The Journal of Real-Time Systems, 3, 67–99 (1991)

Stack-Based Scheduling of Realtime Processes

T.P. BAKER*
Department of Computer Science, Florida State University, Tallahassee, FL 32306-4019 U.S.A.

Abstract. The Priority Ceiling Protocol (PCP) of Sha, Rajkumar and Lehoczky is a policy for locking binary semaphores that bounds priority inversion (i.e., the blocking of a job while a lower priority job executes), and thereby improves schedulability under fixed priority preemptive scheduling. We show how to extend the PCP to handle: multiunit resources, which subsume binary semaphores and reader-writer locks; dynamic priority schemes, such as earliest-deadline-first (EDF), that use static "preemption levels"; sharing of runtime stack space between jobs. These extensions can be applied independently, or together.

The Stack Resource Policy (SRP) is a variant of the SRP that incorporates the three extensions mentioned above, plus the conservative assumption that each job may require the use of a shared stack. This avoids unnecessary context switches and allows the SRP to be implemented very simply using a stack. We prove a schedulability result for EDF scheduling with the SRP that is tighter than the one proved previously for EDF with a dynamic version of the PCP.

The Minimal SRP (MSRP) is a slightly more complex variant of the SRP, which has similar properties, but imposes less blocking. The MSRP is *optimal* for stack sharing systems, in the sense that it is the least restrictive policy that strictly bounds priority inversion and prevents deadlock for rate monotone (RM) and earliest-deadline-first (EDF) scheduling.

1. Introduction

Hard realtime computer systems are subject to absolute timing requirements, which are often expressed in terms of deadlines. They are often subject to severe resource constraints; in particular, limited memory. They are also expected to be reliable in the extreme, to that it is necessary to verify *a priori* that a system design will meet timing requirements within the given resource constraints.

Verifying timing and resource utilization properties of programs is inherently difficult. In fact, it is impossible without some constraints on program structure. This is a consequence of the Halting Problem, which is known to be undecidable. To get around the Halting Problem, it is customary to assume the program is divided into a set of *jobs*, whose arrival times, execution times, and other resource requirements are known. Verification that the program satisfies timing and resource constraints then reduces to a scheduling problem. However, scheduling is also difficult. Specifically, determining whether a set of jobs can be scheduled so as to complete execution by a fixed deadline is known to be NP-hard (Garey and Johnson 1979; Leung and Merrill 1980) unless severe restrictions are placed on the problem.

Practical schedulability analysis requires a simple model of software architecture. Liu and Layland (1973) were able to obtain very strong results from such a simple model. They assumed that for jobs with hard deadlines:

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

1. there is a fixed set of such jobs;
2. requests for job executions are periodic, with a constant interval between requests;
3. relative deadlines are the same as the respective periods, i.e., a job need only complete by the arrival of the next request for it;
4. no synchronization or precedence requirements exist between jobs;
5. there is no control over phasing of periodic jobs;
6. processor time is the only resource that needs to be scheduled;
7. execution times are constant.

Subject to these assumptions, they proved that rate monotone (RM) scheduling, in which the job with the shortest period is given highest priority, is optimal among static priority policies, and that earliest-deadline-first (EDF) scheduling is optimal among dynamic priority policies. They also derived conditions for schedulability under these two policies, and for a mixture of the two.

Other researchers have discovered that some of the restrictive assumptions made by Liu and Layland can be relaxed, generally without much change to the schedulability results or their proofs. Most of these extensions have been to the RM policy: Sha, Lehoczky, and Rajkumar (1986) outline approaches to dealing with transient overloads due to variable execution times; Sprunt, Sha, and Lehoczky (1989) describe techniques for handling an aperiodic *server* job; Sha, Rajkumar, and Lehoczky (1987) show that the schedulability results can be adapted to tolerate bounded blocking, such as may be due to scheduling exclusive access to shared data. The problem of bounded blocking has also been addressed for EDF scheduling, by Chen and Lin (1989).

Liu and Layland's Theorem 5 (1973) says that a set of n periodic jobs can be scheduled by the RM policy if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot (2^{1/n} - 1).$$

Here T_i and C_i denote the period and execution time of the ith job, respectively, and the jobs are ordered by increasing period.

Sha, Rajkumar, and Lehoczky's Theorem 15 (1987) generalizes this result, showing that the jobs are schedulable by the RM priority assignment if

$$\underset{k=1,\ldots,n}{\forall k} \left(\sum_{i=1}^{k} \frac{C_i}{T_i} \right) + \frac{B_k}{T_k} \leq k \cdot (2^{1/k} - 1).$$

Here B_k is an upper bound on the duration of blocking that the kth job may experience due to resources held by lower priority jobs. The authors introduce the term *priority inversion* to describe this sort of blocking.

Liu and Layland's Theorem 7 (1973) says that a set of n periodic jobs can be scheduled by the EDF policy iff[1]

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

Chen and Lin's Theorem 4 (1989) extends the *if* part of this result to show that the jobs are schedulable using the dynamic PCP for semaphore locking if

$$\sum_{i=1}^{n} \frac{C_i + B_i}{T_i} \leq 1.$$

In Section 5, we will tighten this result.

For these results to be useful, the priority inversion bound, B_i, of each job must be small. This requirement has motivated the study of resource allocation policies that can strictly bound priority inversion.

A great deal of effort has been spent studying the extreme forms of unbounded blocking that are of interest to conventional operating systems designers, such as deadlock and starvation (Bic and Shaw 1988). Unfortunately, conventional operating systems techniques do not provide a tight enough bound on blocking to be suitable for realtime schedulability analysis. For example, the ordered resource allocation technique of Havender (1968) still allows a job to be blocked by up to n lower priority jobs.

Sha, Rajkumar, and Lehoczky (1987) have devised a locking protocol for binary semaphores, cxalled the Priority Ceiling Protocol (PCP), for which priority inversion is bounded by the execution time of the longest critical section of a lower-priority job. This protocol has since been extended in several directions, including reader-writer resources (Sha, Rajkumar and Lehoczky 1989), mode changes (Sha, Rajkumar and Lehoczky 1989), and multiple processors (Rajkumar, Sha and Lehoczky 1988). Several variations have been defined, including one which is optimal in the sense of avoiding unnecessary blocking (Baker 1989; Rajkumar, Sha and Lehoczky 1988). Chen and Lin have also extended the PCP to use dynamically recomputed priority ceilings, so that it can be applied with EDF as well as RM priorities (1989).

This article describes three more extensions to the PCP. These are:

1. Multi-unit resources.
   This extension is based on treating the priority ceiling of a resource as a function of the number of units that are currently available. It permits us to subsume binary semaphores and reader-writer locks.
2. Simpler support for EDF scheduling.
   This extension is based on separating the priority of a job, which may be dynamic, from its *preemption level*, which is required to be static. Preemption levels are based on the deadlines of jobs, relative to their request times. So long as these do not change, ceilings do not need to be recomputed. This extension supports EDF, RM, deadline-monotone, and combinations of these policies.
3. Sharing runtime stack resources.
   This extension is based on treating the shared runtime stack as a resource, with ceiling zero, which is requested at the time each job starts executing.

Although these three extensions can be applied independently, they work very well together. We present them here in a combined form, which we call the *Stack Resource Policy*, or *SRP*. The SRP takes one step further away from the PCP, by treating every job as if it requires the use of a shared stack. This means that if it is necessary to block a job to wait for a shared resource held by another job, this is done at the time the job attempts to preempt (and thereby occupy runtime stack space), rather than later, when it actually may need the shared resource. At the cost of some reduction in concurrency, this earlier blocking saves unnecessary context switches, and allows us to implement the SRP very simply using a stack.

We are also able to prove a schedulability result for EDF scheduling with the SRP that is tighter than the one proved in (Chen and Lin 1989) for EDF with the dynamic PCP. This proof appears to be independent of the early blocking, so that it could also apply to the dynamic PCP.

The idea of using early blocking based on preemption levels, with a shared stack, has been around for a long time. It is the way many machines handle hardware interrupts, and has been used in real-time executives for at least 15 years. However, the use of this technique appears to have been limited to fixed-priority scheduling, without consideration for locking of individual resources. Moreover, the problem of predicting schedulability of periodic task systems using this technique does not appear to have been formally addressed.

The rest of this article is organized as follows. Section 2 defines the elements of our formal model, including jobs, featherweight processes, and resources. Section 3 outlines the general reasoning underlying the SRP. Section 4 defines the SRP, and proves that it works. Section 5 gives the schedulability result for earliest-deadline-first (EDF) scheduling with the SRP. Section 6 explains the idea of stack sharing, how it leads to interactions with allocation of other resources, and how the SRP solves these interactions. Section 7 compares the SRP to the Priority Ceiling Protocol, and includes a comparative example. Section 8 describes the MSRP, which is a slightly more complex variant of the SRP, and proves that it is optimal with respect to minimizing unnecessary blocking under certain assumptions. Section 9 very briefly discusses the implementation of the SRP and its relation to more complex process models, such as Ada tasking. Section 10 summarizes the results and mentions some ongoing research.

2. Definitions

This section establishes notation, and defines the elements of our formal model. These elements include jobs, featherweight processes, resources, priorities, and preemption levels.

2.1. Jobs and processes

A *job* is a finite sequence of instructions to be executed on a single processor. It may have some branching control flow, but its maximum execution time and its other resource requirements must be fixed. A job might correspond to a subprogram in some programming

language. Jobs are considered to be the lowest level of schedulable activity in a system. A job may be preempted, but never intentionally waits. (Here waiting means suspending execution until a specified time or event, as opposed to blocking because a needed resource is busy.) Names of the forms J, J', J'', ... and J_i denote jobs.

A *job execution* is an instance of execution of a specific job, in response to a *job execution request*. Each job execution request *arrives* at some time, $Arrival(\mathcal{J})$. The execution of J in response to request $\mathcal{J}$ starts at some time, $Start(\mathcal{J})$, where $Arrival(\mathcal{J}) \leq Start(\mathcal{J})$. Requests that have arrived, but for which the corresponding executions have not yet completed are called *pending*. (Note that the pending jobs include both those that have not started and those that have started execution but have not finished yet.) Names of the forms $\mathcal{J}$, $\mathcal{J}'$, $\mathcal{J}''$, ... and $\mathcal{J}_i$ denote both job execution requests and job executions.

A *featherweight process* (*process* for short) is a higher level abstraction. Every job belongs to one of a fixed finite set of processes, $\mathcal{P}_1$, ..., $\mathcal{P}_n$. Each process $\mathcal{P}_i$ is characterized by an (infinite) sequence of job execution requests $\mathcal{J}_{i,1}$, $\mathcal{J}_{i,2}$, ... A process is *periodic* if the interval between successive execution requests is a constant (called the *period*); otherwise it is *aperiodic*. The jobs requested by each process are assumed to belong to a finite set, which are known *a priori*. Names of the forms $\mathcal{P}$ and $\mathcal{P}_i$ always denote processes.

There should be no more than one execution of any job going on at the same time. This may be taken as an assumption, or as a consequence of other assumptions we will make: that each job has a static preemption level and that there is only one processor. Thus, it is usually not necessary to be very careful about distinguishing jobs from job executions and job execution requests. The current execution of job J may be referred to by the same name as the job, i.e., J. In particular, if we say *job J* is actively doing something (such as holding or requesting a resource), we mean *the current execution of job J*.

2.2. *Resources*

An execution of a job requires the use of a processor and runtime stack space, and may require certain other serially reusable nonpreemptable resources. Allocation of processor time, stack space, and nonpreemptable resources to jobs is governed by *processor and resource allocation policies*.

