Optimal Multiprocessor Locking Protocols under FIFO Scheduling

Shareef Ahmed
University of North Carolina at Chapel Hill, USA

James H. Anderson
University of North Carolina at Chapel Hill, USA

Abstract

Real-time locking protocols are typically designed to reduce any priority-inversion blocking (pi-blocking) a task may incur while waiting to access a shared resource. For the multiprocessor case, a number of such protocols have been developed that ensure asymptotically optimal pi-blocking bounds under job-level fixed-priority scheduling. Unfortunately, no optimal multiprocessor real-time locking protocols are known that ensure tight pi-blocking bounds under any scheduler. This paper presents the first such protocols. Specifically, protocols are presented for mutual exclusion, reader-writer synchronization, and k-exclusion that are optimal under first-in-first-out (FIFO) scheduling when schedulability analysis treats suspension times as computation. Experiments are presented that demonstrate the effectiveness of these protocols.

1 Introduction

In recent years, a number of suspension-based multiprocessor real-time locking protocols have been developed that provide asymptotically optimal bounds on priority-inversion blocking (pi-blocking) under suspension-oblivious (s-oblivious) schedulability analysis, which treats suspension time analytically as computation time [11, 13, 14]. For mutual-exclusion (mutex) sharing, most (if not all) known asymptotically optimal locking protocols ensure a per-task s-oblivious pi-blocking bound of \( m - 1 \) request lengths on an \( m \)-processor platform under job-level fixed-priority (JLFP) scheduling [11, 13].¹ The commonality of this bound is somewhat surprising as these protocols include ones that target different scheduling strategies (e.g., partitioned, global, and clustered scheduling) and employ different mechanisms to cope with pi-blocking (e.g., priority inheritance vs. priority donation [11, 13]).

In contrast, under s-oblivious analysis, the current best lower bound yields a worst-case per-task pi-blocking bound of at least \( m - 1 \) request lengths [11]. This gap between the existing lower bound and upper bound raises an obvious question: is a pi-blocking bound of \( 2m - 1 \) request lengths fundamental under JLFP scheduling?

In this paper, we answer this question negatively by showing that, under s-oblivious analysis, the existing lower bound of \( m - 1 \) request lengths is tight under first-in-first-out (FIFO) scheduling. To show this, we give a suspension-based locking protocol for mutex sharing that ensures a per-lock-request s-oblivious pi-blocking bound of at most \( m - 1 \) request lengths under FIFO scheduling, matching the lower bound. Our protocol is designed for

¹ We refine this statement later by distinguishing between request blocking and release blocking.
clustered scheduling, so it can be applied under global and partitioned scheduling as well.

To our knowledge, this is the first truly optimal suspension-based multiprocessor locking protocol under any practical scheduling algorithm.

In designing our protocol, we exploit the fact that independent (non-resource-sharing) tasks are non-preemptive under FIFO scheduling. Such non-preemptivity is a property of the scheduler itself and does not have to be otherwise enforced: under FIFO scheduling, a newly released instance of a task cannot cause any other task instance to have insufficient priority to be scheduled. Asymptotically optimal locking protocols such as the C-OMLP [13] enforce such a property via an explicit progress mechanism. We show that such mechanisms are not required under FIFO scheduling.

Our locking protocol strengthens the case for using FIFO scheduling on multiprocessors. In addition to enabling a tight pi-blocking bound, FIFO scheduling has low overheads, ensures bounded response times (and hence bounded deadline tardiness in soft real-time systems) without capacity loss [2,22], and is sustainable with respect to execution times, meaning that it is safe to perform schedulability analysis assuming all instances of a task take its worst-case execution time (WCET) to complete. Moreover, non-preemptive execution also eases the determination of WCETs, which is challenging on modern multiprocessors [31].

According to a recent survey, around 30% of industrial respondents reported using FIFO scheduling [3].

Contributions. Our contributions are fourfold.

First, we propose a suspension-based mutex locking protocol called the optimal locking protocol under FIFO scheduling (OLP-F). The OLP-F restricts a task from issuing a resource request until it has high enough priority. Together with properties of FIFO scheduling, this ensures that the OLP-F has a tight s-oblivious pi-blocking bound under FIFO scheduling.

Second, we consider an extension of mutex sharing called k-exclusion sharing, which permits k simultaneous lock holders. For k-exclusion, we propose the optimal locking protocol for k-exclusion under FIFO scheduling (k-OLP-F) and show that it has a tight s-oblivious pi-blocking bound under FIFO scheduling.

Third, we expand even further beyond mutex sharing by considering reader-writer (RW) sharing, where exclusive resource usage is only required for write accesses and concurrent read accesses are permitted. For RW sharing, we propose the read-optimal RW locking protocol under FIFO scheduling (RW-OLP-F), which provides a tight s-oblivious pi-blocking bound for read requests under FIFO scheduling. Additionally, under the RW-OLP-F, the pi-blocking bound for write requests is just under two request lengths of optimal.

Finally, we provide experimental results that show the benefits of our locking protocols.

Organization. In the rest of this paper, we provide needed background (Sec. 2), delve further into s-oblivious pi-blocking (Sec. 3), establish a FIFO-based progress property for resource sharing (Sec. 4), present the above-mentioned protocols (Secs. 5–7), present our experimental results (Sec. 8), more fully review related work (Sec. 9), and conclude (Sec. 10).

2 System Model and Background

In this section, we provide needed definitions; Tbl. 1 summarizes the notation given here.

Task model. We consider a system of n sporadic tasks τ₁, τ₂, ..., τₙ to be scheduled on m identical processors by a FIFO scheduler. Each task τᵢ releases a potentially infinite sequence of jobs Jᵢ,1, Jᵢ,2, ... (We omit job indices if they are irrelevant.) Each task τᵢ has a period Tᵢ, specifying the minimum spacing between consecutive job releases. Each task has a relative
### Table 1 Notation summary.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>Number of tasks</td>
</tr>
<tr>
<td>( m )</td>
<td>Number of processors</td>
</tr>
<tr>
<td>( \tau_i )</td>
<td>( i^{th} ) task</td>
</tr>
<tr>
<td>( J_{i,j} )</td>
<td>( i^{th} ) job of ( \tau_i )</td>
</tr>
<tr>
<td>( T_i )</td>
<td>Period of ( \tau_i )</td>
</tr>
<tr>
<td>( C_i )</td>
<td>WCET of ( \tau_i )</td>
</tr>
<tr>
<td>( D_i )</td>
<td>Relative deadline of ( \tau_i )</td>
</tr>
<tr>
<td>( u_i )</td>
<td>Utilization of ( \tau_i )</td>
</tr>
<tr>
<td>( n_r )</td>
<td>Number of resources</td>
</tr>
<tr>
<td>( \ell_q )</td>
<td>( q^{th} ) shared resource</td>
</tr>
<tr>
<td>( N_q^\ell )</td>
<td>Maximum number of requests for ( \ell_q ) by ( \tau_i )</td>
</tr>
<tr>
<td>( L_q^\ell )</td>
<td>Maximum request length for ( \ell_q ) by ( \tau_i )</td>
</tr>
<tr>
<td>( L_{\text{max}}^\ell )</td>
<td>( \max_{1 \leq q \leq n} { L_q^\ell } )</td>
</tr>
<tr>
<td>( L_{\text{max}} )</td>
<td>( \max_{1 \leq q \leq n} { L_q^\ell } )</td>
</tr>
<tr>
<td>( \mathcal{R} )</td>
<td>A request</td>
</tr>
<tr>
<td>( r_{i,j} )</td>
<td>Release time of ( J_{i,j} )</td>
</tr>
<tr>
<td>( f_{i,j} )</td>
<td>Finish time of ( J_{i,j} )</td>
</tr>
<tr>
<td>( L_{\text{sum},h}^\ell )</td>
<td>sum of the ( h ) highest ( L_q^\ell ) values</td>
</tr>
</tbody>
</table>

**Deadline** \( D_i \). Task \( \tau_i \) has an implicit deadline if \( D_i = T_i \), a constrained deadline if \( D_i \leq T_i \), and an arbitrary deadline if no relationship between \( D_i \) and \( T_i \) is assumed. Each task has a WCET denoted \( C_i \). Task \( \tau_i \)'s utilization is defined as \( u_i = C_i/T_i \).

The release time (resp., finish time) of a job \( J_{i,j} \) is given by \( r_{i,j} \) (resp., \( f_{i,j} \)). \( J_{i,j} \) is pending at time \( t \) if \( r_{i,j} \leq t < f_{i,j} \). Jobs of a task \( \tau_i \) are sequential, i.e., \( J_{i,j+1} \) cannot commence execution before \( J_{i,j} \) finishes. Job \( J_{i,j} \) is eligible to execute at time \( t \) if \( J_{i,j} \) is pending at time \( t \) and \( t \geq f_{i,j-1} \) holds (if \( j > 1 \)). An eligible job is either ready (when it can be scheduled) or suspended (when it cannot be scheduled).

