Supporting Nested Locking in Multiprocessor Real-Time Systems[*]

Bryan C. Ward and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

This paper presents the first real-time multiprocessor locking protocol that supports fine-grained nested resource requests. This locking protocol relies on a novel technique for ordering the satisfaction of resource requests to ensure a bounded duration of priority inversions for nested requests. This technique can be applied on partitioned, clustered, and globally scheduled systems in which waiting is realized by either spinning or suspending. Furthermore, this technique can be used to construct fine-grained nested locking protocols that are efficient under spin-based, suspension-oblivious or suspension-aware analysis of priority inversions. Locking protocols built upon this technique perform no worse than coarse-grained locking mechanisms, while allowing for increased parallelism in the average case (and, depending upon the task set, better worst-case performance).

1 Introduction

To support real-time applications on multiprocessor platforms, real-time scheduling and synchronization algorithms are required that enable timing constraints to be met. While prior scheduling-related work has produced many viable scheduling options, serious limitations remain pertaining to synchronization. Perhaps most significantly, all current state-of-the-art real-time multiprocessor locking protocols (see [2, 12] for relevant citations) directly support only *non-nested* shared resource requests; *nested* requests, which are commonly employed in practice, are only indirectly supported through the use of coarse-grained locking techniques such as *group locks*. A group lock treats a set of shared resources as a single resource, and arbitrates access to the group using a single-resource locking protocol [1].

Unfortunately, group locks are inflexible because resources must be statically grouped before execution. They also result in pessimistic analysis because a job waiting on one resource in a group must block while a job holds another resource in the group. This pessimism limits the degree of concurrency, which is particularly concerning in today's increasingly parallel system architectures.

Alternatively, resource groups can be broken into smaller elements that are acquired individually. This is called *fine-grained* locking, and it is useful in a variety of settings. For example, in a shared linked list, each element can be controlled with a single lock, instead of locking the whole list. This allows different jobs to access separate sections of the list concurrently. Nested locking protocols can also be used in applications such as real-time database systems.

To enable fine-grained locking in multiprocessor real-time systems, we discuss a family of efficient locking protocols for partitioned, clustered, and globally scheduled job-level static priority (JLSP) task systems, i.e., systems in which a job's priority is constant (e.g., as in earliest-deadline-first (EDF) or static-priority scheduling). Tasks can either wait by spinning (busy-waiting) or by suspending. We analyze locking protocols on the basis of *priority inversion blocking* (*pi-blocking*), i.e., the duration of time a job is blocked while a lower-priority job is running. In schedulability analysis, the execution time of a task must be inflated by the duration of pi-blocking to ensure timing constraints are met. Under analysis assumptions used previously [4, 2, 5], most of the locking protocols we develop are asymptotically optimal.

Prior work. The uniprocessor *priority ceiling protocol* (*PCP*) has been extended to multiple processors in a number of locking protocols, for example, the *multiprocessor PCP* (*PCP*) [12], *distributed PCP* (*DPCP*) [12], *multiprocessor dynamic PCP* (*MDPCP*) [6] and the *parallel PCP* (*PPCP*) [8]. Of these, only the MDPCP claims to support nested resource requests, albeit in a somewhat limited fashion—the MDPCP is limited to periodic tasks scheduled by partitioned EDF, has somewhat high worst-case blocking bounds, and enables higher concurrency than group locking (with respect to nested resource accesses) only in certain corner cases. Recently, several other protocols have been developed that are asymptotically optimal for multiprocessor JLSP systems. These protocols and their blocking bounds, which are summarized in Table 1, are discussed next.

In work on optimal suspension-oriented multiprocessor locks, Brandenburg and Anderson [4] presented two definitions of pi-blocking for suspension-oriented multiprocessor locking protocols: *suspension-oblivious* (*s-oblivious*), in which suspensions are modeled as computation, and *suspension-aware* (*s-aware*), in which suspensions are accounted for. They also established a per-request lower bound on pi-blocking of $\Omega(n)$ in the s-aware case, and $\Omega(m)$ in the s-oblivious case, where n is the number of tasks and m is the number of processors [4].

Block et al. [1] developed the *flexible multiprocessor locking protocol* (*FMLP*), in which waiting can be realized as either spinning or suspending. Under the FMLP, a spinning or resource-holding job runs non-preemptively. Later, Brandenburg extended the FMLP to obtain the *FIFO mutex locking protocol* (FMLP$^+$), which is more flexible because it lifts this requirement [2]. Under s-aware analysis, for a large category

[*]Work supported by NSF grants CNS 1016954 and CNS 1115284; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

Analysis	Scheduler	Locking Protocol	Every Job Pi-blocking	Per-Request Pi-blocking
spin	Any	FMLP/FMLP$^+$	mL^{max}	$(m-1)L^{max}$
s-aware	Partitioned	FMLP$^+$	nL^{max}	$(n-1)L^{max}$
	Clustered	FMLP$^+$	$O(\phi \cdot n)^\dagger$	$(n-1)L^{max}$
	Global‡	FMLP	$O(n)$	$(n-1)L^{max}$
s-oblivious	Partitioned	P-OMLP	mL^{max}	$(m-1)L^{max}$
	Clustered	C-OMLP	mL^{max}	$(m-1)L^{max}$
	Global	OMLP	0	$(2m-1)L^{max}$

† ϕ is the ratio of the maximum to minimum period as discussed in Sec. 5.4. The issue of optimality is still open for the s-aware clustered case.

‡ Applicable only under certain schedulers as discussed in Sec. 5.4.

Table 1: Summary of existing single-resource locking protocols and their blocking complexity. The column "Every Job Pi-Blocking" indicates how long any job in the system (whether it accesses shared resources or not) can be pi-blocked when it is not currently utilizing the locking protocol to access a shared resource. The column "Per-Request Pi-Blocking" indicates how long a job can be pi-blocked per request. L^{max} denotes the maximum critical section length. All listed protocols are asymptotically optimal, except as noted.

of global schedulers and all partitioned schedulers, the FMLP and/or the FMLP$^+$ are asymptotically optimal;[1] their spin-based counterparts are also optimal. Brandenburg and Anderson also developed the $O(m)$ *locking protocol* (*OMLP*) family of suspension-based protocols, which are asymptotically optimal under s-oblivious analysis [4, 5]. The OMLP can be used on partitioned, clustered, and globally scheduled systems. None of these previous multiprocessor locking protocols support fine-grained lock nesting.

Contributions. In this paper, we present the first multiprocessor real-time locking protocol that supports fine-grained nested resource requests. This locking protocol, called the *real-time nested locking protocol* (*RNLP*), employs a k-exclusion lock,[2] which can be implemented in different ways, giving rise to a family of mutual exclusion (mutex) locking protocols. We show that the RNLP can be used on global, clustered, and partitioned JLSP systems and when waiting is realized through spinning or suspensions. We conduct s-aware, s-oblivious, and spin-based analysis of pi-blocking under the RNLP.

Unlike group locks, the RNLP does not require resources to be statically grouped before execution. All that is required is a partial order on resource acquisitions, which is a common assumption in practice to ensure that deadlock is impossible.

Organization. In Sec. 2, we formally describe our system model and assumptions. In Sec. 3 we describe the basic architecture of the RNLP, which is composed of two components. In Secs. 4 and 5, we describe and analyze each of these components. We then give provide a brief discussion of the benefits of fine-grained locking in Sec. 6 and conclude in Sec. 7.

2 Background and Definitions

We consider a system of n sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ that execute on m processors. Each task T_i is composed of a sequence of jobs; we let $J_{i,j}$ denote the j^{th} job of the i^{th} task. We omit the job index j if it is insignificant. Each task is characterized by a *worst-case execution time* e_i, a *minimum job separation* p_i, and a *relative deadline* d_i. For simplicity, we assume for each T_i that $p_i = d_i$ and that each job of T_i must complete before its deadline (no tardiness); these assumptions have no bearing on any optimality claims made in this paper. A job is said to be *released*, when it is made available for execution, and it is *pending* until it finishes its execution.

Resources. We assume a similar resource model to that presented by Brandenburg and Anderson [4]. The system contains q *shared resources* $\mathcal{L} = \{\ell_1, \ldots, \ell_q\}$ such as shared data objects or I/O devices. In this paper, we consider only mutex locks (though we employ a k-exclusion lock to realize the locks that we construct), and as such, at most one job can hold each resource ℓ_a at any time. Access to shared resources is controlled by a *locking protocol*. When a job J_i requires a resource ℓ_a, it makes a request for ℓ_a to the locking protocol. J_i's request is said to be *satisfied* when J_i acquires ℓ_a, and *completes* when J_i *releases* ℓ_a. A job that has issued a resource request that has not yet been satisfied is said to have an *outstanding* resource request. A job that has issued a resource request that is not complete is said to have an *incomplete resource request*. We let $wait(J_i, t)$ denote the resource for which J_i is waiting at time t if any. The segment of a job between one of its requests being satisfied and completed is called a *critical section*. A job can either *spin* (i.e., busy-wait) or suspend while waiting for one of its requests to be satisfied. A *ready* job is one that can be scheduled; thus, a job that is suspended and waiting for a shared resource is not ready.

If J_i holds no resources when it makes a request, then the request is said to be an *outermost request*. We denote J_i's k^{th} outermost request as $\mathcal{R}_{i,k}$ and the corresponding resource $\mathcal{F}_{i,k}$. Once J_i acquires a resource, it may make a *nested request* for another resource. If J_i acquires a resource at time t via an outermost request, and t' is the earliest subsequent time when J_i holds no resources, then $(t, t']$ is called an *outermost critical section*. Note that resource requests do not have to be properly nested, as seen in Fig. 1 (here, ℓ_a is acquired first, but ℓ_b is released last). The maximum number of outermost requests J_i makes is given by N_i. The maximum duration of J_i's k^{th} outermost critical section is $L_{i,k}$. We say that J_i

[1]S-aware optimality in clustered systems is still an open issue, as discussed in Sec. 5.

