18 CONCURRENT PROGRAMMING

It is the objective of this chapter to extend the Clang language and its implementation to do what its name suggests - handle simple problems in concurrent programming. It is quite likely that this is a field which is new to the reader, and so we shall begin by discussing some rudimentary concepts in concurrent programming. Our treatment of this is necessarily brief, and the reader would be well advised to consult one of the excellent specialist textbooks for more detail.

18.1 Fundamental concepts

A common way of introducing programming to novices is by the preparation of a recipe or algorithm for some simple human activity, such as making a cup of tea, or running a bath. In such introductions the aim is usually to stress the idea of **sequential** algorithms, where "one thing gets done at a time". Although this approach is probably familiar by now to most readers, on reflection it may be seen as a little perverse to try to twist all problem solving into this mould - indeed, that may be the very reason why some beginners find the sequential algorithm a difficult concept to grasp. Many human activities are better represented as a set of interacting processes, which are carried out in parallel. To take a simple example, a sequential algorithm for changing a flat tyre might be written

```
begin
  open boot
  take jack from boot
  take tools from boot
  remove hubcap
  loosen wheel nuts
  jack up car
  take spare from boot
  take off flat tyre
  put spare on
  lower jack
  tighten wheel nuts
  replace hubcap
  place flat tyre in boot
  place jack in boot
  place tools in boot
  close boot
end
```

but it might be difficult to convince a beginner that the order here was correct, especially if he or she were used to changing tyres with the aid of a friend, when the algorithm might be better expressed

```
begin
                     open boot
   take tools from boot and     take jack from boot
                   remove hubcap
                 loosen wheel nuts
   jack up car           and     take spare from boot
                 take off flat tyre
                    put spare on
   lower jack            and     place flat tyre in boot
   tighten wheel nuts   and     place jack in boot
   replace hubcap       and     place tools in boot
                     close boot
end
```

Here we have several examples of **concurrent processes**, which could in theory be undertaken by two almost autonomous processors - provided that they co-operate at crucial instants so as to keep

in step (for example, taking off the flat tyre and getting the spare wheel from the boot are both processes which must be completed before the next process can start, but it does not matter which is completed first).

We shall define a **sequential process** as a sequence of operations carried out one at a time. The precise definition of an operation will depend on the level of detail at which the process is described. A **concurrent program** contains a set of such processes executing in parallel.

There are two motivations for the study of concurrency in programming languages. Firstly, concurrent facilities may be directly exploited in systems where one has genuine multiple processors, and such systems are becoming ever more common as technology improves. Secondly, concurrent programming facilities may allow some kinds of programs to be designed and structured more naturally in terms of autonomous (but almost invariably interacting) processes, even if these are executed on a single processing device, where their execution will, at best, be interleaved in **real time**.

Concurrent multiprocessing of peripheral devices has been common for many years, as part of highly specialized operating system design. Because this usually has to be ultra efficient, it has tended to be the domain of the highly skilled assembly-level programmer. It is only comparatively recently that high-level languages have approached the problem of providing reliable, easily understood constructs for concurrent programming. The modern programmer should have at least some elementary knowledge of the constructs, and of the main problem areas which arise in concurrent programming.

18.2 Parallel processes, exclusion and synchronization

We shall introduce the notation

COBEGIN S_1 ; S_2 ; S_3 ; . . . S_n COEND

to denote that the statements or procedure calls S_k can be executed concurrently. At an abstract level the programmer need not be concerned with how concurrency might be implemented at the hardware level on a computer - all that need be assumed is that processes execute in a non-negative, finite (as opposed to infinite) time. Whether this execution truly takes place in parallel, or whether it is interleaved in time (as it would have to be if only one processor was available) is irrelevant.

To define the effect of a concurrent statement we must take into account the statements S_0 and S_{n+1} which precede and follow it in a given program. The piece of code

S_0 ; COBEGIN S_1 ; S_2 ; S_3 ; . . . S_n COEND; S_{n+1}

can be represented by the **precedence graph** of Figure 18.1.

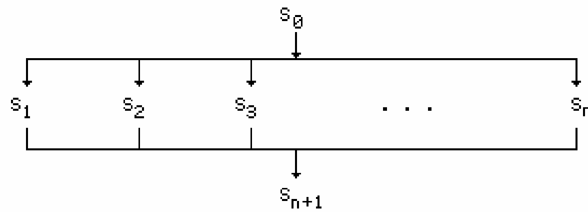

Figure 18.1 A precedence graph for a simple set of concurrent processes

Only after all the statements $S_1 \ldots S_n$ have been executed will S_{n+1} be executed. Similarly, the construction

```
S0;
COBEGIN
  S1;
  BEGIN
    S2; COBEGIN S3; S4 COEND; S5
  END;
  S6
COEND;
S7
```

can be represented by the precedence graph of Figure 18.2.

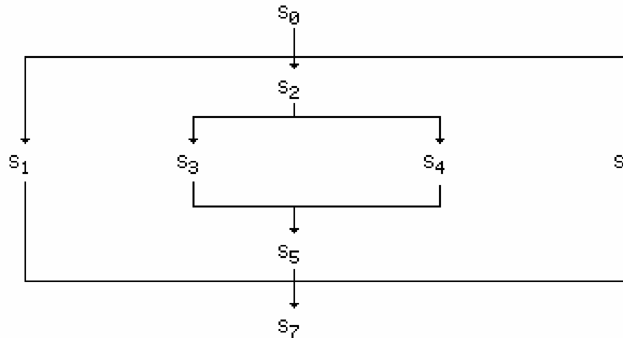

Figure 18.2 A precedence graph for a more complex set of processes

Although it is easy enough to depict code using the `COBEGIN ... COEND` construct in this way, we should observe that precedence graphs can be constructed which cannot be translated into this highly structured notation. An example of this appears in Figure 18.3.

