A Stack-Based Resource Allocation Policy for Realtime Processes

T.P. Baker[1]
Department of Computer Science
Florida State University
Tallahassee, FL 32304-4019

Abstract

The Stack Resource Policy (SRP) is a resource allocation policy which permits processes with different priorities to share a single runtime stack. It is a refinement of the Priority Ceiling Protocol (PCP) of Sha, Rajkumar and Lehoczky, which strictly bounds priority inversion and permits simple schedulability tests.

With or without stack sharing, the SRP offers improvements over the PCP, by: (1) unifying the treatment of stack, reader-writer, and multiunit resources, and binary semaphores; (2) applying directly to some dynamic scheduling policies, including EDF, as well as to static priority policies; (3) with EDF scheduling, supporting a stronger schedulability test; (4) reducing the maximum number of context switches for a job execution request by a factor of two. It is at least as good as the PCP in reducing maximum priority inversion.

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.

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 Ada[1], the programming language mandated by the the U.S. Department of Defense for all mission-critical software. Unfortunately, process-based models such as Ada tasking do not impose strong enough structural constraints on software to support verification of timing requirements, or efficient management of resources.

Some progress has been made toward reconciling the process model with the need for predictable timing, by mapping restricted process models onto classical scheduling models. One approach uses off-line scheduling [12,7], based on deterministic scheduling theory [6]. A more flexible approach, exemplified by [4], uses on-line preeemptive priority scheduling and is based on the work of [11].

This paper is motivated by concern for another aspect of adapting the process model to hard realtime requirements: the efficient allocation of memory for process' runtime stacks. Conventionally, 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, where there may be thousands of actions that are to be performed at different times in response to appropriate triggering events, a great deal of storage may be required for the stacks of waiting processes — storage which is unused most of the time.

The requirement for stack space can be dramatically reduced by using a more primitive model of concurrency, closer to classical deterministic scheduling theory. In such a model, the work is divided into simple schedulable units, which we will call *jobs*. The key difference between a job and a process is that when a job execution completes, all resources required by the job may be released. In particular, stack space may be allocated when the job begins execution and completely freed when it completes.

A conventional process may be viewed as a sequence of jobs, if the set of sequences of instructions executed by the process between waits is finite and no resources are retained between waits. Each sequence of instructions (i.e. execution path) executed by the process between waits is a job.

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. The only special requirement is that if J is preempted it cannot resume execution until all the jobs that occupy stack space above it have completed. Since these jobs must have higher

[1] This work supported in part by grants from the U.S. Office of Naval Research (N00014-87-J-1166) and the Florida High Technology Industry Research Council. This paper to appear in the proceedings of the IEEE Real-Time Systems Symposium, 1990.

CH2933-0/90/0000/0191$01.00 © 1990 IEEE

priority, this requirement is consistent with priority scheduling.

Stack sharing may result in rather large storage savings if there are many more processes than relative priority levels. For example, if each job needs up to 10 kilobytes of stack space and there are 10 jobs at each of 10 priority levels, the space savings is 900 kilobytes; that is, 90%.

A problem with stack sharing is that it can cause blocking. When jobs are not independent, this leads easily to deadlock. For example, suppose processes $\mathcal{P}_1$ and $\mathcal{P}_2$ both use a nonpreemptable resource (e.g. binary semaphore) R. Suppose $\mathcal{P}_2$ starts to execute while $\mathcal{P}_1$ is holding R. Process $\mathcal{P}_2$ will start to execute, occupying the stack space above $\mathcal{P}_1$, but will eventually try to obtain exclusive access to R. It cannot do this, since $\mathcal{P}_1$ is still holding R. Unfortunately, $\mathcal{P}_2$ is now also blocking $\mathcal{P}_1$, by sitting on top of its stack space.

Deadlock is only part of the problem. Even if deadlock is avoided, stack blocking can cause priority inversion[15] – the situation where a higher priority job is blocked by a lower priority job. Priority inversion is bad, since it reduces the effectiveness of priority preemptive scheduling, resulting in unnecessarily missed deadlines for high priority jobs.

To strictly bound priority inversion in a system with stack sharing it is essential that the system resource management policy take a unified view, managing the stack along with the CPU and other resources. The Stack Resource Policy(SRP), which is presented and analyzed here, is such a policy. As it turns out, this policy also offers advantages for systems in which runtime stack space is not a major concern, where there may be no stack sharing.