We assume time to be discrete and a unit of time to be 1.0. All scheduling decisions are taken at integer points in time. We also assume all task parameters to be integers.

**Multiprocessor scheduling.** Multiprocessor scheduling approaches can be broadly classified into two categories: partitioned and global. Under partitioned scheduling, a task is statically assigned to a processor and cannot migrate to another processor. Global scheduling allows a task to execute on any of the \( m \) processors. **Clustered scheduling** is a hybrid of partitioned and global scheduling. Under clustered scheduling, all \( m \) processors are partitioned into \( m/c \in \mathbb{N} \) clusters (without loss of generality, we assume \( m \) is an integer multiple of \( c \) each containing \( c \) processors). Each task is assigned to a cluster and can migrate only among the processors of the cluster. We consider clustered scheduling, as both partitioned and global scheduling are special cases (\( c = 1 \) and \( c = m \), respectively).

Under a job-level fixed-priority (JLFP) scheduler, each job is assigned a fixed priority throughout its execution, but a task’s priority may change over time. Common JLFP schedulers include earliest-deadline-first (EDF), FIFO, and fixed-priority scheduling algorithms. When such algorithms are employed with clustered scheduling, the \( c \) highest-priority ready jobs (if that many exist) of each cluster are scheduled on the processors of that cluster. In this paper, we consider clustered FIFO (C-FIFO) scheduling where, within a cluster, jobs with earlier release times have higher priority. We assume ties are broken arbitrarily but consistently. Hereafter, we assume all schedules to be C-FIFO unless otherwise stated.

**Resource model.** We consider a system that has a set \( \{ \ell_1, \ldots, \ell_n \} \) of shared resources. For now, we limit attention to mutual exclusion (mutex) sharing, although other notions of sharing will be considered later. Under mutex sharing, a resource \( \ell_q \) can be held by at most one job at any time. When a job \( J_i \) requires a resource \( \ell_q \), it issues a request \( \mathcal{R} \) for \( \ell_q \). \( \mathcal{R} \) is satisfied as soon as \( J_i \) holds \( \ell_q \), and completes when \( J_i \) releases \( \ell_q \). \( \mathcal{R} \) is active from its issuance to its completion. \( J_i \) must wait until \( \mathcal{R} \) can be satisfied if it is held by another job.

---

2 Our results can be adapted for non-uniform cluster sizes at the expense of additional notation.
**Table 2** Asymptotically optimal locking protocols for mutex locks under s-oblivious analysis.

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Protocol</th>
<th>Release blocking</th>
<th>Request blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global JLFP</td>
<td>OMLP [11]</td>
<td>0</td>
<td>((2m - 1)L_{\text{max}}^q)</td>
</tr>
<tr>
<td>Clustered JLFP</td>
<td>C-OMLP [13]</td>
<td>(mL_{\text{max}})</td>
<td>((m - 1)L_{\text{max}}^q)</td>
</tr>
<tr>
<td>Clustered JLFP</td>
<td>OMIP [7]</td>
<td>0</td>
<td>((2m - 1)L_{\text{max}}^q)</td>
</tr>
<tr>
<td>C-FIFO</td>
<td>OLP-F (This work)</td>
<td>0</td>
<td>((m - 1)L_{\text{max}}^q)</td>
</tr>
</tbody>
</table>

It may do so either by busy-waiting (or spinning) in a tight loop, or by being suspended by the operating system (OS) until \(R\) is satisfied. We assume that if a job \(J_i\) holds a resource \(\ell_q\), then it must be scheduled to execute.\(^3\) A resource access is called a critical section (CS).

We assume that each job can request or hold at most one resource at a time, i.e., resource requests are non-nested. We let \(N_q^i\) denote the maximum number of times a job of task \(\tau_i\) requests \(\ell_q\), and let \(L_q^i\) denote the maximum length of such a request. We define \(L_q^i\) to be 0 if \(N_q^i = 0\). Finally, we define \(L_{\text{max}}^q = \max_{1 \leq q \leq n} \{L_q^i\}\), and \(L_{\max} = \max_{1 \leq q \leq n} \{L_q^i\}\), and let \(L_{\text{sum},h}^q\) be the sum of the \(h\) largest \(L_q^i\) values. We assume all \(L_q^i\) and \(N_q^i\) to be constant.

**Priority inversions.** Priority-inversion blocking (or pi-blocking) occurs when a job is delayed and this delay cannot be attributed to higher-priority demand for processing time. Under a given real-time locking protocol, a job may experience pi-blocking each time it requests a resource—this is called request blocking—and/or upon its release and each time it releases a resource—this is called release blocking.

On multiprocessors, the formal definition of pi-blocking actually depends on how schedulability analysis is done. Of relevance to suspension-based locks, schedulability analysis may be either suspension-oblivious (s-oblivious) or suspension-aware (s-aware) [11]. Under s-oblivious analysis (the focus of this work), suspension time is analytically treated as computation time.

**Blocking complexity.** Request lengths are unavoidable in assessing maximum pi-blocking, as a request-issuing job may have to wait for a current resource-holder to complete before its request can be satisfied. As such, maximum pi-blocking bounds are usually expressed as an integer multiple of the maximum request length, i.e., the number of requests that are satisfied while a resource-requesting job is pi-blocked.

**Asymptotically optimal locking protocols.** For mutex locks, Brandenburg and Anderson established a lower bound of \(m - 1\) request lengths on per-request s-oblivious pi-blocking under any JLFP scheduler [11]. Thus, under s-oblivious analysis, an asymptotically optimal locking protocol achieves \(O(m)\) per-job pi-blocking. Locking protocols such as the OMLP [11], the OMIP [7], and the C-OMLP [13] are asymptotically optimal under JLFP scheduling. Tbl. 2 provides a summary of existing asymptotically optimal locking protocols.\(^4\)

**Optimal locking protocols.** We call a locking protocol optimal under a scheduling algorithm if it ensures a pi-blocking bound that is tight, i.e., it matches the lower bound on pi-blocking under that scheduling algorithm.

---

\(^3\) This is a common assumption in work on synchronization. It is needed for shared data, but may be pessimistic for other shared resources such as I/O devices.

\(^4\) Note that, for the C-OMLP, the \(2m - 1\) bound mentioned in Sec. 1 comes from a combination of release and request blocking.
Figure 1 A schedule illustrating s-oblivious pi-blocking for arbitrary-deadline tasks.

3 Suspension-Oblivious Pi-Blocking

Under s-oblivious schedulability analysis, each task’s WCET is inflated by the amount of worst-case s-oblivious pi-blocking any of its jobs may suffer. Such s-oblivious pi-blocking was originally defined for implicit-deadline hard real-time systems [11]. In this section, we show that this definition needs refinement for systems with arbitrary deadlines or soft timing constraints. We also provide a refined definition that works under such cases. We begin by reviewing the original definition of s-oblivious pi-blocking under clustered scheduling.

Definition 1 ([11]). 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 its cluster.

Example 2. Fig. 1 illustrates two consecutive jobs $J_{i,j}$ and $J_{i,j+1}$ of a task $\tau_i$ with $T_i = 7$ and $D_i = 11$. Job $J_{i,j+1}$ is released at time 7 and job $J_{i,j}$ finishes execution at time 10. Thus, job $J_{i,j+1}$ is pending but not eligible during the time interval $[7, 10]$. Assume that both $J_{i,j}$ and $J_{i,j+1}$ are among the $c$ highest-priority pending jobs in their cluster during $[7, 10]$. Assuming $c > 1$, by Def. 1, $J_{i,j+1}$ is s-oblivious pi-blocked during the interval $[7, 10]$.

However, $J_{i,j+1}$’s delay during $[7, 10]$ is not due to a locking-related suspension. Under s-oblivious schedulability analysis, it is not necessary to inflate $\tau_i$’s WCET to include such a delay. In fact, doing so may cause a circular problem, i.e., the inflated WCET may cause additional delays, which can then necessitate further inflation.

The above example motivates refining the notion of s-oblivious pi-blocking as follows.

Definition 3. Under s-oblivious schedulability analysis, a job $J_i$ incurs s-oblivious pi-blocking at time $t$ if $J_i$ is eligible but not scheduled and fewer than $c$ higher-priority jobs are pending in its cluster.