[2]k-exclusion generalizes mutual exclusion by allowing up to k simultaneous lock holders.

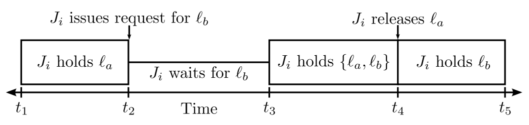


Figure 1: Illustration of a job J_i's outermost critical section. At time t_1, J_i acquires resource ℓ_a. At time t_2, J_i issues a nested resource request for ℓ_b, and is blocked during the interval $[t_2, t_3)$. At time t_4, J_i releases ℓ_a. J_i's outermost critical section spans from t_1 to t_5 when J_i no longer holds any shared resources.

makes *progress* if a job that holds a resource for which J_i is waiting is scheduled and executing its critical section.

The RNLP requires a strict (irreflexive) partial order, $\prec$, on the set of resources such that a job holding resource ℓ_b cannot make a resource request for ℓ_a if $\ell_a \prec \ell_b$. In addition to preventing deadlock, this ordering is used by the RNLP to improve pi-blocking bounds. There can be a few exceptions to this requirement, as described in Appendix A, that allow for increased concurrency, however none of these exceptions are required to derive worst-case results.

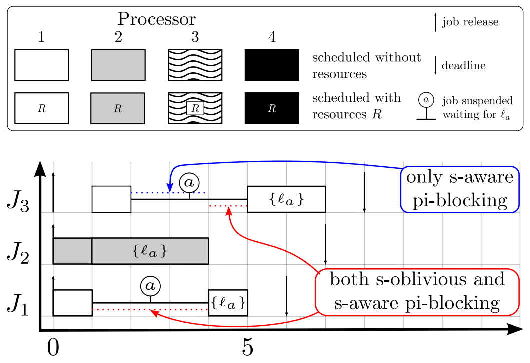
Scheduling. We consider partitioned, clustered, and globally scheduled systems. Under clustered scheduling, the m processors are partitioned into $\frac{m}{c}$ non-overlapping sets of c processors.[3] Each task is statically assigned to a cluster, and may migrate freely among the processors in the cluster. Partitioned and global scheduling are special cases of clustered scheduling with $c = 1$ and $c = m$, respectively.

Within each cluster, jobs are scheduled from a single ready queue using a JLSP scheduling algorithm. In such an algorithm, each job has a *base priority*. The locking protocols we develop allow a job's *base priority* to be elevated such that it has a higher *effective priority*. We denote a job J_i's effective priority at time t as $p(J_i, t)$. Within each cluster, the (at most) c jobs with the highest effective priority are scheduled at any point in time. Three techniques can be used (separately) to elevate a job's effective priority: *priority boosting*, *priority inheritance*, and *priority donation*. A job that is priority boosted has its priority unconditionally raised to ensure it is scheduled. Under priority inheritance, a job executes with an effective priority equal to that of a suspended job. Under priority donation, a higher-priority job J_i that, upon release, should preempt some lower-priority job J_k with an incomplete resource request instead suspends and donates its priority to J_k to ensure that J_k makes progress [5]. Upon the completion of J_k's outermost critical section, J_i ceases to be a priority donor, and will never be a priority donor again.

Blocking. We adopt the following definition of s-oblivious and s-aware pi-blocking and spin-blocking defined by Brandenburg and Anderson [4, 5, 2]; the first two definitions only apply to suspension-based locks while the third only applies to spin-based locks.

Definition 1. Under **s-aware** schedulability analysis, a job J_i incurs *s-aware pi-blocking* at time t if J_i is pending but not scheduled and fewer than c higher-priority jobs are **ready** in T_i's cluster.

[3]Non-uniform cluster sizes could be integrated into our analysis at the expense of more verbose notation.



Processor
1
2
3
4
R
R
R
R
scheduled without resources
scheduled with resources R
job release
deadline
job suspended waiting for ℓ_a
J_3
J_2
J_1
$\{\ell_a\}$
$\{\ell_a\}$
$\{\ell_a\}$
only s-aware pi-blocking
both s-oblivious and s-aware pi-blocking
0
5

Figure 2: Illustration adapted from [4] of the difference between s-oblivious and s-aware analysis. In this example, three EDF-scheduled jobs share a single resource ℓ_a on two processors. During $[2, 4)$, J_3 is blocked, but there are m jobs with higher priority, thus J_3 is not s-oblivious pi-blocked. However, because J_1 is also suspended, J_3 is s-aware pi-blocked. Intuitively, under s-oblivious analysis, the suspension time of higher-priority jobs is modeled as computation, but under s-aware analysis, it is not. Note the legend applies to all subsequent figures.

Definition 2. Under **s-oblivious** schedulability analysis, a job J_i incurs *s-oblivious pi-blocking* at time t if J_i is pending but not scheduled and fewer than c higher-priority jobs are **pending** in T_i's cluster.

Definition 3. A job J_i incurs *spin-based blocking* at time t if J_i is spinning (and thus scheduled) waiting for a resource.

The difference between s-oblivious and s-aware pi-blocking is demonstrated in Fig. 2.

Note that if a job spins non-preemptively then it may cause pi-blocking for jobs that otherwise would have been scheduled. The duration of pi-blocking caused by non-preemptivity must be analyzed in this case as well. Similarly, priority donation and priority boosting may cause jobs that are not currently utilizing a locking protocol to be pi-blocked, and this blocking must be analyzed as well (as we do later).

Similar to [4, 5], we measure the blocking behavior of the RNLP using the maximum duration of pi-blocking. However, when supporting nested resource requests, a job can be pi-blocked while holding a resource, as seen in Fig. 1. This "inner" pi-blocking must be included in the analysis of the total duration of pi-blocking. Furthermore, existing analysis of locking protocols is conducted in terms of the maximum critical section length. In our analysis, we instead consider the maximum execution time of a critical section, since the maximum critical section length can depend on the duration of "inner" pi-blocking caused by the locking protocol. The maximum critical section length and the maximum execution time are the same when the RNLP is compared with single-resource locking protocols.

Assumptions. Tight blocking bounds are a function of a number of variables such as the frequency of resource requests, duration of each critical section, and the number of nested requests made. In our asymptotic analysis, we consider the number of outermost requests per job N_i, as well as the maximum critical section length $L^{max} = \max L_{i,k}$ to be constant. We consider both n and m to be variables,

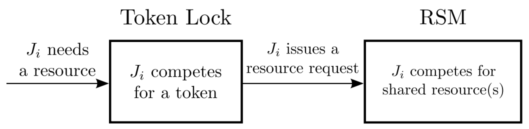

Token Lock
RSM
J_i needs a resource
J_i competes for a token
J_i issues a resource request
J_i competes for shared resource(s)

Figure 3: Components of the RNLP.

and assume $n \geq m$. All other parameters are considered to be constant. Note that we do not impose any restrictions on the number of tasks sharing each resource, the ratio of largest to smallest task period or relative deadline, or the maximum depth of nested requests.

3 The RNLP

The RNLP is composed of two components, a k-exclusion *token lock*, and a *request satisfaction mechanism* (*RSM*). The token lock restricts the number of jobs that can have an incomplete resource request while the RSM determines when requests are satisfied. In order for a job to issue a resource request, it must first acquire a token through the token lock. The k token-holding jobs can then compete for shared resources according to the rules of the RSM. Depending upon the system (partitioned, clustered, or global), how waiting is realized (suspension or spinning), and the type of analysis being conducted (s-oblivious, s-aware, or spin-based), different token locks, number of tokens (most often k is n or m), and RSMs can be paired to yield an asymptotically optimal locking protocol supporting nested requests. This architecture is shown in Fig. 3.

We specify the RSM via a set of rules. Without loss of generality, these rules are presented assuming a uniform cluster size of c. We assume a few basic properties of the token lock defined as follows (specific token locks are considered in Sec. 5).

T1 There are at most k token-holding jobs at any time, of which there are no more than c from each cluster.

T2 If a job is pi-blocked waiting for a token, then it makes progress. This can be accomplished by elevating the priority of a token-holding job through priority boosting, inheritance, or donation.

Once a job acquires a token, it is allowed to compete for a shared resource under the rules of the RSM. There are several rules and key ideas common to all RSMs. For each shared resource ℓ_a, there is a resource queue RQ_a of length at most k. The timestamp of token acquisition is stored for each job J_i, and denoted $ts(J_i)$.[4] Each resource queue is priority ordered by increasing timestamp. In the absence of any nested resource requests, this ordering is the same as FIFO ordering. This queue ordering allows a job performing a nested resource request to effectively "cut in line" to where it would have been had it requested the nested resource at the time of its outermost resource request. We denote the job at the head of RQ_a as $hd(a)$. The rules below (illustrated below in Example 1) are common to all RSMs.

[4] $ts(J_i)$ is really a function of time because it is updated for every outermost critical section. However, because we analyze the RNLP on a per-request basis, we omit the time parameter for notional simplicity.

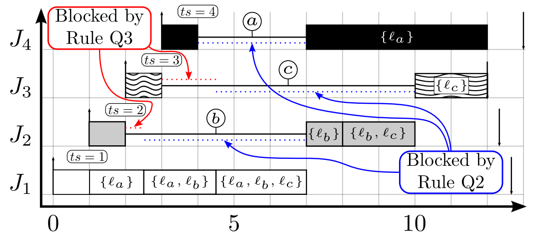


Figure 4: Illustration of Example 1 where $m = 4$ and $q = 3$.

Q1 When J_i acquires a token at time t, its timestamp is recorded: $ts(J_i) := t$. We assume a total order on such timestamps.