We assume there is a single processor, which is preemptable, and a finite set of *resources*, R_1, ..., R_m. These resources are all nonpreemptable and serially reusable. For each resource R there is a fixed number of units in the system, N_R. Names of the forms R and R_i always denote resources.

A job acquires an allocation of a nonpreemptable resource by executing a request instruction. Formally, an *allocation* is a triple (J, R, m), where J is a job, R is a nonpreemptable resource, and m is the number of units requested. The number of units being requested must be less than or equal to N_R. The job making a request must wait to execute its next instruction until the allocation is granted. While a job is waiting for a resource allocation the job (and the request) are said to be *blocked*. After the allocation is granted, the job holds it until the job executes an instruction that releases it. While the job holds an allocation the allocation is said to be *outstanding*.

The sequence of instructions performed by the job between the request and release operations for a resource allocation is called a *critical section* of the job for that resource. Note that there is no implication of serialization between critical sections for the same resource, if the resource has more than one unit. Each job is required to request and release resources in Last-In-First-Out (LIFO) order; so critical sections of the same job can only overlap if they are properly nested.

A critical section is *trivial* if it involves a resource that cannot cause any blocking. (For example, in Section 6, we will show shared stack resources are trivial under the SRP.) An *outermost nontrivial* critical section is nontrivial and is not nested within any other nontrivial critical section. For bounding priority inversion, we will only be interested in the execution times of *outermost nontrivial* critical sections.

Without loss of generality, semaphores and reader/writer locks can be treated as special cases of multiunit resources. For a binary semaphore, $N_R = 1$. For a reader/writer lock, N_R can be any number greater than or equal to the number of jobs that may request R. While writing, a job needs to hold all N_R units, thus blocking both readers and writers. While reading, a job needs to hold an allocation of one unit, which blocks writers but does not block any other readers.

Example. Suppose J_1, J_2, and J_3 are jobs, where the relationships of the jobs' critical sections are as shown by Figure 1. Here, an operation of the form *request* (J_i, R_j, m) means the job J_i is requesting m units of resource R_j. A *release* operation releases the most recently acquired resource allocation. (Since we assume resources are released in LIFO order, the resource and number of units are uniquely determined.) The relationship of jobs to resources in this example is also shown, schematically, in Figure 2. The arrows indicate the *may request* relationship between jobs and resources, and are labeled with the number of units the jobs may request. We are supposing $N_{R_1} = 3$, $N_{R_2} = 1$, $N_{R_3} = 3$, resource R_2 behaves as a binary semaphore, R_3 behaves as a reader/writer lock, and R_1 behaves as a more general multiunit resource.

J_1	J_2	J_3
... request $(J_1, R_2, 1)$; ... request$(J_1, R_1, 3)$; ... release; ... release; ... request$(J_1, R_3, 1)$; ... release; ...	... request$(J_2, R_3, 3)$; ... request$(J_2, R_2, 1)$; ... release; ... release; ... request$(J_2, R_1, 2)$; ... release; ...	... request$(J_3, R_3, 1)$; ... request$(J_3, R_1, 1)$; ... release; ... release; ...

Figure 1. The critical sections of J_1, J_2 and J_3.

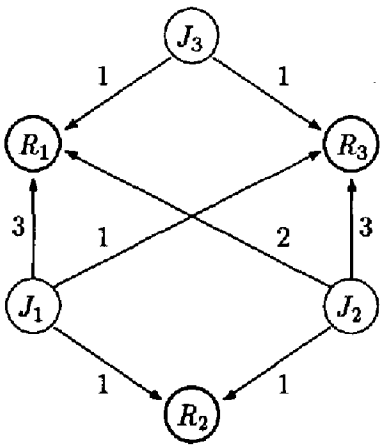

Figure 2. The resource graph for J_1, J_2 and J_3.

2.3. *Blocking*

The resource allocation policy is constrained to block a request (at least) when there are insufficient resources available to satisfy the request. We call such a conflict a *direct blockage*. Since we are assuming a job never makes a request that exceeds the total resources in the system, a job execution $\mathcal{J}$ can only be directly blocked if there is an identifiable set of other jobs that are *directly blocking* J, in the sense that there will be sufficient resources available to satisfy $\mathcal{J}$'s request as soon as one or more of these other jobs releases an allocation.

For a multiunit nonpreemptable resource, a request (J, R, m) is blocked directly iff $v_R < m$, where v_R denotes the number of units of R that are currently available (i.e., not outstanding). As a consequence of this definition, if R is a binary semaphore, any request for an allocation of R is blocked directly by any outstanding allocation of R to another job. Similarly, if R is a reader/writer lock, any request for an allocation of R is blocked directly by any outstanding write-allocation of R to another job, and any request for a write-allocation of R is blocked by any outstanding read-allocation of R to another job.

In addition to direct blocking, there may be other blocking. The resource allocation policy may choose to block some requests that are not blocked directly. In particular, it may do this to insure priority inversion is bounded. However, we will assume that the resource policy preserves the property that whenever a job J is blocked there is an identifiable set of other jobs that are *blocking* J; i.e., if some (or all) of the jobs blocking J released their current allocations J would become unblocked.

2.4. *Priorities*

Each job execution request $\mathcal{J}$ has a *priority*, $p(\mathcal{J})$. Priorities are values from some ordered domain, where $\mathcal{J}$ has higher priority than $\mathcal{J}'$ iff $p(\mathcal{J}) > p(\mathcal{J}')$. $\mathcal{J}$ having higher priority than $\mathcal{J}'$ means that expediting $\mathcal{J}$ is sufficiently important that completion of $\mathcal{J}'$ is permitted

to be delayed. For concreteness in our examples, we will use numeric priorities, where *larger values indicate greater urgency.* Examples of priority assignments of interest in real-time systems include RM and EDF.

A *processor allocation policy* determines which one of the pending unblocked jobs is allowed to use the processor. The primary objective of the processor and resource allocation policies is to expedite the highest priority pending job execution request. Normally, expediting the highest priority pending job means allocating the processor to that job, but this is not possible when the job is blocked. If a job J is blocked, the only way to expedite it is to expedite another (lower-priority) job that is blocking J, until the resources released by such jobs remove the cause of the blocking. This rule, which is called *priority inheritance* in (Sha, Rajkumar and Lehoczky 1987), can be applied transitively to expedite any directly blocked job that is not involved in a deadlock.

The rest of this article assumes that use of the processor is allocated to jobs preemptively, according to the priorities of requests and First-In-First-Out (FIFO) among jobs of equal priority, with priority inheritance. More precisely, let $\mathcal{J}_{cur}$ denote the currently executing request, and $\mathcal{J}_{max}$ denote the oldest highest priority pending job execution request. (Note that $\mathcal{J}_{cur}$ may, but need not, be the same as $\mathcal{J}_{max}$.) Under the priority inheritance policy, either $\mathcal{J}_{cur} = \mathcal{J}_{max}$ or there is a chain of job executions $\mathcal{J}_1, \ldots, \mathcal{J}_k$ such that $\mathcal{J}_1 = \mathcal{J}_{max}$, $\mathcal{J}_k = J_{cur}$, and $\mathcal{J}_i$ is blocked by $\mathcal{J}_{i+1}$ for $i = 1, \ldots, k - 1$. For there to be no multiple priority inversion, the resource allocation policy must insure that there is at most one such chain and the length of this chain never exceeds one.

2.5. *Preemption levels*

In addition to the priorities which are attached to individual requests for job executions, there are *preemption levels* (*level* for short), which are attached to jobs. Each job J has a preemption level $\pi(J)$. The level of a job is statically assigned to the job and applies to all execution requests for the job. The essential property of preemption levels is that a job J' is not allowed to preempt another job J unless $\pi(J) < \pi(J')$. This is also true for priorities. The reason for distinguishing preemption levels from priorities is to enable us to predict potential blocking, in the presence of dynamic priority schemes such as EDF scheduling.

2.5.1. Relative deadlines. For the specific priority assignments mentioned in this article, the preemption level of a job is based on the *relative deadline* of the job. The relative deadline of a job J is a fixed value, $D(J)$, such that if a request for execution of J arrives at time t, that execution must be completed by time $t + D(J)$. In other words, the relative deadline of a job is the size of the scheduling *window* in which each execution of the job must fit.

We define the *preemption levels* of jobs, $\pi(J)$, so that they are ordered inversely with respect to the order of relative deadlines; that is:

$$\pi(J) < \pi(J') \Leftrightarrow D(J') < D(J).$$

Suppose there are two jobs, J and J', with relative deadlines $D(J) = D$ and $D(J') = D'$, respectively. Suppose $\mathcal{J}$ is a job execution request of J such that $Arrival(\mathcal{J}) = t$, and $\mathcal{J}'$ is a request of J' such that $Arrival(\mathcal{J}') = t'$. In order for $\mathcal{J}'$ to preempt $\mathcal{J}$, we must have:

i. $t < t'$ (so $\mathcal{J}$ can get started);
ii. $p(\mathcal{J}) < p(\mathcal{J}')$ (so $\mathcal{J}'$ can preempt).

This is illustrated in Figure 3.

With EDF scheduling, $p(\mathcal{J}) < p(\mathcal{J}')$ iff $t' + D' < t + d$, so the essential property of preemption levels is satisfied; i.e., job J' is not allowed to preempt J unless $\pi(J) < \pi(J')$.

Example. An example will emphasize the difference between EDF priority and preemption level. Let $\mathcal{P}$ and $\mathcal{P}'$ be two periodic processes, each of which has a single job. Let the jobs be J and J', with relative deadlines 20 and 10, respectively. Preemption level 1 is assigned to J and preemption level 2 is assigned to J', since the relative deadline of J' is shorter than the relative deadline of J.

J' can never be preempted by J. This does not mean that requests for J' always have higher priority than those for J, or that we are allowing enforcement of preemption levels to cause priority inversion. It is just that the only way a request for J can have higher priority than a request for J' is if it arrived earlier, before the request for J' could have started, in which case it will have *no need* to preempt J.

Suppose request $\mathcal{J}$ arrives at time t, and a request $\mathcal{J}'$ arrives at time $t + 11$, as shown in Figure 4. Since the absolute deadline of $\mathcal{J}$ is $t + 20$ and the absolute deadline of $\mathcal{J}'$ is $t + 21$, $\mathcal{J}$ will have higher priority than $\mathcal{J}'$, and so $\mathcal{J}$ will not be preempted. On the other hand, if $\mathcal{J}'$ had arrived at time $t + 9$ its deadline would have been $t + 19$ and we would have had $p(\mathcal{J}) < p(\mathcal{J}')$, so $\mathcal{J}$ would be preempted. Thus preemption level is different from priority, but there is no priority inversion.

In addition to EDF scheduling, our preemption levels based on relative deadlines can be used with RM, *deadline monotone* (Leung and Whitehead 1982) (where $p(J) < p(J')$ iff $D(J') < D(J)$), and *static least-slack time* scheduling (where $p(J) < p(J')$ iff $D(J') - C(J') < D(J) - C(J)$, and $C(J)$ is the maximum execution time of job J).

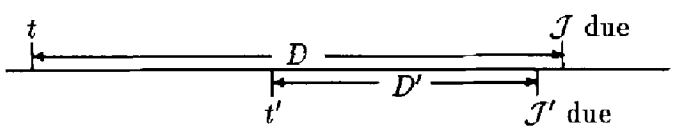

Figure 3. Preemption, in EDF scheduling.