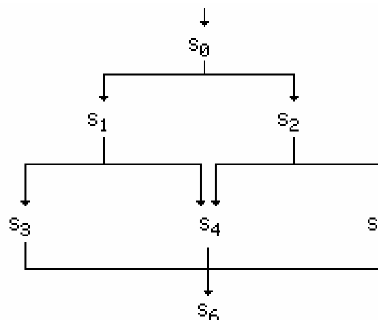

Figure 18.3 A precedence graph that cannot be expressed using COBEGIN ... COEND

As an example of the use of the `COBEGIN ... COEND` construct, we show a small program that will compute three simple summations simultaneously

```
PROGRAM Concurrent;
  VAR
    s1, s2, s3, i1, i2, i3;

  BEGIN
    COBEGIN
      BEGIN s1 := 0; FOR i1 := 1 TO 10 DO s1 := s1 + i1 END;
      BEGIN s2 := 0; FOR i2 := 1 TO 10 DO s2 := s2 + i2 END;
      BEGIN s3 := 0; FOR i3 := 1 TO 10 DO s3 := s3 + i3 END;
    COEND;
    WRITE(s1, s2, s3)
  END.
```

We may use this example to introduce two problem areas in concurrent programming that simply do not arise in sequential programming (at least, not that the high-level user can ever perceive). We have already intimated that we build concurrent programs out of sequential processes that must be regarded as executing simultaneously. A sequential process must be thought of as a sequential algorithm that operates on a data structure; the whole has the important property that it always gives the same result, regardless of how long it takes to execute. When sequential processes start to execute in parallel their time independence remains invariant only if their data structures remain truly private. If a process uses variables which other processes may simultaneously be changing, it is easy to see that the behaviour of the program as a whole may depend crucially on the relative speeds of each of its parts, and may become totally unpredictable.

In our example the three processes access totally private variables, so their concurrent composition is equivalent to any of the six possible ways in which they could have been arranged sequentially. As concurrent processes, however, the total execution time might be reduced. However, for the similar program below

```
PROGRAM Concurrent;
  VAR
    s1, s2, s3, i1, i2, i3;

  BEGIN
    COBEGIN
      BEGIN s2 := 0; FOR i1 := 1 TO 10 DO s1 := s1 + i2 END;
      BEGIN s3 := 0; FOR i2 := 1 TO 10 DO s2 := s2 + i3 END;
      BEGIN s1 := 0; FOR i3 := 1 TO 10 DO s3 := s3 + i1 END;
    COEND;
    Write(s1, s2, s3)
  END.
```

chaos would result, because we could never predict with certainty what was in any of the shared variables. At the same time the reader should appreciate that it must be possible to allow processes to access non-private data structures, otherwise concurrent processes could never exchange data and co-operate on tasks of mutual interest.

If one wishes to succeed in building large, reliable, concurrent programs, one will ideally want to use programming languages that cater specially for such problems, and are so designed that time dependent errors can be detected at compile-time, before they cause chaos - in effect the compiler must protect programmers from themselves. The simple `COBEGIN ... COEND` structure is inadequate as a reliable programming tool: it must be augmented with some restrictions on the forms of the statements which can be executed in parallel, and some method must be found of handling the following problems:

- **Communication** - processes must somehow be able to pass information from one to another.

- **Mutual exclusion** - processes must be guaranteed that they can access a critical region of code and/or a data structure in real-time without simultaneous interference from competing processes.

- **Synchronization** - two otherwise autonomous processes may have to be forced to wait in real-time for one another, or for some other event, and to signal one another when it is safe to proceed.

How best to handle these issues has been a matter for extensive research; suffice it to say that various models have been proposed and incorporated into modern languages such as Concurrent Pascal, Pascal-FC, Pascal-Plus, Modula, Edison, Concurrent Euclid, occam and Ada. Alarums and excursions: we propose to study a simple method, and to add it to Clang in this chapter.

We shall restrict the discussion to the use of shared memory for communication between processes (that is, processes communicate information through being able to access common areas of the same memory space, rather than by transmitting data down channels or other links).

The exclusion and synchronization problems, although fundamentally distinct, have a lot in common. A simple way of handling them is to use the concept of the **semaphore**, introduced by Dijkstra in 1968. Although there is a simple integer value associated with a semaphore `S`, it should really be thought of as a new type of variable, on which the only valid operations, beside the assignment of an initial associated integer value, are `P(S)` (from the Dutch *passeren*, meaning to pass) and `V(S)` (from the Dutch *vrijgeven*, meaning to release). In English these are often called **wait** and **signal**. The operations allow a process to cause itself to wait for a certain event, and then to be resumed when signalled by another process that the event has occurred. The simplest semantics of these operations are usually defined as follows:

> `P(S)` or `WAIT(S)` Wait until the value associated with `S` is positive, then subtract 1 from `S` and continue execution
>
> `V(S)` or `SIGNAL(S)` Add 1 to the value associated with `S`. This may allow a process that is waiting because it executed `P(S)` to continue.

Both `WAIT(S)` and `SIGNAL(S)` must be performed "indivisibly" - there can be no partial completion of the operation while something else is going on.

As an example of the use of semaphores to provide mutual exclusion (that is, protect a critical region), we give the following program, which also illustrates having two instances of the same process active at once.

```
PROGRAM Exclusion;
  VAR Shared, Semaphore;

  PROCEDURE Process (Limit);
    VAR Loop;
    BEGIN
      Loop := 1;
      WHILE Loop <= Limit DO
        BEGIN
          WAIT(Semaphore);
            Shared := Shared + 1;
          SIGNAL(Semaphore);
          Loop := Loop + 1;
        END
    END;

  BEGIN
    Semaphore := 1; Shared := 0;
    COBEGIN
      Process(4); Process(5+3)
    COEND;
    WRITE(Shared);
  END.
```

Each of the processes has its own private local loop counter `Loop`, but both increment the same

global variable `Shared`, access to which is controlled by the (shared) `Semaphore`. Notice that we are assuming that we can use a simple assignment to set an initial value for a semaphore, even though we have implied that it is not really a simple integer variable.

