A Fixed-Point Approach to Bounding Blocking in Real-Time Systems

T.P. Baker*
Department of Computer Science
Florida State University
Tallahassee, FL 32304-4019

July 5, 1991

Abstract

This paper presents an allocation policy for nonpreemptible resources in a real-time environment, which imposes a set of blocking conditions that are minimally adequate to bound the length of time a higher priority job may be forced to wait while a lower priority job executes. The blocking conditions are defined as the fixed point of a set of symmetric recurrences. This policy generalizes the "semaphore control protocol" of Rajkumar *et. al.*[5], by extending it to more abstract nonpreemptible resources, of which semaphores and reader-writer resources are special cases.

1 Introduction

This paper describes a minimal allocation policy for nonpreemptible resources in a real-time environment. The intent of this policy is to impose blocking conditions that are minimally adequate to assure bounded "impedance" — that is, to bound the length of time a higher priority job may be forced to wait while a lower priority job executes.

This is an outgrowth of the work of Sha, Rajkumar, and Lehoczky [6], in which they define the "priority ceiling protocol" — a resource allocation policy that assures "bounded blocking". Given such a bound, Liu and Layland's [4] analysis of the worst case for rate-monotone preemptive scheduling is applied to derive schedulability conditions for a set of tasks. The priority ceiling protocol is more restrictive than necessary to achieve bounded blocking. Rajkumar, Sha, and Lehoczky [5] describe a less restrictive, "optimal", resource allocation policy based on a refinement of the notion of priority ceiling. they call the optimal policy the "semaphore control protocol". We call it the SCP, for short.

Working independently, this author also extended the results of [6], discovering the optimal resource allocation policy by a different route. This route abandons the notion of priority ceiling, and seeks to derive a policy by more direct means, that are known *a priori* to

*This work supported in part by grant N00014-87-J-1166 from the U.S. Office of Naval Research.

produce a minimal set of restrictions. Specifically, the policy is defined as the least fixed-point solution to a set of recursive conditions, that must be satisfied by any policy that bounds impedance. This fixed-point policy is called the "minimal blocking policy", or MBP for short.

If both the fixed-point approach and that of [5] lead to policies that are optimal in the same sense, they must be equivalent, and they are. However, the formulation of the MBP policy in this paper is still of interest for the following reasons:

- It is more general, applying to resources other than semaphores, such as data shared by readers and writers.
- The fixed-point approach more clearly demonstrates why the policy is minimal.
- It corrects a circularity problem with the definition of the SCP.
- The defining recurrences expose symmetry in the blocking conditions, that is not so apparent in the SCP.
- Properties due to the processor allocation policy (priority inheritance) are separated from those due to the resource allocation policy.
- It is defined in a form that appears to be better suited for static (i.e., compile-time) analysis to eliminate code for resource requests that can be predicted to be non-blocking.

Section 2 introduces the notation and terminology used in the rest of the paper, as well as the basic model of jobs and nonpreemptible resources. Section 3 distinguishes the role of processor allocation policies from the role of nonpreemptible resource allocation policies, establishes some basic assumptions about nonpreemptible resource management, and explains the priority inheritance processor allocation policy. Section 4 explains the difference between impedance and blocking, and the need for bounding the number of times a job may be blocked, if its impedance is to be bounded. Section 5 states and proves a set of necessary conditions for bounded blocking. Section 6 defines the MBP, as a least fixed-point satisfying the necessary conditions for bounded blocking, and Section 7 proves that the fixed-point is reached in one iteration. Section 8 shows that the MBP has the bounded impedance property, as desired. Section 9 shows that when a resource allocation is requested, the MBP blocking conditions need only be checked against a single outstanding allocation, so that the method is practically implementable. Section 10 compares the MBP to the SCP, showing that it is a consistent generalization. Section 11 is a summary, and discussion of further research.

2 Notation and Definitions

A *job* is a finite sequence of instructions to be executed on a single processor. It *arrives* at some time, after which it can begin execution. Jobs that have arrived and have not yet

completed are called *pending*. Two jobs are said to be of the same *type* if they have the same sequence of instructions. Names of the forms J,J', J'',... and J_i always denote jobs.

The execution of a job requires the use of a processor, and may require certain other resources. We assume there is a single processor, which is preemptible, and several *nonpreemptible resources*, R_1,..,R_m. Allocation of the processor and nonpreemptible resources to jobs is governed by *processor and resource allocation policies*, respectively. Names of the forms R and R_i always denote resources.

Every job belongs to one of a fixed finite set of *tasks*, τ_1,...,τ_n. Each task τ_i consists of an (infinite) sequence of jobs $J_{i,1}, J_{i,2}, ...$ that arrive at different times and are to be executed to completion in order of arrival. The jobs generated by each task are assumed to be of a fixed finite set of *types*, which are known *a priori*. Ideally, each job of a task should complete before the arrival of the next job of that task, but it may not. A task is *periodic* if the interval between successive arrivals of its jobs is a constant, called the *period*; otherwise, the task is *aperiodic*. Tasks may be periodic or aperiodic. Names of the forms τ and τ_i always denote tasks.

There is a fixed finite set of resource utilization *modes*. A job acquires a nonpreemptible resource in some mode by executing an instruction that requests an allocation of the resource to the job in that mode. Formally, an *allocation* is a triple $A=(J, R, m)$, where J is a job, R is a nonpreemptible resource, and m is a mode. The job making a request must wait to execute its next instruction until a matching allocation is granted. While it is waiting the job (and the request) are said to be *blocked*. After the allocation is granted, the job holds the allocation until the job executes an instruction that releases it. While some job holds an allocation the allocation is said to be *outstanding*.

Names of the forms A,A', A'',... and A_i always denote allocations (or requests). For any allocation A, J always denotes the corresponding job — i.e. the job holding the allocation, or requesting it. (This rule applies similarly for names of the other forms, like A' and J', and A_i and J_i.)

The sequence of instructions performed by the job between the request and release operations for an allocation is called a *critical section* of the job for that allocation.

Each job J has a *priority* $\pi(J)$. Priorities are values from some ordered domain, where J has higher priority than J' if and only if $\pi(J) > \pi(J')$; i.e. expediting job J is sufficiently important that completion of job J' is permitted to be delayed in order to expedite J. For concreteness in our examples, we will use integer priorities, where numerically greater values indicate greater urgency. Examples of possible priority assignments include rate-monotone, shortest processing time, earliest arrival, and earliest-deadline. Our definitions and basic results only require that the priority of each job be fixed at the time of its arrival, but later results assume that the priority of each type of job is fixed. The priority $\pi(A)$ of an allocation $A=(J, R, m)$ is the priority of the corresponding job, $\pi(J)$.

