RESOURCE CONTROL FOR HARD REAL-TIME SYSTEMS: A REVIEW

Neil C. Audsley†

Real-Time Systems Research Group,
Department of Computer Science,
University of York,
York, UK.

5th August 1991

ABSTRACT

In general, for hard real-time systems, determining the schedulability of a set of processes that require mutually exclusive access to some or all of a set of resources is an NP-hard problem [Mok83]. To counter this complexity, many sub-optimal approaches have been proposed. After an initial discussion of some issues relating to the use of resources, this paper provides a comprehensive review of resource control techniques applicable to the hard real-time domain. Approaches that permit processes to block waiting for a resource, and those that do not allow blocking are considered, in the context of both a uniprocessor and multiprocessor architecture. A discussion of the issues highlighted in the review is provided, incorporating a framework for choosing an appropriate resource control method for a given system. Finally, in conclusion, a number of open research questions are raised.

† This work is supported, in part, by the Information Engineering Advanced Technology Programme, Grant GR/F 35920/4/1/1214

1. INTRODUCTION

In the literature, standard solutions to scheduling processes to meet deadlines in hard real-time systems constrain those systems to have no resources. This is seen in the scheduling methods introduced by Lui and Layland, namely the *rate-monotonic* and the *earliest deadline* [Liu73]. Both of these methods constrain the system designer to a fixed set of periodic processes; each process execution time is deterministic and bounded to a worst-case value; no inter-process synchronisation. The latter implies that processes may not compete for limited system resources: they must be entirely independent.

Permitting shared resources is fundamental to the usefulness of scheduling techniques for the hard real-time application engineer: it is unclear how hard real-time systems can be designed and implemented without such resources. The provision of shared resources creates a scheduling scenario in which processes can directly affect the runnability of other processes. Thus the complexity of the scheduling problem is greatly increased. Indeed, Mok has shown that deciding the schedulability of a process set using semaphores to enforce mutual-exclusion a non-sharable resource is an NP-hard problem [Mok83]. This refers to optimal solutions. To avoid this intractability, proposed resource control techniques for hard real-time systems opt for tractable but sub-optimal solutions to the problem.

This paper reviews a number of techniques that have been proposed as solutions to the resource allocation and control problem. To be applicable for use in the hard real-time systems domain they must have the following attributes:

(a) Predictability - resource allocation decisions must be predictable before the system is run. This prohibits resource allocation schemes whose decisions are based either partially or completely upon environment values which are not known pre-runtime.

(b) Boundedness - the execution time of a process must be bounded with respect to any resource accesses that it makes. This bound must be calculable pre-runtime so that the schedulability of the process set can be determined.

These attributes ensure that the proposed method has bounded deterministic runtime behaviour. If a method does not fulfill these criteria, then arguments about the schedulability of the process set become very difficult to make. In the worst scenario, no guarantees about deadlines of processes can be made.

Techniques that are excluded from the review include optimistic methods [Har90], and conventional operating systems approaches, such as First-Come-First-Served [Lis84]. Neither of these techniques is able to bound resource access times and they can also be unpredictable in terms of runtime behaviour.

The following sub-section introduces the terminology used throughout the remainder of the paper. Section 2 discusses blocking in shared resource systems. An overview is provided describing how, when and why blocking occurs. Section 3 provides classifications of the resource control techniques discussed in the paper. The classifications form an index for the remainder of the paper. Section 4 reviews uniprocessor blocking resource control, with section 5 discussing non-blocking techniques for the same architecture. Section 6 reviews multiprocessor resource control techniques. Section 7 compares and contrasts the resource control techniques discussed, providing criteria by which appropriate resource control techniques amy be chosen for a given system. Finally, section 8 provides the conclusions to this work.

1.1. Terminology

Resources are considered to be logical, with physical resources mapped onto logical representations. Resources can be accessed in either a shared or exclusive manner. We consider only exclusive access in this review. To access a resource, a process must obtain the lock on that resource. Similarly, to release a resource, a process unlocks that resource.

If a process wishes to access a resource, but the lock on that resource is held by another process, the former process becomes blocked. The policy of allocation of an unlocked resource amongst several processes blocked on that resource is entirely dependent on the resource control method employed.

1.2. Nomenclature

Within this review, some of the resource control techniques reviewed are assigned acronyms. These are given now for convenience:

4SM	Four-Slot Mechanism
CSP	Ceiling Semaphore Protocol
DPCP	Dynamic Priority Ceiling Protocol
GMPCP	Generalised Multiprocessor Priority Ceiling Protocol
MPCP	Multiprocessor Priority Ceiling Protocol
PCP	Priority Ceiling Protocol
PIP	Priority Inheritance Protocol
RP	Reservation Protocols
SCP	Semaphore Control Protocol
SMP	Shared Memory Protocol
SRP	Stack Resource Policy

2. NATURE OF BLOCKING

Conventionally, the simplest system to consider in terms of scheduling is one with no resources, no inter-process interaction (including precedence constraints), and a single processor. This implies that the only resource in the system that needs to be scheduled is the processor itself. Hence, scheduling arguments are simplified because no process can affect the running of another process: the interleaving of processes on the processor can be entirely controlled by the scheduler.

For such a constrained system, determining the schedulability of the processes in the system is relatively straightforward . Indeed, the schedulability can be determined by examining the timing requirements of the processes, that is period, deadline and worst-case execution time [Liu73, Leh89, Aud90a]. For such a system, arguments about meeting process deadlines can be made, predictably, offline before the system is executed.

The problem is exacerbated when resources other than the processor are introduced. Whenever resources are required by a process in such a way as to prohibit unrestricted access of another process to the resource, an added scheduling complexity has been introduced. The implicit effect of such resource usage is that processes have gained the ability to influence the running of other processes.

The forms that these influencing operations take are varied, and include the following:

(a) *Mutual Exclusion*

This can occur in two forms: exclusion required over data access (conventionally a shared resource) or exclusion around code access (i.e. non-reentrant code shared by different processes).

(b) *State/Value Condition Synchronisation*
Essentially, this is apparent in problems such as buffer handling. Before a write operation can proceed, a process may wish to ascertain that a free slot exists in the buffer. Therefore the process is synchronised upon the state of a data object. Another alternative is a process that becomes blocked waiting for a data item to attain a certain value or lie in a given interval of data values.

(c) *Synchronous Message Passing*
This mechanism is conventionally used for inter-processor communication. Synchronisation reflects the notion of Remote Procedure Call, or the Ada *rendezvous*. A process communicates a message, but cannot continue executing until a reply is forthcoming.

(d) *Inter-Process Precedence Constraints*
These reflect the view that processes co-operate to reach a goal, and as such, an application may require that co-operating processes are constrained to execute in a pre-defined manner. Thus precedence constraints are formed.

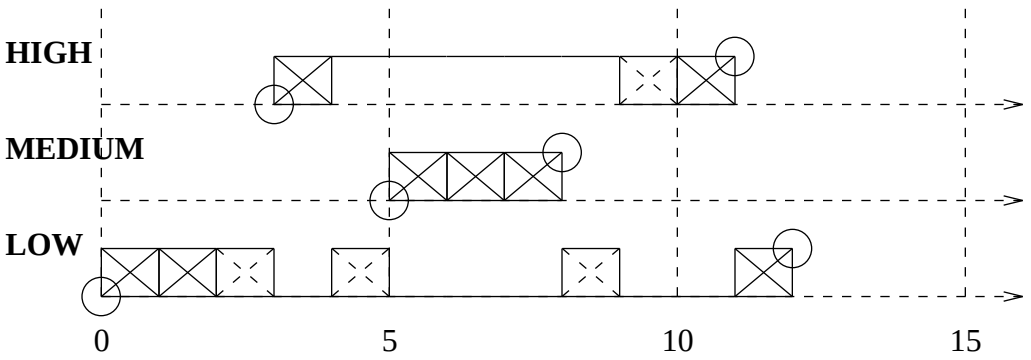
(e) *Data Age*
In hard real-time systems, timeliness of response is essential. One aspect of this is that data may have a limited useful lifetime. Hence, a process may wait until a datum is updated before proceeding.

The literature does not provide solutions to (b), (c) and (f). These remain as open research questions. However, current literature would suggest that solutions are available for (a), (d) and (e). The latter, inter-process precedence constraints can be catered for offline, via a static schedule [Foh89], or online using process priorities . Mutual exclusion (a), and synchronous message passing (d), are considered together throughout the remainder of this paper, as the latter can be considered as a distributed instance of the former.

The requirement to have mutual exclusion over some data can destroy any assumptions made about the meeting of process deadlines as the predictability of process execution has been removed. For example, consider two periodic processes, both of which need to access a logical resource exclusively. Within a static priority system, the situation could arise where the **low** priority process has locked the resource and is preempted by a **high** priority process. The latter attempts to access the resource. **High** is blocked by the lower priority process. A **medium** priority process now preempts **low** and is running at the expense of **high**. This is a form of *priority inversion*[Sha90] where a *low* priority process blocks a *high* priority process but is itself prevented from running by a *medium* priority process. Thus, the *high* priority process has to wait for *medium* to complete execution and *low* to finish its critical section before it can lock the resource and continue execution. Priority inversion is shown in Figure 1†.

---

† The figures show the execution of a set of processes, each with its own horizontal timeline. A process release is indicated by a circle cut by the timeline. Process execution is indicated by hashed boxes, with boxes with dashed hashed lines showing that the process is executing whilst holding a resource. Whilst a process is blocked a solid line above the timeline is drawn, with a solid line on the timeline indicating that the process is preempted. When a process completes execution, a circle above the timeline is drawn. If a deadline is missed a solid bullet is drawn above the timeline.



t = 0 : **LOW** released and executes
t = 2 : **LOW** requests and locks **R**
t = 3 : **HIGH** released and preempts **LOW**
t = 4 : **HIGH** requests **R** but blocked, **LOW** runs
t = 5 : **MEDIUM** released and preempts **LOW**
MEDIUM is now running at the expense of LOW and HIGH
Priority Inversion has occured
t = 7 : **MEDIUM** completes
t = 8 : **LOW** completes its critical region and unlocks **R**
t = 9 : **HIGH** locks *R* and executes critical region
t = 10 : **HIGH** completes
t = 11 : **LOW** completes

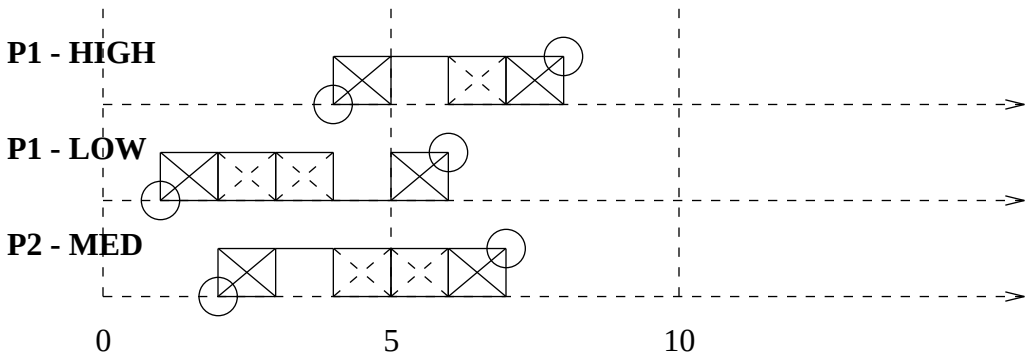
Figure 1. Priority Inversion.