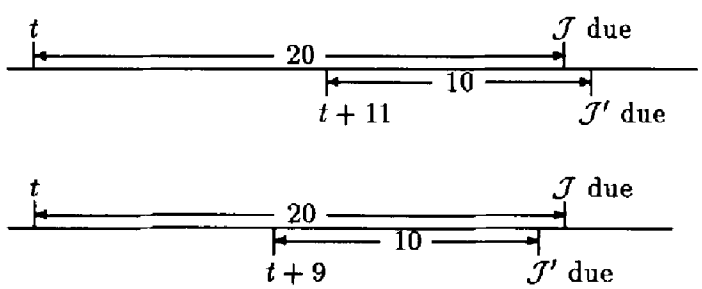

Figure 4. Preemption level *vs.* priority with EDF.

2.5.2. Abstract preemption levels. Although relative deadlines are the basis for preemption levels for all these examples, the theoretical results proven in this article do not depend on this. The only property of preemption levels on which these results do depend is the following condition:

$$p(\mathcal{J}) \leq p(\mathcal{J}') \text{ or } Arrival(\mathcal{J}) \leq Arrival(\mathcal{J}') \text{ or } \pi(J) > \pi(J'). \tag{2.1}$$

This is equivalent to requiring that if $\mathcal{J}$ has higher priority than $\mathcal{J}'$, but $\mathcal{J}$ arrives after $\mathcal{J}'$, then J must have a higher preemption level than J'. Note that our preemption levels based on relative deadlines do satisfy condition (2.1) above for all the priority assignments mentioned in this article, and condition (2.1) is sufficient to guarantee that J can preempt J' only if $\pi(J') < \pi(J)$.

Condition (2.1) and the other definitions enable us to prove the following lemma, which characterizes the relationships between allocation of processor time, preemption levels, priorities, and arrival times.

LEMMA 1. For every preempted job execution $\mathcal{J}$:

1. $p(\mathcal{J}) < p(\mathcal{J}_{cur})$;
2. $\pi(J) < \pi(J_{cur})$;
3. $Start(\mathcal{J}) < Arrival(\mathcal{J}_{cur})$.

Moreover, if $\mathcal{J}_{cur} \neq \mathcal{J}_{max}$, for every preempted or executing request $\mathcal{J}$, including $\mathcal{J}_{cur}$:

1. $p(\mathcal{J}) < p(\mathcal{J}_{max})$;
2. $\pi(J) < \pi(J_{max})$;
3. $Start(\mathcal{J}) < Arrival(\mathcal{J}_{max})$.

Proof. Recall from Section 2 that $\mathcal{J}_{cur}$ stands for the currently executing request and $\mathcal{J}_{max}$ stands for the oldest highest priority pending request. A request $\mathcal{J}$ cannot preempt another request $\mathcal{J}'$ unless $\mathcal{J}$ arrives after $\mathcal{J}'$ has started execution and $\mathcal{J}$'s priority is higher than $\mathcal{J}'$'s. By condition (2.1) this means the preemption level of J must also be higher than that of J'. These relations are transitive. It follows that $p(\mathcal{J}) < p(\mathcal{J}_{cur})$, $\pi(J) < \pi(J_{cur})$, and $Start(\mathcal{J}) < Arrival(\mathcal{J}_{cur})$. From the definition of $\mathcal{J}_{max}$, $p(\mathcal{J}) \leq p(\mathcal{J}_{max})$, for every pending $\mathcal{J}$. If $\mathcal{J}_{cur} \neq \mathcal{J}_{max}$ we have $p(\mathcal{J}) < p(\mathcal{J}_{max})$. If $\mathcal{J}_{max}$ arrived before $\mathcal{J}_{cur}$ started, it would have been chosen to execute ahead of $\mathcal{J}_{cur}$, so $Start(\mathcal{J}_{cur}) < Arrival(\mathcal{J}_{max})$. Given these two facts, from condition (2.1), we have $\pi(J_{cur}) < \pi(J_{max})$. These three relations then apply to jobs preempted by $\mathcal{J}_{cur}$, transitively.

3. Preventing deadlock and multiple priority inversion

To strictly bound priority inversion, we want to require that the resource management policy not allow deadlock or *multiple priority inversion*—that is, situations where a job is blocked for the duration of more than one outermost nontrivial critical section of a lower priority

job. Given the model and assumptions described above, it is possible to derive general conditions that are sufficient to guarantee there is no deadlock or multiple priority inversion. (Moreover, we will show in Section 6 that these conditions are necessary if all jobs share a single stack.) The conditions are:

> To prevent deadlock, a job should not be permitted to start until the resources currently available are sufficient to meet the maximum requirements of the job. (3.1)

> To prevent multiple priority inversion, a job should not be permitted to start until the resources currently available are sufficient to meet the maximum requirement of any single job that might preempt it. (3.2)

Note that condition (3.1) above is similar to Havender's collective allocation approach to avoiding deadlock (1968), but Havender proposes to actually allocate all the resources to the job before it starts. In contrast, condition (3.1) only requires that the resources be *available*. They need only be allocated to the job during the critical sections in which it actually needs to use them. A higher-priority job may preempt and use the resources between these critical sections, if the available quantities are sufficient to meet its requirements.

LEMMA 2. Condition (3.1) guarantees that a job cannot block after it starts.

Proof. Suppose condition (3.1) is enforced. We have assumed there are only finitely many jobs, and that a second execution of a job is not permitted to start while an execution of the same job is active. Thus, an executing job can be preempted by only finitely many other jobs. We will prove by induction on N that if $\mathcal{J}$ is preempted by no more than N other jobs, $\mathcal{J}$ executes to completion without blocking.

Suppose the induction hypothesis fails for some N; that is, suppose $\mathcal{J}$ is blocked making request (J, R, m), and N is the number of other jobs that preempt $\mathcal{J}$ during its lifetime. By the condition (3.1), at least m units of R were available when $\mathcal{J}$ started. If $N = 0$, no other job preempts $\mathcal{J}$, so these resources will still be available when $\mathcal{J}$ requests them, and $\mathcal{J}$ will execute to completion without blocking. If $N > 0$, suppose job $\mathcal{J}_H$ preempts $\mathcal{J}$. By condition (2), all the resources required by $\mathcal{J}_H$ are available when $\mathcal{J}_H$ preempts. Since any job that preempts $\mathcal{J}_H$ also preempts $\mathcal{J}$, the induction hypothesis guarantees that $\mathcal{J}_H$ executes to completion without blocking, as will any job that preempts $\mathcal{J}_H$, transitively. Since all of the jobs that preempt $\mathcal{J}$ execute to completion without blocking, the priority inheritance policy will not permit $\mathcal{J}$ to resume execution until there are no higher priority pending jobs. At this point, since the completing jobs must have released all their resource holdings, the only resources outstanding will be those held by $\mathcal{J}$ and jobs preempted by $\mathcal{J}$. It follows that $\mathcal{J}$ cannot be blocked.

THEOREM 3. Condition (3.1) is sufficient for preventing deadlock.

Proof. Observe that a job cannot hold resources until it starts, and by Lemma 2 it cannot be blocked after it starts. Since a job cannot be blocked while holding resources, there can be no deadlock.

THEOREM 4. Assuming condition (3.1) is enforced, condition (3.2) is sufficient to prevent multiple priority inversion.

Proof. Suppose there is multiple priority inversion. By Lemma 2, the only way a job $\mathcal{J}_H$ can be subject to such multiple priority inversion is if there are two or more lower priority jobs, $\mathcal{J}$ and $\mathcal{J}'$, that execute while $\mathcal{J}_H = \mathcal{J}_{max}$. The priority inheritance policy only allows such lower priority jobs to execute if they are blocking $\mathcal{J}_H$. Both $\mathcal{J}$ and $\mathcal{J}'$ must have started executing before $\mathcal{J}$ arrives, and one of them must have preempted the other. Without loss of generality, suppose $\mathcal{J}$ preempted $\mathcal{J}'$. Condition (3.2) must have been violated when $\mathcal{J}$ was allowed to start.

The Stack Resource Policy enforces conditions (3.1) and (3.2) indirectly, by imposing stronger conditions, that are simpler to check.

4. Stack resource policy

This section defines the SRP, and proves that it *works*; that is, it enforces direct blocking requirements, without allowing multiple priority inversion or deadlock.

4.1. Ceilings

4.1.1. Abstract ceilings. The SRP enforces conditions (3.1) and (3.2) in terms of *preemption ceilings* (*ceilings*). Each resource R is required to have a *current ceiling*, $\lceil R \rceil$, which is an integer-valued function of the set of outstanding allocations of R. The correctness of the SRP does not depend on the exact definition of $\lceil R \rceil$, but only requires that ceilings be related to priorities and preemption levels by the following condition:

> If J is currently executing or can preempt the currently executing job, and may request an allocation of R that would be blocked directly by the outstanding allocations of R, then $\pi(J) \leq \lceil R \rceil$. (4.1)

The SRP will work with any definition of ceiling that satisfies these conditions. One specific definition of ceiling, that satisfies condition (4.1), is given below. However, freedom to choose a slightly different definition is a convenience when one implements the SRP. For this reason, the definition and proofs of the SRP are based on abstract ceilings, characterized only by condition (4.1).

4.1.2. Specific ceilings. For a multiunit nonpreemptable resource R, $\lceil R \rceil$ may be defined to be $\lceil R \rceil_{\nu_R}$, where ν_R denotes the number of units of R that are currently available and $\lceil R \rceil_{\nu_R}$ denotes the maximum of zero and the preemption levels of all the jobs that may be blocked directly when there are ν_R units of R available. That is:

$$\lceil R \rceil_{\nu_R} = max(\{0\} \cup \{\pi(J) \mid \nu_R < \mu_R(J)\}),$$

R	N_R	$\mu_R(1)$	$\mu_R(2)$	$\mu_R(3)$	$\lceil R \rceil_0$	$\lceil R \rceil_1$	$\lceil R \rceil_2$	$\lceil R \rceil_3$
R_1	3	3	2	1	3	2	1	0
R_2	1	1	1	0	2	0	0	0
R_3	3	1	3	1	3	2	2	0

Figure 5. Ceilings of resources.

where μ_R is the maximum requirement of job J for R. (Note that this definition satisfies condition (4.1).)

Example. The ceilings of the resources for the example shown in Figures 1 and 2 are shown in Figure 5, under the assumption that $\pi(J_i) = p(\mathcal{J}_i) = i$ for $i = 1, 2, 3$.

4.1.3. Ceilings and deadlock prevention. Given condition (4.1), the following relationships can be established between the current ceiling of a resource and conditions (3.1) and (3.2) of Section 3.

LEMMA 5. Suppose $\mathcal{J} = \mathcal{J}_{max}$, $\mathcal{J}$ is not executing, and R is a resource.

(a) If $\lceil R \rceil < \pi(J)$ then there are sufficiently many units of R available to meet the maximum requirement of J. (Condition (3.1) is satisfied for J and R.)
(b) If $\lceil R \rceil \leq \pi(J)$ then there are sufficiently many units of R available to meet the maximum requirement of every job that can preempt $\mathcal{J}$. (Condition (3.2) is satisfied for J and R.)

Proof. To show (a), suppose $\lceil R \rceil < \pi(J)$ but the maximum request of J for R cannot be satisfied. By condition (4.1), $\pi(J) \leq \lceil R \rceil$—a contradiction.

To show (b), suppose $\lceil R \rceil \leq \pi(J)$, but for some job $\mathcal{J}_H$ that can preempt $\mathcal{J}$ the maximum requirement of $\mathcal{J}_H$ for R cannot be satisfied. By condition (4.1), $\pi(J_H) \leq \lceil R \rceil$, but for $\mathcal{J}_H$ to preempt $\mathcal{J}$ we must have $\pi(J) < \pi(J_H)$—a contradiction.

