Comput. Lang. Vol. 15, No. 3, pp. 153–162, 1990
Printed in Great Britain.

0096-0551/90 $3.00 + 0.00

THE NOTION OF PRIORITY IN REAL-TIME PROGRAMMING LANGUAGES

A. BURNS[1] and A. J. WELLINGS[2]

[1]Department of Computing, University of Bradford, Bradford, West Yorkshire BD7 1DP, U.K.
[2]Department of Computer Science, University of York, Heslington, York YO1 5DD, U.K.

(*Received* 19 *September* 1989; *revision received* 15 *December* 1989)

Abstract—Embedded systems often have to operate within hard real-time constraints. Periodic and aperiodic processes must be scheduled to meet specified deadlines, the failure to do so being seen as an error condition. Priority is one means of representing scheduling information in a concurrent real-time programming language. Following the introduction and criticism of the facilities provided by the programming language Ada, the requirements for deadline scheduling and resource scheduling using priorities are introduced. Unfortunately these requirements, are, in some important particulars, contradictory. These contradictions are investigated and methods of resolving them are proposed.

Real-time programming languages Priority Deadlines

1. INTRODUCTION

In the course of the last two decades considerable research and development effort have gone into the design and implementation of programming languages for concurrent real-time systems. Much of this activity has focussed on the provision of appropriate abstractions for the programming of such systems. The traditional approach to constructing real-time systems is to use low-level language features, and *ad hoc* timing techniques. It is reasonble to assume that improvements in the engineering of dependable real-time systems can be made through the use of well designed and expressive high level languages [1].

Real-time systems often have to operate within hard real-time constraints, which arise in two forms. Periodic processes† typically sample data or execute a control loop and have explicit deadlines that must be met. Aperiodic or sporadic processes arise from asynchronous events outside the embedded computer (for example an alarm signal). These processes have specified response times associated with them. Periodic and aperiodic processes must be scheduled to meet specified deadlines; the failure to do so is seen as a failure of the system to meet its requirements specification. The run-time scheduling mechanisms must be able to guarantee that all critical deadlines are met even if this is at the expense of missing less important deadlines.

The existence of hard real-time constraints introduces significant problems for the designers of programming languages. Although programs in these languages have many other functions—for example maximising resource utilisation or preventing process starvation—if a language lacks the expressive power to deal with deadline scheduling it may well be considered inadequate for the hard real-time programming domain.

The Ada programming language was specifically developed for real-time applications; and yet even this language has a number of well documented difficulties (see Section 3). Changes to Ada have been demanded and there is currently a proposal to come forward with a revision of the language [2]. The details of this, however, have not yet been agreed.

The purpose of this paper is to address the notion of priority as it is used in real-time programming languages. In Section 2 it is noted that most modern real-time languages have concurrent programming facilities and that these facilities introduce non-determinancy. It is to eliminate some non-determinancy (i.e. to give more predictable systems) that priority is used. After a discussion of the specific problems of Ada, the two ways in which the notion of priority is used in languages are described. In Section 4, the role of priority in deadline scheduling is addressed;

†Although we view the terms task and process as synonyms, throughout this paper we will use the term process to be a language independent description of a concurrent activity. We will use the term task to be the Ada unit of concurrency.

in the following section its alternative use in general resource scheduling is considered. Finally, in Section 6, the conflicting use of priority is discussed, and a uniform approach is considered.

2. MODELS OF CONCURRENCY

Earlier real-time languages were based on shared variable communication [1], together with semaphore or monitor-based synchronisation; for example the Modula programming language [3]. Modern languages have incorporated message-based models (e.g. Ada and occam2) and use a synchronised rendezvous for interprocess data communication.

Both shared variable and message-based models require a run-time scheduler to provide process management. The design of a scheduler can be described at three levels; its specification (i.e. the scheduling algorithm), its policy (i.e. how the specification is implemented) and its mechanisms (i.e. the kernel/hardware interface, including the dispatcher). Here concern is focused on the scheduling algorithm and the scheduler policy. At any time during the execution of a concurrent program there may be a number of processes that can run. The scheduling algorithm will specify (either completely or partially) the order in which processes are scheduled for execution. For example an algorithm which specifies that the dispatcher should always run the process which requires the least amount of execution time (shortest job first) fully defines the order for processes with different execution times, but does not specify a complete ordering as it makes no statement concerning the order of execution of processes with identical execution times.