Section 2 defines the elements of our model, including jobs, featherweight processes, and resources. Section 3 defines the SRP, and proves that it works. Section 4 gives a basic schedulability result for earliest-deadline-first (EDF) scheduling with the SRP. Section 5 compares the SRP to the Priority Ceiling Protocol [15], of which it is an evolutionary development. Section 6 briefly discusses the implementation of the SRP and its relation to Ada tasking. Section 7 summarizes the results and mentions some ongoing research.

2 Definitions

Jobs. 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. 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.* The job execution request *arrives* at some time, after which the job execution can begin. Requests that have arrived, but for which the corresponding executions have not yet completed are called *pending.* Pending requests are classified as *waiting*, meaning the job has not yet started, or *active*, meaning the job has started to execute. Names of the forms $\mathcal{J}$,$\mathcal{J}'$, $\mathcal{J}''$,... and $\mathcal{J}_i$ denote both job execution requests and job executions.

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 $\mathcal{J}_{i,1}, \mathcal{J}_{i,2}, \ldots$ 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 need for more than one execution of any job to go 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".

Resources. An execution of a job requires the use of a processor and runtime stack space, and may require certain other serially reusable resources. We assume there is a single processor, which is preemptable, and a finite set of *nonpreemptable resources*, R_1,..,R_m. Allocation of processor time, stack space, and nonpreemptable resources to jobs is governed by *processor and resource allocation policies.* 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 a *mode.* 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 alloca-

tion 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. Each job is required to request and release resources in Last-in-First-Out order, so critical sections of the same job can only overlap if they are properly nested. For bounding priority inversion, we will mainly be interested in the execution times of "outermost" (i.e. non-nested) critical sections.

Binary semaphores and reader-writer locks are examples of nonpreemptable resources. A binary semaphore may only be allocated in only one mode. A reader/writer lock may be allocated in two modes, either *read* or *write*. There may also be multi-unit resources. For each multinuit resource R there is a fixed number of units in the system, N_R, and the mode of a request for a multiunit resource is the number of units being requested, which must be less than or equal to N_R.

Without loss of generality, semaphores and reader/writer locks can be treated as special cases of multiunit resources. For a binary semaphore, $N_R = 1$, and the mode is 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, $read = 1$, and $write = N_R$.

Example. Suppose J_1, J_2, and J_3 are jobs, where the critical sections of the jobs are given schematically in Figure 1. Here, an operation of the form "request (R_j, m)" means the job is requesting m units of resource R_j. A "release" operation releases the most recently acquired resource allocation. (Since we assume any overlapping critical sections must be properly nested, the resource and mode are uniquely determined.) The relationship of the 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 $N_{R_1} = 3$, $N_{R_2} = 1$, and $N_{R_3} = 3$. R_2 behaves as a binary semaphore, R_3 behaves as a reader/writer lock, and R_1's behaves as a more general multiunit resource.

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

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

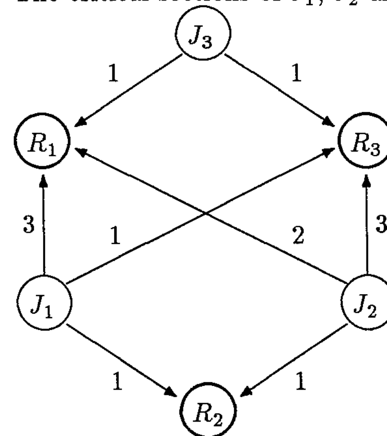

Figure 2: The resource graph for J_1, J_2 and J_3.

Stack Space. Shared runtime stack space is a nonpreemptable resource, but it is treated differently from the other nonpreemptable resources, since the location of the requested space, rather than the quantity, matters. Based on the way in which programming language implementations typically use the runtime stack, we make the following assumptions:

1. Every job using the stack 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.
2. After a job starts execution, if it makes any request that is blocked it must continue to hold its stack resources 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 holding the top of the stack.
4. Only the job at the top of the 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.

Direct 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 *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 a resource allocation.

For a multinunit nonpreemptable resource, a request (J, R, m) is blocked directly iff[1] $\nu_R < m$, where ν_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 further blocked by any outstanding read-allocation of R to another job.