Blocking can only occur when two processes have a priority or importance that is directly comparable. Hence, a partial order of processes (ordered on priority or some other measure of importance) can be constructed. Blocking can arise in dynamic priority systems where a static importance is associated with each process, or the partial ordering of processes according to priority is re-evaluated whenever a process changes priority.

The blocking problem is exacerbated when a distributed environment is considered. Here there is the prospect of being blocked by processes on the same processor (local processes) and by processes running on other processors (remote processes). For the purposes of the protocols discussed in this section blocking in distributed systems is subdivided into two categories:

(i) *local blocking* occurs when a process is blocked on a resource allocated to a lower priority process resident on the same processor.

(ii) *remote blocking* occurs when a process is blocked on a resource allocated to a process resident on a different processor.



Resource R resident on P1
t = 1 : **LOW** released and executes on **P1**
t = 2 : **MED** released and executes on **P2**
: **LOW** requests and locks**R** on **P1**
t = 3 : **MED** requests **R**, but remotely blocked by **LOW**
: **LOW** completes critical region and uunlocks **R**
t = 4 : **MED** granted **R**
: **HIGH** released and preempts **LOW** on **P1**
t = 5 : **HIGH** requests **R** but remotely blocked by **MED**
: **LOW** executes and completes
: **MED** completes critical region and unlocks **R**
t = 6 : **HIGH** enters and completes critical region
: **MED** executes and completes
t = 7 : **HIGH** executes and completes

Figure 2. Remote Blocking.

Remote blocking can arise in two main ways. Firstly a process can make a request for a remote resource and be blocked by a lower priority process already holding the lock on that resource. This is seen at time 3 in Figure 2 where process **MED** is remotely blocked by process **LOW**. Secondly, a process could make a request for a local resource only to find that resource occupied by a process making a remote request from a different processor. This is seen at time 5 in Figure 2 where **HIGH** is remotely blocked by **MED**. The problems are compounded if nested resource accesses are permitted: there is the prospect of a single process attempting to gain access to one resource on a remote processor, then whilst holding that resource making a nested request to access a resource on a third processor. The definition of remote blocking to include blocking by any remote process is to attempt to classify all waiting time in the system as blocking time.

The following sections examine resource control protocols that fit the predictability and boundedness criteria identified in Section 1. With respect to mutual exclusion, the boundedness criteria equates to being able to place a worst-case blocking time upon a resource access.

3. CLASSIFICATION

Many resource control techniques have been proposed for real-time systems. These vary from techniques for use with priority preemptive scheduling algorithms, for example the collection of techniques derived from *priority inheritance* [Sha90], to those that rely upon scheduling resources along with processes in a rigid manner pre-runtime [Sta87b, Dam89]. All these techniques are summarised in Figure 3 according to the following series of criteria:

(a) predictable or non-predictable;

(b) blocking or non-blocking;

(c) runtime non-blocking or pre-runtime non-blocking;

(d) preemptive blocking or non-preemptive blocking.

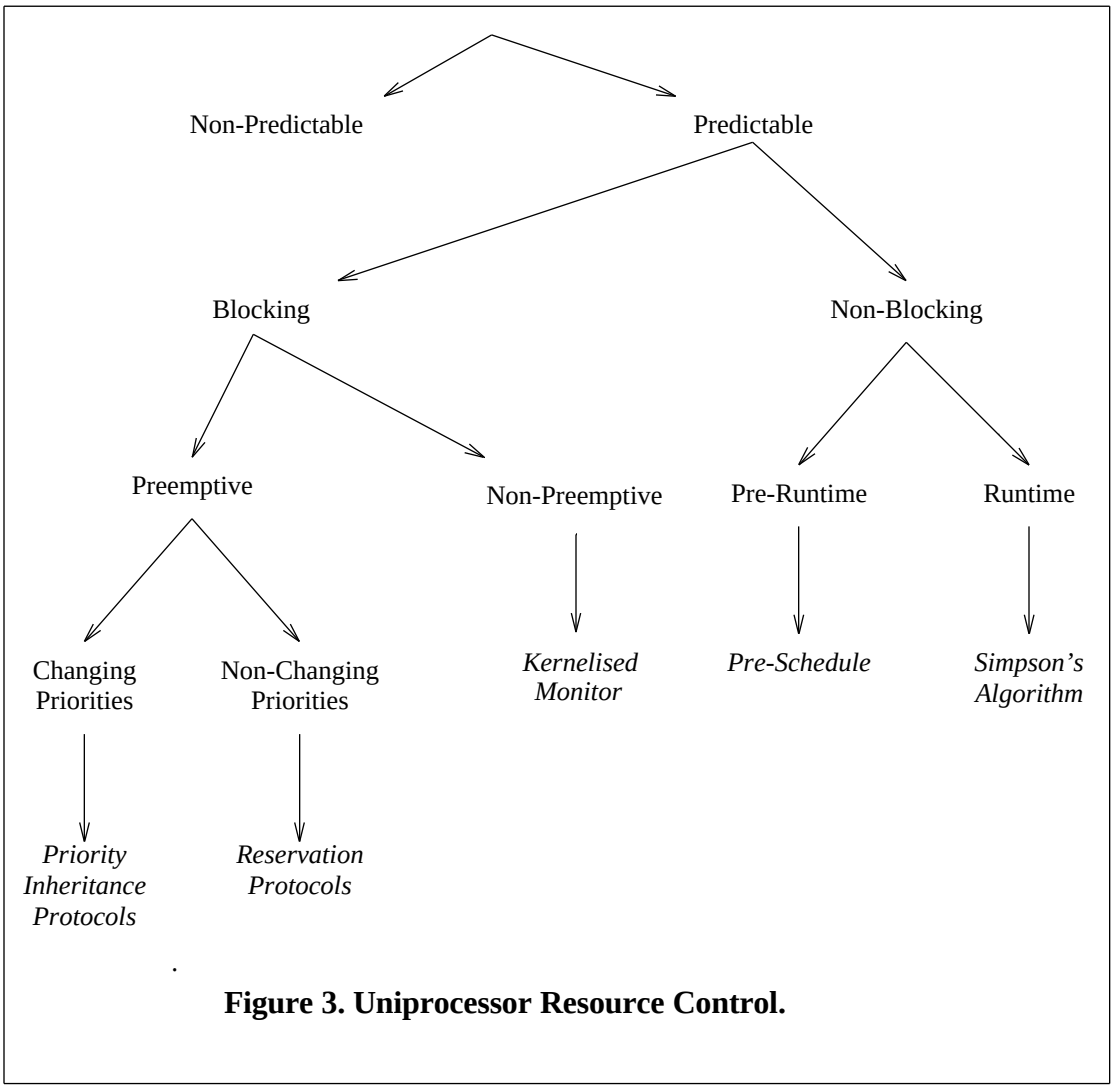


Figure 3. Uniprocessor Resource Control.

Referring to Figure 3, non-predicatable resource control techniques include those that are generally used in conventional multiprocessing computers [Lis84]. For example, First-Come-First-Served can be used for ordering requests for a printer resource. Non-predictable methods are not applicable for real-time resource control and are therefore precluded from the remaining discussions.

Blocking resource control methods include those that allow processes to become suspended waiting for a resource to become free. Such methods must have predictable

policies for allotting resources amongst competing processes. For example, if priority ordered queues are used for ordering requests for a common resource then bounds must be calculable for the time that any process can spend waiting on that queue. One such family of methods that have been developed are those based on *priority inheritance* [Sha90]. These blocking methods can be categorised as those that permit preemption of a process holding a resource (although the resource is not relinquished by the preempted process). Also, a low priority process blocking a higher priority process on a resource can have its priority altered. This can reduce the time that the higher priority process is blocked (by preventing processes with priorities between those of the low and high processes executing). In contrast, another preemptive blocking protocol prevents priorities of processes changing [Bab90]. In this scheme a low priority process is prevented from using a resource if there is a higher priority process that could possibly request the resource whilst held by the low priority process (the higher priority process reserves the resource).

Other resource control techniques based upon blocking prevent a process being preempted whilst holding a resource. One such method, the *kernelised monitor* was proposed by Mok [Mok83]. This method ensures that processes within critical sections cannot be preempted.

In contrast to blocking schemes, non-blocking techniques never permit one process to suspend the running of another process due to resource contention. This can be achieved by allowing processes to use potentially old versions of the resource rather than become blocked waiting for the resource to become free. This approach was suggested by Simpson [Sim90]. Another non-blocking approach is one that pre-schedules all resources and processes. In this manner all blocking at runtime is prevented by the schedule calculated offline. These two non-blocking approaches can be identified as runtime and pre-runtime respectively.

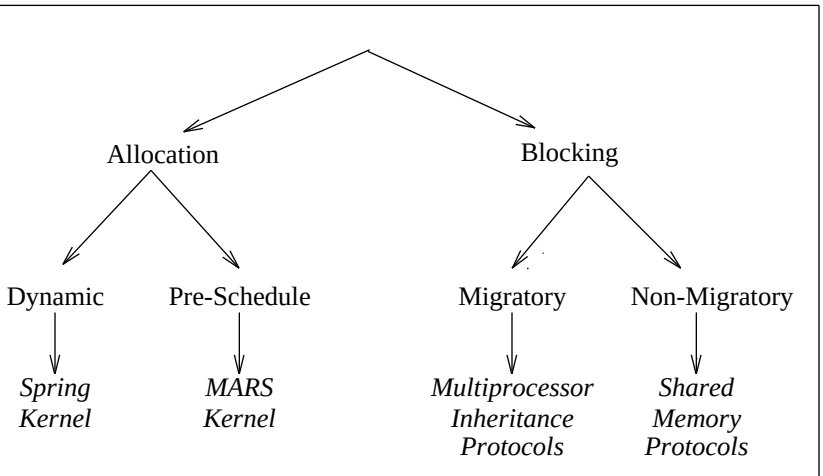


Figure 4. Multiprocessor Resource Control.

Whilst many resource control protocols have been proposed for uniprocessor systems, those for multiprocessors are less numerous in the literature. The approaches adopted fall into two main categories, allocation and blocking. This is illustrated in Figure 4.

The first approach redefines the problem to be one of allocation. That is, resources and the processes that use them are allocated to the same processor. Two allocation methods have been proposed. Firstly, pre-scheduling, where resources and the processes

that use those resources are allocated to the same processor statically. Secondly, dynamic re-allocation, where processes can be moved, at runtime, between processors that can meet its resource requirements. These methods remove the problems of remote blocking directly.

In the second approach, the problem of remote blocking on a multiprocessor architecture is tackled. A number of techniques enable remote and local blocking to be bounded and calculable. These include developments of the uniprocessor priority inheritance protocols for multiprocessors. Two developments enable processes that require remote resources to migrate, in an abstract sense, to the processor holding those resources. These are termed migratory. Another proposal (for shared memory multiprocessor systems) enables the resource to be brought to the processor of the requesting process. This approach is termed non-migratory.

The following sections review the uniprocessor and multiprocessor techniques outlined above.

4. UNIPROCESSOR BLOCKING APPROACHES

Resource control protocols for priority-based process sets are required to base their decisions upon resource allocation amongst contending processes on the priority of the processes themselves [Bur90]. Even using such criteria it is possible for processes to be blocked for an unbounded time [Pil90]. This is not acceptable for hard real-time systems. Such systems require deadlines of processes to be guaranteed [Aud90b]: this is clearly impossible if processes could have both unbounded blocking and a hard deadline.