Example 2 (Cont’d). $J_{i,j+1}$ is pending but not eligible during the interval $[7, 10]$. Thus, it is not s-oblivious pi-blocked during that interval. However, $J_{i,j+1}$ is eligible during $[12, 13]$. Assume that $J_{i,j+1}$ is among the $c$ highest-priority eligible jobs during $[12, 13]$, but is suspended. Then, by Def. 3, $J_{i,j+1}$ is s-oblivious pi-blocked during $[12, 13]$.

4 Resource-Holder’s Progress Under FIFO Scheduling

Any real-time locking protocol needs to ensure a resource-holding job’s progress whenever a job waiting for the same resource is pi-blocked, for otherwise, the maximum per-job
pi-blocking can be very large or even unbounded. To ensure that the maximum pi-blocking is reasonably bounded, real-time locking protocols employ progress mechanisms that may temporarily raise a job’s effective priority. One such mechanism is priority inheritance [26,28], which raises the effective priority of a job holding resource \( \ell_q \) to the maximum of its priority and the priorities of all jobs waiting for \( \ell_q \). Another alternative is priority donation [14], which ensures that a job \( J_i \) can only issue a request when its priority is high enough to be scheduled. Moreover, if a job \( J_j \)’s release causes \( J_i \) to have insufficient priority to be scheduled, then \( J_j \) “donates” its priority to \( J_i \). This ensures that a resource holder is always scheduled. This property makes priority donation particularly effective under clustered scheduling.

**Progress under FIFO scheduling.** The above-mentioned progress mechanisms can be utilized to design multiprocessor locking protocols that are asymptotically optimal under any JLF scheduling policy [11,14]. Interestingly, for the case of C-FIFO scheduling, no such progress mechanism is required to design optimal locking protocols. In fact, the C-FIFO scheduling policy itself has properties that ensure the progress of a resource-holding job. The key property that enables such progress is given in the following lemma.

**Lemma 4.** Under C-FIFO scheduling, if a job \( J_{i,j} \) becomes one of the \( c \) highest-priority eligible jobs in its cluster at time \( t_h \), then it remains so during \( [t_h, f_{i,j}] \).

**Proof.** Assume for a contradiction that \( t \) is the first time instant in \( [t_h, f_{i,j}] \) such that \( J_{i,j} \) is not one of the \( c \) highest-priority eligible jobs in its cluster. Then, \( t > t_h \) holds. By the definition of time \( t \), there are at most \( c - 1 \) (resp., at least \( c \)) eligible jobs with higher priority than \( J_{i,j} \) at time \( t - 1 \geq t_h \) (resp., \( t \)) in \( J_{i,j} \)'s cluster. Thus, there is a task \( \tau_u \) that has an eligible job \( J_{u,v} \) with higher priority than \( J_{i,j} \) at time \( t \), but has no such job at time \( t - 1 \).

Since \( J_{u,v} \)'s priority exceeds \( J_{i,j} \)'s, \( r_{u,v} \leq r_{i,j} \) holds. Since \( J_{i,j} \) is eligible at time \( t_h \), \( r_{i,j} \leq t_h \) holds. Thus, \( r_{u,v} \leq t_h \) and \( J_{u,v} \) is pending at time \( t - 1 \). We now consider two cases.

**Case 1.** \( v = 1 \). In this case, \( J_{u,v} \) is also eligible at time \( t - 1 \). Thus, \( \tau_u \) has an eligible job with higher priority than \( J_{i,j} \) at time \( t - 1 \), a contradiction.

**Case 2.** \( v > 1 \). Since \( J_{u,v} \) is not eligible at time \( t - 1 \), job \( J_{u,v-1} \) is eligible at time \( t - 1 \). We have \( r_{u,v-1} < r_{u,v} \leq r_{i,j} \). Thus, \( \tau_u \) has an eligible job with higher priority than \( J_{i,j} \) at time \( t - 1 \), a contradiction.

Therefore, we reach a contradiction in both cases.

Utilizing Lemma 4, we have the following lemma.

**Lemma 5.** If a job \( J_{i,j} \) issues a request \( R \) when it is one of the \( c \) highest-priority jobs in its cluster, then \( J_{i,j} \) is always scheduled from \( R \)'s satisfaction to completion.

**Proof.** Let \( t_r, t_s, \) and \( t_c \) be the time instants when \( R \) is issued, satisfied and complete, respectively. Thus, \( t_r \leq t_s \leq t_c \) holds. Since \( J_{i,j} \) is one of the \( c \) highest-priority eligible jobs in its cluster at time \( t_r \), by Lemma 4, \( J_{i,j} \) remains one of the \( c \) highest-priority eligible jobs in its cluster throughout \( [t_r, t_c] \). Since \( R \) is satisfied at time \( t_s \geq t_r \), \( J_{i,j} \) is ready throughout \( [t_s, t_c] \). Thus, \( J_{i,j} \) is scheduled during \( [t_s, t_c] \).

Thus, by requiring a request to be issued only when the request-issuing job is one of the top-\( c \)-priority jobs in its cluster, we can ensure a resource-holder’s progress under FIFO scheduling. We exploit this property in designing our protocols. Note that the C-OMLP ensures this property by employing priority donation as its progress mechanism at the expense of additional release blocking that may be incurred by a job even if it does not require any resource [13]. Due to this, our protocols have features in common with the C-OMLP.
5 Mutex Locks

In this section, we introduce the optimal locking protocol for mutual exclusion sharing under C-FIFO scheduling (OLP-F), which achieves optimal pi-blocking under C-FIFO scheduling. To match the lower bound on pi-blocking, the OLP-F ensures that each job suffers pi-blocking for the duration of at most $m - 1$ request lengths and incurs no release blocking.

**Structures.** For each resource $\ell_q$, we have a FIFO queue $FQ_q$ that contains requests for $\ell_q$. A request $R$ is satisfied if and only if $R$ is the head of $FQ_q$.

**Rules.** When a job $J_i$ attempts to issue a request $R$ for a resource $\ell_q$, it proceeds according to the following rules.

- **M1** $J_i$ is allowed to issue $R$ only if it is one of the $c$ highest-priority eligible jobs in its cluster. $J_i$ suspends if necessary to ensure this condition.
- **M2** When $J_i$ issues $R$, $R$ is enqueued in $FQ_q$. If $J_i$ becomes the head of $FQ_q$, then it is immediately satisfied. Otherwise, it suspends.
- **M3** $R$ is satisfied when it is the head of $FQ_q$. $R$ is removed from the $FQ_q$ when it is complete.

▶ **Example 6.** Fig. 2 illustrates a C-FIFO schedule of three jobs on a two-processor cluster. $J_1$ and $J_2$ are released earlier (hence, have higher priorities) than $J_3$. Both $J_1$ and $J_2$ issue requests for resource $\ell_q$ at time 3 and $J_1$’s request is enqueued first. Assuming no job in a different cluster holds $\ell_q$, $J_1$ acquires $\ell_q$ at time 3 by Rule M2. At time 3, since $J_2$ is suspended, $J_3$ starts to execute. At time 4, $J_3$ attempts to issue a request for $\ell_q$, but it is suspended due to Rule M1 as it is not one of the top-2-priority jobs at that time. At time 6, $J_1$ releases $\ell_q$ and $J_2$’s request is satisfied according to Rule M3. Since $J_3$ becomes one of the top-2-priority jobs when $J_1$ completes, it issues a request for $\ell_q$ at time 7.

**Analysis.** To derive an upper bound on the pi-blocking suffered by a job, we first show that $FQ_q$ contains no more than $m$ requests at any time.

▶ **Lemma 7.** Under the OLP-F, at any time, $FQ_q$ contains at most $m$ requests.

**Proof.** Assume that $t$ is the first time instant when $FQ_q$ contains more than $m$ requests. Each job has at most one active request at any time. Thus, at time $t$, $FQ_q$ must contain a request $R$ issued by a job $J_i$ that is not one of the $c$ highest-priority eligible jobs in its cluster. Let $t' \leq t$ be the time instant when $J_i$ issues $R$. By Rule M1, $J_i$ is one of the $c$ highest-priority eligible jobs in its cluster at time $t'$. Since $J_i$ is not complete at time $t$, by Lemma 4, it is one of the $c$ highest-priority eligible jobs in its cluster at time $t$, a contradiction.

We now determine an upper bound on the request blocking suffered by job $J_i$ when it issues a request $R$ for resource $\ell_q$. Fig. 3 depicts the timeline of $R$ from when $J_i$ attempts...
to issue $\mathcal{R}$ to when $\mathcal{R}$ completes. Let $t_1$ be the time instant when job $J_i$ attempts to issue request $\mathcal{R}$. Let $t_2$ be the first time instant at or after time $t_1$ when $J_i$ becomes one of the top-$m$-priority eligible jobs. Therefore, by Rule M1, $\mathcal{R}$ is issued at time $t_2$. Let $t_3$ and $t_4$ be the time instants when $\mathcal{R}$ is satisfied and completes, respectively.

