SHARING NON-PROCESSOR RESOURCES IN MULTIPROCESSOR REAL-TIME SYSTEMS

Bryan C. Ward

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2016

Approved by:

James H. Anderson

Sanjoy K. Baruah

Björn B. Brandenburg

Alan Burns

F. Donelson Smith

ABSTRACT

BRYAN C. WARD: Sharing Non-Processor Resources in Multiprocessor Real-Time Systems
(Under the direction of James H. Anderson)

Computing devices are increasingly being leveraged in cyber-physical systems, in which computing devices sense, control, and interact with the physical world. Associated with many such real-world interactions are strict timing constraints, which if unsatisfied, can lead to catastrophic consequences. Modern examples of such timing constraints are prevalent in automotive systems, such as airbag controllers, anti-lock brakes, and new autonomous features. In all of these examples, a failure to correctly respond to an event in a timely fashion could lead to a crash, damage, injury and even loss of life. Systems with imperative timing constraints are called real-time systems, and are broadly the subject of this dissertation.

Much previous work on real-time systems and scheduling theory assumes that computing tasks are *independent*, *i.e.*, the only resource they share is the platform upon which they are executed. In practice, however, tasks share many resources, ranging from more overt resources such as shared memory objects, to less overt ones, including data buses and other hardware and I/O devices. Accesses to some such resources must be *synchronized* to ensure *safety*, *i.e.*, logical correctness, while other resources may exhibit better run-time performance if accesses are explicitly synchronized. The goal of this dissertation was to develop new synchronization algorithms and associated analysis techniques that can be used to synchronize access to many classes of resources, while improving the overall resource utilization, specifically as measured by real-time schedulability.

Towards that goal, the *Real-Time Nested Locking Protocol* (RNLP), the first multiprocessor real-time locking protocol that supports lock nesting or fine-grained locking is proposed and analyzed. Furthermore, the RNLP is extended to support reader/writer locking, as well as k-exclusion locking. All presented RNLP variants are proven optimal. Furthermore, experimental results demonstrate the schedulability-related benefits of the RNLP.

Additionally, three new synchronization algorithms are presented, which are specifically motivated by the need to manage shared hardware resources to improve real-time predictability. Furthermore, two new

classes of shared resources are defined, and the first synchronization algorithms for them are proposed. To analyze these new algorithms, a novel analysis technique called *idleness analysis* is presented, which can be used to incorporate the effects of blocking into schedulability analysis.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help and support of many people. First and foremost, I would like to thank my advisor Jim Anderson for his guidance and patience during the course of my time at UNC. I'd also like to thank my dissertation committee, Sanjoy Baruah, Björn Brandenburg, Alan Burns, and Don Smith, for their feedback on my research and this document.

I'd also like to thank all of my co-authors, Micaiah Chisholm, Glenn Elliott, Jeremy Erickson, Cheng-Yang Fu, Jonathan Herman, Chris Kenna, Namhoon Kim, Catherine Nemitz, Nathan Otterness, and Abhilash Thekkilakattil, who all helped me on many different projects, and often stayed up late into the night and even early into the morning to polish a paper before a last-minute submission deadline. I'd like to also thank the rest of the UNC real-time systems group, who provided helpful conversations and feedback on ideas and presentations.

Outside of research, I am tremendously thankful for the friends I have made throughout my time at UNC, including Jacob Bartel, Bipasa Chattopadhyay, Glenn Elliot, Victor Heorhiadi, Jonathan Herman, Chris Kenna, Alan Kuntz, Pavel Krajcevski and Adam Domby. Finally, I'd like to thank my parents for their love and support through both graduate school and the entirety of my life. And lastly, I cannot express how thankful I am for my wife, Heather, who has been so loving and supportive throughout the trials and tribulations of graduate school.

The funding for this research was provided by NSF grants CSR 1016954, CSR 1115284, CSR 1218693, CSR 1239135, CPS 1239135, CNS 1409175, CPS 1446631, and a Graduate Research Fellowship (GRFP); AFOSR grants FA9550-09-1-0549, and FA9550-14-1-0161; AFRL grant FA8750-11-1-0033; ARO grants W911NF-09-1-0535, and W911NF-14-1-0499; with additional support from General Motors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abi	This has to be removed eventually
DGL	Dynamic Group Lock
C-EDF	Clustered EDF
DGL	Dynamic Group Lock(ing)
DRAM	Dynamic Random Access Memory
EDF	Earliest Deadline First
FIFO	First-In, First-Out
FP	Fixed Priority
G-EDF	Global EDF
GPU	Graphics Processing Unit
HRT	Hard Real Time
I/O	Input/Output
IPC	Inter-Process Communication
IPI	Inter-Processor Interrupt
JLFP	Job-Level Fixed Priority
L1	Level 1 Cache
L1-I	Level 1 Instruction Cache
L1-D	Level 1 Data Cache
L2	Level 2 Cache
LIFO	Last-In, First-Out
LLC	Last-Level Cache
LP	Linear Program
MMU	Memory Management Unit
PCIe	Peripheral Component Interconnect Express
PF	Phase Fair
P-FP	Partitioned Fixed Priority
RAM	Random Access Memeory
RRPD	Replica Request Priority Donation

RSB	Restricted Segment Boosting
TLB	Translation Lookaside Buffer
SOC	System On Chip
SRT	Soft Real Time
SWaP	Size, Weight, and Power
RNLP	Real-time Nested Locking Protocol
WCET	Worst-Case Execution Time
WSS	Working Set Size

CHAPTER 1: INTRODUCTION

Increasingly, computing systems are being used to sense and interact with the physical world. Examples of such systems include automotive and avionic systems, among many others. In many such systems, computations must be performed within precise time constraints, otherwise there may be catastrophic consequences. These systems are called *real-time systems*. Designers of real-time systems must leverage highly predictable algorithms that can be rigorously shown to realize such timing constraints.

To support the ever-increasingly complex real-time applications that our society demands (*e.g.*, autonomous vehicles), more computational power is needed. Multiprocessor and multicore technologies offer the promise of such computational power. Indeed, multiprocessor technologies have enabled huge advances in general-purpose and throughput-oriented application domains where strict timing constraints are absent. In contrast, in real-time systems where tight timing constraints exist and must be rigorously shown to be satisfied, there are a number of challenges associated with leveraging multiprocessor platforms. Many of these challenges chiefly relate to the sharing of resources. While there exists a number of ways to safely share such resources, this dissertation provides many new methods to support such sharing in ways that allow for improved platform utilization.

We begin this chapter with a brief introduction to real-time systems and synchronization. We then describe common multiprocessor architectures and highlight the challenges they impose in the context of real-time applications. We then present the thesis of this dissertation as well as its contributions. Finally, we outline the remainder of the dissertation.

1.1 Real-Time Systems

The term "real-time" takes on different meanings in different application domains. It is therefore necessary to clarify the definition of "real-time" used in this dissertation. We assume a real-time system to be one in which associated with each *job* (*i.e.*, program invocation) is a *deadline*, or time by which that job should complete. Deadlines can be either *hard*, in which case the job must complete before its deadline, or *soft*, in which case some deadline misses are acceptable. In a hard real-time system, in which all deadlines

are hard, if a job may miss a deadline, the system must be characterized as *incorrect*. There are numerous soft-real-time correctness conditions. Erickson (2014) gave a complete review of soft-real-time correctness conditions in his dissertation.

These types of real-time requirements are common in safety-critical applications, in which the violation of timing constraints may have catastrophic consequences. For example, the airbag controller in a car must detect a crash and deploy the airbag within a predefined time interval so as to ensure the airbag protects the passenger. Missing the deadline in this application can have catastrophic consequences including loss of life, and therefore assurances must be built into the design process to ensure such deadlines will never be missed.

In contrast, in many alternative application domains, a system may be considered "real-time" if it permits a seamless user experience. For example, in computer graphics, an application may be considered "real-time" if frames are rendered at approximately 30 frames per second (FPS), resulting in a smooth, realistic user experience. In such applications, there are often no performance assurances (*e.g.*, 30FPS), but in extensive testing, the applications perform well enough to be useful in practice. McKenney (2009) coined the term "real fast" to distinguish such applications from real-time applications with strict timing requirements. The phrase "real time" is also sometimes casually used to refer to information or behaviors that are "live" or very recent. For example, the website `http://trendsmap.com` provides "real-time local Twitter trends" showing the most common twitter words or hashtags used in different regions of the world recently.

In the remainder of this dissertation, we exclusively use the term real-time to denote systems in which there exist strict timing requirements. In particular, as will be formalized later in Chapter 2, we consider a real-time workload or *task set* consisting of a set of computational *tasks*. Each task is composed of a sequence of *jobs*, each of which is a piece of work or computation, associated with which is a deadline by which the job must complete. Jobs are *released* or made available to execute at a predictable rate or time interval. The jobs are *scheduled* upon a processor according to a real-time scheduling algorithm. For example, *earliest deadline first* (*EDF*) is a well studied real-time scheduling algorithm in which the job with the earliest deadline is always scheduled. A task set, or collection of tasks is deemed *schedulable* when it can be rigorously and mathematically proven that all timing constraints will be satisfied, and *unschedulable* otherwise.

In order to provide such mathematical proof that all timing requirements will be satisfied, there must be *schedulability analysis* corresponding to the real-time scheduling algorithm. Schedulability analysis takes as input a mathematical model of the task system and determines for a given scheduling algorithm whether all jobs will complete by their deadlines. Non-optimal scheduling algorithms, or conservative or pessimistic

schedulability analysis can lead to *utilization loss*, in which some task systems that do not fully utilize the system resources may be deemed unschedulable. Advances in real-time systems seek to minimize utilization loss, so as to better utilize system resources. In turn, this allows for real-time systems to be designed and developed within tighter size, weight, and power (SWaP) constraints.

1.2 Multicore Architectures

Another means of reducing the SWAP of a system is through advances in computer architectures. One of the most important and influential recent advances in computer architecture is the advent of the multicore platform. In a multicore platform, two or more processors are contained within a single integrated circuit. Processor architectures have shifted away from single-core to multicore designs as a result of diminishing returns in instruction-level parallelism, as well as the increasing power demands of higher clock frequencies (Hennessy and Patterson, 2007). Multicore architectures have trickled down from high-performance systems all the way to embedded and real-time systems. For example, many modern commodity phones include as many as eight cores, and the European Space Agency has developed the Next-Generation Multiprocessor (NGMP), a quad-core processor designed for real-time computations, specifically in space-based applications (Fernàndez et al., 2012).

While multicore processors offer great performance potential, they raise a number of significant challenges for the design of real-time systems. Multicore architectures are often designed with a number of architectural features shared among cores. As seen in Figure 1.1, a multicore platform may consist of multiple processing cores, each with core-local instruction and data caches, L1-I and L1-D, respectively. In this example, all cores share the L2 cache, which is also the *last-level cache* (*LLC*) in this example. These caches store data that resides in *dynamic random-access memory* (*DRAM*) banks, which may also be accessed from any core. There exist other hardware features not depicted in Figure 1.1 that are also shared among multiple cores. Examples of such features include buses, the memory controller, and registers used within the LLC itself (Valsan et al., 2016), among others. Interactions among cores through these shared hardware features can significantly affect the timing behavior of computations on multicore platforms. For example, if a core were to execute a memory-intensive code segment in isolation, *i.e.*, while all other cores are idle, it will enjoy the full bandwidth of the memory bus and memory controller. If instead all cores were executing memory-intensive code segments causing contention on the memory controller and bus, each core would

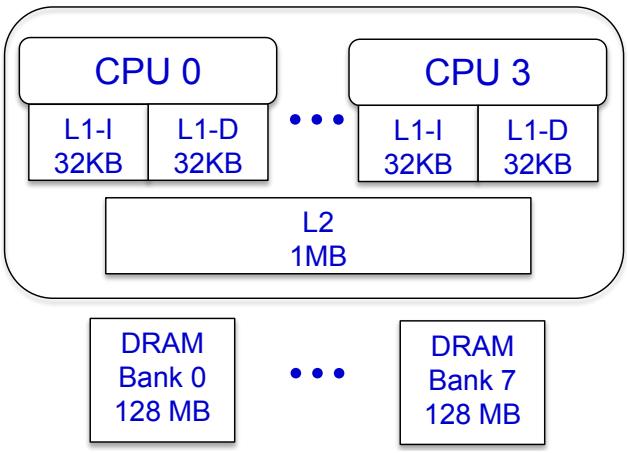


Figure 1.1: Common architecture for multicore processors. Note that the LLC and memory banks are shared among all cores.

receive only a fraction of the available memory bandwidth. Such contention may seriously affect the timing behavior of real-time tasks by introducing significant *analysis pessimism*. This analysis pessimism results in reduced platform utilization as a result of the inability to tightly characterize and analyze the behavior of the system.

This issue of cross-core interference has resulted in what we have coined the "one-out-of-m" problem: when validating real-time constraints on an m-core platform, excessive analysis pessimism can effectively negate the processing capacity of the additional $m-1$ cores so that only "one core's worth" of capacity is available (Erickson et al., 2015; Kim et al., 2016). In fact, due precisely to such effects, the current best practice in safety-critical domains such as avionics is to turn off all but one core in a multicore processor. To address this issue, the U.S. Federal Aviation Administration (FAA) Certification Authorities Software Team (CAST) (2014) recently released a position paper (known in the community as CAST-32) on multicore processors that identifies sources of shared-hardware interference, and suggests that the effects of such interference be identified, analyzed, and certifiably mitigated. Several of the core objectives in my dissertation address these concerns through new models for mitigating such shared-hardware interference.

1.3 Synchronization

In much work on real-time schedulability analysis, task systems are assumed to be simple *independent systems*, *i.e.*, systems in which there do not exist any dependencies among tasks. This assumption significantly eases work on schedulability analysis. However, in practice dependencies among real-time tasks are common.

Indeed, with the advent of the multicore era, in order to leverage the computing capacity of additional cores, workloads must be parallelized to execute on multiple cores concurrently. All but the simplest parallel algorithms require shared state or communication among concurrently executing threads. In order to realize the benefits of the increased computational capacity of modern multicore platforms in the context of real-time systems, synchronization algorithms are required that allow for run-time parallelism that can be reflected in improved analytical worst-case behaviors. In turn, bounds on the worst-case behavior are then incorporated into schedulability analysis. Therefore, improving the worst-case synchronization behavior affords better overall platform utilization.

Many synchronization algorithms have been developed for multiprocessor real-time systems, *e.g.*, (Block et al., 2007; Brandenburg, 2014; Brandenburg and Anderson, 2014; Gai et al., 2001; Rajkumar, 1991). However, one serious limitation of most previous multiprocessor synchronization algorithms is their lack of support for *fine-grained locking*, or *lock nesting*, for resources shared among multiple processors. Nested requests, which are commonly employed in practice to improve run-time parallelism, 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 (Block et al., 2007; Rajkumar, 1990). In contrast, under fine-grained locking, different resources are locked individually, which enables concurrent accesses of separate resources. Clearly, fine-grained locking is desirable in practice, and should result in improved worst-case performance.

In addition to shared memory objects, there are also less overt resources within the hardware platform as discussed above, that if explicitly managed, can result in better platform utilization, thereby improving SWAP. Shared hardware resources, such as I/O devices (*e.g.*, graphics processing units or digital signal processors), buses, or caches, are all resources that may be shared among tasks either explicitly or implicitly. If left unmanaged, tasks can interfere with one another through these resources, which can negatively affect the worst-case timing behavior of the tasks, due to the often unpredictable nature of such interactions. If instead access to such resources are expressly managed, then synchronization costs can be traded for improved predictability.

1.4 Thesis Statement

The thesis supported by this dissertation is broadly motivated by the need to solve the "one-out-of-m" problem. While there are many avenues to address this important problem, in this dissertation we focus on the issues that arise due to resource sharing among cores. We thus consider resources that are shared explicitly via semaphores, as well as shared hardware resources that are often shared implicitly, *i.e.*, contention for the resource may be managed (suboptimally) in hardware. Towards this end, we present the following thesis statement:

> *"Dependencies among tasks in real-time systems through shared resources, both memory objects, as well as shared hardware resources, can be managed through synchronization protocols. Such protocols can be designed to exploit the inherent sharing constraints of the managed resources in order to achieve improved resource utilization."*

To support this thesis, we have designed and developed new synchronization algorithms, formalized new sharing constraints, and evaluated the proposed algorithms with respect to real-time schedulability.

1.5 Contributions

In the remainder of this chapter, we overview the contributions that support this thesis, and overview the organization of the remainder of the dissertation.

1.5.1 RNLP Family of Asymptotically Optimal Fine-Grained Multiprocessor Locking Protocols

In Chapter 3, we present the first *fine-grained* multiprocessor real-time locking protocol, the *Real-Time Nested Locking Protocol* (*R/W RNLP*). This solves a glaring open problem raised by previous work on multiprocessor real-time synchronization, that one job could not hold two locks simultaneously. As a result, only *coarse-grained* locking was allowed, in which resources that may need to be accessed concurrently were grouped, and the group was treated as a single lockable entity. In contrast, with fine-grained locking as provided by the RNLP, resources may be individually locked, reducing blocking and improving platform utilization.

The RNLP has a configurable, "plug-and-play" architecture that allows it to be configured in different ways depending upon the platform and analysis assumptions being used. For each such platform and analysis-

assumption combination, the RNLP can be configured to achieve asymptotically optimal worst-case blocking. All of the different RNLP configurations give rise to a family of mutual exclusion (mutex) locking protocols.