The following sub-sections outline preemptive and non-preemptive resource control methods respectively.

4.1. Preemptive Blocking Methods

Preemptive blocking methods are those that permit processes to be preempted whilst holding resources. There are two sub-categories: priority changing and non-priority changing. The priority changing blocking resource control techniques subject to review in this paper are all based upon the the *priority inheritance* model proposed by Sha *et al* [Sha90]. The following sub-section describes priority inheritance, with subsequent sub-sections introducing developments of priority inheritance. Finally, a non-priority changing protocol is reviewed.

4.1.1. Priority Inheritance

The priority inheritance protocol (PIP) [Sha90] assumes that:

(a) static priorities are assigned to processes;

(b) resources are accessed in a mutually exclusive manner;

(c) resource accesses are properly nested;

(d) a preemptive priority driven scheduler is used (where the highest priority runnable process is given the processor);

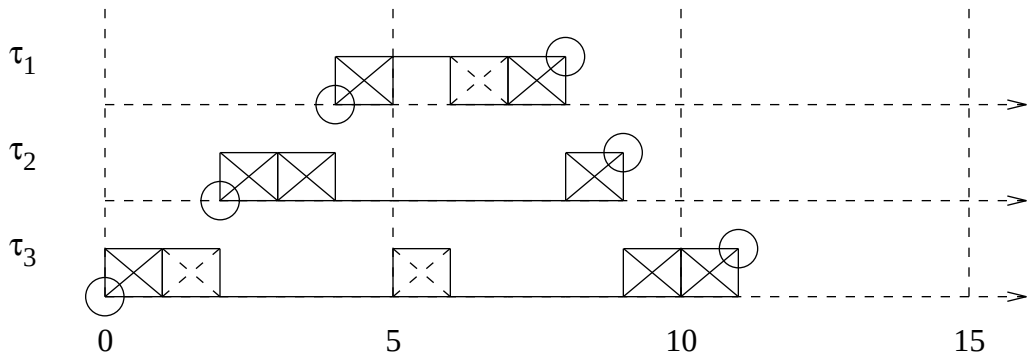
(e) the resources that a process accesses can be determined pre-runtime.

The PIP can be stated as the following rules to be used when a resource **R** is requested by process τ_i:

- if **R** is free, then it is granted to τ_i.
- if **R** is already locked (this must be by a lower priority process τ_j as τ_i has preempted the locking process), then τ_i becomes blocked on **R** by τ_j. At this point, the τ_j inherits the priority of τ_i and hence becomes runnable. When **R** is unlocked by τ_j, the priority of τ_j returns to normal with τ_i immediately becoming runnable (if no process of higher priority has been released).

The definition of the PIP permits transitive inheritance of priorities. For example if process τ_3 locks **R** and is preempted by τ_2 which becomes blocked on **R**, which is in turn preempted by τ_1 which is also blocked on **R**, then τ_3 inherits the priority of τ_2 and then the priority of τ_1.

The behaviour of priority inheritance is illustrated in Figure 5.



t = 0 : τ_3 released and executes
t = 1 : τ_3 requests and is granted **R**
t = 2 : τ_2 released, preempts τ_3 and executes
t = 3 : τ_2 executes
t = 4 : τ_1 released, preempts τ_3 and executes
t = 5 : τ_1 requests **R** and is blocked
τ_3 *inherits the priority of* τ_1
: τ_3 completes critical regions and unlocks **R**
τ_3 *priority returns to normal*
t = 6 : τ_1 granted **R** and completes critical region
t = 7 : τ_1 executes and completes
t = 8 : τ_2 executes and completes
t = 9 : τ_3 executes
t = 10 : τ_3 executes and completes

Figure 5. Priority Inheritance Protocol.

The protocol is able to bound the time that a process is blocked during its execution. The blocking time for a process τ_i evaluates to *min(m,n)* critical regions of lower priority processes, where *n* is the number of lower priority jobs which are able to block τ_i, and *m* is the number of resources used by lower priority processes that can block τ_i. Effectively, the blocking of τ_i is equal to the sum of the longest critical regions of each lower priority process.

The blocking arises from two sources. Firstly, *direct* blocking occurs when a high priority process attempts to access a locked resource. This is seen in Figure 5 when process τ_1 attempts to lock the semaphore but is blocked. Secondly, *push-through* blocking occurs when priority inversion is avoided. This is seen in Figure 5: τ_1 is blocked by τ_3 which has inherited the priority of τ_1 (since τ_3 holds a resource required by τ_1). Process τ_2 is also blocked. This avoids τ_2 running whilst τ_1 is blocked on a resource held by τ_3. Hence, the blocking has been pushed-through onto τ_2.

The PIP suffers from two major problems, namely deadlock and chained blocking. As described above, the protocol does not provide protection from deadlocks. This can be seen if τ_2 locks **R1** but before it can lock **R2**, is preempted by τ_1. The latter process locks **R2** and now requires **R1**. Thus deadlock has occurred.

Chains of blocking can be formed where processes can be blocked for a time greater than one critical region. This can be seen if τ_3 locks **S1** with τ_2 preempting and locking **S2**. Process τ_1 now preempts and requires **S1** and **S2**. Process τ_1 will be blocked for a chain of 2 critical regions.

The problems apparent in the PIP are due to a process being able to lock a free resource at any instant, irrespective of its priority relationship with other processes that have locked resources. Hence a high priority process can arrive to find several of the resources it requires locked by lower priority processes.

4.1.2. Priority Ceiling Protocol

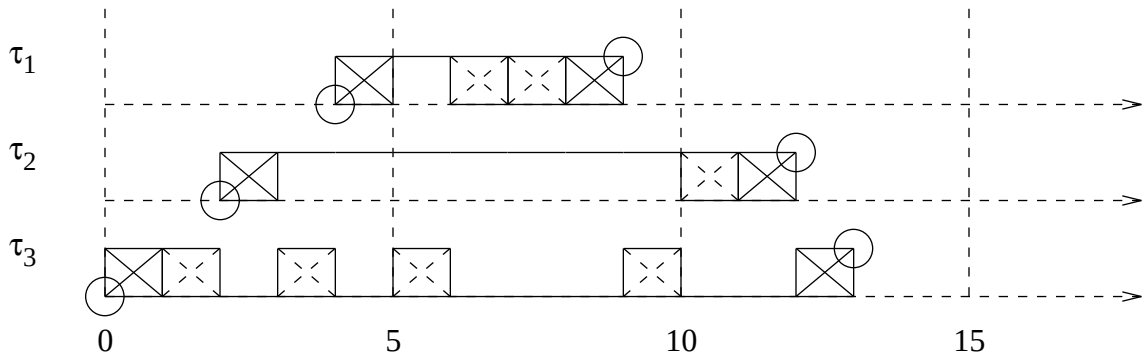
The priority ceiling protocol (PCP) is one instance of a class of priority inheritance protocols [Sha87, Sha90]. The motivation of the PCP is to address the deadlock and chaining problems of the priority inheritance protocol. This is achieved by ensuring that a strict ordering of critical region execution is maintained. The same assumptions are made about processes and resources as in priority inheritance.

The notion underpinning PCP is as follows. A process τ_i can only lock a resource if that resource, or any other locked resource in the system, is not accessed by a process with higher priority than τ_i. Thus, the priority that the process holds whilst holding a resource is guaranteed to be higher than can be inherited by any preempted process.

The PCP can be summarised as:

- a priority ceiling is assigned to each resource equal to the highest priority of all processes that could lock it.
- a resource is allocated if the priority of the requesting process is strictly greater than the ceilings of all currently held resources. If the resource is not allocated, the requesting process becomes blocked upon that resource.
- A process executes at its assigned priority unless it blocks a higher priority process at which time it inherits the priority of the blocked process for the duration of the current critical region (as in priority inheritance protocol).

The maximum priority that a process can inherit whilst holding a resource is equal to the ceiling of that resource. The working of the PCP is illustrated in Figure 6.



Resource R1 shared by τ_1 *and* τ_3*: ceiling 1*
Resource R2 shared by τ_2 *and* τ_3*: ceiling 2*
t = 0 : τ_3 released and executes
t = 1 : τ_3 requests and is granted R2
t = 2 : τ_2 released, preempts τ_3 and executes
t = 3 : τ_2 requests R1 but denied as priority not greater than
ceiling of currently locked resources (i.e. 1)
τ_3 *inherits priority of* τ_2 *and runs*
τ_3 requests and is granted R1 as no other
process holds a resource with higher ceiling
t = 4 : τ_1 released, preempts τ_3 and executes
t = 5 : τ_1 requests R1 but blocked
τ_3 inherits priority of τ_1 runs and releases R1
t = 6 : τ_1 executes inside critical region
t = 8 : τ_1 executes and completes
t = 9 : τ_2 blocked so τ_3 runs and completes critical region releasing R2
NB τ_3 still running at priority of τ_2
t = 10: τ_2 granted R2 and enters and completes critical region
t = 11: τ_2 executes and completes
t = 12: τ_3 executes and completes (now running at normal priority)

Figure 6. Priority Ceiling Protocol.

Deadlock avoidance is inherent in the above protocol as a strict priority ordering of critical region executions is maintained. A formal proof of this was developed by Pilling *et al* [Pil90]. Chaining is also avoided with the result that the maximum blocking time to which a process can be subjected is the longest critical region of all lower priority processes.

Blocking time arises from three sources. Direct and push-through blocking were discussed in the previous section. The third form of blocking, *ceiling blocking*, is required for the avoidance of deadlock and chained blocking. It arises when if a process is denied a free resource due to a locked resource having higher ceiling than the requesting process's priority. This can be seen at time 3 in Figure 6: τ_2 requests unlocked resource R1, but is denied.

One disadvantage of the PCP is its pessimism in terms of blocking times. The only

circumstances that a high priority process **H** can be blocked for the entire duration of the critical region of a lower priority process **L** is when **L** locks a resource required by **H** (or required by an even higher priority process **H**) and performs no execution before **H** requires a resource. Effectively, the **L** must lock the resource momentarily before **H** becomes runnable. This is clearly pessimistic.

A second disadvantage is pessimism in terms of resource access. This can be seen at time 3 in Figure 6. Process τ_2 requests unlocked resource R1 but is denied. However, if it were allocated to τ_2, this process would complete its critical region before τ_1 became runnable, thus avoiding any deadlock and blocking chaining problems. This problem cannot be circumvented without considering exactly how long a resource is required when allocation decisions are made.

4.1.3. Semaphore Control Protocol

The semaphore control protocol (SCP) [Raj88b] provides an extension of the PCP with the same assumptions regarding resources and processes. The behaviour of SCP is identical to that of PCP with the exception of the conditions under which requests for resources are granted. Three conditions are given for process τ_i to be granted resource **R**. If any of these conditions are met then the resource is granted:

(i) if the priority of τ_i is strictly greater than the ceilings of all currently locked resources, **R** is granted.

(ii) if τ_i has priority equal to the highest ceiling of currently locked semaphores and the current critical region of τ_i will not attempt to lock resources with ceiling equal to its own priority, then the request is granted.

(iii) if τ_i has priority equal to the ceiling of **R** and no currently active process will request **R** during the execution of their current critical regions, then τ_i is granted **R**.

