Computer Science 415.340

Operating systems

IMPLEMENTATION.

EXAMPLES OF SYSTEMS PROGRAMMING LANGUAGES

In this chapter we give examples of three languages designed for programming operating systems (or the broader class of real-time systems). This discussion is not intended as an exhaustive survey, but only to illustrate something of the range of possibilities, and in each case to emphasise some feature of programming which is required in an operating system, but not addressed in conventional languages.

ADA.

Ada is a member of the Pascal extended family, though greatly extended beyond Pascal itself. It was specifically designed for real-time systems, of which operating systems form a rather specialised subset. It was deliberately constructed from well known and well understood traditional techniques in the interests of reliability; as far as possible, new and untried techniques were not used. Ada was intended to do everything, and in consequence is very large and complex, both in the extent of the language itself and in the extent of the software which implements it. It has been strongly criticised because of its old-fashioned design and its complexity, and – like C – has come in for its share of ridicule[IMP8], but its defenders argue that the complexity is inevitable in a language which covers such a wide range, and the wide range is desirable so that compatibility can be guaranteed for any application area.

This example[IMP9] illustrates the use of the *rendezvous*, which is Ada's process synchronising technique.

10.4.3 Ada

Ada is a higher-level programming language, sponsored by the United States Department of Defense, designed under the leadership of Jean Ichbiah. The language can be used for conventional programming, as well as for special technical requirements, such as driving or monitoring various devices in real time. In this section, we are only concerned with those language constructs that are intended to provide a facility for writing concurrent programs. Central to this facility is the concept of the *task*, which is a program module that is executed asynchronously (a task can be viewed as a sequential process, as defined in Chapter 9). Tasks may communicate and synchronize their actions through the following:

- The **accept statement** is a combination of procedure calls and message transfer.
- The **select statement** is a non-deterministic control structure based on Dijkstra's guarded command construct.

We now briefly elaborate on these two language constructs.

Central to the communication facility is the **accept** statement, which has the following form. (Square brackets [ ] denote an optional part, while braces { } denote a repetition of zero or more times.)

```
accept <entry-name> [<formal
                parameter list>]
   [do <statements> end;]
```

The statements of an **accept** statement can be executed only if another task invokes the entry-name. Invoking an entry-name is syntactically the same as a procedure call. At this point, parameters are also passed. After the **end** statement has been reached, parameters may be passed back, and both tasks are free to continue. Either the calling task or the called task may be suspended until the other task is ready with its corresponding communication. Thus the facility serves both as a communication mechanism and a synchronization tool.

Choices among several entry calls is accomplished by the **select** statement, which is based on Dijkstra's guarded command concept. For brevity, we describe a restricted form of the **select** statement, with no *delay* and *terminate* statements. The **select** statement has the form:

```
select
   [when <boolean-expression>    ]
      <accept-statement>
      [<statements>]
   {or [when <boolean-expressions>
                    ]
      <accept-statement>}
      [<statements>]
   [else <statements>]
end select;
```

Execution of a **select** statement proceeds as follows:

1. All the boolean expressions appearing in the **select** statement are evaluated. Each **accept** statement whose corresponding boolean expression is evaluated to be true is tagged as open. An **accept** statement that is not preceded by a **when** clause is always tagged as open.

2. An open **accept** statement may be selected for execution only if another task has invoked an entry corresponding to that **accept** statement. If several open statements may be selected, an arbitrary one will be chosen for execution. If none can be selected and there is an **else** part, the **else** part is executed. If there is no **else** part, then the task waits until an open statement can be selected.

3. If no **accept** statement is open and there is an **else** part, the **else** part is executed. Otherwise an exception condition is raised.

The **accept** statement provides a task with a mechanism to wait for a predetermined event in another task. On the other hand, the **select** statement provides a task with a mechanism to wait for a set of events whose order cannot be predicted in advance.

These concepts can be illustrated with the bounded-buffer producer/consumer problem:

```
task body bounded-buffer is
   buffer: array [0..9] of item;
   in,out: integer;
   count: integer;
   in := 0;
   out := 0;
   count := 0;
begin
   loop
     select
       when count < 10
       accept insert (it: item)
         do buffer[in mod 10] :=
                it end;
       in := in + 1;
       count := count + 1;
     or when count > 0
       accept remove (it: out
                item)
         do it := buffer[out mod
                10] end;
       out := out + 1;
       count := count – 1;
     end select;
   end;
end.
```

The producer task puts an item *p* into the bounded-buffer by executing:

bounded-buffer.insert (*p*);

The consumer task gets an item *q* from the bounded-buffer BB by executing:

bounded-buffer.remove (*q*);

In contrast to CSP, we have complete symmetry between the producer and consumer tasks.