Q2 All jobs in RQ_a are waiting with the possible exception of $hd(a)$.

Q3 A job J_i acquires resource ℓ_b when it is the head of the RQ_b, i.e., $J_i = hd(b)$, and there is no resource ℓ_a such that $\ell_a \prec \ell_b$ and $ts(hd(a)) < ts(J_i)$.[5]

Q4 When a job J_i issues a request for resource ℓ_a it is enqueued in RQ_a in increasing timestamp order.[6]

Q5 When a job releases resource ℓ_a it is dequeued from RQ_a and the new head of RQ_a can gain access to ℓ_a, subject to Rule Q3.

Q6 When J_i completes its outermost critical section, it releases its token.

These rules do not specify how waiting is realized. A specific RSM may employ either spinning or suspending.

Example 1. To illustrate these rules, we present an example, which is depicted in Fig. 4. Consider a global-EDF (G-EDF) scheduled system with three shared resources, ℓ_a, ℓ_b, and ℓ_c (so $q = 3$), and $m = 4$, and a total order on resources by index (i.e., $\ell_a \prec \ell_b \prec \ell_c$). As shown in Fig. 4, each job J_i acquires a token at time $t = i$, and thus by Rule Q1, $ts(J_i) = i$. Furthermore, $N_i = 1$, and $\mathcal{F}_{1,1} = \ell_a$, $\mathcal{F}_{2,1} = \ell_b$, $\mathcal{F}_{3,1} = \ell_c$, and $\mathcal{F}_{4,1} = \ell_a$. At times $t = 2.5$ and $t = 4.5$, J_1 issues nested requests for ℓ_b and ℓ_c, respectively. These requests are satisfied immediately, because J_1 has an earlier timestamp than any job in either RQ_b and RQ_c. Note that at time $t = 3$, J_3 has the earliest timestamp of the jobs in RQ_c. However, by Rule Q3, J_3 must wait until J_1 completes its outermost critical section before it can acquire ℓ_c. Thus, when J_1 requests ℓ_c at time $t = 4.5$, J_3 has not acquired ℓ_c and hence J_1's request is satisfied immediately. At time $t = 7$, J_1 finishes its outermost critical section, and J_4 acquires ℓ_a. Because J_2 has an earlier timestamp than J_4, J_2 can also acquire ℓ_b at time $t = 7$. However, J_3 must wait until $t = 10$ for J_2 to finish its outermost critical section before its request for ℓ_c is satisfied. Note that during the interval $[7, 10)$, both J_2 and J_4 hold shared resources to which a group lock would have required serial access.

To analyze the behavior of an RSM, we first develop terminology and notation to describe when and how jobs can be

[5] As shown in Appendix A, if more information is known about the task set, this rule can be relaxed to allow for more concurrency.

[6] We assume the acquisition of a token and subsequent enqueueing into some RQ_a occur atomically.

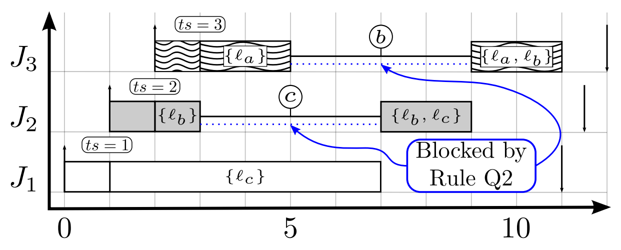
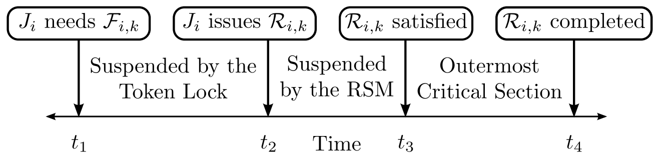


Figure 5: Illustration of Example 2 where $m = 4$ and $q = 3$.



J_i needs $\mathcal{F}_{i,k}$
J_i issues $\mathcal{R}_{i,k}$
$\mathcal{R}_{i,k}$ satisfied
$\mathcal{R}_{i,k}$ completed
Suspended by the Token Lock
Suspended by the RSM
Outermost Critical Section
t_1
t_2
Time
t_3
t_4

Figure 6: Phases of a resource request in the RNLP.

blocked. Job J_i in some resource queue RQ_a is said to be *directly blocked* by every job before it in RQ_a. In our previous example, at time $t = 4$, J_4 is directly blocked by J_1 while J_1 holds resource ℓ_a. The set of jobs that J_i is directly blocked by is denoted

$$DB(J_i, t) = \{J_k \in \mathrm{RQ}_{wait(J_i,t)} \mid ts(J_k) < ts(J_i)\}.$$

Note that J_i can be directly blocked by at most one resource-holding job J_h. This is because only one job can hold $wait(J_i, t)$ at time t. It is possible that J_h itself is directly blocked by another resource holding job. In this case, all jobs that are blocking J_h also block J_i. We call this *transitive blocking*. Transitive blocking is the transitive closure of direct blocking. The set of jobs that transitively block J_i at time t is given by

$$TB(J_i, t) = \bigcup_{J_k \in DB(J_i,t)} DB(J_k, t).$$

Note that $DB(J_i, t) \subseteq TB(J_i, t)$.

Example 2. To illustrate transitive blocking we consider the schedule shown in Fig. 5, which pertains to the same task system as in Example 1. At time $t = 1$, job J_1 acquires ℓ_c, at time $t = 2$, J_2 acquires ℓ_b, and at time $t = 3$, J_3 acquires ℓ_a. Also, at time $t = 3$, J_2 issues a nested resource request for ℓ_c, and at time $t = 5$, J_3 issues a nested request for ℓ_b. At time $t = 5$, J_3 is directly blocked by J_2, and J_2 is directly blocked by J_1. Thus, J_3 is transitively blocked by both J_1 and J_2.

Reconsidering Example 1, at time $t \in [3, 4.5)$ in Fig. 4, J_3 waits by Rule Q3 even though it is at the head of its resource queue. This gives rise to a different form of blocking that we must also quantify in our analysis. We say that a job that is blocked by a job with an earlier timestamp in another queue is *indirectly blocked*. The set of jobs that J_i is indirectly blocked by at time t is given by

$$IB(J_i, t) = \{J_k \in \mathrm{RQ}_a \mid \ell_a \prec wait(J_i, t) \wedge ts(J_k) < ts(J_i)\}.$$

We use the general term *blocked* to refer to either transitive or indirect blocking. We denote the set of jobs that block J_i as

$$B(J_i, t) = TB(J_i, t) \cup IB(J_i, t).$$

From the definition of $B(J_i, t)$, we have the following.

Lemma 1. *For any job J_i and any time t, $\forall J_k \in B(J_i, t)$, $ts(J_k) < ts(J_i)$.*

To ensure a bounded duration of pi-blocking, every RSM must satisfy the following property.

P1 If J_i is pi-blocked (s-oblivious, s-aware, or spin-based) by the RSM, then J_i makes progress.

Properties P1 and T2 combine to ensure that a job that is pi-blocked waiting for a token makes progress towards acquiring the shared resource it needs. Property P1 will be proved in Sec. 4 for a number of individual RSMs.

Analysis. We now prove a bound on the maximum duration of pi-blocking experienced by a token-holding job J_i. In the following analysis, let t_1 denote the time that J_i makes a request for a token and t_2 be the time that J_i receives a token. Also, let t_3 be the time that J_i's outermost request is satisfied and t_4 be the time that its outermost critical section completes. These times are depicted in Fig. 6.

A job's worst-case duration of pi-blocking is equal to the sum of the maximum duration of pi-blocking caused by the token lock during $[t_1, t_2)$ and by the RSM during $[t_2, t_3)$ before the J_i's outermost request is satisfied, as well as during $[t_3, t_4)$ if J_i issues a nested request. We now consider the pi-blocking caused by the RSM. Later, in Sec. 5 we consider worst-case pi-blocking under various token locks.

Theorem 1. *The maximum duration of pi-blocking (regardless of whether waiting is realized by spinning or suspending, or in the latter case if analysis is s-oblivious or s-aware), during $[t_2, t_4)$ for any RSM is $(k-1)L^{max}$.*

Proof. Property P1 ensures that if a job is pi-blocked, it makes progress. By Lemma 1, a job can never be pi-blocked by a job with a later timestamp. By Property T1, there are at most $k-1$ jobs with earlier timestamps. Thus, a job can be pi-blocked in any RSM for at most $k - 1$ outermost critical sections, each of length at most L^{max}. □

4 Specific Request Satisfaction Mechanisms

In this section, we describe four request satisfaction mechanisms, the spin RSM (S-RSM), boost RSM (B-RSM), inheritance RSM (I-RSM), and donation RSM (D-RSM).

4.1 S-RSM

The S-RSM is the RSM used when waiting is realized by spinning instead of suspending. Spinning is advantageous when critical section lengths are short in comparison to the overhead of a context switch [1, 2]. To construct an RSM in which waiting is realized by spinning, we add an additional rule to those common to all RSMs.

S1 All token-holding jobs execute non-preemptively. A job that is waiting in a resource queue spins.

This rule can be used on partitioned, clustered, or globally scheduled systems. However, there can be no more than c spinning jobs per cluster, and thus there can be at most $k = m$ tokens, c from each cluster. Additionally, non-preemptivity can cause jobs that are not currently utilizing the locking protocol to be pi-blocked. Thus, the execution time of every job must be inflated to account for this possibility.

Lemma 2. *The S-RSM for partitioned, clustered, and globally scheduled systems in which waiting is realized by spinning ensures Property P1.*

Proof. By Rule S1, every token-holding job is scheduled (and is spinning if it is waiting). Thus, every resource-holding job is scheduled, which ensures that progress is made for all token-holding jobs. □

4.2 B-RSM

The B-RSM can be applied in partitioned, clustered, or globally scheduled systems in which waiting is realized by suspending instead of spinning. Under the B-RSM, progress is ensured by boosting the priority of a resource-holding job, similar to the FMLP$^+$[2]. Priority boosting, like non-preemptive spinning, can cause jobs that are not utilizing the locking protocol to be pi-blocked. The following rule defines the B-RSM.

