PROGRAMMING LANGUAGES FOR MULTIPROCESSOR SYSTEMS

R.H. PERROTT

On leave from :
Department of Computer Science
Queens's University
Belfast
N.Ireland

PS Division
CERN
CH-1211 Geneva 23
Switzerland

Abstract

High level languages for multiprocessor systems are required to provide features which control the concurrent activities (called processes) which can occur in an application program. Specifically the activities which need to be regulated arise from mutual exclusion situations where it is required that only one process can access a critical resource at any time without interruption and synchronisation situations where processes wish to co-operate to their mutual benefit.

To illustrate these problems this paper considers a solution to the bounded buffer problem in two different concurrent programming languages, viz, Pascal Plus and Ada. These languages represent the two different approaches which have been used to regulate concurrent activities in a programming language. Pascal Plus, like Concurrent Pascal and Modula, uses a monitor structure while Ada uses message passing primitives.

The solutions indicate the different design techniques which must be used depending on which synchronisation technique is employed.

Keywords

Concurrency: Mutual exclusion: Synchronisation: Ada.

1. INTRODUCTION

During the 1970's it was the design and implementation of operating systems which led the research into techniques to regulate concurrent activities. At the same time the price of hardware components was decreasing enabling the construction of truly parallel configurations, such as multiprocesser and distributed systems. As these systems became more widely available it was necessary to introduce concurrent programming languages to enable applications other than operating systems to be programmed.

The term process is used to describe a sequence of program instructions that can be performed in parallel with other program instructions. The point at which a processor is withdrawn from one process and given to another is dependent on the progress of the processes and the algorithm used to assign the processor(s). The simple and well-defined processor allocation strategy of a sequential system is replaced in order to achieve greater processor utilisation. The nett effect is that processes are independent and thus capable of interacting in a time-dependent manner. The series of states which the system passes through is not necessarily identical when the same batch of programs is presented with the same data for execution.

Thus in the concurrent programming environment a programmer requires not only program and data structures similar to those required in a sequential programming environment but also tools to control the interaction of the processes - processes which are proceeding at fixed but unknown rates.

The situations in which the processes interact can be divided into two categories. The first occurs whenever processes wish to update a shared variable or a resource at the same time (or in an interleaved fashion).

For example, when one process wishes to use a resource which is currently reserved by another process. A process must be able to carry out such actions without interference from the other processes; this is described as **mututal exclusion.** The second category occurs when processes are co-operating on some task, they must be correctly interleaved in time. For example, when one process requires a result not yet produced by another process. The processes are communicating or scheduling one another and are now aware of each other's existence and purpose; this is described as **process synchronisation.**

In this paper two concurrent programming languages, namely Ada (Ada, 1983) and Pascal Plus (Welsh and Bustard, 1979) are considered. These languages are chosen as being representative of the two main synchronisation techniques that have evolved; message passing primitives and monitors with condition variables. Only the concurrent features of each language are described in detail and these are then illustrated by means of the bounded buffer problem. The solutions illustrate the main considerations which must be addressed in a concurrent programming environment as well as enabling a comparison of the two languages to be made.

2. PASCAL PLUS

Pascal Plus, as its name implies, is an extension of Pascal. The extensions are as follows:

i) the envelope structure which is an aid to program modularisation and data abstraction;

ii) the process, monitor and condition structures which provide a means of representing parallel processes and controlling any subsequent interaction.

Processes are used to identify independent actions which may take place in parallel. After a process has been defined, instances of it can be declared. Once activated the processes proceed conceptually in parallel.

A process is defined as a block with a suitable heading which may include parameters. For example :

```
process  PRODUCER;
  (*local data definitions and declarations*)
begin
  (*statements of PRODUCER*)
end ; (*PRODUCER*)
instance
    BEE1,BEE2 : PRODUCER;
```

The last statement activates two processes which may subsequently run in parallel.