The first condition (equivalent to the PCP) ensures that a total priority ordering of critical region executions. This ensures the prevention of deadlock. The second condition maintains the priority ordering of critical region executions even if the priority of the requesting process is equal to the highest ceiling of all currently held resources. The third condition maintains the ordering even if a higher priority process eventually preempts. This is achieved since if the requesting process is the highest priority process to use **R** then it cannot force a higher priority process to be blocked on **R**.

In [Raj88b] the authors prove that the granting of resources according to the above rules is sufficient and necessary (with respect to approaches based upon the priority inheritance model). Condition (i) above provides sufficiency. However, requests for resources may be turned down when they could be granted without the danger of ensuing deadlock. Hence condition (i) is not necessary. Conditions (ii) and (iii) provide necessity. (Hence the PCP is sufficient and not necessary).

However, the SCP suffers from the same disadvantage as the PCP in terms of denying a process a free resource which could not possibly lead to deadlock or chained blocking. This problem arises due to non-consideration of the timing characteristics of the critical regions of requesting processes.

The SCP bounds blocking in exactly the same manner as the PCP. Blocking time arises in exactly the same manner as PCP. However, SCP allows a process to be blocked only when absolutely necessary. The actual blocking time is equal, but the points in time that the blocking time is endured by a process is more reflective of the exact nature of the

resource request (by conditions (ii) and (iii) above).

4.1.4. Dynamic Priority Ceiling Protocol

The motivation behind the Dynamic Priority Ceiling Protocol (DPCP) [Che89] was to enable the coupling of a resource control protocol based upon static priorities (i.e. the PCP) with a scheduling algorithm based upon dynamic priorities, such as the Earliest Deadline algorithm [Liu73]. By permitting processes to have dynamic priorities the DPCP requires a mapping between the priority ceiling associated with each resource, and the priorities of the processes. This is achieved by:

- the effective process set contains exactly one instance of all the processes in the system. The priority of a process is defined to be the priority of its currently active execution, or that of the next execution otherwise. Thus, we can assign priorities at any time to processes in the effective process set, with the process closest to its deadline assigned the highest priority.
- the dynamic priority ceiling of a semaphore, **S**, is the priority of the highest priority process in the effective process set that may lock **S**.

Since the effective process set is constantly changing, the ceilings of all semaphores in the system must also be continually updated. If the Earliest Deadline scheduling algorithm is used then the effective process set can only change at the release or the deadline of a process, with ceilings being updated at these points only. Resource locking proceeds in exactly the same manner as PCP.

Chen *et al* have proved that chained blocking and deadlock are prevented in the DPCP [Che89]. Also, since a process may only preempt another process if it has a shorter deadline than the running process, the maximum blocking time a process endures is exactly as in the PCP: the longest critical region of all processes with longer periods.

The one major problem of the DPCP is the implementation cost of re-evaluating ceilings of semaphores every time a process is released or meets a deadline, although Chen *et al* do present a possible implementation reducing this overhead.

4.1.5. Stack-Based Resource Control

The Stack Resource Policy (SRP) [Bak90b] provides three refinements to the PCP:

(i) multiunit resources - this allows for multiple reader / single writer resources: to read the resource a process can request 1 unit, for a write a process may require all the units. If a resource only has one unit then it is equivalent to a binary semaphore.

(ii) support for processes with dynamic priorities - this supports Earliest Deadline scheduling [Liu73].

(iii) sharable runtime stack - this avoids the overhead of having to assign a separate stack to each process with the associated memory overhead.

Within SRP each process is assigned a priority (which may be dynamic) and a static preemption level. The latter is a measure of how processes can preempt one another. For example, a process with a low preemption level may not preempt a process with a high preemption level. For Rate-Monotonic and Earliest Deadline scheduling, preemption levels are assigned according to deadline of the process: the process with the shortest deadline is assigned the highest preemption level; the longest deadline process is assigned the lowest preemption level.

Each resource has a dynamic ceiling. This gives a measure of which processes will become blocked on that resource if they were runnable. Specifically, at any time, a resource will have *n* units free. The ceiling is given as the maximum preemption level amongst those processes that could becomed blocked on the resource (by requesting more than *n* units of that resource).

When a process τ_i is released, it can preempt the running process if and only if the following conditions hold:

(a) the entire resources required by the process are available. This requires that the preemption level of τ_i be numerically less than the ceiling of all resources it requires.

(b) the resources required to complete all processes that can preempt τ_i are also available. This requires that all processes with higher preemption levels than τ_i must be examined to see if they have a preemption level numerically less than the ceiling of all resources that they require.

Condition (a) ensures that a job cannot block after it commences execution. This also prevents deadlock in the system. Conditions (a) and (b) together are sufficient to prevent multiple priority inversion and so bound the maximum blocking time a process can endure to the duration of the longest critical region of all the lower priority processes (i.e. identical to PCP).

Note that the SRP does not actually allocate resources to processes until they are requested. This is an important consideration when examining the behaviour of SRP. Consider a process τ_i which has sufficient resources to run. Process τ_j has a higher preemption level than τ_i and could therefore preempt τ_i if its resources were available. The sets of resources required by τ_j and τ_i are not disjoint. Two specific cases emerge. Firstly, if the resources required by τ_j have already been requested by and allocated to τ_i, then τ_j cannot preempt τ_i. Secondly, if these resources were still available then τ_j can be run. This latter case only occurs since τ_i is not assigned its resources until it has requested them.

The stack sharing aspect of this work is motivated by the observation that in certain situations, allocation of stacks on a per-process basis can be wasteful. Certainly if the SRP is used, then since a process cannot be blocked by another process once it begins to execute, then a single stack can be shared by all processes. The saving that is made is related to the number of processes and preemption levels. If there are more processes than preemption levels, then a saving will be made. For example, 100 processes all have unique process levels and so a shared stack of 100 units is required. If the processes are assigned one of 10 preemption levels then a stack of size 10 units is required since at most one process from each preemption level can be active at any time.

The main refinement offered by the SRP to the PCP is to permit processes to have dynamic priorities. This was also seen in the previous subsection regarding the work of Chen *et al*. However, the overheads of the SRP are less than for the DPCP since ceilings are based on static preemption levels and do not have to be dynamically recalculated as the dynamic priority based ceilings of the DPCP.

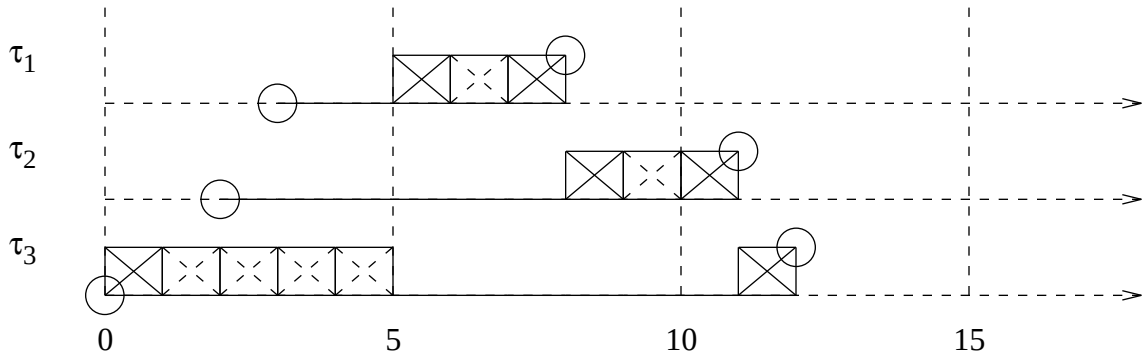
The SRP also reduces the number of context switches needed. The PCP requires two context switches per process, and two for occasions where a higher priority process is run and becomes blocked by a lower priority process. The SRP only requires two context switches as a process cannot become blocked once it has commenced execution.

The major criticism of the SRP is that it can increase the response time of processes over that of the PCP. This is due to always forcing blocking to occur at the release of the

process. This is pessimistic in the context of processes whose exact runtime behaviour is data-dependent. For example, a process may be forced to wait for a resource to be free even if the process uses that resource infrequently.

4.1.6. Ceiling Semaphore Protocol

The ceiling semaphore protocol (CSP) is a version of the PCP implicitly suggested in [Kle90, Bak90a]. and outlined in [Raj89]. This refinement of the PCP requires a process, when granted a resource, to set its priority equal to that of the priority ceiling of the resource. This is in contrast to the ordinary PCP which only raises process priorities when a process actually blocks a higher priority process, and even then the priority is not necessarily raised upto the priority ceiling.



Resource shared by all three processes: ceiling 1
t = 0 : τ_3 released and executes
t = 1 : τ_3 requests and is granted R
τ_3 *inherits ceiling priority of 1*
t = 2 : τ_2 released but τ_3 continues running as it has higher priority
t = 3 : τ_1 released but τ_3 continues running as τ_1 has not got higher priority
t = 4 : τ_3 completes critical region, unlocks R and returns to natural priority
t = 5 : τ_1 executes
t = 6 : τ_1 allocated R
t = 7 : τ_1 executes and completes
t = 8 : τ_2 executes
t = 9 : τ_2 allocated R
t = 10: τ_2 executes and completes
t = 11: τ_3 executes and completes

Figure 7. Ceiling Semaphore Protocol.

The advantages of CSP over PCP are twofold. Firstly context switches are reduced. For example, consider processes τ_1, τ_2 and τ_l which all access resource **R** using the PCP (with priority $\tau_1 > \tau_2 > \tau_3$). Process τ_3 locks **R**. Process τ_2 preempts τ_3 and is blocked on **R**. The priority of process τ_3 is raised to 2. Process τ_1 preempts τ_3 and is also blocked on **R**. The priority of process τ_3 is now raised to that of the priority ceiling of **R** (i.e. 1). The cost in terms of context switches due to preemptions whilst τ_3 held **R** is 4. The execution

of the same processes under the CSP removes these context switches as τ_3 would assume the ceiling priority on acquiring **R**. This is illustrated in Figure 7.

The second advantage of this scheme is reduced complexity at runtime. This is due to the elevation of a process to the ceiling priority occurring in one stage, rather than (potentially) many as under the PCP.

The major disadvantage of this scheme is an increase of process response time when compared with the PCP. This can be seen by the example above. If τ_3 locks resource **R**, its priority is raised to the ceiling of **R**, namely 1. Process τ_2 is now prevented from running, even if it does not require any resources. Under the PCP, τ_2 would be able to preempt τ_3.

4.1.7. Reservation Protocols

The work presented by Babaoglu *et al*[Bab90] presents an opposing view to that expressed by the family of protocols derived from priority inheritance. Whilst both approaches solve the same problem, namely priority inversion, Reservation Protocols (RP) do not alter process priorities to achieve this goal. Also, RP allows incomparable process priorities to exist. This situation can occasionally occur in real-time systems. For example, interrupt handlers that cope with interrupt masks: handler A can mask interrupts B and C, handler B can mask interrupt C and handler C can mask interrupts A. Circular masking can occur, and so the priorities of the interrupt handlers are incomparable.

The RP is based upon a graphical definition of priority inversion. Specifically, a graph can be drawn with representing processes. Two sets of directed arcs are then drawn between the nodes:

(i) directed arcs between a process to all other processes with a higher priority (*priority-relation*);

(ii) directed arcs between a process to all processes holding resources that the former process is waiting for (*wait-for-relation*).