> **Lemma 8.** During $[t_1, t_3]$, $J_i$ incurs pi-blocking for at most $L_{\text{sum},m-1}$ time units.

**Proof.** By the definition of $t_2$, $J_i$ is not one of the top-$c$-priority eligible jobs in its cluster during $[t_1, t_2]$. Hence, $J_i$ is not pi-blocked during that time. By Lemma 4, $J_i$ is pi-blocked throughout $[t_2, t_3]$. By Lemma 5, $J_i$ is continuously scheduling during $[t_3, t_4]$. Thus, from $t_1$ to $t_4$, $J_i$ is only pi-blocked during $[t_2, t_3]$.

By Lemma 7, at most $m-1$ other requests precede $\mathcal{R}$ in FQ$_q$ at time $t_2$. By Rule M3 and Lemma 5, each job at the head of FQ$_q$ is continuously scheduled until its request is complete. Since each task has at most one eligible job and each job has at most one request at any time, $t_3 - t_2$ is not more than $L_{\text{sum},m-1}$ time units and the lemma follows. $\triangleright$

We now show that the OLP-F does not cause any release blocking under C-FIFO scheduling.

> **Lemma 9.** Under the OLP-F, no job incurs release blocking.

**Proof.** Since a resource-holding job is scheduled only when its priority is among the top $c$ in its cluster, a resource request $\mathcal{R}$ does not cause pi-blocking to any job (within and across cluster boundaries) that does not issue a request during the time $\mathcal{R}$ is satisfied. $\triangleright$

> **Theorem 10.** Under the OLP-F, $J_i$ is pi-blocked for at most $b_i = \sum_{q=1}^{n_r} N_i^q \cdot L_{\text{sum},m-1}$ time units.

**Proof.** Follows from Lemmas 8 and 9. $\triangleright$

Thus, the OLP-F is an optimal locking protocol under C-FIFO scheduling.

### 6 k-Exclusion Locks

$k$-exclusion generalizes mutual exclusion by allowing up to $k$ simultaneous lock holders; thus, mutual exclusion is equivalent to 1-exclusion. In this section, we give an optimal $k$-exclusion locking protocol under C-FIFO scheduling. We assume that a resource $\ell_q$ can be concurrently held by up to $k_q \leq m$ jobs. We begin by reviewing lower bound results for $k$-exclusion.

**Lower bound on pi-blocking.** For $k$-exclusion, Elliot et al. showed that a task system and a release sequence for it exist such that a job requesting a resource $\ell_q$ incurs s-oblivious pi-blocking for the duration of \( \left\lceil \frac{m-k_q}{k_q} \right\rceil \) request lengths under any JLFP scheduler [18].
Asymptotically optimal locking protocols. Under s-oblivious analysis, the CK-OMLP [11], the OKGLP [18], and the R²DGLP [30] ensure asymptotically optimal pi-blocking for k-exclusion. Tbl. 3 summarizes these protocols.

The k-OLP-F. We now introduce the optimal locking protocol for k-exclusion under C-FIFO scheduling (k-OLP-F), which achieves optimal pi-blocking for k-exclusion under C-FIFO scheduling. The k-OLP-F ensures that a job suffers pi-blocking for the duration of no more than \( m/k \) request lengths for each request for \( \ell_q \) and incurs no release blocking.

Structures. For each resource \( \ell_q \), we have a FIFO queue \( FQ_q \) that contains waiting requests for \( \ell_q \). We also have a queue \( SQ_q \) of length at most \( k_q \) that contains the satisfied requests for \( \ell_q \). Initially, both queues are empty. A request \( R \) is satisfied if and only if \( R \) is in \( SQ_q \).

Rules. When a job \( J_i \) attempts to issue a request \( R \) for a resource \( \ell_q \), it proceeds according to the following rules.

1. \( J_i \) is allowed to issue \( R \) only if \( J_i \) is one of the \( c \) highest-priority eligible jobs in its cluster.
2. \( J_i \) suspends if necessary to ensure this condition.
3. If the length of \( SQ_q \) is less than \( k_q \) when \( J_i \) issues \( R \), then \( R \) is enqueued in \( SQ_q \) and is immediately satisfied. Otherwise, \( R \) is enqueued in \( FQ_q \) and \( J_i \) suspends.
4. When \( R \) completes, it is removed from \( SQ_q \). If \( FQ_q \) is non-empty at that time, then the head of \( FQ_q \) is dequeued, enqueued in \( SQ_q \), and satisfied.

Example 11. Fig. 4 shows a schedule of five jobs that share a resource \( \ell_q \) with \( k_q = 2 \). Jobs \( J_1, J_2, \) and \( J_3 \) (resp., \( J_4, \) and \( J_5 \)) are FIFO scheduled on a two-processor cluster \( G_1 \) (resp., \( G_2 \)). Since \( SQ_q \) is initially empty, by Rule K2, \( J_4 \) and \( J_1 \) acquire \( \ell_q \) at times 2 and 3, respectively. Since both \( J_2 \) and \( J_5 \) are one of the top-2-priority eligible jobs in their clusters, by Rule K1, they issue requests for \( \ell_q \) at times 4 and 5, respectively. At time 5, \( J_3 \) attempts to issue a request for \( \ell_q \), but is suspended, by Rule K1. At time 5, \( J_4 \) releases \( \ell_q \) and is removed from \( SQ_q \) by Rule K3. \( J_3 \)'s request is at the head of \( FQ_q \) at time 5, so by Rule K3, it is removed from \( FQ_q \), enqueued in \( SQ_q \), and satisfied. At time 7, \( J_1 \) completes and \( J_3 \) becomes one of the top-2-priority jobs in \( G_1 \) and issues its request, by Rule K1.

Analysis. We now derive an upper bound on the pi-blocking suffered by a job under the k-OLP-F. We first derive an upper bound on the number of waiting requests in \( FQ_q \).

Lemma 12. Under the k-OLP-F, \( FQ_q \) contains at most \( m - k_q \) requests.

Proof. Assume otherwise. Let \( t \) be the first time instant such that \( FQ_q \) contains more than \( m - k_q \) requests. Thus, a new request \( R' \) is enqueued in \( FQ_q \) at time \( t \). By Rule K2, \( SQ_q \) contains \( k_q \) requests at time \( t \). Thus, the number of active requests (either satisfied or waiting) is more than \( k_q + m - k_q = m \) at time \( t \). Since each job has at most one active request at any time, there is an active request \( R \) issued by a job \( J_i \) that is not one of the \( c \) highest-priority jobs in its cluster. By Rule K1, \( J_i \) is one of the \( c \) highest-priority jobs in its cluster at time \( t' \leq t \). By Lemma 4, \( J_i \) remains as one of the \( c \) highest-priority jobs in its cluster at time \( t \), a contradiction.

Table 3 Asymptotically optimal locking protocols for k-exclusion locks under s-oblivious analysis.

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Protocol</th>
<th>Release blocking</th>
<th>Request blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered JLFP</td>
<td>CK-OMLP [11]</td>
<td>( \max_q { \frac{m}{k_q}</td>
<td>L^q_{\max} } )</td>
</tr>
<tr>
<td>Global JLFP</td>
<td>OKGLP [18]</td>
<td>0</td>
<td>( 2(m/k_q + 4)L^q_{\max} )</td>
</tr>
<tr>
<td>Global JLFP</td>
<td>R²DGLP [30]</td>
<td>0</td>
<td>( 2(m/k_q - 2)L^q_{\max} )</td>
</tr>
<tr>
<td>C-FIFO</td>
<td>k-OLP-F (This work)</td>
<td>0</td>
<td>( (m/k_q - 1)L^q_{\max} )</td>
</tr>
</tbody>
</table>
We now determine an upper bound on request blocking. We consider a job \( J_i \) that issues a request \( R \) for resource \( \ell_q \). As in Fig. 3, let \( t_1, t_2, t_3, \) and \( t_4 \) be the time instants corresponding to when \( J_i \) attempts to issue \( R \), and when \( R \) is issued, satisfied, and complete, respectively.