In Chapter 4, we present extensions of the RNLP that support alternative sharing constraints including reader/writer and k exclusion. These alternative sharing constraints allow for improved blocking in many cases, and can be used to improve overall platform utilization.

1.5.2 Synchronization Algorithms for Shared Hardware Resources

While the RNLP family of protocols and extensions thereof were primarily designed to be used to arbitrate access to shared memory objects, it has been used to in the context of shared hardware resources as well (Ward et al., 2013b). Through the application of the RNLP in this domain, we learned that shared hardware resources have inherently different synchronization requirements than shared memory objects. In Chapter 5, we present two new resource-sharing constraints (*e.g.*, mutex, k-exclusion, reader/writer), designed expressly for arbitrating access to shared hardware resources in an effort to improve the temporal predictability of tasks using those resources. The first such sharing constraint is call *premptive mutual exclusion*, and it is motivated the need to manage resources such as data buses. The second sharing constraint is called *half protected exclusion*, and it is motivated by the need to manage access to caches.

Additionally, we present the R^2DGLP, an optimal k-exclusion locking protocol that has been applied and proven highly useful in the context of predictably controlling access to multiple graphics processing units (GPUs) (Elliott, 2015; Elliott et al., 2013). Furthermore, the R^2DGLP can be used as a "plug-and-play" component in the RNLP to achieve improved blocking bounds in some cases.

1.5.3 Idleness Analysis

Traditionally, to incorporate the effects of synchronization algorithms into schedulability analysis, *blocking analysis* is conducted to determine the worst-case blocking behavior, and the results of blocking analysis are incorporated into schedulability analysis. In Chapter 5, we present an alternative analysis framework for incorporating the effects of synchronization into schedulability analysis. We call this analysis technique, *idleness analysis*. By directly analyzing idleness, we "flip the analysis." Instead of asking "how long can this request be blocked?" we instead ask "how much idleness can this request cause?"

Idleness analysis is particularly useful in the context of the aforementioned synchronization models developed for shared hardware resources. However, as we describe in Chapter 5, it can also be applied to

other synchronization algorithms. We explore empirically how idleness analysis compares with traditional blocking analysis, and show cases in which each approach is superior to the other.

1.5.4 Evaluation

In Chapters 4, and 5 we present results from schedulability studies were conducted to evaluate the improved platform utilization made possible through the aforementioned synchronization algorithms and associated analysis techniques. In this schedulability study, task systems are randomly generated based on a number of task-system parameters, and then evaluated for schedulability assuming a number of combinations of synchronization and scheduling algorithms. This is the *de facto* standard means of evaluating the proposed synchronization algorithms and analysis techniques.

The results of these schedulability studies reinforce that analytical conclusions drawn in previous chapters, and demonstrate the extent to which platform utilization can be improved through the techniques presented in this dissertation.

1.6 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we discuss several background topics relevant to the contents and contributions of this dissertation, including real-time scheduling and synchronization and multicore architectures. In Chapter 3, we present the Real-Time Nested Locking Protocol (RNLP), and in Chapter 4, we present extensions to the RNLP that support reader/writer and k-exclusion locking. In Chapter 5, we present synchronization models and associated algorithms that are designed to be used to more effectively arbitrate access to different classes of shared hardware resources. Finally, we conclude in Chapter 6.

CHAPTER 2: BACKGROUND AND PRIOR WORK

In this chapter, we present the requisite background material for understanding the contributions of this dissertation. We begin with real-time task models and schedulability. We then define models for shared resources, and cover related work on real-time synchronization. Finally, we present relevant features and functionality of shared caches, memory buses, and GPUs, which motivate the synchronization problems and solutions presented in Chapter 5.

2.1 Real-Time Scheduling

Before we can describe synchronization-related resource models, we must first formally define models for tasks. To begin, we present these task models assuming that all tasks are independent. Later, we extend such models to consider shared resources. Formally specifying the task and resource models is a necessary first step towards towards formal schedulability analysis.

2.1.1 Task Model

In this dissertation, we consider the well-studied sporadic task model (Mok, 1983). In this model, a task system Γ is composed of *n sporadic tasks* $\Gamma = \{\tau_1, \ldots, \tau_n\}$ that are scheduled on *m* processors.[1] Each task τ_i is composed of a (potentially infinite) sequence of jobs $J_{i,1}, J_{i,2}, \ldots$, which are released sequentially. For convenience of notation, we let J_i denote a job of τ_i, when the job index is inconsequential. Each task τ_i is characterized by a tuple $\tau_i = (e_i, d_i, p_i)$. The *worst-case execution time* $(WCET)$ of any job of τ_i is denoted e_i. A job $J_{i,j}$ is released (*arrives*) at time $a_{i,j}$, and successive jobs of τ_i are released with a minimum separation[2] of p_i, *i.e.*,

$$a_{i,j} - a_{i,j-1} \geq p_i. \tag{2.1}$$

[1] We use the terms "processor," "core," and "CPU." interchangeably.

[2] The sporadic task model is an extension of the *periodic task model* (Liu and Layland, 1973), which specifies an *exact* release separation.

Job $J_{i,j}$ completes (*finishes*) at time $f_{i,j}$. The *relative deadline* d_i of τ_i specifies the time after the job release $a_{i,j}$ by which $J_{i,j}$ must complete, called the *absolute deadline* of $J_{i,j}$ denoted $D_{i,j}$.

$$D_{i,j} \triangleq a_{i,j} + d_i \tag{2.2}$$

Task systems with $p_i = d_i$ for all tasks are called *implicit deadline*, and are sometimes denoted using a two-tuple $\tau_i = (e_i, p_i)$ notation for simplicity. Task systems with $d_i \leq p_i$ are called *constrained deadline*, and task systems containing tasks with $d_i > p_i$ are called *arbitrary deadline*.

There exists a *precedence constraint* between successive jobs in that $J_{i,j}$ cannot be scheduled before $J_{i,j-1}$ completes. The *response time* $r_{i,j}$ of a job $J_{i,j}$ is the time between its release and its completion.

$$r_{i,j} \triangleq f_{i,j} - a_{i,j} \tag{2.3}$$

The *utilization* of τ_i is $u_i = e_i/p_i$, and the total processor utilization is $U = \sum_{i=1}^{n} u_i$.

For hard real-time (HRT) schedulability, the response time of each job $J_{i,j}$ must satisfy $r_{i,j} \leq d_i$. However, for soft real-time (SRT) scheduling, some deadline misses are acceptable. In this dissertation, we adopt the notation for lateness and tardiness as defined by (Devi, 2006; Erickson, 2014). *Lateness* is defined by

$$l_{i,j} \triangleq f_{i,j} - D_{i,j}. \tag{2.4}$$

Lateness is negative for a job completing before its deadline. *Tardiness* is defined to be non-negative lateness.

$$x_{i,j} \triangleq \max(0, l_{i,j}) \tag{2.5}$$

A number of more expressive task models have been proposed and analyzed. Stigge and Yi (2015) presented a survey of many graph-based task models, and Burns and Davis (2016) maintain an excellent survey of mixed-criticality task models, scheduling, and analysis. However, these models are outside the scope of this work, and are orthogonal to the issues that arise due to synchronization. In this dissertation, we exclusively consider the sporadic task model, and extensions thereof that support shared resources.

2.1.2 Scheduling

There exist a number of different policies for scheduling the aforementioned tasks on m processors. We begin our review of such schemes by considering the single-processor case, in which $m = 1$. A scheduling policy can be viewed as a prioritization scheme, and the highest-priority job is scheduled at any given point in time. One such well-studied prioritization scheme is *fixed priority* (*FP*), in which each task is assigned a static priority. RM scheduling is a FP scheduling policy in which tasks are prioritized in decreasing order by period, *i.e.*, the shortest-period job is assigned the highest priority. Another well-studied prioritization scheme is *earliest deadline first* (*EDF*), in which each job is prioritized by its deadline, with the job with the earliest deadline having the highest priority. EDF is an example of a *dynamic-priority* scheduling algorithm, as different jobs of the same task may have different priorities. In this dissertation, we consider *job-level fixed-priority* (*JLFP*) schedulers, in which the priority of each job is fixed. Both EDF and FP scheduling are examples of JLFP schedulers.

Example 2.1. Consider the example task system depicted in Figure 2.1. In this task system, $\tau_1 = (1,3)$, $\tau_2 = (2,6)$, and $\tau_3 = (4,12)$. Under FP scheduling, assuming lower-indexed tasks are higher priority, τ_1 is scheduled at time 0. When it completes at time $t = 1$, τ_2 executes until it completes at time $t = 3$. Also at $t = 3$, the next job of τ_1 is released, which has the highest priority and therefore begins running. At time $t = 4$, τ_1 completes its second job. Because τ_2 has no ready job, τ_3 begins running. At time $t = 6$, another job of τ_1 is released, and it begins running. This is an example of a *preemption*, in which τ_3 was "paused" to allow the higher-priority task to execute. The schedule continues in a similar fashion until time $t = 10$, when τ_3 resumes execution, and runs until its completion at time $t = 12$.

Note that if jobs were instead prioritized by deadlines with ties broken in favor of the lower-indexed task, the same schedule would result in this particular example task system. In general, EDF and FP do not produce the same schedule. ◊

Preemptivity. Non-preemptive schedulers disallow preemptions such as that at time $t = 6$ in Example 2.1. For example, non-preemptive FP scheduling disallows all preemptions. Non-preemptive scheduling eases timing analysis, or the derivation of the e_i terms (for reasons that will be made more clear in Section 2.3), but can lead to *non-preemptive blocking*, in which a high-priority job is blocked while a low-priority job executes instead. In Example 2.1, under non-preemptive FP, τ_3 would run until completion at time $t = 8$ before $J_{1,3}$ would be allowed to execute. During $t \in [6,8)$, $J_{1,3}$ would be non-preemptively blocked. Non-

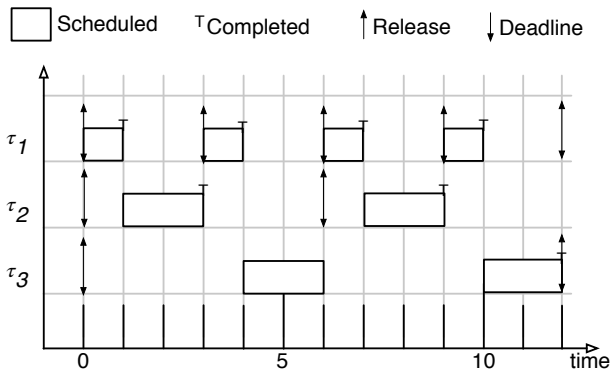


Figure 2.1: This schedule is possible for the example task system described in Example 2.1. In this particular example, EDF and FP produce the same scheduling decisions, assuming tasks are indexed in decreasing priority order.

preemptive blocking can lead to poor resource utilization, and therefore, unless otherwise specified, we focus on preemptive schedulers in this dissertation.

2.1.3 Schedulability

To use a given scheduling algorithm in a real-time system, a *schedulability test* must be applied to evaluate whether all jobs will provably satisfy their timing constraints. If a schedulability test returns true (false), we say the task system is *schedulable* (*unschedulable*). In our discussion of schedulability, we begin by describing the uniprocessor case before addressing the complexities that arise when adding additional processors.

Clearly, a task system that overutilizes the processor, *i.e.*, has a utilization exceeding one, cannot meet its timing constraints as jobs will complete progressively later and later. This is an example of a task system that is *infeasible*, or impossible to schedule such that all timing constraints are satisfied. If a scheduling algorithm exists that can schedule a task system while satisfying all timing constraints, then that task system is *feasible*. In the seminal work of Liu and Layland (1973), it was shown that under EDF scheduling an implicit-deadline task system is HRT schedulable if and only if

$$U \leq 1. \tag{2.6}$$

EDF has been shown to be be *optimal* on a uniprocessor platform, as it correctly schedules any feasible task system.

A schedulability test may be either *exact*, or *sufficient*. A sufficient schedulability test is one in which the task system *may* be correctly scheduled, but due to pessimistic or imprecise analysis, the schedulability test cannot show the task system to be schedulable. If a task system can be correctly scheduled, then an exact schedulability test must return that the task system is schedulable. Therefore, the aforementioned EDF schedulability test in (2.6) is an exact schedulability test. Liu and Layland (1973) presented a sufficient schedulability condition for RM scheduling of implicit-deadline task systems:

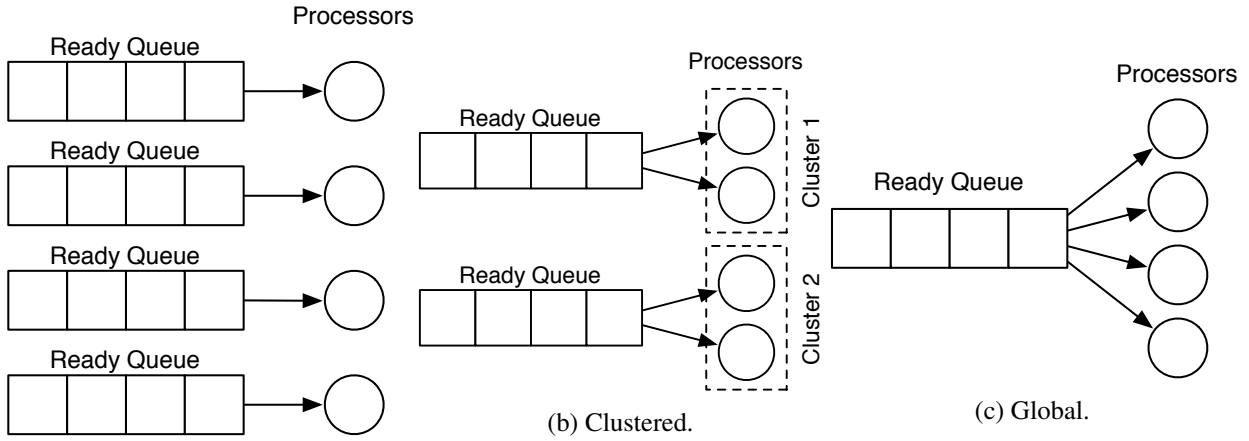
$$U \leq n(2^{\frac{1}{n}} - 1). \tag{2.7}$$

A task system with utilization $n(2^{\frac{1}{n}} - 1) < U \leq 1$ is feasible to schedule, and indeed schedulable with EDF, but is not deemed schedulable by this schedulability condition. We call this *utilization loss*, as it represents processor utilization that cannot be used, at least analytically, either due to pessimistic schedulability analysis, or a suboptimal scheduling algorithm. Work on scheduling and schedulability tests, a central theme of work in real-time system research, strives to minimize utilization loss. This is a particularly active area of research for multiprocessor systems, where additional processors add significant complexity to scheduling and schedulability analysis.

Response-time analysis. The schedulability conditions in (2.6) and (2.7) are both *utilization-based schedulability tests*, in that they only consider the total task-system utilization in determining schedulability. Utilization-based schedulability tests are very easy to check, usually in polynomial time, but are often subject to greater utilization loss than other schedulability tests. Schedulability conditions based on *response-time analysis* (*RTA*) often reduce utilization loss at the expense of more complex analysis. RTA is used to determine an upper bound on the response time of any job of each task. Such bounds are compared to the relative deadline of the corresponding task to determine if the task system is schedulable. Joseph and Pandya (1986) presented the first RTA for FP scheduling.

Theorem 2.1. *(Joseph and Pandya, 1986). Let $\Gamma = \{\tau_1, \ldots, \tau_n\}$ denote a set of constrained-deadline sporadic tasks indexed in order of decreasing priority. On a uniprocessor, under FP scheduling, the response time r_i of task τ_i is bounded by the smallest $R_i \geq e_i$, that satisfies the following equation:*

$$R_i = e_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{p_j} \right\rceil e_j. \tag{2.8}$$



(a) Partitioned. (b) Clustered. (c) Global.

Figure 2.2: Visual depictions of partitioned, clustered, and global scheduling.

This recurrence can be iteratively computed starting with $R_i = e_i$ and re-evaluating until both the left and right sides of the equation converge. This convergence is guaranteed if $U \leq 1$ (Joseph and Pandya, 1986). Furthermore, this convergence will occur in a pseudo-polynomial number of operations. RTA forms the basis of many schedulability tests for a number of different schedulers and extended task models.

Multiprocessor schedulability. In the multiprocessor case, in which $m > 1$, there are additional factors to consider in the development of scheduling algorithms. Under *global scheduling*, the m highest-priority jobs are scheduled at any given time. Alternatively, under *partitioned scheduling*, tasks are statically assigned to processors, and each processor is scheduled according to a uniprocessor scheduling algorithm. *Clustered scheduling* is a hybrid of these two approaches in which tasks are assigned to a specific group or cluster of processors, and global scheduling is applied within each cluster. These three scheduling approaches are visually depicted in Figure 2.2. Notationally, we let c be the size of each cluster, and therefore there are $\frac{m}{c}$ clusters. With this notation, global and partitioned scheduling are special cases of clustered scheduling in which $c = m$ and $c = 1$, respectively. These different scheduling alternatives give rise to different challenges with respect to synchronization, as we will discuss in more detail later.

Under partitioned scheduling, the aforementioned uniprocessor scheduling and schedulability conditions can be applied on each processor. While optimal uniprocessor scheduling algorithms exist, this does not imply that partitioned scheduling is optimal on a multiprocessor. When tasks are partitioned onto the processors there is the possibility of bin-packing-related utilization loss.