THEOREM 6. If no job J is permitted to start until $\lceil R_i \rceil < \pi(J)$, for every resource R_i, then:

(a) no job can be blocked after it starts;
(b) there can be no deadlock;
(c) no job can be blocked for longer than the duration of one outermost nontrivial critical section of a lower priority job.

Proof. Part (a) follows directly from part (a) of Lemma 5 and Lemma 2. Part (b) follows directly from part (a) of Lemma 5 and Theorem 3. Part (c) follows from part (b) of Lemma 5 and Theorem 4.

4.2. Definition of the SRP

The Stack Resource Policy is defined as follows: Each job execution request $\mathcal{J}$ is blocked from starting execution (i.e., from receiving its initial stack allocation) until $\mathcal{J}$ is the oldest highest priority pending request, and if $\mathcal{J}$ would preempt an executing job,

$$\forall_{\substack{R_i \\ i=1,\ldots,m}} \lceil R_i \rceil_{\nu_{R_i}} < \pi(J). \tag{4.2}$$

Thereafter, once $\mathcal{J}$ has started execution, all its resource requests are granted immediately, without blocking.

This can be stated more simply, by introducing the concept of a system-wide *current ceiling*. At any instant of time, let the *current ceiling* of the system, $\bar{\pi}$, be the maximum of the current ceilings of all the resources. That is,

$$\bar{\pi} = max\{\lceil R_i \rceil \mid i = 1, \ldots, m\}.$$

The SRP preemption test (4.2) then reduces to

$$\bar{\pi} < \pi(J). \tag{4.3}$$

If $\mathcal{J}$ is blocked, it is blocked by the preemption test. The other jobs holding resources R such that $\pi(J) \leq \lceil R \rceil$ are said to be *blocking* $\mathcal{J}$.

Note that the SRP does not restrict the order in which resources may be acquired, in contrast to the ordered resource allocation approach of (Havender 1968). It is also less restrictive than another approach of (Havender 1968), collective allocation. That is, even though the condition $\bar{\pi} < \pi(J)$ is tested before the job J starts to execute, the SRP does not at that time actually allocate all the resources that may ever be requested by J. They are only allocated when requested, and are released as soon as they are not needed. Thus, even if J will later request some allocation of R that would block a higher level job J_H, J_H is free to preempt until J actually requests enough of R to block J_H directly.

Note also that the SRP preemption test (4.2) has the effect of imposing priority inheritance (that is, an executing job that is holding a resource resists preemption as though it inherits the priority of any jobs that might need that resource), though the effect is accomplished without modifying the formal priority of the job, $p(\mathcal{J})$. This is important for understanding the meaning of the preemption test, above, and the rest of this article.

Example. Two possible executions of jobs J_1, J_2, and J_3 under the SRP are shown in Figure 6 and Figure 7. The solid horizontal lines indicate which job is executing, while the barred lines indicate the relative value of the current ceiling, $\bar{\pi}$. Figure 6 shows what happens if J_1 acquires R_2 before J_2 and J_3 arrive. Since $\lceil R_2 \rceil_0 = 2$, J_2 is unable to preempt J_1 after it acquires R_2, and since $\lceil R_1 \rceil_0 = 3$, J_3 is unable to preempt J_1 after it acquires all of R_1. J_3 preempts J_1 as soon as J_1 releases R_1, and J_2 preempts J_1 as soon as J_1 releases R_2. Figure 7 shows what happens if J_3 arrives before J_1 acquires R_1; it is able to preempt immediately, but J_2 still has to wait for J_1 to release R_2. (Note that in both cases the current ceiling happens not to change when J_3 acquires R_3; that is because J_3 only needs one unit, and $\lceil R_3 \rceil_2 = 2$ in our example.)

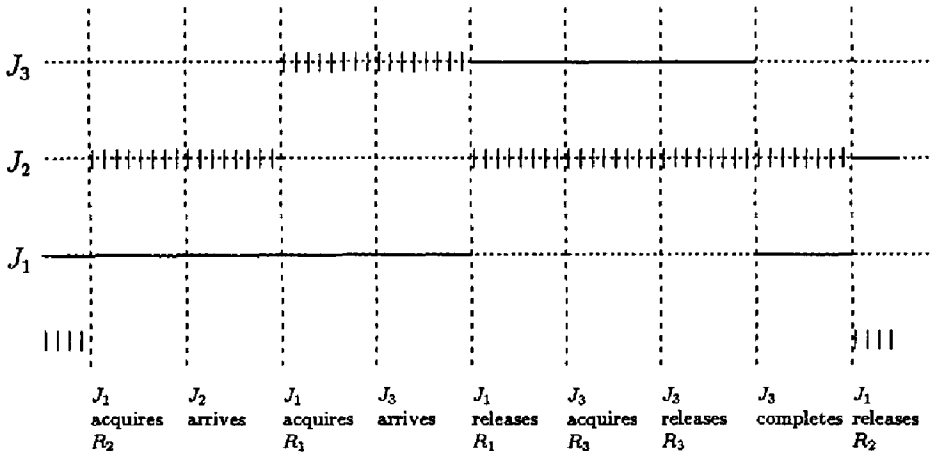

Figure 6. J_3 arrives after J_1 acquires R_1.

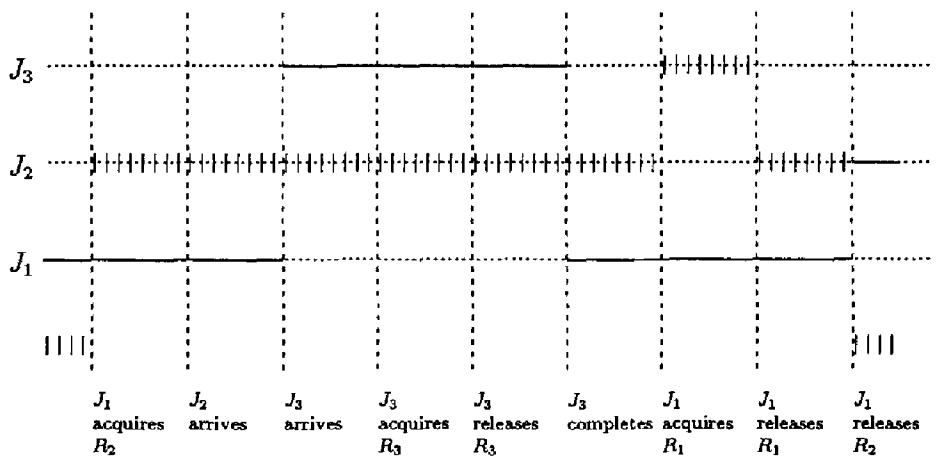

Figure 7. J_3 arrives before J_1 acquires R_1.

4.3. Blocking properties of the SRP

The SRP enforces direct blocking, prevents deadlock, and strictly bounds priority inversion. We shall prove this.

COROLLARY 7. If no job J is permitted to start until $\bar{\pi} < \pi(J)$, then:

(a) no job can be blocked after it starts;
(b) there can be no deadlock;

(c) no job can be blocked for longer than the execution time of one outermost nontrivial critical section of a lower priority job.

Proof. This follows from Theorem 6, since $\lceil R \rceil \leq \bar{\pi}$ for every resource R.

Since no job is blocked after it starts executing, there can be no transitive blocking. A consequence of this and priority inheritance is that whenever the processor is not idle it is executing either the oldest highest-priority request, $\mathcal{J}_{max}$, or a job that is directly blocking $\mathcal{J}_{max}$. These facts and Lemma 1 enable us to prove the following theorem, which says that the current job, J_{cur}, is the only job that may block $\mathcal{J}_{max}$.

THEOREM 8. If $\mathcal{J}_{max}$ is blocked, it will become unblocked no later than the first instant that $\mathcal{J}_{cur}$ is not holding any nonpreemptable resources.

Proof. Suppose $\mathcal{J}_{max}$ is blocked. Since $\mathcal{J}_{max}$ is not executing, it must have arrived after $\mathcal{J}_{cur}$, and so by condition (2.1), $\pi(J_{cur}) < \pi(J_{max})$.

Consider the moment that $\mathcal{J}_{cur}$ releases the last resource allocation it was holding when $\mathcal{J}_{max}$ arrived. Suppose $\mathcal{J}_{max}$ remains blocked. Since such resources are required to be requested and released in LIFO order, the same resource allocations are outstanding as when $\mathcal{J}_{cur}$ started, and $\bar{\pi}$ is the same as it was then. Since $\mathcal{J}_{cur}$ was not blocked then, $\bar{\pi} < \pi(J_{cur})$. It follows that $\mathcal{J}_{max}$ is not blocked now.

THEOREM 9. The SRP never grants a request that is directly blocked.

Proof. This is true because the SRP never allows a job to start unless it can be guaranteed not to block, as shown by Corollary 7. More directly, suppose J makes a request for resource R that is directly blocked. By condition (4.1), $\pi(J) \leq \lceil R \rceil$. By definition of $\bar{\pi}$, $\lceil R \rceil \leq \bar{\pi}$, but the SRP would not let J start unless $\bar{\pi} < \pi(J)$—a contradiction.

5. Schedulability with the SRP

Corollary 7 is sufficient to support the schedulability results for RM and EDF scheduling cited in Section 1. We can also derive a tighter schedulability test for the EDF policy with semaphores than the one proved in (Chen and Lin 1989). To show this, we restrict our process model so that it more closely resembles that used in previous work (Liu and Layland 1973). In particular, let there be a one-to-one correspondence between processes and jobs. Suppose there are n (periodic or aperiodic) processes, $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, ordered by increasing relative deadlines of the corresponding jobs, $\{J_1, \ldots, J_n\}$. Let the relative deadlines all be positive, and let each be less than or equal to the period of the corresponding process. Let T_i denote the period or minimum interarrival time of $\mathcal{P}_i$, let D_i denote the relative deadline of J_i, and let C_i denote the maximum execution time of J_i. Let B_i denote the maximum execution time of the longest nontrivial critical section of every job J_k such that $D_i < D_k$ and $i \neq k$, or zero if there is no such J_k. (This maximum includes all the critical section of other jobs that might subject J_i to priority inversion.) Assume there is a system start time, before which no jobs are requested, and that $D_i \leq T_i$ for every process.

THEOREM 10. A set of n (periodic and aperiodic) processes is schedulable by EDF shedluling with SRP semaphore locking if

$$\forall_{k=1,\ldots,n} \left(\sum_{i=1}^{k} \frac{C_i}{D_i} \right) + \frac{B_k}{D_k} \leq 1.$$

Proof. Assume the theorem is false. Let t be the first time a job misses its deadline. Let t' be the last time before t such that there are no pending job execution requests with arrival times before t' and deadlines before or at t. Since no requests can arrive before system start time, t' is well defined. Since deadlines are all positive, $t' < t$. By choice of t and t', there is no idle time in $[t', t]$.

Let $\mathfrak{A}$ be the set of jobs that arrive in $[t', t]$ and have deadlines in $[t', t]$. By choice of t', there are pending requests of jobs in $\mathfrak{A}$ at all times during the interval $[t', t]$. Thus, by the EDF priority assignment, the only jobs that will be allowed to start in $[t', t]$ will be in $\mathfrak{A}$. These jobs can only be preempted by other jobs in $\mathfrak{A}$.

If a job not in $\mathfrak{A}$ executes in $[t', t]$ it must have started before t' and have been preempted while holding some resource allocation that is blocking a job in $\mathfrak{A}$. Once such a job releases the resources that are causing the blocking, it cannot execute further in $[t', t]$.