As an example of the use of semaphores to effect synchronization, we present a solution to a simple producer - consumer problem. The idea here is that one process produces items, and another consumes them, asynchronously. The items are passed through a distributor, who can only hold one item in stock at one time. This means that the producer may have to wait until the distributor is ready to accept an item, and the consumer may have to wait for the distributor to receive a consignment before an item can be supplied. An algorithm for doing this follows:

```
PROGRAM ProducerConsumer;
  VAR
    CanStore, CanTake;

  PROCEDURE Producer;
    BEGIN
      REPEAT
        ProduceItem;
        WAIT(CanStore); GiveToDistributor; SIGNAL(CanTake)
      FOREVER
    END;

  PROCEDURE Consumer;
    BEGIN
      REPEAT
        WAIT(CanTake); TakeFromDistributor; SIGNAL(CanStore);
        ConsumeItem
      FOREVER
    END;

  BEGIN
    CanStore := 1;  CanTake := 0;
    COBEGIN
      Producer; Consumer
    COEND
  END.
```

A problem which may not be immediately apparent is that communicating processes which have to synchronize, or ensure that they have exclusive access to a critical region, may become **deadlocked** when they all - perhaps erroneously - end up waiting on the same semaphore (or even different ones), with no process still active which can signal others. In the following variation on the above example this is quite obvious, but it is not always so simple to detect deadlock, even in quite simple programs.

```
PROGRAM ProducerConsumer;
  VAR
    CanStore, CanTake;

  PROCEDURE Producer (Quota);
    VAR I;
    BEGIN
      I := 1;
      WHILE I <= Quota DO
        BEGIN
          ProduceItem; I := I + 1;
          WAIT(CanStore); GiveToDistributor; SIGNAL(CanTake);
        END
    END;

  PROCEDURE Consumer (Demand);
    VAR I;
    BEGIN
      I := 1;
      WHILE I <= Demand DO
        BEGIN
          WAIT(CanTake); TakeFromDistributor; SIGNAL(CanStore);
          ConsumeItem; I := I + 1;
        END
    END;

  BEGIN
```

```
    CanStore := 1;  CanTake := 0;
    COBEGIN
      Producer(12); Consumer(5)
    COEND
  END.
```

Here the obvious outcome is that only the first five of the objects produced can be consumed - when `Consumer` finishes, `Producer` will find itself waiting forever for the `Distributor` to dispose of the sixth item.

In the next section we shall show how we might implement concurrency using `COBEGIN ... COEND`, and the `WAIT` and `SIGNAL` primitives, by making additions to our simple language like those suggested above. This is remarkably easy to do, so far as compilation is concerned. Concurrent execution of the programs so compiled is another matter, of course, but we shall suggest how an interpretive system can give the effect of simulating concurrent execution, using run-time support rather like that found in some real-time systems.

Exercises

18.1 One of the classic problems used to illustrate the use of semaphores is the so-called "bounded buffer" problem. This is an enhancement of the example used before, but where the distributor can store up to `Max` items at one time. In computer terms these are usually held in a circular buffer, stored in a linear array, and managed by using two indices, say `Head` and `Tail`. In terms of our simple language we should have something like

```
CONST
  Max = Size of Buffer;
VAR
  Buffer[Max-1], Head, Tail;
```

with `Head` and `Tail` both initially set to 1. Adding to the buffer is always done at the tail, and removing from the buffer is done from the head, along the lines of

```
add to buffer:
    Buffer[Tail] := Item;
    Tail := (Tail + 1) MOD Max;

remove from buffer:
    Item := Buffer[Head];
    Head := (Head + 1) MOD Max;
```

Devise a system where one process continually adds to the buffer, at the same time that a parallel process tries to empty it, with the restrictions that (a) the first process cannot add to the buffer if it is full (b) the second process cannot draw from the buffer if it is empty (c) the first process cannot add to the buffer while the second process draws from the buffer at exactly the same instant in real-time.

18.2 Another classic problem has become known as Conway's problem, after the person who first proposed it. Write a program to read the data from 10 column cards, and rewrite it in 15 column lines, with the following changes: after every card image an extra space character is appended, and every adjacent pair of asterisks is replaced by a single up-arrow ↑.

This is easily solved by a single sequential program, but may be solved (more naturally?) by three concurrent processes. One of these, `Input`, reads the cards and simply passes the characters (with the additional trailing space) through a finite buffer, say `InBuffer`, to a process `Squash` which simply looks for double asterisks and passes a stream of modified characters through a second finite

buffer, say `OutBuffer`, to a process `Output`, which extracts the characters from the second buffer and prints them in 15 column lines.

18.3 A semaphore-based system - syntax, semantics, and code generation

So as to provide a system with which the reader can experiment in concurrent programming, we shall add a few more permissible statements to our language, as described by the following EBNF:

```
Statement            =   [ CompoundStatement | Assignment | ProcedureCall
                          | IfStatement | WhileStatement
                          | WriteStatement | ReadStatement
                          | CobeginStatement | SemaphoreStatement ] .
CobeginStatement     =   "COBEGIN" ProcessCall { ";" ProcessCall } "COEND" .
ProcessCall          =   ProcIdentifier ActualParameters .
SemaphoreStatement   =   ( "WAIT" | "SIGNAL" ) "(" Variable ")" .
```

There is no real restriction in limiting the statements which may be processed concurrently to procedure calls, as any other statement may be packaged into a trivial procedure. However, for our simple implementation we shall limit the number of processes which can execute in parallel, and restrict the `COBEGIN ... COEND` construction to appearing in the main program block. These restrictions are really imposed by our pseudo-machine, which we shall augment as simply as possible by incorporating five new instructions, `CBG, FRK, CND, WGT` and `SIG`, with the following semantics:

`CBG N`	Prepare to instantiate a set of `N` concurrent processes
`FRK A`	Set up a suspended call to a level 1 procedure whose code commences at address `A`
`CND`	Suspend parent process and transfer control to one of the processes previously instantiated by `FRK`
`WGT`	Wait on the semaphore whose address is at top-of-stack
`SIG`	Signal the semaphore whose address is at top-of-stack.

The way in which parsing and code generation are accomplished can be understood with reference to the Cocol extract that follows:

```
CobeginStatement
=                           (. int processes = 0;
                               CGEN_labels start; .)
   "COBEGIN"                (. if (blockclass != TABLE_progs) SemError(215);
                               CGen->cobegin(start); .)
   ProcessCall              (. processes++; .)
   { WEAK ";" ProcessCall   (. processes++; .)
   }
   "COEND"                  (. CGen->coend(start, processes); .) .

ProcessCall
=                           (. TABLE_entries entry; TABLE_alfa name;
                               bool found; .)
   Ident<name>              (. Table->search(name, entry, found);
                               if (!found) SemError(202);
                               if (entry.idclass == TABLE_procs)
                                 CGen->markstack();
                               else { SemError(217); return; } .)
   ActualParameters<entry> (. CGen->forkprocess(entry.p.entrypoint); .) .

SemaphoreStatement
=                           (. bool wait; .)
   (   "WAIT"               (. wait = true; .)
     | "SIGNAL"             (. wait = false; .)
   )
   "(" Variable             (. if (wait) CGen->waitop(); else CGen->signalop(); .)
```

```
")" .
```

The reader should note that:

- Code associated with `COBEGIN` and `COEND` must be generated so that the run-time system can prepare to schedule the correct number of processes. A count of the processes is maintained in the parser for *CobeginStatement*, and later patched into the code generated by the call to the `cobegin` code generating routine when the call to `coend` is made.

