ETH
Eidgenössische Technische Hochschule
Zürich
Institut für Informatik

N. Wirth

SCHEMES FOR MULTIPROGRAMMING AND THEIR IMPLEMENTATION IN MODULA-2

REVISIONS AND AMENDMENTS TO MODULA-2

June 1984

59

Address of the author:

Institut für Informatik
ETH-Zentrum
CH-8092 Zürich / Switzerland

Schemes for Multiprogramming and their Implementation in Modula-2

N. Wirth

Abstract

Two sets of primitive operators for communication among concurrent processes are presented. They characterise the schemes of communication via *shared variables* and signals for synchronisation and by *message passing* through channels. Their use is demonstrated by simple and typical examples, and their implementation is described in terms of Modula-2. Both implementations are based on coroutines for a single-processor computer (Lilith). The primitives for coroutine handling are also presented as a Modula-2 module, demonstrating the language's low-level facilities. A variant of the module for signals is adapted to the needs of discrete event simulation.

These modules, whose implementation is brief and efficient, demonstrate that, although Modula-2 lacks a specific construct for multiprogramming, such facilities are easily expressible and that a programmer can choose a suitable scheme by selecting the appropriate low-level (library) module. The conclusion is that, if a language offers low-level facilities and an encapsulation construct, it need not offer specific features for multiprogramming. To the contrary, this might hamper the programmer in his search for an appropriate and effective solution.

Keywords: Multiprogramming, coroutines, simulation, Modula-2.

Introduction

A multiprogram is a program specifying several (perhaps many) sequential processes which are executed concurrently. The objective of multiprogramming is to achieve and guarantee a harmonious cooperation among the processes. This requires principles, paradigms, and primitives for communication among and synchronization of concurrent activities. Numerous concepts have been postulated in the literature, several have been incorporated in programming languages, and thorough comparisons and evaluations have recently been published [6,9,10].

Common to these proposals is the notion that individual processes are sequences of actions performed at arbitrary speed. Hence the notion of time is absent, with the exception that synchronization primitives allow a given process to be delayed until a certain condition has been established by other processes. Typically, such synchronizations occur infrequently; hence, we speak of *loosely coupled* processes in contrast to arrays of processes which "march in locked step".

Processes have to be synchronized when a cooperation is planned. In computing processes, cooperation is synonymous with communication. Communication implies the exchange of information, i.e. of data. Here we distinguish between two views, namely communication by *sharing variables* and by *passing messages*. They reflect an implied assumption of an underlying mechanism to execute the processes, which, if all details are ignored, distinguishes between individual processors sharing a common store (i.e. being connected by a common bus with high bandwidth), or being truly distributed and communicating via "thin" wires. Although it is possible to realize each view with a system representing the other, it is appropriate not to carry the idea of abstraction too far, because the price of inefficiency can become unreasonably high. The message passing paradigm has only recently become relevant, because modern technology has made systems with larger numbers of distributed processors economically attractive. It is therefore not surprising that programming languages are mainly oriented towards facilities for communication via shared variables (e.g. Concurrent Pascal, Modula, Ada). The shift of interest to message based communication is evident in Occam [11].

Although we shall see that implementations of the two schemes on a single-processor system are very similar, we must keep in mind that the essence of message passing is that the information is transmitted *by value*, and that no shared variables exist. A system in which so-called messages are pointers (e.g. pointing to sections of a shared buffer), cannot honestly be classified as a message passing system. It is the very fact of transmission by value which makes this scheme conceptually simple and therefore attractive.

A particular instance of synchronization that stems from the paradigm of communication via shared variables is *mutual exclusion*. It embodies a facility to grant exclusive access to certain variables, i.e. to exclude other processes from access until the one process relinquishes its priviledge. Mutual exclusion is a practical necessity. Although it can be programmed using synchronization primitives, it is advantageous to offer a specific language construct to express mutual exclusion over specific procedures. Such a construct is the so-called monitor [2]. Message passing systems require no mutual exclusion.

Another important, basic notion is that processes are truly concurrent, that each is executed by an individual processor. In practice, thanks to our postulate that no assumption about speed is made, a processor may be utilized to execute several processes in pieces. We therefore must distinguish between logical processes and physical processors, keeping in mind that this distinction is a matter of implementation technique. If individual computer users represent the processes, we speak about time-sharing systems; if the processes

represent distinct concurrent activities of the same user, we rather speak about multiprocessing, but both represent the same idea of utilizing the same processor for several processes. Instead of genuine concurrency we speak about *quasi-concurrency*. The important point is that the processes are conceived such that it does not matter whether several processors are employed or a single one is used in time-shared fashion. This gives the implementor the desirable freedom to utilize his resources optimally, and makes the program independent of the configuration of actual systems. This kind of abstraction from actual processor configurations is at the heart of multiprogramming.

It follows that actual resource management functions must be evoked only when a process specifies synchronization or communication which - as we have seen - are tightly coupled concepts. As a consequence, resource allocation, i.e. processor switching activities, can be hidden behind synchronization and communication statements. The former are implemented in lower-level modules, the latter are calls of procedures of these modules. It is crucial to keep these levels conceptually separated, for this is an indispensible precondition for being able to adapt the resource managing procedures to individual processor configurations.

Subseqently, several schemes of multiprogramming will be presented. Each provides a set of primitives for synchronization or communication in the form of a set of procedures. They are specified in terms of a Modula-2 definition module [8]. Brief examples of applications will be shown which are clients of these definition modules. Then, we discuss their corresponding implementation modules representing the lower-level resource management. The implementations are based on a single processor configuration. The implementation is such that specifics of the used computer occur only rarely.

Coroutines

The basic facility of all systems that allow the switching of a processor from one logical process to another is the *coroutine*. A process is implemented as a coroutine. It is evident to the programmer that processes are executed in alteration, and his program explicitly states the places when a switch must occur. We denote the primitive statement of switching from a coroutine p to a coroutine q by *Transfer(p,q)*. The receiving coroutine q is resumed after the Transfer statement it last executed, and finds the computation in exactly the same state as the sender p put it before executing that transfer statement. Hence, this statement is nothing more than an explicit processor scheduling operator.