For a shared stack, a job J is directly blocked iff there is another job J' holding the space immediately above the the space occupied by 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. (This does not seriously limit the effectiveness of scheduling, even though it significantly narrows the set of scheduling choices.)

The resource management policy may block some requests that are not directly blocked. In particular, other blocking may be introduced to bound priority inversion. 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.

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 realtime systems include RM (priority $\propto$ 1/period)[2] and EDF (priority $\propto$ 1/deadline).

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 [15], can be applied transitively to expedite any directly blocked job that is not involved in a deadlock.

The rest of this paper 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 $\mathcal{J}_c$ denote the currently executing request and $\mathcal{J}_{top}$ denote the oldest highest priority pending job execution request. Under the priority inheritance policy, either $\mathcal{J}_c = \mathcal{J}_{top}$ or there is a chain of blocked job executions $\mathcal{J}_1, ..., \mathcal{J}_n$ such that $\mathcal{J}_1 = \mathcal{J}_{top}$, $\mathcal{J}_n = J_c$, and $\mathcal{J}_i$ is blocking $\mathcal{J}_{i+1}$ for $i = 1, .., n-1$. (One of the objectives of a resource allocation policy that bounds priority inversion is to insure that there is at most one such chain and the length of this chain never exeeds one.)

Preemption levels. In addition to the priorities which are attached to individual job execution requests, each job J has a *preemption level* $\pi(J)$ (just "level", for short). The level of a job is a positive integer that 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 $\pi(J) < \pi(J')$. Of course, this is also true for priorities. The reason for introducing preemption levels as distinct from priorities ($p(\mathcal{J})$) is to enable us to do static analysis of potential resource conflicts, even for some dynamic priority schemes such as EDF schedul-

[1] Here, "iff" is an abbreviation for "if-and-only-if".

[2] Here, $\propto$ means "is proportional to", in the sense of obeying the same ordering.

ing.

For the specific priority assignments mentioned in this paper, the preemption level of a job will be inversely proportional to 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.

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 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. $p(J) < p(J')$ (so $\mathcal{J}'$ can preempt).

With EDF scheduling, $p(J) < p(J')$ iff $t' + D' < t + D$, so it follows that $D' < D$. It is therefore consistent to define preemption levels so that $\pi(J) < \pi(J')$ iff $D(J) > D(J')$.

An example will emphasize the difference between EDF priority and preemption level. Let $\mathcal{P}$ and $\mathcal{P}'$ be two periodic processes, with relative deadlines 20 and 10 (relative to arrival times), respectively. Preemption level 1 is assigned to jobs of $\mathcal{P}$ and preemption level 2 is assigned to jobs of $\mathcal{P}'$, since the relative deadline of $\mathcal{P}'$ is shorter than the relative deadline of $\mathcal{P}$. $\mathcal{P}'$ can never be preempted by $\mathcal{P}$, but this does not mean that job execution requests of $\mathcal{P}'$ always have higher priority than those of $\mathcal{P}$. Suppose a job-request $\mathcal{J}$ of $\mathcal{P}$ arrives at time t, and a job-request $\mathcal{J}'$ of $\mathcal{P}'$ arrives at time $t + 11$. 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 $\mathcal{J}'$. 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 $p(\mathcal{J}) < p(\mathcal{J}')$. Thus preemption level is different from priority. This is shown in Figure 3.

Relative deadlines can also serve as a basis for preemption levels with RM and *deadline monotone* scheduling[10], 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.

Although relative deadlines are the basis for preemption levels for all these examples, the theoretical results proven in this paper do not depend on preemption levels being the same as relative deadlines. The only property of preemption levels on which these results do depend is the following condition (P1): [3]

$$p(\mathcal{J}) \leq p(\mathcal{J}') \\ \vee\ Arrival(\mathcal{J}) \leq Arrival(\mathcal{J}') \\ \vee\ \pi(J') < \pi(J).$$

In words, this says 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 relative deadlines do satisfy condition (P1) above for all the priority assignments mentioned in this paper, and condition (P1) is sufficient to guarantee that J can preempt J' only if $\pi(J') < \pi(J)$.