Example. Suppose there are four types of jobs $\{J_1, J_2, J_3, J_4\}$, with $\pi(J_i) = i$ (integer priorities, with larger numbers indicating higher priority, i.e. greater urgency), and three resources $\{R_1, R_2, R_3\}$. Suppose R_1 and R_2 are reader/writer resources, and R_3 is

J_1	J_2	J_3	
... request $(1, 1, w)$; ... release; ...	... request (2,2,w); ... request (2,3,l); ... request (2,1,r); ... release; ... release; ... release; ...	... request (3,1,r); ... release; ... request (3,3,l); ... request (3,2,r); ... release; ... release; ...	request (4,2,r) release;

Figure 1: The jobs, as instruction sequences.

a semaphore. The four types of jobs are shown in Figure 1, represented as schematic sequences, with all the instructions that do not affect resource allocations abstracted away. The notation "(i,j,k)" denotes a request from job J_i for an allocation of resource R_j, where k="r" denotes read mode, k="w" denotes write mode, and k="l" denotes locked mode. Note that the release operations have no explicit arguments. Since we are going to assume any critical sections that overlap must be properly nested, the implied argument is the last allocation granted to the job that executes the release instruction.

3 Policies

3.1 Resource Allocation

The *resource allocation policy* determines when to grant a requested allocation to a job. Thereby, it determines which of the pending jobs are blocked, and which are ready to continue execution. The resource allocation policy is constrained to block a request so long as there are conflicting allocations held by other jobs. We call such a conflict a *direct blockage*. Exactly what constitutes a direct blockage may vary with the type of resource, and is not critical to the design of a resource allocation policy. Let $BD(A, A')$ be a predicate that is true if and only if A will be *Blocked Directly* by A', if A is requested while A' is outstanding.

For example, if the resource is shared data, the modes may be {*read*, *write* }. A read request is blocked directly if and only if there is a outstanding write allocation on the same resource. A write request is blocked directly iff there is an outstanding read or write allocation on the same resource. Similarly, if the resource is a binary semaphore, the only mode is *write*, and a request is blocked directly if there is any outstanding allocation of the same resource in any mode.

	(1,1,w)	(2,1,r)	(3,1,r)	(2,2,w)	(3,2,r)	(4,2,r)	(2,3,l)	(3,3,l)
(1,1,w)		D	D					
(2,1,r)	D							
(3,1,r)	D							
(2,2,w)					D	D		
(3,2,r)				D				
(4,2,r)				D				
(2,3,l)								D
(3,3,l)							D	

Figure 2: The direct blocking relation.

Example. The direct blocking relationships for our example set of job types (in Figure 1) are shown by the occurrences of the letter "D" in the table of Figure 2.

Subject to the constraint that it must enforce direct blocking, the resource allocation policy should work with the processor allocation policy to expedite the highest priority pending job. In order to do this it may block some requests that are not blocked directly. An objective of this paper is to define a resource allocation policy that imposes the minimum non-direct blocking necessary to prevent a job from being blocked by more than one other job, or deadlocked.

3.2 Processor Allocation

The processor allocation policy determines which one of the pending jobs that are not blocked is allowed to use the processor. The primary objective of the processor allocation policy, along with the resource allocation policy, is to expedite the highest priority pending job. Normally, expediting the highest priority pending job means allocating the processor to it, so it can execute, but this is not possible when that job is blocked.

If a job is blocked directly, the only way to expedite it is to expedite the job that is directly blocking it, until that job releases the allocation(s) responsible for the blocking. This rule can be applied transitively to expedite any directly blocked job that is not involved in a deadlock.

This still leaves open the issue of what to do when the highest priority job is not blocked directly. This question can be answered independently of the resource allocation policy, if the resource allocation policy satisfies certain assumptions.

3.3 Blocking Assumptions

We will assume the following conditions on blocking are satisfied.

1. *(no direct self-blocking)*

Every direct blockage is caused by another job. That is, a job never blocks itself directly, by requesting an allocation that would be blocked directly if all allocations held by other jobs were released. For example, a job never requests a resource that is not present initially, or requests an allocation that is blocked directly by an allocation already held by the same job. This can be checked by *a priori* examination of the jobs, so that the resource allocation policy need not enforce it.

2. *(nested critical sections)*

   A job must exit all its critical sections before it can complete; i.e., every resource allocation is released before the end of the corresponding job. Also, if a job has more than one critical section, each pair of critical sections is either properly nested or completely disjoint; that is, resource are requested and released by each job in LIFO order. This can be checked *a priori*.

3. *(initially no allocations)*

   Initially, there are no outstanding allocations.

4. *(acquiring/releasing requires execution)*

   A job cannot acquire an allocation or become blocked until it is able execute a request, and it cannot release an allocation that it is holding without executing further; that is, these system state transitions require that the processor be allocated to the job. (Among other things, this implies that a blocked job cannot become unblocked until every job that is blocking it executes further.)

5. *(identifiable blockers)*

   If job J is blocked, directly or otherwise, it has requested some allocation A for which there is at least one allocation A', held by another job, that is identifiable as *blocking* A. J cannot become unblocked until at least one of these blocking allocations is released.

Note that the assumption of identifiable blockers imposes a requirement on the resource allocation policy, so we will need to be careful to verify that whatever resource allocation policy we choose satisfies this assumption independently of anything we prove using it.

3.4 Priority Inheritance Policy

Under the assumption of identifiable blockers, the only way to expedite a blocked job is to expedite some job that is blocking it. This rule can also be applied transitively.

Let the *top* job be the oldest among the highest-priority pending jobs. That is, if we list the pending jobs in decreasing order of priority, and then rank the jobs of equal priority in increasing order of arrival time, the top job is at the head of the list.

We now state a processor allocation policy that always expedites the top job. Let $Block(A, A')$ mean that a request A is blocked by an outstanding allocation A'.

The *priority inheritance policy* is to allocate the processor to job J only if either J is top job or there is a "chain" of blocked jobs leading from the top job to J. That is, if J_H is the top job, job $J \neq J_H$ can execute only if:

$$\exists n \underset{n \geq 1}{} \quad \exists J_1, ..., J_n \qquad [J_1 = J_H \ \wedge$$

$$\forall i \underset{1 \leq i < n}{} \quad Block(J_i, J_{i+1}) \ \wedge$$

$$Block(J_n, J)].$$

Let us call such a sequence of jobs $J_1, ..., J_n, J$, a *chain*, and n the *length* of the chain. Every blocked job is the tail of chain of length one. A special case of a chain is a *cycle*, in which $J_1 = J$. Since we are assuming a job never blocks itself, every cycle has length ≥ 2.

Note that if the top job is deadlocked, via a cycle in the chain headed by the top job, the processor is not allocated to any job. This policy will also deadlock if the top job does not have at least one identifiable blocker, but the blocker need not be unique. That is, there may be a "tree" of chains of jobs blocking the top job. In this case, the policy permits executing any of the "leaves" of this tree.

Finally, note that this definition does not imply that the priority of any job changes. The notation $\pi(J)$ always denotes a value that is constant after J arrives, for each J.

In what follows, we shall assume the priority inheritance policy is followed, and that the blocking assumptions are satisfied. We will need to be careful to prove that whatever resource allocation policy we choose satisfies the assumption of identifiable blockers, without relying on that assumption to prove it.

