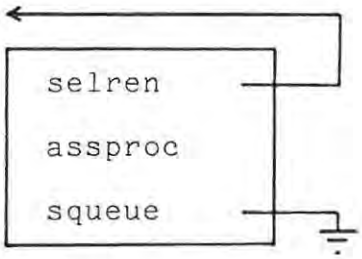
```
PROCEDURE BuyNothing;
BEGIN
 .
 .
 .
END BuyNothing;
```

The procedure does not execute as its last statement a call to procedure EndAccept. The object of a call to procedure EndAccept is to break the rendezvous, and since no rendezvous was set up, there is no reason to attempt to break it.

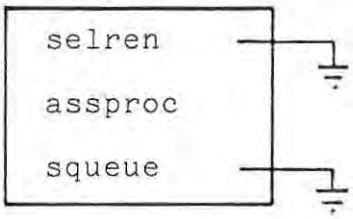
Underlying mechanics

Each element in the queue of contending entry calls has the

following components:



When an ElseAccept is encountered the same structure is used but the pointer to the rendezvous variable is set to NIL:



When the queue is subsequently scanned, a NIL rendezvous variable pointer will indicate an ELSE part. (If the ELSE part is reached it is indicative of the fact that no preceding entries have been called and that the ELSE option can be selected) Its associated procedure will be noted and the scanning process terminated in the usual selective wait rendezvous style.

When an ELSE option is selected no changes are made either to:

a) the queues pointed to by caller and server of any rendezvous variable, or
b) the ready field of any TaskDescriptor

The queue is dismantled and the elements re-used in the same way as they are in the ordinary selective wait rendezvous.

The server once again maintains processor control as soon as the ELSE part's associated procedure has completed execution.

5.7 Review

a) Data transfer

In Ada, data can be transferred from callers to servers, and vice versa, by means of parameters included in the entry call. In the Modula-2 implementation it is not possible to transfer data in the form of parameters. At present global variables provide the only means of transferring data during a rendezvous.

This limitation partly destroys the rendezvous concept and degrades it to a synchronising mechanism.

Consider the following example:

```
MODULE ThatsLife;

FROM AdaTasks IMPORT StartTask, EndTask,
                     Rendezvous,
                     Entry, Call, Accept, EndAccept;
FROM SomeWhere IMPORT Produce, Consume;

VAR A,B: SomeType;
    PUT,GET: Rendezvous;

PROCEDURE Producer;
BEGIN
 LOOP
  Produce(A);
  Call(PUT)
 END
END Producer;
```

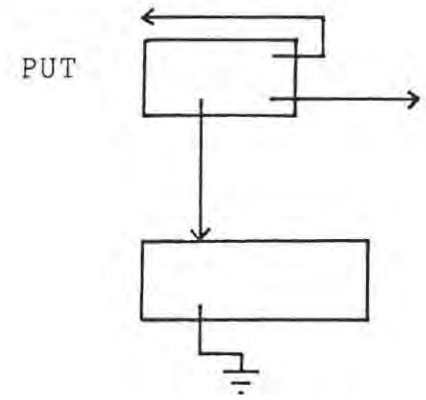
```
PROCEDURE Consumer;
BEGIN
 LOOP
  Call(GET);
  Consume(B)
 END
END Consumer;

PROCEDURE Warehouse;
VAR Buf: SomeType;
BEGIN
 LOOP
  Accept(PUT);
   Buf:=A;
  EndAccept(PUT);
  Accept(GET);
   B:=Buf;
  EndAccept(GET)
 END
END Warehouse;

BEGIN
 Entry(PUT); Entry(GET);
 StartTask(Producer, 400);
 StartTask(Consumer, 400);
 StartTask(Warehouse, 400);
 EndTask
END ThatsLife.
```

The integrity of Buf is ensured: it is locally declared and its use is strictly guarded. Unfortunately A and B are open to abuse. There is no way of transferring their contents as parameters of PUT and GET (Modula-2 limitation), if A and B are locally declared. This reduces the rendezvous mechanism to one of synchronisation and not commmunication.

One rather long-winded method of solving this problem would be on the following lines:

- add to each rendezvous variable the number of parameters it should take

```
Entry(PUT);
 Parm(PUT);
```



- prior to requesting a rendezvous, the contents of the parameters would be set up:

```
SetParm(PUT, 1, A);
Call(PUT);
```

- the server would then be able to extract the data:

```
Accept(PUT);
  Buf:=GetParm(PUT, 1);
EndAccept(PUT);
```

But even this solution has a flaw. What if two callers wish to request PUT ? Both would attempt to set the parameters, and the resultant data would be wrong.

b) In Ada, accept clauses may include entry calls.

eg:

```
Accept Put (......);
  .
  .
  Accept Check (......);
    .
    .
  end Check;
  .
  .
end Put;
```

However, in the Modula-2 implementation nested entries are not allowed. Each time an accept is encountered the task's TaskDescriptor is linked to the appropriate rendezvous variable queue. Clearly the TaskDescriptor can not be in two positions in the same queue, or in two or more queues simultaneously. The possibility of declaring multiple instances of a TaskDescriptor has been briefly looked at and not as yet totally discounted. The limitation does not appear severely to impede programming, at least not of a tutorial nature.

c) Selective wait rendezvous

i) In Ada, a select structure with no select alternatives is syntactically incorrect. In the Modula-2 implementation such a situation would look as follows:

```
Select(Store);
EndSelect(Store, What); What;
```

If execution were allowed to continue, a simple task of assigning an empty procedure (contains no executable statements) to "What," would result in nothing but extensive deadlock - numerous callers requesting non-existent services. Consequently, the implementation ensures that the program is terminated with an appropriate error message and the user can correct this obvious error.

ii) There is also the possibility that the select structure will place the task in which it is located in an

indefinite wait. This occurs when:

a) the selective wait rendezvous offers services no caller gets the opportunity of requesting
b) the selective wait rendezvous offers services no caller ever requests.

This will result in a never-ending scan of the queue to determine whether any caller has requested any of the services being offered.

Two possible solutions exist:

1) check whether the task offering the select alternatives is the only task remaining in the system, in which case the program can be terminated because there are no callers to request any services
2) check whether the task offering the select alternatives is the only task in the ready-to-run mode, in which case the program can be terminated because there is no prospect of any caller requesting any service.

The current implementation does not address itself to this problem.

The same situation can result in Ada, ie. an Ada selective wait rendezvous can be structured in such a way that it too will wait forever.

However, the Ada **delay** statement [Uni81] does provide a solution. If no task has requested any of the services presented by the selective wait rendezvous after a user-defined period of time, the delay part of the selective wait rendezvous is executed.

iii) A Modula-2 system usually offers garbage collection intrinsics (DEALLOCATE and DISPOSE). The implementation of module AdaTasks, however, provides its own garbage collection facilities for the selective wait rendezvous. The method employed is to maintain a secondary structure into which re-usable elements can be linked and later removed.

iv) The reader will remember that as soon as one of the select alternatives is chosen the queue is dismantled and stored for later use. In many instances, however, the selective wait rendezvous is placed within a loop:

eg:

```
loop
  select

    or

    or

  end select
end
```

Is there any motivation to retain the queue if the selective wait rendezvous is placed within a loop?

If the selective wait rendezvous does not contain any guarding conditions, in which case the queue will

always be the same, then there is reason to retain the queue. But, if there are any guarding conditions, in which case the queue could be different each time it is constructed, there is little reason to retain the queue.

Retaining the queue, if the selective wait rendezvous is placed within a loop, would save unnecessary reconstruction. Unfortunately there is no way of determining whether the selective wait rendezvous is placed within a loop other than by stating this explicitly:

eg:
add a second parameter to procedure Select to indicate:

a) that the selective wait rendezvous is enclosed by a loop and that the queue structure should be maintained, or
b) that the queue structure can be dismantled as soon as a select alternative has been chosen.