This is illustrated in Figure 8. *Priority-relation* arcs (dashed arrows) show that the priority ordering of the processes is $\tau_1 > \tau_2 > \tau_3 > \tau_4$. *Waits-for-relation* arcs indicate τ_1 requires a resource held by τ_2, τ_2 requires a resource held by τ_3, τ_3 requires a resource held by τ_4.

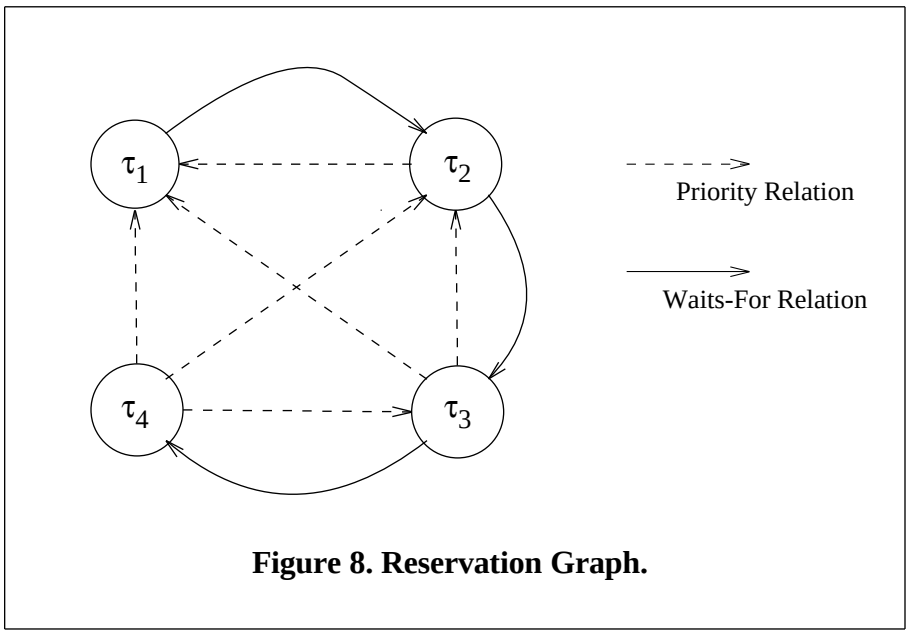


Figure 8. Reservation Graph.

A priority inversion can be detected in such a graph by the existence of a loop. Such a loop, or *π–cycle*, starts with a *priority-relation* arc from a node, continues with *waits-for arcs*, eventually returning to the original node. In Figure 8 two *π–cycles* are illustrated: $\tau_3 \rightarrow \tau_1 \rightarrow \tau_2 \rightarrow \tau_4 \rightarrow \tau_3$ and $\tau_3 \rightarrow \tau_2 \rightarrow \tau_4 \rightarrow \tau_3$. Hence τ_3 is responsible for delaying all the other processes.

Four methods are cited for avoiding such cycles (and thus avoiding priority inversion) including:

(i) no priority assignment - all processes have incomparable priorities;

(ii) preemption - a process is permitted to force another to relinquish a resource;

(iii) no *π–cycle* - so no priority inversions can exist.

Condition (iii) above is developed into the RP by having each process reserve in advance the interval during which it will hold a resource. Thus, by ensuring that a lower priority process reservation never overlaps with a high priority reservation, *π–cycles* can be prevented and priority inversion avoided. Two policies are identified:

(i) a process is granted a resource only if it will release that resource before any process with a higher priority (or incomparable priority) requests that resource.

(ii) a process is granted a resource only if it will release that resource before any higher priority process requests any resource.

The RP (i) is conservative as wait-for relations between incomparable processes are prevented from occurring in case a *π–cycle* is formed between two comparable processes. However, if all processes have comparable priorities then reservation is only made with respect to processes with higher priority processes, which is intuitive. The first RP is superior when process priorities are totally ordered, which is conventionally the situation in real-time systems.

These protocols suffer from one major problem, that of increasing process response times. This is due to processes not being able to lock resources unless they will unlock those resources before any higher priority processes require them. Consider the following example. A low priority process τ_l requires a resource **R** for 5 units of time with a high priority process τ_h becoming runnable in 4 units and requiring **R** immediately. Under the RP, τ_l will be denied **R**. Thus the system will be idle for 4 units of time. If τ_l were assigned the resource, τ_h would only be blocked for 1 unit, thus reducing the average response time at the cost of blocking a higher priority process.

By inspection, the Reservation Protocols avoid deadlock. The blocking time that a process can encounter is at its worst in the following scenario:

- process τ_1 ... τ_n in descending priority order.
- each process consists solely of the use of critical section **S**.
- processes are released in the order τ_n ... τ_1 (i.e. in ascending priority order).
- τ_n is released and requests **S**. The request is refused die to the next highest priority process, τ_{n-1}, requiring **S** fractionally before τ_n is projected to complete its use of **S**.
- this occurs successively to processes τ_n to τ_2. At this point, there has been an idle time approximately equal to the sum of computation times of τ_n ... τ_2.
- τ_1 is released, requests and is granted **S**, and runs to completion.
- τ_2 now runs to completion, followed by τ_3 ... τ_{n-1}, .., τ_n.

Each process is (effectively) blocked for a length of time equal to twice the worst-case execution time of all higher priority processes.

4.2. Non-Preemptive Blocking Methods

An alternative to preemptive blocking, is to ensure that a process holding a resource is never preempted. This approach is similar in concept to protocols used in non-real-time systems where issues of bounded blocking and fairness are not considered.

The kernelised monitor [Der89] prohibits a preemption of a process inside a critical region. The length of critical regions is required to be small. In this way, the blocking time that any process can endure is limited to the maximum length of a critical region. This is similar to an extension to CSP where all resources have a priority ceiling equal to the maximum priority of all processes.

The kernelised monitor requires programming and design discipline to keep critical regions small. If critical regions become large, then the blocking times that processes are forced to endure whilst waiting for a process to finish a critical region can be large. The kernelised monitor is especially useful in systems where preemption costs are high compared to critical region execution times as system overheads can be minimised.

4.3. Summary

Two forms of resource control protocols permitting process blocking have been introduced. The first permitted processes to be preempted whilst holding a resource, the second did not permit such action.

The protocols permitting processes to be preempted are mostly based upon priority inheritance. These protocols avoid deadlock and unbounded blocking time and have been defined for both static priority and dynamic priority systems. The variations on the original priority ceiling protocol attempt to refine the protocol. Most introduce increase process response times (i.e. shared resource protocol and priority ceiling inheritance) although the semaphore control protocol improves the PCP without introducing fresh disadvantages.

The reservation protocol also permits preemption. The RPs allow a more semantic view of resource control, as allocation policy actually examines when in a processes execution it will use the resource, rather than assuming that it will attempt to lock all required resources with the first instruction of the execution as in the PCP.

All protocols examined which permit preemption are pessimistic. Blocking times are sometimes unnecessary due to preventing deadlock scenarios. In contrast, the non-preemption protocol examined, the kernelised monitor, is not as pessimistic. Resources are not denied to processes on the grounds of a higher priority process may require that resource (as in the reservation protocol and the priority inheritance family). However, the price for this increased optimism is that processes must hold resources for a minimum time: critical regions must be kept short.

5. UNIPROCESSOR NON-BLOCKING APPROACHES

In the previous section it was seen that blocking protocols inflict timing penalties on processes. Even in the best cases this causes the schedulability of the system to degrade. Non-blocking resource control protocols are not permitted to inflict blocking times onto processes. This is achieved in two main ways. Firstly, resources are pre-scheduled in the same manner as processes to ensure that whenever a process is runnable, all the resources it requires are available. Secondly, asynchronous communication can be used at runtime. These approaches are examined in the following two subsections.

5.1. Pre-Runtime

One method of ensuring that all resources are available whenever a process runs, thus avoiding runtime blocking, is to statically define a schedule offline. This can be achieved by exhaustive search, although such a technique is intractable. A number of scheduling approaches have been proposed that pre-schedule both processes and resources are now discussed.

Stankovic *et al* [Sta87b] for example begin with an empty schedule. Periodic processes are then placed into the schedule in such a position as to ensure the availability of required resources and the deadline of the process is met. If all processes can be inserted in such a manner, a valid schedule has been found. If some processes have not been inserted, backtracking is performed to consider other options. Heuristic functions are used in two places in the search:

(i) to limit the scope of backtracking - achieved by having a feasibility function which computes whether any feasible schedules can result from the current unfinished schedule.

(ii) to provide suggestions as to which process to insert into the schedule next. Options at this stage include the process with the least laxity or the earliest deadline.

The graph based scheduling approach developed by Koza and Fohler [Foh89] is similar.

The advantages of this approach are the avoidance of deadlock and blocking. The main disadvantage is the inevitable inflexibility that is brought into the system by pre-scheduling of resources and processes [Aud90b]. It would appear difficult to accommodate any change required by a long lifetime system if such a static approach were adopted.

5.2. Runtime

An asynchronous communication mechanism has been proposed by Simpson [Sim90]. This mechanism is designed for use with the Mascot-3 design methodology [Bat86] which is based upon a data-flow model, with a single reader and single writer process for each item of data. Within this model, Simpson has developed a *four-slot mechanism* (4SM) which prevents the reader and writer of an item of data ever interfering with each other.

The 4SM provides four data slots to be used by the reader and writer processes such that the writer will never write to the slot that is currently being read by the reader, and the reader will never read from a slot that is being written to. The advantages of this approach are the avoidance of deadlock and chained blocking. Indeed, processes are not subjected to any blocking with the obvious advantage when the schedulability of the system is calculated.

The first problem with this approach is that time coherence of data can be violated. For example, as well as processes having deadlines in terms of their timeliness, processes may also need data that is no older than a given time value. It can be seen that the 4SM does not, in itself, guarantee such time coherence of data. However, no blocking is ever introduced into the system as a reader process is always able to read an item of data (although this may be out of date) and a writer is always able to write an item of data (although this may overwrite an unread piece of data). A secondary problem with the 4SM is that an item of data can only have a single writer and single reader process, although the approach does appear extensible.

5.3. Summary

Non-blocking approaches provide inherent deadlock avoidance by either constraining the schedulability of the system by pre-scheduling all resources, or by introducing an asynchronous communication mechanism with associated data age problems. The first solution is inflexible since it is difficult to see how such a prescheduled solution can react in the dynamic manner generally associated with hard real-time systems.

6. MULTIPROCESSOR RESOURCE CONTROL PROTOCOLS

The potential performance increase promised by multiprocessor systems has lead to their widespread use in the real-time domain. This has lead to the development of several mechanisms to support resource accesses across such systems. In a similar manner to the single processor resource control protocols examined in previous sections, multiprocessor resource control must be able to bound blocking and avoid deadlock.

Two main approaches have been identified in the literature. *Allocation* methods reduce the resource control problem in multiprocessor systems to solving the uniprocessor blocking problems on individual processors. Remote blocking is not permitted and so can be removed from consideration. *Concurrent* methods attempt to provide solutions to resource allocation for a truly concurrent system. Remote blocking is permitted in these methods and so must be bounded and calculable.