MODULA-2.

If Ada can be described as a member of the Pascal extended family, then Modula-2 is a member of the Pascal nuclear family, for its immediate parent (Modula) was developed from Pascal by Niklaus Wirth, the developer of Pascal itself[IMP10]. A significant aim of Modula-2 is to facilitate concurrent programming by "factoring out" the concurrency features into separate modules which can be called on by other, conventional, models for concurrency services when needed. A programme in Modula-2 runs as a set of modules operating as coroutines, with a prescribed discipline for switching between the processes.

The example[IMP11] below shows how a semaphore system can be implemented in Modula-2. (There's another Modula-2 example later, in *DEVICE CONTROL SOFTWARE*.)

As an example, the following is a simple process coordinator that implements a semaphore mechanism written in MODULA-2.

```
MODULE coordinator;
FROM SYSTEM IMPORT process,
                   newprocess,
                   transfer;
EXPORT sem, wait, signal, init;
CONST maxproc = 8;   {number of
                   processes}

TYPE sem = RECORD    {semaphore data
                   structure}
   c: 0..maxint;     {semaphore
                   count}
   q: 0..maxproc     {queue header}
   END;

VAR proclist: ARRAY [1..maxproc] OF
                   {process list}
   RECORD
   p: process;       {process
                   variable}
   s: (halted, free);
   1: 0..maxproc     {semaphore
                   queue link}
   END;

   cproc: 1..maxproc;
   coord: process;   {for return to
                   coordinator}

PROCEDURE init (VAR s: sem; val:
                   0..maxint);
{initialise semaphore}
   BEGIN
   s^.c := val;      {initialise
                   count}
   s^.q := 0         {no processes
                   halted}
   END;

PROCEDURE wait ( VAR s: sem);
   BEGIN
   IF s^.c > 0 THEN
      s^.c := s^.c - 1
   ELSE
      proclist[cproc].1 := s^.q;
      s^.q := cproc;  {add cproc to
                      sem queue}
      proclist[cproc].s := halted;
                      {mark cproc
                      halted}
                                                          tran
   END
   END;

PROCEDURE signal (VAR s: sem);
   BEGIN
   IF s^.q = 0 THEN  {no processes
                      halted}
      s^.c := s^.c + 1
   ELSE
      proclist [s^.q].s: = free;
                      {make one
                      process free}
      s^.q := proclist[s^.q].I
   END
   END;
{status}
   BEGIN
   FOR cproc := 1 TO maxproc DO
      newprocess ( ?, ?, ?,
                      proclist[cproc
{current process}     ].p);
      proclist[cproc].s := free :
      proclist[cproc].l := 0
      END;

   loop
      FOR cproc := 1 TO maxproc DO
      IF proclist[cproc].s = free
                      THEN
         transfer(coord,
                      proclist[cproc
                      ].p)
      END
   END
   END;
```

OCCAM.

Occam is a curious language developed for use with Transputer processors. It is derived from an earlier language called CSP, which stands for *Communicating Sequential Processes*, a rather precise description of its speciality. Transputers were designed specifically for use in multiprocessor systems, and were equipped with four high speed communications channels intended for interprocessor communication. CSP's provision for interprocess communication was well suited for this purpose, and occam is derived from CSP by emphasising simplicity and certain other useful features – note the last paragraph in the transcribed text.

The name "occam" reflects this concern for simplicity. William of Occam (around 1280 – 1349) apparently didn't propound the heuristic principle

nevertheless known as Occam's Razor, which is usually quoted in some such form as "entities should not be multiplied without justification" (or, if you want to show off, "entia non sunt multiplicanda praeter necessitatem"); being interpreted, that is an exhortation to keep arguments as simple as possible.

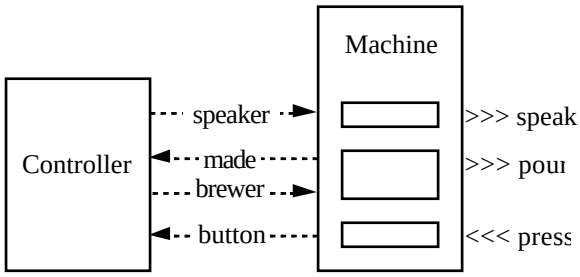
The language is unusual in that structure is indicated by indenting (not itself a syntactic feature in many languages), and by the convention that instructions appearing in the same "compound statement" are executed in parallel unless sequential execution has been explicitly requested by a SEQ instruction.