This would, however, seem to place an unnecessary burden on the user - having to determine whether the selective wait rendezvous is in a loop and, if so, whether the queue will be the same each time it is constructed - and would just increase the scope for potential error.

Consequently, the situation is best left alone. In any case, the queue elements are used again, leaving the only disadvantage an increase in execution time resulting from queue construction.

5.8 Assessment

The Modula-2 implementation does not include all Ada concurrent features (the problems of data transfer and nested entries have already been mentioned). What is provided is a workable subset with which the user can come to grips and come to appreciate the basic concepts of concurrency in Ada.

The structures are simple and match the Ada equivalents fairly closely:

	Ada	Modula-2
initialise:	`Entry PUT ...`	`Entry(PUT)`
Service request:	`PUT ...`	`Call(PUT)`

Entry:	`accept PUT ...`
	`.`
	`.`
	`end PUT;`
		`Accept(PUT);`
		`.`
		`.`
		`EndAccept(PUT);`

selective wait rendezvous:	`select`
	`  accept PUT ...`
	`  .`
	`  .`
	`  end PUT;`
	`or`
	`  when T = 50 =>`
	`  accept Store ...`
	`  .`
	`  .`
	`  end Store;`
	`else Ship`
	`end select;`
		`Select(Wareh);`
		`  SAccept(Wareh, Put, P);`
		`  IF T = 50`
		`  THEN`
		`  SAccept(Wareh, Store, S)`
		`  END;`
		`  ElseAccept(Wareh, Ship);`
		`EndSelect(Wareh, Action);`
		`Action;`
		
		`(and procedures for P,S and Ship)`

The necessity of providing separate procedures for the selective wait rendezvous does cause slight inconvenience. The clarity of the structure has, however, not been effected.

The exercise was conducted in a modular way, starting with the implementation of the task first and then including the various means of launching a rendezvous afterwards.

The underlying structures, representing the tasks and rendezvous, consequently expanded as each new feature was added - it was not planned in its entirety and then implemented. This method proved to be most successful.

The ease with which the features can be used is a result of both the Modula-2 system and the implementation itself. The module AdaTasks can be imported into a user program quite simply (using the IMPORT command) making the facilities immediately available for use. Furthermore, the procedure names match the Ada equivalents (eg: select) or actions (eg: Call) fairly closely, making their use relatively obvious.

5.9 Conclusion

Although the implementation does not include all the features offered by Ada, it does provide the user with sufficient to be able to gain a working knowledge of the mechanisms involved. In an environment lacking an Ada implementation, the Modula-2 version proves to be a useful substitute.

CRITICAL ASSESSMENT

6. Critical Assessment

6.1 Introduction

This section takes a closer look at those particular Modula-2 language constructs and facilities provided for the implementation of concurrency.

A brief note concerning concurrency in Modula, Modula-2's direct ancestor, has also been included.

The reader is directed to the following references for general critical assessments of Modula-2: [Cai82], [Cla83], [Gle83], [Spe82] and [Sum82].

6.2 The Coroutine

The low level coroutine construct has been used to implement the more sophisticated process.

A call to the standard procedure NEWPROCESS designates a particular procedure as a process. NEWPROCESS takes four parameters:

a) the procedure name
b) start address of a workspace area
c) size of the workspace area
d) a process variable (coroutine identifier).

Two important considerations stem from this procedure call:

a) The nature of the procedure to be designated a process

The procedure may not have any parameters. The available literature on Modula-2 simply states that this is the case, and does not extend any justification for restricting processes to parameterless procedures.

The method by which processes are designated poses an immediate problem:

```
NEWPROCESS(Consumer, ADR(Workspace), SIZE(Workspace), CorId)
```

The procedure to be designated a process is a parameter itself and if it were now merely given parameters a syntactic problem would result. Since NEWPROCESS is a general-purpose routine, it would be unacceptable to allow the parameters of the procedure to be parameters of NEWPROCESS.

This problem is not unresolvable. The current standard for Pascal [Add80] allows procedures with parameters to be passed as parameters themselves.

Such a parameter restriction imposes an unfortunate limitation on the resultant concurrent program. There is every possibility that a number of processes will be identical in structure, eg: numerous user jobs each invoking a printer process or, on a more tutorial scale, five producers and three consumers. Some method of transmitting

information to a process or distinguishing one process from the other, by means of parameters, is highly desirable. At present numerous instances of a particular process can be declared, but there is no way of telling them apart, and "parameters" have to be "passed" as global variables.

Many concurrent languages allow procedures with parameters to be designated as processes:

Clang -

```
Procedure Terminal(Channel: Integer);

cobegin
  Terminal(1);
  Terminal(2)
coend
```

Concurrent Euclid -

Concurrent Euclid appears to have reserved what would otherwise have been a formal parameter declaration area for the specification of process stack sizes. The syntax for a process module is as follows:

```
{ process id [(MemoryRequirements)]
      procedureBody }
```

Modula -

```
Process Terminal(Channel: Integer);

Terminal(1);
Terminal(2);
```

Pascal-Plus -

```
Process Terminal(Channel: Integer);

Instance T1: Terminal(1);
         T2: Terminal(2);
```

UCSD Pascal IV.0 -

```
Process Terminal(Channel: Integer);

START(Terminal(1));
START(Terminal(2));
```

It is interesting to note how Pascal (earlier and later versions) and Modula-2, both products of Wirth, have dealt with the issue of procedural and functional parameters.

The early versions of Pascal [Jen74], [Gro80] allow procedural and functional parameters with the option of value parameters only. However, no mention is made of these value parameters in the main procedure's formal parameter list.

eg.

```
Procedure Tom(Procedure A; var I,J:integer);
```

No attempt has been made to check what kind of parameters procedure A might carry, and together with the following declarations:

```
Procedure Dick(I:integer);
Procedure Harry(X, Y:real);
```

one can attempt the following calls:

```
Tom(Dick, ActualParm1, ActualParm2);
Tom(Harry, ActualParm1, ActualParm2);
```

Many microcomputer implementations, including JRT Pascal [Jrt83], Turbo Pascal [Bor83] and UCSD Pascal [Koe82] do not allow procedural or functional parameters, even though the Pascal standard permits them.

Later versions of Pascal, including [Add80] and Pascal/MT+ [Mtm81] allow procedural and functional parameters with the option of value, and it would seem, variable parameters (the references are unclear in this regard).

Modula-2 does not allow procedural or functional parameters.

Since procedural and functional parameters are such useful and obvious constructs to have in a serious language, it is surprising that Modula-2 does not support such a facility. The fact that Pascal and Modula-2 have common authors makes the issue even more perplexing.

b) The concept of a workspace area

Each process requires a stack area for procedure call information, and for the allocation of local variables belonging to the process procedure (and any other procedures called by it). The size of this stack area must be provided by the user in the call to NEWPROCESS.

This method, when compared with an alternative such as static stack size, has both advantages and disadvantages:

i) User provided

Advantages

a) A process may declare as many local variables and make as many procedure calls as it wishes by merely increasing or decreasing the stack area. (This will obviously be restricted by the memory of the machine

as a whole.)

b) The user can make use of the absolute minimum stack area by successively reducing the stack area until an error occurs.

Disadvantages

a) Providing a meaningful value requires an inner knowledge of how the stack space will be utilised. The task becomes progressively more difficult when a process makes multiple procedure calls, or calls a recursive procedure.

   This requisite knowledge of low level implementation details seems to be at odds with the intent of high level languages - that of steering the user as far away from low level details as possible. Few programmers know much about such details and even fewer wish to acquire this knowledge.

b) In most cases the user will provide an excessive amount of stack space, just to be on the safe side. This leads to an unnecessary waste of space.

The situation is further complicated by the fact that in some implementations a stack overflow error message is not guaranteed if insufficient stack space is allocated, in some ways justifying the action taken in Disadvantages b). In fact, it is more likely to crash the system.