A coroutine consists of a piece of program specifying the activities and (in general) a set of variables that are local to it. In Modula-2, it is expressed in terms of a procedure and a so-called workspace within which local data are allocated upon procedure calls. A coroutine is established by the primitive procedure *InitCoroutine*. Its parameters are the procedure P which constitutes the program, the address *wsp* of its workspace, and the size of that workspace. A call of InitCoroutine does not activate the coroutine, i.e. does not divert the processor. Instead, it merely initializes a descriptor placed within the workspace such that a later Transfer statement starts the new coroutine with the first statement of P. The workspace address is used as parameter in the Transfer statement. The use of the type ADDRESS manifests that the coroutine is a low-level concept. The two primitives Transfer and InitCoroutine are defined in terms of the following definition module.

```
DEFINITION MODULE Coroutines;  (*NW 16.3.84*)
  FROM SYSTEM IMPORT ADDRESS;
  PROCEDURE Transfer(VAR from, to: ADDRESS);
  PROCEDURE InitCoroutine(P: PROC; wsp: ADDRESS; size: CARDINAL);
END Coroutines.
```

The following implementation of the module is specific for the Lilith computer. For other machines, it would typically be programmed in assembler code, as the actual procedure bodies consist of a few instructions only. The type Coroutine defines the structure of the coroutine descriptor which is placed at the head of the workspace and represents the coroutine's state while it is suspended (G, L, PC, M, S, H denote actual processor registers).

```
IMPLEMENTATION MODULE Coroutines;   (*NW 18.3.84*)
  (*implementation for Lilith system*)
  FROM SYSTEM IMPORT ADDRESS, ADR;

  TYPE CorPtr = POINTER TO Coroutine;

    Coroutine =
      RECORD
        G:      ADDRESS;
        L:      ADDRESS;
        PC:     ADDRESS;
        M:      BITSET;
        S:      ADDRESS;
        H:      ADDRESS;
        err:    CARDINAL;
        trapMask: BITSET;
        start: PROC;      (*start of workspace*)
        scnt:  CARDINAL
      END ;

  PROCEDURE GlobalBase(): ADDRESS;
    CODE 25B; 0     (*LGA 0*)
  END GlobalBase;

  PROCEDURE CALL;
    CODE 357B   (*call procedure variable*)
  END CALL;

  PROCEDURE TRA(VAR from, to: ADDRESS);
    CODE 256B; 0   (*transfer*)
  END TRA;

  PROCEDURE Transfer(VAR from, to: ADDRESS);
  BEGIN TRA(from, to)
  END Transfer;

  PROCEDURE InitCoroutine(P: PROC; wspa: ADDRESS; size: CARDINAL);
    VAR cor: CorPtr;

    PROCEDURE SetPC;
      PROCEDURE pc(): CARDINAL;
        CODE 40B; 2  (*LLW 2*)
      END pc;
    BEGIN cor↑.PC := pc() + 1
    END SetPC;

   EGIN cor := wspa;
    WITH cor↑ DO
      G := GlobalBase(); L := 0;
      M := {};           S := ADR(scnt)+1;
```

```
    H := wspa + size;  err := 0;
    trapMask := {};
    start := P;        scnt := 0
  END ;
  SetPC;
  RETURN;
  CALL; HALT
 END InitCoroutine;
END Coroutines.
```

In general, we do not recommend the use of coroutines in this direct form. It leaves a system no free room for its own resource management and burdens the program with details of processor allocation that obscure the actual task. However, in cases where the management strategy is simple and obvious, and where efficiency is of prime importance, the use of this lowest-level facility is justified. Such a case is the handling of coprocessors (devices) which communicate by interrupts.

We regard a device (such as a printer) as a processor. Since we usually wish to include in the printer process certain activities, such as buffer handling and status checking, which are beyond the device's capabilities, the process is split up into two parts. One part consists of the (non-programmable) activities of the device (such as the actual printing), the other of the programmed activities requiring the capabilities of a general processor. This obviously requires processor switching. When the programmed part is completed (which involves the activation of the device), the programmable processor is switched by an *explicit* transfer statement to any other resumable activity. When the device has completed its part, it indicates this fact by an interrupt signal. This causes an *implicit* transfer statement to be executed, switching the general processor back to the point of resumption determined by the explicit transfer. The following application example shows this scheme in connection with a laser printer. The programmed part is usually called *interrupt routine*, the part executed by the printer is represented by the explicit Transfer statement. We consider it as crucial to view the interrupt routine as part of the whole, cyclic process, and the interrupt as a non-scheduled coroutine transfer. This view clarifies otherwise rather obscure matters considerably. It raises the machine-level facility of interrupt to that of a structured language and permits implementation without sacrifice of efficiency, which in this case is essential.

```
MODULE PrinterDriver;
  IMPORT ADDRESS, WORD, ADR, InitCoroutine, Transfer;
  EXPORT out;

  CONST size = 100;
  VAR printer, main: ADDRESS;  (*coroutine pointers*)
      buffer: ...;
      wsp: ARRAY [0..size-1] OF WORD;

  PROCEDURE P;
  BEGIN
    LOOP
      (*if buffer not empty, fetch data from buffer and feed them to
        printer interface register, then activate printer*)
      Transfer(printer, main)
    END
  END P;

  PROCEDURE out(data: Type);
```

```
    BEGIN (*deposit data in buffer*)
      IF "printer idle" THEN Transfer(main, printer) END
    END out;

  BEGIN (*initialize buffer*)
    InitCoroutine(P, wsp, size); printer := ADR(wsp)
  END PrinterDriver
```

Signals

If we wish to genuinely abstract from the handling of physical processors and wish to assume that an individual agent executes each process, the coroutine concept fails. The key to a better abstraction lies in making processes anonymous in the sense that they do not explicitly specify suspension and resumption of a named partner. Their synchronization is achieved by a new primitive. Such primities are for example the *semaphore* [1] and the *condition* [2]. Here we present *signals* which effectively are identical to conditions [4].

A signal is declared like a variable, although it has no value and therefore must not be copied or assigned. It can be sent and received. The sending of a signal s signifies that a certain condition P_s (among variables) has been established. After receiving s the receiving process can therefore proceed under the assumption of this condition. P_s is a precondition of send(s) and a postcondition of receive(s). The signal s is the message that P_s holds. These operations are defined in the definition module *Signals*, which also contains a procedure *StartProcess (P)* and a Boolean function *Expected (s)*. P is a parameterless procedure that constitutes the program of the started process, and Expected(s) means "at least one process is waiting to receive s".

```
DEFINITION MODULE Signals;  (*NW 23.3.84*)
  TYPE Signal;

  PROCEDURE StartProcess(P: PROC);
    (*start a concurrent process with program P*)
  PROCEDURE Send(VAR s: Signal);
    (*one process waiting to receive s is resumed*)
  PROCEDURE Receive(VAR s: Signal);
    (*wait until you receive s*)
  PROCEDURE Expected(s: Signal): BOOLEAN;
  PROCEDURE InitSignal(VAR s: Signal);
END Signals.
```

The use of signals is demonstrated by the well-known example of a pair of processes cooperating as producer and consumer of data that are fed through a buffer. The buffer and its associated variables - a count n of the number of items in the buffer and indices denoting the next empty slot and the next item to be fetched - together form the interface between the two processes. This interface is appropriately formulated as a (local) module and constitutes a monitor over the buffer. It contains the signals *nonempty* with associated condition $n > 0$ and *nonfull* with condition $n < N$.

An interface (monitor) typically contains those local variables which are shared among processes. Since signals are inherently shared, signals should only occur within interfaces. The rule that shared objects always be declared within the interface constitutes an important discipline in multiprogramming and was postulated by Hoare and Brinch Hansen.

Shared variables should be protected by mutual exclusion. This means that within the

interface the ordinary rules of sequential programming apply, because no two processes can be performing interface actions at the same time. In our example, we have omitted a mutual exclusion specification. We have the right to do so under the assumptions that the program is executed on a system with a single (shared) processor, and that processor switching cannot occur except through sending or receiving a signal.