- The `forkprocess` routine generates code that resembles a procedure call, but when this code is later executed it stops short of making the calls. In a more sophisticated code generator employing the use of an AST, as was discussed in section 17.6.2, the `PROCNODE` class would need to incorporate two code generating members, one for normal calls, and one for such suspended calls.

- The code generated by the `coend` routine signals to the run-time system that all the processes that have been initiated by the code generated by `forkprocess` may actually commence concurrent execution, and at the same time suspends operation of the main program. This will become clearer in the next section, where we discuss run-time support.

As before, the discussion will be clarified by presenting the code for an earlier example, which shows the use of semaphores to protect a critical region of code accessing a shared variable, the use of processes that use simple value parameters, and the invocation of more than one instance of the same process.

```
Clang 4.0 on 28/12/95 at 15:27:13

  0 PROGRAM Exclusion;
  0   VAR Shared, Semaphore;
  2
  2   PROCEDURE Process (Limit);
  2     VAR Loop;
  4     BEGIN
  4       Loop := 1;
 10       WHILE Loop <= Limit DO
 21         BEGIN
 21           WAIT(Semaphore);
 25             Shared := Shared + 1;
 36           SIGNAL(Semaphore);
 40           Loop := Loop + 1;
 51         END
 51     END;
 56
 56   BEGIN
 58     Semaphore := 1; Shared := 0;
 70     COBEGIN
 70       Process(4); Process(5+3)
 83     COEND;
 86     WRITE(Shared);
 92   END.
```

The code produced in the compilation of this program would read

```
  0 BRN       56    to start of main program
  2 DSP        1  BEGIN Process
  4 ADR  2    -6      address of Loop
  7 LIT        1      Constant 1
  9 STO             Loop := 1
 10 ADR  2    -6      address of Loop
 13 VAL               value of Loop
 14 ADR  2    -5      address of Limit
 17 VAL               value of Limit
 18 LEQ               Loop <= Limit ?
 19 BZE       53    WHILE Loop <= Limit DO
 21 ADR  1    -2        Address of Semaphore
 24 WGT               WAIT(Semaphore)
 25 ADR  1    -1        Address of Shared
```

```
   28 ADR  1    -1        Address of Shared
   31 VAL                 Value of Shared
   32 LIT        1        Constant 1
   34 ADD                 Value of Shared + 1
   35 STO               Shared := Shared + 1
   36 ADR  1    -2        Address of Semaphore
   39 SIG               SIGNAL(Semaphore)
   40 ADR  2    -6        Address of Loop
   43 ADR  2    -6        Address of Loop
   46 VAL                 Value of Loop
   47 LIT        1        Constant 1
   49 ADD                 Value of Loop + 1
   50 STO               Loop := Loop + 1
   51 BRN       10    END
   53 RET  2     0  END Process
   56 DSP        2  BEGIN Exclusion
   58 ADR  1    -2      Address of Semaphore
   61 LIT        1      Constant 1
   63 STO             Semaphore := 1
   64 ADR  1    -1      Address of Shared
   67 LIT        0      Constant 0
   69 STO             Shared := 0
   70 CBG        2    COBEGIN (2 processes)
   72 MST                 Mark stack
   73 LIT        4        Argument 4
   75 FRK        2      Process(4)
   77 MST                 Mark Stack
   78 LIT        5        Constant 5
   80 LIT        3        Constant 3
   82 ADD                 Argument 5 + 3
   83 FRK        2      Process(5+3)
   85 CND             COEND
   86 ADR  1    -1      Address of Shared
   89 VAL               Value of Shared
   90 PRN             WRITE(Shared)
   91 NLN             WriteLn
   92 HLT           END Exclusion
```

Exercises

18.3 If you study the above code carefully you might come up with the idea that it could be optimized by adding "level" and "offset" components to the `WGT` and `SIG` instructions. Is this a feasible proposition?

18.4 What possible outputs would you expect from the example program given here? What outputs could you expect if the semaphore were not used?

18.5 Is it not a better idea to introduce `PROCESS` as a reserved keyword, rather than just specifying a process as a `PROCEDURE`? Discuss arguments for and against this proposal, and try to implement it anyway.

18.4 Run-time implementation

We must now give some consideration to the problem of how one might execute a set of parallel processes or, in our case, interpret the stack machine code generated by the compiler. Perhaps this is a good point to comment that any sequential process forming part of a (generally) concurrent program may be in one of four states:

- *running* - instructions are being executed

- *ready* - suspended, waiting to be assigned to a processor

- *blocked* - suspended, waiting for some event to occur (such as I/O to be completed, or a signal on a semaphore)

- *deadlocked* - suspended, waiting for an event that will never occur (perhaps because of a failure in some other part of the system).

These states may conveniently be represented on a state diagram like that of Figure 18.4.

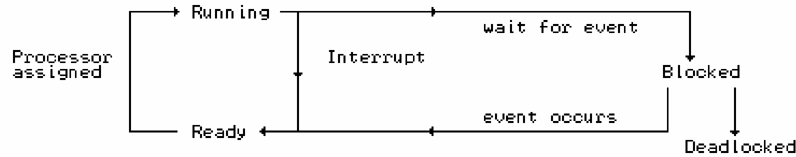

Figure 18.4 The possible execution states of one of a number of sequential processes

In practical implementations, concurrent behaviour is achieved in one of several ways. For example, processes can either

- share execution time on a single real processor (pseudo-concurrency);
- execute on a true multiprocessor system with shared memory, perhaps with each processor handling at most one process;
- execute on a true multiprocessor system without shared memory (distributed processing).

The implementation usually depends critically on a run-time support system or **kernel**, which may take one of a number of forms:

- a software structure, programmed as part of the application (as must be done in Modula-2);
- a standard software system, linked in with the object code (as usually done in Ada);
- a microcoded hardware structure (as typified by the Transputer).