Although a scheduler policy may completely order processes (e.g. by executing several processes with identical execution times in a first-in-first-out fashion) it may still be impossible to predict the set of runnable processes at any instant if there are unpredictable external events (e.g. interrupts). From the programmer's point of view the scheduler can be considered *non-deterministic* if either the scheduling algorithm is not completely specified or if there are unpredictable external events.

In general it is desirable for the logical behaviour of a concurrent program to be independent of the scheduler. The real-time behaviour will however often be sensitive to the scheduler's actions. This key issue is considered in Section 4.

With message-based languages there is also a form of non-determinism introduced into the "selective wait for message" construct. In Ada, for example, a task can choose between accepting two or more entries using a select statement:

```
select
  accept get_resource(...) do
  ...
  end get_resource;
or
  accept return_resource(...) do
  ...
  end return_resource;
end select;
```

The semantics of the select statement state that if there are outstanding calls to both "get_resource" and "return_resource" then one is chosen *arbitrarily* (that is, it is not defined by the language). The specification of the select statement's behaviour is non-deterministic, although again any implementation of the select will define a policy which is in essence predictable.

The rationale behind making the select arbitrary is that from the programmer's point of view the scheduler is assumed to contain a degree of non-determinism. To illustrate this consider a process P that will execute a selective wait statement on which processes S and T could call. If the scheduler is non-deterministic with respect to P, S and T then there are a number of possible "histories" for this program; these are due either to different implementations of the scheduling algorithm specification, or to the effect of interrupts on the times processes become runnable. For processes P, S and T the following are possible (and legal) histories.

(i) P runs first; it is blocked on the select. S (or T) then runs and rendezvous with P.
(ii) S (or T) runs first and blocks on the call to P; P now runs and executes the select with the result that a rendezvous takes place with S (or T).

(iii) S (or T) runs first and blocks on the call to P; T (or S) now runs and is also blocked on P. Finally P runs and executes the select on which T and S are waiting.

These three possible interleavings lead to P having either none, one or two calls outstanding on the selective wait. The select is defined to be arbitrary precisely because the scheduler is non-deterministic. If P, S and T can execute in any order then, in case (iii), P should be able to choose to rendezvous with S or T—it will not affect the program's correctness.

If we extend this argument to include a scheduler which is fully non-deterministic (i.e. there is no algorithm defined, or the algorithm is not effective, for example all processes having identical execution times in a shortest job first strategy) then there is no rationale for imposing a queue discipline on any synchronisation primitive. Non-deterministic scheduling implies that all such queues should release processes in a non-deterministic order. Although semaphore queues are often defined in this way, entry queues and monitor queues are specified to be FIFO. The rationale here is that FIFO queues prohibit starvation. This argument is however spurious; if the scheduler is non-deterministic then starvation can occur (a process may never be given a processor). It is inappropriate for the synchronisation primitive to attempt to prevent starvation.

In order to influence the scheduler's choice the concept of *task* or *process priority is used together with preemption.* In effect priority is used to order (either partially or completely) the otherwise non-deterministic behaviour of the scheduler.

3. THE ADA PRIORITY MODEL

To illustrate how the notion of priority is actually introduced into a language, the facilities available in the real-time programming language Ada will be introduced and then criticised.

A priority can be associated with a task using a "pragma". Although an Ada implementation must support this pragma, the range of priorities is not defined and, indeed, could be as restrictive as to include only a single value. However, the language does define that if two tasks with different priorities could reasonably use the same processing resources then it cannot be the case that the lower priority task is executing while the high priority task is also executable (but not actually running).

The Ada model is essentially static. Only when a task is in rendezvous with a higher priority task does its priority change (to that of the higher value). Many criticisms of Ada's priority facilities have been made; for example Burns *et al.* [4] note the following:

(a) All tasks of the same type must have equal priority.
(b) Tasks which do not have a priority assigned could act at a higher or lower priority than tasks with an assigned priority (i.e. there is no default priority).
(c) The language defines that all hardware priorities are higher than those that can be assigned via the priority pragma. A device handling task cannot therefore execute at the hardware priority level of the device itself other than when it is in rendezvous with a device, i.e. handling an interrupt.
(d) Client tasks waiting to rendezvous with a server task are queued on each entry in a FIFO order—no account is taken of the client tasks' priorities.
(e) The choice made by a select statement when there is more than one open alternative is arbitrary, i.e. the priorities of the tasks at the head of the queues are not be taken into account.