\[ \text{Lemma 13. For request } R, J_i \text{ suffers request blocking for at most } L^q_{\text{sum}, \left\lceil \frac{m - k}{m - k} \right\rceil \text{ time units.} \]

\[ \text{Proof. By Def. 3, } J_i \text{ does not suffer any pi-blocking during } [t_1, t_2) \text{ and } [t_3, t_4). \text{ By Lemma 4 and the definition of } t_2, J_i \text{ suffers pi-blocking during the entire duration of } [t_2, t_3), \text{ so it suffices to upper bound } (t_3 - t_2). \text{ If } SQ_q \text{ contains fewer than } k_q \text{ requests at time } t_2, \text{ then } t_3 - t_2 = 0 \text{ holds by Rule K2, so assume otherwise. At time } t_2, \text{ no two requests in } SQ_q \text{ and } FQ_q \text{ are from the same task. By Rule K3, } R \text{ is satisfied when it is dequeued from } FQ_q. \text{ Thus, by Lemma 12, at most } m - k_q \text{ requests are required to be dequeued to satisfy } R. \text{ By Rule K2, } k_q \text{ jobs hold } \ell_q \text{ throughout } [t_2, t_3). \text{ By Rule K1 and Lemma 5, each resource-holding job is always scheduled. Thus, per } L^q_{\text{sum}, h} \text{ time units during } [t_2, t_3) \text{ at least } h \cdot k_q \text{ requests complete—and hence, by Rule K3, at least } h \cdot k_q \text{ requests are dequeued from } FQ_q. \text{ Dequeuing } m - k_q \text{ requests from } FQ_q \text{ thus requires at most } L^q_{\text{sum}, \left\lceil \frac{m - k}{m - k} \right\rceil} \text{ time units, so } t_3 - t_2 \leq L^q_{\text{sum}, \left\lceil \frac{m - k}{m - k} \right\rceil}. \]

Similar to the OLP-F, no release blocking occurs under the \( k\)-OLP-F. Therefore, by Lemma 13, we have the following theorem.

\[ \text{Theorem 14. Under the } k\text{-OLP-F, } J_i \text{ suffers pi-blocking for at most } b_i = \sum_{q=1}^{m} N_i^q \cdot L^q_{\text{sum}, \left\lceil \frac{m - k}{m - k} \right\rceil} \text{ time units.} \]

Thus, the \( k\)-OLP-F is optimal for \( k\)-exclusion locking under C-FIFO scheduling.

### 7 Reader-Writer Locks

Some resources can be read without alteration. For such resources, it may be desirable to support reader-writer (RW) sharing. Here, writers have mutually exclusive access to the resource, but multiple readers can access the resource simultaneously.

Under RW sharing, it is often desirable to ensure fast read access. However, enabling fast read access may cause write requests to starve. This can happen under a read-preference RW lock that never satisfies a write request if a read request is active. Moreover, these observations give rise to an important question: what is the minimum request blocking a read request can incur without causing a write request to starve?

**Lower bound on read request blocking.** As we show next, ensuring a read request delay of \( 2L^q_{\text{max}} - 2 \) time units can in fact cause writer starvation.
Theorem 15. For \( m \geq 8 \), a task system and a release sequence for it exist such that any locking protocol that ensures request blocking of at most \( 2L_{\text{max}}^q - 2 \) time units for read requests causes unbounded request blocking for write requests under any work-conserving scheduler.

Proof. We give an example task system \( \Gamma \) and a release sequence for it supporting the claim. Let \( \tau_1, \tau_2, \ldots, \tau_m \) be \( m \) sporadic tasks scheduled on \( m \) processors. All tasks have WCETs of \( L + 1 \) time units with \( 2 \leq L \leq (m - 2)/3 \). Fig. 5 illustrates this for \( m = 8 \) and \( L = 2 \). Each job’s execution consists of 1.0 time unit of non-CS execution followed by \( L \) time units of CS execution. Tasks \( \tau_1, \tau_2, \ldots, \tau_{m-1} \) issue read requests for resource \( \ell_q \), while \( \tau_m \) issues a write request for \( \ell_q \). The periods of all tasks are \( m - 1 \). Each task has an implicit deadline.

Feasibility of \( \Gamma \). We show that \( \Gamma \) is feasible under a write-preference RW lock. Such lock does not satisfy any read request if a write request is waiting. Since \( \tau_m \) is the only writer task, under a write-preference RW lock, \( \tau_m \)’s jobs acquire \( \ell_q \) immediately (if no reader jobs hold \( \ell_q \)) or immediately after the currently satisfied read requests complete (otherwise). Thus, each of \( \tau_m \)’s jobs acquires \( \ell_q \) within \( L \) time units of its request issuance.

Since there are \( m \) tasks, a processor is always available for \( \tau_m \). Thus, with a WCET of \( L + 1 \) and resource acquisition time of at most \( L \), each job of \( \tau_m \) completes within \( L + 1 + L = 2L + 2 \leq 2(m - 2)/3 + 1 < m - 2 + 1 = n - 1 = T_i \) time units after its release.

For reader tasks \( \tau_1, \tau_2, \ldots, \tau_{m-1} \), a read request \( \mathcal{R} \) issued at time \( t \) is satisfied immediately if there is no waiting write request. Otherwise, by (E), the pending write request by \( \tau_m \)’s job is satisfied by time \( t + L \) and complete by time \( t + L + L = t + 2L \) (as a processor is available). Since \( \tau_m \) is the only writer task, after completion of the write request, there is no pending write request. Thus, \( \mathcal{R} \) is satisfied by time \( t + 2L \). With a WCET of \( L + 1 \), the job issuing \( \mathcal{R} \) completes within \( L + 1 + 2L = 3L + 1 \leq 3(m - 2)/3 + 1 = m - 2 + 1 = m - 1 = T_i \) time units after its release. Therefore, \( \Gamma \) is feasible.

Release sequence for \( \Gamma \). \( \tau_m \) releases its jobs periodically from time 1. \( \tau_1 \) releases its first job at time 0 and its subsequent jobs’ release times are defined as \( r_{1,j+1} = f_{m-1,j} - L \). The tasks release their jobs periodically from time 1.

Figure 5 A schedule illustrating Theorem 15.
release times of $\tau_i$’s jobs with $2 \leq i < m$ are $r_{i,j} = f_{i-1,j} - L$. Thus, for $2 \leq i < m$, we have

$$r_{i,j} = f_{i-1,j} - L \geq \{\text{Since } J_{i-1,j} \text{ executes for } L + 1 \text{ time units}\}$$

$$r_{i-1,j} + L + 1 - L = r_{i-1,j} + 1. \quad (1)$$

Similarly, for $\tau_1$, it can be shown that

$$r_{1,j+1} \geq r_{m-1,j} + 1. \quad (2)$$

We now show that consecutive jobs of $\tau_i$ with $i < m$ are released at least $T_i$ time units apart. For $2 \leq i < m$, by (1), we have

$$r_{i,j+1} \geq r_{i-1,j+1} + 1$$

$$\geq \{\text{Applying (1) repeatedly for } i - 2 \text{ times}\}$$

$$r_{1,j+1} + 1 + (i - 2)$$

$$\geq \{\text{By (2)}\}$$

$$r_{m-1,j} + 1 + (i - 1)$$

$$\geq \{\text{Applying (1) repeatedly for } m - 1 - i \text{ times}\}$$

$$r_{i,j} + (m - 1 - i) + i$$

$$= r_{i,j} + m - 1$$

$$= r_{i,j} + T_i. \quad (3)$$

Similarly, we can show that consecutive jobs of $\tau_1$ are released at least $T_1$ time units apart.

We now show that each job of $\tau_i$ with $i < m$ is eligible when it is released by showing that $J_{i,j}$ completes before $J_{i,j+1}$’s release. For $2 \leq i < m - 1$, in the third step of the derivation of (3), applying (1) repeatedly for $m - 2 - i$ times instead of $m - 1 - i$ times, we have $r_{i,j+1} \geq r_{i+1,j} + (m - 2 - i) + i = r_{i+1,j} + m - 2$. Since $L \leq (m - 2)/3 < m - 2$ and $r_{i+1,j} = f_{i,j} - L$, we get $r_{i,j+1} > r_{i+1,j} + L = f_{i,j}$. For $i = m - 1$, the first step in the derivation of (3) yields $r_{m-1,j} \geq r_{1,j} + 1 + (m - 1 - 2) = r_{1,j} + m - 2 > r_{1,j+1} + L$. Since $r_{1,j+1} = f_{m-1,j} - L$, we get $r_{m-1,j} > f_{m-1,j}$. For $i = 1$, applying (1) in (2) repeatedly for $m - 3$ times, we have $r_{1,j+1} \geq r_{2,j} + m - 2 > r_{2,j} + L = f_{1,j}$. Thus, $r_{i,j+1} > f_{i,j}$ for $i < m$.

**Finishing up.** We now prove the theorem by showing that $J_{m,1}$’s write request is never satisfied if the request delay for read requests is at most $2L - 2$. Assume that $J_{m,1}$’s request is satisfied at time $t$. We have $t > 2$, as $J_{m,1}$ issues its request at time 2 and $J_{1,1}$ holds $\ell_q$ then (under a work-conserving scheduling policy, $J_{1,1}$ acquires $\ell_q$ at time 1). Since the scheduling policy is work-conserving, a job $J_{i,j}$ must release $\ell_q$ at time $t$. Thus, $f_{i,j} = t$.