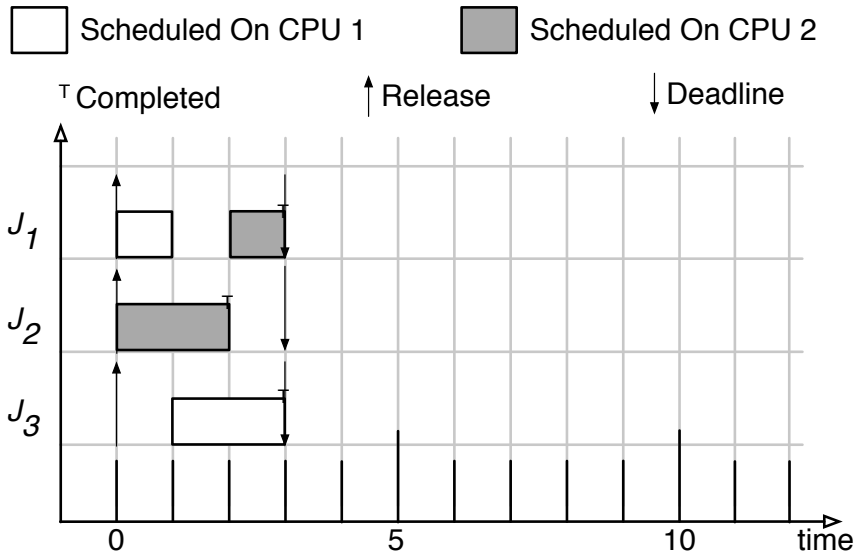


Figure 2.3: Example schedule demonstrating that the example task system in Example 2.2 is feasible to schedule on two processors. Note that J_1 migrates from CPU 1 to CPU 2 in this schedule.

Example 2.2. Consider a task system with three tasks, $\Gamma = \{(2,3),(2,3),(2,3)\}$. The total utilization of this task system is two, but there is no partitioning of tasks to two processors that does not overutilize at least one processor—every partitioning has one processor with at least two tasks; that processor is thus overutilized with a utilization of $\frac{4}{3}$. This task system is therefore infeasible under partitioned scheduling. However, as depicted in Figure 2.3, the task system is feasible to schedule on two processors. ◊

Global scheduling. An alternative to partitioned scheduling and the aforementioned bin-packing-related utilization loss is global scheduling. However, global scheduling requires new schedulers and schedulability analysis that considers the task system being scheduled on multiple processors. (Global schedulability analysis is also applied to each cluster with $c > 1$, similarly to how uniprocessor schedulability analysis is applied to each partition with $c = 1$.) Given the observation of bin-packing-related utilization loss in Example 2.2, is there a global scheduling algorithm that is optimal, similar to EDF on a uniprocessor?

As it turns out, the answer to this question is quite nuanced. Hong and Leung (1988) and Dertouzos and Mok (1989) independently showed that the optimal online scheduling of independent jobs (*i.e.*, a system with no task abstraction) is impossible. Later, Fisher et al. (2010) showed that the optimal online scheduling of sporadic task systems is impossible by showing a feasible system that no online scheduler can correctly schedule without *clairvoyance*, or knowledge of future events. However, with minor restrictions on the task systems, optimal algorithms have been developed. These include *Proportionate fair* (*Pfair*) (Baruah et al.,

1996) algorithms such as PD^2 (Srinivasan and Anderson, 2002), and RUN (Regnier et al., 2011), which are optimal for implicit-deadline task systems. One common trait of all optimal multiprocessor schedulers is that they are *job-level dynamic priority* (*JLDP*) schedulers, and therefore the priority of individual jobs may change during execution. JLDP algorithms often incur more preemptions than JLFP algorithms, and incur higher runtime overheads due to the need to change job priorities (Brandenburg, 2011). More efficient optimal scheduling algorithms are an active area of research, and we refer the reader to Nellisen (2012) for an excellent presentation of several such algorithms, as well as a discussion of optimal multiprocessor scheduling.

In addition to the increased overheads associated with JLDP scheduling, priority changes also add complexity to the design of synchronization algorithms. While there do exist techniques for synchronization in some optimal schedulers (Holman and Anderson, 2006), the vast majority of work on real-time synchronization focuses on JLFP schedulers, which are more efficient in practice. For these reasons, we also focus exclusively on JLFP schedulers in the remainder of this dissertation.

Global JLFP schedulers. Suboptimal schedulers inherently suffer from utilization loss. However, the runtime efficiency of the scheduling algorithm itself, as well as the accuracy of the associated schedulability analysis can make suboptimal schedulers preferable to optimal schedulers in practice. This is indeed the case for two well-studied global scheduling algorithms, *global fixed priority* (*G-FP*) and *global earliest-deadline first* (*G-EDF*). Both G-FP and G-EDF generalize their uniprocessor counterparts by scheduling the m highest-priority jobs at any given point in time.

Example 2.3. Consider again the example task system described in Example 2.1. If instead that task system were scheduled on two processors, the resulting schedule would be that depicted in Figure 2.4. At time $t = 0$, both $J_{1,1}$ and $J_{2,1}$ are scheduled. At time $t = 1$, $J_{1,1}$ completes, allowing $J_{3,1}$ to begin. At time $t = 3$ when $J_{1,2}$ is released, it need not preempt another job, as in Example 2.1, but instead may be scheduled on the idle processor. The schedule proceeds similarly. ◊

As compared to uniprocessor schedulability tests, it is much more difficult to derive multiprocessor schedulability conditions. Intuitively, the key reason for the additional complexity for multiprocessor analysis is *idleness*. On a uniprocessor platform, the entire platform (one processor) is either busy running a single job, or *idle*. On a multiprocessor platform, one processor may be busy while another is idle, as seen at time $t \in [2,3)$ in Figure 2.4. This issue, among others, has given rise to a number of different approaches to

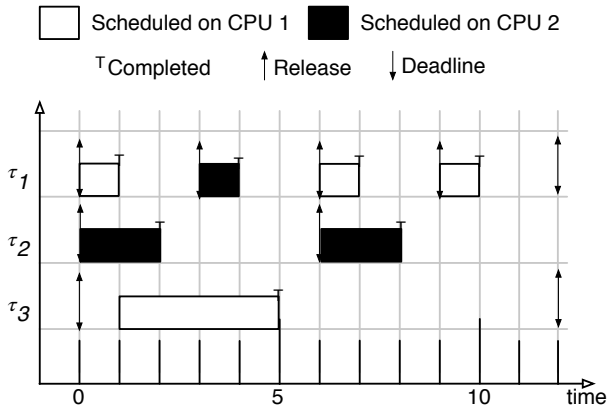


Figure 2.4: Example G-EDF schedule. This schedule depicts the example task system described in Example 2.3. In this particular example, G-EDF and G-FP produce the same scheduling decisions, assuming tasks are indexed in decreasing priority order.

multiprocessor schedulability conditions. Bertogna and Baruah (2011) provide a summary of seven different incomparable G-EDF schedulability conditions. Many of these conditions are implemented in the publicly available library SchedCAT (SchedCAT, 2016), which is used in the evaluation in subsequent chapters. Specifics concerning the derivation of these schedulability conditions are not essential background to the contributions of this dissertation, so we refer the interested reader to (Baruah et al., 2015; Bertogna and Baruah, 2011; Brandenburg, 2011) for further discussion.

Soft real-time schedulability. There exist many applications in which some deadline misses are acceptable, particularly if allowing for such deadline misses reduces utilization loss. Many multimedia applications fall into this domain—some deadline misses may degrade the user experience and should be avoided, but do not have catastrophic consequences.

Devi and Anderson (2005) showed that under the bounded-deadlines-tardiness definition of SRT scheduling, G-EDF has no utilization loss and is therefore optimal. Therefore, under SRT timing constraints, the G-EDF schedulability condition is

$$U \leq m. \tag{2.9}$$

Furthermore, Devi and Anderson (2005) showed that, assuming 2.9 holds, the deadlines tardiness x_i of any job is upper bounded by

$$x_i = e_i + \frac{E_{m-1} - e_{min}}{m - U_{m-1}}, \tag{2.10}$$

where E_{m-1} and U_{m-1} denote the sum of the largest $m-1$ execution times and per-task utilizations, respectively, and e_{min} denotes the smallest e_i. In subsequent work, tardiness bounds were improved (Erickson, 2014), and computed on a per-task basis. Furthermore, it has been shown that if tasks are prioritized by a *priority point* or "pseudo deadline" in a fashion similar to G-EDF, then tardiness bounds can be further reduced. Erickson et al. (2014) showed how these priority points can be set via a linear program (LP). This LP framework allows the LP solver to optimize the priority-point settings subject to different optimization objectives (*e.g.*, maximum per-task tardiness), and/or tardiness-bound constraints (*e.g.*, $x_i \leq 10\,\text{ms}$). In the context of this dissertation, however, we consider only SRT schedulability, and assume that the tardiness bounds derived via the above methods are sufficient.

2.2 Real-Time Synchronization

In the previous discussion, and indeed much of the published scheduling-theory literature, tasks are assumed to be *independent*, in that they do not share resources or otherwise have any dependencies among them. However, in practice many dependencies exist, and the aforementioned models and analysis techniques must be extended to support such dependencies. In this dissertation, we exclusively consider dependencies that arise due to resource sharing.

2.2.1 Resource Model

We assume that the n tasks share n_r non-processor shared resources, $\ell_1, \ldots, \ell_{n_r}$. A job J_i that requires access to ℓ_q must issue a *resource request*, denoted $\mathcal{R}_i$, to a *locking protocol* before it is granted access to ℓ_q to execute its *critical section*. Each resource is subject to a *resource-sharing constraint*, which defines the semantics of how the resource can be safely shared. For example, most locking protocols implement *mutual exclusion* (*mutex*), in which at most one critical section is allowed to run at a time, and must complete before the next critical section is allowed to execute. To satisfy the resource-sharing constraints, requests may be *blocked*, or forced to wait, by the locking protocol. A request $\mathcal{R}_i$ is said to be *satisfied* when J_i acquires the requested resource. 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*. A job can either *spin* (*i.e.*, busy-wait) or suspend while waiting for one of its

requests to be satisfied. A job that has been released but has not yet completed is *pending*. A job that can be scheduled is *ready*; thus, a job that is suspended and waiting for a shared resource is pending but not ready.

Granularity of locking. A resource is a logical abstraction of a single lockable entity. This entity may itself be composed of a number of other elements or abstractions. For example, a list data structure may be a resource that is treated as a single lockable entity, even though it consists of a number of list elements. If resources are logical abstractions of large or complex objects or data structures, we say that the synchronization is *coarse-grained*. In contrast, *fine-grained* locking allows for individual elements within a larger data structure to be accessed individually. Perhaps the best example of coarse-grained locking is older versions of the Linux kernel (*e.g.*, versions 2.0–2.2), which employed a *big kernel lock*, with the functions `lock_kernel()` and `unlock_kernel()`, which guard access to most of the linux-kernel data structures as a single resource. The granularity of locking in this example is very coarse. Over time the kernel was modified to use fine-granularity locking (notably in versions between 2.4 and 2.5), and eventually the big kernel lock was completely removed (version 2.6.37).

Coarse-grained locking is much easier to use as a developer, as developers need not concern themselves with possible states that may arise when different tasks access different resources concurrently. However, coarse-grained locking serializes potentially non-conflicting requests, or requests that do not operate on the same resources. Therefore, fine-grained locking can reduce the blocking that tasks experience while waiting for shared resources. To support fine-grained locking, it is necessary to *nest* lock requests, or allow a resource-holding job to issue a request for another resource within its critical section. While in practice the granularity of locking is application specific, for the purpose of this dissertation, we say that a locking protocol that supports a single job holding multiple resources concurrently supports fine-grained locking. Protocols that do not support fine-grained locking are coarse-grained locking protocols.

To formalize the nested-locking terminology, if J_i holds no resources when it makes a request, then the request is said to be an *outermost request*. 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*. 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 $L_{i,k}$. A job J_i makes *progress* if a job that holds a resource for which J_i is waiting is scheduled and executing its critical section.

Locking protocols can cause tasks to wait or *block* while others execute critical sections. A *priority inversion* occurs if a high-priority job is blocked waiting for a low-priority job to complete its critical section. Such blocking is an example of *priority-inversion blocking* (*pi-blocking*). This can affect the scheduling decisions that are made, and therefore must be incorporated into schedulability analysis. The first step in this process, is to formally analyze priority-inversion blocking.

2.2.2 Blocking Analysis

Brandenburg and Anderson (Brandenburg and Anderson, 2014) formalized two classes of techniques to incorporate the effects of priority-inversion blocking into schedulability analysis for suspension-based locking protocols: *suspension-oblivious* (*s-oblivious*) analysis, in which suspensions due to pi-blocking are modeled as computation, and *suspension-aware* (*s-aware*) *analysis*, in which suspensions due to pi-blocking are incorporated directly into schedulability analysis. These techniques rely upon different definitions of pi-blocking given next, which are visually compared in Figure 2.5.

Definition 2.1. 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 m higher-priority jobs are **ready**.

Definition 2.2. 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 m higher-priority jobs are **pending**.

For spin-based locking protocols, there is a third type of pi-blocking.

Definition 2.3. A job J_i incurs *spin-based blocking* (*s-blocking*) at time t if J_i is spinning (and thus scheduled) waiting for a resource.

Note that spin-based blocking is computation, as the task actually occupies a processor while spinning. In comparison, both s-oblivious and s-aware pi-blocking bound times in which the task is suspended and therefore not executing.

Example 2.4. Consider the example EDF schedule depicted in Figure 2.5. At time $t = 1$, three tasks are released and task τ_1 and τ_2 begin executing. At time $t = 2$, τ_2 issues a request that is satisfied, and task τ_1 also issues a request that is blocked by τ_2. τ_1 suspends to wait until the resource is available, and relinquishes its processor to τ_3. At $t = 3$, τ_3 also issues a request, which is blocked by τ_2. When τ_2 completes its critical section at time $t = 5$, τ_1 acquires the resource and executes its critical section until $t = 7$ when it completes and τ_3 executes its critical section.

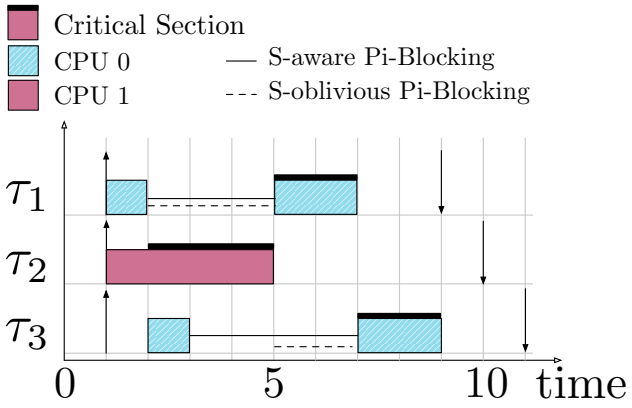


Figure 2.5: Illustration adapted from (Brandenburg and Anderson, 2010a) of the difference between s-oblivious and s-aware analysis. G-EDF scheduling is assumed.

Now let us consider the blocking that occurs in this schedule. First, consider the blocking by τ_1 during $t \in [2,5)$. τ_1 is the highest priority job and therefore there are no higher-priority pending or ready jobs, so by Definitions 2.1 and 2.2, τ_1 is both s-aware and s-oblivious pi-blocked during this interval.

Next consider the blocking by τ_3 during $t \in [3,5)$. τ_1 is pending but not ready, as it is suspended, and therefore only one higher-priority task, τ_2 is ready. By Definition 2.1, τ_3 is s-aware pi-blocked during this interval. However, both τ_1 and τ_2 are pending during $t \in [3,5)$, and therefore by Definition 2.2, τ_3 is not s-oblivious pi-blocked. After τ_2 completes at time $t = 5$, τ_3 becomes both s-aware and s-oblivious pi-blocked.

$\Diamond$

An intuitive way to think about s-oblivious vs. s-aware pi-blocking is to consider the suspensions of higher-priority jobs as if they occupied a processor. If a task τ_i is blocked but all m processors are busy, either with actual computation, or the suspensions of higher-priority jobs, which we are thinking of as occupying a processor, then τ_i is not s-oblivious pi-blocked. This observation is reflected in the following corollary to Definition 2.2

Corollary 2.1. *A job J_i may only be s-oblivious pi-blocked if it is among the top m highest-priority pending jobs.*

This corollary in particular informs the design of many locking protocols, which we will discuss later in this section.

Causes of pi-blocking. Pi-blocking can occur in one of two scenarios. First, a job that has issued a request and is blocked waiting to acquire a resource is said to experience *request blocking*. Second, many locking

protocols employ a *progress mechanism* such as *priority inheritance* or *priority boosting* (described in more detail later) to ensure that a resource-holding job is scheduled. These progress mechanisms work by increasing the priority of a task from its original or *base priority*, to a higher *effective priority*. For example, if a low-priority resource-holding job is priority boosted (or scheduled at the highest priority) to ensure progress, then a high-priority non-resource-accessing job may be pi-blocked, as it is pending but not scheduled. This gives rise to *progress-mechanism-related* (*PMR*) *pi-blocking*,[3] as a lower-priority job can execute while a higher-priority job is forced to wait, even if it has not issued a resource request. This second source of blocking can introduce a great deal of analysis pessimism, as it affects all tasks, regardless of whether they access shared resources or not. Brandenburg (2013a) defined a locking protocol with no progress-mechanism-related pi-blocking to be *independence preserving*, because independent tasks are not affected by the progress mechanism of the dependent, resource-using tasks.