Lemma 1 (FIFO service within priorities) *A job cannot start to execute until all the jobs that arrived before it and have equal or higher priority have completed.*

Proof. A job cannot execute unless it is top job or is blocking some other job. If it is top job all the jobs that arrived before it and have equal or higher priority must have completed. If it is not top job, it must be holding an allocation, but it cannot hold an allocation since it has not started to execute — a contradiction. □

This lemma means that the processor allocation policy assures that jobs of equal priority can never be in a situation of competing for nonpreemptible resources. This fact is used frequently below. Some examples of consequences of this lemma are listed in Corollary 2, below.

Corollary 2 *The following are true:*

1. *No two jobs that have started executing have equal priority.*
2. *If A and A' are outstanding, and $\pi(A) = \pi(A')$, then $J = J'$.*
3. *If A is a request and A' is held by another job, then $\pi(A) \neq \pi(A')$.*

Lemma 3 *Suppose $\pi(J') < \pi(J)$. Then J' cannot execute between the time J arrives and the time J completes, unless J' is holding an allocation that it acquired before J arrived.*

Proof. Suppose J' holds no allocation when J arrives. Then J preempts J', and J' does not execute again until J completes.

Suppose J' holds an allocation when J arrives. Let A be the allocation that J' acquired first, among the allocations held by J' when J arrives. By the assumption that overlapping critical sections must be properly nested, J' cannot release A until it releases all the allocations it is holding. Once J' releases all its allocations, since J has higher priority, J' does not execute again until J completes. □

Lemma 4 *Suppose job J is blocked by job J'. One of the following cases must hold:*

1. *J' has lower priority than J, J' arrives before J, and J' holds an allocation that was granted before J arrived.*[1]
2. *J has lower priority than J', J arrives before J', and J holds an allocation that was granted before J' arrived.*

Proof. Suppose J and J' have the same priority. Neither of these jobs can preempt the other, so whichever arrives first will complete before the other can start to execute. Thus, neither can block the other — a contradiction.

Suppose J' has lower priority than J. By Lemma 1, if J arrives first J' will not start to execute until J completes. Thus, J' cannot obtain any allocation to block J — a contradiction.

By Lemma 3, if J' arrives first, J' cannot execute after J arrives unless J' is still holding an allocation that it acquired before J arrived.

Suppose J has lower priority than J'. By Lemma 1, if J' arrives first, J will not start to execute until J' completes. Thus, J cannot request any allocation, and J' cannot block J — a contradiction.

By Lemma 3, if J arrives first, J cannot execute after J' arrives unless J is still holding an allocation that it acquired before J' arrived. □

4 Bounding Impedance

We say a pending job is *impeded* when a lower priority job is executing, and *preempted* when a higher priority job is executing. Thus, at any instant each pending job is in one of three states:

1. *Executing.* The job is executing.

[1] Lemma 1 of [5].

2. *Impeded.* A lower priority job is executing.

3. *Preempted.* A higher priority job is executing.

Since we are assuming there is only one processor, at most one job is executing at any instant.

The *impedance* of a job is the time spent by the job in the impeded state. Impedance is an inverse measure of how well the processor and resource allocation policies honor task priorities.

Note that a job may be impeded for reasons other than blocking. In particular, with priority inheritance a medium priority job may be impeded by a low priority job while the low priority job is blocking a high priority job. This condition is called "inheritance blocking" in [6, 5]. We have chosen to introduce a new term, because we found the multiple meanings of "blocking" used in [6, 5] confusing. The term "blocking" is well established in the literature of operating systems as applying only to situations where a job is forced to wait for a resource other than the processor. Part of the analysis of the priority ceiling protocol and semaphore control protocol policies in [6, 5] is classifying the possible causes of impedance into various forms of "blocking", including "direct blocking", "inheritance blocking", and "avoidance blocking". Of these, only "inheritance blocking" does not fit our definition of blocking.

Bounding the worst-case impedance of jobs is a key to *a priori* task schedulability analysis. Sha, Rajkumar and Lehoczky [6] have refined and generalized Liu and Layland's [4] analysis of the worst case for rate-monotone preemptive scheduling to obtain schedulability conditions for a set of tasks with critical sections, under the following assumption:

Bounded Impedance Property: For each task τ_i there is a constant B_i that bounds the length of time any job of τ_i may be impeded.

A necessary condition for the bounded impedance property to be satisfied is that the execution times of individual critical sections must be bounded.

Lemma 5 (critical sections must be bounded) *If there is a constant B bounding the length of time a job J may be impeded, then for every job that has lower priority than J, the execution time (exclusive of preemptions and impedance) of every critical section that conflicts with some request of J must be bounded by B.*

Proof. Suppose job J' has lower priority than J, and J' requests an allocation A' that is held for longer than B time, and which conflicts with some request A of J. J may preempt J' just after it has acquired A'. When it requests A, J will be impeded for longer than B time — a contradiction. □

Given the execution time of each critical section is bounded, we can only assure the bounded impedance property by bounding the number of critical sections that may impede each job. We next characterize the situations which we must avoid if we want to assure each job is impeded by at most one critical section.

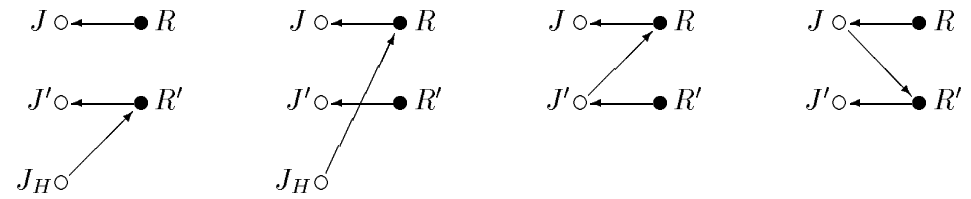


Figure 3: $Cover()$ and $HB()$

5 Necessary Blocking Conditions

Let us assume the resource management policy enforces direct blocking. We will define a set of other conditions under which a request must be blocked, if we are to avoid deadlock or a job being impeded by more than one other job.

Assume $Block(A, A')$ is true if and only if request A will be blocked (directly or otherwise) if it is requested while A' is outstanding. Also define the following two predicates:

$$\begin{aligned} HB(A, A') &\equiv \exists A_2 \\ &\quad [Block(A_2, A') \wedge J \text{ requests } A_2 \text{ while holding } A]. \\ Cover(A, A') &\equiv \pi(A) \neq \pi(A') \wedge \\ &\quad \exists A_H \; [\pi(A_H) > \pi(A), \pi(A') \wedge Block(A_H, A)]. \end{aligned}$$

The meanings of these definitions may be clearer as they are restated less formally, in English:

1. $HB(A, A')$ means that J may be *Holding* A while requesting an allocation that is *Blocked* by A'.
2. $Cover(A, A')$ means that the range of priorities of jobs that may make a request that is blocked by A *Cover* the priorities of A and A' — that is, A and A' may be outstanding when a job preempts and makes a request that conflicts with A.