By the job release pattern of $\tau_1, \tau_2, \ldots, \tau_{m-1}$, there exists a job $J_{u,v}$ such that $r_{u,v} = f_{i,j} - L = t - L$. Since each job is eligible when it is released and there are $m$ tasks, $J_{u,v}$ issues a read request $R$ at time $r_{u,v} + 1 = t - L + 1 < t$ (as $L \geq 2$). Since $J_{m,1}$’s write request is satisfied at time $t$, $R$ cannot be satisfied before time $t + L$. Since the task count is $m$, $J_{u,v}$ is pi-blocked for a duration of at least $t + L - (t - L + 1) = 2L - 1$ time units. Thus, request blocking for read requests exceeds $2L - 2$ time units, reaching a contradiction. □

Thus, read request blocking of at least $2L_q^{\text{max}} - 1$ time units is fundamental to avoid writer starvation. We now establish a lower bound on write request blocking when read
requests suffer request blocking for at most \(2L_{\text{max}}^6 - 1\) time units.\(^5\)

**Theorem 16.** For \(m \geq 4\), there exists a task system and a release sequence for it such that any locking protocol that ensures at most \(2L_{\text{max}}^6 - 1\) read request blocking causes write request blocking of \((2m - 5)L_{\text{max}}^6 - 1\) time units under any work-conserving scheduler.

**Proof.** Let \(\tau_1, \tau_2, \ldots, \tau_n\) be \(n\) tasks scheduled on \(m \geq 4\) processors, where \(n = 2m - 4\). Each task has a WCET of \(L + 1\) time units with \(L \geq 1\). Fig. 6 illustrates this for \(m = 5\) and \(L = 3\). Each job’s execution consists of 1.0 time unit of non-CS execution followed by \(L\) time units of CS execution. Tasks \(\tau_1, \tau_2, \ldots, \tau_{m-2}\) issue write requests for resource \(\ell_q\), while \(\tau_{m-1}, \tau_m, \ldots, \tau_{2m-4}\) issue read requests for \(\ell_q\). Each task’s period is \(T \geq (2m - 4) \cdot (L + 1)\).

The task WCETs sum to \((2m - 4) \cdot (L + 1)\), so assuming implicit deadlines, the task system can be scheduled by sequentially executing the jobs on a single processor (i.e., it is feasible).

Tasks \(\tau_1, \tau_2, \ldots, \tau_{m-2}\) release their first jobs at time 1. Task \(\tau_{m-1}\) releases its first job at time 0. For \(i > m - 1\), the release time of \(J_i,1\) is determined as \(r_{i,1} = f_{i-1,1} - 1\). Hence, from time 0, there is always an eligible first job of a task until all first jobs are complete.

Since all WCETs sum to \((2m - 4) \cdot (L + 1)\), under a work-conserving scheduler, the first job of each task completes by time \((2m - 4) \cdot (L + 1) \leq T\). Subsequent job release times can be easily defined so that each task’s consecutive job releases are at least \(T\) time units apart.

We now prove that each first job \(J_i,1\) always incurs pi-blocking when it is waiting for \(\ell_q\).

For any job \(J_i,1\) with \(i > m\), we have \(r_{i,1} = f_{i-1,1} - 1 \geq r_{i-1,1} + L + 1 - 1 = f_{i-2,1} - 1 + L\).

Since \(L \geq 1\), we have \(r_{i,1} \geq f_{i-2,1}\). Thus, at most two first jobs of the last \(m - 2\) tasks are pending at the same time. Therefore, at most \(m - 2 + 2 = m\) first jobs are pending at any time, which implies that a job \(J_i,1\) incurs pi-blocking if it is waiting.

Finally, we prove the claim of the theorem by showing that there is a writer job that incurs pi-blocking for the duration of \((2m - 5)L + 1\) time units. Job \(J_{m-1,1}\) issues a read request at time 1 and acquires \(\ell_q\) (as the scheduling policy is work-conserving). Fig. 6 illustrates this.

Each job \(J_i,1\) with \(i < m - 1\) issues a write request at time 2.

Each job \(J_i,1\) with \(i > m - 1\) (e.g., the jobs of \(\tau_5\) and \(\tau_6\) in Fig. 6) is released 1.0 time unit before \(J_{i-1,1}\) completes and issues a read request when \(J_{i-1,1}\) completes. Thus, \(J_i,1\)’s read request cannot be delayed to satisfy two or more pending write requests without incurring

---

\(^5\) Assuming higher read request blocking would yield a smaller lower bound on write request blocking. Note that deriving tight lower bounds for RW locks is much more complicated than for the other locks considered in this paper because much leeway exists regarding the interplay between readers and writers.
read request blocking of at least 2\(L\) time units. As a result, at most one write request can be satisfied between two consecutive read requests. Thus, there is a write request from a job \(J_{u,1}\) with \(i < m - 1\) (e.g., \(\tau_3\)’s job in Fig. 6) that must be satisfied after all read and write requests of each job \(J_{i,1}\) with \(i \neq u\) complete.

Since \(J_{u,1}\) issues its request at time 2 and \(J_{m-1,1}\) (e.g., \(\tau_4\)’s job in Fig. 6) acquires \(\ell_q\) at time 1, \(J_{m-1,1}\) pi-blocks \(J_{u,1}\) for \(L - 1\) time units. The stated job release pattern ensures that no two of the remaining \(m - 3\) read requests (e.g., those by \(\tau_5\) and \(\tau_6\) in Fig. 6) overlap, so they pi-block \(J_{u,1}\) for \((m - 3)L\) time units. Finally, \(J_{u,1}\) is pi-blocked by each of the other \(m - 3\) write requests (e.g., those by \(\tau_1\) and \(\tau_2\) in Fig. 6) for \((m - 3)L\) time units. Thus, \(J_{u,1}\) incurs pi-blocking for \(L - 1 + (m - 3)L + (m - 3)L = (2m - 5)L - 1\) time units. □

For simplicity, Theorems 5 and 16 are stated for work-conserving scheduling. However, both theorems are also true under a wider class of schedulers and locking protocols that are top-c-work-conserving. On a c-processor cluster, a top-c-work-conserving scheduling ensures that any top-c-highest priority ready job immediately acquires a shared resource (including processor) if such a resource is idle. Note that a work-conserving scheduler and locking protocol combination is also top-c-work-conserving.

Asymptotically optimal RW locking protocols. For RW locks, the CRW-OMLP is an asymptotically optimal locking protocol under clustered JLFP scheduling [11]. The CRW-OMLP is a phase-fair RW locking protocol. Phase-fair RW locks satisfy read and write requests in alternating phases [12]. At the beginning of a reader phase, all waiting read requests are satisfied simultaneously, while at the beginning of a writer phase, a single waiting write request is satisfied.Tbl. 4 summarizes the CRW-OMLP.

The RW-OLP-F. We now introduce the read-optimal RW locking protocol under C-FIFO scheduling (RW-OLP-F), which achieves optimal pi-blocking for read requests under C-FIFO scheduling. The RW-OLP-F is a phase-fair RW locking protocol that achieves \(2L_q^{\text{max}}\) - 1 (resp., \((2m - 3)L_q^{\text{max}}\)) request blocking for read (resp., write) requests—here, however, we only prove a bound of \(2L_q^{\text{max}}\) for reads due to space limitation. Unlike the CRW-OMLP, the RW-OLP-F has no release blocking under C-FIFO scheduling.

Structures. For each resource \(\ell_q\), we have two queues \(RQ_q^1\) and \(RQ_q^2\) that contain read requests for \(\ell_q\), and a FIFO queue WQ\(q\) that contains write requests for \(\ell_q\). One of the read queues acts as a collecting queue and the other acts as a draining queue. The roles of \(RQ_q^1\) and \(RQ_q^2\) alternate, i.e., each switches over time between being the collecting queue and being the draining queue. Initially, \(RQ_q^1\) is the collecting queue and \(RQ_q^2\) is the draining queue.

Reader rules. Assume that a job \(J_i\) attempts to issue a read request \(R\) for resource \(\ell_q\). Let \(RQ_q^1\) and \(RQ_q^2\) be the collecting and draining queues, respectively, when \(J_i\) issues \(R\).

1. \(J_i\) is allowed to issue \(R\) only if it is one of the \(c\) highest-priority eligible jobs in its cluster. \(J_i\) suspends if necessary to ensure this condition.
2. If WQ\(q\) is empty when \(J_i\) issues \(R\), then \(R\) is immediately satisfied and enqueued in \(RQ_q^2\). Otherwise, \(J_i\) suspends and \(R\) is enqueued in \(RQ_q^1\).