B1 The (at most) m jobs with the earliest timestamps among the resource-holding jobs without outstanding resource requests (i.e., that are not waiting) are boosted above the priority of all non-resource-requesting jobs.

This rule allows for any value of k, as it ensures that no more than m jobs can be priority boosted concurrently. The following lemma follows immediately.

Lemma 3. *The B-RSM for partitioned, clustered, or globally scheduled systems in which waiting is realized by suspending ensures Property P1.*

Note that the B-RSM ensures progress under any JLSP scheduler, but does not always lead to an asymptotically optimal locking protocol due to the pi-blocking boosting can cause on jobs that are not currently utilizing the locking protocol, as described by Brandenburg [2].

4.3 I-RSM

The I-RSM is only applicable on globally scheduled systems because it requires that the priorities of all resource-requesting jobs can be compared. It also requires waiting to be realized by suspending instead of spinning. The I-RSM uses priority inheritance instead of priority boosting as a progress mechanism, which is advantageous because it does not induce pi-blocking on non-resource-requesting jobs.

To motivate the design of the I-RSM, consider again Example 1. Suppose at time $t = 6$ there exist m jobs (not shown) that do not utilize the locking protocol that have deadlines just after $t = 12$ but before J_1's deadline. Then J_3 is the only token-holding job that has a sufficient priority to be scheduled. However, J_3 is blocked by J_1, which holds ℓ_c, and J_1 does not have sufficient priority to be scheduled, and thus is also suspended. J_3 therefore does not make progress and it can thus have an unbounded duration of pi-blocking. Priority inheritance can be applied to limit such pi-blocking.

We call the job with the earliest timestamp that blocks J_i the *inheritance candidate* of J_i. The inheritance candidate is thus given by

$$ic(J_i, t) = \underset{J_k \in B(J_i,t)}{\arg\min}\ ts(J_k).$$

A job J_c may be the inheritance candidate of several jobs. We define the *inheritance candidate set* (ICS) of J_c to be the set of jobs for which J_c is the inheritance candidate.

$$ICS(J_c, t) = \{J_i \mid \exists T_i \in \tau, ic(J_i, t) = J_c\}.$$

In Example 1, at time $t = 6$, J_2, J_3, and J_4 are all blocked by J_1. J_1 is therefore the inheritance candidate of J_2, J_3, and J_4. J_1 is also the earliest job by timestamp that blocks each of J_2, J_3 and J_4. Thus, $ic(J_2, 6) = J_1$ and $ICS(J_1, 6) = \{J_2, J_3, J_4\}$.

The I-RSM builds upon these ideas. If a job is pi-blocked, then the resource-holding job that blocks it, its inheritance candidate, should be scheduled.

I1 A ready job J_i holding resource ℓ_k inherits the highest priority of the jobs for which it is an inheritance candidate:

$$p(J_i, t) = \max_{J_k \in \{J_i\} \cup ICS(J_i,t)} p(J_k, t).$$

In Example 1, at time $t = 6$, $ICS(J_1, 6) = \{J_2, J_3, J_4\}$. Of these jobs J_3 has the highest priority, and thus J_1 inherits the priority of J_3.

Lemma 4. *A job J_i's priority can be inherited by at most one job at a time.*

Proof. By construction, J_i has at most one inheritance candidate at any time t, and thus there is only one job J_c for which $J_i \in ICS(J_c, t)$. Thus, J_i's priority will be inherited by J_c or by no job at all. □

Lemma 4 ensures that there are no two jobs executing with the same identity, which is effectively equivalent to a task having two threads. This would break the sporadic task model, and thus the resulting system would not be analyzable using existing schedulability tests.

Lemma 5. *For any job J_i, $ic(J_i, t)$ is ready.*

Proof. By contradiction. Assume that $J_c = ic(J_i, t)$. If J_c is not ready, then it is suspended by either Rule Q2 or Q3. In either case, by Lemma 1, J_c is blocked by a job J_b with an earlier timestamp. Thus, J_i is also blocked by J_b. This contradicts the fact that J_c is J_i's inheritance candidate. □

Lemma 6. *The I-RSM for globally scheduled systems in which waiting is realized by suspending ensures Property P1.*

Proof. If a job J_i is pi-blocked (s-oblivious or s-aware) at time t, then J_i has sufficient priority to be scheduled under either definition of pi-blocking. By Lemma 5, $ic(J_i, t)$ is ready. By Rule I1, $ic(J_i, t)$ has priority $p(ic(J_i, t), t) \geq p(J_i, t)$, and thus $ic(J_i, t)$ is scheduled. □

The I-RSM does not place any restrictions on the number of tokens in the system, i.e., the value of k. Depending upon the scheduler, analysis type, and token lock, k can be chosen to allow for increased parallelism or decreased worst-case pi-blocking. This issue is considered in Sec. 5.

4.4 D-RSM

The D-RSM is designed for clustered (and hence global and partitioned) systems in which waiting is realized by suspending. In these systems, the I-RSM is not sufficient to ensure progress because priorities cannot be compared across clusters. A job in one cluster therefore cannot inherit the priority of a job in another cluster. In clustered systems, progress can be ensured through priority donation, which prevents problematic preemptions of resource-requesting jobs [5]. For instance, in Example 1, if an additional job J_5 were released at time $t = 7$ with a deadline of $t = 11$, then it would donate its priority to J_4, the lowest priority job, which has an incomplete resource request. There are no new rules for the D-RSM, however the D-RSM does require an additional constraint on the token lock.

C1 A token-holding job has one of the highest c effective priorities in its cluster.

Because there are at most m jobs that have one of the highest c effective priorities in their cluster, there can be at most m token holding jobs and thus $k \leq m$.

Lemma 7. *Property C1 implies Properties P1 on partitioned, clustered, and globally scheduled systems in which waiting is realized by suspending.*

Proof. Property C1 ensures that a token holding job has a sufficient effective priority to be scheduled. Thus, a ready resource-holding job (which necessarily holds a token) is scheduled, and progress is ensured. □

The D-RSM itself does not cause pi-blocking for non-resource-requesting jobs. However, as we shall see, a token lock that satisfies Property C1 can cause non-resource-requesting jobs to be pi-blocked.

5 Token Locks

In this section, we describe how existing k-exclusion locking protocols can be used as token locks. For each token lock, we describe the best choice of k, how to pair the token lock with an RSM, and the analytical worst-case pi-blocking complexity of the complete resulting locking protocol. The results of this section are summarized in Table 2.

5.1 Spin k-exclusion

When waiting is realized by non-preemptive spinning as in the S-RSM, the best choice of token lock is essentially no token lock at all, because the S-RSM alone upholds the properties of both an RSM as well as a token lock. We call this token lock the *trivial token lock* (*TTL*) because a job acquires a token immediately upon request. Because there can be at most m jobs running non-preemptively on m processors, $k = m$ under the TTL. For the remainder of this subsection, we assume $k = m$.

Lemma 8. *Properties T1 and T2 are ensured by Rule S1.*

Proof. By Rule S1, once a job issues a resource request, it runs non-preemptively until it finishes its outermost critical section. No more than m jobs can therefore have incomplete resource requests at a time. This ensures Property T1. Property T2 is ensured because all jobs with incomplete resource requests are scheduled, and thus make progress. □

The non-preemptive nature of spin-locks, just like priority boosting and priority donation, can cause a job to be pi-blocked even when it has no incomplete resource request.

Theorem 2. *Any job in the system can be pi-blocked by a job spinning non-preemptively for a duration of at most mL^{max}.*

Proof. In the worst case, there can be m jobs that are not currently utilizing the locking protocol that have sufficient priority to be scheduled but are not, due to m other jobs spinning non-preemptively. All m of the token holding jobs must complete their outermost critical sections before the m^{th} blocked job can be scheduled. □

Theorem 3. *The maximum duration of s-blocking per-request in the S-RSM is $(m-1)L^{max}$.*

Proof. Follows from Theorem 1 and $k = m$. □

5.2 CK-OMLP for Clustered Systems

The most versatile of existing k-exclusion locking protocols is the clustered k-exclusion OMLP (CK-OMLP) developed by Brandenburg and Anderson [5]. The CK-OMLP can be employed on partitioned, clustered, and globally scheduled systems in which waiting is realized by suspending, and it has asymptotically optimal s-oblivious pi-blocking behavior on all such systems. The CK-OMLP relies upon priority donation to ensure progress, and thus every job with an incomplete resource request has one of the c highest priorities in its cluster. In the remainder of this section, we assume that $k = m$, which ensures that a job is not pi-blocked waiting for a token (while priority donors pi-block, the donation mechanism ensures that the [up to] m token holders have the highest effective priorities in the system).

Lemma 9. *The CK-OMLP ensures Properties T1, T2, and C1.*

Proof. The CK-OMLP is a k-exclusion locking protocol, and thus satisfies Property T1. Lemma 1 of [5] proves that a token-holding job (i.e., a job that is "within its critical section" from the CK-OMLP's perspective) has sufficient priority to be scheduled, which yields Property C1. By Lemma 7, a token holding job makes progress. Thus, a job waiting for a token makes progress, satisfying Property T2. □

Under priority donation any job can be forced to donate its priority on job release for a period of time. If a job issues many resource requests, the amortized cost per request is reduced. However, because any job can be pi-blocked while it is a priority donor, every job must inflate its execution cost. Because priorities cannot be compared across clusters, we believe this donation cost for all jobs is fundamental on clustered systems.