Although the *logical behaviour* of a correct concurrent program will not - or should not - be dependent on the kernel, the *performance* of a real-time system may depend critically on the characteristics of the scheduling algorithms used in the kernel.

The shared memory, semaphore-based implementation upon which we have been focusing attention lends itself to the idea of multiplexing the processes on a single processor, or distributing them among a set of processors. Our interpreter, of course, really runs on one processor, although there is no reason why it should not emulate several real processors - with an interpretive approach any architecture can be modelled if one is prepared to sacrifice efficiency. What we shall do here is to emulate a system in which one controlling processor shares its time between several processes, allowing each process to execute for a few simulated fetch-execute cycles, before moving on to the next. This idea of **time-slicing** is very close to what occurs in some time-sharing systems in real life, with one major difference. Real systems are usually interrupt-driven by clock and peripheral controller devices, with hardware mechanisms controlling when some process switches occur, and software mechanisms controlling when others happen as a result of `WAIT` and `SIGNAL` operations on semaphores. On our toy system we shall simulate time-slicing by letting each active process

execute for a small random number of steps before control is passed to another one.

The simulated shared memory of the complete system will be divided up between the parallel processes while they are executing. This is not the place to enter into a study of sophisticated memory management techniques. Instead, what we shall do is to divide the memory which remains after the allocation to the main program stack frame, the program code, and the string pool, equally among each of the processes which have been initiated.

The processes are started by the main program; while they are executing, the main program is effectively dormant. When all the processes have run to completion, the main program is activated once more. While they are running, one can think of each as a separate program, each requiring its own stack memory, and each managing it in the way discussed previously. Each process conceptually has its own processor - or, more honestly, keeps track of its own set of processor registers, its own display, and so on. To accomplish this, we extend the data structures used by the interpreter, and in particular introduce a linked ring structure of so-called **process descriptors**, as follows:

```
const int STKMC_procmax = 10;          // Limit on concurrent processes
typedef int STKMC_procindex;           // Really 0 .. procmax

struct processrec {                    // Process descriptor records
  STKMC_address bp, mp, sp, pc;        // Shadow registers
  STKMC_procindex next;                // Ring pointer
  STKMC_procindex queue;               // Linked, waiting on semaphore
  bool ready;                          // Process ready flag
  STKMC_address stackmax, stackmin;    // Memory limits
  int display[STKMC_levmax];           // Display registers
};

processrec process[STKMC_procmax + 1]; // Ring of process descriptors
STKMC_procindex current, nexttorun;    // Process pointers
const int maxslice = 8;                // Maximum time slice
int slice;                             // Current time slice
```

The reader should note the following:

- The important `ready` field indicates whether the process is still active (`ready = true`), or has run to completion or been suspended on a semaphore (`ready = false`).

- Each process descriptor needs to maintain copies of the processor registers, so that when a **context switch** is done to allow another process to take charge of the single processor, the real CPU registers can be restored to the values they had when that process last executed.

- Similarly, each process descriptor maintains its own set of display registers, and its own limits on the memory that it is allowed to access for storing local variables and performing stack manipulations.

- The zeroth entry in the `process` array is used for the main program. As already mentioned, the other process descriptors are linked to form a circular ring, and the `next` and `queue` fields are used to connect these descriptors together.

As an example, consider the case where the main program has just launched four concurrent processes. The process descriptors would be linked as shown in Figure 18.5(a).

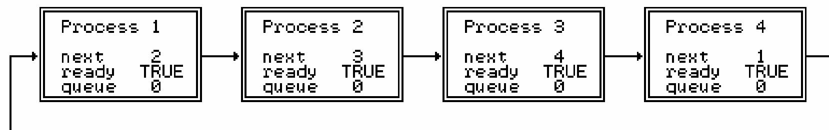

Figure 18.5(a) Process ring immediately after launching four concurrent processes

If process 2 is then forced to wait on a semaphore, the descriptor ring would change to the situation depicted in Figure 18.5(b).

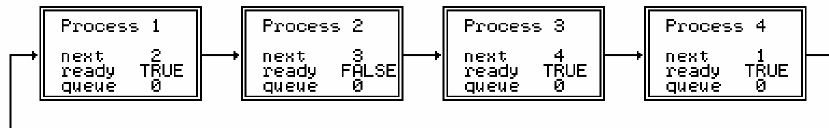

Figure 18.5(b) Process ring immediately after process 2 has been forced to wait

If process 3 runs to completion in the next time slice, the ring will then change to the situation depicted in Figure 18.5(c).

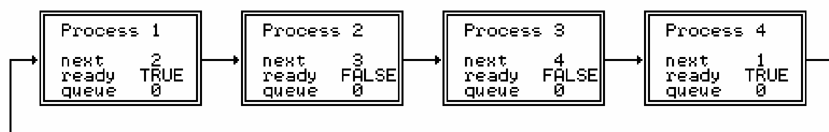

Figure 18.5(c) Process ring immediately after process 3 has run to completion

Finally, if process 4 then waits on the same semaphore, the ring changes to the situation depicted in Figure 18.5(d).

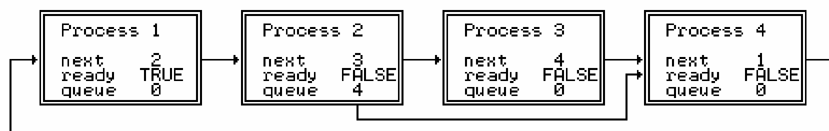

Figure 18.5(d) Process ring after process 4 has also been forced to wait

When a group of processes are all waiting on a common semaphore, their `ready` fields will all have been set to `false`, and their `queue` fields will have been used to link them in a FIFO queue, set up in real time as the `WAIT` operations were handled. We return to this point a little later on.