3 Stack Resource Policy

3.1 Background

The SRP is based on the concept of preemption ceiling, which is a refinement of the concept of "priority ceiling" defined in [15,13,16,5]. We unify and extend those definitions in the following ways:

1. Priorities are replaced by preemption levels. This allows EDF priorities to be handled without requiring ceilings to be recomputed at run time.
2. Ceilings are defined for multiunit resources, subsuming both binary semaphores and reader/-writer locks.

Abstract ceilings. Each resource R has a *current ceiling*, $\lceil R \rceil$, which is an integer-valued function of the set of outstanding allocations of R. 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 (C2):

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 $\pi(J) \leq \lceil R \rceil$.

One specific definition of ceiling, that satisfies condition (C1), is given below.

Specific ceilings. For a multinit 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:

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

[3] Here $\vee$ denotes "or".

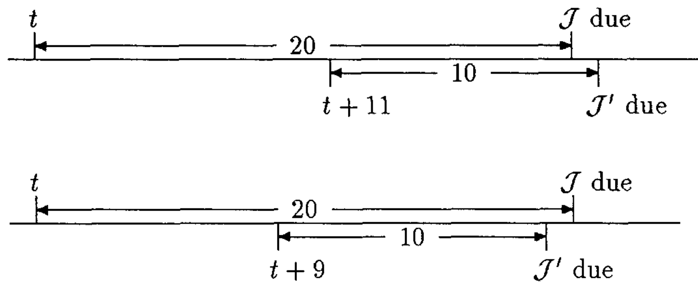

Figure 3: Preemption level *vs.* priority.

where μ_R is the maximum number of units of R that job J may need to hold at any one time. Note that this definition satisfies condition (C1).

Example. The ceilings of the resources for the example shown in Figures 1 and 2 are shown in Figure 4, under the assumption that $\pi(J_i) = p(\mathcal{J}_i) = i$ for $i = 1, 2, 3$.

Ceilings and stack allocations. With a shared stack, the stack space allocated to a job is also a critical resource. Because of the assumptions we have made about stack usage, the stack space held by a job can only block jobs that it might preempt; that is, jobs with lower preemption levels. It follows that condition (C1) imposes no restriction on the current ceiling of a stack, which is therefore defined to be zero.

3.2 Definition of the SRP

Current ceiling. At any instant of time, let the *current ceiling* of the system, $\overline{\pi}$, be the maximum of the preemption level of the current job and the current ceilings of all the resources. That is, if job J_c is currently executing,

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

If there are no jobs currently executing, $\overline{\pi}$ is defined to be zero.

The SRP. The SRP requires that a job execution request $\mathcal{J}$ be blocked from starting execution (i.e. from receiving its initial stack allocation) until

$$\overline{\pi} < \pi(J).$$

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

Note that the SRP does not restrict the order in which resources may be acquired, in contrast to the ordered resource allocation approach of [8]. It is also less restrictive than the other approach of [8], which is based on preallocation. That is, even though the condition $\overline{\pi} < \pi(J)$ is tested before the job J starts to execute, the SRP does not at that time actually allocate all the resources that may 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.

Example. Two possible executions of jobs J_1, J_2, and J_3 under the SRP are shown in Figure 5 and Figure 6. The solid horizontal lines indicate which job is executing, while the barred lines indicate the relative value of the current ceiling, $\overline{\pi}$. Figure 5 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 6 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.

3.3 Blocking properties of the SRP

In [3], we prove that the SRP enforces direct blocking and strictly bounds priority inversion, independent of stack sharing.

Theorem 1 *If no job J is permitted to start until $\overline{\pi} < \pi(J)$, then:*

(a) No job can be blocked after it starts.

(b) There can be no transitive blocking or deadlock.

(c) If the oldest highest-priority job is blocked, it will become unblocked no later than the first instant

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 4: Ceilings of Resources.