Incorporating blocking into schedulability analysis. S-oblivious and s-aware pi-blocking bounds are used in different schedulability tests. S-oblivious pi-blocking bounds can be incorporated into schedulability tests that do not explicitly model suspensions by inflating the execution time of each task by its s-oblivious pi-blocking bound. Analytically treating suspensions as computation is safe as the corresponding schedule without suspensions will have response-time bounds at least that of the schedule with suspensions. This s-oblivious approach allows for blocking bounds to be easily applied in the context of many schedulability tests developed for independent systems, discussed in the previous section.

Alternatively, schedulability analysis may be *suspension-aware*. In such s-aware schedulability tests, the maximum suspension length is incorporated into the task model, and the effects of those suspensions are dealt with explicitly in the schedulability test. Such tests can be used to account for the effects of blocking by treating s-aware blocking bounds as the suspension lengths. For example, Liu and Anderson (2013) extended the schedulability test of Baruah (2007) to consider tasks that suspend. Suspensions are notoriously difficult to tightly analyze, and indeed such analysis is NP-complete in the strong sense (Ridouard et al., 2004). Consequently, there exist fewer s-aware schedulability tests, especially for multiprocessor systems. Furthermore, many studies have shown s-oblivious blocking analysis to be competitive, if not preferable to, s-aware analysis in many cases (Brandenburg, 2011, 2014; Ward, 2015). For a more complete discussion

[3]Under some locking protocols and analysis assumptions, PMR pi-blocking can only occur at job release, and in such cases is sometimes referred to as *release blocking*.

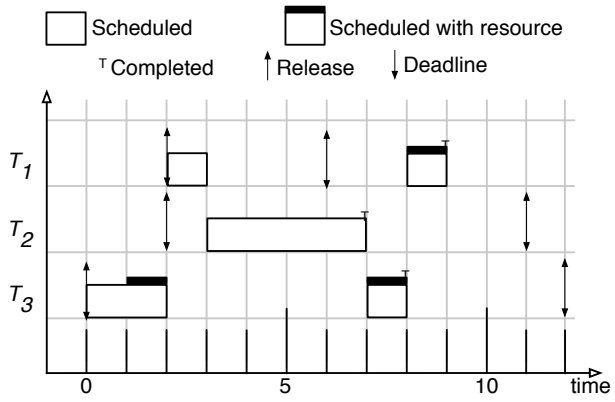


Figure 2.6: Example of an unbounded priority inversion, described in Example 2.5.

of suspensions and suspension-aware schedulability analysis, we refer the interested reader to (Chen et al., 2016) and (Liu, 2013).

2.2.3 Prior Real-Time Synchronization Algorithms

Next we review prior real-time synchronization algorithms. As in previous discussions, we start again with results for uniprocessor platforms, before we review multiprocessor algorithms.

2.2.3.1 Priority Inheritance Protocol (PIP) (Sha et al., 1990)

To illustrate the key properties of the PIP, we first illustrate the challenges that were faced in the design of the first real-time locking protocols, which motivate the design of the PIP.

Example 2.5. Consider the example uniprocessor EDF schedule depicted in Figure 2.6. The lowest-priority job J_3 locks a resource at time $t = 1$, and is then preempted at time $t = 2$ by J_1. At time $t = 3$, J_1 issues a request for the locked resource, and is blocked. J_2 as the highest-priority ready job is then scheduled until its completion at time $t = 7$. Only then can J_3 resume its critical section. J_1 can only resume after J_3 completes its critical section at time $t = 8$.

Note that J_1 is blocked for the entire duration of J_2. This is an example of *unbounded priority inversion*, as any medium-priority job, even ones that do not access shared resources, can delay the resource acquisition of the higher-priority job arbitrarily. ◊

The PIP addresses this issue through a *progress mechanism* called *priority inheritance*. Under priority inheritance, a resource-holding job is scheduled with an *effective priority* equal to the highest-priority job it

blocks. Therefore, if priority inheritance were applied to the previous example, J_3 would resume execution at time $t = 3$ and execute at the priority of J_1 until it completed its critical section at time $t = 4$. This prevents unbounded priority inversion for the higher-priority job J_1. Notably, however, J_2 experiences an s-aware priority inversion in this case as the lower-priority job J_3 executes even though J_2 is ready.

The priority-inheritance progress mechanism is used in many real-time locking protocols, both uniprocessor and multiprocessor.

2.2.3.2 Priority Ceiling Protocol (PCP) (Sha et al., 1990)

While the PIP solves the problem of unbounded priority inversion, it is still subject to the possibility of *deadlock*. Deadlock occurs when resource-holding jobs cannot make progress. Consider the following classic example of deadlock.

Example 2.6. Consider a situation in which J_1 holds ℓ_a and J_2 holds ℓ_b. If J_1 issues a nested request for ℓ_b and J_2 issues a nested request for ℓ_a, then the system is deadlocked. ℓ_a will not be unlocked until ℓ_b is acquired, but ℓ_b is waiting for ℓ_a to be unlocked. Therefore, neither J_1 nor J_2 can make progress and the system is deadlocked. $\Diamond$

The PCP extends the PIP in such a way so as to prevent the possibility of deadlock. To do so, the PCP requires that the set of resources that are accessed by each task are known *a priori*. This information is used to set the *priority ceiling* of each resource ℓ_q as the priority of the highest-priority task that accesses ℓ_q. When a task locks a resource, the *system priority ceiling* is defined to be the highest priority ceiling of any locked resource.

The goal of the system priority ceiling is to prevent a situation in which deadlock can occur. To do so, a task τ_i's request for an unlocked resource is denied or delayed if its priority is lower than the system priority ceiling (unless τ_i was the task that set the system priority ceiling). Referring back to the previous example, the priority-ceiling rules would prevent the scenario in which J_1 held ℓ_a and J_2 held ℓ_b. Instead, one of these two jobs's request would be denied thereby preventing the deadlock.

Not only is this deadlock-avoidance result significant, but the PCP also has worst-case s-aware pi-blocking of at most one outermost critical section. Intuitively, this is because a blocked job is only blocked by one resource-holding job, and the deadlock-avoidance rules prevent any transitive blocking. These two

key properties combine to limit the maximum duration of s-aware pi-blocking to at most the duration of one outermost critical section.

2.2.3.3 Stack Resource Policy (SRP) (Baker, 1991)

The SRP is a clever extension of the PCP that allows a single runtime stack to be used for all tasks. To show how multiple tasks can share the same stack, we first consider stack sharing in the absence of any synchronization.

If independent tasks are scheduled with a preemptive FP scheduling algorithm, they can share the same runtime stack. Consider a lower-priority task being preempted by a higher-priority task. The higher-priority task adds its own frame(s) to the runtime stack above the stack frame(s) of the lower-priority job. However, when the lower-priority job resumes, the higher-priority job will have completed, and popped its stack frames off the stack, leaving the stack in the same state as when the preemption occurred. Note, however, that if the higher-priority job ever suspended, for example, to block on a shared resource, then a lower-priority job would resume, but the high-priority job's stack frames would still be present on the stack. This behavior is seen in Example 2.5 and Figure 2.6. τ_1 suspends at time $t = 3$, and the preempted lower-priority job τ_3 resumes at time $t = 7$ while τ_1 is still on the stack. For this reason, if using the PIP or the PCP, each task must maintain its own runtime stack.

The SRP allows all tasks to share the same runtime stack, just as in the case of preemptively scheduled independent task systems. To highlight the difference between the PCP and the SRP, let us focus on the scheduling of independent tasks, in a system employing the PCP. The rules of the PCP do not apply to independent tasks; only tasks that access resources are affected by the rules of the PCP. In the SRP, both independent and resource-using tasks are prevented from executing if their priority does not exceed the system priority ceiling. The system priority ceiling is set just the same as in the PCP.

By incorporating the system priority ceiling into the scheduling of all tasks, it can be shown that all resources are available to be locked without any blocking whenever a running task issues a request. All of the blocking that tasks experience due to synchronization is shifted from the time of issuing a request to before commencing execution. Subsequently, the SRP also limits the number of context switches that occur

in practice, thereby reducing overheads, in addition to better utilizing memory by sharing a common runtime stack.[4]

2.2.3.4 Multiprocessor and Distributed Priority Ceiling Protocol (MPCP) (Rajkumar et al., 1988; Rajkumar, 1990; Sha et al., 1990; Rajkumar, 1991)

The first multiprocessor real-time locking protocols, the *multiprocessor priority ceiling protocol* (*MPCP*) and the *distributed priority ceiling protocol* (*DPCP*), are both extensions of the uniprocessor PCP for multiprocessor systems (Rajkumar et al., 1988; Rajkumar, 1990; Sha et al., 1990; Rajkumar, 1991).[5] The DPCP is designed for distributed-memory multiprocessors, while the MPCP is designed for shared-memory multiprocessors. Despite the different design objectives, the DPCP is also compatible with shared-memory platforms and so we consider it in our discussion here.

Both the MPCP and the DPCP assume P-FP scheduling. Resources are classified as either *local* if they are shared only by tasks within one partition, and *global* if they are shared among tasks in different partitions. Given the strong synchronization results for uniprocessors discussed previously in the PCP and the SRP, both the MPCP and the DPCP leverage the uniprocessor PCP for local resources. Global resources, however, are more difficult, as will be described next, and therefore must be handled differently. Specifically, nesting resource requests of global resources is prohibited.

One of the key challenges in the design of multiprocessor synchronization algorithms is the design of the progress mechanism. For partitioned or clustered scheduling, priority inheritance is ineffective in ensuring progress and therefore bounded priority inversions. Intuitively, this is because a task τ_l may have a relatively low priority with respect to its processor, but have sufficient priority to be scheduled, while a relatively high-priority task τ_h may not have sufficient priority to execute on another processor due to the presence of other even-higher-priority tasks. If τ_h holds a resource thereby blocking τ_l, inheriting τ_l's priority will not ensure progress. This example demonstrates how priority inheritance is insufficient to ensure resource-holder progress across partitions.

[4]Sharing a common runtime stack is not a fundamental requirement of the SRP, and indeed it can be implemented with each task having its own stack.

[5]The DPCP was originally referred to as the "multiprocessor PCP" in (Rajkumar et al., 1988) and the MPCP was called the "shared memory synchronization protocol" in (Rajkumar, 1990). The names DPCP and MPCP as they are known today were introduced in (Rajkumar, 1991).

To address this issue, both the MPCP and the DPCP employ a different progress mechanism called *priority boosting*. To ensure resource-holder progress, priority boosting unconditionally raises the priority of a resource-holding job to the maximum priority, such that it cannot be preempted by non-resource-using tasks.

In both the MPCP and the DPCP, requests for global resources are satisfied in order of their base priority. The major difference between the two protocols lies in where and how the critical sections are actually executed. In the DPCP, which was designed for distributed memory platforms, each resource is pinned to a single processor. A *remote procedure call* is used to request the *agent* of the resource to actually execute the critical section on the resource. The agent executes these requests with a boosted priority on the processor on which the resource resides, and in the order of the priority of the requesting tasks.

In comparison, the MPCP assumes a shared-memory model, which allows critical sections to be executed directly by the requesting task from any processor. Throughout the duration of any of its critical sections, a global-resource-holding task is priority-boosted. Similar to the DPCP, requests are satisfied in priority order using the priority of the requesting tasks.

For brevity, we omit detailed discussion concerning the blocking analysis of both the MPCP and the DPCP. However, we note that the original blocking analysis (Rajkumar, 1991)[6] has since been significantly improved (Lakshmanan et al., 2009).

2.2.3.5 Multiprocessor Stack Resource Policy (MSRP) (Gai et al., 2001)

Just as the MPCP extends the PCP to partitioned multiprocessor platforms, the MSRP also extends the SRP to partitioned multiprocessor platforms. As such, there are many commonalities between the MSRP and the MPCP. Resources are classified as either local or global resources, and there can be no nested requests for global resources. The SRP is used to arbitrate access to all local resources, and non-preemptive *first-in first-out* (*FIFO*) spinlocks are used to arbitrate access to all global resources. These last two properties are the most significant difference between the MSRP and the MPCP—FIFO queuing is used for global resources instead of priority queueing, and waiting is realized via non-preemptive spinning, instead of suspensions.

While non-preemptive spinning does "waste" processor cycles, it is amenable to simple analysis. There can be at most m concurrent requests for all global resources, one per processor. With FIFO queueing, no

[6]Recently, it has been shown that the original MPCP and DPCP blocking analysis is slightly flawed (Yang et al., 2016).

request can "cut ahead" and block an earlier-issued request. Therefore, under the MSRP, each global request can be blocked by at most one request per processor.

2.2.3.6 Flexible Multiprocessor Locking Protocol (FMLP) (Block et al., 2007)

Block et al. (2007) developed the *flexible multiprocessor locking protocol* (*FMLP*), which supports both partitioned and global scheduling, and spin- and suspension-based waiting. Like the MSRP, the FMLP leverages FIFO-ordered waiting.

In the FMLP, resources are classified as either *short* or *long*, depending upon the length of the corresponding critical sections for that resource. Waiting for short resources is realized via spinning, while waiting for long resources is realized via suspending. The motivation behind this decision is that for short critical sections, the overhead of context switching to another job outweighs the benefits of relinquishing the processor while waiting. For long critical sections, the converse may be true.

Block et al. (2007) also formalized the concept of *group locking* to complement the FMLP. Under group locking, resources are aggregated into groups that are treated as a single lockable entity, a form of coarse-grained locking (recall terminology from discussion in Section 2.2.1). Under group locking, all long (respectively short) resources are aggregated into groups such that if a job holding a resource ℓ_a may issue a nested request for another resource ℓ_b then ℓ_a and ℓ_b are contained in the same resource group. The FMLP does allow a job holding a long-resource group to issue a nested request for a short resource group. While this represents finer-granularity locking than previous protocols, we still consider it coarse-grained locking. It does, however, provide slightly more flexibility than earlier protocols such as the MSRP, as it allows nested resources to be accessed from different processors.[7]

The worst-case blocking of the FMLP is also fairly simple to reason about generally (though fine-grained blocking analysis is much more nuanced (Brandenburg, 2011, 2013b; Wieder and Brandenburg, 2013; Yang et al., 2015)). For spin-based waiting, the FMLP is very similar to the MSRP in that a request may only be blocked by $m-1$ earlier-issued requests. For suspension-based waiting, the length of the wait queues can extend to $O(n)$ requests. Further discussion of the blocking-bound results of the FMLP will be given in the discussion in Section 2.2.4.

[7]Because nested locking of global resources is disallowed in earlier protocols, any nesting must be handled within one processor via a uniprocessor locking protocol.

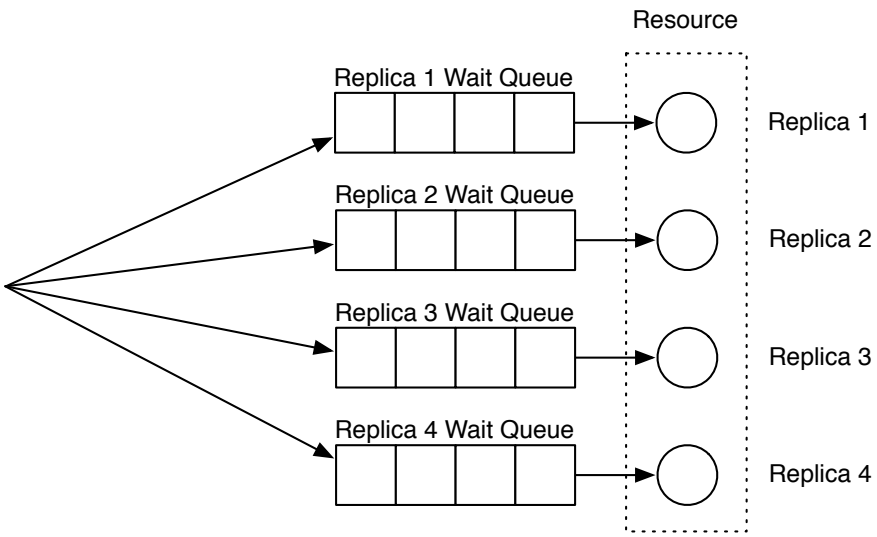


Figure 2.7: Queue structure of the k-FMLP with $k = 4$. Notably, the k-FMLP generalizes the FMLP by allowing $k > 1$. Each queue is FIFO ordered. Priority inheritance is used to ensure progress within each wait queue.

Elliott (2015) extended the FMLP to support the more general k-exclusion sharing constraint, in which at most k critical sections may be satisfied concurrently. k-exclusion is often used to arbitrate access to replicated resources, where a task may issue a request for an arbitrary resource replica. The resulting protocol, called the k-FMLP, is very similar to the FMLP in that it uses a FIFO wait queue for each of the k resource replicas, as depicted in Figure 2.7. When a request is issued, it is enqueued in the shortest replica wait queue, thereby limiting the length of each wait queue to at most $O(n/k)$.

2.2.3.7 FIFO Multiprocessor Locking Protocol (FMLP$^+$) (Brandenburg, 2014)

In his dissertation, Brandenburg (2011) presented an extension of the FMLP for partitioned-scheduled systems, which was originally termed the FMLP$^+$. However, in later work (Brandenburg, 2014), that protocol was generalized to support clustered scheduling as well. We therefore restrict our attention to the later variant.