One solution to the mutual exclusion problem is to gather together any variables which are shared and the operations which can be performed on them; such a construct is called a monitor or secretary (Brinch Hansen, 1973; Dijkstra, 1972; Hoare, 1974). The monitor therefore collects all the critical pieces of code into a single structure and only one process can have access to this structure at any time. Such a language construct takes the form :

```
monitor   MONITORNAME;
  (*declaration of local data*)
  procedure   PROCNAME (parameter list);
     begin   (*procedure body*)   end ;
  (*declaration of other local procedures*)
begin
  (*initialisation of local data*)
end ;
```

On declaration the monitor initialises its local data and all subsequent calls use the values of the local variables obtained on completion of the previous call. To invoke a monitor procedure the monitor name and the required procedure and its parameters are specified as

```
MONITORNAME.PROCNAME (actual parameters);
```

If a process enters a monitor it may have to be suspended pending the action of another process; this is achieved by means of CONDITION queues. The user can declare these queues in the form

```
instance  FULL:CONDITION;
```

To suspend itself on a CONDITION queue a process performs a WAIT operation expressed as

```
FULL.WAIT
```

Such an action deactivates the process and appends it to the queue identified by FULL. It also releases the exclusion on the monitor, otherwise other processes would be prevented from entering the monitor.

To release a process from a queue another process performs a SIGNAL operation, indicating to the signalled process that the reason for its delay no longer holds. This is expressed as

```
FULL.SIGNAL
```

Control is immediately passed to the process at the head of the FULL queue. The signalling process is delayed until the signalled process releases the exclusion of the monitor; a SIGNAL operation has no effect if there are no processes waiting. This form of enforced politeness is necessary in case the condition for resuming a delayed process is subsequently changed by the signalling process or another process intervening.

To illustrate the use of a monitor with condition variables consider the situation where several processes (producers) wish to communicate a series of items to other processes (consumers). This can be achieved by a buffer of a finite capacity into which the producers deposit items and from which the consumers remove items when ready. The processes must be synchronised in such a way that the producers will not produce when the buffer is full and the consumers will not consume when the buffer is empty. This is known as the bounded buffer problem.

The necessary process synchronisation can be achieved by associating a queue with each possible waiting condition, namely a full or an empty buffer. A producer should perform a SIGNAL operation on the appropriate queue after it has placed an item in the buffer. A consumer should join this same queue if the buffer is empty. Another queue is required for the situation when the buffer is full. In this case the producer should wait and the consumer should signal.

By this means the processes can co-operate to their mutual benefit as shown in the following Pascal Plus program fragment.

```
type MESSAGE = definition;
monitor BOUNDEDBUFFER;
 var BUFFER : array[0..(N-1)] of MESSAGE;
      POINTER : 0..N-1;
      COUNT : 0..N;  (*number of items in buffer*)
 instance EMPTY, FULL : CONDITION;

procedure *DEPOSIT (ITEM : MESSAGE);
    begin
       if COUNT = N then FULL.WAIT;
       BUFFER [(POINTER + COUNT) mod N] := ITEM;
       COUNT := COUNT + 1;
       EMPTY.SIGNAL
    end ;  (*deposit*)

procedure *REMOVE ( var  ITEM : MESSAGE);
    begin
        if COUNT = 0 then EMPTY.WAIT;
        ITEM := BUFFER [POINTER];
        COUNT := COUNT - 1;
        POINTER := (POINTER + 1) mod N;
        FULL.SIGNAL
    end ;(*remove*)

begin
   COUNT := 0; POINTER := 0
end . (*bounded buffer*)
```

The procedures DEPOSIT and REMOVE are starred indicating that they may be called from outside the monitor. The value of the variable COUNT determines whether a process should or should not be delayed.

An instance of this monitor can then be declared and the starred procedures called as shown :

```
instance BUFFER1 : BOUNDEDBUFFER;

process  PRODUCER;
var ITEM : MESSAGE;
begin
    repeat
        (*produce item*)
        BUFFER1.DEPOSIT (ITEM);
    until FINISHED
end ;(*producer*)

process CONSUMER;
var ITEM : MESSAGE;
begin
    repeat
        BUFFER1.REMOVE (ITEM);
        (*consume item*)
    until FINISHED
end ; (*consumer*)

instance
    P : array [1..X] of PRODUCER;
    C : array [1..Y] of CONSUMER;
    (* X producers and Y consumers are activated*)
begin
end.
```

3. ADA

Ada was designed for the U.S. Department of Defense with three main objectives :

i) a recognition of the importance of program reliability and maintenance;

ii) a concern for programming as a human activity;

iii) efficiency.

To realise these objectives Ada has insisted on the following :

a) readability,

b) strong typing,

c) programming in the large features,

d) exception handling,

e) data abstraction,

f) generic units,

g) tasking.

In Ada the work task rather than process is used to indicate a sequence of actions. Every task is written in the declarative part of some enclosing program unit, which is called its parent. A program unit is either a subprogram, package or task.