Theorem 4. *The maximum duration of s-oblivious pi-*

Analysis	Scheduler	Token Lock	k	RSM	Every Job Pi-blocking	Per-Request Pi-blocking
spin	Any	TTL	m	S-RSM	mL^{max}	$(m-1)L^{max}$
s-aware	Partitioned	TTL	n	B-RSM	nL^{max}	$(n-1)L^{max}$
	Clustered	TTL	n	B-RSM	$O(\phi \cdot n)$	$(n-1)L^{max}$
	Global†	TTL	n	I-RSM	$O(n)$	$(n-1)L^{max}$
s-oblivious	Partitioned	CK-OMLP	m	D-RSM	mL^{max}	$(m-1)L^{max}$
	Clustered	CK-OMLP	m	D-RSM	mL^{max}	$(m-1)L^{max}$
	Global	CK-OMLP	m	D-RSM	mL^{max}	$(m-1)L^{max}$
		O-KGLP	m	I-RSM	0	$(5m-1)L^{max}$
		I-KGLP	m	I-RSM	0	$(2m-1)L^{max}$

† Applicable only under certain schedulers as discussed in Sec. 5.4.

Table 2: Configuration and worst-case pi-blocking of the RNLP on various platforms. The columns "Token Lock" and "RSM" describe an instantiation of the RNLP that pairs a token lock with an RSM. The remaining columns show the blocking complexity of the resulting locking protocol under the given assumptions just as in Table 1. All listed protocols are asymptotically optimal except the case of clustered schedulers under s-aware analysis for which no asymptotically optimal locking protocol is known.

blocking per job (regardless of whether the job ever issues a resource request) caused by the donation mechanism of the CK-OMLP is mL^{max}.

Proof. By Lemma 1 of [5], a token-holding job has sufficient priority to be scheduled. By Lemma 2 of [5] and the assumption that $k = m$, the maximum duration of s-oblivious pi-blocking caused by priority donation is bounded by the maximum amount of time a job (the donee) can hold a token. A job can hold a token for L^{max} time while it holds shared resource(s), plus $(k-1)L^{max}$ time by Theorem 1. Since we assume $k = m$, the maximum duration of pi-blocking caused by priority donation is thus mL^{max}. □

Theorem 5. *The maximum duration of s-oblivious pi-blocking per outermost resource request is $(m-1)L^{max}$ under the D-RSM and CK-OMLP.*

Proof. Follows from Theorem 1 and $k = m$. □

Theorems 4 and 5 show that the RNLP with the D-RSM and CK-OMLP has the same s-oblivious pi-blocking bound as a mutex lock in the clustered OMLP, as seen in Table 1. However, the RNLP supports nested locking while the OMLP does not. Also note that while the D-RSM and the CK-OMLP produce the same blocking bounds as the TTL and the S-RSM, the former produces a suspension-based lock while the latter produces a spin-based lock. A system designer may choose one over the other depending upon critical section lengths and system overheads.

5.3 O-KGLP for Globally Scheduled Systems

The CK-OMLP can be used on any system with non-overlapping clusters. However, any job can be forced to suspend on release to donate its priority by Theorem 4. An application with a mix of non-resource-requesting jobs with small periods and jobs with long periods and long critical sections may therefore be unschedulable as a short-period job may be forced to donate its priority for longer than its period.

Elliott and Anderson developed a k-exclusion locking protocol specifically for globally scheduled systems called the O-KGLP [9]. In the O-KGLP, non-resource-requesting jobs are not affected by the behavior of the locking protocol. The O-KGLP has $O(m/k)$ worst-case s-oblivious pi-blocking, which is asymptotically optimal.

Lemma 10. *The O-KGLP ensures Properties T1 and T2.*

Proof. The O-KGLP is a k-exclusion lock, and thus satisfies Property T1. By Theorem 1 of [9], the O-KGLP is asymptotically optimal, and progress is ensured because of the bounded duration of pi-block in each component of the O-KGLP by Lemmas 2-5 and 7 of [9]. Thus Property T2 holds under the O-KGLP. □

Theorem 6. *When the O-KGLP is used in tandem with the I-RSM, the maximum s-oblivious pi-blocking per resource request is given by $(2m+3k-1)L^{max}$.*

Proof. From Lemmas 2-5 and 7 of [9], the maximum pi-blocking per token request is $2m/k+2$ times the maximum duration of time a job can hold a token (i.e., the "maximum critical section length" from the perspective of the O-KGLP). By Lemma 1 (of this paper), a token-holding job can be blocked by $k-1$ requests, and thus the maximum duration of time a job can hold a token is kL^{max}. Thus, the total time a job can be pi-blocked waiting for a token per request is $(2m+2k)L^{max}$. By Theorem 1, a job can be pi-blocked by the RSM for $(k-1)L^{max}$ time per request. Thus, the total pi-blocking per request is $(2m+3k-1)L^{max}$. □

The CK-OMLP has a smaller constant factor for the maximum per-request pi-blocking, but the O-KGLP does not require a non-resource-requesting job to inflate its execution time.[7]

5.4 Trivial Token Lock for S-Aware Analysis

The token locks discussed above lead to asymptotically optimal implementations under spin-based or s-oblivious analysis. However, these locking protocols do not perform well under s-aware analysis. Under s-aware analysis, it is best to choose $k = n$ to allow for maximal concurrency, thus the TTL is used. Note that increased concurrency results in fewer suspensions, which are accounted for under s-aware analysis. From the TTL, we have the following lemma.

[7]In Appendix B, we present a new k-exclusion locking protocol (the I-KGLP in Table 2) for globally scheduled systems, that has a better s-oblivious pi-blocking bound under s-oblivious analysis than the O-KGLP.

Lemma 11. *The TTL satisfies Properties T1 and T2.*

Depending upon the scheduler, the TTL can be paired with different RSMs. We pair the TTL with the B-RSM under partitioned scheduling.

Theorem 7. *On a partitioned system, the maximum duration of s-aware pi-blocking per outermost resource request is* $(n-1)L^{max}$ *under the B-RSM and TTL.*

Proof. Follows from Theorem 1 and $k = n$. □

Under s-aware analysis we must carefully consider the pi-blocking boosting itself may cause.

Theorem 8. *Let* $c = 1$ *and* n_c *be the number of tasks assigned to* J_i*'s partition. The worst-case s-aware pi-blocking of job* J_i *caused by the boosting of other jobs in the B-RSM* (*regardless of whether* J_i *ever issues a resource request*) *is* $(n_c - 1)L^{max}$.

Proof. In the worst case, all requests are serialized by the B-RSM, as any concurrency would decrease s-aware pi-blocking. In this case, the blocking behavior is the same as the FMLP$^+$ because the B-RSM employs the same progress mechanism as the FMLP$^+$. Thus, the bound follows directly from the bound for the FMLP$^+$ in Theorem 6.4 of [2]. □

Note that the per-job and per-request pi-blocking are both $O(n)$ and thus the pairing of the TTL and the B-RSM is asymptotically optimal for partitioned systems (given the previously established $\Omega(n)$ lower bound [4]).

The B-RSM can be employed in systems in which $c > 1$ (including $c = m$) and progress is ensured. However, the B-RSM can cause a job that is not currently utilizing the locking protocol to be pi-blocked for longer than $O(n)$. Brandenburg showed a lower bound of $\Omega(\phi)$ on s-aware pi-blocking caused by priority boosting where ϕ is the ratio of the maximum period to the minimum period in the system [2]. This bound is not proven to be tight. In Appendix C, we establish a coarse upper bound on s-aware pi-blocking under the B-RSM where $c > 1$ to be $O(\phi \cdot n)$. This bound is based on the number of other jobs that can execute during one job's period.

Priority donation and priority inheritance are the only remaining progress mechanisms to consider. Priority donation is not particularly effective under s-aware analysis [2]. Priority inheritance is only applicable on globally scheduled systems and in general it has the same $\Omega(\phi)$ s-aware pi-blocking bound as priority boosting [2]. However, under rate monotonic (RM) scheduling, as well as *constrained, fixed priority-point schedulers* (e.g., FIFO, and G-EDF with relative deadlines at most periods), a special class of JLSP schedulers in which each task's *relative priority point* does not exceed its period, priority inheritance yields an asymptotically optimal locking protocol.

Theorem 9. *The worst-case s-aware pi-blocking per job* (*regardless of whether the job ever issues a resource request*) *caused by the TTL and I-RSM under either RM or any constrained, fixed priority-point global scheduler is* $O(n)$.

Proof. In the worst case, all requests are serialized by the I-RSM as any concurrency would decrease s-aware pi-blocking. In this case, the I-RSM is equivalent to the FMLP. From [2], the maximum s-aware pi-blocking caused by priority inheritance is $O(n)$ for RM and constrained, fixed priority-point global schedulers. □

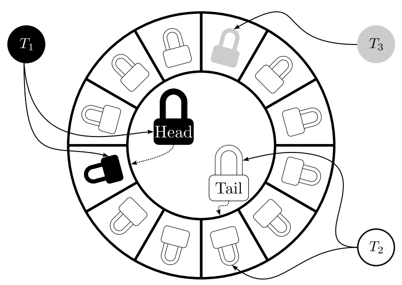


Figure 7: Illustration of a circular buffer for imprecise computation. The head (tail) resource as well as the element at the head (tail) of the queue are locked by one task, while another task has an element in the middle of the queue locked.

Theorem 10. *The maximum duration of s-aware pi-blocking per outermost resource request is* $(n-1)L^{max}$ *under the TTL and I-RSM.*

Proof. Follows from Theorem 1 and $k = n$. □

Note that asymptotically, the RNLP performs no worse than any existing locking protocols under s-aware analysis. However, the increased concurrency afforded by the RSM leads to improved s-aware pi-blocking in practice.