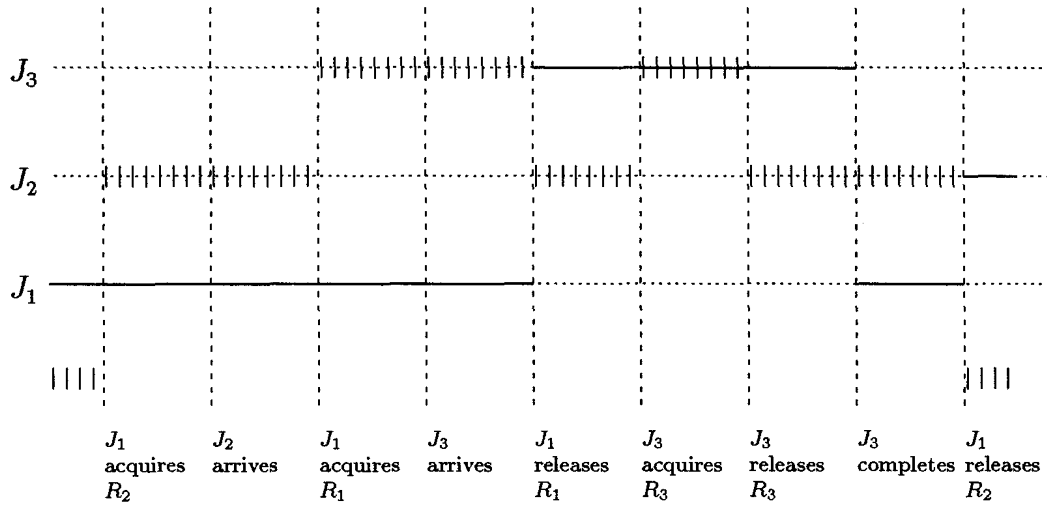

Figure 5: J_3 arrives after J_1 acquires R_1.

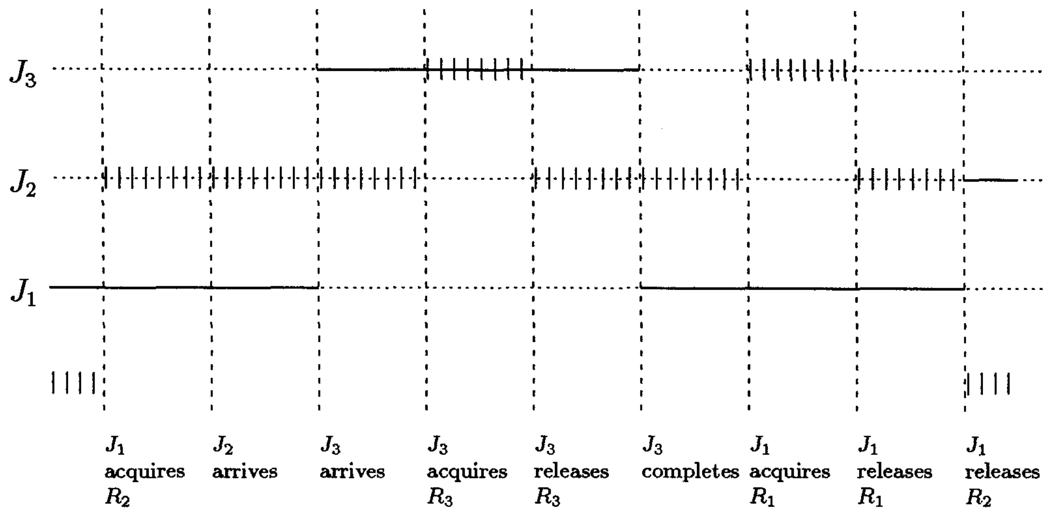

Figure 6: J_3 arrives before J_1 acquires R_1.

that the currently executing job is not holding any nonpreemptable resources.

Note that part (c) means that no job can be subject to priority inversion for longer than the duration of one (outermost) critical section of a lower priority job. Note also that whenever the processor is not idle it is executing either the oldest highest-priority request, $\mathcal{J}_{top}$, or a job that is directly blocking $\mathcal{J}_{top}$. The currently executing job, $\mathcal{J}_c$, always occupies the top position on the shared runtime stack, if all jobs share one runtime stack. (If the SRP is applied to a system in which runtime stacks are not shared, it is still helpful to think of all the jobs that have started executing, but have not finished, as occupying positions on an imaginary stack, which is ordered by preemption level.)

4 Schedulability

Theorem 1 is sufficient to apply the schedulability results for RM scheduling of [11,15]. What is more important, it can be used to derive a schedulability test for the EDF policy, stronger than that obtained by [5]. To do this, we restrict our process model so that it more closely resembles that of [11,5]. In particular, let there be a one-to-one correspondence between processes and jobs. Suppose there are n (periodic or aperiodic) processes, $\{\mathcal{P}_1, ..., \mathcal{P}_n\}$, ordered by increasing relative deadlines of the corresponding jobs, $\{J_1, ..., J_n\}$. Let the relative deadlines all be positive. 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 execution time of the longest critical section of any job J_k such that $D_i \leq D_k$ and $i \neq k$, or zero if there is no such J_k. Assume there is a system start time, before which no jobs are requested. The theorem below is proven in [3].