Volition implementation of Modula-2 on the SAGE IV -

message not guaranteed

University of Hamburg implementation of Modula-2 on the VAX-11 -

message guaranteed ("Process Workspace Overflow")

ii) Static stack-size

Advantage

a) The user is freed of a task which should not have been any of his concern in the first place.

Disadvantage

a) There will always be a process that will require more stack space than has been allocated by the system. Employing the previous method, the user will, at least, have the opportunity to attempt to increase the stack space.

A typical trade-off situation has thus developed:

- execute processes requiring diverse stack space
- the freedom of not having to bother about stack space.

Different approaches to the stack space problem have been provided by various languages:

Clang -

all available stack space is divided amongst active processes. Unfortunately a major difference between Clang and Modula-2 is that in Clang all processes to

be executed concurrently are known beforehand (static activation) as opposed to Modula-2's dynamic activation.

Concurrent Euclid -

a default stack size is allocated if none is specified in the process heading.

eg. Process Terminal (400)

where the default of, say 2000 bytes in a typical implementation, is inappropriate. (Note that processes do not carry parameters. The stack size specification is found where formal parameters would otherwise be specified.)

Pascal-Plus -

the compiler determines the upper bound on the stack space required for execution of each process.

UCSD Pascal IV.0 -

the procedure START, used to initialise a process, has an optional parameter to set the stack size at something other than the default value.

In general, however, stack space allocation is a burden the user could do best without.

6.3 TRANSFER and IOTRANSFER commands

The method of transferring processor control from one coroutine (process) to the other is by means of the TRANSFER and IOTRANSFER commands.

a) TRANSFER

Control is transferred from process A to process B by the following command:

```
TRANSFER(A, B)
```

If control must be returned to process A, it must be done explicitly with the following command:

TRANSFER(B, A)

where A and B are so-called process variables.

b) IOTRANSFER

Suppose control must be transferred from process A to process B, but upon occurrence of an interrupt, control must be returned to process A. Since the interrupt is by definition an unscheduled event, the TRANSFER command will not do (there is no way of predicting when the interrupt will occur to know where to place the TRANSFER statement).

The IOTRANSFER command will transfer control from process A to process B, but upon occurrence of an interrupt via InterruptVector, control is returned to process A:

IOTRANSFER(A, B, InterruptVector)

Both of these commands are clear, concise and easy to use. (The IOTRANSFER command may take a little longer to master because of its close relationship to the machine on which the language has been implemented.)

It is important to remember that the parameters, A and B, of both of these commands must be initialised process variables. Process variables are initialised by the original call to procedure NEWPROCESS. The system will crash if an uninitialised process variable is used. The reader should note that this is

the second instance where the system is likely to crash when dealing with coroutines and processes (the first concerns stack space allocation when designating processes). It is unfortunate that these two important and closely related procedure calls can precipitate such consequences.

6.4 The Library

The Modula-2 library forms a crucial part of the language and its implementation. Many features previously considered intrinsic to a language have been relegated to membership of a library module, eg.

- console I/O
- format conversion between text and binary representation
- file I/O operations, eg. read, write and seek
- file directory operations, eg. create, delete and rename
- storage management, eg. new and dispose
- nested program execution
- process scheduling
- mathematical functions
- dynamic string manipulation.

A library module is divided into two parts:

a) Definition module

The definition module contains the declarations of the objects the library module will export to other modules.

b) Implementation module

The implementation module contains the code implementing the library module.

The definition module is effectively the interface between the actual implementation of the library module and the client-module. As long as the definition module remains constant, the client-module or the implementation module can be modified (and re-compiled) without the other module being re-compiled. This naturally facilitates efficient program development.

Once a library module has been defined its contents become available to any other module wishing to use it, by means of an IMPORT statement. Clearly, nothing stops a library module from being defined in terms of existing library modules. This "snowball" effect promotes a very healthy building-block approach, whereby sophisticated modules can be developed from the more simple, basic modules.

This thesis has dealt specifically with the implementation of concurrency in Modula-2 making use of the library facilities offered.

Certain observations have been made and conclusions drawn:

a) Flexibility

Although the system, as supplied, is considered adequate, there is every possibility that a particular facility is either not offered or does not quite meet one's specific

needs. In both cases the solution is simple - replace the offending module with something of one's own design.

It is important to note that this process is one of addition. Even though the offending module has been superceded it need not necessarily be destroyed. The mature library might well consist of modules very similar in theme (but subtly different) to demonstrate various schools of thought or degrees of complexity. On a more specific level, the library can be stocked with modules providing traditional concurrent structures (SEND and WAIT) or the more exotic forms (Ada rendezvous). The user is then at liberty to choose the module he thinks might best suit his needs.

The library facility has thus provided the user with a flexible, dynamic nature.

Examples of library implemented facilities can be found in:

i) a simple nucleus [Hop81]
ii) a generic sort [Wie84].

b) The library as a teaching aid

If a user obtains an implementation of any of the other existing concurrent languages, he is presented with one method of structuring a concurrent program. Any deviation from this will require another concurrent language.

Since Modula-2 is capable of emulating the concurrent

constructs of other languages, it alone can be used to demonstrate the constructs offered by a number of languages without actually procuring implementations thereof. As a result, Modula-2, by means of its library, becomes a valuable teaching aid.

c) Lack of protection

Two points are worthy of discussion:

i) The library procedure call

Normally a single library procedure is sufficient to achieve a particular goal. If, for example, one wishes to initiate a process, one makes a call to procedure StartProcess:

eg.

```
StartProcess(Producer, 400)
```

It just so happens that to initiate a process one need only call one library procedure, a one-to-one match.

However, the implementation of the Ada rendezvous, and its associated constructs (see section 5 "Modula-2 and the Ada Task"), and the monitor implementation (see section 4 "Modula-2 and the Monitor Concept") have adopted a new composite approach, ie. to achieve a particular goal one must make more than one library procedure call, a one-to-many match.

eg.

```
Select(Warehouse);
  SAccept(Warehouse, Put, P);
  ElseAccept(Warehouse, Ship);
EndSelect(Warehouse, Action); Action;
```

where four calls are necessary to emulate the Ada selective wait rendezvous (see section 5 "Modula-2 and the Ada Task").

If any of the key calls, eg. Select(Warehouse), are omitted, the entire structure collapses. Although the user associates the calls with one another the library considers each call, to a large degree, as a separate entity. This can only be rectified by the introduction of intensive and possibly time-consuming checking.

Whether the capabilities of the library have been over-extended in this instance is a debateable issue.

This loose inter-relationship between library calls will, however, have to be tightened if any form of protection is to be offered to the user when he makes use of such library-implemented facilities.

ii) Responsibility

The success of any of these composite structures is the sole responsibility of the user. If, for example, a user forgets to terminate a monitor procedure with a call to procedure ReleaseEx, the monitor structure is incomplete and as such will not execute correctly.

Ideally, one should strive for an implementation which

allows the user to concentrate almost all his attention on, for example, what the monitor procedure will do, as opposed to what will designate a procedure a monitor procedure.

Cynical as this might sound, the less responsibility the user must assume and the fewer formalities he must remember, so much the better.

6.5 Modula

Modula was designed by Niklaus Wirth in 1975 and is the predecessor of Modula-2.

Concurrency in Modula is an intrinsic part of the language implementation and, as such, differs considerably from its successor.

Processes are declared in much the same way as procedures

```
Process Terminal(Channel: Integer);
```

and are initiated by a call not unlike a procedure call

```
Terminal(1); Terminal(2);
```

Inter-process communication is achieved by condition variables with SEND and WAIT operators being applied to them

```
Send(S);
Wait(S);
```

where S is a condition variable.

The condition queues are ordered on a FIFO basis, but priorities can be assigned to a waiting process

```
Wait(S, n)
```

where n is a cardinal number. When no priority is assigned to a waiting process, it is assigned the highest priority, ie. 1.