Suppose there are two such jobs, $\mathcal{J}_b$ and $\mathcal{J}_c$. Both of these jobs must have been holding resources when they were preempted, and since they are blocking some job(s) in $\mathfrak{A}$, these resources must have current ceilings higher than or equal to $\pi(J_i)$ for some $\mathcal{J}_i$ in $\mathfrak{A}$. (If they are not both blocking the same job in $\mathfrak{A}$, we can choose the $\mathcal{J}_i$ with lower preemption level.) Since both $\mathcal{J}_b$ and $\mathcal{J}_c$ are in nontrivial critical sections at time t', one must have preempted the others; say $\mathcal{J}_b$ preempts $\mathcal{J}_c$. For some resource R held by $\mathcal{J}_b$, we have $\pi(J_i) < \lceil R \rceil$. Since $\mathcal{J}_i$ is capable (transitively) of preempting $\mathcal{J}_c$, we also have $\pi(J_c) < \lceil R \rceil$, but then $\mathcal{J}_c$ should not have been able to preempt $\mathcal{J}_b$—a contradiction.

It follows that there can be at most one job, $\mathcal{J}_j$, that blocks any job in $\mathfrak{A}$, and that this job can only execute for as long as it takes to exit its outermost nontrivial critical section. Therefore J_j is the only job not in $\mathfrak{A}$ that can execute in $[t', t]$. (Actually, by Theorem 8, we know that $\mathcal{J}_j$ must be the job that is executing at time t', if it exists.) If there is no such job J_j, only the jobs in $\mathfrak{A}$ can execute in $[t', t]$.

Note that if J_j uses more than one nonpreemptable resource simultaneously it may execute more than once, since it may unblock several jobs in stages, as it releases successive resources. However, the total execution time of J_j in $[t', t]$ cannot be any longer than it takes J_j to release its last resource, since after that J_j will be preempted continuously by jobs in $\mathfrak{A}$.

Let $\Delta = t' - t$. By choice of t', $D_i \leq \Delta$, for every J_i in $\mathfrak{A}$. If a job J_j is executing at time t', then $\Delta < D_j$. Since the jobs are ordered by increasing value of D_i it follows that there exists a k such that $\mathfrak{A} \subseteq \{J_1, \ldots, J_k\}$, $D_k \leq \Delta$, and $k < j$.

The total length of time that J_j executes in $[t', t]$ is bounded by the longest time J_j uses a resource. This is bounded by B_i for each job J_i in $\mathfrak{A}$, since $D_i < D_j$. In particular, the maximum execution time of J_j in $[t', t]$ is bounded by B_k.

For every J_i in $\mathfrak{A}$, the demand for CPU time in $[t', t]$ is not more than KC_i, where

$$K = \left\lfloor \frac{\Delta - D_i}{T_i} \right\rfloor + 1.$$

In $[t', t]$ there is no idle time and the only jobs executing are J_j and those in $\mathcal{A}$. Since there is an overflow, the total demand for processor time in $[t', t]$ exceeds Δ, so

$$B_k + \sum_{i=1}^{k} \left(\left\lfloor \frac{\Delta - D_i}{T_i} \right\rfloor + 1 \right) C_i > \Delta.$$

Since $\lfloor X \rfloor \leq X$, we have

$$\frac{B_k}{\Delta} + \sum_{i=1}^{k} \frac{\Delta - D_i + T_i}{T_i \Delta} C_i = \frac{B_k}{\Delta} + \sum_{i=1}^{k} \left(1 + \frac{T_i - D_i}{\Delta} \right) \frac{C_i}{T_i} > 1,$$

and since $D_i \leq \Delta$ for $i = 1, \ldots, k$,

$$\frac{B_k}{D_k} + \sum_{i=1}^{k} \left(1 + \frac{T_i - D_i}{D_i} \right) \frac{C_i}{T_i} = \frac{B_k}{D_k} + \sum_{i=1}^{k} \frac{C_i}{D_i} > 1.$$

COROLLARY 11. A set of n (periodic and aperiodic) jobs with relative deadlines equal to their respective periods is schedulable by EDF scheduling if

$$\forall_{k=1,\ldots,n} \left(\sum_{i=1}^{k} \frac{C_i}{T_i} \right) + \frac{B_k}{T_k} \leq 1.$$

Proof. Since $D_i = T_i$, $T_i/D_i = 1$.

Note that this result is tighter than Chen and Lin's Theorem 4 (quoted in Section 1), since there is only one blocking term in the sum. However, the proof does not appear to depend on the use of preemption levels or early blocking. Thus, we believe it can also be applied to the dynamic SRP as described in (Chen and Lin 1989).

6. Stack sharing

This section discusses the sharing of runtime stack space between jobs, and shows that the SRP supports stack sharing without allowing unbounded priority inversion or deadlock. Supporting stack sharing was the original motivation for the development of the SRP, and in particular for the choice of early blocking.

6.1. The motivation for stack sharing

In a conventional process-based model of concurrent programming, such as Ada tasking, each process needs its own runtime stack. The region allocated to each stack must be large

enough to accommodate the maximum stack storage requirement of the corresponding process. Storage is reserved for the stack continuously, both while the process is executing and between executions.

In some hard realtime applications there may be thousands of actions that are to be performed at different times in response to appropriate triggering events. In a conventional process model each action would be implemented by a process, which waits for a triggering event and then executes the action. A problem with this kind of design is that a great deal of storage may be required for the stacks of all the waiting processes—storage which is unused most of the time.

For example, suppose that there are four processes $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$, with respective priorities 1, 2, 2, and 3 (3 is the highest priority). Figure 8 shows the stack usage of these processes during a possible execution sequence, assuming each process is allocated its own stack space. In the figure, $\mathcal{P}_1$ is running at time t_1; $\mathcal{P}_2$ preempts at time t_2, and completes at time t_3, allowing $\mathcal{P}_1$ to resume; $\mathcal{P}_3$ preempts at time t_4; $\mathcal{P}_4$ preempts at time t_5, and completes at time t_6, allowing $\mathcal{P}_3$ to resume; $\mathcal{P}_3$ completes at time t_7, allowing $\mathcal{P}_1$ to resume. The top of each process's stack varies during the process's execution, as indicated by the solid lines. The regions of storage reserved for each stack remain constant, as indicated by the dashed lines.

The requirement for stack space can be dramatically reduced by using the *featherweight* processes described in this article. A key feature of the featherweight process model is that when a job execution completes, all resources required by that execution may be released. In particular, stack space may be allocated when the job begins execution and completely freed when it completes.

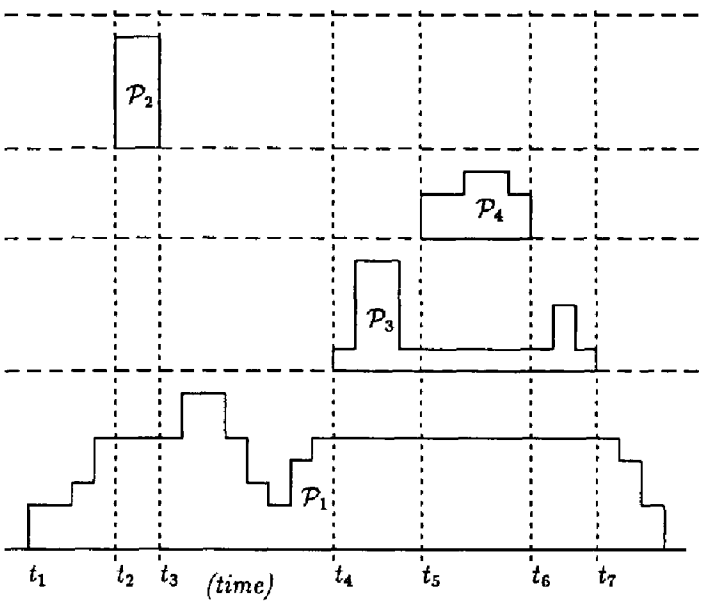

Figure 8. One stack per process.

6.2. *How stack sharing works*

One obvious way to allocate stack space is to partition jobs into groups that cannot preempt one another, and allocate a stack to each group. The SRP provides a more elegant solution than this, by allowing even jobs that preempt one another to share a single stack.

Suppose all jobs share a single stack. When a job J is preempted by a job J', J continues to hold its stack space and J' is allocated space immediately above it on the stack. Figure 9 shows what would happen to the jobs of Figure 8 if they shared a single stack. The space between the two dashed horizontal lines represents space that will no longer be needed, since the priorities of $\mathcal{P}_2$ and $\mathcal{P}_3$ guarantee they will never need to occupy stack space at the same time.

Stack sharing may result in very large storage savings if there are many more processes than preemption levels. For example, suppose we have 100 jobs, with 10 jobs at each of 10 preemption levels, and each job needs up to 10 kilobytes of stack space. Using a stack per job, 1000 kilobytes of storage would be required. In contrast, using a single stack, only 100 kilobytes of storage would be required (since no more than one job per preemption level could be active at one time). The space savings is 900 kilobytes; that is, 90%.

6.3. *Stack usage assumptions*

Let us now assume stack sharing is allowed, and consider how it affects scheduling. Each job may either have its own individual runtime stack, or share the use of a runtime stack

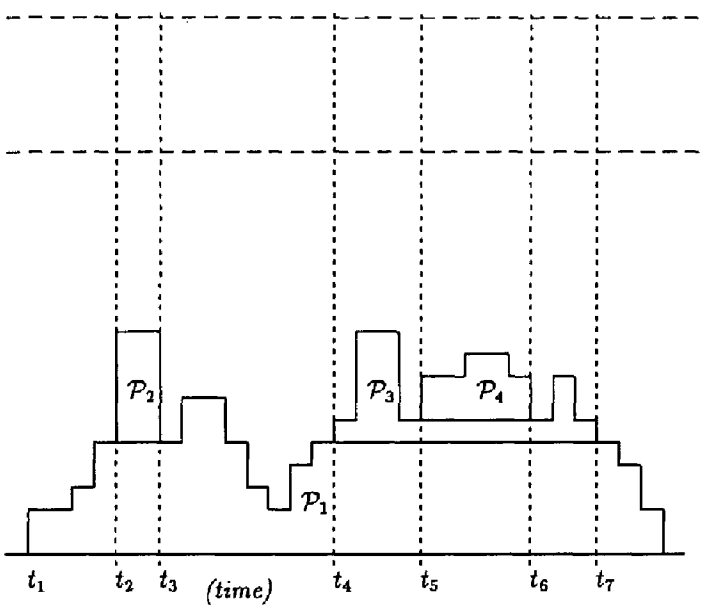

Figure 9. Single stack, for all processes.

with a collection of other jobs. Although runtime stack space is a nonpreemptable resource, it must be treated differently from the other multiunit nonpreemptable resources we have considered so far. The big difference is that the location of the requested space, rather than the quantity, is what matters.

Based on the way in which programming language implementations typically use the runtime stack, we make the following assumptions:

1. Every job requires an initial allocation of at least one cell of stack space before it can start execution, and cannot relinquish that space until it completes execution. This means the entire execution of each job is a *critical section* with respect to the stack which it is using.
2. After a job starts execution, if it makes any request that is blocked it must continue to hold its stack space while it is blocked.
3. A stack storage request can be granted to a job if and only if the job is not yet holding any stack space or it is at the top of the stack it is using.
4. Only a job at the top of a stack may execute, since an executing job may need to increase its stack size at any time.

Due to these assumptions, the request for the initial stack allocation of each job may be treated as part of the request for job execution, and subsequent use of the stack by that job may be allowed without explicit request and release operations.