Theorem 2 *A set of n (periodic and aperiodic) jobs is schedulable by EDF scheduling if*

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

5 Relation to PCP.

The SRP is a refinement of the PCP, just as the PCP is a refinement of previous ordered-resource allocation techniques such as Havender's [8]. The improvements introduced by the SRP are:

1. Ceilings are defined in terms of preemption levels, instead of priorities, so that the SRP applies directly to EDF scheduling (without dynamic recomputation of ceilings).
2. Ceilings are defined for multiunit resources.
3. Stack sharing is supported.
4. The blocking test is only applied when a job tries to start execution; whereas in the PCP this condition is applied each time a job requests a new resource allocation.
5. Resource requests never block, and hence cannot require extra context switches; consequently, there are at most two context-switches per preemption.
6. Because there is no blocking after a job starts executin, a stronger EDF schedulability result can be obtained than with dynamic priority ceilings.
7. Different jobs of a process may have different priorities.

An important technical difference between the two techniques is that the PCP only guarantees that once a job is blocked it will not be blocked again. In contrast, the SRP also guarantees that if a job is blocked, it is only blocked before starting. This simplifies schedulability analysis, is the key to the proof of Theorem 2.

The only way in which the SRP is not a consistent extension of the PCP is the earlier blocking. Since the SRP only blocks a job before it starts, it must make worst-case assumptions about the job's resource requirements. Therefore the SRP will block a job in some situations when the PCP would not. In particular, the PCP may come out ahead on the average when a job contains conditional code, so that sometimes it requests a resource and other times it doesn't require any[4]. This difference may be significant for applications where average-case performance is more important than worst-case. We shall prove below that when the job actually requests all the resources, the PCP is no better in reducing priority inversion than the SRP.

5.1 Priority Inversion.

Theorem 3 *The maximum priority-inversion time of any job under the SRP is no longer than 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

[4] With the PCP, whether a request blocks does not depend on which resource is being requested.

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 1 we know that $\mathcal{J}$ can only be subject to priority inversion from the current job, $\mathcal{J}_c$. Thus, J_c is holding a semaphore S that blocks $\mathcal{J}$ from starting. That is, $\pi(J) < \lceil S \rceil$. Since we are assuming preemption level equals priority, $p(\mathcal{J}) < \lceil S \rceil$.

The same order of events may happen with the PCP. That is, a higher-priority job execution request for J may arrive while J_c is holding S. Under the PCP, J would preempt. Since we are assuming the worst case, suppose J later requests some resource. We have $p(\mathcal{J}) < \lceil S \rceil \leq \lceil S^* \rceil$, where S^* is the ceiling of the semaphore that has the highest ceiling among all semaphores locked by jobs other than J, or is zero if there are no such locked semaphores. 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_c releases S, which is at least as long as under the SRP. □

5.2 Context Switches

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 saves context switches.

Theorem 4 *The SRP requires at most two context switches per 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 from the job that is preempted to the job that is requested, and back when the preempting job completes. □

Theorem 5 *The PCP, like any other policy that waits to block a job until it makes a resource request, may require four context switches per job execution request, for any job that shares a semaphore with a lower priority job.*

Proof. Let J be any job such that there is a lower-priority job J_L and a semaphore S such that both J and J_L lock S. If J is requested while J_L is running and has locked S, 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 S; (3) from J_L to J, when J_L unlocks S; (4) from J back to J_L, when J completes. □

Note that two is the least possible upper bound on the number of context switches per request, if preemption is allowed. Together, Theorem 4 and Theorem 5 say that the upper bound on the number of context switches with the SRP is half of the worst case for the PCP. This improvement is due to earlier blocking.