Any module declaration preceded by the keyword interface is designated a monitor (similar to that proposed by Hoare) [Hoa74].

```
interface module Warehouse;
    {monitor procedures}
begin
    {initialisation code}
end Warehouse;
```

The differences between concurrency in Modula and Modula-2 can thus be summarised as follows:

a) the method by which concurrency has been implemented (hard wired as opposed to library module)
b) the method of process designation and initiation (defined as a process, as opposed to a procedure which can be designated a process)
c) the ability to declare processes with parameters
d) condition queue ordering (priority as opposed to FIFO).

BIBLIOGRAPHY

BIBLIOGRAPHY

Add80 Addyman, A.M. A Draft Proposal for Pascal. ACM SIGPLAN Notices, 15, (4), p28 (1980).

Bar82 Barnes, J.G.P. Programming in Ada. Addison-Wesley, London, England. (1980).

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

Ben82 Ben-Ari, M. Principles of Concurrent Programming. Prentice-Hall Inc., Englewood Cliffs, New Jersey. p68 (1982).

Bod83 Boddy, D.E. Implementing Data Abstractions and Monitors in UCSD Pascal. ACM SIGPLAN Notices, 18, (5), 15-23 (1983).

Bod84 Boddy, D.E. On the design of Monitors with Priority Conditions. ACM SIGPLAN Notices, 19, (2), 38-46 (1984).

Bor83 Turbo Pascal Reference Manual. Borland International. p216 (1983).

Bra82 Branquart, P., Louis, G. and Woden, P. An Analytical Description of CHILL, the CCITT High Level Language. Lecture Notes in Computer Science, 128. Springer-Verlag, Berlin. (1982).

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

Bri81 Brinch Hansen, Per. Edison - A Multiprocessor Language. Software - Practice and Experience, 11, (4), (1981).

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

Cai82 Cailliau, R. (Letter to the editor). ACM SIGPLAN Notices, 17, (12), 10-11 (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).

Cla83 Clancy, P. (Letter to the editor). ACM SIGPLAN Notices, 18, (4), 19-21 (1983).

Gle83 Gleaves, R.E. (Letter to the editor). ACM SIGPLAN Notices, 18, (3), 11-13 (1983).

Gro80 Grogono, P. Programming in Pascal. Addison-Wesley, Reading , Massachusetts. 263-268 (1980).

Ham83 Manual for Modula-2 on the VAX-11. University of Hamburg. (1983).

Hoa74 Hoare, C.A.R. Monitors: An Operating System Structuring Concept. Comm. ACM, 17, (10), 549-557 (1974).

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).

Hop80 Hoppe, J. A Simple Nucleus Written in Modula-2: A Case Study. Software - Practice and Experience, 10, (9), 697-706 (1980).

Jen74 Jensen, K. and Wirth, N. Pascal -- User Manual and Report. Lecture Notes in Computer Science, 18. Springer-Verlag, Berlin. (1974).

Jrt83 JRT Pascal 3.0 User's Guide. JRT Systems. p184 (1983).

Koe82 Koehler, S. and Clark, R. The UCSD Pascal Handbook. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. p120 (1982).

Kri80 Kritz, J. and Sandmayr, H. Extension of Pascal by Coroutines and its Application to Quasi-Parallel Programming and Simulation. Software - Practice and Experience, 10, (10), 773-789 (1980).

Mtm81 Pascal/MT+ User's Guide. MT Microsystems Inc. p149 (1981).

Sof81 UCSD Pascal 1V.0 User Manual and Internal Architecture Guide. Softech Microsystems. (1981).

Spe82 Spector, D. Ambiguities and Insecurities in Modula-2. ACM SIGPLAN Notices, 17, (8), 43-51 (1982).

Sum82 Sumner, R.T. and Gleaves, R.E. Modula-2 -- A Solution to Pascal's Problems. ACM SIGPLAN Notices, 17, (9), 28-33 (1982).

Uni81 United States Department of Defense. The Programming Language Ada -- Reference Manual. Lecture Notes in Computer Science, 106. Springer-Verlag, Berlin. (1981).

Vol83 Manual for Modula-2 on the SAGE IV. Volition Systems. (1983).

Weg80 Wegner, P. Programming with Ada: an introduction by means of graduated examples. Prentice-Hall Inc., Englewood Cliffs, New Jersey. (1980).

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).

Wie84 Wiener, R.S. A Generic Sorting Module in Modula-2. ACM SIGPLAN Notices, 19, (3), 66-72 (1984).

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).

You82 Young, S.J. Real Time Languages - design and development. Ellis Horwood, Chichester, England. (1982).

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

APPENDIX A

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

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

FROM SYSTEM IMPORT PROCESS;

EXPORT QUALIFIED StartProcess,SEND,WAIT,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 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                                  *
 *                                                       *
 *     As suggested by Wirth [Wir83, page 132]           *
 *                                                       *
 *     Machine details: Volition Systems'                *
 *                       implementation of Modula-2      *
 *                      SAGE IV microcomputer            *
 *                                                       *
 *     Date: June, 1984                                  *
 *                                                       *
 *********************************************************)

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

VAR CP:SIGNAL;  (* pointer to current process *)

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

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

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

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;
                          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 *)

(* 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.                       *
 *                                                       *
 *     Machine details: Volition Systems'                *
 *                       implementation of Modula-2      *
 *                      SAGE IV microcomputer            *
 *                                                       *
 *     Date: June, 1984                                  *
 *                                                       *
 *********************************************************)

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

VAR CP:SIGNAL;  (* pointer to current process *)

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

PROCEDURE StopProcess;
 (* Terminate the current process *)
 VAR SØ,NEXTJOB: SIGNAL;  (* temporary *)
     NOTFOUND: BOOLEAN;   (* ready process not found *)
 BEGIN
  IF CP = CP^.next
    THEN (* END OF PROGRAM *)
         HALT
  END;
  SØ:=CP; NEXTJOB:=CP; NOTFOUND:=TRUE;
  REPEAT  (* find next ready-to-run process *)
   CP:=CP^.next;
```

```
   IF ((CP^.ready) AND (NOTFOUND))
     THEN NEXTJOB:=CP;
          NOTFOUND:=FALSE
   END
  UNTIL CP^.next = SØ;
  CP^.next:=SØ^.next;  (* remove descriptor from ring *)
  IF NOTFOUND
    THEN (* DEADLOCK *)
         HALT
    ELSE CP:=NEXTJOB;
         TRANSFER(SØ^.cor, CP^.cor)
  END
 END StopProcess;

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

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

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

APPENDIX C

```
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                                  *
 *                                                       *
 *********************************************************)

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 SØ: SIGNAL;     (* temporary, launching process *)
     WSP: ADDRESS;  (* workspace *)
 BEGIN
  SØ:=CP;
  ALLOCATE(CP, TSIZE(ProcessDescriptor));
  ALLOCATE(WSP, N);
  WITH CP^ DO  (* link descriptor into ring *)
   next:=SØ^.next;
   SØ^.next:=CP;
   ready:=TRUE;
   queue:=NIL;
  END;
  NEWPROCESS(P, WSP, N, CP^.cor);
  TRANSFER(SØ^.cor, CP^.cor)
 END StartProcess;

PROCEDURE StopProcess;
 (* Terminate the current process *)
 VAR SØ,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 SØ:=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 D

```
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;
                          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.                       *
 *      Process priorities included. Process ring        *
 *       ordered according to priority.                  *
 *                                                       *
 *     Machine details: Volition Systems'                *
 *                       implementation of Modula-2      *
 *                      SAGE IV microcomputer            *
 *                                                       *
 *     Date: June, 1984                                  *
 *                                                       *
 *********************************************************)

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

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 SØ,S1,PREVIOUS: SIGNAL;  (* temporary *)
     WSP: ADDRESS;  (* workspace *)
 BEGIN
  SØ:=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;
   queue:=NIL;
  END;
```

```
  NEWPROCESS(P, WSP, N, CP^.cor);
  TRANSFER(SØ^.cor, CP^.cor)
 END StartProcess;