The *Cover* and *HB* relations are illustrated by the diagram in Figure 3. The horizontal lines indicate outstanding allocations, and the diagonal lines indicate *Block* relationships.

Example. Figure 4 gives the actual *Cover* and *HB* relations for our example set of jobs, according to the MBP (which will be defined in the next section). Some of these relationships are also illustrated by the diagrams in Figure 5.

	(1,1,w)	(2,1,r)	(3,1,r)	(2,2,w)	(3,2,r)	(4,2,r)	(2,3,l)	(3,3,l)
(1,1,w)		C		C			C	
(2,1,r)								
(3,1,r)				HB			HB	
(2,2,w)	C,HB		C		C			C,HB
(3,2,r)								
(4,2,r)								
(2,3,l)	C,HB		C		C			C
(3,3,l)				HB				

Figure 4: *Cover* (C) and *HB* Relations.

Indirect Blocking Conditions: To avoid allowing a job to be impeded by more than one other job, a request A must be blocked whenever there is an outstanding allocation A', held by some other job, such that:

$$\begin{aligned}
&(HB(A, A') \wedge Cover(A, A')) \vee \\
&(Cover(A', A) \wedge Cover(A, A')) \vee \\
&(HB(A, A') \wedge HB(A', A)) \vee \\
&(Cover(A', A) \wedge HB(A', A)).
\end{aligned}$$

These conditions are necessary, in the sense that if they are satisfied and a request A is *not* blocked, some job may be impeded by more than one critical section, or deadlocked. This is demonstrated by the following theorem.

Theorem 6 *(necessity of blocking conditions) Any resource allocation policy that does not permit a job to be impeded by more than one (lower priority) job, and does not permit deadlock, must enforce the indirect blocking conditions.*

Proof. Suppose a resource allocation policy enforces direct blocking, and satisfies the blocking assumptions, but does not satisfy the indirect blocking conditions. We will show that a job may be impeded more than once, or be deadlocked. Suppose request A is granted when the indirect blocking conditions are true for some outstanding allocation A'. By Corollary 2, $\pi(J) \neq \pi(J')$.

From the indirect blocking conditions, we have four subcases:

1. $HB(A, A') \wedge Cover(A, A')$.

   Suppose J_H preempts J, and is blocked by A; then J resumes execution and is blocked by A'. J_H is impeded by two jobs.

2. $Cover(A', A) \wedge Cover(A, A')$.

$$\begin{aligned}
\exists A_H, A_{H'} \quad & [\pi(A_H) > \pi(A), \pi(A') \wedge Block(A_H, A) \\
& \pi(A_{H'}) > \pi(A), \pi(A') \wedge Block(A_{H'}, A')].
\end{aligned}$$

There are four cases:

(a) $\pi(J_H) > \pi(J_{H'})$.

Suppose $J_{H'}$ preempts J and is blocked by A', and then J_H preempts $J_{H'}$ and is blocked by A. Now, $J_{H'}$ is impeded by two jobs: first, by J until it releases A, and then by J' until it releases A'.

(b) $\pi(J_H) < \pi(J_{H'})$.

Suppose J_H preempts J and is blocked by A, and then $J_{H'}$ preempts J_H and is blocked by A'. J_H is impeded by two jobs.

(c) $\pi(J_H) = \pi(J_{H'}) \ \wedge \ J_H = J_{H'}$.

Suppose J_H is blocked by A, waits for A to be released, and then is blocked again by A'. J_H is impeded by two jobs.

(d) $\pi(J_H) = \pi(J_{H'}) \ \wedge \ J_H \neq J_{H'}$.

Suppose J_H is blocked by A, and while J_H is waiting $J_{H'}$ arrives. When A is released, J_H completes. $J_{H'}$ preempts and is blocked by A'. $J_{H'}$ is impeded by two jobs.[2].

3. $HB(A, A') \ \wedge \ HB(A', A)$

Suppose J is blocked by A'; then J' resumes execution and is blocked by A. There is a deadlock.

4. $Cover(A', A) \ \wedge \ HB(A', A)$.

Suppose J'_H preempts J, and is blocked by A'; then J' resumes execution and is blocked by A. J_H is impeded by two jobs.

□

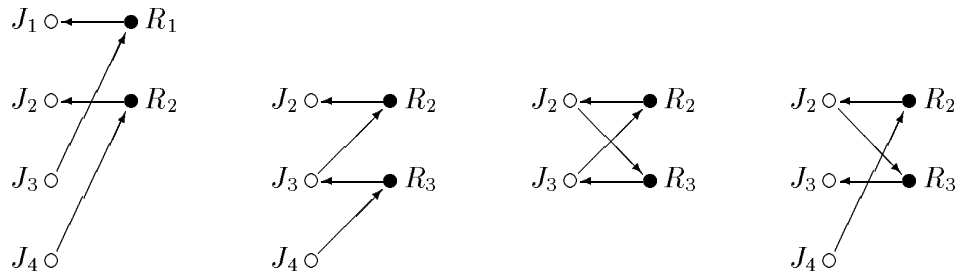
6 A Minimal Blocking Policy

We shall now define a resource management policy that enforces the direct and indirect blocking conditions. Let BD, HB and $Cover$ be as defined above. Let $Block(A, A')$ be defined as the least fixed-point of following recurrences:

$$\begin{aligned} Block(A, A') \ \equiv \ & BD(A, A') \ \vee \\ & (HB(A, A') \ \wedge \ Cover(A, A')) \ \vee \\ & (Cover(A', A) \ \wedge \ Cover(A, A')) \ \vee \\ & (HB(A, A') \ \wedge \ HB(A', A)) \ \vee \\ & (Cover(A', A) \ \wedge \ HB(A', A)). \end{aligned}$$

Assuming that priorities are assigned statically to job *types*, and that we consider allocations to the same job type equivalent, the relation $Block$ can be computed statically, by iterative

[2] If we did not allow different tasks to have the same priority, this case could only arise when a job misses its deadline.