It should be noted that in this example it is evident which process receives a signal when it is sent, because there are only two participants. But this is not the case in general; the receiver may be any of a number of processes waiting for the specified signal. However, a single send operation will cause (at most) one receive operation (no broadcast).

```
MODULE ProdCons;   (*NW 17.3.84*)
  FROM Terminal IMPORT Read, Write;
  FROM Signals IMPORT
    Signal, StartProcess, Send, Receive, InitSignal;

  MODULE Interface;
    IMPORT Signal, StartProcess, Send, Receive, InitSignal;
    EXPORT get, put;
    CONST N = 8;

    VAR n, in, out: CARDINAL;
      nonfull, nonempty: Signal;
      buf: ARRAY [0 .. N-1] OF CHAR;

    PROCEDURE put(ch: CHAR);
    BEGIN
      IF n = N THEN Receive(nonfull) END ;
      n := n+1; buf[in] := ch; in := (in+1) MOD N;
      Send(nonempty)
    END put;

    PROCEDURE get(VAR ch: CHAR);
    BEGIN
      IF n = 0 THEN Receive(nonempty) END ;
      n := n-1; ch := buf[out]; out := (out+1) MOD N;
      Send(nonfull)
    END get;

  BEGIN n := 0; in := 0; out := 0;
    InitSignal(nonfull); InitSignal(nonempty)
  END Interface;


  PROCEDURE Producer;
    VAR i: CARDINAL; ch: CHAR;
      text: ARRAY [0..99] OF CHAR;
  BEGIN Write("(");
    i := 0; text := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    WHILE text[i] > 0C DO
      Write("!"); Write(text[i]); put(text[i]); i := i+1
    END ;
    Write("!"); Write(")"); put(0C)
  END Producer;

  PROCEDURE Consumer;
```

```
    VAR ch: CHAR;
  BEGIN Write("[");
    Write("?"); get(ch);
    WHILE ch > 0C DO
      Write(ch); Write("?"); get(ch)
    END ;
    Write("]")
  END Consumer;

BEGIN
  StartProcess(Producer); Consumer; Write("$"); Write(36C)
END ProdCons.
```

Verification of correctness is now possible without considering possible sequences of process interactions. It begins with establishing an interface invariant, which in this case specifies that the buffer cannot be emptier than empty and not fuller than full: $0 \leq n \leq N$. Together with the postcondition of *Receive(nonfull)*, $n < N$, and of *Receive(nonempty)*, $n > 0$, it follows that an item is fetched only if $0 < n \leq N$, and deposited if $0 \leq n < N$. Note that all verification deliberations can be carried out with the interface, because they involve local objects only.

As Dijkstra noted long ago, this classical solution has one drawback: signals are sent more often than needed. For example, *nonempty* is sent whenever the producer has deposited an item, whereas it has only an effect if the consumer had actually been waiting for it. This could be overcome by subjecting *send(nonempty)* to the condition (guard) *Expected(s)*. Dijkstra thereupon proposed the inclusion of this information in the counter variable n and called his solution the "sleeping barber". Values $n > 0$ denote the number of items (clients) in the buffer (ante room), values $n < 0$ denote the number of waiting consumers (idle barbers). Once again, we note that the scheme smoothly extends to the case of several consumers and procuders without affecting the reasoning necessary to establish correctness. We subsequently present the interface modified to reflect the "sleeping barber" metaphor.

```
MODULE Interface;
  IMPORT Signal, StartProcess, Send, Receive, InitSignal;
  EXPORT get, put;
  CONST N = 8;

  VAR n: INTEGER; in, out: CARDINAL;
    nonfull, nonempty: Signal;
    buf: ARRAY [0 .. N-1] OF CHAR;

  PROCEDURE put(ch: CHAR);
  BEGIN n := n+1;
    IF n > N THEN Receive(nonfull) END ;
    buf[in] := ch; in := (in+1) MOD N;
    IF n = 0 THEN Send(nonempty) END
  END put;

  PROCEDURE get(VAR ch: CHAR);
  BEGIN n := n-1;
    IF n < 0 THEN Receive(nonempty) END ;
    ch := buf[out]; out := (out+1) MOD N;
    IF n = N THEN Send(nonfull) END
  END get;
```

```
BEGIN n := 0; in := 0; out := 0;
  InitSignal(nonfull); InitSignal(nonempty)
END Interface
```

Implementation of Signals

One might suspect that the implementation of the signalling mechanism with its implied processor management might be fairly complex and thereby introduce detrimental inefficiencies. In fact, most available multi-tasking operating systems lend ample justification for such suspicion. Fortunately, an implementation can be straight-forward and efficient, as the following solution demonstrates.

It is based on the premise that each generated process is represented by a descriptor. All descriptors are linked in a ring structure. The one process currently under execution is designated by the pointer variable cp. (In the case of a system with several processors, each processor has its private cp.) StartProcess(P) allocates a new descriptor (RingNode) and inserts it in the ring. It also allocates a workspace for the process and initializes the descriptor. The details of initializing the workspace are the same as in the module Coroutines, witnessing the fact that the scheme using signals is built on the basis of coroutines.

Particularly noteworthy is the representation of signals. Instead of letting processes specify the signal for which they are waiting, a signal specifies the processes which are waiting for another process to send it. Hence, a signal is a pointer variable heading the list (queue) of waiting processes. The procedure Send(s) merely implies a coroutine transfer to a process in the list s and the removal of the respective descriptor from the list. Even fairness of scheduling can easily be guaranteed; processes are always appended at the list's tail, and removed from its head. The procedure Receive(s) is only slightly more complicated: After traversing the list down to its tail to append the descriptor of the current process, a search proceeds along the ring to find any resumable (ready) process. If none exists, the system of processes is in deadlock.