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Protocol</th>
<th>Release blocking</th>
<th>Read request blocking</th>
<th>Write request blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered JLFP</td>
<td>CRW-OMLP [11]</td>
<td>(2mL_q^{\text{max}})</td>
<td>(2L_q^{\text{max}})</td>
<td>((2m - 1)L_q^{\text{max}})</td>
</tr>
<tr>
<td>C-FIFO</td>
<td>RW-OLP-F (This work)</td>
<td>0</td>
<td>(2L_q^{\text{max}} - 1)</td>
<td>((2m - 3)L_q^{\text{max}})</td>
</tr>
</tbody>
</table>

Table 4: Asymptotically optimal locking protocols for RW locks under s-oblivious analysis.
**Writer rules.** When a job $J_w$ attempts to issue a write request $R$ for a resource $\ell_q$, it proceeds according to the following rules.

**W1** $J_w$ is allowed to issue $R$ only if it is one of the $c$ highest-priority eligible jobs in its cluster. $J_w$ suspends if necessary to ensure this condition.

**W2** If $RQ^L_q$, $RQ^d_q$, and $WQ_q$ are empty when $R$ is issued, then $R$ is immediately satisfied and enqueued in $WQ_q$. Otherwise, $R$ is enqueued in $WQ_q$ and $J_w$ suspends.

**W3** Let $RQ^L_q$ and $RQ^d_q$ be the draining and collecting queues, respectively, when $R$ is the head of $WQ_q$. $R$ is satisfied when $R$ is the head of $WQ_q$ and $RQ^d_q$ is empty. When $R$ is complete, $R$ is dequeued from $WQ_q$ and if $RQ^L_q$ is non-empty, then $RQ^L_q$ (resp., $RQ^d_q$) becomes the draining (resp., collecting) queue. Otherwise ($RQ^d_q$ is empty), the new head of $WQ_q$ (if any) is satisfied.

**Analysis.** We now determine an upper bound on request blocking. For $m \leq 2$, by Lemma 4 and Rules R1 and W1, there are at most two active requests and at most one waiting request at any time, so request blocking is at most $L_{max}^q$ time units for both reads and writes. Henceforth, we assume $m \geq 3$. The following lemma follows from Lemma 4 and Rules R1 and W1; we omit its proof as it is similar to Lemma 7.

**Lemma 17.** The total number of requests in $RQ^L_q$, $RQ^d_q$, and $WQ_q$ is at most $m$.

We now give two helper lemmas.

**Lemma 18.** If a write request $R$ is the head of $WQ_q$ at time $t$, then it is satisfied by time $t + L_{max}^q$.

**Proof.** Let $RQ^L_q$ and $RQ^d_q$ be the collecting and draining queue, respectively, at time $t$. If $R$ is not satisfied at time $t$, then by Rule W3, $RQ^d_q$ is non-empty at time $t$. By Rule R3, jobs with requests in $RQ^d_q$ hold $\ell_q$ at time $t$. Let $t'$ be the time instant when all such requests are complete. By Lemma 5 and Rule R1, $t' \leq t + L_{max}^q$. By Rule R2, no read requests are enqueued in $RQ^d_q$ during $[t, t')$. Thus, $RQ^d_q$ becomes empty at time $t'$. By Rule W3, $R$ is satisfied at time $t'$. Thus, the lemma holds.

**Lemma 19.** If a write request $R$ is the head of $WQ_q$ at time $t$, then it is complete by time $t + 2L_{max}^q$.

**Proof.** By Lemma 18, $R$ is satisfied by time $t + L_{max}^q$. By Lemma 5 and Rule W1, $R$ completes within $L_{max}^q$ time units after being satisfied. Thus, the lemma holds.

We now determine an upper bound on the request blocking suffered by a job when it issues a read request. We consider a job $J_i$ that issues a read request $R$ for resource $\ell_q$. As depicted in Fig. 3, let $t_1, t_2, t_3$, and $t_4$ be the time instants corresponding to when $J_i$ attempts to issue $R$, and when $R$ is issued, satisfied, and complete, respectively. In the lemma below, for simplicity, we show that request blocking for read requests is at most $2L_{max}^q$. A tight bound of $2L_{max}^q - 1$ can be established by a detailed analysis involving multiple cases.
Lemma 20. For a read request $R$, $J_i$ suffers request blocking for at most $2L_{\text{max}}^q$ time units.

Proof. $J_i$ suffers pi-blocking for the duration of $[t_2, t_3)$. Let $RQ_q^c$ and $RQ_q^d$ be the collecting and draining queue, respectively, at time $t_2$. If $WQ_q$ is empty at time $t_2$, then $t_2 = t_3$ holds according to Rule R2, so assume otherwise. By Rule R2, $R$ is enqueued in $RQ_q^c$. Let $R'$ be the request at the head of $WQ_q$ at time $t_2$. Let $t'_2$ be the time instant when $R'$ completes. By Lemma 19, $t'_2 \leq t_2 + 2L_{\text{max}}^q$ holds. By Rule W3, $RQ_q^c$ becomes the draining queue at time $t'_2$. Thus, by Rule R3, all requests in $RQ_q^c$, including $R$, are satisfied at time $t'_2$, implying $t_3 = t'_2$. Therefore, we have $t_3 - t_2 \leq 2L_{\text{max}}^q$. \hfill □

Finally, we give an upper bound on the request blocking incurred by a job when issuing a write request. Let $J_w$ be a job that issues a write request $R$ at time $t$.

Lemma 21. For a write request $R$, $J_w$ incurs request blocking for at most $(2m-3)L_{\text{max}}^q$ time units.

Proof. If no request holds $\ell_q$ at time $t$, then by Rule W2, $R$ is immediately satisfied. This leaves two cases.

Case 1. A job with a read request holds $\ell_q$ at time $t$. By Lemma 17, $RQ_q^c$, $RQ_q^d$, and $WQ_q$ hold at most $m$ requests at time $t$. Since there is an active read request, at most $m-2$ write requests precede $R$ in $WQ_q$. By Rule W3, each of those write requests becomes the head of $WQ_q$ when its preceding write request completes. By Lemma 19, a write request at the head of $WQ_q$ completes within $2L_{\text{max}}^q$ time units from when it becomes the head. Thus, all $m-2$ write requests that precede $R$ in $WQ_q$ are complete by time $t + (2m-2)L_{\text{max}}^q$. By Lemma 18, after becoming the head of $WQ_q$, $R$ is satisfied within an additional $L_{\text{max}}^q$ time units. Thus, $R$ is satisfied by time $t + (2m-3)L_{\text{max}}^q$.

Case 2. A job with a write request $R'$ holds $\ell_q$ at time $t$. We consider two subcases.

Case 2a. $WQ_q$ contains $m$ requests at time $t$. Thus, $m-1$ requests precede $R$ in $WQ_q$. By Lemma 5 and Rule W1, $R'$ completes within $L_{\text{max}}^q$ time units from $t$. By Lemma 4 and Rules R1 and W1, no requests are issued before $R'$ completes. Thus, by Rule W3, the write request $R''$ following $R'$ is satisfied when $R'$ is complete. By Lemma 5 and Rule W1, $R''$ completes within $L_{\text{max}}^q$ time from when it is satisfied. Thus, the top two requests in $WQ_q$ are complete by time $t + 2L_{\text{max}}^q$. By Lemma 19, each of the remaining $m-3$ write requests preceding $R$ is complete within $2L_{\text{max}}^q$ time units after becoming the head of $WQ_q$. Thus, $R$ becomes the head of $WQ_q$ by time $t + 2L_{\text{max}}^q + (2m-3)L_{\text{max}}^q = t + 2(m-2)L_{\text{max}}^q$. By Lemma 18, $R$ is satisfied within $L_{\text{max}}^q$ time units after becoming $WQ_q$’s head. Thus, $R$ is satisfied by time $t + (2m-3)L_{\text{max}}^q$.