6.4. How the SRP avoids stack blocking

The problem with stack sharing is that it can cause blocking. For a shared stack, a job J is *directly blocked* iff there is another job J' holding the space immediately above J on the stack, so that J's part of the stack cannot grow without overflowing into the holdings of J'. For this situation to occur, J' must have preempted J; J will be blocked until J' completes and releases all of its stack space. That is, once a job is preempted by another job on the same stack it cannot be resumed until the preempting job completes. Such *stack blocking* effectively requires that use of the processor be allocated according to a LIFO policy. Fortunately, since any job that preeempts must have higher priority than the job it preempts, this requirement is consistent with priority preemptive scheduling.

When there are other nonpreemptable resources, stack blocking can easily lead to deadlock. For example, suppose jobs $\mathcal{J}_H$ and $\mathcal{J}_L$ both use a nonpreemptable resource (e.g., binary semaphore) R. Suppose $\mathcal{J}_H$ preempts while $\mathcal{J}_L$ is holding R. Job $\mathcal{J}_H$ will start to execute, occupying the stack space above $\mathcal{J}_L$, but will eventually try to obtain R. It cannot do this, since $\mathcal{J}_L$ is still holding R. Unfortunately, $\mathcal{J}_H$ is now also blocking $\mathcal{J}_L$, by sitting on top of its stack space.

Note that it is possible to solve the problem of stack deadlock without bounding priority inversion very strictly. For example, reconsider the jobs $\mathcal{J}_H$ and $\mathcal{J}_L$ described above. Suppose a deadlock prevention scheme is used, that prevents job $\mathcal{J}_H$ from preempting while R is in use. Now suppose there is an intermediate priority job $\mathcal{J}_M$, which preempts while $\mathcal{J}_L$ is holding resource R, but before $\mathcal{J}_H$ arrives. Since $\mathcal{J}_L$ cannot execute while $\mathcal{J}_M$ is sitting

on its stack, $\mathcal{J}_H$ will suffer priority inversion until $\mathcal{J}_M$ completes its whole execution, and then until $\mathcal{J}_L$ completes the section in which it is using R.

We showed in Section 4.3 that the SRP will prevent deadlock and bound priority inversion to the duration of a single critical section. It does this by enforcing conditions (3.1) and (3.2), using the preemption ceilings for all resources that can cause blocking. The only requirement for preemption ceilings is condition (4.1). If we can define the ceiling of a stack in a way that satisfies (4.1), the SRP will handle stack blocking.

Because of the assumptions we have made about stack usage, the stack space held by a job J can only block jobs that it might preempt; that is, jobs with lower preemption levels. It follows that condition (4.1) imposes no restriction on the current ceiling of a stack. Therefore, the ceiling of a stack can be defined to be anything we want. We define it to be zero.

By defining the ceiling of a shared stack to be zero, we can ignore stack usage in computation of $\bar{\pi}$, and so stack usage can never cause blocking. *Critical sections* with respect to stack usage are therefore trivial, and can be ignored in the computation of the priority inversion bound, B_i. Note, however, that this does not mean stack resources can be ignored completely; where there is stack sharing, the preemption operation must be treated as a request for stack resources, and may block.

7. Comparison to PCP

Since the SRP is a refinement of the PCP it is natural to compare the two techniques, to see what the differences are and what consequences they have.

7.1. Review of the PCP

As a basis for the comparison, we review the definition of the PCP. Each job is assumed to be requested cyclically, with a fixed priority. There is a fixed set of semaphores, each of which has a priority ceiling.

Sha, Rajkumar, and Lehoczky (1987) define the priority ceiling of a semaphore to be "the priority of the highest priority job that may lock this semaphore." They refine this concept for readers and writers in (Sha, Rajkumar and Lehoczky 1988), defining the "absolute priority ceiling" of an object to be the priority of the highest priority job that may lock the object for reading or writing, and the "write priority ceiling" of an object to be the priority of the highest priority job that may lock it for writing. Chen and Lin (1989) define the "dynamic priority ceiling" of a semaphore to be "the priority of the highest priority task that may lock S in the current effective task set."

If S is a semaphore, let $c(S)$ denote its priority ceiling. Let S^* be the semaphore that has the highest priority ceiling among all semaphores locked by jobs other than $\mathcal{J}_{cur}$, if there are any. For notational simplicity, let $c(S^*)$ be defined to be zero when S^* is undefined (i.e., when there are no locked semaphores).

The Priority Ceiling Protocol consists of the following policy: When a job $\mathcal{J}$ requests a semaphore S it will be blocked unless

$$c(S^*) < p(\mathcal{J}).$$

(That is, $\mathcal{J}$ is blocked until it has strictly higher priority than all the priority ceilings of all the semaphores locked by jobs other than $\mathcal{J}$.) If $\mathcal{J}$ blocks, the job that holds S^* is said to be blocking $\mathcal{J}$ and inherits $\mathcal{J}$'s priority. A job $\mathcal{J}$ can always preempt another job executing at a lower priority level as long as $\mathcal{J}$ does not request any semaphore.

7.2. Differences of the SRP

In most respects, the SRP is a consistent extension of the PCP. The SRP relaxes restrictions of the PCP in the following ways:

1. The PCP assumes that the preemption level of each job is the same as its priority, which is fixed, except for mode changes (Sha, Rajkumar and Lehoczky 1989). With the SRP, the preemption level of a job may be different from its priority, and while the preemption level of a job is required to be static, the priority may be dynamic. Ceilings are based on preemption levels, rather than priorities. This allows the SRP to be applied directly to EDF scheduling without resort to dynamic recomputation of ceilings.
2. The PCP model views each process as a sequence of requests for the same *type* of job. We make the same assumption for the purposes of EDF schedulability analysis, but do not make this assumption in the other proofs. Thus, an SRP process may consist of requests for several different jobs, with different preemption levels.
3. The original PCP handles binary semaphores, and has been extended to handle reader-writer locks by distinguishing two kinds of priority ceilings. The SRP allows multiunit resources, by treating the ceiling of a resource as a function of the number of units currently available.
4. The PCP does not allow stack sharing. The SRP treats a shared stack as a resource with ceiling zero.

The only way in which the SRP is not a consistent extension of the PCP is in earlier blocking. When the PCP blocks a job it does so at the time it makes its first resource request, which is some time after it has started execution. In contrast, the SRP schedules every job as though it starts by requesting use of a shared stack (whether or not it actually does). This eliminates the possiblity of a job blocking after it has started execution, and eliminates the extra context switches associated with blocking and unblocking. (This issue of context switches is discussed further in Section 7.5.)

A technical difference between the SRP and PCP is Theorem 8, which says that if a job is blocked by the SRP, it is blocked while trying to preempt, and only by the one job it is trying to preempt. In contrast, the PCP only guarantees that if a job is blocked it will be blocked at its first resource request and then will not be blocked again.

7.3. A comparative example

To illustrate the differences between the PCP and SRP, we present an example, involving three jobs with the structure shown in Figure 10. Consider the following scenario, in which the jobs execute under the PCP:

J_H	J_M	J_L
... request(J_H, S, 1); ... release; ...	...	... request(J_L, S, 1); ... release; ...

Figure 10. The jobs J_H, J_M, J_L.

1. A high priority job, J_H preempts the processor from a lower priority job, J_M, and executes for a while.
2. J_H is forced to allow a lower priority job J_L (preempted earlier by J_M) to resume execution, because J_L is holding a resource needed by J_H.
3. J_L releases the resource needed by J_H, and J_H resumes execution.
4. J_H completes, and J_M resumes execution.

This is illustrated in Figure 11. The solid horizontal lines indicate which job is executing. The barred horizontal lines indicate the relative value of $c(S^*)$, the current ceiling as defined for the PCP.

For comparison, consider the corresponding scenario under the SRP:

1. J_H waits until $\pi(J_H) > \bar{\pi}$, then begins execution, preempting J_L. (Note that J_M and J_H are forced to wait until J_L releases S, since $\pi(J_H) \leq \bar{\pi}$.)
2. J_H completes and J_M resumes execution.

This is illustrated in Figure 12. The solid horizontal lines indicate which job is executing. The barred horizontal lines indicate the relative value of $\bar{\pi}$.

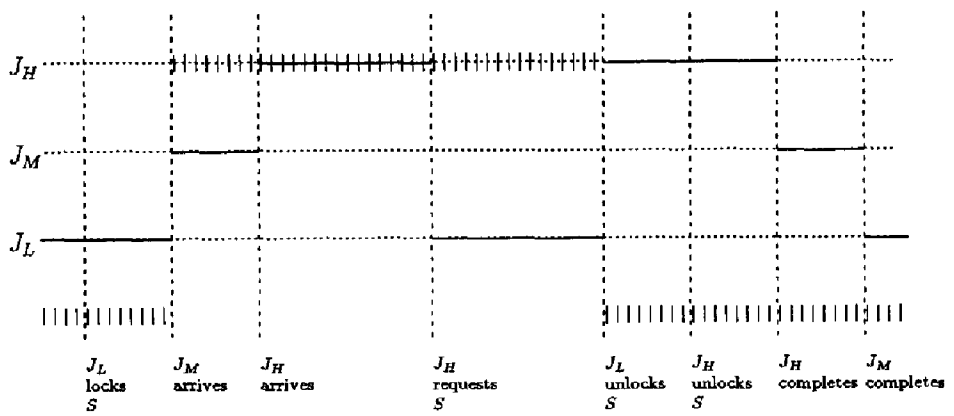

Figure 11. Execution with PCP.

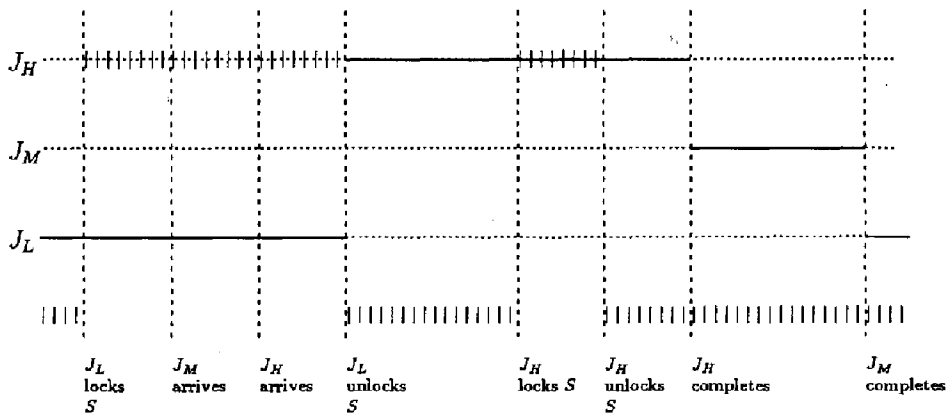

Figure 12. Execution with SRP.

Note that the two ceilings, $c(S^*)$ and $\bar{\pi}$, differ slightly, since $c(S^*)$ does not include the semaphores held by the current job. However, this makes no practical difference, since for both the PCP and SRP, whenever a job performs its first potentially blocking operation $c(S^*) = \bar{\pi}$. That is, it would be a valid optimization of the PCP to ignore $c(S^*)$ after the first potentially blocking operation, just as the SRP does.

7.4. Priority inversion

One weakness of the SRP is that it will block a job in some situations where the PCP would not. This is a consequence of early blocking, and consequent pessimistic assumptions about resource usage. It is not a major problem, since we have shown that the priority inversion due to such blocking is bounded by the execution time of the longest nontrivial critical section of a lower priority job. However, it does mean that a comparison of the effectiveness of the SRP and PCP in preventing unbounded priority inversion must take into account the characteristics of specific jobs.

The most obvious situation where the SRP will block and the PCP would not is when a job uses no nonpreemptible resources. This is illustrated by job J_M in the example of Figures 11 and 12. J_M is subject to priority inversion with the SRP, but not with the PCP. The example also shows that this difference is not a clear-cut advantage for the PCP, since the extra priority inversion for J_M can sometimes reduce the priority inversion for J_H.