Points (a)–(c) are examples of a poor definition of the language feature (as opposed to a poor model) and could easily be circumvented. Problems (d) and (e) are however more significant.

Priorities in Ada can indicate the relative urgency among competing tasks but it cannot reliably enforce scheduling strategies. Consider, for illustration, two processes T_1 and T_2 with periods 50 and 10 units respectively. T_1 requires 10 units of execution to meet its deadline, T_2 needs 2 units of execution. T_1 is given a higher priority than T_2 because it is more important. If both processes start at the same time then the preemptive scheduler will run T_1 for 10 units of time and then suspend it for 40 further units (i.e. until its deadline is due). Unfortunately although the processor is now free to run T_2, it is too late for T_2 to meet its first deadline. Cornhill *et al.* go further [5];

they illustrate how a mixture of periodic and aperiodic tasks can fail to meet deadlines, even when processor utilisation is low (60%), because of the pre-emptive rule that forces the time-run system to always run a high priority task in preference to a lower priority one. Furthermore, they show that the FIFO ordering of tasks on entry queues can cause tasks to start missing deadlines when processor utilisation exceeds 45% and in the worst case only 5% (see Section 4).

A different set of difficulties arises when one attempts to program a deterministic version of the select statement. It is surprisingly difficult [6] to construct a priority version of the select which is reliable in the face of failure of calling tasks (see Section 5).

The priority model supported by Ada is clearly weak but it is not untypical of the facilities available in real-time programming language [1]. The lack of appropriate facilities is, we contend, due at least in part to the need to satisfy two distinct and at times contradictory sets of requirements. Firstly there is the need to enable scheduling strategies to be implemented by the run-time dispatcher (so as to meet deadline requirements). Secondly there is a need to program resource allocation algorithms that ensure liveness or particular utilisation levels. These two sets of demands will be considered in turn.

4. PRIORITY AND DEADLINE SCHEDULING

As indicated in the introduction, many real-time systems have to deal with the time requirements of periodic and aperiodic processes. The run-time kernel must try and run periodic activities at the appropriate time intervals whilst guaranteeing response time for the sporadic external events. To this end a number of scheduling approaches have been advocated that are based on the use of either static or dynamic priorities; for example the rate monotonic approach (for periodic processes only) [7] and the ceiling protocol [8]. With real-time languages that use the notion of priority, deadlines are not directly represented but are used (off-line) to calculate the appropriate priority of each process.

We are not concerned here with the details of actual scheduling algorithms. However, in order to implement these algorithms certain requirements must be fulfilled by the priority mechanism of the language. These requirements are of interests to language designers and it is these that we now discuss. To avoid terminology problems we will assume that the scheduler assigns its own priority to processes as a means of ordering the run queue; we will term this the *scheduler priority* of a process. We will call a language's notion of priority the *preference priority*.

If all processes are runnable (executable) then the run-time dispatcher has complete freedom to organise the scheduling of the work. In situations when not all deadlines can be met at the appropriate times then those processes with a higher *preference priority* are given precedence. Alternatively there will be situations where a low preference priority process is run in advance of a high preference priority one, so that both processes' time requirements are met. In this case the low preference priority process will be given the higher *scheduling priority*. It follows that the language definition *should not* impose a strict pre-emptive meaning to its preference priority. However, Sha *et al.* [9] discuss ways of transforming processes so that there is always agreement between scheduler and preference priority.

As indicated above, if the processes are executable the dispatcher can work according to a specified algorithm. The dispatcher cannot, however, have any control over suspended processes. All synchronisation primitives give rise to the possibility of a process being suspended. In a real-time programming language these primitives must take account of the scheduler priority of the suspended processes *not* their preference priority *or* any other ordering scheme such as FIFO. Failure to do so will result in the needless blocking of high priority processes. This implies that:

- if more than one process can be suspended on a synchronisation primitive then the processes must be released in scheduler priority order;
- if a language construct can choose between two or more synchronisation primitives (in order to release a process), the scheduler priority of the processes involved must control the choice.