A more sophisticated solution would remove waiting processes from the ring, i.e. actually move them from the ring to the signal's list. This would require that they be moved back to the ring when receiving the sent signal. The gain in efficiency (when searching through the ring) does not seem to warrant the additional complexity in pointer manipulation, unless the waiting processes far outnumber the ready ones.

```
IMPLEMENTATION MODULE Signals;   (*NW 23.3.84*)
  FROM SYSTEM IMPORT ADDRESS, WORD, ADR, TSIZE;
  FROM Heap IMPORT Allocate;

  CONST WorkspaceSize = 200B;

  TYPE Signal = POINTER TO RingNode;
       CorPtr = POINTER TO Coroutine;

    RingNode =
      RECORD
        next, prev: Signal;  (*ring*)
        queue:      Signal;  (*queue of waiting processes*)
        cor:        CorPtr;
        ready:      BOOLEAN
      END ;
```

```
  Coroutine =
    RECORD
      G:      ADDRESS;
      L:      ADDRESS;
      PC:     ADDRESS;
      M:      BITSET;
      S:      ADDRESS;
      H:      ADDRESS;
      err:    CARDINAL;
      trapMask: BITSET;
      start: PROC;      (*start of workspace*)
      scnt:  CARDINAL;
      wsp:   ARRAY [0 .. WorkspaceSize-1] OF WORD
    END ;

VAR cp: Signal;  (*current process*)
   aux: Signal;
  free: Signal;  (*chain of free process descriptors*)

PROCEDURE TRANSFER(VAR from, to: CorPtr);
  CODE 256B; 0
END TRANSFER;

PROCEDURE StartProcess(P: PROC);

  PROCEDURE GlobalBase(): ADDRESS;
    CODE 25B; 0    (*LGA 0*)
  END GlobalBase;

  PROCEDURE CALL;
    CODE 357B      (*CF*)
  END CALL;

  PROCEDURE SetPCandTransfer;
    PROCEDURE pc(): CARDINAL;
      CODE 40B; 2  (*LLW 2*)
    END pc;
  BEGIN cp↑.cor↑.PC := pc() + 1; TRANSFER(aux↑.cor, cp↑.cor)
  END SetPCandTransfer;

BEGIN aux := cp;
  (*allocate a RingNode and a workspace contiguously*)
  IF free = NIL THEN
    Allocate(cp, TSIZE(RingNode)); Allocate(cp↑.cor, TSIZE(Coroutine))
  ELSE cp := free; free := free↑.next
  END ;
  WITH cp↑ DO
    next := aux↑.next; prev := aux; queue := NIL; ready := TRUE
  END ;
  aux↑.next := cp; cp↑.next↑.prev := cp;
  WITH cp↑.cor↑ DO
    G := GlobalBase(); L := 0;
    M := {};           S := ADR(wsp);
    H := ADR(wsp) + WorkspaceSize;
```

```
    err := 0;           trapMask := {};
    start := P;         scnt := 0
  END ;
  SetPCandTransfer;
  RETURN;

  CALL;       (*activate process body P*)
  aux := cp; cp := aux↑.next;
  cp↑.prev := aux↑.prev; aux↑.prev↑.next := cp;
  aux↑.next := free; free := aux; aux := cp;
  WHILE NOT cp↑.ready & (cp # aux) DO cp := cp↑.next END ;
  IF cp↑.ready THEN TRANSFER(free↑.cor, cp↑.cor) END ;
  HALT (*deadlock*)
END StartProcess;

PROCEDURE Send(VAR s: Signal);
  VAR this: Signal;
BEGIN this := cp;
  IF s # NIL THEN cp := s;
    s := cp↑.queue; cp↑.ready := TRUE
  ELSE (*release*)
    REPEAT cp := cp↑.next UNTIL cp↑.ready
  END ;
  IF cp # this THEN TRANSFER(this↑.cor, cp↑.cor) END
END Send;

PROCEDURE Receive(VAR s: Signal);
  VAR this: Signal;
BEGIN (*insert cp at end of queue s*)
  IF s = NIL THEN s := cp
  ELSE this := s;
    WHILE this↑.queue # NIL DO this := this↑.queue END ;
    this↑.queue := cp
  END ;
  this := cp; this↑.queue := NIL;
  REPEAT cp := cp↑.next UNTIL cp↑.ready;
  this↑.ready := FALSE;
  IF cp = this THEN (*deadlock*) HALT END ;
  TRANSFER(this↑.cor, cp↑.cor)
END Receive;

PROCEDURE Expected(s: Signal): BOOLEAN;
BEGIN RETURN s # NIL
END Expected;

PROCEDURE InitSignal(VAR s: Signal);
BEGIN s := NIL
END InitSignal;

BEGIN free := NIL; Allocate(cp, TSIZE(RingNode));
  WITH cp↑ DO
    next := cp; prev := cp; ready := TRUE
  END
END Signals.
```

If a process reaches the end of the procedure which constitutes its body, the process supposedly terminates. Control then returns to the point after the statement CALL which initiated the process. The subsequent statements transfer the processor to any ready process in the ring, and they reclaim the terminating process' workspace by inserting it into the list headed by the pointer variable *free*. Hence, the presented implementation also incorporates a primitive management of workspace allocation and recycling. Normally, process creation and termination occurs far less frequently than the sending of a signal, i.e. the switching of the processor. As far as efficiency is concerned, only the procedures Send and Receive need be carefully planned.

Closely related to signals are also the semaphores as originally postulated by Dijkstra. A semaphore consists of a counter and an associated signal. The P-operation decrements the counter and, if the result is negative, waits until the signal is received. The V-operation increments the counter and, if the result is not positive, sends the signal. A negative value of the counter indicates how many processes are waiting in the signal's queue.

Channels

If a multiprocess system consists of truly distributed processors connected with data channels rather than of processors accessing a common store, communication cannot be expressed using shared variables. One will then prefer the scheme proposed by Hoare in CSP [5] and embodied by the language Occam [11]. Basically it consists of the following facilities:

1. The statement PAR S_0, S_1 ... S_{n-1} specifies that the statements S_0, S_1, ..., S_{n-1} are executable concurrently.
2. The declaration CHAN ch introduces a communication channel.
3. The statement "ch ? x" specifies the reception of a value from channel ch and the value's assignment to variable x.
4. The statement "ch ! x" specifies the evaluation of expression x and the sending of the resulting value over channel ch.

We first propose a translation of the terse Occam notation into an equivalent form in Modula-2, and then present an implementation. Evidently, the mentioned Occam facilities are defined in terms of a definition module.

```
DEFINITION MODULE Channels;  (*NW 21.3.84*)
  TYPE Message = INTEGER;
       Process;
       Channel = RECORD prod, cons: Process END ;

  PROCEDURE Parallel(P,Q: PROC);
  PROCEDURE Send(VAR ch: Channel; msg: Message);
  PROCEDURE Receive(VAR ch: Channel; VAR msg: Message);
  PROCEDURE SenderWaiting(VAR ch: Channel): BOOLEAN;
  PROCEDURE ReceiverWaiting(VAR ch: Channel): BOOLEAN;
  PROCEDURE InitChannel(VAR ch: Channel);
END Channels.
```

In order to emphasize their relation to the scheme using signals, the output operator ! is translated into Send, the input operator ? into Receive. The signal, which may be regarded as an empty message, is replaced by a message having a value. The Occam statement PAR P Q is translated into the statement Parallel(P',Q'), where P' and Q' are (parameterless)

procedures representing the Occam statements P and Q. PAR P Q R is expressed as Parallel(P,QR), where QR is a procedure with body Parallel(Q,R). The additional function procedures SenderWaiting and ReceiverWaiting are the analogous to Expected(s).

The properties of channels are the following:

1. If a sender outputs a message to a channel, it is delayed until a receiver picks up the message (at the other end of the channel). This may be instantaneous, if a receiver was already waiting on the channel.
2. A receiver expecting input from a channel gets delayed until a sender outputs a message on the channel. This may be instantaneous, if a sender was already waiting for his message to be picked up.

It follows that a channel automatically also acts as a synchronizing agent; synchronization and communication have become the same. Also, the channel is truly a "wire" and has no implied buffering capability. It therefore forces sender and receiver to a rendez-vous; the channel is not a mailbox, just a meeting point.