This problem cannot arise if there is stack sharing, since the stack is a nonpreemptable resource. If there is no stack sharing, the defect can be reduced by modifying the SRP to exempt jobs that are known to use no nonpreemptable resources from the preemption requirement $\bar{\pi} < \pi(J)$. However, there will remain some cases where whether a job uses such resources is decided by data dependent logic within the job. On the average, such a job may suffer less priority inversion with the PCP than with the SRP. Moreover, if the data-dependent control flow of the job causes it to have the longest execution time in the

cases where it uses no nonpreemptable resources, the unnecessary blocking may cause the worst-case combined execution and blocking time of that job to be longer with SRP than PCP, reducing schedulability.

The comparison is simpler if we assume every job uses at least one nonpreemptable resource. We can prove that in this case the SRP doesn't allow any longer priority inversion than the PCP.

THEOREM 12. Under the SRP the maximum priority-inversion time of any job that uses a nonpreemptable resource is no longer than it would be under the PCP.

Proof. Suppose a set of jobs and a sequence of job execution requests is given. We will compare the maximum priority-inversion time of some job J under both policies. Since we are comparing against the PCP, which only supports binary semaphores and static priorities, we will assume that the only resources are semaphores and that the priority of each job execution request is the same as the preemption level of the job. Under these assumptions, the only significant difference between the SRP and the PCP is that the SRP blocks earlier.

Let $\mathcal{J}$ be a request for J that achieves the maximum priority inversion under the SRP. From Theorem 8 we know that $\mathcal{J}$ can only be subject to priority inversion from the current job, $\mathcal{J}_{cur}$. Thus, $\mathcal{J}_{cur}$ is holding a semaphore S that blocks $\mathcal{J}$ from preempting. That is, $\pi(J) \leq \lceil S \rceil$. Since we are assuming preemption level equals priority, $p(\mathcal{J}) \leq \lceil S \rceil$.

The same order of events may happen with the PCP. That is, a higher-priority job execution request for J may arrive while J_{cur} is holding S. Under the PCP, J would preempt. Suppose J later requests some resource. We have $p(\mathcal{J}) \leq \lceil S \rceil \leq \lceil S^* \rceil$. Since this is the blocking condition for the PCP, this request by J would be blocked. $\mathcal{J}$ would therefore be subject to priority inversion until J_{cur} releases S, which is at least as long as under the SRP.

7.5. Context switches

A positive consequence of the early blocking policy of the SRP is a reduction in unnecessary context switches, as compared with the PCP. The cost of context switches can be significant for certain processors. Architectural features that increase the relative cost of context switching include large register sets, address-translation lookaside buffers, instruction pipelining, prefetching, and cache memory for data and instructions. For such architectures, the early blocking property of the SRP may be important, because it reduces the need for context switches.

For comparison, consider resource allocation policies that bound priority inversion but allow late blocking, such as the PCP and the Priority Limit Protocol (Sha, Rajkumar and Lehoczky 1987). These policies all permit the scenario shown in Figures 10, 11, and 12. The diagrams reveal that with the PCP the execution of J_H requires four context switches, but with the SRP it only requires two context switches. This generalizes.

THEOREM 13. The SRP requires at most two context switches for a job execution request.

Proof. This is a consequence of early blocking, and can be seen immediately from the definition of the SRP. Since a job cannot be blocked after it starts execution, the only context switches are one switch from the job that is preempted to the job that is requested and one switch back when the preempting job completes.

THEOREM 14. The PCP, and any other policy that waits to block a job until it makes a resource request, may require four context switches for a job execution request, for any job that shares a nonpreemptable resource with some lower-priority job.

Proof. Let J be any job such that there is a lower-priority job J_L and a resource R such that both J and J_L lock R. If J is requested while J_L is running and has locked R, there will be four context switches: (1) from J to J_L, when J preempts; (2) from J_L to J, when J tries to lock R; (3) from J_L to J, when J_L unlocks R; (4) from J back to J_L, when J completes.

Together, Theorem 13 and Theorem 14 say that the upper bound on the number of context switches caused by a request with the SRP is half of the maximum for the PCP. This improvement is due to earlier blocking.

Note that this improvement is dependent on the preempting job being one that requires a nonpreemptable resource—which we are assuming to be the normal case. If the job uses no nonpreemptable resources, there would be no extra context switch with the PCP. Moreover, it is possible that whether a job uses such resources is determined by data-dependent control flow. If the control flow causes the job to have the longest execution time in the cases where it uses no nonpreemptable resources, the cost of the extra context-switches may not contribute to the worst-case combined execution and context-switching time of that job, and so it may not be considered significant with respect to schedulability.

8. Minimal SRP

The SRP was designed to solve the problem of stack blocking, but it is not an optimal solution to that problem, in the sense of causing no unnecessary blocking. If we are willing to make the blocking test more complicated, we can define such an optimal resource management policy. The Minimal SRP (MSRP) imposes the minimum blocking necessary to insure there is no deadlock or multiple priority inversion, assuming there is a single stack and RM or EDF priorities are used.

8.1. Definition of the MSRP

The Minimal SRP is defined as follows: Each job execution request $\mathcal{J}$ is blocked from starting execution (i.e., from receiving its initial stack allocation) until $\mathcal{J}$ is the oldest highest priority pending request, and if $\mathcal{J}$ would preempt an executing job, one of the following conditions is satisfied:

$$\bar{\pi} < \pi(J); \tag{8.1}$$

$\bar{\pi} = \pi(J)$ and the presently available resources are sufficient for J to execute to completion without direct blocking. (8.2)

Once J has started execution, all subsequent resource requests by J are granted immediately, without blocking.

Note that for the specific definitions of ceilings given here, condition (8.2) is equivalent to

$$\bar{\pi} = \pi(J) \text{ and } \forall R \ (\mu_R(J) \leq \nu_R).$$

8.2. *Blocking properties of the MSRP*

Outside of the extra complexity of its preemption test, the MSRP has the same desirable properties as the SRP.

THEOREM 15. The MSRP prevents deadlock and multiple priority inversion.

Proof. To show that the MSRP prevents deadlock and multiple blocking, we need to show that it enforces conditions (3.1) and (3.2). Lemma 5 shows that condition (8.1) enforces conditions (3.1) and (3.2). The second part of condition (8.2) is equivalent to (3.1), so the only remaining question is whether (8.2) enforces (3.2).

Suppose (8.2) is satisfied by J_{max}, but (3.2) is not. Let J_H be a job that might preempt J_{max}, such that there are not sufficient resources available for J_H to execute to completion without direct blocking. That means J_H may request an allocation of some resource R that is blocked directly by currently outstanding allocations. Since J_H must also be able to preempt the current job, by condition (4.1), $\pi(J_H) \leq \lceil R \rceil$. By definition of $\bar{\pi}$, $\lceil R \rceil \leq \bar{\pi}$. It follows that $\pi(J_H) \leq \pi(J_{max})$, but then J_H would not be able to preempt J_{max}—a contradiction.

Theorems 8 and 9 also apply to the MSRP, with the same proofs, except that the relation $\bar{\pi} < \pi(J)$ is replaced by $\bar{\pi} \leq \pi(J)$.

8.3. *Minimality of the SRP*

We will show that the MSRP imposes the minimal blocking necessary to prevent unbounded priority inversion and deadlock under conditions of stack sharing. We will start by showing that the conditions (3.1) and (3.2), which were shown in Section 3 to be sufficient to prevent deadlock, are necessary to prevent deadlock if there is stack sharing.

THEOREM 16. Deadlock prevention condition (3.1) is necessary if there is a single stack.

Proof. To see that condition (3.1) is necessary, suppose there is a single stack, and $\mathcal{J}$ is allowed to start execution while there are insufficient resources to meet its maximum requirements. If $\mathcal{J}$ makes its maximum requests for all resources, it will be blocked for some request. The resources required to unblock $\mathcal{J}$ will be held by jobs that sit below $\mathcal{J}$ on the stack, and these jobs are in turn blocked by the stack allocations of the jobs above them, culminating in $\mathcal{J}$—a deadlock.

THEOREM 17. Assuming condition (3.1) is enforced, deadlock prevention condition (3.2) is necessary if there is stack sharing.

Proof. To see that condition (3.2) is necessary, suppose there is a single stack and $\mathcal{J}$ is permitted to start while there are insufficient resources available to meet the maximum requirements of some job $\mathcal{J}_H$ that can preempt $\mathcal{J}$. Condition (3.1) guarantees that $\mathcal{J}_H$ will be blocked if it attempts to preempt $\mathcal{J}$. Due to stack blocking, no other job may execute to release resources until $\mathcal{J}$ completes, but then $\mathcal{J}_H$ will still be blocked. Thus, $\mathcal{J}_H$ will be forced to wait for at least two lower priority jobs—a multiple priority inversion situation.

Up to now, the only assumption made about preemption levels is condition (2.1), from which it follows that J can preempt J' only if $\pi(J') < \pi(J)$. With RM or EDF scheduling, if preemption levels of jobs are assigned based on relative deadlines, we also have

$$\pi(J') < \pi(J) \Leftrightarrow \text{some execution of } J \text{ can preempt some execution of } J'. \quad (8.3)$$

The *only if* direction follows directly from (2.1). To see that the *if* direction holds, suppose $\mathcal{J}$ arrives after $\mathcal{J}'$ has started and J has a shorter relative deadline that J'. By RM, EDF, or deadline monotone scheduling, $\mathcal{J}$ will have higher priority than $\mathcal{J}'$ and preempt. This will happen unless J and J' have harmonic periods and there is precise control over phasing of requests.

THEOREM 18. Under the assumption of a single shared runtime stack, and condition (8.3), any resource allocation policy that does not permit deadlock or multiple blocking must block every request that is blocked by the MSRP.

Proof. Suppose a resource allocation policy is given, and it does not block some request that would be blocked by the MSRP. We will show that a job may be blocked by more than one other job, or be deadlocked. Like the SRP, the MRSP only blocks jobs before they start. Suppose job J is allowed to start when one of the two MSRP blocking conditions is true and job J_{cur} is executing. We know that $p(\mathcal{J}_{cur}) < p(\mathcal{J})$, and since $\mathcal{J}_{cur}$ is executing we know $\mathcal{J}$ arrives after $\mathcal{J}_{cur}$ starts. By condition (2.1), this means $\pi(J_{cur}) < \pi(J)$. Since J is blocked from preempting by the MSRP, we have two cases:

1. $\pi(J) < \bar{\pi}$. There is some outstanding resource R and a job J_H such that $\pi(J) < \pi(J_H)$, and J_H can be blocked directly by the currently outstanding allocations of R. By condition (8.3), J_H may preempt J and be blocked directly by R. Since the allocations outstanding when J started were already enough to block J_H directly, J_H will be blocked until some job J', below J, resumes execution and releases its allocation of R. J' cannot resume execution until J completes, due to stack blocking. Thus, J_H will be blocked by both J and J'. This is at least double blocking, and if J_H is allowed to start before blocking, it will deadlock, since it will be blocking J's stack.
2. $\pi(J) = \bar{\pi}$ and J may make a request, for some resource R, that is blocked directly. Since we are assuming stack sharing, we know that J_{cur} will be blocked by J's stack allocation if it tries to resume execution before J completes. By assumption, J's request

for R may be blocked directly. The resource allocations blocking J must be held by jobs that are on the stack. J_{cur} must complete before any of these jobs can resume and release their allocations of R. There is deadlock.