PROCEDURE StopProcess;
 (* Terminate the current process *)
 VAR SØ,NEXTJOB: SIGNAL;  (* temporary *)
     NOTFOUND: BOOLEAN;   (* ready process not found *)
 BEGIN
  IF CP = CP^.next
    THEN (* END OF PROGRAM *)
         HALT
  END;
  SØ:=CP; NEXTJOB:=CP; NOTFOUND:=TRUE;
  REPEAT  (* find next ready-to-run process *)
   CP:=CP^.next;
   IF ((CP^.ready) AND (NOTFOUND))
     THEN NEXTJOB:=CP;
          NOTFOUND:=FALSE
   END
  UNTIL CP^.next = SØ;
  CP^.next:=SØ^.next;  (* remove descriptor from ring *)
  IF SØ = TOPPROC
    THEN (* process with top priority terminated *)
         TOPPROC:=TOPPROC^.next
  END;
  IF NOTFOUND
    THEN (* DEADLOCK *)
         HALT
    ELSE CP:=NEXTJOB;
         TRANSFER(SØ^.cor, CP^.cor)
  END
 END StopProcess;

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

BEGIN  (* Processes *)
 ALLOCATE(CP, TSIZE(ProcessDescriptor));
 WITH CP^ DO  (* enter main program into ring *)
  next:=CP; ready:=TRUE; prior:=7; queue:=NIL;
 END;
 TOPPROC:=CP
END Processes.
```

APPENDIX E

```
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;
                          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 a 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.             *
 *                                                           *
 *     Machine details: Volition Systems'                    *
 *                       implementation of Modula-2          *
 *                      SAGE IV microcomputer                *
 *                                                           *
 *     Date: June, 1984                                      *
 *                                                           *
 *************************************************************)

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

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 SØ,S1,PREVIOUS: SIGNAL;  (* temporary *)
     WSP: ADDRESS;            (* workspace *)
 BEGIN
  SØ:=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;
   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(SØ^.cor, CP^.cor)
 END StartProcess;

PROCEDURE StopProcess;
 (* Terminate the current process *)
 VAR SØ,NEXTJOB: SIGNAL;  (* temporary *)
 BEGIN
  IF CP = CP^.next
    THEN (* END OF PROGRAM *)
         HALT
  END;
  SØ:=CP;
  REPEAT  (* find next ready-to-run process *)
   CP:=CP^.next;
  UNTIL CP^.next = SØ;
  CP^.next:=SØ^.next;  (* remove descriptor from ring *)
  IF SØ = 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(SØ^.cor, CP^.cor)
    ELSE (* DEADLOCK *)
         HALT
  END
 END StopProcess;

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

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

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

APPENDIX F

Clock Implementation to provide Interrupt Driven Process Switching

The Modula-2 system on the SAGE IV implements soft interrupts, ie. it simulates an interrupt system in software rather than relying directly on the underlying hardware interrupt mechanism.

An interrupt system handler, written in assembly language and residing in the SAGE BIOS, fields hardware interrupts and signals the P-code interpreter to cause a soft interrupt through the proper interrupt vector. The soft interrupt in turn causes an IOTRANSFER which awakens a Modula-2 process.

The IOTRANSFER command takes the following form:

```
IOTRANSFER(A, B, Interruptvector)
```

Assuming the interrupt will trigger via Interruptvector, control will be transferred from process A to process B and upon occurrence of the interrupt, control will be transferred back to process A.

Sixteen interrupt vectors, numbered Ø - 15, are defined, each one being associated with an interrupt priority. Sixteen interrupt priorities are defined. They are numbered Ø - 15, with Ø being the lowest priority. Interrupt vector priorities control the order in which pending (soft) interrupts are acknowledged by the system and are completely independent of the interrupt vector numbers.

Four of these interrupt vectors have been set aside for clock timing, each with the same characteristics:

vector number	event number
9	36
8	37
7	38
6	39

The event number is a number used to program the clock scheduler with the UnitWrite intrinsic.

The actual clock timing is controlled by an array:

```
T: ARRAY [Ø..1] OF CARDINAL
```

The interval can either be a number of seconds or part of a second or a combination of both:

i) a positive number of seconds (0 - 65535)
ii) an unsigned integer (0 - 63999) to represent the number of 1/64000 ths of a second

for a fraction of a second set T as follows:
```
T[0]:=0;
T[1]:= number of 1/64000 ths of a second;
```

for a number of seconds set T as follows:
```
T[0]:= number of seconds;
T[1]:=0;
```

for a combination of both set T as follows:
```
T[0]:= number of seconds;
T[1]:= number of 1/64000 ths of a second;
```

The UnitWrite intrinsic is used to prime the clock device - device 131. It will also be used to "turn off" the clock. UnitWrite has six parameters:

a) device number: 131
b) timing details: address of array T (to start clock) or NIL (to stop clock)
c) 0
d) 0
e) event number associated with particular clock interrupt vector being used
f) start the clock: 1
   stop the clock: 0

Various other procedures related to interrupts are defined:

SetPriority sets the current processor priority to the specified value and returns the old priority as a result. It is used to assign the correct priority value to the clock.

```
eg: P:=SetPriority(3)
```

Attach and **Detach** connect and disconnect the Modula-2 interrupt vectors to the interrupt system defined in the SAGE BIOS.

```
eg: Attach(9)
    Detach(9)
```

ClearVector is used to dissociate a process from an interrupt vector which it has performed IOTRANSFER's to. It is always passed NIL's as address parameters.

```
eg: ClearVector(NIL, NIL, 9)
```

The clock scheduler can now be constructed and fitted into module Processes.

The following data structures are needed:

```
CONST  Clock1Vector  = 9;    (* clock interrupt vector *)
       Event1Number  = 36;   (* clock event number *)
       ClockDevice   = 131;  (* clock device number *)
       ClockPriority = 3;    (* clock priority *)
       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 *)
```

The clock is set up as a process:

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

The code necessary to get the clock started and stopped will be organised as initialisation and finalisation code which will envelope any module importing module Processes.

```
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;
  TRANSFER(Userprocess, CLK);
  CP^.cor:=Userprocess;
  UNITWRITE(ClockDevice, ADR(T), 0, 0, Event1Number,
                                       {StartClock})
 END CLKinit;
```

Here the first call to SetPriority sets the processor priority to ClockPriority, ensuring that the CLOCK process acquires that priority, and the second call resets the processor priority to what it was originally.

The assignments made to T ensure a 100 millisecond timeslice.

A similar problem arises to that experienced in the COBEGIN . . COEND construct, that of attempting to transfer control to the main program using its .cor field before it has been assigned anything. In this instance, CP is pointing to the ProcessDescriptor of the main program and the assignment, CP^.cor:=Userprocess, solves the problem.

```
PROCEDURE CLKterm;
 (* Start 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;
```

The SetEnvelope construct will be used to envelope the two procedures CLKinit and CLKterm around any module that imports module Processes.