The paradigm of Occam is that a program represents a system of processes connected by channels, and that the connections are fixed. We may therefore assume that each channel connects one sender with one receiver.

The following application demonstrates a simple but typical system using a channel. Process P reads a sequence of numbers from the input medium and feeds them into the channel. Every 4th number is a checksum and is not passed on. Process Q receives the numbers from the channel, computes a checksum after every 7th number, and feeds them to the output medium.

```
MODULE Sequences;
  FROM Channels IMPORT Channel, InitChannel, Parallel, Send, Receive;
  FROM InOut IMPORT ReadInt, WriteInt;

  VAR ch: Channel;

  PROCEDURE P;
    VAR i: CARDINAL; x, sum: INTEGER;
  BEGIN ReadInt(x);
    WHILE x # 0 DO i := 3;
      WHILE i > 0 DO
        Send(ch, x); sum := sum + x; i := i-1; ReadInt(x)
      END ;
      sum := sum + x; (*check sum = 0*) ReadInt(x)
    END ;
    Send(ch, 0)
  END P;

  PROCEDURE Q;
    VAR i: CARDINAL; x, sum: INTEGER;
  BEGIN Receive(ch, x);
    WHILE x # 0 DO
      i := 6; sum := 0;
      WHILE i > 0 DO
        WriteInt(x,6); sum := sum - x; i := i-1; Receive(ch, x)
      END ;
      WriteInt(sum, 6)
    END
```

```
  END Q;
BEGIN InitChannel(ch); Parallel(P,Q)
END Sequences.
```

Implementation of Channels

The implementation of channels resembles that of signals to a high degree. Processes are again linked in a ring, each element (node) containing (a pointer to) a coroutine workspace. A further field stores the address of the message to be passed. It is necessary to store an address instead of the message itself, because then upon arrival of the sender it can store the message in the variable designated by the waiting receiver's VAR parameter.

The data type *Channel* assumes the role of the type Signal. It contains two record fields, one for a waiting receiver, the other for a waiting sender. The alternative solution with a single process field and a discriminator between senders and receivers was rejected, because it requires a more cumbersome program.

Because of the chosen scheme of process creation, processes always occur in pairs. Each process descriptor therefore contains a field designating the partner. Whereas in the case of signals the termination of a process had no other effect, in this case termination implies an implicit synchronization. The creating process proceeds only after both offsprings have terminated. This requires knowledge about the identity of the partner. In fact, in the presented implementation the statement Parallel(P,Q) does not create two new processes, but rather only one. The other is identical to the generating process. This is practically mandatory, because otherwise half of the requested workspace would be wasted for ancestors awaiting termination of both of their offsprings.

```
IMPLEMENTATION MODULE Channels;  (*NW 25.3.84*)
  FROM SYSTEM IMPORT ADDRESS, WORD, ADR, TSIZE;
  FROM Heap IMPORT Allocate;

  CONST WorkspaceSize = 200B;

  TYPE
    Process  = POINTER TO RingNode;
    ProcessState = (ready, waiting, terminated);
    CorPtr = POINTER TO Coroutine;

    RingNode =
      RECORD
        next, prev: Process;  (*ring*)
        partner:    Process;
        cor:        CorPtr;
        state:      ProcessState;
        msgAdr:     POINTER TO Message
      END ;

    Coroutine =
      RECORD
        G:      ADDRESS;
        L:      ADDRESS;
        PC:     ADDRESS;
        M:      BITSET;
        S:      ADDRESS;
        H:      ADDRESS;
```

```
      err:   CARDINAL;
      trapMask: BITSET;
      start: PROC;      (*start of workspace*)
      scnt:  CARDINAL;
      wsp:   ARRAY [0 .. WorkspaceSize-1] OF WORD
    END ;

VAR cp: Process;  (*current process*)
  free: Process;  (*chain of free process descriptors*)
   aux: Process;

PROCEDURE TRANSFER(VAR from, to: CorPtr);
  CODE 256B; 0
END TRANSFER;

PROCEDURE Parallel(P,Q: PROC);
  VAR new: Process;  (*process to be created*)

  PROCEDURE GlobalBase(): ADDRESS;
    CODE 25B; 0    (*LGA 0*)
  END GlobalBase;

  PROCEDURE CALL;
    CODE 357B      (*CF*)
  END CALL;

  PROCEDURE SetPCandCall;

    PROCEDURE pc(): CARDINAL;
      CODE 40B; 2  (*LLW 2*)
    END pc;

  BEGIN newt.cort.PC := pc() + 1;
    P;
    WHILE newt.state # terminated DO
      (*release*) aux := cp;
      REPEAT cp := cpt.next UNTIL cpt.state = ready;
      IF cp = aux THEN HALT (*deadlock*) END ;
      auxt.state := terminated; TRANSFER(auxt.cor, cpt.cor)
    END ;
    aux:= newt.prev; newt.prevt.next := newt.next;
    newt.nextt.prev := aux; newt.next := free; free := new
  END SetPCandCall;

BEGIN
  (*allocate a RingNode and a workspace contiguously*)
  IF free = NIL THEN
    Allocate(new, TSIZE(RingNode));
    Allocate(newt.cor, TSIZE(Coroutine))
  ELSE new := free; free := freet.next
  END ;
  WITH newt DO
    next := cpt.next; prev := cp; partner := cp; state := ready
  END ;
  cpt.next := new; newt.nextt.prev := new;
  WITH newt.cort DO
```

```
    G := GlobalBase(); L := 0;
    M := {};           S := ADR(wsp);
    H := ADR(wsp) + WorkspaceSize;
    err := 0;          trapMask := {};
    start := Q;        scnt := 0
  END ;
  SetPCandCall;
  RETURN;

  CALL(*Q*);
  aux := cp;
  IF cp↑.partner↑.state = terminated THEN
    cp := cp↑.partner; cp↑.state := ready
  ELSE
    REPEAT cp := cp↑.next UNTIL cp↑.state = ready;
    IF cp = new THEN HALT (*deadlock*) END
  END ;
  aux↑.state := terminated; TRANSFER(aux↑.cor, cp↑.cor)
END Parallel;

PROCEDURE Send(VAR ch: Channel; msg: Message);
  VAR this: Process;
BEGIN this := cp;
  IF ch.cons # NIL THEN (*wake up consumer*)
    cp := ch.cons; ch.cons := NIL;
    cp↑.state := ready; cp↑.msgAdr↑ := msg
  ELSE (*wait for consumer*)
    IF ch.prod # NIL THEN HALT END ;
    ch.prod := cp; cp↑.msgAdr := ADR(msg);
    REPEAT cp := cp↑.next UNTIL cp↑.state = ready;
    this↑.state := waiting;
    IF cp = this THEN HALT (*deadlock*) END
  END ;
  TRANSFER(this↑.cor, cp↑.cor)
END Send;

PROCEDURE Receive(VAR ch: Channel; VAR msg: Message);
  VAR this: Process;
BEGIN this := cp;
  IF ch.prod # NIL THEN (*wake up consumer*)
    cp := ch.prod; ch.prod := NIL;
    cp↑.state := ready; msg := cp↑.msgAdr↑
  ELSE (*wait for producer*)
    IF ch.prod # NIL THEN HALT END ;
    ch.cons := cp; cp↑.msgAdr := ADR(msg);
    REPEAT cp := cp↑.next UNTIL cp↑.state = ready;
    this↑.state := waiting;
    IF cp = this THEN HALT (*deadlock*) END
  END ;
  TRANSFER(this↑.cor, cp↑.cor)
END Receive;

PROCEDURE SenderWaiting(VAR ch: Channel): BOOLEAN;
```

```
  BEGIN RETURN ch.prod # NIL
  END SenderWaiting;

  PROCEDURE ReceiverWaiting(VAR ch: Channel): BOOLEAN;
  BEGIN RETURN ch.cons # NIL
  END ReceiverWaiting;

  PROCEDURE InitChannel(VAR ch: Channel);
  BEGIN ch.prod := NIL; ch.cons := NIL
  END InitChannel;

BEGIN free := NIL; Allocate(cp, TSIZE(RingNode));
  WITH cp↑ DO
    next := cp; prev := cp; state := ready
  END
END Channels.
```