$Cover(A_1, A_2) \wedge Cover(A_2, A_1)$ | $HB(A_3, A_2) \wedge Cover(A_3, A_2)$ | $HB(A_2, A_3) \wedge HB(A_3, A_2)$ | $Cover(A_2, A_3) \wedge HB(A_2, A_3)$

Figure 5: Indirect Blocking Conditions

	(1,1,w)	(2,1,r)	(3,1,r)	(2,2,w)	(3,2,r)	(4,2,r)	(2,3,l)	(3,3,l)
(1,1,w)		D	D	I			I	
(2,1,r)	D							
(3,1,r)	D							
(2,2,w)	I				D	D		I
(3,2,r)				D				
(4,2,r)				D				
(2,3,l)	I							D
(3,3,l)				I			D	

Figure 6: The direct (D) and indirect (I) blocking relations.

convergence starting from *BD*. That is, $Block = \lim_{i \to \infty} Block_i$, where:

$$
\begin{aligned}
Block_0(A, A') &\equiv BD(A, A') \\
Block_{i+1}(A, A') &\equiv BD(A, A') \vee \\
&\quad (HB_i(A, A') \wedge Cover_i(A, A')) \vee \\
&\quad (Cover_i(A', A) \wedge Cover(A, A')) \vee \\
&\quad (HB_i(A, A') \wedge HB_i(A', A)) \vee \\
&\quad (Cover_i(A', A) \wedge HB_i(A', A)). \\
HB_i(A, A') &\equiv \exists A_2 \\
&\quad [Block_i(A_2, A') \wedge J \text{ requests } A_2 \text{ while holding } A]. \\
Cover_i(A, A') &\equiv \pi(A) \neq \pi(A') \wedge \exists A_H \\
&\quad [\pi(A_H) > \pi(A), \pi(A') \wedge Block_i(A_H, A)].
\end{aligned}
$$

Example. Figure 5 illustrates the four cases in which indirect blocking can occur, as they arise in our example set of jobs (shown in Figure 1). The direct blocking and indirect blocking relationships for this example are shown by letters "D" and "I", respectively, in Figure 6.

Minimal Blocking Policy: A request A is blocked if and only if

$$\exists A' \, [Outstanding(A') \ \wedge \ Block(A, A')].$$

We call this policy the "minimal blocking policy" (MBP, for short), because it is minimal in two senses. First, it is the least fixed-point of a set of the defining equations. Second, these equations are minimal, in the sense that they cannot be weaker without allowing deadlock or a job to be impeded by more than one other job.

Note that it is a consequence of the definition of *Block* that $J = J' \Rightarrow \neg Block(A, A')$; that is, a request is never blocked by an outstanding allocation held by the same job.

Of course, a blocking policy based on a least fixed-point is not meaningful unless the least fixed-point is nontrivial. For example, the predicate that is true for every pair of allocations is a fixed point of these equations (the greatest fixed point). Another trivial fixed point is the predicate that is true for every pair of allocations with equal priority. If this turned out to be the least fixed point, the MBP would still reduce to a single global critical section policy — i.e. a new request would always be blocked whenever there is any allocation outstanding. Our example shows that the fixed point is not trivial.

6.1 Identifiable Blocker

In order to apply the MBP in conjunction with the priority inheritance policy, we need to verify that it satisfies the assumption that there is an identifiable blocker.

Lemma 7 (identifiable blocker(s)) *For every blocked job J, there is at least one* blocker *— that is, an allocation A', held by another job J', that is* blocking *J. J cannot become unblocked until J' releases A'.*

Proof. The definition of the MBP guarantees the existence of at least one blocker A' for each blocked request A, and that A cannot become unblocked so long as such an allocation is outstanding. □

7 One-Step Convergence

If the least fixed-point is computed by the iterative technique described above, convergence is reached in one step. This is demonstrated by the theorem below.

Theorem 8 (one-step convergence) $Block_2 \equiv Block_1$.

The proof makes use of the following two lemmas.

Lemma 9 (*HB* converges) $HB_1(A, A') \Rightarrow HB_0(A, A') \ \vee \ Cover_0(A', A)$.

Proof. Suppose $HB_1(A, A')$; i.e.,

$$\exists A_2 [Block_1(A_2, A') \ \wedge \ J \text{ requests } A_2 \text{ while holding } A].$$

From $Block_1(A_2, A')$, we have three cases:

1. $BD(A_2, A')$.

   By definition, $HB_0(A, A')$.

2. $HB_0(A_2, A')$.

   $$\exists A_3[BD(A_3, A') \ \wedge \ J \text{ requests } A_3 \text{ while holding } A_2].$$

   By the nesting of critical sections, J also requests A_3 while holding A, so that $HB_0(A, A')$ is true.

3. $Cover_0(A', A_2)$.

   $$\pi(A') \neq \pi(A_2) \ \wedge \ \exists A_H \ [\pi(A_H) > \pi(A'), \pi(A_2) \ \wedge \ BD(A_H, A')].$$

   Since A and A_2 are both requests of J, $\pi(A) = \pi(A_2)$, so $\pi(A) \neq \pi(A')$ and $\pi(A_H) > \pi(A)$, and $Cover_0(A', A)$.

□

Lemma 10 ($Cover$ converges) $Cover_1(A, A') \Rightarrow Cover_0(A, A')$.

Proof. Suppose $Cover_1(A, A')$; i.e.,

$$\pi(A) \neq \pi(A') \ \wedge \ \exists A_H[\pi(A_H) > \pi(A), \pi(A') \ \wedge \ Block_1(A_H, A)].$$

From $Block_1(A_H, A)$, we have three cases:

1. $BD(A_H, A)$. By definition, $Cover_0(A, A')$.

2. $HB_0(A_H, A)$.

   $$\exists A_{H'}[BD(A_{H'}, A) \ \wedge \ J_H \text{ requests } A_{H'} \text{ while holding } A_H].$$

   Since A_H and $A_{H'}$ are requested by the same job, $\pi(A_{H'}) = \pi(A_H) > \pi(A), \pi(A')$. Thus, $Cover_0(A, A')$ is true.

3. $Cover_0(A, A_H)$.

   $$\pi(A) \neq \pi(A_H) \ \wedge \ \exists A_{H'}[\pi(A_{H'}) > \pi(A), \pi(A_H) \ \wedge \ BD(A_{H'}, A)].$$

   Since $\pi(A_H) > \pi(A')$, it follows that $\pi(A_{H'}) > \pi(A')$. We know $\pi(A) \neq \pi(A')$, and so $Cover_0(A, A')$ is true.

□

Proof of Theorem 8. By the definition of $Block_i$, we know $Block_1 \subseteq Block_2$. Lemma 9 and Lemma 10 show that $Block_2 \subseteq Block_1$. □

8 Bounded Impedance

In this section, we show that the MBP assures that there is no deadlock, and that each job can never be impeded by more than one other job or by two non-nested critical sections of any one job. The first step is to show that the potential blocking relation is intransitive.

Theorem 11 (intransitivity of blocking) *If J' is blocking some other job, then J' itself is not blocked.*

Proof. The proof is by induction on time. We are assuming that initially there are no outstanding allocations, so the lemma holds trivially. Suppose t is the first time the lemma fails. Before time t there is no blocking chain with length > 1. At time t such a chain comes into being. The only way this can happen is by some job executing a request that is blocked. By the priority inheritance policy, the job which makes the request that creates the first blocking chain with length > 1 is either the top job, or is the tail of a blocking chain whose head is the top job. In either case, the top job is the head of the first blocking chain of length ≥ 2.