The first point implies that semaphores, condition variables, monitor access and entry queues must all release processes in scheduler priority order. In addition the second requirement implies that

a selective waiting construct (such as the Ada select statement) must, if there is a choice, release the process with the highest scheduler priority.

Earlier it was noted that if the scheduler is non-deterministic then it is appropriate for the selective wait construct to be similarly arbitrary. If by the use of priority the policy of the scheduler is now fully deterministic (apart from the effects of interrupts then it is apposite for the synchronisation constructs (i.e. semaphores, entry queues, selective waits, etc.) to be similarly prescribed.

With prioritised preemptive scheduling non-determinism can still be a feature of the scheduler's behaviour if two or more processes are given equal priorities. In this situation any synchronisation construct should again make an arbitrary choice between processes of the same priority.

Unfortunately these requirements are not sufficient to ensure that a high scheduler priority process is not unexpectantly delayed. Consider, as an example, a system with 3 processes: P_1, P_2 and P_3 with priorities PRI_1, PRI_2 and PRI_3 where $PRI_1 > PRI_2 > PRI_3$. Let P_3 be, initially, the only runnable processes; and let it enter a critical section (for example). Assume now that P_1 becomes runnable and wishes to enter the same critical section. To retain the integrity of the critical section P_1 must be suspended and wait for P_3 to exit the critical section. If now P_2 becomes runnable it will execute in preference to P_3. Therefore P_1 will be further delayed because P_2 is executing even though the scheduler priority of P_1 (when it is runnable) is greater than P_2. This phenomenon is known as *priority inversion* [10].

There are two methods of limiting this effect. One involves making critical sections non-preemptable—this would prohibit P_2 from preempting P_3—this is however only practicable if critical sections are short (as P_1 would be prohibited from preempting P_2). The other is to use *priority inheritance* [10]. Hence the scheduler priority of a process not only takes into account is preference priority but also the preference priority of those processes which are waiting for it to perform some action. If a process p is suspended waiting for a lower priority process q to undertake some computation then the scheduler priority of q is raised to the scheduler priority of p. In the example given above, P_3 will be given the scheduler priority of P_1 and will therefore run in preference to P_2.

The general case defines the scheduler priority of a process at a particular time to be a function of its own preference priority, its deadline and the scheduler priorities (and therefore deadlines and preference priorities) of all the other processes that are at that time dependent upon it. If this is done then there is an upper bound on the time a process can be blocked; this is needed to check the schedulability of a system of processes.

Note that the Ada model goes some way towards this inheritance by defining the priority of a rendezvous to be the higher of the two priorities of the tasks involved. A full inheritance model would raise the priority of a server task when a call upon it is made not just when it is accepted. (To give a complete model Ada would also have to have priority based entry queues and select statements.)

Inheritance of scheduler priority should not be restricted to a single step. If process P_1 is waiting for P_3, but P_3 cannot deal with P_1 because it is waiting for P_4, then P_4 as well as P_3 would be given P_1's scheduler priority. The implication for the run-time kernel is that the scheduling priorities of processes will change frequently and that it may be better to choose the appropriate process to run (or make runnable) at the time when the action is needed rather than try to manage a queue that is ordered by scheduler priority.

4.1 Implications for language design

In the design of a concurrent real-time programming language, priority inheritance would seem to be of paramount importance. To have the most effective model, however, implies that the concurrency model should have a particular form. With semaphores and condition variables there is no link between the act of becoming suspended and the identity of the process that will reverse this action. Inheritance is therefore not possible. With synchronous message passing and indirect naming, for example use of the channel in occam2, it may also be difficult to identify the process upon which another is waiting. To maximise the effectiveness of inheritance, direct symmetric naming would be the most appropriate form of communication primitive.

Ada uses direct asymmetric naming and a many to one relationship between clients and servers. If a task calls an entry in another task then its priority can be inherited by the called task. However

if a task executes an accept statement (or a select) upon which there is no immediate call then the potential caller(s) cannot inherit the priority of the accepting tasks as there is no way that the caller(s) can be identified. For example a typical buffer task in Ada would have at its heart a select with guarded entries to PUT and TAKE. If a high priority process calls PUT then the server will immediately be raised to this priority level. But if the buffer is at that time full it is not possible to pass on the high priority level to a caller of TAKE and hence the high priority task will be further delayed until such a call is made. The length of this delay will, in general, be not bounded.