6.1. Allocation

The allocation approach has two main variants: static and dynamic. The former pre-allocates processes and their required resources onto the same processor. This is seen in the MARS kernel [Dam89]. In contrast, the dynamic variation permits a process to be allocated at runtime to a processor containing sufficient available resource for the process to execute. This approach has been proposed by Stankovic *et al* [Sta89] for use in the Spring kernel [Sta87a].

These approaches suffer from a number of weaknesses. Firstly, the requirement that all the resources that a process requires be resident on a single processor is restrictive and inflexible. If a process requires access to two resources, each physically constrained to be resident on diverse processors, it is unclear how these methods can cope. This is not a contrived situation as often physical resources (such as disk drive, actuator controllers, sensors etc) will be physically connected to different processors. These approaches constrain application designers to ensure processes do not access resources that are on separate processors.

Secondly, no processes are permitted to engage in synchronous communication with a remote process. This is due to another restriction in the Spring and MARS kernels: once a process commences execution its resources must be available. Both kernels permit message passing, but only asynchronously.

The major advantage of the allocation approach is that remote blocking is effectively removed from schedulability considerations.

6.2. Blocking

Blocking approaches allow processes to access resources on different physical nodes. This introduces remote blocking into the system. To be of use for hard real-time systems, this blocking needs to be bounded.

Three multiprocessor extensions to the priority ceiling protocol are now examined. The first and second deal with multiprocessor architectures with no shared memory. These approaches, in an abstract sense, migrate processes to remote resources. The third extension considers those systems that have shared memory. This approach does not migrate processes.

6.2.1. Multiprocessor Priority Ceiling Protocol

We have seen that for a single processor system local blocking can be bounded by use of the PCP. However, the PCP does not translate easily to a multiprocessor environment. Indeed, the underlying principle of PCP, namely priority inheritance, does not provide an adequate solution for multiprocessor systems.

Consider the following example (drawn from [Raj88a]). Processes τ_1, τ_2 and τ_3 are allocated processor **P1** with τ_4 assigned to **P2** (priority $\tau_1 > \tau_2 > \tau_3 > \tau_4$). Process τ_3 requests and is allocated resource **R**. Subsequently τ_4 makes a remote request for **R** and becomes blocked: all execution on **P2** has halted. Now τ_2 and then τ_1 preempt τ_3. Hence τ_4 is remotely blocked for a function of the execution times of τ_1 and τ_2 as well as the time τ_3 executes inside its critical region. Enforcing priority inheritance or the PCP will not reduce the remote blocking time that τ_4 endures, or the idle time enforced onto **P2**.

The MPCP bounds remote blocking time to be a function of the critical regions of other processes, not the time spent executing outside critical regions. To achieve this, the MPCP forbids situations where a process is blocked waiting for a resource on a processor, whilst another process executes outside its critical region on the same processor. For this purpose, resources are subdivided thus:

(i) *local resources* - those that only local processes access.

(ii) *global resources* - those that can be accessed by processes on different processors.

Processes are assumed to have static priorities, assigned globally in a rate-monotonic manner. A static priority preemptive scheduler is assumed. Each resource is assigned a priority ceiling in a similar manner as PCP. The ceiling of a local resource is equal to the priority of the highest priority process that accesses that resource. The ceiling of a global resource is defined to be higher than the priority of the highest priority process in the system. This is achieved by selecting a base global ceiling priority which is greater than the priority of all processes in the system. Each global resource is assigned a ceiling priority equal to the base priority plus the priority of the highest priority process that accesses that global resource. Two important observations regarding ceilings of global resources are now given:

(i) they are higher than any process priority;

(ii) when comparing the ceiling priority of two global resources **R1** and **R2**, then **R1** has a higher ceiling if it is accessed by process with higher priority than all processes accessing **R2**.

By such assignment, no process can be blocked remotely by a process executing outside a critical region.

The MPCP imposes a number of architectural limitations. Firstly, the system consists of a single *synchronisation processor* together with one or many *application processors*. Application processes are allocated statically to the latter. Where a resource is accessed by processes on different processors, that resource, together with associated critical regions, is placed onto the synchronisation processor. Such critical regions are termed *global critical regions*. Processes accessing global critical regions are not permitted to make nested critical region calls. Critical regions on application processors are not permitted to make nested calls to global critical regions.

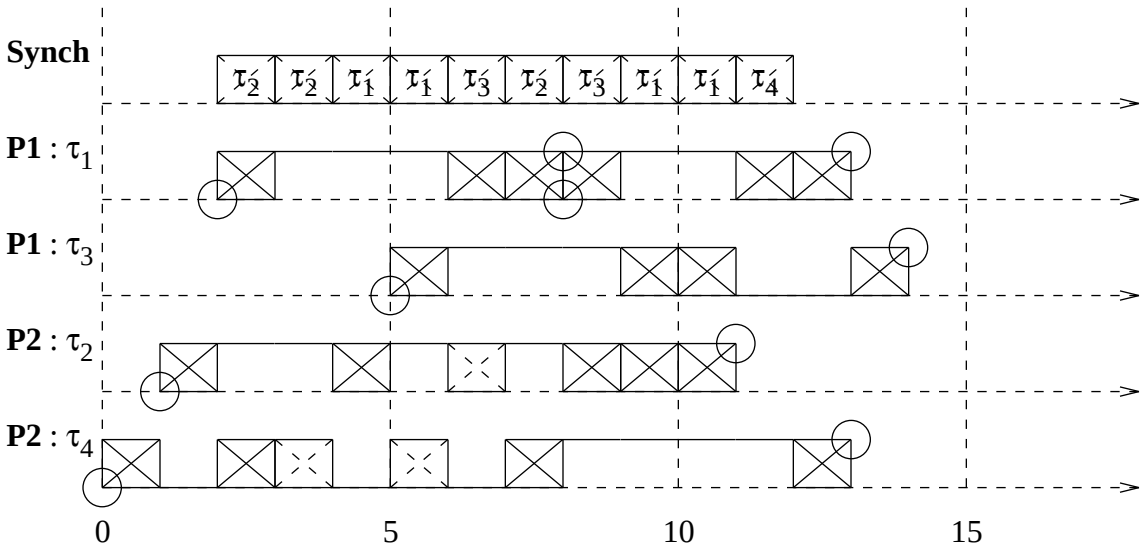
The runtime behaviour of the MPCP is defined to be the PCP on an application processor. When a process requests access to a global critical region, it becomes suspended on the local processor (abstractly the process migrates to the synchronisation processor). The effect of this is to allow lower priority processes to run and lock local semaphores without reference to the suspended process. On the synchronisation processor, the PCP is used with one modification: when a global critical region is requested by a process, it is allocated if it has a higher ceiling priority than all currently occupied global critical regions on the synchronisation processor. That is, resource allocation is made using the priority ceiling of the resource and not the priority of the process, as in the PCP. This is to allow for assigning all global critical regions priority ceilings higher than the priority of all processes. An example illustrating the behaviour of the MPCP is given in Figure 9.

The properties of the MPCP proved in [Raj88a] are that deadlock is avoided and blocking can be bounded. The blocking time that is endured by process τ_i is the sum of the local and remote blocking times. The local blocking time is exactly the same as for the PCP: each process can be blocked for the longest critical region of all lower priority processes on each execution. Remote blocking is heavily dependent upon the number of global critical region requests that τ_i performs. It is the sum of the following four factors [Raj88a]:

(i) *blocking due to execution of global critical section*
When τ_i requests a global critical section, it is suspended on its local processor. A local low priority process can now run and get a local semaphore that will be required by τ_i. Thus, τ_i can be blocked for the longest critical region of all lower priority processes for each global critical region request it makes. This is seen in Figure 9 at time 5 when τ_2 requires a resource (**R**) locked when it executed a global critical region on the synchronisation processor.

(ii) *blocking due to lower priority process executing on the synchronisation processor*
This occurs when τ_i requests a global critical section but becomes blocked because a lower priority process is executing a global critical region with equal or higher global ceiling priority. By the rules of the MPCP, τ_i cannot preempt the lower priority process and so becomes blocked. This can occur once for every global critical region required by τ_i.

(iii) *blocking due to higher priority processes executing on the synchronisation processor*
This can occur for every access by τ_i of a global critical region. Whilst executing inside the region, it can be preempted by a higher priority process requesting a global critical section with greater ceiling priority. The blocking factor encountered is bounded by the frequency that higher priority processes access global critical region with higher ceiling priorities. This is illustrated in Figure 9 at time 8 when τ_4 requests a global critical region (**GR2**) but is blocked



System has three processors: synchronisation processor **Sync** *and application processors* **P1 P2**
Global resources : **GR1** *shared by* τ_1 *and* τ_2, **GR2** *shared by* τ_3 *and* τ_4
Priority ceiling of **GR1** *higher than* **GR2**
Local resource **R** *resident on* **P2** *shared by* τ_2 *and* τ_4 *: priority ceiling 2*

Noteworthy Events In Execution

t = 2 : τ_2 requests and is allocated remote resource **GR1**; τ_2 executes on **Sync**
τ_4 executes on **P2**
t = 3 : τ_1 requests **GR1** but remotely blocked by τ_2; τ_4 requests and is allocated **R**
τ_2 completes execution on **Synch** and unlocks **GR1**
t = 4 : τ_1 allocated **GR1** and executes on **Synch**; τ_2 executes on **P2**
t = 5 : τ_1 executes on **Sync**; τ_2 requests **R** but blocked
τ_4 executes and completes critical region unlocking **R**
t = 6 : τ_3 requests and granted **GR2** and executes on **Sync**
τ_1 executes on **P1** τ_2 allocated **R** and executes on **P2**
t = 7 : τ_2 unlocks **R** and requests **GR1**. Priority ceiling **GR1** > **GR2** so
τ_2 preempts τ_3 on **Sync** and executes
t = 8 : τ_2 unlocks **GR1**, τ_3 executes on **Sync**
τ_4 requests **GR2** but denied and becomes blocked
t = 9 : τ_3 unlocks **GR2**, τ_1 requests **GR1** and executes on **Sync**
τ_4 remains blocked waiting to access **GR2** on **Sync**
t = 10: τ_1 completes critical region and unlocks **GR1** on **Sync**
t = 11: τ_4 allocated **GR2** and executes on **Sync**

Figure 9. Multiprocessor PCP

by a process executing in a global critical region with a higher ceiling.

(iv) *the effects of a higher priority process deferring its execution time*
This can occur when a higher priority process on the same processor is blocked remotely. The process is suspended. There now exists the potential for this process to execute at the end of its period and then execute again at the start of its next period, thus inflicting twice the blocking on τ_i. Lower priority processes, such as τ_i, have to allow for the maximum time that higher priority processes can suspend themselves. This can be seen in the executions of τ_1 and τ_3 in Figure 9.

One disadvantage with this approach is that critical regions must be kept small, otherwise blocking times can become large. Blocking times are also greatly effected by the number of remote requests. A second disadvantage is architectural, namely the specification of a single dedicated processor for global critical regions. In addition, it is unclear what the implementational overheads of migrating (part of) a process state to the synchronisation processor whilst a global critical region is accessed.

6.2.2. Generalised Multiprocessor Priority Ceiling Protocol

The Generalised Multiprocessor Priority Ceiling Protocol (GMPCP) is a development of the MPCP [Raj88a]. The GMPCP weakens the fundamental architectural constraints imposed by the MPCP:

- many synchronisation processors are permitted;
- application processes can be allocated to synchronisation processors.