6 Benefits of Fine-Grained Locking

The worst-case blocking behavior of the RNLP family of locking protocols is no worse than the FMLP or the OMLP family of locking protocols under the analysis assumptions we have made. However, the RNLP can allow multiple jobs to access resources within a group concurrently. In many applications, nested resource requests, which force resources to be grouped, though possible, are relatively infrequent [3]. In such cases, the RNLP often allows jobs to hold individual resources in a resource group concurrently. This improves average-case pi-blocking. Additionally, if more information is known *a priori* about a task set, then it may be possible to achieve tighter blocking bounds.

As an example, consider an unmanned aerial vehicle that employs planning algorithms to compute a route through enemy territory to avoid danger while carrying out its mission. Some such algorithms employ *imprecise computations* [10], in which results improve given more computation time [11]. In such applications, imprecise computation may be used to compute navigation plans for segments of the vehicle's flight path. Later, any spare computational resources can be used to optimize the plans for dangerous segments of the flight path.

To support such an application, we can envision navigation plans being computed by producer task(s), and enqueued into a circular buffer as seen in Fig. 7. When the time comes for a plan to be executed, a consumer task dequeues the plan from the circular buffer. Other tasks can further optimize plans that are already in the circular buffer.

To support this application under group locks, access to any part of the circular buffer would necessarily be arbitrated by a single group lock. Using fine-grained locking with the

RNLP, jobs can concurrently modify different parts of the buffer. In this locking scheme, the head and tail pointers are each considered individual shared resources to ensure jobs cannot either produce or consume concurrently. Each element in the buffer is also controlled by a mutex lock to ensure that a plan is not modified concurrently. In this example, to enqueue (dequeue) a plan to (from) the queue, a job must acquire the head (tail) pointer as well as the lock for the element to which the head or tail pointer points.

Under the RNLP, if the buffer is neither empty nor full, then a producer task can never be blocked by a consumer task or vice versa. This application-specific knowledge can be incorporated into finer-grained analysis to achieve tighter blocking bounds. In this case, the blocking bound for the RNLP would be less than that of a similar group lock.

Even in applications in which *a priori* fine-grained locking properties are not known, it can still be advantageous to use the RNLP over a single-resource locking protocol. For example, in a soft real-time system, the increased parallelism afforded by the RNLP may enable more jobs to meet their deadlines. In systems supporting a mix of hard and soft real-time tasks as well as best-effort work, the increased parallelism may allow more best effort work to get done. Even in a hard real-time setting, the RNLP has a benefit; by reducing the duration of pi-blocking, safety margins are made wider to help ensure anomalous behavior does not cause a job to miss its deadline.

7 Conclusions

Existing locking protocols for multiprocessor real-time systems only support a single resource. Nested resource requests are therefore only supported through the use of coarse-grained locking techniques such as group locks. In this paper, we have presented the RNLP, a modular locking protocol composed of a k-exclusion token lock and an RSM. Token locks and RSMs are paired depending upon the scheduler, type of analysis, and how waiting is realized to achieve asymptotically optimal locking protocols in most cases (and all cases in which asymptotically optimal single-resource locking protocols are known). Furthermore, the modular nature of the RNLP allows for future progress mechanisms or k-exclusion locks to be incorporated into the RNLP to improve performance in particular cases.

The variants of the RNLP we have presented have s-oblivious, s-aware, and spin-based blocking behavior no worse than existing multiprocessor locking protocols under the analysis assumptions we have employed. The RNLP, however, supports nested resource requests, and it is possible for multiple jobs to concurrently hold separate resources in a resource group. This increased parallelism can be advantageous in many settings. The RNLP is also advantageous in that groups do not have to be statically assigned before execution. Resources can be dynamically added or removed so long as the relative order of all other resources is not modified. This property adds flexibility to the RNLP.

This work forms a strong platform on which we intend to continue to build. For example, while this paper only addresses mutex locks, we would like to extend the RNLP to support k-exclusion and reader-writer resources, and allow nested requests between all of these types of resources. Additionally, we intend to explore the possibility of *dynamic group locks*, in which a job can request an arbitrary set of shared resources at once (under the RNLP, a job would have to request each resource in the set individually).

Under the RNLP, all resources are controlled by a single token lock. However, this can cause a job to be blocked waiting for a token held by a job waiting for another resource, possibly one in which critical section lengths are longer. We intend to investigate the possibility of allowing multiple instances of the RNLP to be instantiated to control fine-grained nested requests within individual resource groups, to eliminate the possibility of such an occurrence. Such a scheme would be a hybrid of coarse-grained group locking and fine-grained locking.

Also, we want to conduct a more rigorous analysis of the blocking behavior of the RNLP using additional information about a system, such as the semantics of the operations on shared data structures, maximum depth of nested resource requests, the length of each task's critical sections, and the request order. Finally, we plan to implement the RNLP in LITMUS$^{\mathrm{RT}}$ and conduct an empirical evaluation.

Acknowledgements. We thank Glenn Elliott for his helpful discussions pertaining to k-exclusion locking protocols.

References

[1] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07*, pages 47–56, Aug. 2007.

[2] B.B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[3] B.B. Brandenburg and J.H. Anderson. Feather-trace: A light-weight event tracing toolkit. In *OSPERT '07*, pages 61–70, 2007.

[4] B.B. Brandenburg and J.H. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS '10*, pages 49–60, 2010.

[5] B.B. Brandenburg and J.H. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *EMSOFT '11*, pages 69–78, Sep. 2011.

[6] C.-M Chen and S.K. Tripathi. Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, Univ. of Maryland, College Park, 1994.

[7] U.M.C. Devi and J.H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *RTSS '05*, pages 133–189, Dec. 2005.

[8] A Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *RTSS '09*, pages 377–386, 2009.

[9] G.A. Elliott and J.H. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. In *RTNS '11*, pages 15–24, Sep. 2011.

[10] J.W.S. Liu, K.-J. Lin, W.-K. Shih, A.C.-S. Yu, J.-Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *Computer*, 24(5):58–68, May 1991.

[11] M.S. Mollison, J.P. Erickson, J.H. Anderson, S.K. Baruah, and J.A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *CIT '10*, pages 1864–1871, Jul. 2010.

[12] R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Kluwer Academic Publishers, Boston, 1991.

A Nesting-Aware Modification

The RNLP employs the resource order to ensure that a job J_i cannot be blocked by another job J_k with a later timestamp. This is accomplished through Rule Q3. However, Rule Q3 is pessimistic in that it assumes that J_i can request any resource later in the order. If the subset of resources that J_i can possibly request is known *a priori*, it can be incorporated into the RNLP to allow for increased parallelism.

Let $\mathcal{L}_{i,k} \subseteq \mathcal{L}$ be the subset of resources that J_i can request in its k^{th} outermost critical section. Note that $\mathcal{L}_{i,k}$ may be a superset of the set of resources that J_i actually request in its k^{th} outermost critical section, but it may not be a subset. For example, Rule Q3 assumes that a job J_i holding resource ℓ_a can request any resource higher in the order, and thus $\mathcal{L}_{i,k} = \{\ell_b \in \mathcal{L} \mid \ell_a \prec \ell_b\}$.

If a job J_i is at the head of RQ_a, and there is no job with an earlier timestamp that could potentially request ℓ_a then J_i can safely acquire ℓ_a. Thus Rule Q3 can be slightly modified as follows.

M1 A job J_i with an incomplete request $\mathcal{R}_{i,k}$ acquires ℓ_a when it is the head of RQ_a, i.e., $J_i = hd(a)$, and there is no job J_x with $ts(J_x) < ts(J_i)$ and an incomplete request $\mathcal{R}_{x,y}$ such that $\ell_a \in \mathcal{L}_{x,y}$.

We next adapt the definition of indirect blocking to reflect Rule M1. To incorporate information on a per-request basis, notationally, we consider each resource queue to contain requests $\mathcal{R}_{i,k}$ instead of simply jobs.

$$IB(\mathcal{R}_{i,k}, t) = \{\mathcal{R}_{x,y} \in \text{RQ}_a \mid \ell_a \in \mathcal{L} \wedge wait(J_i, t) \in \mathcal{L}_{x,y} \wedge ts(J_x) < ts(J_i)\}$$

Then the set of jobs that $\mathcal{R}_{i,k}$ can be blocked by, $B(\mathcal{R}_{i,k}, t)$, is the transitive closure of both direct and indirect blocking.

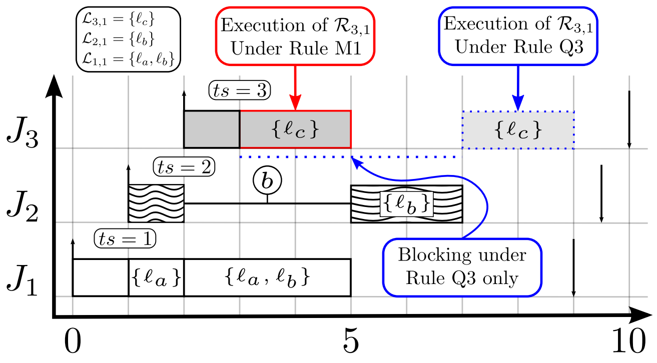
$$B(\mathcal{R}_{i,k}, t) = \bigcup_{\mathcal{R}_{x,y} \in DIB(\mathcal{R}_{i,k}, t)} DIB(\mathcal{R}_{x,y}, t)$$

where $DIB(\mathcal{R}_{i,k}, t) = DB(\mathcal{R}_{i,k}, t) \cup IB(\mathcal{R}_{i,k}, t)$.

By construction, the timestamp of every request that blocks $\mathcal{R}_{i,k}$ has a timestamp later than $ts(J_i)$. Thus Lemma 1 and Theorem 1 also hold when Rule M1 is used instead of Rule Q3. Next we give an example to demonstrate how Rule M1 allows for additional parallelism in some cases.