APPENDIX G

```
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;
                          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 proceses. Always activate   *
 *       process with highest priority.                  *
 *      Interrupt driven process switching added.        *
 *                                                       *
 *     Machine details: Volition Systems'                *
 *                       implementation of Modula-2      *
 *                      SAGE IV microcomputer            *
 *                                                       *
 *     Date: June, 1984                                  *
 *                                                       *
 *********************************************************)

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 *)
      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 INTEGER;         (* 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 SØ,S1,PREVIOUS: SIGNAL;  (* temporary *)
     WSP: ADDRESS;  (* workspace *)
 BEGIN
  SØ:=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;
   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(SØ^.cor, CP^.cor)
 END StartProcess;

PROCEDURE StopProcess;
 (* Terminate the current process *)
 VAR SØ,NEXTJOB: SIGNAL;  (* temporary *)
 BEGIN
  IF CP = CP^.next
    THEN (* END OF PROGRAM *)
         HALT
  END;
  SØ:=CP;
  REPEAT  (* find next ready-to-run process *)
   CP:=CP^.next;
  UNTIL CP^.next = SØ;
  CP^.next:=SØ^.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(SØ^.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;
  IF CP <> CP^.next
    THEN REPEAT  (* find next ready-to-run process *)
          CP:=CP^.next
         UNTIL CP^.ready;
         Userprocess:=CP^.cor
    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.
```

APPENDIX H

```
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                                  *
 *                                                       *
 *********************************************************)

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 INTEGER;         (* 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 SØ,S1,PREVIOUS: SIGNAL;  (* temporary *)
     WSP: ADDRESS;  (* workspace *)
 BEGIN
  SØ:=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(SØ^.cor, CP^.cor)
 END StartProcess;

PROCEDURE StopProcess;
 (* Terminate the current process *)
 VAR SØ,NEXTJOB: SIGNAL;  (* temporary *)
 BEGIN
  IF CP = CP^.next
    THEN (* END OF PROGRAM *)
         HALT
  END;
  SØ:=CP;
  REPEAT  (* find next ready-to-run process *)
   CP:=CP^.next;
  UNTIL CP^.next = SØ;
  CP^.next:=SØ^.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(SØ^.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 = Ø
    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.
```

APPENDIX I

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

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

FROM SYSTEM IMPORT PROCESS;

EXPORT QUALIFIED GainEx,ReleaseEx,
                 Init,
                 StartProcess,StopProcess,WaitCondition,SendCondition,
                 SIGNAL;


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


PROCEDURE GainEx;
     (* Gain monitor exclusivity *)

PROCEDURE ReleaseEx;
     (* Release monitor exclusivity *)

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

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

PROCEDURE StopProcess;
     (* Terminate the current process *)

PROCEDURE WaitCondition(VAR S:SIGNAL);
     (* WaitCondition for some process to SendCondition S *)

PROCEDURE SendCondition(VAR S:SIGNAL);
     (* Reactivate a process WaitConditioning on S *)


END HMonitor.
```

```
IMPLEMENTATION MODULE HMonitor[1];

(*********************************************************
 *                                                       *
 *     MODULE HMonitor                                   *
 *                                                       *
 *     An implementation of Hoare's                      *
 *     monitor mechanism                                 *
 *                                                       *
 *     Machine details: Volition Systems'                *
 *                       implementation of Modula-2      *
 *                      SAGE IV microcomputer            *
 *                                                       *
 *     Date: June, 1984                                  *
 *                                                       *
 *********************************************************)

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

CONST Clock1Vector  = 9;    (* clock interrupt vector *)
      Event1Number  = 36;   (* clock event number *)
      ClockDevice   = 131;  (* clock device number *)
      ClockPriority = 3;    (* clock priority *)
      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 GainEx,ReleaseEx,
       Init,
       StartProcess,StopProcess,WaitCondition,SendCondition,
       Reschedule,CP;

VAR CP: SIGNAL;           (* pointer to current process *)
    HPqueue,              (* high priority queue *)
    Mqueue: SIGNAL;       (* monitor queue *)
    OCCUPIED: BOOLEAN;    (* monitor is occupied *)
```

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

PROCEDURE StopProcess;
 (* Terminate the current process *)
 VAR SØ,NEXTPROCESS: SIGNAL;  (* temporary *)
     NOTFOUND: BOOLEAN;       (* ready process not found *)
 BEGIN
  IF CP=CP^.next
    THEN (*END OF PROGRAM*)
         HALT
  END;
  SØ:=CP; NEXTPROCESS:=CP; NOTFOUND:=TRUE;
  REPEAT  (* find next ready-to-run process *)
   CP:=CP^.next;
   IF ((CP^.ready) AND (NOTFOUND))
     THEN NEXTPROCESS:=CP;
          NOTFOUND:=FALSE
   END
  UNTIL CP^.next = SØ;
  CP^.next:=SØ^.next;  (* remove descriptor from ring *)
  IF NOTFOUND
    THEN (*DEADLOCK*)
         HALT
    ELSE CP:=NEXTPROCESS;
         TRANSFER(SØ^.cor, CP^.cor)
  END
 END StopProcess;

PROCEDURE WAIT(VAR S:SIGNAL);
 (* WAIT for some process to SEND S *)
 VAR SØ,S1: SIGNAL;
 BEGIN
  IF S = NIL
    THEN (* first such process to WAIT *)
         S:=CP
    ELSE (* add to existing queue *)
         SØ:=S;
         S1:=SØ^.queue;
```

```
          WHILE S1 <> NIL DO  (* search for tail *)
           S0:=S1;
           S1:=S1^.queue
          END;
          S0^.queue:=CP
  END;
  S0:=CP;
  REPEAT CP:=CP^.next UNTIL CP^.ready;  (* find next process *)
  IF CP = S0
    THEN (*DEADLOCK*)
          HALT
  END;
  S0^.ready:=FALSE;  (* deactivate process *)
  TRANSFER(S0^.cor, CP^.cor)
 END WAIT;

PROCEDURE SEND(VAR S:SIGNAL);
 (* Reactivate a process WAITing or WaitConditioning for 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 WaitCondition(VAR S:SIGNAL);
 (* WaitCondition for some process to SendCondition S *)
 VAR S0,S1: SIGNAL;  (* temporary *)
 BEGIN
  IF S = NIL
    THEN (* first such process to WaitCondition *)
          S:=CP
    ELSE (* add to existing queue *)
          S0:=S; S1:=S^.queue;
          WHILE S1 <> NIL DO  (* search for tail *)
           S0:=S1;
           S1:=S1^.queue
          END;
          S0^.queue:=CP;
  END;
  CP^.ready:=FALSE;  (* deactivate process *)
  IF HPqueue <> NIL
    THEN (* a process is WAITing on the high priority queue *)
          SEND(HPqueue)
    ELSE IF Mqueue <> NIL
           THEN (* a process is WAITing on the monitor queue *)
                SEND(Mqueue)
           ELSE OCCUPIED:=FALSE;  (* monitor now unoccupied *)
                S0:=CP;
```

```
                REPEAT  (* find next ready process *)
                 CP:=CP^.next
                UNTIL ((CP^.ready) OR (CP = SØ));
                IF CP = SØ
                  THEN (*DEADLOCK*)
                       HALT
                  ELSE TRANSFER(SØ^.cor, CP^.cor)
                END
        END
  END
 END WaitCondition;

PROCEDURE SendCondition(VAR S:SIGNAL);
 (* Reactivate a process WAITCONDITIONing for S *)
 VAR SØ,S1: SIGNAL;  (* temporary *)
 BEGIN
  (* place current process on high priority queue *)
  IF HPqueue = NIL
    THEN (* first process on queue *)
         HPqueue:=CP
    ELSE (* add to existing queue *)
         SØ:=HPqueue; S1:=HPqueue^.queue;
         WHILE S1 <> NIL DO  (* search for tail *)
          SØ:=S1; S1:=S1^.queue
         END;
         SØ^.queue:=CP
  END;
  CP^.ready:=FALSE;  (* deactivate process *)
  IF S <> NIL
    THEN SEND(S)
    ELSE (* no processes WaitConditioning on S, remove
            process from high priority queue *)
         IF HPqueue = CP
           THEN HPqueue:=NIL
           ELSE SØ^.queue:=NIL
         END;
         CP^.ready:=TRUE  (* reactivate process *)
  END
 END SendCondition;

PROCEDURE GainEx;
 (* Attempt to gain monitor exclusivity *)
 BEGIN
  IF CP^.excl > Ø
    THEN (* monitor exclusivity already gained,
            internal monitor call *)
         CP^.excl:=CP^.excl + 1
    ELSE IF OCCUPIED
           THEN WAIT(Mqueue)  (* monitor occupied - WAIT *)
         END;
         OCCUPIED:=TRUE;  (* monitor exclusivity gained *)
         CP^.excl:=1
  END
 END GainEx;
```

```
PROCEDURE ReleaseEx;
 (* Release monitor exclusivity *)
 BEGIN
  IF CP^.excl = 1
    THEN (* release monitor exclusivity *)
         CP^.excl:=0;
         IF HPqueue <> NIL
           THEN (* process WAITing on high priority queue *)
                SEND(HPqueue)
           ELSE IF Mqueue <> NIL
                  THEN (* process WAITing on monitor queue *)
                       SEND(Mqueue)
                END
         END;
         OCCUPIED:=FALSE  (* monitor unoccupied *)
    ELSE (* return from internal monitor call *)
         CP^.excl:=CP^.excl - 1
  END
 END ReleaseEx;

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

PROCEDURE Reschedule;
 (* Interrupt driven process switch. Find the next ready-to-run
    process in the process ring and activate it *)
 BEGIN
  CP^.cor:=Userprocess;
  REPEAT  (* find next ready-to-run process *)
   CP:=CP^.next
  UNTIL CP^.ready;
  Userprocess:=CP^.cor
 END Reschedule;

BEGIN  (* SYNCHRO *)
 ALLOCATE(CP, TSIZE(ProcessDescriptor));
 WITH CP^ DO  (* enter main program into ring *)
  next:=CP; queue:=NIL;  excl:=0; ready:=TRUE
 END;
 OCCUPIED:=FALSE;  (* monitor unoccupied *)
 Init(HPqueue); Init(Mqueue)
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  (* HMonitor *)
 SetEnvelope(CLKinit, CLKterm, FirstCall)
END HMonitor.
```