The complexity of the implementation of Channels is somewhat greater than that of Signals. This is mainly due to the synchronization upon termination. Also, the scheme is less flexible, because each channel can be associated (at a time) with only one sender and one receiver, whereas a signal may (at the same time) be expected by many processes. But this reflects the natural situation in many cases, and a superchannel to which many senders and receivers can be connected simultaneously would certainly be a bad choice for a communications primitive, because its characteristics would already be nontrivial to describe, let alone to implement. Nevertheless, it is probably fair to say that the solution with signals is to be preferred over channels, if the underlying system consists of shared processors and a common, shared store.

Simulation

A language with a facility to express concurrent processes and an implementation with low overhead in process switching are particularly well suited for the simulation of discrete event systems. The previously presented implementation requires only a minimal extension to cater to the needs of discrete event simulation. We subsequently present a possible solution which in its concepts goes back to Simula and proposals by Hoare [3].

The only addition needed, in fact, is the concept of time. In a discrete event simulation, each agent belongs to a category (class, type) of processes (such as customers, tellers, service men in a supermarket), and its behaviour is therefore characterized by a fixed, sequential program. Each identifyable action, taking time t in reality, is in the simulation program expressed by an appropriate statement followed by *hold (t)*. The latter statement suspends the process, until system time has increased by t.

```
DEFINITION MODULE Simulation;  (*NW 3.4.84*)
  TYPE Signal;
    Process = PROCEDURE (CARDINAL);

  VAR Time: CARDINAL;  (*read only*)

  PROCEDURE StartProcess(P: Process; n: CARDINAL);
    (*start a concurrent process with program P(n)*)
  PROCEDURE Send(VAR s: Signal);
    (*one process waiting to receive s is resumed*)
  PROCEDURE Receive(VAR s: Signal);
    (*wait until you receive s*)
```

```
  PROCEDURE Hold(t: CARDINAL);
    (*hold process for t seconds*)
  PROCEDURE InitSignal(VAR s: Signal);
    (*compulsory initialization*)
END Simulation.
```

An example of an application demonstrating the case with which discrete event systems can be expressed in terms of these simple facilties is the following problem:

One day, the Chinese Emperor issued the order that the minimal distance to each of his empire's villages from the capital was to be determined. The method to be followed was the following: Large groups of scouts were to march out into the country, notably at constant speed, in each direction, i.e on each emanating path. Once they would encounter the next village, the group would split up, each subgroup proceeding on an outgoing path. One man would march back to report the time it took to reach the village, another would remain to report to groups arriving later that he had been first.

Our solution is based on each village being represented by a record specifying the village's name (number), the number of outgoing paths, and for each path its destination and distance. The record is also used to record whether the village had been visited already.

```
MODULE Army;  (*NW 4.5.84*)
  FROM InOut IMPORT Done,
       OpenInput, ReadCard, Write, WriteLn, WriteCard, CloseInput;
  FROM Simulation IMPORT Time, StartProcess, Hold;

  CONST MaxNofVil = 32; MaxNofPaths = 8;

  TYPE Village =
    RECORD nofp: CARDINAL;
      visited: BOOLEAN;
      path: ARRAY [0..MaxNofPaths-1] OF
              RECORD destination, distance: CARDINAL END
    END ;

  VAR i: CARDINAL;
    vil: ARRAY [0..MaxNofVil-1] OF Village;

  PROCEDURE Scout(x: CARDINAL);
  (*x = currentVillage * MaxNofPath + direction*)
    VAR here, dir: CARDINAL;
  BEGIN Write("+"); here := x DIV MaxNofPaths; dir := x MOD MaxNofPaths;
    LOOP Hold(vil[here].path[dir].distance);
      here := vil[here].path[dir].destination;
      IF vil[here].visited THEN EXIT END ;
      WriteCard(here, 6); WriteCard(Time, 6); WriteLn;
      vil[here].visited := TRUE; dir := vil[here].nofp - 1;
      WHILE dir > 0 DO
        StartProcess(Scout, here * MaxNofPaths + dir); dir := dir-1
      END
    END ;
    Write("-")
  END Scout;

  PROCEDURE ReadData;
    VAR A, B, d, i: CARDINAL;
```

```
  BEGIN OpenInput("NUM"); ReadCard(A);
    FOR i := 0 TO MaxNofVil-1 DO
      vil[i].nofp := 0; vil[i].visited := FALSE
    END ;
    WHILE Done DO
      ReadCard(B); ReadCard(d);
      WITH vil[A] DO
        path[nofp].destination := B; path[nofp].distance := d;
        nofp := nofp + 1
      END ;
      WITH vil[B] DO
        path[nofp].destination := A; path[nofp].distance := d;
        nofp := nofp + 1
      END ;
      ReadCard(A)
    END ;
    CloseInput
  END ReadData;

BEGIN ReadData; vil[0].visited := TRUE;
  FOR i := 0 TO vil[0].nofp - 1 DO StartProcess(Scout, i) END ;
  Hold(9999); Write("!"); WriteLn
END Army.
```