It follows from the above that in terms of priority the relationship between the run-time scheduler and the concurrent (i.e. multi-process) program being scheduled should be as follows:

- Within the program the relative importance of the processes is indicated by assigning a static *preference priority* to each process.
- The behaviour of synchronisation primitives is controlled by the current *scheduler priority* of the processes involved.

The current scheduler priority of a process is dynamic and is at any instance a function of:

(a) its preference priority;
(b) its own scheduler priority needed to meet its deadline;
(c) its inherited scheduler priority.

5. RESOURCE SCHEDULING

Although in many hard real-time systems resources are statically preallocated to avoid contention, it is not always possible (in dynamic systems) to determine resource requirements during design. Furthermore many real-time systems contain both critical and non-critical components. Even where worst-case analysis is possible designers may feel that transient overloads are preferable to unacceptable low utilisations which would result from statically allocating resources. Real-time programming languages must therefore contain facilities for the allocation and de-allocation of resources. To this end the client–server structure has proved an important programming paradigm; server processes (or monitors) control access to the system resources whilst client processes make requests upon them. In the context of client-server interaction Bloom [11] expressed the constraints for resource scheduling as follows. A server must be able to deal with requests according to:

(a) the type of service requested;
(b) the order in which requests arrive;
(c) the parameters of a request;
(d) the state of the server and the objects it manages.

In general there are two linguistic approaches to constraining access to a service [12]. The first is *conditional waiting*; all requests are accepted but those that cannot immediately be met are suspended on some synchronisation primitive. A conventional monitor [13] uses this method. The second approach uses *avoidance;* a request is only accepted if it can immediately be dealt with. Here guards are used to avoid certain requests. Avoidance synchronisation is used in Ada and occam2, along with other concurrent programming languages like SR [14].

A monitor structure can deal adequately with three of Bloom's criteria. Queues on access to the monitor and on condition variables are usually organised in a FIFO manner. The parameters of a request are easily taken into account as is the state of the server itself. Where the monitor does not possess sufficient expressive power is in dealing with the type of the service requested. A monitor only supports a single queue of client processes waiting to enter, it is therefore possible to have a higher scheduler priority service request queued behind lower scheduler priority requests. For example, calls to return resources may be behind new requests to obtain resources.

With message-passing systems using an active server and avoidance synchronisation different problems manifest themselves. In particular, it is difficult to deal with the parameters of a request. This is not, however, strictly a priority issue and so is not considered here. A discussion of the expressive power of the Ada guard is to be found elsewhere [15].

If the client–server relationship is many-to-one then FIFO queues will enable requests to be dealt with in their order of arrival (this may be needed to ensure liveness). To deal with different types of request a selective waiting structure is used. However, to give priority to one type of request over another requires a form of selective waiting that is deterministic when there is more than one outstanding request. Ada uses the COUNT attribute to express this requirement. In the example below, a task attempts to give priority to the DEALLOCATE request.

```
select -- priority given to calls to DEALLOCATE
  accept DEALLOCATE(...) do
  ...
  end;
or
  when DEALLOCATE'COUNT = 0⇒
  accept ALLOCATE(...) do
  ...
  end;
end select;
```

Unfortunately, the use of COUNT is not reliable [16] (as a calling task could be aborted or timed out after the evaluation of the COUNT attribute but before the call had been accepted) and the Ada select statement has been criticised in terms of its ability to deal with many priority levels [6]. By comparison with Ada, occam2 provides a form of selective waiting (PRI ALT) that gives each alternative a different priority according to lexicographical order. If such a structure was available in Ada, the above code would be programmed as:

```
priority select
  accept DEALLOCATE(...) do
  ...
  end;
or
  accept ALLOCATE(...) do
  ...
  end;
end select;
```

The use of a priority select gives a static deterministic select. Elrad and Maymir-Ducharme argue that a more dynamic form is required [17] (such as that provided by Concurrent C[18]). To each branch of the select statement they assign a preference level:

```
select
  pref → X:accept DEALLOCATE(...) do
  ...
  end
or
  pref → Y:accept ALLOCATE(...) do
  ...
  end;
end select;
```

In the above both X and Y are integer expressions.