Initialization and emulation of the machine proceeds much as before, save that we now initialize a parent process (main program) process descriptor as well as the virtual processor:

```
process[0].queue = 0;            // Initialize parent process descriptor
process[0].ready = true;         // (memory limits and display)
process[0].stackmax = initsp;
process[0].stackmin = codelen;
for (int l = 0; l < STKMC_levmax; l++) process[0].display[l] = initsp;
cpu.sp = initsp;                 // Initialize stack pointer
cpu.bp = initsp;                 // Initialize registers
cpu.pc = initpc;                 // Initialize program counter
nexttorun = 0; nprocs = 0;       // Initialize emulator variables
slice = 0; ps = running;
do
```

```
    { current = nexttorun;          // Set active process descriptor pointer
      pcnow = cpu.pc;               // Save for tracing purposes
      if (unsigned(mem[cpu.pc]) > int(STKMC_nul)) ps = badop;
      else
      { cpu.ir = STKMC_opcodes(mem[cpu.pc]); cpu.pc++;  // Fetch
        if (tracing) trace(results, pcnow);
        switch (cpu.ir) {                               // Execute
                                                        // various cases
        }
      }
      if (nexttorun != 0) chooseprocess();
    } while (ps == running);
```

As we shall see later, `display[0]` is set to `initsp` for all processes, and will not change, for all processes are able to access the global variables of the main program. This is the most effective means we have of sharing data between processes.

Two pointers are used to index the array of process descriptors. `nexttorun` indicates the process that has most recently been *assigned* to the processor, and `current` indicates the process that is currently *running*. Each iteration of the fetch-execute cycle begins by copying `nexttorun` to `current`; some operations will alter the value of `nexttorun` to indicate that the real processor should be assigned to a new process. In particular, once the concurrent processes begin execution, `nexttorun` will no longer have the value of zero. The last part of the processing loop detects this as an indication that it may have to choose another process.

The algorithm for `chooseprocess` makes use of the variable `slice`. This is set to a small random number at the start of concurrent processing, and thereafter is decremented after each pseudo-machine instruction, or set to zero when a process is forced to wait, or terminates normally. When `slice` reaches zero, the process descriptor ring is searched cyclically (using the `next` pointer) so as to find a suitable process with which to continue for a further small (random) number of steps. Once found, a **context switch** is performed - the current CPU registers must be saved in the process descriptor, and must then be replaced by the values apposite to the process that is about to continue. The search and context switch are easily programmed:

```
  void STKMC::swapregisters(void)
  // Save current machine registers; restore from next process
  { process[current].bp = cpu.bp;   cpu.bp = process[nexttorun].bp;
    process[current].mp = cpu.mp;   cpu.mp = process[nexttorun].mp;
    process[current].sp = cpu.sp;   cpu.sp = process[nexttorun].sp;
    process[current].pc = cpu.pc;   cpu.pc = process[nexttorun].pc;
  }

  void STKMC::chooseprocess(void)
  // From current process, traverse ring of descriptors to next ready process
  { if (slice != 0) { slice--; return; }
    do { nexttorun = process[nexttorun].next; }
    while (!process[nexttorun].ready);
    if (nexttorun != current) swapregisters();
    slice = random(maxslice) + 3;
  }
```

We are here presuming that we have a suitable library function `random(limit)` for generating a sequence of random numbers, suitably scaled to lie in the range `0 <= random < Limit`.

There is a point of some subtlety here. If the search is instigated by virtue of one process being forced to wait on a semaphore or terminating normally, it must find *another* process to execute. There may be no such process, in which case a state of deadlock can be detected. However, if the search is instigated simply by virtue of a process reaching the end of its allotted time slice, then control can legitimately return to the same process if no other ready process can be found.

The mechanism of the `COBEGIN ... COEND` system is next to be discussed. As we have extended the language, processes are syntactically indistinguishable from procedures, and the code generation between the `COBEGIN` and `COEND` very nearly, but not quite, generates a set of procedure

calls. There is a fundamental difference, of course, in the way in which such procedure "calls" execute. After the COBEGIN, transfer of control must not pass immediately to the process procedures, but must remain with the main parent program until all child processes can be started together - the reason being that parameters may have to be set up, and this will have to be done in the environment of the parent.

For our stack machine, the code generated by the cobegin routine (in our simple machine, the CBG N sequence) is used by the kernel to decide on how to divide the remaining memory up among the imminent processes. This is achieved by the following code in the emulator:

```
case STKMC_cbg:
  if (mem[cpu.pc] > 0)                          // any processes?
  { partition = (cpu.sp - codelen) / mem[cpu.pc];  // divide rest of memory
    parentsp = cpu.sp;                          // for restoration by cnd
  }
  cpu.pc++;
  break;
```

The necessity of remembering the current value of cpu.sp will immediately become apparent after studying the interpretation of the FRK A instruction which is executed in place of the rather similar CAL L A so as to set up a process. Essentially what has to be achieved is the setting up of a complete activation record and process descriptor for a procedure, but without transferring control to this:

```
case STKMC_frk:
  nprocs++;                                     // one more process
  // first initialize the shadow CPU registers and display
  process[nprocs].bp = cpu.mp;                  // base pointer
  process[nprocs].mp = cpu.mp;                  // mark stack pointer
  process[nprocs].sp = cpu.sp;                  // stack pointer
  process[nprocs].pc = mem[cpu.pc];             // process entry point
  process[nprocs].display[0] =
    process[0].display[0];                      // for global access
  process[nprocs].display[1] = cpu.mp;          // for local access
  // now initialize frame header
  mem[process[nprocs].bp - 2] =
    process[0].display[1];                      // display copy
  mem[process[nprocs].bp - 3] = cpu.bp;         // dynamic link
  mem[process[nprocs].bp - 4] = processreturn;  // return address
  // descriptor house keeping
  process[nprocs].stackmax = cpu.mp;            // memory limits
  process[nprocs].stackmin = cpu.mp - partition;
  process[nprocs].ready = true;                 // ready to run
  process[nprocs].queue = 0;                    // clear semaphore queue
  process[nprocs].next = nprocs + 1;            // link to next descriptor
  cpu.sp = cpu.mp - partition;                  // bump parent sp below
  cpu.pc++;                                     // memory reserved for process
  break;
```

where the reader should note that:

- The return address for a process procedure is set to an artificial value (this might be zero, but any other "impossible" value would suffice). This can later be detected at procedure exit as an indication that the process is complete, and may be deactivated.

- The penultimate step involves resetting the stack pointer for the parent process so as to skip over the area in memory that is being reserved for the process workspace.