The original FMLP was thought to have an s-aware pi-blocking bound of $O(n)$ under any JLFP global scheduler. Brandenburg (2011, 2014) showed this not to be true by constructing a task system with worst-case pi-blocking lower-bounded by $\Omega(\Phi)$ where Φ is the ratio of the maximum to minimum task periods. In this example task system, a high-priority task may be repeatedly pi-blocked due to the progress mechanism of either priority inheritance or (unrestricted) priority boosting. To address this issue, Brandenburg (2014) presented a new progress mechanism called *restricted segment boosting* (*RSB*). Under RSB, jobs are

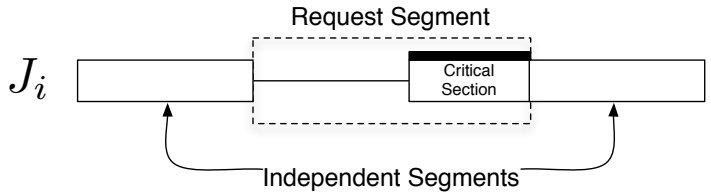


Figure 2.8: Illustration of independent and request segments for RSB.

decomposed into segments of execution, depicted in Figure 2.8: *independent segments*, in which the job has not requested a resource; and *request segments*, in which the job is waiting on or using a shared resource. In order to balance s-aware pi-blocking among independent and request segments, RSB boosts the priority of at most one request segment at a time, and may *co-boost* some other segments to ensure that boosted low-priority jobs do not cause undue pi-blocking for independent segments.

More specifically, let J_b be the lock-holding job with the earliest segment start time in its cluster, which we denote t_b. J_b is boosted, and is therefore scheduled. Let $C(J_b)$ be the (at most) $c-1$ highest-priority ready jobs with priority higher than J_b and segment start times before t_b. (By construction, all jobs in $C(J_b)$ are in independent segments.) All jobs in $C(J_b)$ are co-boosted, and therefore scheduled. The remaining cores (if any) schedule the highest-priority jobs that are neither boosted nor co-boosted.

Example 2.7. Consider the clustered EDF system with $c = 2$ depicted in Figure 2.9. We focus our attention on one cluster in a larger system. At time $t = 0$, J_2 begins an independent segment, and at time $t = 2$, J_3 is released and begins an independent segment on another processor. At time $t = 3$, J_3 issues a request, and therefore enters a request segment. J_3 is blocked waiting for other critical section(s) to complete on other cluster(s) (not shown). At $t = 4$, J_1 is released and begins execution. At time $t = 5$, J_3 acquires a lock, and is boosted and therefore scheduled. J_2 has both a higher priority than J_3 and an earlier segment start time of $t = 0$, and therefore is co-boosted. Therefore J_1 is pi-blocked at time $t = 5$ due to the boosting and co-boosting of J_3 and J_2, respectively. Under unrestricted priority boosting, J_1 would be scheduled at time $t = 5$ instead of J_2, leaving J_2 pi-blocked. By co-boosting jobs using this technique, jobs cannot be repeatedly pi-blocked leading to undue pi-blocking. ◊

The FMLP$^+$ uses RSB in conjunction with simple FIFO queuing, similar to the FMLP, for each resource. However, at most one job may be boosted at any given time, even if multiple resource-holding jobs (holding different resources) are ready. Note that multiple resource-holding jobs may execute concurrently if they

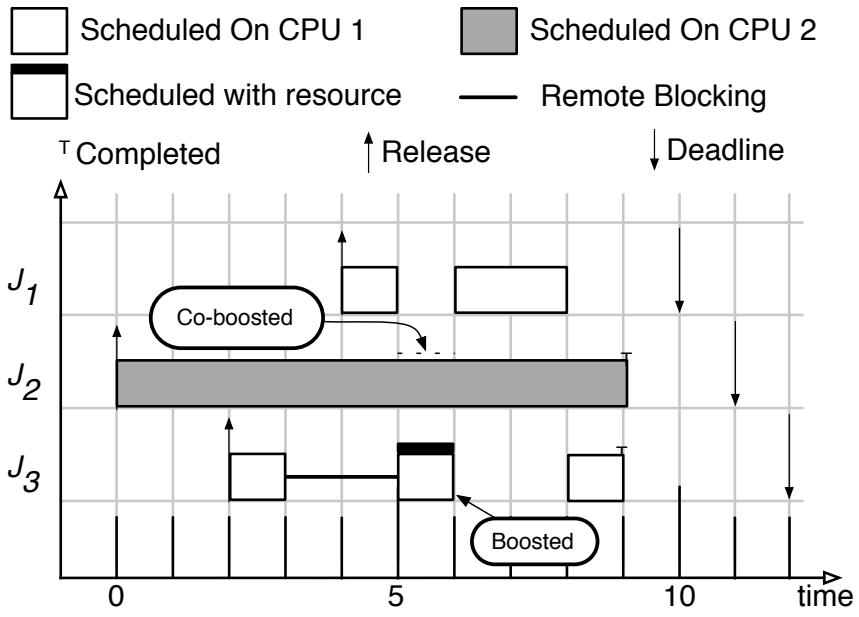


Figure 2.9: Example schedule depicting boosting and co-boosting in RSB. This schedule depicts only one cluster, of size $c = 2$, in a larger system. Within the cluster G-EDF is assumed with RSB.

have sufficient priority, but only one resource-holding job may be boosted. Using these techniques, it can be shown that the FMLP$^+$ has $O(n)$ s-aware pi-blocking under both clustered and global JLFP scheduling.

2.2.3.8 $O(m)$ Multiprocessor Locking Protocol (Brandenburg and Anderson, 2010a, 2011)

After formally defining s-aware and s-oblivious pi-blocking analysis (recall Definitions 2.1 and 2.2), Brandenburg and Anderson (2010a) presented the $O(m)$ *multiprocessor locking protocol* (*OMLP*). In later work (Brandenburg and Anderson, 2011, 2014), the OMLP was actually extended to a family of locking protocols that can be used under many different platform configurations. We begin our discussion with the global OMLP (G-OMLP), which is only designed for global scheduling.

Prior to the G-OMLP, all multiprocessor locking protocols used either FIFO or priority-ordered queueing. The G-OMLP uses a combination of FIFO and priority-ordered queueing, depicted in Figure 2.10. Requests are initially queued into a priority-ordered wait queue, which feeds a FIFO wait queue of length m. Priority inheritance is used to ensure resource-holder progress. Brandenburg and Anderson (2010a) showed that a request can experience at most $O(m)$ s-oblivious pi-blocking while it is queued in the priority-ordered queue, and $O(m)$ s-oblivious pi-blocking in the FIFO wait queue. In the aggregate, the G-OMLP therefore has a worst-case s-oblivious pi-blocking bound that is $O(m)$.

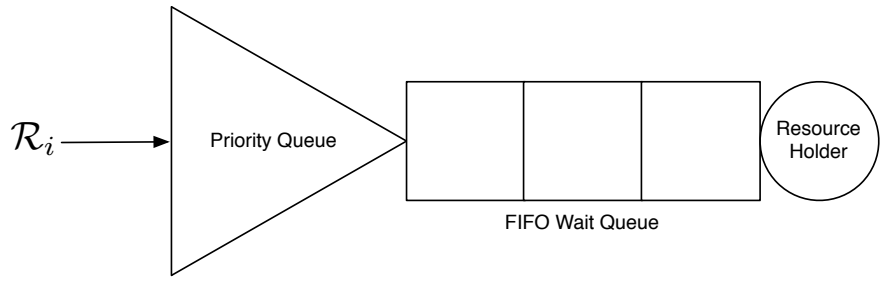


Figure 2.10: Figure depicting the queue structure of the G-OMLP. The triangular queue structure represents a priority queue, while the rectangular queue represents a FIFO queue.

Brandenburg (2013a) also presented the $O(m)$ *independence preserving* (*OMIP*) locking protocol, a close cousin of the G-OMLP, which can be applied to clustered systems by using *migratory priority inheritance*. Migratory priority inheritance allows critical sections to migrate to another cluster to be executed using a priority inherited from that cluster. For example, let J_r be a resource-holding job in Cluster 1 and J_h be a job with sufficient priority to be scheduled in Cluster 2. If J_h is blocked waiting for J_r to complete its critical section, under migratory priority inheritance J_r would migrate to Cluster 2 and continue executing its critical section with the inherited priority of J_h. Like the G-OMLP, the OMIP is *independence preserving* in that non-resource-using tasks do not experience s-oblivious pi-blocking.

If migrating critical sections among clusters is disallowed, priority inheritance alone is insufficient to ensure resource-holder progress in partitioned and clustered systems. Likewise, in clustered systems, where $1 < c < m$, priority boosting is also problematic as one job may be repeatedly pi-blocked as a result of priority boosting. To address this issue, Brandenburg and Anderson (2011) developed a new progress mechanism called *priority donation*, which limits the maximum duration of pi-blocking attributable to the progress mechanism itself to $O(m)$. To do so, priority donation is shown to satisfy the following two properties (Brandenburg and Anderson, 2011):

PD1 A resource-holding job is always scheduled.

PD2 The duration of s-oblivious pi-blocking caused by the progress mechanism (*i.e.*, the rules that maintain Property PD1) is bounded by the maximum *request span* of any job, where the request span is defined to be the time between the issuance and completion of a request.

The rules of priority donation ensure that there are at most c outstanding requests per processor, and therefore at most m total. When a job is released, if it has a higher effective priority than a job with an

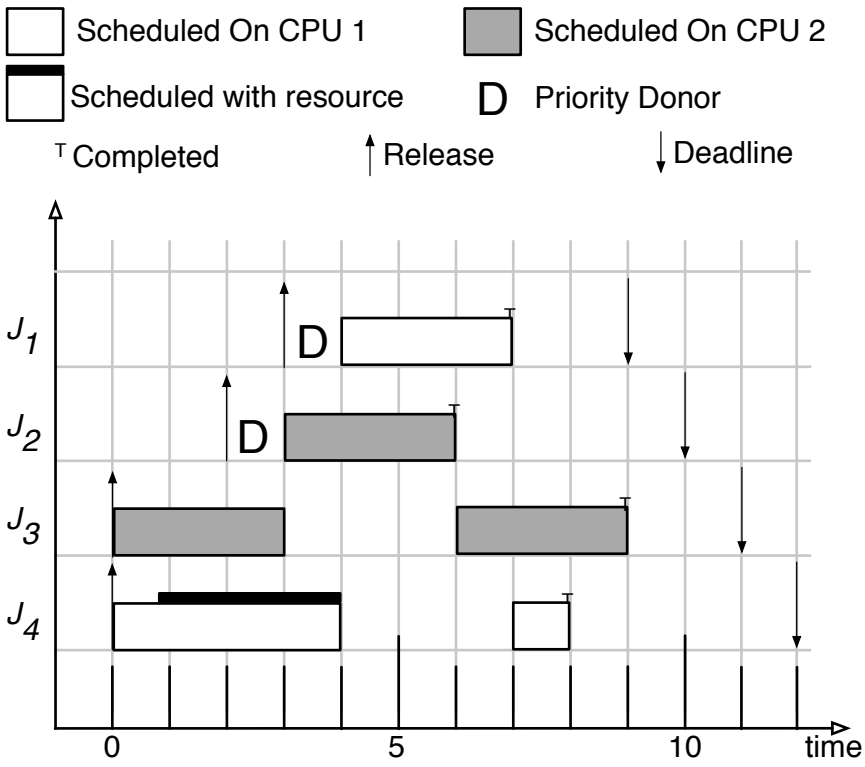


Figure 2.11: Example schedule depicting priority donation on two processors scheduled according to G-EDF.

outstanding request in that cluster, it is forced to *donate* its priority to that job. Note that this may relinquish a previous donor from its donation obligation. These rules ensure that a job with an incomplete resource request always has sufficient effective priority to be scheduled in its cluster, which means that when it acquires its needed resource, Property PD1 is satisfied. Furthermore, a job may only donate its priority to one other job, and therefore after that job completes its critical section, the job's donation obligation is complete. Therefore, the duration of donation is bounded by the maximum request span, or the time between request and critical-section completion.

Example 2.8. To demonstrate some of the principles of priority donation, consider the example depicted in Figure 2.11 . In this example, J_4 holds a resource from time $t = 1$ to time $t = 4$. When J_2 is released at time $t = 2$, it has sufficient priority to preempt the resource-holding job J_4. Under priority donation, J_2 donates its priority to J_4 so as to ensure progress. At time $t = 3$, J_1 is released, and donates its priority to J_4, relinquishing J_3 from its donation obligation, allowing J_2 to preempt J_3. At time $t = 4$ when J_4 completes its critical section, the donation obligation of J_1 is relinquished, and it preempts J_4. The remainder of the example schedule proceeds according to G-EDF. ◊

Priority donation is a very powerful progress mechanism that can be used to construct locking protocols for many use cases. Brandenburg and Anderson (2011) showed how priority donation can be used to construct mutex, *k*-exclusion, and *reader/writer locks*, all coexisting on the same platform. A reader/writer lock distinguishes between two types of requests: *reads*, which can be satisfied concurrently, and *writes*, that require mutual exclusion. The mutex variant, which we denote the C-OMLP, uses a single FIFO wait queue for each resource, and therefore the maximum request span is $O(m)$. The *k*-exclusion variant, which we denote the CK-OMLP, also uses a single FIFO wait queue in contrast to the *k*-FMLP, which uses a FIFO wait queue per replica. The maximum request span for *k* exclusion is therefore $O(m/k)$. Finally, the R/W variant is designed using phase-fair logic (Brandenburg and Anderson, 2009), which is described in more detail later.

2.2.3.9 Multiprocessor Resource-Sharing Protocol (MrsP) (Burns and Wellings, 2013)

The MrsP is another multiprocessor locking protocol for P-FP scheduling, with similarities to many previously discussed locking protocols, but with some novel improvements. The MrsP is a spin-based protocol, similar to the MSRP. A major difference between these two protocols is the priority at which waiting tasks spin. In the MSRP, tasks employ non-preemptive spinning, or priority boosting through the duration of the request span. Burns and Wellings (2013) identified that this can cause significant blocking, especially for high-priority, short-deadline jobs, which can cause significant schedulability loss.

The MrsP combats this issue by letting tasks spin while waiting for requested global resources at their base priority. This shields high-priority jobs from excessive blocking, but raises a new issue: resource-holding jobs may be preempted, which can cause undue blocking for requests on other processors. Burns and Wellings (2013) addressed this issue in the MrsP with a "helping mechanism," which allows waiting jobs to "help" preempted jobs by completing the execution of an in-progress critical section. This helping mechanism has many parallels to migratory priority inheritance (Brandenburg, 2013a) as used in the OMIP, multiprocessor bandwidth inheritance (Faggioli et al., 2010), and a technique called "local helping" (Hohmuth and Peter, 2001).

Unlike all of the previously discussed multiprocessor locking protocols, the MrsP includes support for nested locking of global resources. However, the blocking bound associated with nested locking is greater than that with non-nested locking, negating the benefit of fine-grained locking.

2.2.3.10 Multiprocessor Bandwidth Inheritance (M-BWI) (Faggioli et al., 2010, 2012)

Resource reservations, or *budgets* are used to ensure *temporal isolation*, in which one task overrunning its provisioned or budgeted execution time cannot cause a deadline violation for another task. Such budgeting is useful in *open systems*, in which tasks may dynamically enter and leave the system. *Bandwidth inheritance* (*BWI*) extends similar concepts from priority inheritance discussed earlier to such budgeted systems, allowing processing bandwidth to be inherited to ensure a resource-holding task has sufficient priority as well as bandwidth to make progress.

M-BWI (Faggioli et al., 2010, 2012), is an extension of BWI for multiprocessor systems. M-BWI can be applied under partitioned, clustered, and globally scheduled systems. M-BWI employs FIFO wait queues per resource, and waiting tasks busy wait. If the resource-holding task is preempted or its budget is exhausted, the bandwidth-inheritance mechanism allows the critical section to be migrated to another task and executed.

Lock nesting is supported in M-BWI, and deadlock can be prevented using a lock ordering. Faggioli et al. (2012) present a brute-force analysis of the blocking bounds under M-BWI. In that analysis, all possible request-order permutations or *blocking chains* are considered. This results in blocking analysis that is super-exponential in the number of requesting tasks. Furthermore, the authors acknowledge the bound is pessimistic as it does not incorporate some more detailed information, such as task periods.

2.2.3.11 Phase-Fair Reader-Writer Locking (PF) (Brandenburg and Anderson, 2009, 2010b, 2011)

Until this point, our discussion of previous locking protocols has predominantly focused on mutex locking. Next, we turn our attention to reader/writer locking. Brandenburg and Anderson (2009) presented *phase-fair* (*PF*) *reader/writer locking*, the first multiprocessor real-time reader/writer locking protocol.[8] They also presented the first blocking analysis, needed for real-time schedulability, of many previously studied throughput-oriented multiprocessor reader/writer algorithms. We focus our attention here on PF locking, as it is the best known reader/writer lock both theoretically, and in practice for the majority of cases in which reader/writer locking is preferable to mutex locking. For brevity, we refer the interested reader to (Brandenburg and Anderson, 2009; Brandenburg, 2011) for complete analysis of other throughput-oriented reader/writer locks.

[8]The uniprocessor PCP and SRP can be extended to support reader/writer locking (Baker, 1991; Rajkumar, 1991).

PF locking is based on the idea that read requests and write requests should "take turns." Each of these "turns" is called a *phase*. During a write phase, one write request is satisfied. During a read phase, many read requests are satisfied concurrently. Formally, phase-fair locking is defined as follows.