Example 3. To illustrate the increased parallelism that Rule M1 allows, consider the schedule shown in Fig. 8, which pertains to a similar task system as in Example 2. Now let $\mathcal{L}_{1,1} = \{\ell_a, \ell_b\}$, $\mathcal{L}_{2,1} = \{\ell_b\}$, and $\mathcal{L}_{3,1} = \{\ell_c\}$. At time $t = 1$, J_1 acquires ℓ_a and at time $t = 2$ J_1 acquires ℓ_b. At time $t = 2$, J_2 issues a request for ℓ_b and is enqueued behind J_1 in RQ_b. At time $t = 3$, J_3 requests ℓ_c. Because neither J_1 nor J_2 will ever request ℓ_c (i.e., $\ell_c \notin \mathcal{L}_{1,1} \cup \mathcal{L}_{2,1}$), by Rule M1, J_3 acquires ℓ_c at time $t = 3$. However, under Rule Q3, since $ts(J_1) < ts(J_3)$ and $\ell_a \prec \ell_c$, then J_3 would be blocked indirectly by J_1.

In many applications, nested resource requests are infrequent, and when resource requests are nested, the degree of nesting is often not very deep [3]. If this information is available to the RNLP, the degree of concurrency can often be improved. If the majority of jobs do not make nested requests, then indirect blocking is uncommon and jobs only wait for other jobs that have requested the same resource.



$\mathcal{L}_{3,1} = \{\ell_c\}$
$\mathcal{L}_{2,1} = \{\ell_b\}$
$\mathcal{L}_{1,1} = \{\ell_a, \ell_b\}$
Execution of $\mathcal{R}_{3,1}$ Under Rule M1
Execution of $\mathcal{R}_{3,1}$ Under Rule Q3
$ts = 3$
$\{\ell_c\}$
$\{\ell_c\}$
J_3
$ts = 2$
b
$\{\ell_b\}$
J_2
$ts = 1$
$\{\ell_a\}$
$\{\ell_a, \ell_b\}$
Blocking under Rule Q3 only
J_1
0
5
10

Figure 8: Illustration of Example 3 where $m = 4$ and $q = 3$.

Order relaxation. To prevent deadlock, application developers are often required to enforce a partial order on lock acquisition. Specifically, a job holding ℓ_b cannot request ℓ_a. This is because a job holding ℓ_a could then request ℓ_b and cause deadlock. However, the partial order can be relaxed for Rule Q3 and further relaxed for Rule M1 to lift this requirement in some cases. For example, under Rule Q3, a job holding ℓ_a and ℓ_c can request ℓ_b without causing deadlock.

Lemma 12. *Under an RSM employing Rule Q3, a job J_i holding a resource ℓ_a can request any resource ℓ_b such that $\ell_a \prec \ell_b$ without causing deadlock.*

Proof. Assume for contradiction that a job J_i holding resource ℓ_a makes a nested request at time t for resource ℓ_b that causes deadlock. Then J_i is blocked by the job holding ℓ_b, J_k. By Lemma 1, $ts(J_k) < ts(J_i)$. Then since J_i has a timestamp later than J_k, J_k cannot be blocked either directly or transitively by J_i by Lemma 1, and thus there cannot be deadlock. □

Thus under the RSM, a job holding ℓ_a can request any resource ℓ_b such that $\ell_a \prec \ell_b$, even if it holds a another resource ℓ_c such that $\ell_b \prec \ell_c$.

Under Rule M1, an RSM does not need to be aware of the resource order because each outermost request must specify which resources it may request in the future. For every job J_i, Rule M1 in effect "saves a spot" in every RQ_a associated with a resource J_i may request. Consequently, no resource order is required under an RSM implementing Rule M1.

Lemma 13. *Under an RSM implementing Rule M1, within J_i's k^{th} outermost critical section, J_i can request any resource $\ell_b \in \mathcal{L}_{i,k}$ without causing deadlock.*

Proof. Assume for contradiction that a job J_i holding resource ℓ_a makes a nested request at time t for a resource ℓ_b that causes deadlock. Then J_i is blocked by the job holding ℓ_b, J_k. By Lemma 1, $ts(J_k) < ts(J_i)$. By the assumption that there is deadlock, $J_i \in B(J_k, t)$. Thus J_k (or a job that J_k is transitively blocked by) has an outstanding request for ℓ_a. However, by Rule M1, J_i has an earlier timestamp than any job that could possibly request ℓ_a. Thus, $ts(J_i) < ts(J_k)$

which is a contradiction. □

Rule M1 incorporates knowledge of what resources a job may request in the future to prevent deadlock while allowing jobs to request resources in an arbitrary order.

Under Rule Q3, if requests are not properly nested, a job J_i may release its lowest order resource ℓ_a within its k^{th} outermost critical section. However, by Lemma 12, J_i may not request ℓ_a within the same outermost critical section. This allows another job J_k to acquire ℓ_a. Under Rule M1, since $\ell_a \in \mathcal{L}_{i,k}$, J_i can request ℓ_a again. However, if J_i does not require ℓ_a again within its k^{th} outermost request, Rule M1 does not allow another job to acquire ℓ_a until J_i completes its k^{th} outermost critical section.

To allow further concurrency under Rule M1, a job may inform the RNLP that it will not make a nested request for a given resource. Thus the set of resources a job J_i may request in its k^{th} outermost critical section is a function of time, $\mathcal{L}_{i,k}(t)$. To ensure proper blocking bounds, a job cannot add resources to $\mathcal{L}_{i,k}(t)$. Thus we require that for all $t, t' \in [t_3, t_4]$ (recall the definitions of t_3 and t_4 from Fig. 6) with $t \leq t'$ that $\mathcal{L}_{i,k}(t') \subseteq \mathcal{L}_{i,k}(t)$. Thus if J_i releases ℓ_a at time t', it can update $\mathcal{L}_{i,k}(t')$ such that $\ell_a \notin \mathcal{L}_{i,k}(t')$ and another job can acquire ℓ_a.

In some applications, a job J_i may make requests for different resources depending upon a conditional within its k^{th} outermost critical section. For example, a job could acquire a resource ℓ_a, check a condition, and either request ℓ_b or ℓ_c (in this case $\mathcal{L}_{i,k}(t_3) = \{\ell_a, \ell_b, \ell_c\}$). After the conditional is checked, J_i could update $\mathcal{L}_{i,k}$ to reflect the code path it took. For example, if J_i acquires ℓ_b instead of ℓ_c at time t', then $\mathcal{L}_{i,k}(t') = \{\ell_a, \ell_b\}$. This may allow another job to acquire ℓ_c during J_i's k^{th} outermost critical section.

B Improved $O(m/k)$ k-exclusion Global Locking Protocol

In this Appendix, we develop the *improved k-exclusion global locking protocol* (*I-KGLP*), a k-exclusion locking protocol that is optimal under s-oblivious analysis on globally scheduled systems in which waiting is realized by suspending. The I-KGLP also has a slightly improved pi-blocking bound over the O-KGLP. As is described in Sec. 5, the O-KGLP has a worst-case pi-blocking bound of $2\lceil \frac{m}{k} \rceil + 2$ times the maximum critical section length. While this is asymptotically optimal, it can be slightly improved. When a k-exclusion lock is employed as a token lock, the maximum duration of a critical section as seen by the k-exclusion lock is kL^{max}. When k is large, a minor improvement in worst-case pi-blocking from the perspective of the k-exclusion lock can lead to a signification improvement in pi-blocking from the perspective of the RNLP. For this reason, we developed the I-KGLP that has a worst-case pi-blocking bound of $(2\lceil \frac{m}{k} \rceil - 1)$ times the maximum critical section length.

B.1 Resource-Request Priority Donation

The I-KGLP achieves an improved pi-blocking bound by employing a variant of priority donation we developed for globally scheduled systems, instead of clustered systems. We call

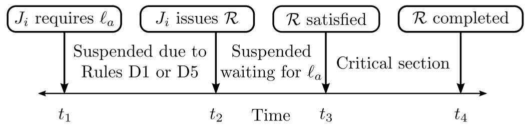


Figure 9: Phases of a resource request under the I-KGLP.

this variant of priority donation *resource-request priority donation* (*RRPD*). Under RRPD, only jobs that engage in the locking protocol are forced to suspend and donate their priority, instead of every job in the system. This is accomplished by forcing jobs to donate their priority on resource request instead of job release. This is sufficient to ensure progress (when coupled with priority inheritance, as we show in the definition of the I-KGLP) under a globally scheduled system, because priorities can be compared between all jobs.

In the following rules, let J_i be a job that requires a replica of ℓ_a at time t_1. Let t_2 be the time that J_i issues its request for ℓ_a. Let t_3 be the time J_i's request is satisfied, and t_4 be the time J_i's request is completed. Additionally, let J_d be a priority donor of J_i that first requires a resource at time t_a. J_d suspends to let J_i complete its resource request. These times are seen in Fig. 9.

D1 J_i may issue a request for ℓ_a only if it is among the m jobs of highest effective-priority that currently require ℓ_a (including jobs with incomplete resource requests). If necessary, J_i suspends until it may issue a request.

D2 J_d becomes J_i's priority donor a time t_a if (a) J_d has one of the m highest effective priorities among jobs that are waiting for ℓ_a, (b) J_i is the lowest effective-priority job with an incomplete resource request at time t_a, and (c) there are m jobs with incomplete resource requests.

D3 J_i assumes the priority of J_d (if any) during $[t_2, t_4)$. J_d is considered to have no effective priority while it is a donor.

D4 If a job J_d donating its priority to J_i is displaced from the set of the m job of highest base priority that require a replica of ℓ_a by a job J_h, then J_h becomes J_i's priority donor and J_d ceases to be a priority donor. (By Rule D3, J_i thus assumes J_h's priority.)