The mechanics of COEND are now easy: we merely deactivate the main program, close the descriptor ring, and choose one of the processes (at random) to continue execution. When all processes have run to completion their workspaces can, of course, all be reclaimed. Provision for doing this was made when we saved the value of cpu.sp as part of the action of the CBG N instruction; the saved value is restored to the process descriptor for the main program as part of the interpretation of the CND instruction:

```
case STKMC_cnd:
  if (nprocs > 0)                        // check for pathological case
  { process[nprocs].next = 1;            // close ring of descriptors
    nexttorun = random(nprocs) + 1;      // choose first process at random
    cpu.sp = parentsp;                   // restore parent stack pointer
  }
  break;
```

Processes, like procedures, terminate when they encounter a `RET` instruction. The interpretation requires slight modification from what we have seen previously, and may be understood with reference to the code below:

```
case STKMC_ret:
  process[current].display[mem[cpu.pc] - 1] = mem[cpu.bp - 2];
                                         // restore display
  cpu.sp = cpu.bp - mem[cpu.pc + 1];     // discard stack frame
  cpu.mp = mem[cpu.bp - 5];              // restore mark pointer
  cpu.pc = mem[cpu.bp - 4];              // get return address
  cpu.bp = mem[cpu.bp - 3];              // reset base pointer
  if (cpu.pc == processreturn)           // kill a concurrent process
  { nprocs--; slice = 0;                 // force choice of new process
    if (nprocs == 0)                     // must reactivate main program
    { nexttorun = 0; swapregisters(); }
    else                                 // complete this process only
    { chooseprocess();                   // may fail
      process[current].ready = false;
      if (current == nexttorun) ps = deadlock;
    }
  }
  break;
```

Much of this is as before, except that we must check for the artificial return address mentioned above. If this is detected, but uncompleted processes are known to exist, we reset the time slice, attempt to choose another process, switch context, deactivate the completed process, and only then check for deadlock. On the other hand, when all processes have been completed, we simply do a context switch back to the main program (`process[0]`).

The last point to be considered is that of implementing semaphore operations. This is a little subtle. The simplest semantic meaning for the `WAIT` and `SIGNAL` operations is probably

```
WAIT(S)       WHILE S < 0 DO (* nothing *) END; S := S - 1;

SIGNAL(S)     S := S + 1;
```

where, as we have remarked, the testing and incrementing must be be done as indivisible operations. The interpreter allows easy implementation of this otherwise rather awkward property, because the entire operation can be handled by one pseudo-operation (a `WGT` or `SIG` instruction).

However, the simple semantic interpretation above is probably never implemented, for it implies what is known as a **busy-wait** operation, where a processor is tied up cycling around wasting effort doing nothing. Implementations of semaphores prefer to deactivate the waiting process completely, possibly adding it to a queue of such processes, which may later be examined efficiently when a signal operation gives the all-clear for a process to continue. Although the semantics of `SIGNAL` do not require a queue to be formed, we have chosen to employ one here.

The `WAIT` and `SIGNAL` primitives can then be implemented in several ways. For example, `WAIT(Semaphore)` can be realized with an algorithm like

```
IF Semaphore.Count > 0
  THEN DEC(Semaphore.Count)
  ELSE set Slice to 0 and ChooseProcess
       Process[Current].Ready := FALSE
       add Process[Current] to Semaphore.Queue
       Process[Current].Queue := 0
END
```

provided that the matching `SIGNAL(Semaphore)` is realized by an algorithm like

```
IF Semaphore.Queue is empty
  THEN INC(Semaphore.Count)
  ELSE find which process should be Woken
       Process[Woken].Ready := TRUE
       set start of Semaphore.Queue to point to Process[Woken].Queue
END
```

The problem then arises of how to represent a semaphore variable. The first idea that might come to mind is to use something on the lines of a structure or record with two fields, but this would be awkward, as we should have to introduce further complications into the parser to treat variables of different sizes. We can retain simplicity by noting that we can use an integer to represent a semaphore if we allow negative values to act as `Queue` values and non-negative values to act as `Count` values. With this idea we simply modify the interpreter to read

```
case STKMC_wgt:
  if (current == 0) ps = badsem;
  else { cpu.sp++; wait(mem[cpu.sp - 1]); }
  break;
case STKMC_sig:
  if (current == 0) ps = badsem;
  else { cpu.sp++; signal(mem[cpu.sp - 1]); }
  break;
```

with `wait` and `signal` as routines private to the interpreter, defined as follows:

```
void STKMC::signal(STKMC_address semaddress)
{ if (mem[semaddress] >= 0)                  // do we need to waken a process?
  { mem[semaddress]++; return; }             // no - simply increment semaphore
  STKMC_procindex woken = -mem[semaddress]; // negate to find index
  mem[semaddress] = -process[woken].queue;  // bump queue pointer
  process[woken].queue = 0;                 // remove from queue
  process[woken].ready = true;              // and allow to be reactivated
}

void STKMC::wait(STKMC_address semaddress)
{ STKMC_procindex last, now;
  if (mem[semaddress] > 0)                  // do we need to suspend?
  { mem[semaddress]--; return; }            // no - simply decrement semaphore
  slice = 0; chooseprocess();               // choose the next process
  process[current].ready = false;           // and suspend this one
  if (current == nexttorun) { ps = deadlock; return; }
  now = -mem[semaddress];                   // look for end of semaphore queue
  while (now != 0) { last = now; now = process[now].queue; }
  if (mem[semaddress] == 0)
    mem[semaddress] = -current;             // first in queue
  else
    process[last].queue = current;          // place at end of existing queue
  process[current].queue = 0;               // and mark as the new end of queue
}
```

There are, as always, some subtleties to draw to the reader's attention:

- A check should be made to see that `WAIT` and `SIGNAL` are only attempted from within a concurrent process. Because of the way in which we have extended the language, with processes being lexically indistinguishable from other procedures, this cannot readily be detected at compile-time, but has to be done at run-time. (See also Exercise 18.5.)

- Although the semantic definition above also seems to imply that the value of a semaphore is always increased by a `SIGNAL` operation, we have chosen not to do this if a process is found waiting on that semaphore. This process, when awoken from its implied busy-wait loop, would simply decrement the semaphore anyway; there is no need to alter it twice.

- The semantics of `SIGNAL` do not require that a process which is allowed to proceed actually gain control of the processor immediately, and we have not implemented `signal` in this way.

Exercises

18.6 Add concurrent processing facilities to Topsy on the lines of those described here.

18.7 Introduce a call to a random number generator as part of Clang or Topsy (as a possibility for a *Factor*), which will allow you to write simple simulation programs.

18.8 Ben-Ari (1982) and Burns and Davies (1993) make use of a `REPEAT - FOREVER` construct. How easy is it to add this to our language? How useful is it on its own?