The definition of a task consists of two parts, its specification and its body. The specification part is the interface presented to other tasks and may contain entry specifications which are a list of the services provided by the task. The task body contains the sequence of statements to be

executed when each of the services is requested; it represents the dynamic behaviour of the task. For example, -- indicates a comment in Ada,

```
task PRODUCER is
   -- specification
end ;
task body PRODUCER is
   -- body
end PRODUCER;
begin -- active here
   -- parent
end ; -- terminates here
```

The task automatically becomes active, its body is executed, when the **begin** of the parent unit is reached. The task then executes in parallel with the statements of its parent, i.e. those statements between the **begin** and **end** . The task terminates normally by reaching the final **end** .

The program unit in which a task (or tasks) is declared cannot be left until all tasks in that unit have terminated. If several tasks are required with similar properties it is possible to define a task type and then to declare as many instances as are necessary. For example :

```
task type PRODUCER_TASK is
    -- specification
end ;
task body PRODUCER_TASK is
  -- body
end PRODUCER_TASK;
```

defines a type called PRODUCER_TASK which is a task and

```
BEE2, BEE1: PRODUCER_TASK;
```

declares two instances of the type PRODUCER_TASK. These tasks will be activated at the **begin** of the parent unit in which they are declared.

In Ada tasks can be regarded as either active or passive. Passive tasks have the function of providing some service (or services) which are used by other active tasks. An example of a passive and several active tasks occurs in the bounded buffer problem. In Ada the buffer itself must be described as a task - a passive task which will accept calls from the other active tasks, the producers and consumers.

The communication method used is that one task calls a procedure defined in another task and the parameter list is used to provide for the transfer of data.

The task specification is used to declare the procedures which other tasks can call, that is, the services available from this task. For this reason they are referred to as entry procedures. For example, the specification of a task BOUNDED_BUFFER which allows items to be deposited and removed from a buffer could have its specification defined as

```
task  BOUNDED_BUFFER  is
  entry  DEPOSIT (ITEM : in MESSAGE);
  entry  REMOVE (ITEM : out MESSAGE);
end ;
```

indicating that it will accept calls to the procedures DEPOSIT and REMOVE from other tasks. Thus the specification contains a list of entry procedures which are the parts of a task which are available to other tasks to call and thus use. If the task provides no services then it requires no entry procedures in its specification. In the body of the task BOUNDED_BUFFER more details of the entry procedures must be given in the following form :

```
accept  DEPOSIT (ITEM : in MESSAGE)  do
   -- statements
end ;
-- statements

accept  REMOVE (ITEM : out  MESSAGE) do
   -- statements
end ;
```

The **accept** statement must use the same identifier name and a list of parameters of the same number and type as appeared in the specification part. The statements between the **do** and **end** perform the service associated with the entry procedure. To use any of the entry procedures another task must make a call of the form

```
BOUNDED_BUFFER.DEPOSIT (ITEM);
```

The task name qualified with an entry procedure name and a list of actual parameters.

Thus the structure of a solution for the bounded buffer problem takes the following form :

```
task BOUNDED_BUFFER is
  entry DEPOSIT (ITEM : in MESSAGE);
  entry REMOVE (ITEM : out MESSAGE);
end BOUNDED_BUFFER;
task BOUNDED_BUFFER is
  -- depends on the representation of the buffer
begin
  -- statements
  accept DEPOSIT (ITEM : in MESSAGE) do
  -- statements
  end ;
end BOUNDED_BUFFER;

task type PRODUCER is
  -- no entry procedures required
end ;
task body PRODUCER is
  ITEM : MESSAGE;
begin
  loop
     -- produce item
     BOUNDED_BUFFER.DEPOSIT (ITEM);
     exit when FINISHED;
  end loop ;
end PRODUCER;

task type CONSUMER is
  -- no entry procedures required
end ;
task body CONSUMER is
   ITEM : MESSAGE;
begin
  loop
    BOUNDED_BUFFER.REMOVE (ITEM);
    -- consume item
    exit when FINISHED;
  end loop ;
end CONSUMER;
```

Several instances of these task types can be declared, for example:

```
P : array (1..X) of PRODUCER;
C : array (1..Y) of CONSUMER;
```

so that the X producers, the Y consumers and the BOUNDED_BUFFER task are all operating independently and in parallel.

The question now is what happens when they wish to communicate. The information accepted by the BOUNDED_BUFFER task is that of a deposit or removal of an item as indicated in the specification part. Thus the part of the task body which will receive the information is the statements following the **accept** . Another task requests the information by calling one of the relevant entry procedures of the task.