Without loss of generality, suppose the top job is requesting A, which is blocked by A', and J' is requesting A'_2, which is blocked by A''. Since J' holding A' and blocked by A'', we have $HB(A', A'')$. Since J is top job, and we are assuming a job never blocks itself, $\pi(A) \geq \pi(A'), \pi(A'')$. By Corollary 2, we know $\pi(A) \neq \pi(A')$ and $\pi(A') \neq \pi(A'')$.

Suppose $\pi(A) = \pi(A'')$. Then, by Corollary 2, $J = J''$. Since J is holding A'' while blocked by A', we have $HB(A'', A')$. This means whichever of A' and A'' was granted first should have blocked the other— a contradiction.

Suppose $\pi(A) > \pi(A'')$. Since $\pi(A) > \pi(A')$ and A is blocked by A', $Cover(A', A'')$. This means whichever of A' and A'' is granted first should have blocked the other— a contradiction. □

Corollary 12 *The currently executing job is always the top job, or is blocking the top job.*

Proof. By the priority inheritance policy, the executing job is the tail of a chain whose head is the top job. Since blocking is intransitive, such a chain must have length zero or one. □

Corollary 13 *There can be no deadlock.*

Proof. This follows directly from the intransitivity of blocking, and the assumption that a job cannot block itself. □

Lemma 14 (monotonicity) *A job can only be blocked by a lower priority job.*

Proof. Suppose A is blocked by A'. By Corollary 2, $\pi(A) \neq \pi(A')$. Suppose $\pi(A) < \pi(A')$. By the priority inheritance policy, J cannot execute to request A after J' arrives unless J is the tail of a chain whose head is the top job. Since J is also blocked, by Corollary 12, J must be the top job — a contradiction. □

Lemma 15 *A job cannot be impeded by more than one job.*

Proof. Suppose J is impeded by J' — that is, $\pi(J') < \pi(J)$ but J' executes between the time J arrives and J completes. Consider the first time this happens. By the priority inheritance policy, J' must be the tail of a chain whose head is the top job. Since blocking is intransitive, and $\pi(J') < \pi(J)$, J' must be holding an allocation A' that is blocking the top job, $J_{H'}$. Since $J_{H'}$ is top job, $\pi(J) \leq \pi(J_{H'})$. Since this is the first time J is impeded by J', J' must acquire A' before J arrives.

Suppose J is also impeded by $J'' \neq J'$. By the same reasoning as above, $\exists J_{H''}, A''$ such that $\pi(J) \leq \pi(J_{H''})$, A'' blocks $J_{H''}$, and A'' is acquired before J arrives.

Thus $\pi(J_{H'}), \Pi(J_{H''}) \geq \pi(J) > \pi(J'), \pi(J'')$, and so $Cover(A', A'')$ and $Cover(A'', A')$. Since A' and A'' are both granted before J arrives, and are both outstanding during J's execution, one of them is granted while the other is outstanding. Whichever of A' and A'' was granted first should have blocked the other — a contradiction. □

Theorem 16 *A job can only be impeded by one (outermost) critical section.*

Proof. Suppose J is impeded by two critical sections. By Lemma 15, there is at most one job J' that impedes J. Suppose J is impeded by two critical sections of J'. By Lemma 3, J can only be impeded by J' if J' is in a critical section that it entered before J arrived. Since we are assuming that overlapping critical sections are properly nested, the theorem is satisfied. □

9 Simplifications

The MBP is simpler and more efficient to apply than might be immediately apparent. Specifically, when a request is made it is only necessary to check the indirect blocking conditions for one of the outstanding allocations. The following simplifications are based on the notion of priority ceiling, as defined in [5]:

Let the *ceiling* $\lceil A \rceil$ of an allocation A be the maximum of the priorities of A and all the requests that may be directly blocked by A; that is:

$$\lceil A \rceil = max\{\pi(A') \mid BD(A', A) \vee A = A'\}.$$

Example. The ceilings of the allocations in our example are shown in Figure 7.

Because the fixed-point, *Block*, is reached from *BD* in only one step, the *BD* may be replaced by *Block* in the definition of $\lceil A \rceil$:

Lemma 17 $\lceil A \rceil = max\{\pi(A') \mid Block(A', A) \vee A = A'\}$.

Proof. Let $c(A) = max\{\pi(A') \mid Block(A', A) \vee A = A'\}$. Since $BD \subseteq Block$, $\lceil A \rceil \leq c(A)$.

Ceilings

A	$\lceil A \rceil$
(1,1,w)	3
(2,1,r)	2
(3,1,r)	3
(2,2,w)	4
(3,2,r)	3
(4,2,r)	4
(2,3,l)	3
(3,3,l)	3

Figure 7: Ceilings of Allocations.

Suppose $\pi(A_2) = \lceil A \rceil$. From the definition of $Block_1$ and Theorem 8, if $Block(A_2, A)$ and not $BD(A_2, A)$, then either $Cover(A, A_2)$ or $HB(A_2, A)$.

If $Cover(A, A_2)$, then

$$\exists A_H \; [\pi(A_H) > \pi(A_2), \pi(A) \; \wedge \; Block(A_H, A)],$$

and so $\lceil A \rceil \geq \pi(A_H) > \pi(A_2)$. If $HB(A_2, A)$, then $BD(A_2, A)$, so that $\lceil A \rceil \geq \pi(A_2)$. Thus $\lceil A \rceil \geq c(A)$. □

It follows that the relation $Cover(A, A')$ is equivalent to $\pi(A) \neq \pi(A') \; \wedge \; \lceil A \rceil > \pi(A')$.

Lemma 18 (blocker has highest ceiling) *A job J can only be blocked by an allocation that has the highest ceiling of all outstanding allocations not held by J itself.*

Proof. Suppose J is blocked by A', and there is another outstanding allocation A'' such that $\lceil A' \rceil < \lceil A'' \rceil$. Since J is blocked by A' and blocking is not transitive, J is top job. Since J is top job, and neither A' nor A'' is held by J, by Corollary 2, $\pi(J) > \pi(A'), \pi(A'')$. Thus, $Cover(A', A'')$. Since $\lceil A'' \rceil > \lceil A' \rceil$, $Cover(A'', A')$. Whichever of A' and A'' was granted first should have blocked the other — a contradiction. □

Lemma 19 $Block(A, A') \Rightarrow \pi(A) \leq \lceil A' \rceil$.

Proof. The proof follows directly from the definition of $Block$:

$$\begin{aligned} Block(A, A') &\Rightarrow Cover(A', A) \; \vee \; HB(A, A'); \\ Cover(A', A) &\Rightarrow \pi(A) < \lceil A' \rceil; \\ HB(A, A') &\Rightarrow \pi(A) \leq \lceil A' \rceil. \end{aligned}$$

□

Lemma 20 (equal ceilings) *If J requests A while A' and A'' are held by jobs other than J, and $\lceil A' \rceil = \lceil A'' \rceil$, then $Block(A, A') \Leftrightarrow Block(A, A'')$.*

Proof. If $\lceil A' \rceil < \pi(A)$, then by Lemma 19, $\neg Block(A, A') \; \wedge \; \neg Block(A, A'')$.