Definition 2.4. (Brandenburg and Anderson, 2009). A RW lock is phase-fair if and only if it has the following properties:

PF1 reader phases and writer phases *alternate*;

PF2 writers are subject to FIFO ordering, but only with regard to other writers;

PF3 at the start of each reader phase, all currently unsatisfied read requests are satisfied (exactly one writer request is satisfied at the start of a write phase); and

PF4 during a reader phase, newly issued read requests are satisfied only if there are no unsatisfied write requests pending.

Example 2.9. To illustrate PF locking, consider the example depicted in Figure 2.12. So as not to confuse the effects of scheduling and synchronization, in this example, we assume one task per processor. To begin, J_4 issues a read request which is immediately satisfied. At time $t = 1$, J_3 issues a read request, which by Property PF4, is also satisfied, concurrently with J_4. At time $t = 2$, J_2 issues a write request, which blocks to wait for the read phase to complete. At time $t = 3$, J_4 completes, but J_2 continues to block as J_3 is still reading. Also at time $t = 3$, J_1 issues a read request, but by Property PF4, it must block until the next read phase. At time $t = 4$, J_3 completes, ending the current read phase, allowing J_2 to begin its write critical section. At time $t = 6$ when J_2 completes its write critical section, the write phase completes, and alternates to a read phase, allowing J_1 to begin its read critical section. $\Diamond$

From these rules, it can be shown that a read request may be blocked for at most the duration of one read request and one write request, totaling $O(1)$ blocking. Similarly, each write phase may be preceded by one read phase, and therefore the worst-case blocking for a write request is also $O(m)$. The original PF lock was a spin lock (Brandenburg and Anderson, 2009), but later phase-fair logic was used to construct a suspension-based lock using priority donation (Brandenburg and Anderson, 2011).

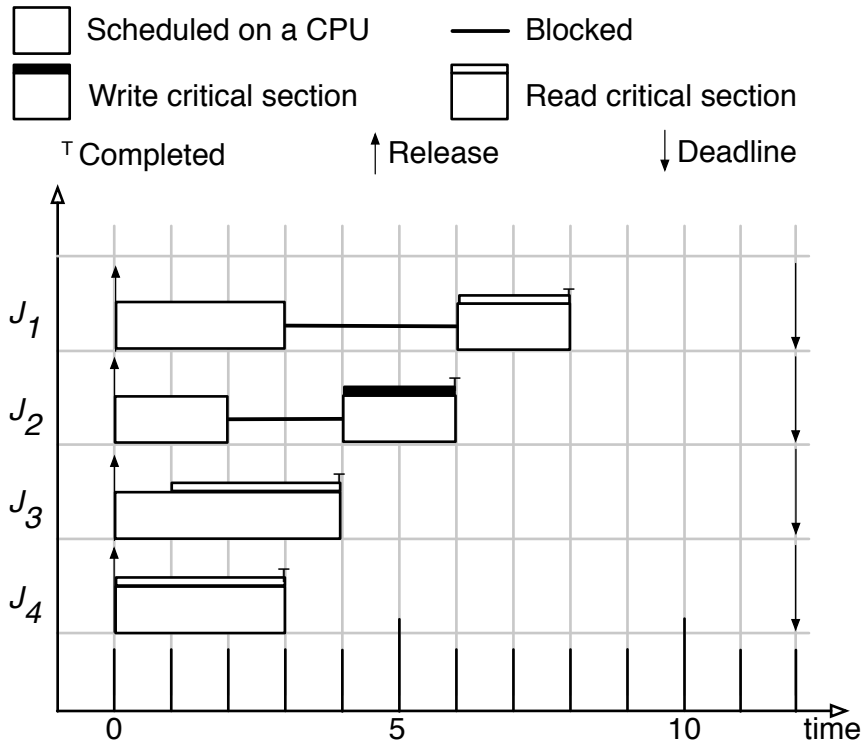


Figure 2.12: Example schedule depicting PF locking. In this example, four tasks are scheduled on four processors, and therefore the scheduler is inconsequential, allowing us to focus on the effects of the PF locking.

Analysis	Scheduler	Locking Protocol	PMR Pi-blocking	Per-Request Pi-blocking
spin	Any	FMLP	$O(m)$	$O(m)$
s-aware	Partitioned	FMLP$^+$	$O(n)$	$O(n)$
	Clustered	FMLP$^+$	$O(n)$	$O(n)$
	Global	FMLP$^+$	$O(n)$	$O(n)$
s-oblivious	Partitioned	C-OMLP	$O(m)$	$O(m)$
	Clustered	C-OMLP	$O(m)$	$O(m)$
		OMIP	0	$O(m)$
	Global	G-OMLP	0	$O(m)$

Table 2.1: Summary of existing single-resource locking protocols and their blocking complexity. The column "PMR Pi-Blocking" indicates how long any job in the system (whether it accesses shared resources or not) can be progress-mechanism-related pi-blocked. The column "Per-Request Pi-Blocking" indicates how long a job can be pi-blocked per request. All listed protocols are asymptotically optimal.

2.2.4 Summary of Prior Multiprocessor Locking Protocols

After describing many of the most prominent results in uniprocessor and multiprocessor real-time synchronization, we briefly summarize the results, and highlight other contributions concerning real-time synchronization.

Real-time locking optimality. Brandenburg and Anderson (2010a) presented two task systems that, regardless of the lock-request ordering, have $\Omega(n)$ and $\Omega(m)$ worst-case s-aware and s-oblivious pi-blocking, respectively. Based on these results, several of the previously discussed locking protocols are optimal, assuming that both the number of requests, and the length of each request, is bounded by a constant. A summary of the optimality results for multiprocessor real-time synchronization is given in Table 2.1.

The protocols given in Table 2.1 are all optimal. Many other non-optimal protocols have been developed, including some of which that have been discussed herein. It is indeed possible that suboptimal locking protocols may exhibit better performance in practice than optimal protocols. This could be attributable to higher runtime overheads (similarly to our discussion in Section 2.1) for the optimal locking protocols, or smaller *constant factors* in the blocking analysis of the suboptimal protocols. These issues have been studied in detail in followup studies (Brandenburg, 2011). In practice none of the previously presented multiprocessor locking protocols dominates any other.

In addition to work on developing locking protocols themselves, there has also been significant research effort on developing improved analysis for existing locking protocols. Recently, Brandenburg (2013b), Wieder and Brandenburg (2013), and Yang et al. (2015) have presented a more flexible and modular form

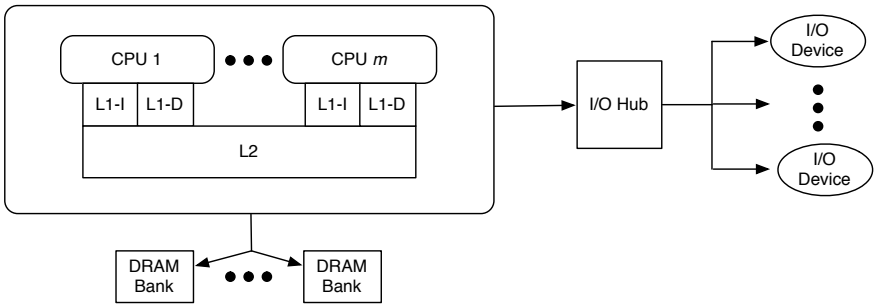


Figure 2.13: Depiction of the architecture of a common multicore platform.

of blocking analysis based on *linear programming* (*LP*), which can be applied to many different locking protocols. Using this LP-based blocking analysis results in more exact blocking bounds than previous analysis. Additionally, the LP-based analysis has a modular nature to it, in that constraints for one locking protocol may be easily applied to the blocking-analysis LP for another similar locking protocol (provided the constraints are proven safe for both protocols). The LP-based analysis also allows for the formulation of application-specific blocking constraints. For example, it is possible to add constraints that model a task system in which only every other job issues a request for a given resource.

Of the multiprocessor real-time locking protocols discussed in this section, the only ones to support fine-grained nested locking are the MrsP and M-BWI. Those protocols are not optimal under any definition of pi-blocking.

2.3 Multiprocessor Hardware Platforms

The aforementioned synchronization algorithms can be applied to arbitrate access to many different classes of resources, ranging from shared data objects to physical hardware components. In this section, we review key components of multicore architectures and describe how applying synchronization algorithms to hardware components can provide *isolation* to the accesses to such components that can improve runtime performance and timing predictability.

2.3.1 Multicore-Architecture Considerations

While a brief overview of the architecture of multicore platforms was presented in Chapter 1, here we describe in greater detail multicore architectures and their implications in the design of real-time systems. Figure 2.13 depicts a common multicore architecture, which is the subject of discussion in this section.

2.3.1.1 Processing Cores

Multicore platforms include two or more *processing cores*. The processing cores themselves contain all of the hardware to execute arithmetic and logic operations (*e.g.*, addition, disjunction, shift, *etc.*), control-flow operations (*e.g.*, jump, branch, *etc.*), as well as potentially many other more complex operations (*e.g.*, test-and-set, single-instruction-multiple-data (SIMD) instructions, *etc.*). Each core also contains *registers* to store data, and any internal state necessary to carry out such operations. We assume that each core has exactly one *program counter*, and can therefore execute at most one program or thread of computation at a time.[9] All of this functionality is logically encapsulated into one processing core (Hennessy and Patterson, 2007).

2.3.1.2 Memory

Programs executing on each of the processing cores may access instructions and data, which must be accessed and stored in *main memory*. Computers have long been designed with hierarchical memory, dating at least as far back as the seminal work in which the von Neumann computer architecture was proposed (Burks et al., 1946). Hierarchical memory is necessitated by the tradeoff between size and speed—smaller memories can be made much faster, but larger memories are desirable due to their capacity to store data. Modern processors use relatively small, fast *caches* built into the silicon of the processor to store a subset of data from the larger, slower main memory. This hierarchy can be seen in Figure 2.13.

Memory is managed by the operating system to allocate different regions of memory to different processes or real-time tasks. To do so, memory is subdivided into contiguous, fixed-length blocks called *pages*. All memory management is conducted at the page granularity. To simplify the task of memory management and application programming, an additional level of indirection is added in the memory system, called *virtual memory*. With virtual memory, the operating system maps *virtual-memory addresses* used by the application, to *physical-memory pages*, corresponding to the addresses of physical memory in hardware, as depicted in Figure 2.14. The translation of virtual-memory addresses to physical memory addresses is conducted in hardware in the *memory management unit* (*MMU*). The mapping of physical-memory to virtual-memory addresses is stored in *page tables*, which are set up by the operating system and stored in memory. To

[9] Some architectures have multiple program counters to support multiple hardware threads. Such threads share functional units within the core. However, the interactions between threads on a single core are unpredictable and difficult to quantify. Traditionally, in real-time applications hardware threading is disabled, but we note that improving the predictability of hardware threading is an interesting avenue of future research.

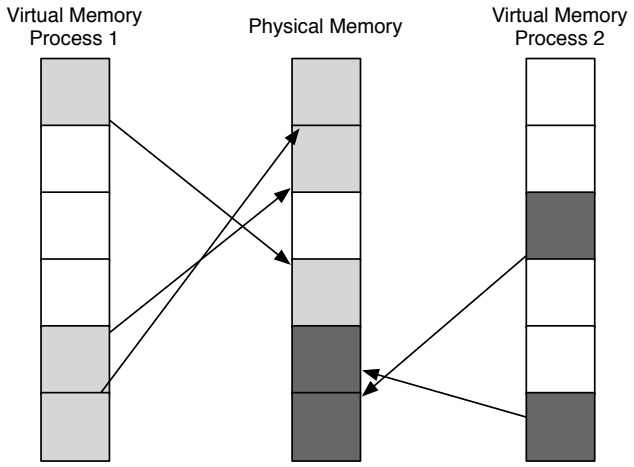


Figure 2.14: Depiction of virtual memory.

facilitate faster translation in the MMU, a special page-table cache is implemented in hardware called the *translation lookaside buffer* (*TLB*).

Virtual memory has a number of desirable properties. It simplifies application development as applications are presented with a linear memory model. With a linear memory model, memory appears to the program as a single contiguous address space. Programs need not be concerned with the memory allocations of other processes. Not only does this improve programmability, but it also improves security by providing memory isolation. With virtual memory, one process cannot access the memory of another process. Finally, virtual memory allows for *demand paging*, where memory can be stored in persistent disk storage, thereby allowing for a larger effective memory size. However, demand paging can introduce significant latency to memory accesses to disk-resident data. In this dissertation, we assume that memory accessed by real-time tasks is *pinned* into RAM, and therefore never swapped out to disk.

Physical memory that is external to the chip where the CPU cores reside is stored in hardware in *dynamic random-access memory* (*DRAM*) banks. The architecture of DRAM banks is shown in Figure 2.15. Modern DRAM designs contain multiple *banks*. Each bank consists of memory in an array of rows and columns, along with a *row buffer*. For a memory location to be read or written, the row corresponding to that location must be stored in the row buffer. If the necessary row is not stored in the row buffer, a row-buffer miss occurs, and the row buffer must be loaded from DRAM. Otherwise, a row-buffer hit occurs, which reduces the latency of the memory fetch. Memory controllers are designed to re-order memory references to achieve

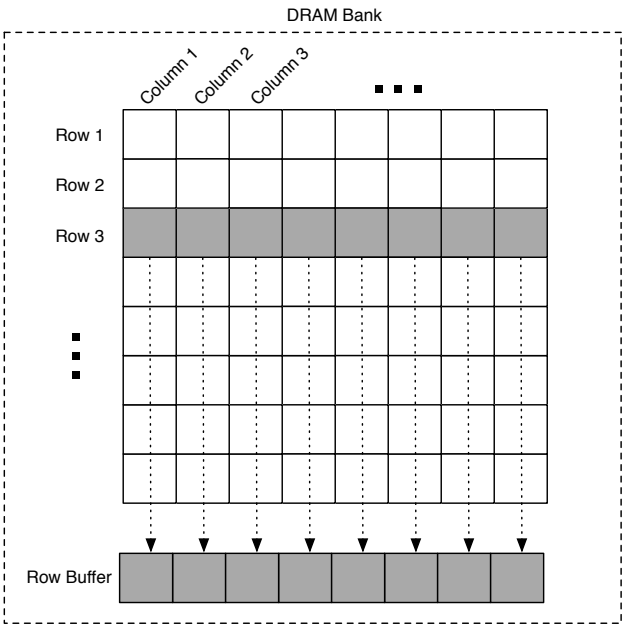


Figure 2.15: Figure depicting the architecture of a DRAM bank. Data is fetched one row at a time into the row buffer, which acts as a cache of the contents of that row, accelerating memory lookups until another row is loaded in.

better performance from the row buffer. Notably, for real-time applications, the performance metrics used in the design of memory controllers are different from those in general-purpose computing applications.

2.3.1.3 Caches

While main memory can store significant amounts of data (up to few gigabytes for smaller embedded applications, and a few terabytes for high-performance, general-purpose machines), the latency of accessing that data can be significant. In the ARM Cortex A9 platform considered in Kim et al. (2016), memory references have an access latency of between 120 and 300 ns. For that machine, which is clocked at 800 MHz, this access latency is as much as 240 processor cycles. The purpose of smaller on-chip caches is to allow some memory references to be cached so fewer clock cycles are wasted fetching data from memory. The ARM Cortex A9 platform contains two cache levels, the access latencies of which are depicted in Table 2.2.

Caches store copies of data that reside in memory, thereby enabling faster access. Accesses to memory addresses that are not cached must be fetched from memory itself. Application developers do not have explicit access to the cache to decide when and where to cache what data, but instead, such decisions are made by the cache hardware. (As an alternative or supplement to caches, some hardware platforms provide *scratchpad memory*, a small fast memory to which application developers have direct access.)

Level	Access Latency	Size
L1-I	5 ns	32 KB
L1-D	5 ns	32 KB
L2	38 ns	1 MB
Main Memory	120–300 ns	1 GB

Table 2.2: Access latency for different levels in the memory hierarchy in the ARM Cortex A9 platform considered in (Kim et al., 2016) as measured using lmbench (McVoy and Staelin, 1996). The L1 is the cache closest to the CPU, and is split between the L1-I, which stores instructions, and the L1-D which stores data. The L2 cache is a second level further from the CPU than the L1.

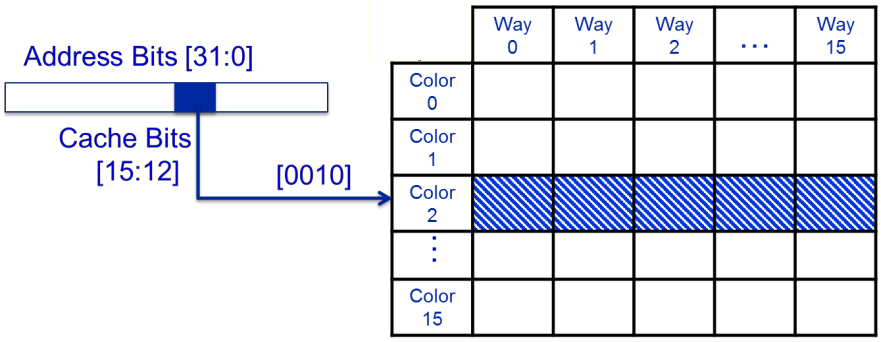


Figure 2.16: Depicting the architecture of the set-associative cache in the ARM Cortex A9 platform. Each column represents a way of set associativity, and the 16 colors correspond to 2048 cache sets, 128 cache sets per color.