Whilst many synchronisation processors are permitted, global critical sections are not permitted to make nested global critical section calls to other processors. This is to prevent excessive remote blocking times that could arise if global critical section performed a call to a global critical section on a different processor, which then performed a call to a global critical section on yet another processor. It is clear that blocking times can become very large if this is permitted.

Application processes allocated to synchronisation processors can be preempted by any process executing a global critical section. This can lead to priority inversion. For example, a high priority process allocated to a synchronisation processor can be preempted by a low priority process resident on another processor making a remote call to a global critical section on the synchronisation processor.

The GMPCP is bounded in the same manner as the MPCP. However, an additional blocking factor has to be included for application processes resident on a synchronisation processor. This is because the application process can be preempted by any other process calling a global critical section. The additional blocking factor is equivalent to the sum of all global critical region accesses made by all other processes within the period of the application process.

6.2.3. Shared Memory Protocol

The Shared Memory Protocol (SMP) [Raj90] is an extension to the MPCP for tightly-coupled multiprocessor architectures where processors share memory. In the MPCP, all global critical regions that have a common resource are executed on a common synchronisation processor. The SMP uses the presence of shared memory to allow some of the architectural constraints imposed by the MPCP to be relaxed:

- global critical regions can be executed on any processor;
- no distinction between application and synchronisation processors: both global critical regions and application processes can be allocated to the same processor.

The SMP executes global critical regions on the processor of the requesting process, rather than the separate synchronisation processor specified by the MPCP and GMPCP. In common with the latter two protocols, a process always executes at the ceiling priority when executing inside a critical section to enable remote blocking to be bounded to critical section computations only.

The assignment of global and local resource priority ceilings is essentially unchanged from the MPCP and GMPCP. However, each process assigns its own ceiling to global resources. A global resource is assigned base priority, greater than the highest priority of all system processes. This is raised by a value equal to the priority of the highest priority process on remote processors that accesses that resource. Whenever a process locks a global resource, it assumes the priority ceiling of that resource.

The motivation behind the global ceiling assignment rule is that a process which has locked a global resource cannot be preempted by a local process requesting the same resource. Therefore, only remote processes need to be considered in the derivation of the global ceiling. Also, a process executing a global critical section cannot be preempted by a local process outside a critical region (as all other processes will have lower priorities). The implication of the rule is that, potentially, all processors have different priority ceilings for the global semaphores (although all statically calculable offline).

The SMP avoids deadlock and bounds blocking. Indeed, the blocking times of a process for SMP and MPCP are similar. The local blocking is identical (as defined by the PCP). The remote blocking is made up of the four blocking factors of the MPCP, with two additional factors due to permitting global critical regions to be executed on any processor [Raj90]:

(i) *blocking due to the preemption of global critical regions*
A process that is blocked on a critical region, held by a process on another processor, can experience additional blocking due to a higher priority process preempting that process. This occurs if the higher priority process requested a global semaphore with higher priority ceiling. Such blocking can be bounded on the frequency of higher priority processes requesting global critical regions.

(ii) *blocking due to lower priority processes executing global critical regions*
Whenever a high priority process requests a global critical region it can suspend allowing lower priority processes to execute and enter global critical regions. Thus when the higher priority process finishes the global region call, it will be preempted by lower priority processes executing global critical regions.

The SMP suffers from higher blocking than the MPCP, but permits greater architectural flexibility.

6.3. Summary

Two general methods of resource allocation in multiprocessor systems have been examined: allocation and blocking. The former restricts resource considerations to be pre-scheduled, along with all processes, offline. Blocking is avoided, although the approach is inflexible and restrictive. The latter method considers techniques for dynamically allocating resources amongst competing processes on multiprocessors whilst maintaining boundable local and remote blocking.

Three blocking multiprocessor resource allocation schemes have been examined: MPCP, GMPCP and SMP. All three are attempts to define the PCP for a multiprocessor environment. The MPCP is defined only for distributed systems where there is no shared memory. The critical sections associated with resources accessed by processes on different processors are executed on a common synchronisation processor. This method is oriented toward a message-based system architecture. The GMPCP represents a relaxation of the MPCP where architectural restrictions regarding the allocation of global critical regions are relaxed. The SMP is defined for tightly-coupled systems with shared memory. It permits efficient implementation as the overheads of message-based communication in a shared memory system are avoided. All three protocols effectively define the CSP for shared and no-shared memory multiprocessor architectures as the priority of a process entering a global critical section is immediately raised to the global ceiling.

Blocking factors for the protocols can be excessive, particularly in terms of remote blocking. This is due to the pessimism that is implicitly used when attempting to place a worst-case blocking bound upon a process. This would seem to indicate that remote-blocking is expensive in terms of the schedulability of the system.

It is unclear what the exact overheads of the blocking methods are. Firstly, the MPCP and GMPCP demand that processes, or at least their state, is migrated to a synchronisation processor. With the SMP, it is unclear how physical resources can be moved between processors. For example, physical devices are usually connected to a single processor with the device drivers also associated with that processor. Here, the resource (device driver) cannot be arbitrarily moved between processors. Hence, in practice, the SMP may need to constrain the execution of some global critical regions to be confined to a single processor.

The MPCP, GMPCP and SMP require a static process priority system. All three however appear extensible for use with dynamic priorities by utilising the approach of Baker (see section 4.1.5). Process deadlines can be used to assign static preemption levels for processes, with priority ceilings set using those preemption levels.

A comparison between the allocation and blocking approaches is difficult. Inflexibility is imposed by the former, whilst large blocking times are a result of the latter. Hence a tradeoff exists between the difficulty of searching for a static schedule that satisfies all resource requirements and the performance degradation of runtime blocking.

7. DISCUSSION

This review has described many resource control approaches for hard real-time systems implementable on a variety of hardware architectures. A comparative discussion of these approaches is now presented. This can be viewed as a framework for selecting a resource control approach for a system of given architecture, design characteristics and required behaviour. We proceed by identifying and discussing the key issues contained in the preceding review.

Can systems be designed using blocking or non-blocking resource control?

The choice between a non-blocking or blocking approach to resource control is complex. One consideration is architectural. To build a system using a non-blocking approach demands that all resources within that system are utilised in a non-blocking manner. This may not always be possible. For example, to access a disk-drive in such a manner could be difficult, since there is a definite delay between the request and reply. Thus, the drive may have to be locked, so denying access to other processes.

A second example is a simple A/D converter. Here, a time delay between the request to sample and digital reply exists (due to the propagation delay of the converter hardware). It is not possible to permit other processes to access the device whilst conversion is occurring, else the reply may be invalidated.

Counter arguments exist: such devices can be made to appear non-blocking. With this approach a possibility exists that software layers are added between device and logical process in order that devices appear non-blocking by using, for example, Simpson's 4SM (see section 5.2). This can increase the complexity of software design. For example, consider a single A/D converter sampling 3 unrelated analogue channels. The digital output from each channel is handled by functionally different code. To represent such a system using 4SM requires the three diverse functions to be embedded into one process. This is contrary to the software engineering principle of decomposition. The alternative is to use a software device driver which continually samples the three channels, placing the readings into buffers, each of which is read by a different process. The channels are effectively polled. This approach can be inefficient.

It is probable that systems can be designed to use either blocking or non-blocking resource control. The choice is largely dependent upon exact requirements and system architecture. However, hybrid control, utilising aspects of both blocking and non-blocking, is perhaps a more pragmatic approach, applicable to many hard real-time system requirements.

Is remote blocking useful and necessary for hard real-time multiprocessor systems?

When multiprocessor architectures are considered, the problem of blocking is exacerbated. This is due to the possibility of remote blocking. One approach toward solving this problem is the exclusion of remote blocking. This is adopted by the scheduling schemes of Fohler[Foh89] and Stankovic [Sta89]: all resources required by a process are local. Indeed, required resources are always available when a process is released (see section 5.1). Any remote operations are constrained to be via non-blocking message-passing.

Whilst such approaches circumvent the difficulties imposed by the presence of remote blocking, design restrictions are introduced. It may be difficult, from a design viewpoint, to develop multiprocessor systems where some resources are physically constrained to given processors. The system has to ensure that all processes using a bound resource are allocated to that processor.

Other problems are introduced by the exclusion of remote blocking. For example, consider a process which has a replica for fault-tolerant purposes. The original and replica must be placed onto diverse processors. The process uses a single resource, resident on a third processor. Hence, the resource, the replica and the original are allocated to different processors for fault-tolerant reasons. The original and replica processes can only access the resource remotely. This implies the presence of remote blocking if the resource is accessed in a mutually exclusive manner.

Remote blocking introduces inefficiencies into a system. However, without it, it is unclear whether hard real-time systems, in the general case, can be designed. Undoubtedly, for efficiency reasons, remote resource accesses should be minimal. This can be achieved by realising the cost of remote blocking whilst both designing and implementing the system (specifically allocating processes and resources to processors). At runtime, the bounded resource control approaches (e.g. GMPCP) can be utilised whenever remote resource accesses are required.

What are the effects on system schedulability of resource control?

The underlying requirements of resource control for hard real-time systems are boundedness and predictability (see section 1). The simplest way to meet these criteria is by adopting a non-blocking approach. In this review, two such approaches have been described.

Firstly, Simpson's approach (see section 5.2) dictates that any resource has at most one reader process and one writer process. The approach is (possibly) extensible to permit many readers, although the expansion to many writers would introduce blocking. Simpson's approach has no real implication for schedulability: no direct restrictions are imposed on choice of priority model (i.e. static or dynamic) or scheduling mechanism (i.e. preemptive or non-preemptive). One possible problem is introduced if limitations are placed on the reader and writer processes of a resource. For example, there may be a requirement that a piece of data be less than a given age when read. Schedulability must take account of this requirement by adjustment of process timing characteristics or the introduction of precedence constraints.

Secondly, non-blocking can be introduced by searching for a schedule such that, although processes share resources to which they require mutually exclusive access, a process will never request a resource that is already locked. This is seen in the work of Fohler [Foh89] and Stankovic [Sta87b] where heuristic search methods are utilised (see section 5.1). Schedulability of these approaches is defined as the finding of a valid schedule. This becomes increasingly difficult as the ratio of length of resource access to process execution time increases. In the extreme, all processes require the same resource for their entire execution. Thus non-preemptive scheduling is required with associated increase in schedulability complexity [Yua89, Jef88]. Also search-space is exponential in the number of processes and resources. Hence for systems with large numbers of processes and resources, a substantial amount of time is required to determine schedulability.

Boundedness and predictability can also be achieved if a blocking approach is adopted. The PCP and derivatives (see section 4.1) place pessimistic (but bounded) worst case estimations on the blocking that a process will experience at runtime. This is applicable to both static priority (PCP) and dynamic priority systems (DPCP, SRP).

The approaches define a maximum blocking time. This can be added to the worst-case execution time of a process for schedulability purposes. Hence, schedulability of a system is not determined by considering resources directly, only the adjusted computation times. Schedulability of multiprocessor systems, incorporating remote blocking, is similar in nature. Blocking times are added to worst-case execution times of processes, although blocking due to remote resources may be large compared to local resources.

Generally, the effects of introducing resource control into a system increases the complexity of determining schedulability. However, this complexity should not impinge upon design decisions regarding the type of resource accesses permitted (i.e. blocking of non-blocking) since although determining schedulability can be complex and time-consuming, it can be performed offline, however, the implication of design decisions are felt throughout the lifetime of the system.