Suppose $\lceil A' \rceil \geq \pi(A)$. Without loss of generality, suppose $\pi(A') \geq \pi(A'')$. We have three cases:

1. $\pi(A) > \pi(A')$. Since $\lceil A'' \rceil = \lceil A' \rceil > \pi(A') \geq \pi(A'')$, we have $Cover(A', A'') \wedge Cover(A'', A')$; i.e. either A' or A'' should have blocked the other — a contradiction.

2. $\pi(A) = \pi(A')$. By Corollary 2, $J = J'$ — a contradiction to the assumption that A' is held by another job.

3. $\pi(A) < \pi(A')$. By Corollary 12, J must be blocking the top job. By the intransitivity of blocking (Lemma 11), $\neg Block(A, A') \wedge \neg Block(A, A'')$.

□

The SCP is defined in terms of a special semaphore, S^*, which is defined as "the semaphore with the highest priority ceiling of all semaphores currently locked by jobs other than job J[5]". This is a very useful concept, but it needs to be defined more carefully. The problem is that there may be more than one semaphore satisfying this description. Therefore, while generalizing the definition of S^* to A^*, we add some restrictions.

Suppose job J is requesting resource allocation A. Let

$$\begin{aligned} Others(A) &= \{A' \mid A' \text{ outstanding and } J' \neq J\} \\ c^*(A) &= max\{\lceil A' \rceil \mid A' \in Others(A)\} \\ \pi^*(A) &= max\{\pi(A') \mid A' \in Others(A) \wedge \lceil A' \rceil = c^*(A)\}. \end{aligned}$$

By Corollary 2, there is a unique job J^* holding a resource allocation with priority $\pi^*(A)$. Let A^* be the allocation held by J^* that J^* has acquired least recently.

Theorem 21 (A^* is representative) *Suppose job J is executing a request for resource allocation A.*

$$\exists A'[A' \text{ outstanding} \wedge Block(A, A')] \Rightarrow Block(A, A^*).$$

Proof. Suppose $Block(A, A')$. By Lemma 18, we know that $\lceil A' \rceil = \lceil A^* \rceil$. By Lemma 20, $Block(A, A^*)$. □

Note that this theorem means that it is only necessary to test A^*, one representative of the outstanding allocations held by other jobs, to determine whether a request should be blocked. Since the relation *Block* is entirely static (assuming static priorities), it can be computed for a program at compile time, so this check reduces to a table look-up. Note also that the conditions we added to make the definition of A^* unique were not needed in the proof. That is, any allocation that satisfied the original definition in [5] would do. This gives additional freedom to the implementation, in choosing a convenient representative.

10 Comparison to SCP

The MBP is a consistent extension of the SCP, though the forms of the two definitions do not make this immediately apparent. In this section, we will demonstrate their equivalence.

The SCP is defined in [5] as follows:

> Let J be the highest priority job among the jobs ready to run. J is assigned the processor and let S^* be the semaphore with the highest priority ceiling of all semaphores currently locked by jobs other than job J. Let the job holding the lock on S^* be J^*. Before job J enters its critical section, it must first obtain the lock on the semaphore S guarding the shared data structure. If the semaphore S is unlocked, job J will be granted the lock on S if and only if at least *one* of the following conditions is true:
>
> **a. Condition C1**: The priority of job J is greater than the priority ceiling of S^*.
>
> **b. Condition C2**: The priority of job J is equal to the priority ceiling of S^* and the current critical section of J will not attempt to lock any semaphore already locked by J^*.
>
> **c. Condition C3**: The priority of job J is equal to the priority ceiling of S and the lock on semaphore S will not be requested by J^*'s preempted critical section.

The definition goes on to say that "if job J blocks higher priority jobs, J *inherits* P_H, the priority of the highest priority job blocked by J."

This definition presumes that S^* and J^* are well-defined. Since the SCP assume there is one dummy semaphore that is always outstanding and is held by the "idle" job, there is at least one semaphore that satisfies the definition of S^*. As mentioned above, the problem is that there may be more than one such semaphore.

For the MBP, we have proven that all such semaphores must be held by one job J^*, and that they are all equivalent with respect to the test for blocking, but this proof relies on the definition of blocking being well-founded. We will be able to apply this theorem after we show that the SCP is equivalent to the MBP, but first we must make the definition well-founded. We restate the definition of the SCP, using our definition of A^* (from the preceding section), and generalizing it to fit our model of abstract nonpreemptible resources.

Let $p(J)$ denote the *inherited priority* of job J; i.e.:

$$\begin{aligned} p(J) &= \pi(J) \text{ if } J \text{ is top job} \\ p(J) &= max\{\pi(J), p(J')\} \text{ if } J \text{ is blocking } J'. \end{aligned}$$

The Semaphore Control Protocol (SCP): Suppose A is requested. If there is an outstanding allocation that directly blocks A, the request is blocked. Otherwise, let A^* be

as defined in Section 9 and used in Theorem 21. The request is granted if and only if at least *one* of the following conditions is true:

Condition C1: $p(A) > \lceil A^* \rceil$.

Condition C2: $p(A) = \lceil A^* \rceil \ \wedge \ \forall A_2^* [A_2^* \text{ outstanding} \Rightarrow \neg HB(A, A_2^*)]$.

Condition C3: $p(A) = \lceil A \rceil \ \wedge \ \neg HB(A^*, A)$.

Theorem 22 *MBP blocks a request if and only if SCP blocks the same request under the same conditions.*

Proof. The proof is by induction on the number of requests. Initially, there are no requests, so the set of outstanding allocations is the same for both the MBP and the SCP. Suppose A is requested. The MBP and SCP both block a request if there is an allocation that directly blocks it. We therefore only need to consider the case that A is not blocked directly.

By induction, the set of outstanding allocations is the same for both policies. Consequently, the theorems and lemmas proven for the MBP are true, to the extent that they apply to outstanding allocations and blockages.

Suppose A is blocked indirectly by the MBP. By Theorem 11, $p(A) = \pi(A)$. By Theorem 21, one of the following cases applies for A^*:

1. $Cover(A, A^*) \ \wedge \ Cover(A^*, A)$.
   From $Cover(A^*, A)$, $\pi(A) < \lceil A^* \rceil$ (and $\pi(A) = p(A)$), so neither C1 nor C2 is satisfied.
   From $Cover(A, A^*)$, we have $\pi(A) < \lceil A \rceil$, and so C3 is not satisfied.
2. $HB(A, A^*) \ \wedge \ HB(A^*, A)$.
   From $HB(A, A^*)$, we have $\lceil A \rceil \geq \pi(A^*)$ and $\lceil A^* \rceil \geq \pi(A)$, so C1 is not satisfied.
   From $HB(A, A^*)$, C2 is not satisfied.
   From $HB(A^*, A)$, C3 is not satisfied.
3. $HB(A, A^*) \ \wedge \ Cover(A, A^*)$.
   From $HB(A, A^*)$, we have $\pi(A) < \lceil A^* \rceil$, so C1 is not satisfied.
   From $HB(A, A^*)$, C2 is not satisfied.
   From $Cover(A, A^*)$, we have $\pi(A) < \lceil A \rceil$, so C3 is not satisfied.
4. $Cover(A^*, A) \ \wedge \ HB(A^*, A)$.
   From $Cover(A^*, A)$, we have $\pi(A) < \lceil A^* \rceil$, so neither C1 nor C2 is satisfied.
   From $HB(A^*, A)$, C3 is not satisfied.