18.9 A multi-tasking system can easily devote a considerable part of its resources to process switching and housekeeping. Try to identify potential sources of inefficiency in our system, and eradicate as many as possible.

18.10 One problem with running programs on this system is that in general the sequence of interleaving the processes is unpredictable. While this makes for a useful simulation in many cases, it can be awkward to debug programs which behave in this way, especially with respect to I/O (where individual elements in a read or write list may be separated by elements from another list in another process). It is easy to use a programmer-defined semaphore to prevent this; can you also find a way of ensuring that process switching is suspended during I/O, perhaps requested by a compiler directive, such as `(*$S-*)`?

18.11 Is it difficult to allow concurrent processes to be initiated from within procedures and/or other processes, rather than from the main program only? How does this relate to Exercise 18.5?

18.12 Develop an emulation of a multiprocessor system. Rather than have only one processor, consider having an (emulated) processor for each process.

18.13 Remove the restriction on a fixed upper limit to the number of processes that can be accommodated, by making use of process descriptors that are allocated dynamically.

18.14 Our round-robin scheduler attempts to be fair by allocating processor time to each ready process in rotation. Develop a version that is unfair, in that a process will only relinquish control of the processor when it is forced to wait on a semaphore, or when it completes execution. Is this behaviour typical of any real-time systems in practice?

18.15 As an extension to Exercise 18.14, implement a means whereby a process can voluntarily suspend its own action and allow another process of the same or higher priority to assume charge of the processor, perhaps by means of a routine `SUSPEND`.

18.16 Do you suppose that when a process is signalled it should be given immediate access to the processor? What are the implications of allowing or disallowing this strategy? How could it be implemented in our system?

18.17 Replace the kernel with one in which semaphores do not have an associated queue. That is, when a `SIGNAL(S)` operation finds one or more processes waiting on `S`, simply choose one of these processes at random to make `ready` again.

18.18 Our idea of simply letting a programmer treat a semaphore as though it were an integer is

scarcely in the best traditions of strongly typed languages. How would you introduce a special semaphore type into Clang or Topsy (allow for arrays of semaphores), and how would you prevent programmers from tampering with them, while still allowing them to assign initial values to their `Count` fields? You might like to consider other restrictions on the use of semaphores, such as allowing initial assignment only within the parent process (main program), forbidding assignment of negative values, and restricting the way in which they can be used as parameters to procedures, functions or processes (you will need to think very carefully about this).

18.19 In our system, if one process executes a `READ` operation, the whole system will wait for this to be completed. Can you think of a way in which you can prevent this, for example by checking to see whether a key has been pressed, or by making use of real-time interrupts? As a rather challenging exercise, see if you can incorporate a mechanism into the interpreter to provide for so-called *asynchronous input*.

18.20 The idea of simulating time-slicing by allowing processes to execute for a small random number of steps has been found to be an excellent teaching tool (primarily because subtly wrong programs often show up faults very quickly, since the scheduler is essentially non-deterministic). However, real-time systems usually implement time-slicing by relying on interrupts from a real-time clock to force context switches at regular intervals. A Modula-2 implementation of an interpreter can readily be modified to work in this way, by making use of coroutines and the `IOTRANSFER` procedure. As a rather challenging exercise, implement such an interpreter. It is inexpedient to implement true time-slicing - pseudo-code operations (like `WGT` and `SIG`) should remain indivisible. A suggested strategy to adopt is one where a real clock interrupt sets a flag that the repetitive fetch-execute cycle of the emulator can acknowledge; furthermore, it might be advantageous to slow the real rate of interrupts down to compensate for the fact that an interpreter is far slower than a "real" computer would be.

Many kernels employ, not a ring of process descriptors, but one or more prioritized queues. One of these is the **ready queue**, whose nodes correspond to processes that are ready to execute. The process at the front of this queue is activated; when a context switch occurs the descriptor is either moved to another queue (when the process is forced to wait on a semaphore), is deallocated (when the process has finished executing), or is re- queued in the ready queue behind other processes of equal priority (if fair scheduling is employed on a round- robin basis). This is a method that allows for the concept of processes to be assigned relative priorities rather more easily than if one uses a ring structure. It also gives rise to a host of possibilities for redesigning the kernel and the language.

18.21 Develop a kernel in which the process descriptors for ready processes (all of the same priority) are linked in a simple ready queue. When the active process is forced to wait on a semaphore, transfer its descriptor to the appropriate semaphore queue; when the active process reaches the end of its time slice, transfer its descriptor to the end of the ready queue. Does this system have any advantages over the ring structure we have demonstrated in this section?

18.22 Extend the language and the implementation so that processes may be prioritized. When a context switch occurs, the scheduler always chooses the ready process with the highest priority (that is, the one at the front of the queue) as the one to execute next. There are various ways in which process priority might be set or changed. For example:

- Develop a system where process priorities are determined at the time the processes are spawned, and remain constant thereafter. This could be done by changing the syntax of the `COBEGIN ... COEND` structure:

```
CobeginStatement =  "COBEGIN" ProcessCall { ";" ProcessCall } "COEND" .
```

```
ProcessCall      = ProcIdentifier ActualParameters [ Priority ] .
Priority         = "[" Expression "]" .
```

Take care to ensure that the *Expression* is evaluated and stored in the correct context.

- Develop a system where processes may alter their priorities as they execute, say by calling on a routine `SETPRIORITY(Priority)`.

Pay particular attention to the way in which semaphore queues are manipulated. Should these be prioritized in the same way, or should they remain FIFO queues?

Further reading

Several texts provide descriptions of run-time mechanisms rather similar to the one discussed in this chapter.

In Ben-Ari's influential book *Principles of Concurrent Programming* (1982) may be found an interpreter for a language based on Pascal-S (Wirth, 1981). This implementation has inspired several others (including our own), and also formed the starting point for the Pascal-FC implementation described by Burns and Davies in their excellent and comprehensive book (1993). Burns and Davies also outline the implementation of the support for several other concurrent paradigms allowed by their language.

We should warn the reader that our treatment of concurrent programming, like that of so much else, has been rather dangerously superficial. He or she might do well to consult one or more of the excellent texts which have appeared on this subject in recent years. Besides those just mentioned, we can recommend the books by Burns and Welling (1989), and Bustard, Elder and Welsh (1988) and the survey paper by Andrews and Schneider (1983).