9. Practical considerations

9.1. Implementation of the SRP

The SRP can be implemented very simply and efficiently. The implementation is similar to that of the PCP (Sha, Rajkumar and Lehoczky 1987), but the locking operations are simpler, since they cannot block, do not require any blocking test, and never require a context switch. The blocking test is also simplified slightly, since it does not need to distinguish the ceilings of resources held by the current job from those of resources held by other jobs. The ceilings are static, and so may be precomputed and stored in a table. A stack may be used to keep track of the current ceiling. When a resource R is allocated its current state, ν_R, is updated, and $\bar{\pi}$ is set to $\lceil R \rceil_{\nu_R}$ iff $\bar{\pi} < \lceil R \rceil_{\nu_R}$. The old values of ν_R and $\bar{\pi}$ are pushed onto the stack. When resource R is released, the values of $\bar{\pi}$ and ν_R are restored from the stack. If the restored ceiling is lower than the previous ceiling, a dispatching procedure is invoked to check whether a waiting higher level job should be allowed to preempt.

The dispatching procedure checks the priority queue to see if $\mathcal{J}_{max}$ is different from $\mathcal{J}_{cur}$ and satisfies the preemption criterion, $\bar{\pi} < \pi(J_{max})$. If $\mathcal{J}_{max}$ passes this test, the identity of $\mathcal{J}_{cur}$ is pushed on the stack, runtime stack space is allocated to $\mathcal{J}_{max}$, and $\mathcal{J}_{max}$ starts execution. If $\mathcal{J}_{max}$ fails the test, the dispatcher simply returns. Whenever a job completes, $\mathcal{J}_{cur}$ is restored from the stack. The dispatcher is called whenever $\mathcal{J}_{max}$ changes, due to arrival of a higher priority request or completion of the old $\mathcal{J}_{max}$.

9.2. Relationship to conventional process models

As realtime systems grow in complexity they strain the limits of existing software technology. One response to this increasing complexity has been movement toward process-based models of concurrent programming. Such models have been very successful in the design of operating systems and interactive computer applications. One manifestation of this movement is the multitasking model of the Ada programming language (Military Standard Ada Programming Language 1983). Ada has been mandated by the U.S. Department of Defense for all mission-critical software. Another manifestation is the development by the IEEE of a proposed realtime extension to its standard POSIX operating system interface (IEEE Computer Society 1988), which is derived from the UNIX[2] process model.

Unfortunately, these conventional process models are too general to permit direct application of the SRP. The SRP is based on a *featherweight* process model. We view this model as an *alternative*, which is superior to conventional process models for hard realtime applications, because it allows better *a priori* schedulability analysis. However, there are situations where other considerations may dictate the use of a less restricted model.

This problem of mismatching models has already been addressed by people who have attempted to apply the PCP to Ada (Borger and Rajkumar 1989). Their approach has been

to identify a set of restrictions that define a subset of the more general model that can be mapped into the PCP's model. This same approach can be applied with the SRP, with very little difference.

A conventional process can be viewed as a featherweight process if the sequence of instructions between each blocking operation (e.g., rendezvous, delay) and the next is bounded, no nonpreemptable resources are retained while the process is blocked, and resources are released in LIFO order. Such a process can use the SRP to synchronize with other processes, if no attempt is made to share stack space. The SRP preemption test is applied each time a process becomes unblocked; that is, the next sequence of instructions executed by the process from that point to the next blocking operation is viewed as a new *job*.

The SRP appears to be applicable to the same range of Ada task systems as the PCP. Passive tasks, sometimes also called *monitor* or *server* tasks, may be implemented as collections of procedures, interlocked via semaphores. Other tasks can be treated as featherweight processes, if they satisfy the restrictions stated in the previous paragraph. Stack sharing may be possible between such tasks, if further restrictions are imposed, such as that the tasks do not block within any subprogram calls or *declare* blocks. In this fashion, a sufficiently simple Ada task system may be transformed into a system of featherweight processes and semaphores.

This idea of doing optimizing transformations on special kinds of Ada tasks, based on *idioms*, is well known and has been discussed by several authors (Borger and Rajkumar 1989; Habermann and Nassi 1989; Hilfinger 1982; Giering and Baker 1989). However, it has not yet been widely implemented. It is not yet clear how far it will be practical to go with this approach for the SRP.

10. Conclusions and further research

Starting from the motivation of supporting stack sharing, we have shown that the PCP can be extended in three ways, and that earlier blocking may be advantageous in some situations. We have defined the SRP and MSRP, two extensions of the PCP that incorporate these extensions and early blocking.

One strength of the SRP and MSRP is that they support EDF priorities, as well as fixed priorities. EDF scheduling permits higher utilization than fixed-priority scheduling, but fixed-priority scheduling has an advantage of *stability*—that is, it guarantees lower priority jobs will not prevent higher priority jobs from meeting their deadlines during periods of processing overload. Since the SRP supports both fixed and EDF priorities, it is possible to run EDF jobs as *background* in a system where the critical jobs are scheduled in *foreground* according to a RM policy. In particular, it appears that the schedulability result of (Liu and Layland 1973) on using a mixture of RM and EDF policies can be applied to this situation, if B_k is subtracted from the processor availability function.

The SRP has been implemented. In continuing research, we plan to conduct some empirical studies of the SRP versus other scheduling and resource allocation policies. We also hope to extend the theory in several directions.

Several of the ideas presented in this article, which are embodied in the SRP, appear to have wider applicability. These ideas are:

1. Distinguishing preemption level from priority.
2. Early blocking.
3. Stack sharing.

Further research, to explore wider application of these ideas, seems warranted. More specifically, it seems that the concepts of preemption level and early blocking can be applied to much of the work that has been done on RM scheduling and the PCP, including the multiprocessor version of the PCP. Already, we have observed that the SRP is compatible with the aperiodic server concepts of (Sprunt, Sha and Lehoczky 1989).

While this article was being reviewed, Ghazalie (Ghazalie 1990) has shown that the deferred and sporadic server models can be adpated to improve average response times with EDF scheduling, as they are known to do with RM scheduling. The basic idea is to associate a deadline with each replenishment of server execution time. This enables the server to be scheduled within the EDF paradigm. For a sporadic server, there is no reduction in schedulability. For a deferred server, a special term must be added to the schedulability test, to account for the effect of the server.

Acknowledgments

This article is a more formal development of ideas first proposed in (Baker, Malec and Wilson 1989). The motivation to reduce wasted space for stacks of inactive tasks came from discussions with Russ Wilson of Boeing Aerospace and Electronics (BAE), regarding a BAE project which involved thousands of tasks. The importance of avoiding unnecessary context switches, due to the increasingly high relative cost of context switches in recent generations of 32-bit microprocessors, also came from conversations at BAE, with Russ Wilson, Carl Malec, and Greg Scallon. The idea of modifying the idea of the PCP to address these two issues crystalized during a discussion of the need for lighter-weight alternatives to Ada tasking at the July 1989 meeting of the Ada Runtime Environment Working Group, in Seattle.

Mike Victor, of Raytheon, pointed out that an executive based on the basic stack-sharing model described here has been in use at Raytheon for over fifteen years, though there are apparently no published descriptions of that executive. Lui Sha pointed out the possible advantage of the PCP in the case where there is no stack sharing and resource usage depends on data-dependent control flow. Ted Giering pointed out that the proof of Theorem 10 does not require early blocking.

The author is indebted to the referees for their careful readings of this article and their constructive criticisms. In particular, the author is especially thankful to the referees for suggesting that the applicability to multiunit resources be made more explicit, and that schedulability results such as Theorem 10 should be explicitly included.

Notes

1. The abbreviation "iff" is used for "if and only if."
2. UNIX is a registered trademark of AT&T.

References

U.S. Department of Defense. 1983. *Military Standard Ada Programming Language*, ANSI/MILSTD1815A, Ada Joint Program Office.

Baker, T.P., and Scallon, G.L. 1986. An Architecture for Real-Time Software Systems. *IEEE Software*, 50–59; reprinted in *Hard-Real-Time Systems*, Washington, DC: IEEE Press (1988).

Baker, T.P. 1989. A Fixed-Point Approach to Bounding Blocking Time in Real-Time Systems. Technical Report, Department of Computer Science, Florida State University, Tallahassee, FL 32306.

Baker, T.P., Malec, C., and Wilson, R. 1989. Practical Tasking. Boeing Aerospace and Electronics Company white paper.

Baker, T.P. 1990. Preemption vs. Priority, and the Importance of Early Blocking. *Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, VA (May): 44–48.

Baker, T.P. 1990. A Stack-Based Resource Allocation Policy for Realtime Processes. *Proceedings of the IEEE Real-Time Systems Symposium*.

Borger, M.W., and Rajkumar, R. 1989. Implementing Priority Inheritance Algorithms in an Ada Runtime System. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

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

Coffman, E.G. Jr., and Denning, P.J. 1973. *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall.

Garey, M.R., and Johnson, D.S. 1979. *Computers and Intractability*. New York: W.H. Freeman.

Ghazalie, T. 1990. Improving Aperiodic Response with Deadline Scheduling. Master's Thesis, Florida State University.

Giering, E.W. III, and Baker, T.P. 1989. Toward the Deterministic Scheduling of Ada Tasks. *Proceedings of the IEEE Real-Time Systems Symposium*, 31–40.

Habermann, A.N., and Nassi, I.R. 1980. Efficient Implementation of Ada Tasks. Technical report, Department of Computer Science, Carnegie Mellon University.

Havender, J.W. 1968. Avoiding Deadlock in Multitasking Systems. *IBM Systems Journal* 7, 2: 74–84.

Hilfinger, P.N. 1982. Implementation Strategies for Ada Tasking Idioms. *Proceedings of the AdaTEC Conference on Ada*, Arlington, VA: 26–30.

Holt, R.C. 1971. On Deadlock in Computer Systems. Ph.D. Thesis, TR 71-91, Department of Computer Science, Cornell University.

IEEE Computer Society. 1988. IEEE Standard Portable Operating System Interface for Computer Environments, Washington, DC: IEEE Press.

Leung, J.Y.-T., and Merrill, M.L. 1980. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters* 11, 3: 115–118.

Leung, J.Y.-T., and Whitehead, J. 1982. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation* 2: 237–250.

Liu, C.L., and Layland, J.W. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM* 20.1: 46–61.

Mok, A.K.-L. 1983. Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment. Ph.D. Thesis, MIT.

Rajkumar, R., Sha, L., Lehoczky, J.P., and Ramamritham, K. 1989. An Optimal Priority Inheritance Protocol for Real-Time Synchronization. Technical report, Carnegie Mellon University (submitted for publication).

Rajkumar, R., Sha, L., and Lehoczky, J.P. 1988. Real-Time Synchronization Protocols for Multiprocessors. *Proceedings of the Real-Time System Symposium*, IEEE, 259–272.

Sha, L., Lehoczky, J.P., and Rajkumar, R. 1986. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. *Proceedings of the IEEE Real-Time Systems Symposium*, 181–191.

Sha, L., Rajkumar, R., and Lehoczky, J.P. 1987. Priority Inheritance Protocols, An Approach to Real-Time Synchronization. Technical report CMU-CS-87-181, Carnegie Mellon University.

Sha, L., Rajkumar, R., and Lehoczky, J. 1988. A Priority Driven Approach to Real-Time Concurrency Control. Technical report, Carnegie Mellon University.

Sprunt, B., Sha, L., and Lehoczky, J. 1989. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real Time Systems* 1, 1: 27–60.

Sha, L., Rajkumar, R., and Lehoczky, J. 1989. Mode Change Protocols for Priority-Driven Preemptive Scheduling. *Real Time Systems* 1, 3: 243–264.

Bic, L., and Shaw, A.C. 1988. The Logical Design of Operating Systems. Englewood Cliffs NJ: Prentice-Hall.