To better understand the inner workings of the cache, we begin our discussion with a single-level cache before expanding the discussion to the complexities associated with multi-level caches. Cache data is transferred to and from memory in fixed-size blocks called *cache lines*. Associated with each cache line is a *tag*, indicating the address in memory from which it originated. Each memory address maps to a single cache line. Because the cache is significantly smaller than memory, many memory addresses map to the same cache line. In a *direct-mapped* cache, only one such address may be cached, while in a *set-associative* cache, many addresses that map to the same cache set may be cached concurrently. In other words, several conflicting cache lines may be concurrently cached, one per *way* of associativity. Set-associative caches are most common, and therefore we focus on them for the remainder of this discussion. The collection of addresses in a page map to a collection of sets, which is collectively referred to as a *color*. This is depicted in Figure 2.16. Given this discussion, a set-associative cache can be thought of as a 2d grid of colors and ways.

When the processor issues a memory reference, it first looks to see if that address is cached. If it is, the data is quickly returned, in what is called a *cache hit*. Otherwise, a memory request is issued to retrieve the data, in what is called a *cache miss*. The fetched data is then stored in the cache in the set to which it maps.

In the case of a set-associative cache, that data may be cached in one of several *ways*, so an *eviction policy* decides where to store the data. The most well-known eviction policy is *least-recently used* (*LRU*). Under LRU, the least-recently used cache line from the same set is *evicted* and replaced with the data that was just fetched from memory. Another common eviction policy, which is found in the previously discussed ARM platform, is *pseudo random*, which is rather common due to its simple implementation.

Multi-level caches. Next we expand our discussion to understand the complexities that arise when adding additional cache levels, as is common in most processors today. Multi-level caches introduce additional cache configuration options. Two common multi-level cache-configurations are *strictly inclusive*, in which data in the lower-level cache (*e.g.*, L1) must also reside in the upper-level cache (*e.g.*, L2); and *exclusive*, in which data may be cached in either the lower-level cache or the higher-level cache, but not both. Other hybrid policies exist, but they are outside the scope of this discussion.

Caches may also be either *split* or *unified*. In a split cache, program instructions are stored in one cache (L1-I in Figure 2.13 and Table 2.2), and program data is stored in another cache (L1-D in Figure 2.13 and Table 2.2). These two caches are logically and physically separate. Split caches are common at the L1 level, as depicted in Figure 2.13. Split caches prevent program instructions from *thrashing*, or repeatedly evicting, the cached data, and vice versa. Higher cache levels are most often unified in that both instructions and data are stored in the same cache.

Shared caches. Caches can also be *shared* or *local*. Local caches are only accessible by one processor, while shared caches are accessible by two or more processors. Local caches are used for lower-level caches, while shared caches are often used for the *last-level cache* (*LLC*). For example, instruction caches are always local, as program instructions are rarely if ever shared across processors. *Cache coherence protocols* are used to ensure that memory cached in two or more caches is consistent, or that a reference to either cache will return the same result.

Shared caches can be quite useful in throughput-oriented applications, particularly multithreaded applications that share data. However, for real-time applications, they can be quite problematic. Tasks executing on different cores can interfere with one another through the shared cache by evicting useful data. In turn, this interference must be reflected in timing analysis, which is used to bound the execution time, e_i, of each task. Because this interference is difficult to analyze and quantify, common practice in multicore real-time

applications is to simply disable all but one processing core. We will discuss later techniques that have been presented to mitigate the effects of this interference.

2.3.1.4 I/O Devices and Accelerators

Modern computing platforms are becoming increasingly *heterogenous*. Special-purpose hardware is designed and built to accelerate specific applications. Common examples of such accelerators include *digital signal processors* (*DSPs*), and *graphics processing units* (*GPUs*). Recently, Google announced that they use a new special-purpose machine-learning co-processor, which they call a *tensor processing unit* (*TPU*) (Jouppi, 2016). The TPU reportedly offers ten times better performance on machine-learning applications, and was used to power Google's AlphaGo, the first artificial intelligence framework to beat the Go world champion (Jouppi, 2016).

Given the diversity of the design of these different accelerators and I/O devices, we focus our discussion on the challenges of interfacing with these devices, and for brevity omit discussion of the complexity of the devices themselves. As depicted in Figure 2.13, I/O devices are connected to the multicore processor itself via an I/O hub. As an example, recent Intel x86 processors are connected to the I/O hub via the *QuickPath Interconnect* (*QPI*), and the I/O devices are connected to the I/O hub via the *PCI Express* (*PCIe*) bus. These buses can be a point of contention as they have a finite bandwidth that may need to be multiplexed among multiple concurrent data transfers.

2.3.2 Mitigating Shared-Hardware Interference

After describing the key components of the architecture of a multicore platform, we now turn our attention to previous work that has identified and proposed approaches to mitigating interference that can arise due to contention for shared hardware components. Quantifying and mitigating the effects of interference in real-time systems has been studied extensively for quite some time, and therefore we limit this discussion to only those works that are most relevant to the contributions of this dissertation.

2.3.2.1 Caches

We begin our discussion of interference caused by shared hardware with caches. Even on a uniprocessor platform, local caches are shared among multiple processes or real-time tasks. However, on such platforms, at most one task may be scheduled at a time, which limits the number of tasks that can access the cache

at any time to at most one. However, when one task τ_l is preempted by a higher-priority task τ_h, τ_h may evict the previously cached data of τ_l. When τ_l resumes execution, subsequent accesses to previously cached data will miss, inflating the execution time of the task. Quantifying the cost of these cache misses is called *cache-related preemption delay* (*CRPD*) *analysis*. For brevity, we refer the reader to (Altmeyer et al., 2012) for further discussion on CRPD analysis for local caches.

Shared caches, which can be concurrently accessed by tasks executing on different processors, are a more challenging issue. Two concurrently executing tasks may thrash one another with *cross-core cache evictions*, thereby negating the benefit of the cache in the first place. This issue in particular has been identified by the FAA in a recent position paper on the use of multicore processors in avionics (Certification Authorities Software Team (CAST), 2014). For these reasons, as mentioned earlier, the current *de facto* standard practice for multicore processors in industrial real-time applications is to simply disable all but one processing core, so as to eliminate the possibility of cross-core evictions.

One general approach to mitigate cache interference from other tasks is *cache partitioning* (Kirk, 1989; Bui et al., 2008; Altmeyer et al., 2014). Under cache partitioning, tasks are allocated memory in such a way so as to minimize or eliminate the possibility of interference. Cache partitioning can be applied to local caches on uniprocessors to eliminate cache-related preemption delays by partitioning all tasks to disjoint cache sets (rows in Figure 2.16). Such allocation can also be automatically determined by the compiler (Mueller, 1995). At the OS level, *page coloring* (Kessler and Hill, 1992) can be used to allocate memory pages that realize cache partitioning. Under page coloring, pages that map to the same cache sets are assigned the same color, and differently colored pages do not conflict in the cache.

Cache partitioning can also be applied to eliminate cross-core cache evictions in a shared cache. For a partitioned scheduler, the cache can be partitioned among the processors, thereby eliminating the possibility of cross-core cache evictions. This partitioning strategy effectively renders a shared cache as several smaller local caches. The CRPD analysis mentioned previously could be used to quantify the effects of cache-related preemption delays within each cache partition.

Cache lockdown. The previously discussed cache-partitioning techniques are *set based*. Using additional hardware available on some platforms, it is also possible to partition based on the *ways* of the cache, or the columns in Figure 2.16. To support way-based cache partitioning, the cache must support some form of *cache lockdown*. Cache lockdown allows software to specify which ways of the cache may be evicted, and which

are locked down, or prevented from being evicted. This feature can be implemented in a number of different ways. For example, on some platforms, a global lockdown register specifies which ways are locked and which are not. On others, a lockdown register exists per core. In the later case, way-based cache partitioning can be realized by setting the cache-lockdown registers to all be disjoint, such that each core is partitioned to a subset of the available ways. This lockdown hardware was used recently by Mancuso et al. (2013) in a technique they called *colored lockdown* to permanently lock the most commonly accessed pages into the cache.

Cache partitioning is static in that partitions are determined offline and remain throughout the execution of the task system. However, in more recent work, several approaches have been presented that seek to more dynamically manage the allocation of the cache so as to realize better cache performance, while still providing cache isolation (Ward et al., 2013b; Xu et al., 2016; Kim et al., 2013). These techniques seek to provide larger cache allocations, which in turn result in smaller execution-time bounds, while also limiting or eliminating cache interference.

2.3.2.2 Memory

Memory references that do not hit in the cache are deferred to main memory. However, as described previously, there are a number of different factors that affect the latency of memory requests. Furthermore, interference from other memory references coming from other cores may also affect the latency of such requests. Much recent research seeks to minimize the effect of such interference to increase the timing predictability of memory references.

Similar to cache partitioning, the memory banks (recall from Figure 2.13) may also be partitioned, such that there is no bank interference across cores. Yun et al. (2014) presented a framework called PALLOC that allows for bank-aware memory allocations to minimize or eliminate interference through bank isolation. PALLOC only affects memory allocations, and therefore does not aim to address memory interference at runtime. Yun et al. (2013) presented a tool called MemGuard, which can be used to dynamically limit the memory bandwidth tasks receive at runtime. MemGuard leverages the hardware performance counters and suspends tasks that have used too much memory bandwidth in a given time interval, thereby freeing memory bandwidth to other tasks. MemGuard is best suited to SRT applications, though the authors note that by disabling some optimistic features, it may be possible to apply in HRT applications as well.

does not make performance guarantees, and is therefore best suited to SRT applications.

Memory controllers have also been a focus in work on more predictable memory systems (*e.g.*, (Krishnapillai et al., 2014)). Real-time memory controllers offer the potential of more predictable memory-access latency, and more effective use of the row buffer. In this dissertation, we focus more on software-based techniques to improve the platform utilization of more widely available commercial off-the-shelf (COTS) hardware platforms, and therefore discussion of special-purpose memory controllers, which are built into the hardware platform, are beyond the scope of this dissertation.

In order to "hide" memory latency, as well as more carefully arbitrate access to memory resources, Wasly and Pellizzoni (2014) presented an algorithm that schedules both processor and memory accesses. In that work, the authors assume a two-phase memory model, in which during the first phase, the job loads all of its data from memory into a cache, and during the second phase it executes on the processor. In this model, it is assumed that during the second phase the task does not access main memory. They assume the existence of a *direct memory access* (*DMA*) engine, which facilitates background memory transfers that do not occupy processor cycles.

2.3.2.3 GPUs and Accelerators

GPUs and other hardware accelerators may be used by multiple tasks or processors. Similar to the previously discussed resources, such sharing can also cause interference that can affect the timing behavior of the use of such resources. How access to such resources is controlled is often specific to the underlying hardware being controlled in order to better utilize the hardware. In this section, we briefly highlight common techniques, and discuss a few examples.

Gai et al. (2003), who developed the previously discussed MSRP multiprocessor locking protocol, considered the use of that locking protocol to control access to hardware resources on a *system-on-chip* (*SOC*). The hardware platform they considered was a two-core system. They considered an automotive power-train control system with I/O channels and analogue-to-digital (A/D) converters, and applied the MSRP to synchronize access to these hardware resources, in addition to other memory resources. Locking protocols have also been used to arbitrate access to other hardware resources as well. Elliott et al. (2013) presented GPUSync, a framework that applies a synchronization-based approach to control access to GPUs and the hardware elements therein. Locking protocols are particularly amenable to both of these motivating examples due to the nature of the hardware being controlled. In both of these examples, tasks require exclusive access

to the hardware to prevent interference that could negatively impact either the logical correctness, or the timing behavior of the hardware.

Others have taken a more scheduling-oriented approach to managing GPU resources (Kato et al., 2011a,b, 2012). Fundamentally, these approaches seek to resolve contention for hardware resources on the GPU. However, the implementation details associated with such management may differ greatly. GPUSync provides an additional layer of abstraction on top of the closed-source GPU driver. This design reduces contention for GPU resources at the driver level, thereby encouraging the driver to make more predictable resource-allocation decision more amenable to real-time analysis. In other work, such as RGEM (Kato et al., 2011a) and Gdev (Kato et al., 2012), the authors develop their own open-source GPU drivers through reverse engineering, which allows the driver to more predictably manage access to GPU resources. For a complete discussion of GPUs in real-time systems, we refer the interested reader to (Elliott, 2015).

2.4 Chapter Summary

In this chapter, we have formalized the real-time task and resource models that will be considered in this dissertation. We also discussed relevant real-time scheduling theory, as well as prior work on real-time synchronization. We have also discussed the architecture of modern multicore hardware platforms, and how contention for features in the hardware architecture can cause interference that can cause adverse time-performance effects. Several previous techniques for managing such contention were presented.

CHAPTER 3: Real-Time Nested Locking Protocol (RNLP)[1]

This chapter presents the *Real-Time Nested Locking Protocol* (*RNLP*), the first real-time multiprocessor locking protocol that both algorithmically and analytically supports fine-grained locking. Fine-grained locking allows groups of resources to be decomposed into smaller elements that are individually acquired, thereby allowing for the possibility of non-conflicting requests being satisfied concurrently. Such concurrency can reduce the blocking time for other waiting requests, thereby allowing for improved schedulability.

The RNLP 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. As a result, the RNLP is actually a family of efficient locking protocols. Under analysis assumptions used previously (Brandenburg, 2011; Brandenburg and Anderson, 2010a, 2011, 2014), all of the RNLP variants are asymptotically optimal.

Unlike group locks, the RNLP does not require resources to be statically grouped before execution. Instead, the RNLP uses either a partial ordering on resource acquisitions, which is a common assumption in practice to ensure that deadlock is impossible, or *dynamic group locks*, which allow for a collection of resources (a subset of a larger group) to be atomically locked.

Organization. The remainder of this chapter is organized as follows. We begin in Section 3.1 by clarifying assumptions specific to this chapter. In Section 3.2, we describe the basic architecture of the RNLP, which is composed of two components. In Section 3.3 and Section 3.4, we describe and analyze each of these components and how these components can be instantiated to minimize pi-blocking. In Section 3.5, we describe extensions of the RNLP that support dynamic group locks. In Section 3.6, we present fine-grained

[1]Contents of this chapter previously appeared in preliminary form in the following papers:
Ward, B. and Anderson, J. (2012). Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 223–232.
Ward, B. and Anderson, J. (2013). Fine-grained multiprocessor real-time locking with improved blocking. In *Proceedings of the 21st Conference on Real-Time Networks and Systems*, pages 67–76.

blocking analysis for the RNLP. Finally, we conclude in Section 3.7. The RNLP presented in this chapter is later considered as part of a larger experimental study presented in Chapter 4.

3.1 Resource Model

In this chapter, we assume a simplified resource model based on that described in Section 2.2. Specifically, we assume a system containing n_r shared mutex resources $\mathcal{L} = \{\ell_1, \ldots, \ell_{n_r}\}$ such as shared data objects or I/O devices. Notably, however, we employ a k-exclusion lock to realize the locks that we construct. Unlike many prior algorithms, we make no distinction between local and global resources, as the RNLP can be applied to both. Furthermore, lock nesting is allowed, in accordance with the following assumptions.

3.1.1 Nesting

The classic means of supporting fine-grained locking is through *lock nesting*, in which a resource-holding job issues another *nested request*. If J_i holds no resources when it issues a request, then the request is 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 issue 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 Figure 3.1 (here, ℓ_a is acquired first, but ℓ_b is released last). We let $wait(J_i,t)$ denote the resource for which J_i is waiting at time t if any. The maximum number of outermost requests J_i issues is given by N_i. The maximum duration of the k^{th} outermost critical section of J_i is $L_{i,k}$. For notational convenience, the second subscript in all request-related notation may be omitted when it is inconsequential, *e.g.*, $\mathcal{R}_i$ has a critical-section length of L_i.

Definition 3.1. A pi-blocked (s-blocked, s-aware, or s-oblivious) job J_i *makes progress* if at least one resource-holding job that J_i is blocked by is scheduled.

As we will discuss later, there are a few ways that J_i can be blocked by other requests.

As described in Chapter 2, 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 Figure 3.1. 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

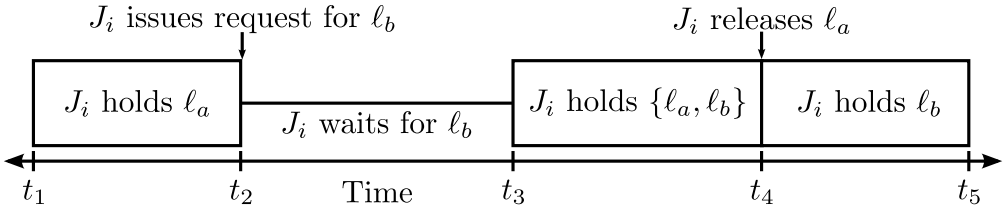


Figure 3.1: Illustration of a job J_i's outermost critical section. At time t_1, J_i acquires resource ℓ_a. At time t_2, J_i issues a nested resource request for ℓ_b, and is blocked during the interval $[t_2, t_3)$. At time t_4, J_i releases ℓ_a. J_i's outermost critical section spans from t_1 to t_5 when J_i no longer holds any shared resources.

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 critical-section execution time are the same when the RNLP is compared with single-resource locking protocols.

In order to prevent deadlock, we assume a strict (irreflexive) partial order, $\prec$, on the set of resources $\mathcal{L}$ such that a job holding resource ℓ_b cannot issue a nested request for ℓ_a if $\ell_a \prec \ell_b$. In addition to preventing deadlock, this ordering can be used in blocking analysis to reduce pi-blocking bounds.