We now want to show that if A is *not* blocked indirectly by the MBP then it must not be blocked by the SCP. This time, we cannot assume $p(A) = \pi(A)$, since A is not blocked by the MBP. We can still assume there are no chains of length greater than one, however, so either $p(A) = \pi(A)$, or $p(A) = \pi(A_H)$ and A is blocking A_H, where A_H is the top job.

If A is not blocked indirectly by the MBP then for every outstanding allocation, and in particular for A^*, one of the following cases applies:

1. $\neg Cover(A, A^*) \wedge \neg HB(A^*, A)$.

   We know that $p(A) \leq \lceil A \rceil$, since there are not yet any chains of length greater than one.

   If $p(A) = \lceil A \rceil$ then since $\neg HB(A^*, A)$, C3 is satisfied.

   If $p(A) < \lceil A \rceil$ then, since $\neg Cover(A, A^*)$ and $p(A) \geq \pi(A)$, we know $\pi(A^*) \geq \lceil A \rceil \geq \pi(A)$. By the definition of ceiling, $\lceil A \rceil \geq \pi(A)$. By Corollary 2, we know $\pi(A^*) \neq \pi(A)$, so $\pi(A^*) > \pi(A)$. Since J is executing and has lower priority than A^*, it must be blocking a request with priority $\geq \pi(A^*)$. Thus we have $\pi(A^*) \leq \lceil A \rceil$, and since we already know $\pi(A^*) \geq \lceil A \rceil$, $\pi(A^*) = \lceil A \rceil$. By Corollary 2, there is at most one job at this priority level holding an allocation, so J must be blocking J^*. This means $p(A) = \pi(A^*) = \lceil A \rceil$. Since $\neg HB(A^*, A)$, C3 is satisfied.

2. $\neg Cover(A^*, A) \wedge \neg HB(A, A^*)$.

   If $p(A) < \lceil A^* \rceil$ then $\pi(A) \leq \pi(A) < \lceil A^* \rceil$. Since J is executing, $p(A^*) < p(A)$, but then $Cover(A^*, A)$ — a contradiction.

   If $p(A) = \lceil A^* \rceil$, suppose C2 is not satisfied; that is, $\exists A_2^* HB(A, A_2^*)$. Since A_2^* does not block A according to the MBP, $Cover(A, A_2^*)$ must be false. Since $p(A)$ is either the priority of A or the priority of a job that is directly blocked by A, we have $p(A) \geq \pi(A)$. From $\neg Cover(A, A_2^*)$, either $\pi(A) \geq \lceil A \rceil$ or $\pi(A_2^*) \geq \lceil A \rceil$.

   If $\pi(A) \geq \lceil A \rceil$, then
   $$\pi(A) \geq \lceil A \rceil \geq p(A) \geq \pi(A).$$

   If $\pi(A_2^*) \geq \lceil A \rceil$, then
   $$p(A) = \lceil A_2^* \rceil \geq \pi(A_2^*) \geq \lceil A \rceil \geq p(A).$$

   In either case, $p(A) = \lceil A \rceil$. Supposing C3 is not satisfied, $HB(A_2^*, A)$ must be true. We already have $HB(A, A_2^*)$, so A_2^* blocks A by the MBP — a contradiction.

   If $p(A) > \lceil A^* \rceil$ then C1 is satisfied.

□

11 Conclusion and Further Research

We have defined a policy for allocation of nonpreemptible resources that can be applied to both binary semaphores and reader/writer resources, the MBP. When used in conjunction with priority inheritance processor allocation, the MBP assures that no job can ever be impeded by more than one (outermost) critical section of a lower priority job. This policy is minimal, in the sense that relaxing any of the restrictions results in a job being blocked by critical sections of more than one job. The MBP is defined as the least fixed-point of a recurrence, which is shown to converge in one iteration.

The MBP is a consistent generalization of the SCP proposed in in [5], to more abstract resources. The reader is referred to that paper for a discussion of the schedulability analysis, and implementation techniques (including how the value A^* can be maintained on a stack at run time), since these apply equally to the MBP.

One possible advantage of the MBP formulation over the SCP may be that it exposes additional possibilities for compiler optimizations. In particular, if a tabular for the relation *Block* has an empty rows, or a row containing only allocations with higher priority, code to check the blocking conditions for the corresponding request may be omitted.

The fixed-point construction described here is very general. We have already made several applications to more general resource allocation problems. First, the MBP may be generalized slightly further, so that it applies to multiple-unit resources with requests for "m-out-of-n" units. We are currently studying whether the blocking conditions can still be tested efficiently at run time. It also appears to be possible to state necessary conditions for higher limits on the number of critical sections that may impede a job (higher than one), but then there appears to be an explosion in the number of cases. The MBP can also be applied to the runtime stack resource, where several tasks share a single runtime stack. This can significantly reduce the need for storage in a system with many tasks. A preliminary description of this technique is reported in [1], and a more complete description is given in [2].

Chen and Lin[3] have already shown that the "priority inheritance protocol" (PCP) can be applied to earliest-deadline scheduling. The same appears to be true for the MBP. In [2] it is shown that for the single-stack storage model separating the concept of "preemption level" from priority allows the MBP definitions to be applied to earliest-deadline scheduling. We have not yet reformulated the MBP to reflect this further separation of concerns, but are optimistic that this may permit an application of the MBP to earliest deadline scheduling that is efficient enough to be practical.

Acknowledgements

The author thanks Kevin Jeffay, Ragunathan Rajkumar, Edward Giering, and James Groh for their constructive criticisms of early versions of this paper.

References

[1] T.P. Baker, C. Malec, R. Wilson, "Practical Tasking", Boeing Aerospace and Electronics Company white paper (1989).

[2] T.P. Baker, "Stack-Based Scheduling of Realtime Processes", Department of Computer Science, Florida State University (December 1989).

[3] M.I. Chen and K.J. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems", techical report UIUCDCS-R-89-1511, Department of Computer Science, University of Illinois at Urbana-Champaign (April 1989).

[4] C.L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", JACM 20.1 (January 1973) 46-61.

[5] R. Rajkumar, L. Sha, and J.P. Lehoczky, "An Optimal Priority Inheritance Protocol for Real-Time Synchronization", technical report, Carnegie Mellon University (17 October 1988) submitted for publication.

[6] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols, An Approach to Real-Time Synchronization", technical report CMU-CS-87-181, Carnegie Mellon University (November 1987).