On each execution of the select, X and Y (in this example) are evaluated and compared, the greater value being used to determine which alternative is chosen (if there are outstanding calls on both entries).

To summaries: resource scheduling with a message-based concurrent programming language provides an appropriate level of expressive power if its queues of suspended processes are handled in a FIFO manner and if a deterministic priority form of selective waiting is available. For monitors there is the difficulty of not being able to deal with requests according to the actual type of request being made.

6. A UNIFIED APPROACH TO PRIORITY

The previous sections have outlined the requirements of two distinct allocation mechanisms that use the notion of priority. Clearly there are conflicting requirements. In particular should queues of suspended processes be ordered by the priority of each process or by their order of arrival? Moreover should a select construct be cognisant of the priorities of the waiting processes or the priorities of the alternatives? To illustrate this last point consider a select structure with two alternatives A1 and A2; on each of the alternatives there is a process waiting (P1 and P2) the priority of the processes being Pri1 and Pri2.

```
(priority) select
      A1 -- process P1 calling
   or
      A2 -- process P2 calling
   end select;
```

Table 1 indicates which alternative should be chosen for various values of Pri1 and Pri2 and a select or priority select construct.

The difficulty arises when the priority select and the priority of the clients are in conflict.

The designers of real-time programming languages must develop primitives that deal adequately with these conflicts. In general two approaches could be advocated.

(a) A single construct catering for both requirements.
(b) Two orthogonal primitives, each dealing with one requirement.

If a single construct is used then the programmer must be able to express the appropriate actions to be taken in all circumstances. It follows that the current priority of each process must be available even though this value may change asynchronously due to priority inheritance. Moreover, detailed knowledge of the scheduler's algorithm must be known to the programmer (portability of programs to different schedulers would therefore present difficulties).

To obtain orthogonal solutions to the two sets of requirments necessitates a judgement as to the relative importance of the two requirements. A possible model would have the following characteristics.

(a) Queues are ordered according to priority, but FIFO within each priority level.
(b) A selective waiting construct uses the priority of the processes to determine its choice.
(c) A priority select (i.e. a deterministic select) is considered *weak*, in that the priority of the waiting processes has overriding control over the action of the select.
(d) If necessary a *strong* version of the priority select could be provided that does not take into account the priority of the clients.

The latter form of the select may be necessary in situations where to accept a particular request, even from a high priority process, would lead to deadlock. It is even possible to postulate a *semi-strong* version of the priority select. In this, the alternative of the select with the highest priority is chosen if the process executing the select (the server) has a higher scheduler priority than any of the processes calling in on the select (the clients). If however one of the client processes has a greater or equal priority than the alternative it is calling is chosen. Note that with inheritance the server process will always have a priority of at least the maximum of its clients.

Notwithstanding the possibilities of strong or semi-strong sementics it is the authors' contention that a weak priority select is most appropriate. This contention is based on the view that resource scheduling is a secondary issues when dealing with hard real-time processes. Whether a language should also support a strong version of the select is arguable; we take the view that a weak select with appropriate guards would be adequate under most circumstances.

Table 1. Behaviour of (priority) select

	Pri1 > Pri2	Pri1 = PRi2	Pri1 < Pri2
Select	A1	Arbitrary	A2
Priority select	A1	A1	?

The primary advantage in using the orthogonal approach is that the priority of each process is a matter for the run-time scheduler only (apart from giving the preference priority to each process). The programmer is not aware, and cannot manipulate priorities and thereby undermine the algorithm being used by the scheduler. Moreover resource allocation can be programmed to ensure liveness, or to maximise utilisation, on the assumption that all clients are of equal priority. It is not the function of such an allocator to protect against the indefinite postponement of low priority clients; that is the concern of the run-time scheduler.

7. CONCLUSION

One of the primary functions of many embedded systems is the scheduling of events so as to meet critical (hard) real-time constraints. To this end synchronisation primitives must be aware of the priority of the suspended processes. In addition to avoid priority inversion, inheritance must be employed. This is easier to accomplish if the naming convention in the concurrent programming language uses a direct symmetric form or if indirect naming, with a single reader/single writer, is used.

Notwithstanding the requirements for run-time scheduling most languages use FIFO queues and guarded select statements to ease the programming of resource allocation routines. A need for a deterministic version of the select has been illustrated.