Implementation of Simulation

Implementation is straight-forward: delayed processes are queued in a list corresponding to an internal signal called TQ, which is sent whenever no process remains ready at the current time. In contrast to the scheme used in module Signals, however, this list is not a first-in first-out queue, but the processes are ordered according to their wake-up time. When no ready process is found (either after a Receive or a hold statement), instead of detecting deadlock, the first process in the queue TQ is resumed after increasing system time (Time) to the value specified as wake-up time. This scheme requires an additional field in the process descriptor, and insertion after hold must include a search for the proper insertion point. Actually, the signal TQ represents a level of lower priority: delayed processes are resumed, i.e. the system time is stepped up only after no processes are found ready at the current time.

```
IMPLEMENTATION MODULE Simulation;   (*NW 3.4.84*)
  FROM SYSTEM IMPORT ADDRESS, WORD, ADR, TSIZE;
  FROM Heap IMPORT Allocate;

  CONST WorkspaceSize = 200B;

  TYPE Signal = POINTER TO RingNode;
       CorPtr = POINTER TO Coroutine;

    RingNode =
      RECORD
        next, prev: Signal;  (*ring*)
        queue:      Signal;  (*queue of waiting processes*)
        cor:        CorPtr;
        ready:      BOOLEAN;
        waketime:   CARDINAL
      END ;
```

```
  Coroutine =
    RECORD
      G:      ADDRESS;
      L:      ADDRESS;
      PC:     ADDRESS;
      M:      BITSET;
      S:      ADDRESS;
      H:      ADDRESS;
      err:    CARDINAL;
      trapMask: BITSET;
      start: Process;      (*start of workspace*)
      param: CARDINAL;
      scnt:  CARDINAL;
      wsp:   ARRAY [0 .. WorkspaceSize-1] OF WORD
    END ;
VAR cp: Signal;  (*current process*)
   aux: Signal;
  free: Signal;  (*chain of free process descriptors*)
    TQ: Signal;  (*chain of delayed processes; time queue*)
PROCEDURE TRANSFER(VAR from, to: CorPtr);
  CODE 256B; 0
END TRANSFER;
PROCEDURE StartProcess(P: Process; n: CARDINAL);
  PROCEDURE GlobalBase(): ADDRESS;
    CODE 25B; 0    (*LGA 0*)
  END GlobalBase;
  PROCEDURE CALL;
    CODE 357B      (*CF*)
  END CALL;
  PROCEDURE SetPCandTransfer;
    PROCEDURE pc(): CARDINAL;
      CODE 40B; 2  (*LLW 2*)
    END pc;
  BEGIN cp↑.cor↑.PC := pc() + 1; TRANSFER(aux↑.cor, cp↑.cor)
  END SetPCandTransfer;
BEGIN aux := cp;
  (*allocate a RingNode and a coroutine workspace*)
  IF free = NIL THEN
    Allocate(cp, TSIZE(RingNode)); Allocate(cp↑.cor, TSIZE(Coroutine))
  ELSE cp := free; free := free↑.next
  END ;
  WITH cp↑ DO
    next := aux↑.next; prev := aux; queue := NIL; ready := TRUE
  END ;
  aux↑.next := cp; cp↑.next↑.prev := cp;
  WITH cp↑.cor↑ DO
    G := GlobalBase(); L := 0;
    M := {};           S := ADR(wsp);
    H := ADR(wsp) + WorkspaceSize;
```

```
    err := 0;          trapMask := {};
    start := P;
    param := n;        scnt := 1
  END ;
  SetPCandTransfer;
  RETURN;
  CALL;      (*activate process body P*)
  aux := cp; cp := aux↑.next;
  cp↑.prev := aux↑.prev; aux↑.prev↑.next := cp;
  aux↑.next := free; free := aux; aux := cp;
  WHILE NOT cp↑.ready & (cp # aux) DO cp := cp↑.next END ;
  IF cp↑.ready THEN TRANSFER(free↑.cor, cp↑.cor)
  ELSIF TQ # NIL THEN
    cp := TQ; TQ := TQ↑.queue; cp↑.ready := TRUE;
    Time := cp↑.waketime; TRANSFER(free↑.cor, cp↑.cor)
  ELSE (*deadlock*) HALT
  END
END StartProcess;

PROCEDURE Send(VAR s: Signal);
  VAR this: Signal;
BEGIN
  IF s # NIL THEN
    this := cp; cp := s;
    WITH cp↑ DO
      s := queue; ready := TRUE
    END ;
    TRANSFER(this↑.cor, cp↑.cor)
  END
END Send;

PROCEDURE release;
  VAR this: Signal;
BEGIN
  this := cp;
  REPEAT cp := cp↑.next UNTIL cp↑.ready;
  this↑.ready := FALSE;
  IF cp = this THEN (*advance time*)
    IF TQ # NIL THEN
      cp := TQ; TQ := TQ↑.queue; cp↑.ready := TRUE;
      Time := cp↑.waketime
    ELSE (*deadlock*) HALT
    END
  END ;
  TRANSFER(this↑.cor, cp↑.cor)
END release;

PROCEDURE Receive(VAR s: Signal);
  VAR this: Signal;
BEGIN (*insert cp at end of queue s*)
  IF s = NIL THEN s := cp
  ELSE this := s;
    WHILE this↑.queue # NIL DO this := this↑.queue END ;
```

```
      this↑.queue := cp
    END ;
    cp↑.queue := NIL; release
  END Receive;

  PROCEDURE Hold(t: CARDINAL);
    VAR T: CARDINAL; this, q0, q1: Signal;
  BEGIN T := Time + t;
    IF TQ = NIL THEN
      TQ := cp; cp↑.queue := NIL
    ELSIF TQ↑.waketime > T THEN
      cp↑.queue := TQ; TQ := cp
    ELSE q0 := TQ;
      LOOP (*q0 # NIL*) q1 := q0↑.queue;
        IF q1 = NIL THEN
          q0↑.queue := cp; cp↑.queue := NIL; EXIT
        ELSIF q1↑.waketime > T THEN
          cp↑.queue := q1; q0↑.queue := cp; EXIT
        ELSE q0 := q1
        END
      END
    END ;
    cp↑.waketime := T; release
  END Hold;

  PROCEDURE InitSignal(VAR s: Signal);
  BEGIN s := NIL
  END InitSignal;

BEGIN free := NIL; Time := 0; TQ := NIL; Allocate(cp, TSIZE(RingNode));
  WITH cp↑ DO
    next := cp; prev := cp; ready := TRUE
  END
END Simulation.
```

Evaluation

The three schemes of process cooperation by coroutine transfer, by signalling, and by message passing have been compared by three test programs CorTest, SigTest, and ChanTest listed subsequently. Each of them contains two processes, among which the processor is switched repeatedly. The times measured for 30000 switches back and forth are