APPENDIX J

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

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

FROM SYSTEM IMPORT PROCESS;

EXPORT QUALIFIED StartTask,EndTask,Call,Accept,EndAccept,Entry,
                 Select,SAccept,ElseAccept,EndSelect,
                 RENDEZVOUS,SELECTREND;


TYPE TASK = POINTER TO TaskDescriptor;
     TaskDescriptor = RECORD
                        next: TASK;
                        callerqueue: TASK;
                        serverqueue: TASK;
                        cor: PROCESS;
                        ready: BOOLEAN
                      END;

     RENDEZVOUS = POINTER TO RendezvousDescriptor;
     RendezvousDescriptor = RECORD
                             caller: TASK;
                             server: TASK
                            END;

     SELECTREND = POINTER TO SelectRendDescriptor;
     SelectRendDescriptor = RECORD
                             selren: RENDEZVOUS;
                             assproc: PROC;
                             squeue: SELECTREND
                            END;


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

PROCEDURE EndTask;
     (* Terminate the current task *)

PROCEDURE Call(VAR R:RENDEZVOUS);
     (* Request a rendezvous *)

PROCEDURE Accept(VAR R:RENDEZVOUS);
     (* Offer a service *)

PROCEDURE EndAccept(VAR R:RENDEZVOUS);
     (* End of rendezvous *)

PROCEDURE Entry(VAR R:RENDEZVOUS);
     (* Rendezvous variable initialisation *)

PROCEDURE Select(VAR SR:SELECTREND);
     (* Selective wait rendezvous initialisation *)
```

```
PROCEDURE SAccept(VAR SR:SELECTREND; R:RENDEZVOUS; SPROC:PROC);
     (* Offer a service within a selective wait rendezvous *)

PROCEDURE ElseAccept(VAR SR:SELECTREND; SPROC:PROC);
     (* Else option within a selective wait rendezvous *)

PROCEDURE EndSelect(VAR SR:SELECTREND; VAR APROC:PROC);
     (* End of selective wait rendezvous *)


END AdaTasks.
```

```
IMPLEMENTATION MODULE AdaTasks[1];

(*********************************************************
 *                                                       *
 *     MODULE AdaTasks                                   *
 *                                                       *
 *     An implementation of the Ada task                 *
 *      together with the associated constructs          *
 *      available for concurrent programming.            *
 *                                                       *
 *     Machine details: Volition Systems'                *
 *                       implementation of Modula-2      *
 *                      SAGE IV microcomputer            *
 *                                                       *
 *     Date: June, 1984                                  *
 *                                                       *
 *********************************************************)

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

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

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

MODULE TaskSynchro[4];
  (* non-interruptable module to protect task ring *)

IMPORT ADDRESS,TSIZE,PROCESS,NEWPROCESS,TRANSFER,LISTEN;
IMPORT ALLOCATE;
IMPORT WriteString;
IMPORT Usertask;
IMPORT TASK,TaskDescriptor;
IMPORT RENDEZVOUS,RendezvousDescriptor;
IMPORT SELECTREND,SelectRendDescriptor;

EXPORT StartTask,EndTask,Call,Accept,EndAccept,Entry,
       Select,SAccept,ElseAccept,EndSelect,
       Reschedule,CT;
```

```
VAR CT: TASK;              (* pointer to current task *)
    AVAIL: SELECTREND;     (* pointer to available selective
                              wait rendezvous entries *)

PROCEDURE StartTask(P:PROC; N:CARDINAL);
 (* Start a task P with workspace size of N *)
 VAR T0: TASK;         (* temporary, launching task *)
     WSP: ADDRESS;     (* workspace *)
 BEGIN
  T0:=CT;
  ALLOCATE(CT, TSIZE(TaskDescriptor));
  ALLOCATE(WSP, N);
  WITH CT^ DO  (* link descriptor into ring *)
   next:=T0^.next;
   T0^.next:=CT;
   callerqueue:=NIL;
   serverqueue:=NIL;
   ready:=TRUE
  END;
  NEWPROCESS(P, WSP, N, CT^.cor);
  CT:=T0
 END StartTask;

PROCEDURE EndTask;
 (* Terminate the current task *)
 VAR T0,NEXTTASK: TASK;  (* temporary *)
     NOTFOUND: BOOLEAN;  (* ready-to-run task not found *)
 BEGIN
  IF CT = CT^.next
    THEN (* END OF PROGRAM *)
         HALT
  END;
  T0:=CT; NEXTTASK:=CT; NOTFOUND:=TRUE;
  REPEAT  (* find next ready-to-run task *)
   CT:=CT^.next;
   IF ((NOTFOUND) AND (CT^.ready))
     THEN NEXTTASK:=CT;
          NOTFOUND:=FALSE
   END
  UNTIL CT^.next = T0;
  CT^.next:=T0^.next;  (* remove task from ring *)
  IF NOTFOUND
    THEN (* DEADLOCK *)
         HALT
    ELSE CT:=NEXTTASK;
         TRANSFER(T0^.cor, CT^.cor)
 END
END EndTask;
```

```
PROCEDURE Call(VAR R:RENDEZVOUS);
 (* Request a rendezvous *)
 VAR T0,T1: TASK;  (* temporary *)
 BEGIN
  IF R^.caller = NIL
    THEN (* first such task to request a rendezvous *)
         R^.caller:=CT
    ELSE (* add to existing queue *)
         T0:=R^.caller;
         T1:=T0^.callerqueue;
         WHILE T1 <> NIL DO  (* search for tail *)
          T0:=T1;
          T1:=T0^.callerqueue
         END;
         T0^.callerqueue:=CT
  END;
  IF ((R^.server = NIL) OR ((R^.server <> NIL)
                       AND (R^.server^.ready)))
    THEN (* service not available - wait *)
         T0:=CT;
         REPEAT  (* find next ready-to-run task *)
          CT:=CT^.next
         UNTIL CT^.ready;
         IF CT = T0
           THEN (* DEADLOCK *)
                HALT
         END;
         T0^.ready:=FALSE;  (* deactivate caller *)
         TRANSFER(T0^.cor, CT^.cor)
    ELSE (* service available - rendezvous *)
         T0:=CT;
         T0^.ready:=FALSE;  (* deactivate caller *)
         CT:=R^.server;
         CT^.ready:=TRUE;  (* reactivate server *)
         TRANSFER(T0^.cor, CT^.cor)
  END
 END Call;