To accommodate the two uses of priority within a single language requires either the joining together of the two sets of needs—so that the programmer has complete control over the behaviour of all synchronisation primitives—or the orthogonal delimitation of the two concepts.

REFERENCES

1. Burns, A. and Wellings, A. J. *Real-Time Systems and their Programming Languages*. Reading, MA: Addison-Wesley; November 1989.
2. Anderson, C., Ada 9X project report to the public. Ada 9X Bulletin Board; January 1989.
3. Wirth, N. Modula: A language for modular multiprogramming. *Software Pract. Exper* **7**(1): 3–35; January–February 1977.
4. Burns, A., Lister, A. M. and Wellings, A. J. A review of Ada tasking. *Lecture Notes in Computer Science*, Vol. 262. Berlin: Springer; 1987.
5. Cornhill, D., Sha, L., Lehoczky, J. P., Rajkumar, R. and Tokuda, H., Limitations of Ada for real-time scheduling. *Proc. International Workshop on Real Time Ada Issues. ACM Ada Lett.* **7**(6): 33–39; 1987.
6. Burns, A. Using large families for handling priority requests. *Ada Lett.* **7**(1): 97–104; January/February 1987.
7. Liu, C. L. and Layland, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM* **20**(1): 46–61; 1973.
8. Sha, L., Rajkumar, R. and Lehoczky, J. P. Priority inheritance protocols: An approach to real-time synchronization. Department of Computer Science, Carnegie-Mellon University; May 1988.
9. Sha, L., Lehoczky, J. P. and Rajkumar, R. Task scheduling in distributed real-time systems. *Proc. IEEE Industrial Electronics Conference*; 1987.
10. Sha, L., Rajkumar, R. and Lehoczky, J. P. The priority inheritance protocol: an approach to real-time synchronization. Department of Computer Science, Carnegie-Mellon University; 1987.
11. Bloom T., Evaluating synchronisation mechanisms. *Proc. 7th ACM Symposium on Operating System Principles*, Pacific Grove, pp. 24–32; December 1979.
12. Liskov, B., Herlihy, M. and Gilbert, L. Limitations of remote procedure call and static process structure for distributed computing. *Proc. 13th Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, pp. 150–159; January 1986.
13. Hoare, C. A. R. Monitors—An operating system structuring concept. *Commun. ACM* **17**(10), 549–557; October 1974.
14. Andrews G. R. Synchronising resources. *ACM Trans. Prog. Lang. Systems* **3**(4): 405–431; October 1981.
15. Keeffe, D., Tomlinson, G. M., Wand, I. C. and Wellings, A. J. A review of Ada tasking. In *PULSE: An Ada-based Distributed Operating System*, pp. 193–227. APIC Studies in Data Processing Series. New York: Academic Press; 1985.
16. Wellings, A. J., Keeffe, D. and Tomlinson, G. M. A problem with Ada and resource allocation. *Ada Lett.* **3**(4): 112–123; January/February 1984.
17. Elrad, T. and Maymir-Ducharme, F. Introducing the preference control primitive: Experiences with controlling nondeterminism in Ada. *Proc. Washington Ada Symposium;* March 1986.
18. Gehani, N. H. and Roome, W. D. Concurrent C. *Software—Pract. Exper.* **16**(9), 821–843; 1986.

About the Author—ALAN BURNS received a B.Sc. in mathematics from the University of Sheffield in 1974 and a D.Phil. in computer science from the University of York in 1978. Dr Burns worked as a lecturer (and senior lecturer) at the University of Bradford for ten years before taking up a Readership at the University of York on 1st January 1990. His research interests centre around real-time systems; in particular the design and use of real-time programming languages. He has recently published a book with Dr Wellings on *Real-Time Systems and their Programming Languages* in the Addison-Wesley International Computer Science Series.

About the Author—ANDY WELLINGS graduated with a combined degree in physics and computer science in 1977 from the University of York, he then spent two years working for the software company Systems Programmers Limited International. He then returned to York as a Research Associate working on distributed operating systems. As a result of this work he successfully submitted a D.Phil. thesis entitled "Distributed operating systems and the Ada programming language". Dr Wellings is currently a senior lecturer; his main interests include distributed operating systems and distributed real-time systems.