D5 A priority donor is suspended throughout the duration of its donation.

D6 J_d ceases to be a priority donor as soon as either (a) J_i completes its critical section (i.e., at time t_4), or (b) J_d is relieved by Rule D4.

Lemma 14. *A job with an incomplete resource request for ℓ_a has one of the m highest effective priorities among jobs that currently require ℓ_a (those that have issued a request, as well as those that are suspended waiting to issue a request).*

Proof. By contradiction. Assume J_i has an incomplete resource request but does not have one of the m highest effective priorities among jobs that require ℓ_a. Thus, there are at least m jobs of higher effective priority that require ℓ_a than J_i. Then either all m of these higher effective priority jobs have issued requests, or there is at least one such higher effective priority job that is suspended waiting to issue its request by either Rule D1 or D5. We consider these cases separately.

If all m jobs with higher effective priority than J_i that require ℓ_a issued their request for ℓ_a before J_i, then J_i would not have been allowed to issue its request by Rule D1.

If a job J_d first required a replica of ℓ_a after J_i issued its resource request, then by Rule D2, J_d is J_k's priority donor. Thus, by Rule D3, J_i has one of the highest m effective priorities among jobs that currently require ℓ_a. □

Lemma 15. *There are at most m jobs with incomplete requests for ℓ_a at any time.*

Proof. Assume for contradiction that there are more than m jobs with incomplete resource requests. Let J_i be the job that issued the $(m+1)^{st}$ resource request. By Rule D1, when J_i issued its resource request, it was one of the m highest effective priority jobs with an incomplete requests for ℓ_a. However, there were also m other jobs with incomplete requests for ℓ_a, that, by Lemma 14, had the highest m effective priorities of jobs that needed shared resources. This contradicts the assumption that J_i was allowed to issue a resource request. □

Lemma 16. *Under RRPD, if a job J_i that requires ℓ_a is s-oblivious pi-blocked it either has an incomplete resource request or it is a priority donor.*

Proof. Assume for contradiction that a job J_i is s-oblivious pi-blocked and does not have an incomplete resource request and is not a priority donor. By the definition of s-oblivious pi-blocking, if J_i is pi-blocked, then it has one of the highest m effective priorities among the set of jobs that need a shared resource. Thus by Rule D1, J_i would issue a resource request. □

I-KGLP. To construct an optimal k-exclusion lock using RRPD, we employ a similar queue structure to the O-KGLP [9]. In the I-KGLP, there is a single FIFO queue FQ_x for each replica x. Within each queue, priority inheritance is used to ensure progress. As we prove later, the maximum length of each FIFO queue is at most $\lceil \frac{m}{k} \rceil$ and thus a job can be pi-blocked by $(\lceil \frac{m}{k} \rceil - 1)$ requests. The rules of the I-KGLP are as follows.

K1 When a job J_i issues a resource request, it enqueues on the shortest FQ_x and suspends until its request is satisfied.

K2 The job at the head of each FQ_x holds replica x and all other enqueued jobs are suspended. All resource holders are ready.

K3 A job holding resource ℓ_x inherits the highest effective priority of any job in FQ_x.

K4 When a job J_i holding replica x completes its critical section, J_i is dequeued from FQ_x, and the new head of FQ_x, if any, acquires replica x.[8] J_i's priority donor, if any, may issue a resource request subject to Rule D1.

[8]As an implementation optimization, if FQ_x is left empty after J_i dequeues, a request from another FQ_y can migrate to FQ_x and acquire ℓ_x to reduce average-case pi-blocking. For example, the highest effective priority job could be chosen to migrate, or the job with the least remaining slack.

Next we analyze the worst-case pi-blocking of the I-KGLP.

Analysis. By Lemma 16, if a job is pi-blocked it either has an incomplete resource request or it is a priority donor. Thus the total duration of pi-blocking is equal to the maximum duration of time a job can be pi-blocked while it is a priority donor as well as the maximum duration of time a job can be pi-blocked while it has an incomplete resource request. We analyze each of these times separately.

Lemma 17. *The maximum length of FQ_x is $\lceil \frac{m}{k} \rceil$.*

Proof. By Lemma 15 there are no more than m jobs with incomplete requests for any given resource ℓ_a. By Rule K1, jobs are enqueued in the shortest queue upon request, and thus a job J_i will never be enqueued on a queue of length longer than $\lceil \frac{m}{k} \rceil$, otherwise there would have been a shorter queue on which J_i would have enqueued. □

Lemma 18. *A job J_i can be pi-blocked for $(\lceil \frac{m}{k} \rceil - 1)L^{max}$ in FQ_x.*

Proof. Follows from Rule K3, Lemma 17, and the fact that a job is not pi-blocked while it holds a resource. □

Lemma 19. *A job J_d can be pi-blocked for a maximum duration of $\lceil \frac{m}{k} \rceil L^{max}$ while it is a priority donor.*

Proof. In the worst-case, J_d donates its priority to a job J_i at the end of FQ_x. By Lemma 18, J_i can be pi-blocked for $(\lceil \frac{m}{k} \rceil - 1)L^{max}$. J_d also donates its priority to J_i while J_i holds the shared resource, and thus the maximum duration of pi-blocking during J_d's priority donation is $\lceil \frac{m}{k} \rceil L^{max}$. □

Theorem 11. *The maximum duration of pi-blocking a job J_i can experience due to the I-KGLP is $(2\lceil \frac{m}{k} \rceil - 1)L^{max}$ per request.*

Proof. Follows from Lemma 16, 18, and 19. □

I-KGLP as a token lock. While the I-KGLP can be employed by itself as an efficient k-exclusion locking protocol, we developed it for use as a token lock in the RNLP. On globally scheduled systems in which waiting is realized by suspending, the I-KGLP is paired with the I-RSM to achieve an efficient locking protocol supporting nested resource requests under s-oblivious analysis. First we prove that the I-KGLP is a token lock.

Lemma 20. *The I-KGLP ensures Properties T1 and T2.*

Proof. Property T1 is satisfied because the I-KGLP is a k-exclusion locking protocol. The I-KGLP employs both resource-request priority donation and priority inheritance to ensure progress for pi-blocked jobs. If a job J_d is pi-blocked duration priority donation, its priority donee J_i has sufficient effective priority to be scheduled, and the resource holding job J_a for which it is waiting inherits the priority of J_d. Thus J_d makes progress. □

Theorem 12. *When the I-KGLP is used in tandem with the I-RSM, the maximum s-oblivious pi-blocking per request is given by $(2\lceil \frac{m}{k} \rceil k - 1)L^{max}$.*

Proof. By Theorem 11, the maximum duration of pi-blocking per token request is $(2\lceil \frac{m}{k} \rceil - 1)$ times the maximum duration of time a job can hold a token (i.e., the "maximum critical section length" from the perspective of the I-KGLP). The maximum duration a job can hold a token is kL^{max}. By Theorem 1, a job can also be pi-blocked by the RSM for $(k-1)L^{max}$. Thus the total pi-blocking under the I-KGLP and the I-RSM is given by $\left(\left(2\left\lceil\frac{m}{k}\right\rceil-1\right)k+(k-1)\right)L^{max} = \left(2\left\lceil\frac{m}{k}\right\rceil k-1\right)L^{max}$. □

When we let $k=m$, we allow for increased concurrency, and also $\lceil\frac{m}{k}\rceil k=m$. Thus, when $k=m$, the worst-case duration of pi-blocking per resource request under the I-KGLP and I-RSM is $(2m-1)L^{max}$. Note that this is the same worst-case pi-blocking bound as the single-resource OMLP for globally scheduled systems given in Table 1, but the RNLP allows for nested resource requests. Under resource-request priority donation, a job can be forced to donate its priority on each resource request, while under job-release priority donation, a job only donates its priority once. Thus the per-request pi-blocking under job-release priority donation is better than resource-request priority donation, at the expense of every job in the system possibly being forced to be a priority donor. This is a trade-off that must be considered when choosing a locking protocol for globally scheduled systems.

C Priority Boosting Upper Bound

As is shown in Sec. 5, priority boosting (via the B-RSM) can be employed in partitioned systems to produce an asymptotically optimal locking protocol under s-aware analysis. When employed in a partitioned system, priority boosting can still be used as a progress mechanism, however, under s-aware analysis, priority boosting can cause significant pi-blocking for jobs that are not utilizing the locking protocol. In this section, we develop a coarse bound on the maximum duration of such pi-blocking.

Let $tardiness(T_i)$ be the tardiness bound of task T_i [7]. Note that under a hard-real-time scheduled system, $\forall T_i \in \tau, tardiness(T_i)=0$. For a task set τ, let ϕ_τ be the ratio of the maximum relative deadline plus tardiness to the minimum relative deadline.

$$\phi_\tau = \frac{\max_{T_i\in\tau}(d_i + tardiness(T_i))}{\min_{T_i\in\tau}(d_i)}$$

Theorem 13. *For a clustered systems with $c>1$, priority boosting (i.e., the B-RSM) can cause $O(\phi_\tau \cdot n)$ pi-blocking among jobs not utilizing the locking protocol.*

Proof. A job J_i can have a pending request for a duration of at most $d_i + tardiness(T_i)$. During this interval, each task T_k can release at most ϕ_τ jobs. Then each job J_k can issue N_k outermost requests, which J_i could be blocked by. Then in the worst case, J_i can be s-aware pi-blocked by $\sum_{T_k\in\tau\setminus\{T_i\}}(N_k\phi_\tau)=O(\phi\cdot n)$. □

Note that this bound is coarse. It is only meant to demonstrate that the B-RSM can be used under s-aware analysis on a clustered system without causing unbounded pi-blocking. In future work, we plan to investigate new progress mechanisms that lead to asymptotically optimal clustered locking protocols under s-aware analysis.