Case 2b. $WQ_q$ contains at most $m-1$ requests at time $t$. Thus, at most $m-2$ requests precede $R$ in $WQ_q$. By Lemma 5, $R'$ completes within $L_{\text{max}}^q$ time units from $t$. By Lemma 19, each of the remaining $m-3$ write requests preceding $R'$ completes within $2L_{\text{max}}^q$ time units from when it becomes the head of $WQ_q$. Thus, $R$ becomes the head of $WQ_q$ within $L_{\text{max}}^q + (2m-3)L_{\text{max}}^q = (2m-5)L_{\text{max}}^q$ time units from $t$. By Lemma 18, $R$ is satisfied within $L_{\text{max}}^q$ time units after becoming $WQ_q$’s head. Thus, $R$ is satisfied by time $(2m-4)L_{\text{max}}^q$. \hfill □

Similar to the OLP-F, no job suffers release blocking due to a resource-holding job under the RW-OLP-F. By Lemma 20 and 21 and letting $N_{i}^{q,w}$ and $N_{i}^{q,t}$ denote the maximum number of read and write requests for $\ell_q$ by $\tau_i$, we have the following.
Theorem 22. Under the RW-OLP-F, $J_i$ is pi-blocked for at most

$$b_i = \sum_{q=1}^{n_c} \left( N_i^q r \cdot 2L_{\text{max}}^q + N_i^{q,w} \cdot (2m - 3)L_{\text{max}}^q \right).$$

As mentioned already, the $2L_{\text{max}}^q$ term above can be replaced by $2L_{\text{max}}^q - 1$ at the expense of more lengthy analysis. By Rules R1, R2, W1, and W2, FIFO scheduling and RW-OLP-F ensures top-$c$-work-conserving property. Thus, by Theorems 15 and 16, the RW-OLP-F ensures optimal request-conserving blocking for read requests, while ensuring that the request blocking for write requests is just under two request lengths of optimal.

8 Experimental Evaluation

In this section, we present the results of experiments we have conducted using the SchedCAT toolkit [1] to evaluate our proposed locking protocols.

Task system generation. Our task-system generation method is similar to that used in prior locking-related schedulability studies [6,9,32]. We generated task systems randomly for systems with $\{4,8,16\}$ processors. For each processor count, we generated task systems that have a normalized utilization, i.e., $\sum_{i=1}^{n} u_i/m$, from 0.2 to 0.9 with a step size of 0.1. We chose the number of tasks uniformly from $[2m, 150]$. We generated each task's utilization uniformly following procedures from [19]. We chose each task's period randomly from $[3,33]$ms (short), $[10,100]$ms (moderate), or $[50,500]$ms (long). We set each task's WCET $C_i$ to $T_i \cdot u_i$ rounded to the next microsecond.

We considered $\{m/4,m/2,m,2m\}$ number of shared resources. For each $\tau_i$ and resource $\ell_q$, we selected $\tau_i$ to access resource $\ell_q$ with probability $p^{\text{acc}} \in \{0.1,0.25,0.5\}$. If so selected, $\tau_i$ was defined to access $\ell_q$ via $N_i^q \in \{1,2,\ldots,5\}$ requests. For each $N_i^q > 0$, we chose the maximum request length $L_i^q$ randomly from three uniform distributions ranging over $[1,15]$us (short), $[1,100]$us (medium), or $[5,1280]$us (long). A chosen $L_i^q$ value was decreased accordingly if it caused the sum of all request length of $\tau_i$ to exceed $C_i$. For each combination of $m$, normalized utilization, $T_i$, $L_i^q$, $p^{\text{acc}}$, and $n_r$, we generated 1,000 task systems. We call each combination of these parameters a scenario.

Experiment 1. In our first experiment, we considered mutex sharing. Each task had a soft timing constraint, meaning that it was deemed schedulable if its response time was bounded.

We considered resource synchronization under the OLP-F, the OMLP [11], the C-OMLP [13], the OMIP [7], and the FMLP [5]. For the OLP-F, each task system’s schedulability was tested under global FIFO scheduling [22]. For the remaining protocols, s-oblivious schedulability tests were performed under global EDF scheduling [16]. For each scenario, we assessed acceptance ratios, which give the percentage of task systems that were schedulable under each locking protocol. We present a representative selection of our results in Fig. 7.

Observation 1. The average improvement under the OLP-F over the OMLP, the C-OMLP, the OMIP, and the FMLP was 20.2%, 14.9%, 16.4%, and 27.5%, respectively.

This can be seen in insets (a) and (b) of Fig. 7. Unsurprisingly, schedulability was improved under the OLP-F because of lower pi-blocking compared to the other protocols. In some cases, as depicted in Fig. 7(b), all protocols had similar schedulability. This can

6 The same schedulability test also applies for a wider class of global schedulers including FIFO.
Experiment 2. This experiment pertains to RW sharing. To generate task systems, we used one additional parameter $p_{\text{write}} \in \{0.1, 0.2, 0.3, 0.5, 0.7\}$. We defined each resource access to be a write (resp., read) access with probability $p_{\text{write}}$ (resp., $1 - p_{\text{write}}$). In this experiment, we considered soft real-time scheduling with resource synchronization under the RW-OLP-F, the CRW-OMLP [13], and the OLP-F. Each task system’s schedulability was tested under global FIFO scheduling when the OLP-F and the RW-OLP-F were employed, and under global EDF scheduling otherwise. We have the following observation.

Observation 2. The RW-OLP-F improved schedulability over the CRW-OMLP across all scenarios. The RW-OLP-F had less schedulability than the OLP-F when write accesses were more frequent, i.e., high $p_{\text{write}}$ values.

This can be seen in Fig. 7(c). The improved pi-blocking bound enabled higher schedulability under the RW-OLP-F. The RW-OLP-F had better or equal schedulability than the OLP-F across 90% of the total scenarios. Since the RW-OLP-F has higher write request blocking compared to the OLP-F (which does not have optimal read request blocking), the OLP-F had better schedulability than the RW-OLP-F when $p_{\text{write}}$ values are high, e.g., $p_{\text{write}} = 0.7$. 

Experiment 3. In this experiment, we considered hard real-time scheduling under mutex locks. For each task $\tau_i$, we randomly chose a relative deadline between $[T_i, 2T_i]$. We considered partitioned scheduling because of the lack of hard real-time schedulability tests for global
FIFO scheduling. We used the worst-fit bin packing heuristic to partition each task system. We compared schedulability under the OLP-F and partitioned FIFO scheduling with the partitioned OMLP (the C-OMLP with \( c = 1 \)) and partitioned EDF scheduling.

**Observation 3.** The partitioned OMLP had better schedulability compared to the OLP-F.

This can be seen in Fig. 7(d). Despite having lower pi-blocking and bounded response times, the partitioned OMLP enabled better schedulability because of the optimality of uniprocessor EDF in scheduling hard real-time workloads. Note that, unlike for EDF, the employed FIFO schedulability test was non-exact [4].

### Related Work

The literature on suspension-based multiprocessor real-time locking protocols is quite vast (e.g., [7, 11, 13–15, 17, 20, 21, 23–25, 27, 29]). An excellent recent survey is given in [10]. Below, we comment further on a few specific relevant protocols.

In work on mutex locks, the FMLP [5] was the first multiprocessor locking protocol to be studied under \( s \)-oblivious analysis. While relatively simple, the FMLP has \( O(n) \) pi-blocking under \( s \)-oblivious analysis. The first mutex protocols that were shown to have asymptotically optimal \( s \)-oblivious pi-blocking were the OMLP and its variants, which include protocols applicable under partitioned, global, and clustered JLFP scheduling [11, 13, 14]. In later work, the OMIP [7] was presented; it upholds an independence preserving property that results in asymptotically optimal \( s \)-oblivious pi-blocking under clustered JLFP scheduling.

The first multiprocessor mutex locking protocols were designed to be studied under \( s \)-aware analysis. Many of these protocols (e.g., the MPCP [27], the PPCP [17], the PIP [26], etc.) were inspired by classical uniprocessor locking protocols. The FMLP+ [9] is an extension of the FMLP that has been shown to have asymptotically optimal \( s \)-aware pi-blocking under clustered JLFP scheduling. In other work, linear-programming techniques were proposed that enable improved \( s \)-aware analysis of various protocols, including the PIP, the PPCP, and the FMLP, under global and partitioned fixed-priority scheduling [8, 32].

### Conclusion

In this paper, we have presented optimal suspension-based multiprocessor locking protocols for mutex, \( k \)-exclusion, and RW synchronization. In particular, we have shown that the \( s \)-oblivious lower bound of \( m - 1 \) request lengths for mutex locks is indeed tight under FIFO scheduling. We have also provided a tight \( s \)-oblivious lower bound on read-request blocking for RW locks. All three locking protocols presented herein can be used together in the same system without jeopardizing the presented analysis. Moreover, spin-based versions of these protocols can be easily obtained by following the same design principles.

For some non-FIFO JLFP schedulers, it may be possible that \( 2m - 1 \) request lengths is indeed a tight lower bound on \( s \)-oblivious pi-blocking for mutex locks. Showing this would require a new lower-bound proof. As seen in Sec. 7, finding task systems that justify such a lower bound can be quite difficult. The results of this paper show that any task system used to justify a \( 2m - 1 \) lower bound must necessarily not be FIFO-scheduled. In some sense, this is unfortunate, as FIFO schedules are somewhat easier to deal with in lower-bound arguments, given that having “top-\( c \)” priority is a stable property for FIFO-scheduled jobs.
References