CorTest	1.8 s
SigTest	5.0 s
ChanTest	5.5 s

```
MODULE CorTest;  (*NW 25.6.84*)
  FROM Coroutines IMPORT InitCoroutine;
  FROM Terminal IMPORT Read, Write;
  FROM SYSTEM IMPORT ADR, WORD, ADDRESS;

  CONST WspSize = 200;

  VAR n: CARDINAL; ch: CHAR;
      main, proc: ADDRESS;
      wsp: ARRAY [0..WspSize-1] OF WORD;
```

```
  PROCEDURE Transfer(VAR from, to: ADDRESS);
    CODE 256B; 0
  END Transfer;

  PROCEDURE P;
  BEGIN
    LOOP n := n-1; Transfer(proc, main) END
  END P;

BEGIN n := 60000; proc := ADR(wsp); InitCoroutine(P, proc, WspSize);
  Write("["); Read(ch);
  REPEAT Transfer(main, proc) UNTIL n = 0;
  Write("]")
END CorTest.


MODULE SigTest;  (*NW 24.6.84*)
  FROM Signals IMPORT Signal, StartProcess, Send, Receive, InitSignal;
  FROM Terminal IMPORT Read, Write;

  VAR n: CARDINAL; ch: CHAR; tested: Signal;

  PROCEDURE P;
  BEGIN
    LOOP n := n-1; Receive(tested) END
  END P;

BEGIN n := 60000; InitSignal(tested);
  Write("["); Read(ch); StartProcess(P);
  REPEAT (* n>0 *) Send(tested) UNTIL n = 0;
  Write("]")
END SigTest.


MODULE ChanTest;  (*NW 24.6.84*)
  FROM Channels IMPORT Channel, Parallel, Send, Receive, InitChannel;
  FROM Terminal IMPORT Read, Write;

  VAR n: INTEGER; ch: CHAR; chan: Channel;

  PROCEDURE P;
    VAR k: INTEGER;
  BEGIN
    REPEAT Receive(chan, k) UNTIL k = 0
  END P;

  PROCEDURE Q;
  BEGIN
    REPEAT n := n-1; Send(chan, n) UNTIL n = 0
  END Q;

BEGIN n := 30000; InitChannel(chan);
  Write("["); Read(ch); Parallel(Q,P); Write("]")
END ChanTest.
```

Conclusions

Several schemes of multiprogramming have been presented. They are based on different

sets of primitives for synchronization and communication. Implementation reveals that despite seeming conceptual differences they are closely related. The presented programs provide an estimate for the required complexity and overhead in the primitive operations.

All implementations have been expressed exlusively in Modula-2 which displays the adequacy of this language as a system programming tool. Even details which pertain to the particular computer used (Lilith) are fully expressible with the language's low-level facilities, and their extent is very small (see definition module Coroutines).

The moral of the story is that it may be wise to refrain from including multiprogramming facilities in a language, unless concurrency plays a dominant role and syntactic conveniences may become sufficiently important. In most cases, a general-purpose language can be used, provided it offers adquate low-level facilities whose use can be properly encapsulated by an appropriate structuring concept.

References

1. E.W. Dijkstra. Cooperating sequential processes. In *Programming Languages*, F. Genuys (ed.), Academic Press, 1968.
2. C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17 (8), 549-557 (1974).
3. W.H. Kaubisch, R.H. Perrott, C.A.R. Hoare. Quasiparallel programming. *Software - Practice and Experience*, 6, 341-356 (1976).
4. N. Wirth. Modula: A programming language for modular multiprogramming. *Software - Practice and Experience*, 7 (1), 37-52 (1977).
5. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21 (8), 666-677 (1978).
6. J. Welsh, A. Lister, E. Salzman. A comparison of two notations for process communications. *Language Design and Programming Methodology*, 1, 225-254 (1980).
7. J. Welsh, A. Lister. A comparative study of task communication in Ada. *Software - Practice and Experience*, 11, 257-290 (1981).
8. N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.
9. M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall International, 1982.
10. R. Williamson, E. Horowitz. Concurrent communication and synchronization mechanisms. *Software - Practice and Experience*, 14 (2), 135-151 (1984).
11. Inmos Ltd. *Occam Programming Manual*. Prentice-Hall International, 1984.

Revisions and Amendments to Modula-2

N. Wirth, ETH Zürich, 24.5.84

On November 21, 1983, a meeting was held with participants from several firms who had implemented Modula-2. Numerous features and facilities were proposed for addition or correction. The following subset was agreed upon. These rules should be regarded as revisions of Modula-2. Future implementors are encouraged to comply with these revisions, and existing compilers should be adapted. Although any change in a language is subject to resentment, the number of changes adopted here is very small and, I believe, each one is a genuine improvement.

1. Restrictions and Clarifications

1.1 The types of a formal VAR-parameter and that of its corresponding actual parameter must be *identical* (i.e. not merely compatible). This rule is relaxed in the case of a formal parameter of type ADDRESS, which is also compatible with all pointer types, and in the case of the type WORD, where the compatible types are specified for each implementation.

1.2 The types of the expressions specifying the starting and limiting values of the control variable in a for statement must be *compatible* (i.e. not merely assignment compatible) with the type of the control variable.

1.3 A process initiated in a module at priority level n must not call a procedure declared in a module at priority level m < n. Calls of procedures declared without priority are allowed.

1.4. Pointer types can be exported from definition modules as opaque types. Opaque export of other types may be subject to implementation restrictions. Assignment and test for (in)equality are applicable to opaque types.

1.5 All modules imported to the main module are initialized *before* the importing module is initialized. If there exist circular references, the order of initialization is not defined.

1.6 If a module identifier is imported, this does *not* imply that the identifiers of objects of this module become visible. However, those which are exported in qualified mode can be accessed by prefixing them with the module identifier.

2. Changes

2.1 *All* objects declared in a definition module are exported. The explicit export list is discarded. The definition module may be regarded as the implementation module's separated and extended export list.

DefinitionModule = DEFINITION MODULE ident ";" {import} {definition} END ident "." .

2.2 The syntax of a variant record type declaration with missing tag field is changed from

FieldList = | CASE [ident ":"] qualident OF ...

to

FieldList = | CASE [ident] ":" qualident OF ...

The fact that the colon is always present makes it evident which part was omitted, if any.

2.3 The type PROCESS in module SYSTEM is deleted. Its place is taken by the type ADDRESS.

3. Extensions

3.1 The syntax of the case statement and the variant record declaration is changed from

```
case = CaseLabelList ":" StatementSequence.
variant = CaseLabelList ":" FieldListSequence.
```

to

```
case = [CaseLabelList ":" StatementSequence].
variant = [CaseLabelList ":" FieldListSequence].
```

The inclusion of the empty case and the empty variant allows the insertion of superfluous bars similar to the empty statement allowing the insertion of superfluous semicolons.

3.2 A string consisting of n characters is said to have *length* n. A string of length 1 is compatible with the type CHAR.

3.3 The syntax of the subrange type is changed from

```
SubrangeType = "[" ConstExpression ".." ConstExpression "]".
```

to

```
SubrangeType = [ident] "[" ConstExpression ".." ConstExpression "]".
```

The optional identifier allows to specify the base type of the subrange. Example: INTEGER [0 .. 99]

3.4 Elements of sets had been restricted to be constants. This restriction is now relaxed. The syntax of sets and factors changes to

```
ConstFactor = ... | ConstSet | ... .
ConstSet = [qualident] "{" [ConstElement {"," ConstElement}] "}".
ConstElement = ConstExpression ["..." ConstExpression].

factor = ... | set | ... .
set = [qualident] "{" [element {"," element}] "}".
element = expression ["..." expression].
```

3.5 The character "~" is a synonym for the symbol NOT.

3.6 The identifiers LONGCARD, LONGINT, and LONGREAL denote standard types (which may not be available on some implementations)..

3.7 The type ADDRESS is compatible with all pointer types and with either CARDINAL, or LONGCARD, or LONGINT. The interpretation of addresses as numbers depends on the implementation.

3.8 The new standard functions MIN and MAX take as argument any scalar type (including REAL). They stand for the type's minimal resp. maximal value.