PROCEDURE Accept(VAR R:RENDEZVOUS);
 (* Offer a service / rendezvous *)
 VAR T0,T1: TASK;  (* temporary *)
 BEGIN
  IF R^.server = NIL
    THEN (* first such task to offer service *)
         R^.server:=CT
    ELSE (* add to existing queue *)
         T0:=R^.server;
         T1:=T0^.serverqueue;
         WHILE T1 <> NIL DO  (* search for tail *)
          T0:=T1;
          T1:=T0^.serverqueue
         END;
         T0^.serverqueue:=CT
  END;
```

```
    IF R^.caller = NIL
      THEN (* no requests for service *)
           T0:=CT;
           REPEAT  (* find next ready-to-run task *)
            CT:=CT^.next
           UNTIL CT^.ready;
           IF CT = T0
             THEN (* DEADLOCK *)
                  HALT
           END;
           T0^.ready:=FALSE;  (* deactivate server *)
           TRANSFER(T0^.cor, CT^.cor)
    END
  END Accept;

 PROCEDURE EndAccept(VAR R:RENDEZVOUS);
  (* End of rendezvous *)
  VAR T0,T1: TASK;  (* temporary *)
  BEGIN
   (* remove caller from rendezvous caller queue *)
   T0:=R^.caller;
   R^.caller:=R^.caller^.callerqueue;
   T0^.ready:=TRUE;  (* reactivate caller *)
   (* remove server from rendezvous server queue *)
   T0^.callerqueue:=NIL;
   R^.server:=R^.server^.serverqueue
  END EndAccept;

 PROCEDURE Entry(VAR R:RENDEZVOUS);
  (* Initialise a rendezvous variable *)
  VAR R0: RENDEZVOUS;
  BEGIN
   ALLOCATE(R0, TSIZE(RendezvousDescriptor));
   R:=R0;
   WITH R0^ DO
    caller:=NIL;
    server:=NIL
   END
  END Entry;

 PROCEDURE Select(VAR SR:SELECTREND);
  (* Initialise a selective wait rendezvous structure *)
  BEGIN
   SR:=NIL
  END Select;
```

```
PROCEDURE SAccept(VAR SR:SELECTREND; R:RENDEZVOUS; SPROC:PROC);
 (* Offer a service / rendezvous from within a selective wait
    rendezvous *)
 VAR SR0,SR1,SR2: SELECTREND;
 BEGIN
  IF AVAIL = NIL
    THEN (* no available descriptors *)
         ALLOCATE(SR0, TSIZE(SelectRendDescriptor))
    ELSE (* descriptors available *)
         SR0:=AVAIL;
         AVAIL:=AVAIL^.squeue
  END;
  IF SR = NIL  (* link descriptor into queue *)
    THEN (* first entry in queue *)
         SR:=SR0
    ELSE (* add to existing queue *)
         SR1:=SR;
         REPEAT  (* search for tail *)
          SR2:=SR1;
          SR1:=SR1^.squeue
         UNTIL SR1 = NIL;
         SR2^.squeue:=SR0
  END;
  WITH SR0^ DO
   selren:=R;
   assproc:=SPROC;
   squeue:=NIL
  END
 END SAccept;

PROCEDURE ElseAccept(VAR SR:SELECTREND; SPROC:PROC);
 (* Else option within a selective wait rendezvous *)
 VAR SR0,SR1,SR2: SELECTREND;  (* temporary *)
 BEGIN
  IF AVAIL = NIL
    THEN (* no available descriptors *)
         ALLOCATE(SR0, TSIZE(SelectRendDescriptor))
    ELSE (* descriptors available *)
         SR0:=AVAIL;
         AVAIL:=AVAIL^.squeue
  END;
  IF SR = NIL  (* link descriptor into queue *)
    THEN (* first entry in queue *)
         SR:=SR0
    ELSE (* add to existing queue *)
         SR1:=SR;
         REPEAT  (* search for tail *)
          SR2:=SR1;
          SR1:=SR1^.squeue
         UNTIL SR1 = NIL;
         SR2^.squeue:=SR0
  END;
```

```
  WITH SRØ^ DO
   selren:=NIL;
   assproc:=SPROC;
   squeue:=NIL
  END
 END ElseAccept;

(* Function *) PROCEDURE Checking(SR:SELECTREND;
                                  VAR ACTIONPROC:PROC): BOOLEAN;
 (* Check whether any task has requested a service offered within
    the selective wait rendezvous or for an ELSE option *)
 VAR SRØ: SELECTREND;  (* temporary *)
     TØ,T1: TASK;      (* temporary *)
     FOUND: BOOLEAN;   (* rendezvous requested or
                          ELSE option encountered *)
 BEGIN
  IF SR = NIL
    THEN WriteString('Select alternatives missing');
         HALT
    ELSE SRØ:=SR; FOUND:=FALSE;
         REPEAT  (* search for rendezvous request or ELSE option *)
          IF SRØ^.selren = NIL
            THEN (* ELSE option encountered *)
                 FOUND:=TRUE;
                 ACTIONPROC:=SRØ^.assproc;
                 RETURN FALSE
            ELSE IF SRØ^.selren^.caller <> NIL
                   THEN (* caller requesting rendezvous,
                           link server to server queue *)
                        IF SRØ^.selren^.server = NIL
                          THEN SRØ^.selren^.server:=CT
                          ELSE TØ:=SRØ^.selren^.server;
                               T1:=TØ^.serverqueue;
                               WHILE T1 <> NIL DO
                                TØ:=T1;
                                T1:=TØ^.serverqueue
                               END;
                               TØ^.serverqueue:=CT
                        END;
                        FOUND:=TRUE;
                        ACTIONPROC:=SRØ^.assproc;
                        RETURN FALSE
                   ELSE (* check next entry *)
                        SRØ:=SRØ^.squeue
                 END
          END
         UNTIL ((FOUND) OR (SRØ = NIL));
         IF NOT FOUND
           THEN (* no rendezvous requests or ELSE option *)
                RETURN TRUE
         END
 END
END Checking;
```

```
PROCEDURE EndSelect(VAR SR:SELECTREND; VAR APROC:PROC);
 (* End of selective wait rendezvous *)
 VAR SR1: SELECTREND;  (* temporary, selective wait rendezvous
                          pointer *)
 BEGIN
  WHILE Checking(SR, APROC) DO  (* check for rendezvous request *)
   LISTEN                       (* or ELSE option               *)
  END;
  WHILE SR <> NIL DO  (* add to queue of available descriptors *)
   SR1:=SR;
   SR:=SR^.squeue;
   SR1^.squeue:=AVAIL;
   AVAIL:=SR1
  END
 END EndSelect;

PROCEDURE Reschedule;
 (* Interrupt driven task switch. Find the next ready-to-run
    task in the task ring and activate it *)
 BEGIN
  CT^.cor:=Usertask;
  REPEAT  (* find next ready-to-run task *)
   CT:=CT^.next
  UNTIL CT^.ready;
  Usertask:=CT^.cor
 END Reschedule;

BEGIN  (* TaskSYNCHRO *)
 ALLOCATE(CT, TSIZE(TaskDescriptor));
 WITH CT^ DO  (* enter main program into ring *)
  next:=CT; callerqueue:=NIL; serverqueue:=NIL;
  ready:=TRUE
 END;
 AVAIL:=NIL
END TaskSynchro;

PROCEDURE CLOCK;
 (* Interrupt driver *)
 BEGIN
  Attach(Clock1Vector);
  LOOP
   IOTRANSFER(CLK, Usertask, 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(Usertask, CLK);
  CT^.cor:=Usertask;
  UnitWrite(ClockDevice, ADR(T), 0, 0, Event1Number, {StartClock})
 END CLKinit;

PROCEDURE CLKterm;
 (* Stop the 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  (* AdaTasks *)
 SetEnvelope(CLKinit, CLKterm, FirstCall)
END AdaTasks.
```