Which resource control approach to use?

Without specific timing and behavioural characteristics for a hard real-time system, it is very difficult to be prescriptive in terms of recommending a particular resource control approach. However, a summary of choices, reflecting the assumptions and properties of the reviewed approaches is presented in Figure 10.

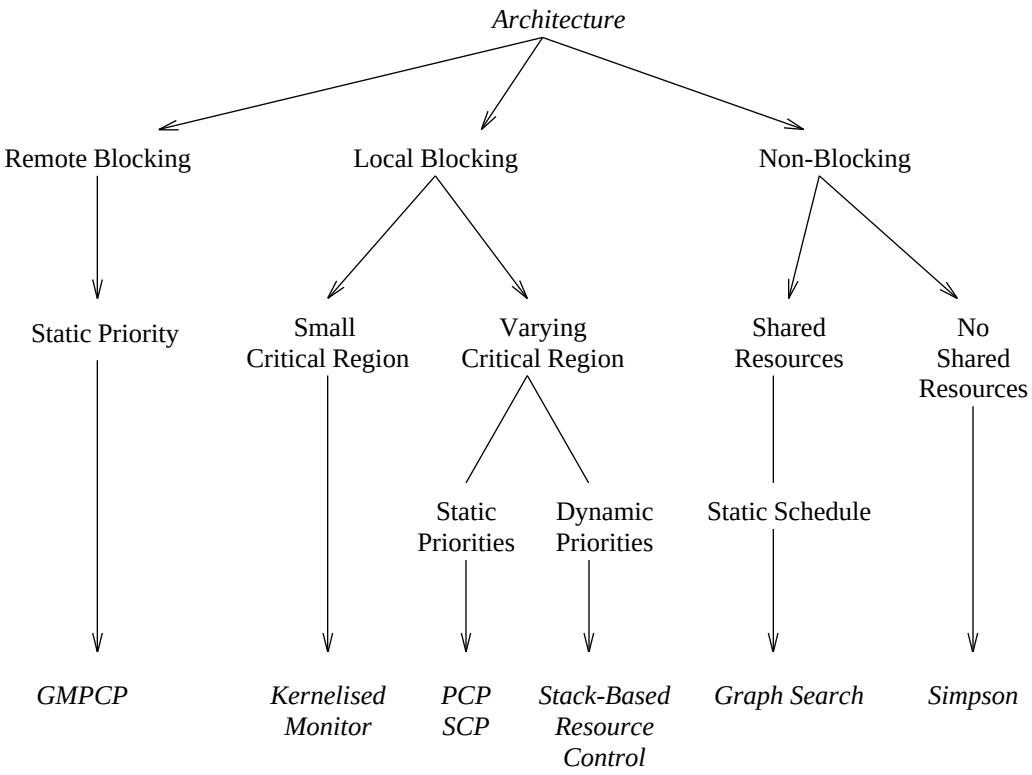


Figure 10. Choice of Resource Control Approach

The Figure assumes that architectural decisions such as whether to permit local blocking or remote blocking have been made. We do not contend that the Figure covers all possible resource control approaches, merely those reviewed positively in this paper.

8. CONCLUSIONS

Any resource control method for hard real-time systems must be predictable and bounded in terms of the time that a process waits for a resource to be unlocked. These conditions enable arguments regarding the schedulability of processes to be made. Many control techniques meeting these criteria have been reviewed, for blocking and non-blocking paradigms and for a variety of hardware architectures.

The discussion provided in the previous section draws out the major issues highlighted throughout the review. A framework for choosing a resource control method was presented. It was seen that the reviewed resource control methods do not provide a complete solution for hard real-time system resource control. The weaknesses and exclusions of the methods form a number of open research questions:

(a) *worst-case blocking time estimation*
blocking times calculated for use in schedulability are inherently pessimistic (e.g. the PCP). Such estimations need to be more accurate in much the same way as greater accuracy is required from worst-case timing analysis research.

(b) *remote blocking is expensive*
abstract architectures may enable remote blocking to be removed (i.e. expanding

Simpson’s approach and Mascot-3 to multiprocessor). However, it is unclear if hard real-time systems can be designed without a requirement for remote blocking. Currently, bounds placed remote blocking are large and pessimistic. Either a greater degree of accuracy is required from these bounds, or else better mechanisms for locking remote resources are needed.

(c) *condition synchronisation is not permitted*
in the current generation of resource control approaches, mutual exclusion is the motivating goal. However, many applications require a richer variety of inter-process interaction, for example condition synchronisation on the state of a buffer; time coherence on the age of a datum etc. These requirements need to be reflected in resource control for general hard real-time systems.

Resource control approaches must address these issues if the usefulness of hard real-time systems, in particular those designed for multiprocessor architectures, is to be exploited.

REFERENCES

Aud90a. N. C. Audsley, ‘‘Deadline Monotonic Scheduling’’, YCS 146, Department of Computer Science, University of York (October 1990).

Aud90b. N. C. Audsley and A. Burns, ‘‘Scheduling Real-Time Systems’’, YCS 134, Department of Computer Science, University of York (1990).

Bab90. O. Babaoglu, K. Marzullo and F. B. Schneider, ‘‘Priority Inversion and its Prevention in Real-Time Systems’’, TR-90-1088, Deptartment of Computer Science, Cornell University (March 1990).

Bak90a. T. Baker, ‘‘Protected Records, Time Management and Distribution’’, *ACM ADA Letters* **X**(9), pp. 17-28 (Fall 1990).

Bak90b. T. P. Baker, ‘‘A Stack-Based Resource Allocation Policy for Realtime Processes’’, *Proceedings 11th IEEE Real-Time Systems Symposium*, IEEE Computer Society Press (December 1990).

Bat86. G. Bate, ‘‘Mascot-3: An Informal Introductory Tutorial’’, *Software Engineering Journal* **1**(3), pp. 95-102 (May 1986).

Bur90. A. Burns and A.J. Wellings, ‘‘The Notion of Priority in Real-time Programming Languages’’, *Computer Languages* **15**(3), pp. 153-162 (1990).

Che89. M. I. Chen and K. J. Lin, ‘‘Dynamic Priority Ceilings: A Concurrency Control Protocol For Real-Time Systems’’, Report No. UIUCDCS-R-89-1511, Department of Computer Science, University of Illinois at Urbana-Champaign (April 1989).

Dam89. A. Damm, J. Reisinger, W. Schwabl and H. Kopetz, ‘‘The Real-Time Operating System MARS’’, *ACM Operating Systems Review* **Special Issue**, pp. 141-157 (1989).

Der89. M. L. Dertouzos and A. K. L. Mok, ‘‘Multiprocessor On-Line Scheduling of Hard Real-Time Tasks’’, *IEEE Transactions on Software Engineering* **15**(12), pp. 1497-1506 (December 1989).

Foh89. G. Fohler and C. Koza, ‘‘Heuristic Scheduling for Distributed Real-Time Systems’’, Research Report Nr. 6/89, Instiut fur Technische Informatik, Technische Universitat Wien, Austria (April 1989).

Har90. J. R. Haritsa, M. J. Carey and M. Livny, ‘‘Dynamic Real-Time Optimistic Concurrency Control’’, *Proceedings 11th IEEE Real-Time Systems Symposium*, Lake Buena Vista, FL, USA, pp. 94-103 (5-7 Decmeber 1990).

Jef88. K. Jeffay, ‘‘On Optimal, Non-Preemptive Scheduling of Periodic Tasks’’, Technical Report 88-10-03, University of Washington, Department of Computer Science (October 1988).

Kle90. M. H. Klein and T. Ralya, ‘‘An Analysis of Input/Output Paradigms for Real-Time Systems’’, CMU/SEI-90-TR-19, SEI, Carnegie-Mellon University (July 1990).

Leh89. J. Lehoczky, L. Sha and Y. Ding, ‘‘The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour’’, *Proceedings IEEE Real-Time Systems Symposium*, Santa Monica, California, pp. 166-171, IEEE Computer Society Press (5-7 December 1989).

Lis84. A. M. Lister, ‘‘Fundamentals of Operating Systems’’, 3rd. Edition, Macmillan Computer Science Series (1984).

Liu73. C. L. Liu and J. W. Layland, ‘‘Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment’’, *Journal of the ACM* **20**(1), pp. 40-61 (1973).

Mok83. A. K. L. Mok, ‘‘Fundamental Design Problems of Distributed Systems For The Hard Real-Time Environment’’, MIT/LCS/TR-297, Laboratory of Computer Science, Massachsetts Institute of Technology (1983).

Pil90. M. Pilling, A. Burns and K. Raymond, ‘‘Formal Specifications and Proofs of Inheritance Protocols for Real-Time Scheduling’’, *Software Engineering Journal*, pp. 263-279 (September 1990).

Raj90. R. Rajkumar, ‘‘Real-Time Synchronisation Protocols for Shared Memory Multiprocessors’’, *Proceedings 10th IEEE International Conference on Distributed Computing Systems*, Paris, IEEE Computer Society Press (28 May - 1 June 1990).

Raj88a. R. Rajkumar, L. Sha and J. P. Lehoczky, ‘‘Real-Time Synchronisation Protocols for Multiprocessors’’, *Proceedings IEEE Real-Time Systems Symposium*, pp. 259-269 (December 1988).

Raj89. R. Rajkumar, L. Sha and J. P. Lehoczky, ‘‘An Experimental Investigation of Synchronisation Protocols’’, *Proceedings 6th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 11-17 (May 1989).

Raj88b. R. Rajkumar, L. Sha, J. P. Lehoczky and K. Ramamithram, ‘‘An Optimal Priority Inheritance Protocol for Real-Time Synchronisation’’, COINS Technical Report 88-98, Deptartment of Computer and Information Science, University of Massachusetts (October 17, 1988).

Sha87. L. Sha, R. Rajkumar and J. P. Lehoczky, ‘‘Priority Inheritance Protocols: An Approach to Real-Time Synchronisation’’, CMU-CS-87-181, Computer Science Department, Carnegie-Mellon

University (December 1987).

Sha90. L. Sha, R. Rajkumar and J. P. Lehoczky, ‘‘Priority Inheritance Protocols: An Approach to Real-Time Synchronisation’’, *IEEE Transactions on Computers* **39**(9), pp. 1175-1185 (September 1990).

Sim90. H. Simpson, ‘‘Four-Slot Fully Asynchronous Communication Mechanism’’, *IEE Proceedings Part E* **137**(1), pp. 17-30 (Jan 1990).

Sta87a. J. A. Stankovic and K. Ramamritham, ‘‘The Design of the Spring Kernel’’, *IEEE Proceedings Real-Time Systems Symposium*, pp. 146-157 (1987).

Sta87b. J. A. Stankovic, K. Ramamritham and W. Zhao, ‘‘Preemptive Scheduling Under Time and Resource Constraints’’, *IEEE Trans Computers* **38**(8), pp. 949-960 (August 1987).

Sta89. J. A. Stankovic, K. Ramamritham and W. Zhao, ‘‘Distributed Scheduling of Tasks with Deadlines and Resource Requirements’’, *IEEE Transactions on Computers* **38**(8), pp. 1110-1123 (August 1989).

Yua89. X. Yuan and A. Agrawala, ‘‘A Decompositional Approach to Non-Preemptive Real-Time Scheduling’’, *Proceedings IEEE Real-Time Systems Symposium* (December 1989).