Whichever task reaches its communication statement first, that is, the **accept** or **call** statement, waits for the other task, when both tasks are ready to communicate we have a rendezvous. At the rendezvous the parameters are transferred and the body of the **accept** statement executed. The calling task is held up and can only proceed when the execution of the **accept** statement has been completed. The two tasks then proceed independently. Thus the synchronisation of the tasks is implicit.

The BOUNDED_BUFFER task does not know which of the consumer tasks has called it.

Associated with each entry procedure there is a single queue of waiting tasks which are processed on a first come first served basis.

Non-determinism has been introduced by means of the **select** statement. The **select** statement enables a choice among several entry calls to be specified. For example, the BOUNDED_BUFFER task is not programmed to

accept producer calls before consumer calls, the order of selection is non-deterministic and programmed as

```
select
   accept DEPOSIT (ITEM : in MESSAGE) do
      -- statements
   end ;
or
   accept REMOVE (ITEM : out MESSAGE) do
      -- statements
   end ;
end select ;
```

If neither of the entry procedures has been called the BOUNDED_BUFFER task waits; if one of them has been called it is executed; if both have been called then either one is selected and executed. It is not known which call will be selected, the choice is at random and the programmer cannot rely on the selection algorithm used.

A conditional execution of a **select** alternative can be introduced by means of a **when** statement which specifies what is called a guard. For example:

```
when COUNT< N=>
   accept DEPOSIT (ITEM : in MESSAGE) do
      -- statements
   end ;
```

will ensure that the value of COUNT is less than N before the **accept** statement is activated. The guards are evaluated at the beginning of the **select** statement, if an alternative of the **select** statement does not have a guard then it is regarded as being true. One of the true

guards is selected at random for execution. If all the guards evaluate to false an error is reported. It is possible for a **select** statement to have an else part which will be executed if other alternatives are false.

A solution for the bounded buffer problem is as follows

```
task BOUNDED_BUFFER is
  entry DEPOSIT (ITEM : in MESSAGE);
  entry REMOVE (ITEM : out MESSAGE);
end BOUNDED_BUFFER;

task body BOUNDED_BUFFER is
   BUFFER : array (INTEGER range 0..N-1) of MESSAGE;
   POINTER : INTEGER range 0..N-1 :=0; -- initially 0
   COUNT : INTEGER range 0..N :=0;
begin
   loop
      select
         when COUNT< N=>
            accept DEPOSIT (ITEM : in MESSAGE) do
              BUFFER ((POINTER + COUNT) mod N) :=ITEM;
            end ;
         COUNT := COUNT + 1;
      or
         when COUNT > 0 =>
            accept REMOVE (ITEM : out MESSAGE) do
              ITEM := BUFFER (POINTER);
            end ;
         POINTER := (POINTER + 1) mod N;
         COUNT := COUNT - 1;
      end select ;
   end loop ;
end BOUNDED_BUFFER;
```

The accept statements are the critical sections and the calling task is delayed while an **accept** statement is executed. However, the updating of the variables POINTER and COUNT is not part of the critical sections which was the case in the monitor solution.

4. CONCLUSION

The Pascal Plus and Ada solutions to the bounded buffer problem illustrate the essential differences between these two synchronisation methods. In Ada the transfer of data is direct and synchronised while in Pascal Plus it is through a passive abstract data structure. These two different communication techniques require different program design methods.

In Ada the buffer is represented as a task with the synchronisation specified in terms of **entry, select** and **accept** statements while in Pascal Plus the CONDITION, WAIT and SIGNAL primitives are used to control the buffer. In both cases the buffer is accessed in a mutually exclusive manner. In Ada the tidying up operations, for example the adjustment of the value of the variable COUNT can be performed without delaying the calling process.

5. REFERENCES

Ada 1983, Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A). United States Department of Defense, Washington D.C.

Brinch Hansen, P., 1973: Operating Systems Principles. Prentice-Hall, New Jersey.

Dijkstra, E.W., 1972: Hierarchical Ordering of Sequential Processes in Operating Systems Techniques. (Eds.) C.A.R. Hoare and R.H. Perrott, Academic Press, London, pp.72-79.

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

Welsh, J., and Bustard,D.W., 1979: Pascal Plus - Another Language for Modular Multiprogramming. Software Practice and experience, 9, 947-957.