6 Implementation Considerations

The SRP can be implemented very simply and efficiently. The implementation is similar to that of the PCP[15,4], but the locking operations are simpler because they cannot involve blocking. The ceilings $\lceil R \rceil_n$ are static, and so may be precomputed and stored in a table. A stack may be used to keep track of the current ceiling, $\overline{\pi}$. When a resource R is allocated its current state, ν_R, is updated, and $\overline{\pi}$ is set to $\lceil R \rceil_{\nu_R}$ iff $\overline{\pi} < \lceil R \rceil_{\nu_R}$. The old values of ν_R and $\overline{\pi}$ are pushed on the stack. When resource R is released, the values of $\overline{\pi}$ 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 job at the top of the queue, $\mathcal{J}_{top}$, to see if it is different from the current job, $\mathcal{J}_c$, and satisfies the preemption criterion, $\overline{\pi} < \pi(J_{top})$. If $\mathcal{J}_c$ passes this test, the identity of $\mathcal{J}_c$ is pushed on the stack, runtime stack space is allocated to $\mathcal{J}_{top}$ and $\mathcal{J}_{top}$ starts execution. If $\mathcal{J}_{top}$ fails the test, the dispatcher simply returns. Whenever a job completes, $\mathcal{J}_c$ is restored from the stack, and the dispatcher is invoked. The dispatcher is also called when a request arrives, if it becomes the new $\mathcal{J}_{top}$.

7 Conclusions, and Further Research

By transforming processes into a classical job scheduling model, they may be allowed to share runtime stack space. However, such stack sharing requires a unified approach to processor and resource allocation in order to prevent deadlock and bound priority inversion tightly. The SRP is such a resource allocation policy.

The SRP is a good policy even when there is no

stack sharing. It is an improvement over previously published versions of the PCP in several respects. One of these is that it is directly compatible with EDF scheduling.

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 may be 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 [11] 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. It also appears that this model is compatible with the sporadic server concept of [9,17].

We have also developed a refinement of the SRP, called the Minimal SRP (MSRP)[3]. This is a slightly more complex policy, which is the least restrictive policy for rate monotone (RM) and earliest-deadline-first (EDF) scheduling that can strictly bound priority inversion and prevent deadlock, given that jobs share a single runtime stack.

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. More specifically, it appears that Theorem 3 can be generalized to show the efficacy of early blocking for arbitrary policies. The distinction between preemption level and priority may also have broader applications. It appears Theorem 2 can be extended to processes with multiple jobs. Extensions to multiprocessor applications, similar to [14], also seem to be possible.

Acknowledgement

This paper is a development of ideas first proposed in [2], based on discussions with Russ Wilson, Carl Malec, and Greg Scallon of Boeing Aerospace and Electronics.

References

[1] *Military Standard Ada Programming Language*, ANSI/MILSTD1815A, U.S. Department of Defense, Ada Joint Program Office (January 1983).

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

[3] T.P. Baker, "Stack-Based Scheduling of Realtime Processes", technical report, Computer Science Department, Florida State University, Tallahassee, FL (April 1990).

[4] M.W. Borger, and R. Rajkumar, "Implementing Priority Inheritance Algorithms in an Ada Runtime System", technical report, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA (February, 1989).

[5] M.I. Chen and K.J. Lin, "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 (April 1989).

[6] E.G. Coffman, Jr. and P.J. Denning, "Operating Systems Theory", Prentice-Hall (1973).

[7] E.W. Giering III and T.P. Baker, "Toward the Deterministic Scheduling of Ada Tasks", *Proceedings of the IEEE Real-Time Systems Symposium*, (December 1989) 31-40.

[8] J.W. Havender, "Avoiding deadlock in multitasking systems", *IBM Systems Journal* 7,2 (1968) 74-84.

[9] J.P. Lehoczky, L. Sha, J.K. Strosnider, "Aperiodic Scheduling in a Hard-Real-Time Environment", Technical Report, Carnegie-Mellon University (1987).

[10] J.Y.-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks", *Performance Evaluation 2* (1982) 237-250.

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

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

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

[14] R. Rajkumar, L. Sha, and J.P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors", *Proceedings of the Real-Time Systems Symposium*, IEEE (1988) 259-272.

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

[16] L. Sha, R. Rajkumar, J. Lehozcky, "A Priority Driven Approach to Real-Time Concurrency Control", technical report, Carnegie Mellon University (July 1988).

[17] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard-Real-Time Systems", *Real Time Systems* 1,1 (June 1989) 27-60.