This example[IMP12] illustrates how a simple machine controller can be implemented in occam.

The use of occam is illustrated by the design of an automatic tea maker. This wakes you up in the morning with a message and offers a hot cup of tea. It is also a clock and will make tea at any other time, on request.

The tea maker has a number of units which interact with each other: the tea brewer which makes and pours the tea, a speech synthesiser for saying 'good morning' and telling the time, request buttons and an overall controller.

These units can be represented as a network:



In occam, each of the units is described by a process and each connection by a channel. The processes communicate by sending messages via the channels. A process can be constructed from smaller processes, as in the case of this machine which has a number of parts. Indeed the collection of processes is itself a process in occam, and could be part of some larger system.

This network is represented by defining the channels and the processes like this:

```
CHAN speaker, made, brewer,
                    button :
PAR
  ...       -- controller
  PAR       -- machine
    speech.synthesiser
    tea.brewer
    control.panel
```

Consider programming the controller. One of three things can happen. Firstly, it may receive a message on the button channel :

```
button ? request
  IF
    (request = tea.please) AND NOT
                  brewing
      PAR
        brewer ! make.tea
        brewing := TRUE
    request = time.please
      speaker ! say.time; NOW
```

This inputs a request from the button channel, and determines whether it is a request for tea, or a request for the time. A request for tea causes an output on the brewer channel telling the tea brewer to make tea, and the variable brewing is set to prevent further attempts to initiate tea making. A request for the time causes an output on the speaker channel to activate the speech synthesiser. The multiple output to the speaker channel has the same effect as two single outputs.

Secondly, the controller may receive a message from the tea brewer, telling it that tea is made:

```
made ? ANY
  SEQ
    speaker ! say.message;
                  tea.made
    brewer ! pour.tea
    brewing := FALSE
```

Finally, at daily intervals, the tea maker says 'good morning' and makes tea:

```
WAIT AFTER alarm.time
  SEQ
    alarm.time := alarm.time +
                  one.day
    speaker ! say.message;
                  good.morning
    IF
      NOT brewing
        PAR
          brewer ! make.tea
          brewing := TRUE
```

These individual program sections, each of which is a process, are combined into the complete controller process by declaring the local variables, and by using WHILE and ALT to enable the controller to perform whichever alternative is required :

```
VAR alarm.time, brewing :
SEQ
  alarm.time := 0
  brewing := FALSE
  WHILE TRUE
    ALT
      button ? request
        IF
          (request = tea.please)
                AND NOT brewing
            PAR
              brewer ! make.tea
              brewing := TRUE
          request = time.please
            speaker ! say.time;
              NOW
      made ? ANY
        SEQ
          speaker ! say.message;
                tea.made
          brewer ! pour.tea
          brewing := FALSE
    WAIT AFTER alarm.time
      SEQ
        alarm.time :=
              alarm.time +
              one.day
        speaker ! say.message;
              good.morning
        IF
          NOT brewing
            PAR
              brewer !
              make.tea
                brewing := TRUE
```

15. Summary.

There are many applications, especially those in which high performance or a fast response to external events is required, which can be simplified by the use of a number of communicating processors, where each processor deals with a part of the problem. Even when only one processor is used, many applications are most easily structured in terms of a number of concurrent operations.

In the future systems will be developed which involve large numbers of communicating processors. Occam is designed for implementing them, and for implementing applications languages which can exploit concurrency in their implementation.

Occam is intended to be the smallest language which is adequate for its purpose; however, suggestions for further simplification would be welcome.

REFERENCES.

IMP8 : H.G. Baker : "I have a feeling we're not in Emerald City anymore", *Sigplan Notices* **32#4**, 22-26 (April, 1997)

IMP9 : J.L. Peterson, A. Silberschatz : *Operating System Concepts* (Addison-Wesley, 2nd Edition, 1985)

IMP10 : N. Wirth : "Towards a discipline of real-time programming", *Comm.ACM* **20**, 577 (1977)

IMP11 : C.J. Theaker, G.R. Brookes : *A Practical Course on Operating Systems* (Macmillan, 1983)

IMP12 : D. May : "OCCAM", *Sigplan Notices* **18#4**, 69 (April 1983)