3.1.2 Dynamic Group Locks

An alternative means of supporting fine-grained locking instead of lock nesting is a mechanism called a *dynamic group lock* (*DGL*). Lock nesting allows for multiple resources to be held concurrently, but resources must be acquired individually. In contrast, under coarse-grained group locking, tasks acquire a single lock on the entire set of resources in one operation; however, this set may include far more resources than actually needed. Dynamic group locking merges these two means of allowing for jobs to concurrently access multiple resources. DGLs extend the notion of fine-grained locking by allowing a request to specify a *set* of resources to be locked. In this way, DGLs provide better concurrency than group locks, and the potential for lower overheads than nested locking when the set of resources to lock in a nested fashion is known *a priori*. As we will show, DGLs have the same analytical worst-case blocking as nested locking in the RNLP under the analysis techniques we present herein.

Note that DGLs can be supported in addition to nested locking, that is, tasks can issue nested DGL requests. Also, with the RNLP extended to support DGLs, individual nested requests can still be performed like before. Such nesting is necessary, for example, when the resource-access sequence is determined by executing conditional statements. Such nesting may be preferable to improve response times, as tasks

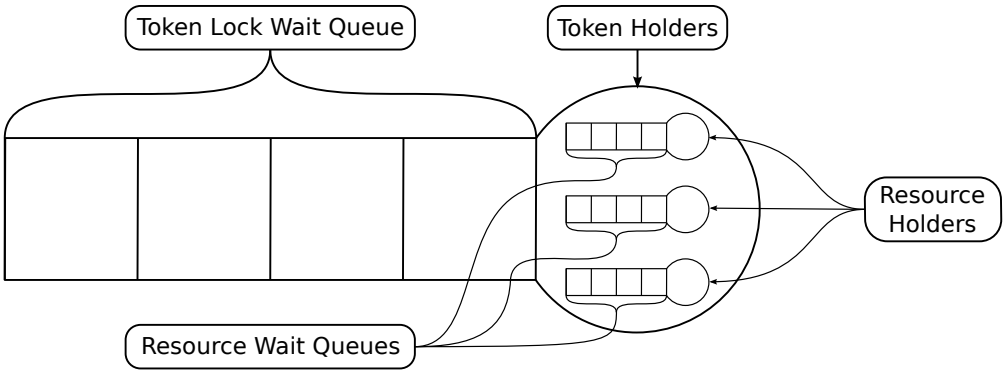


Figure 3.2: Architecture of the RNLP.

are likely blocked by fewer requests. However, even if the set of resources that will actually be required is unknown—for example, when the resource-access sequence is determined by executing conditional statements—DGLs can still be employed to request all resources that might possibly be accessed, to reduce locking overheads.

For ease of exposition, we first present the RNLP assuming lock nesting and no DGLs. Later, in Section 3.5, we extend the rules of the RNLP to support DGLs, and show the parallels in their analyses.

3.2 RNLP Architecture

The RNLP is composed of two components, a k-exclusion *token lock*, and a *request satisfaction mechanism* (*RSM*), as shown in Figure 3.2. The token lock restricts the number of jobs that can have an incomplete resource request to the number of tokens $\mathcal{T}$, while the RSM determines when requests are satisfied. As depicted in Figure 3.3, in order for a job to issue a resource request, it must first acquire a token through the token lock. The $\mathcal{T}$ 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 ($\mathcal{T}$ is either n or m in all considered variants), and RSMs can be paired to yield an asymptotically optimal locking protocol supporting nested requests.

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 Section 3.4).

T1 There are at most $\mathcal{T}$ token-holding jobs at any time.

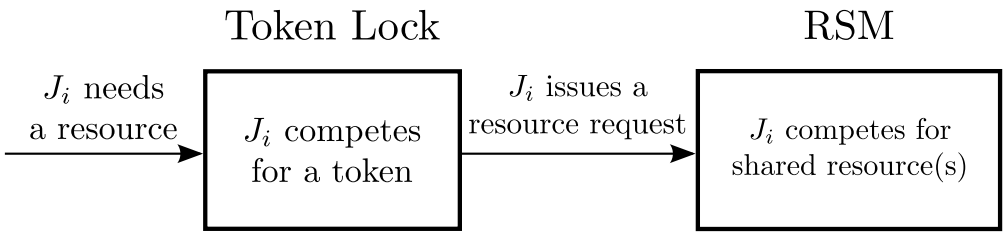


Figure 3.3: Depiction of the the stages of a request in the RNLP.

T2 If a job is pi-blocked waiting for a token, then it makes progress.

Progress can be ensured by elevating the effective priority of a token-holding job through a progress mechanism such as priority boosting, inheritance, or donation, as described in Section 2.2.

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 $\mathcal{T}$. The timestamp of token acquisition is stored for each job J_i, and denoted $ts(J_i)$.[2] 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. Ordering request queues by timestamp 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 3.1) are common to all RSMs.

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 $ts(hd(a)) < ts(J_i)$.

Q4 When a job J_i issues a request for resource ℓ_a, it is enqueued in RQ_a in increasing timestamp order.[3]

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.

[2] $ts(J_i)$ 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.

[3] We assume the acquisition of a token and subsequent enqueueing into the wait queue RQ_a of the resource requested in the outermost request occur atomically.

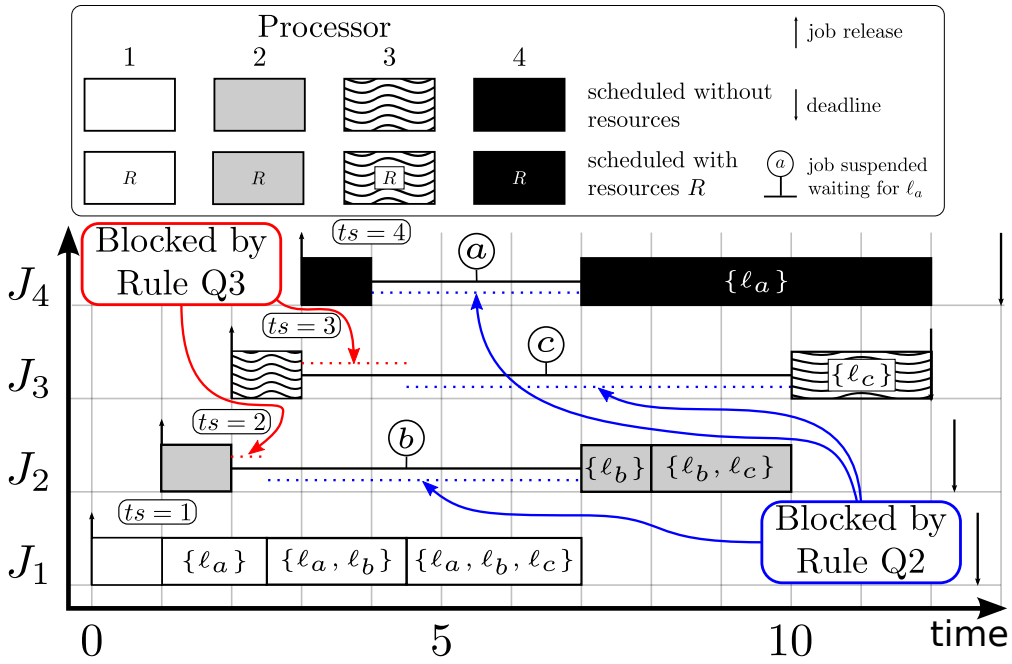


Figure 3.4: Illustration of Example 3.1.

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. Also, notably, Rule Q3 serves a similar purpose to the priority ceiling in ceiling-based protocols such as the PCP and SRP. In those protocols, the system ceiling delays some lock acquisitions so as to prevent subsequent deadlock, or otherwise adverse blocking conditions. In this way, the priority ceiling is a non-greedy rule for lock acquisitions. Similarly, Rule Q3 also defers the lock acquisitions of some requests in a non-greedy fashion, so as to also prevent subsequent adverse blocking. Effectively, Rule Q3 ensures that resources are "reserved" for earlier-issued requests that may access them.

Example 3.1. To illustrate these rules, we present an example, which is depicted in Figure 3.4. Consider a system scheduled according to G-EDF with three shared resources, ℓ_a, ℓ_b, and ℓ_c, and $m = 4$, and a total order on resources by index (*i.e.*, $\ell_a \prec \ell_b \prec \ell_c$). As shown in Figure 3.4, each job J_i acquires a token at time $t = i$, and thus by Rule Q1, $ts(J_i) = i$. Furthermore, $\mathcal{F}_1 = \ell_a$, $\mathcal{F}_2 = \ell_b$, $\mathcal{F}_3 = \ell_c$, and $\mathcal{F}_4 = \ell_a$. 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. $\Diamond$

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

$$DB(J_i,t) = \{J_k \in \mathrm{RQ}_{wait(J_i,t)} \mid ts(J_k) < ts(J_i)\}. \tag{3.1}$$

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

$$TB(J_i,t) = TB^n(J_i,t), \tag{3.2}$$

where

$$TB^k(J_i,t) = TB^{k-1}(J_i,t) \bigcup_{J_x \in TB^{k-1}(J_i,t)} DB(J_x,t), \tag{3.3}$$

$$TB^0(J_i,t) = DB(J_i,t). \tag{3.4}$$

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

Example 3.2. To illustrate transitive blocking we consider the schedule shown in Figure 3.5, which pertains to the same task system as in Example 3.1. 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. $\Diamond$

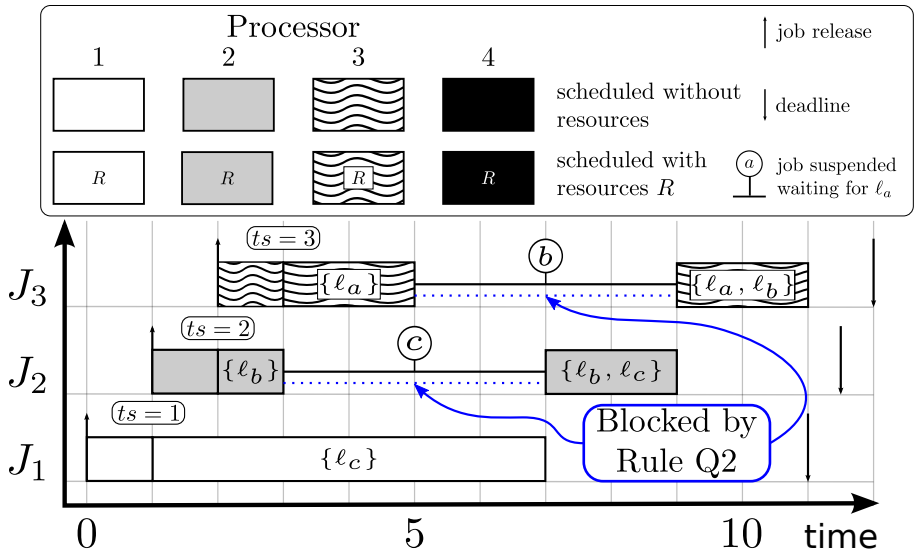


Figure 3.5: Illustration of Example 3.2.

Reconsidering Example 3.1, at time $t \in [3, 4.5)$ in Figure 3.4, 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

$$IB(J_i,t) = \{J_k \in \mathrm{RQ}_a \mid \ell_a \prec wait(J_i,t) \ \wedge ts(J_k) < ts(J_i)\}. \tag{3.5}$$

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

$$B(J_i,t) = TB(J_i,t) \cup IB(J_i,t). \tag{3.6}$$

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

Lemma 3.1. *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 Section 3.3 for a number of individual RSMs.

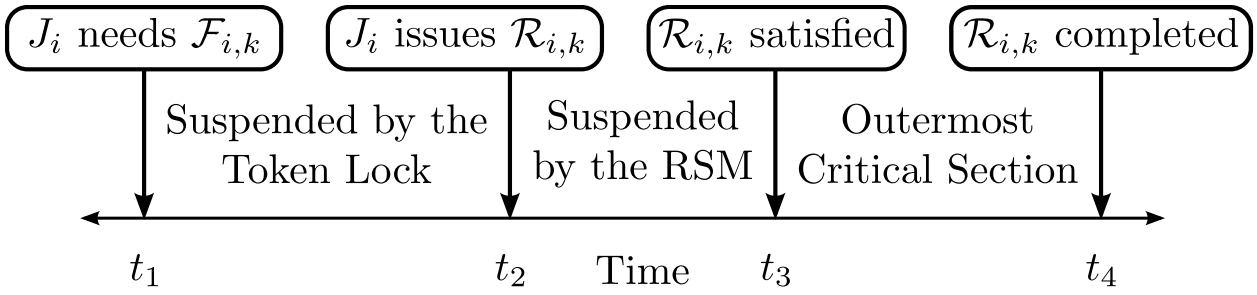


Figure 3.6: Phases of a resource request in the RNLP.

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 Figure 3.6.

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 Section 3.4 we consider worst-case pi-blocking under various token locks.

Theorem 3.1. *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 $(\mathcal{T} - 1)L_{\max}$.*

Proof. Property P1 ensures that if a job is pi-blocked, it makes progress. By Lemma 3.1, a job can never be pi-blocked by a job with a later timestamp. By Property T1, there are at most $\mathcal{T} - 1$ jobs with earlier timestamps. Thus, a job can be pi-blocked in any RSM for at most $\mathcal{T} - 1$ outermost critical sections, each of length at most $L_{\max}$. □

3.3 Request Satisfaction Mechanisms

In this section, we describe five RSMs that rely on different progress mechanisms, the spin RSM (S-RSM), boost RSM (B-RSM), restricted segment boosting RSM (RSB-RSM), inheritance RSM (I-RSM), and donation RSM (D-RSM).

3.3.1 Spinning

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 (Block et al., 2007; Brandenburg, 2011). 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 non-preemptively.

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 $\mathcal{T} = m$ tokens, c from each cluster. Additionally, non-preemptivity can cause jobs that are not currently utilizing the locking protocol to be pi-blocked. This progress-mechanism-related pi-blocking must be analyzed and incorporated into schedulability analysis.

Lemma 3.2. *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. □

3.3.2 Unrestricted Boosting

The B-RSM can be applied in partitioned-scheduled systems in which waiting is realized by suspending instead of spinning. Under the B-RSM, progress is ensured by boosting the priority of (some) resource-holding jobs. 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 effective priority of the resource-holding job with the earliest timestamp, if any, in each partition is boosted above that of all other jobs in its partition.

This rule allows for any value of $\mathcal{T}$, as it ensures that no more than one job can be priority boosted concurrently. This rule is similar to the original partitioned-only version of the FMLP^+ (Brandenburg, 2011).

Rule B1 gives rise to another form of blocking, which we call *processor blocking*. Because each processor can schedule at most one resource-holding job at a time, requests may be forced to block until a processor is available. Given this observation, coupled with Definition 3.1, the following lemma follows immediately.

Lemma 3.3. *The B-RSM for partitioned-scheduled systems in which waiting is realized by suspending ensures Property P1.*

Proof. Let J_i be a pi-blocked job within the RSM, and therefore blocked, directly or by a processor, at least one other earlier-timestamp request. By Rule B1, the earliest-timestamp resource-holding job on each processor is scheduled, therefore ensuring progress for J_i. □

We note that Rule B1 has been adapted from its original presentation (Ward and Anderson, 2012) to be specific to partitioned-scheduled systems only. In its original form, the B-RSM supported clustered and globally scheduled systems, albeit suboptimally. Since the original work on the RNLP, Brandenburg (2014) has presented *restricted segment boosting* (*RSB*) (recall from Chapter 2), which can be used under partitioned, clustered, and global scheduling to achieve an optimal locking protocol in the FMLP$^+$. As such, we next present the RSB-RSM, based off of this new progress mechanism, and encourage its use for clustered and global scheduling.

3.3.3 Restricted Segment Boosting

When Brandenburg (2014) presented the FMLP$^+$ and RSB, he also briefly described how RSB could be used in the context of the RNLP to support fine-grained locking. In the following discussion, we describe more concretely how this is accomplished through the presentation of the RSB-RSM. The RSB-RSM is designed for partitioned, clustered, and globally scheduled systems.

Under RSB, each job is decomposed into disjoint segments: *independent* segments, in which the job is not engaged in the locking protocol at all, and *request* segments, in which the job is engaged in the locking protocol by being either blocked, or holding resource(s). Importantly, in the context of nested locking the request segment begins when the job engages the locking protocol to request a resource, and ends when it completes its outermost critical section.

RSB1 Let J_b be the resource-holding job (if any) with the earliest request-segment start time, denoted t_b, in its cluster. J_b is priority boosted such that it is scheduled.

RSB2 Let $C(J_b)$ be the (at most) $c-1$ highest-priority ready jobs with priority higher than J_b and segment start times before t_b. All jobs in $C(J_b)$ are co-boosted, and therefore also scheduled.

Recall the illustration of RSB in Example 2.7 and Figure 2.9. Note that Rule RSB1, similar to Rule B1, gives rise to processor blocking, as at most one request per cluster is boosted at a time. Therefore, similarly to the B-RSM, we have the following lemma.

Lemma 3.4. *The RSB-RSM for partitioned, clustered, and globally scheduled system in which waiting is realized by suspending ensures Property P1.*

Proof. Let J_i be a pi-blocked job within the RSM, and therefore blocked, directly or by a processor, by at least one other request with an earlier request-segment start time. By Rule RSB1, the job with the earliest request-segment start time on each processor is scheduled, therefore ensuring progress for J_i. □

Notably, co-boosting via Rule RSB2 is not necessary to ensure resource-holder progress. However, as will be discussed later, Rule RSB2 is essential to achieve asymptotically optimal s-aware pi-blocking under clustered scheduling.

3.3.4 Inheritance

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.

Example 3.3. To motivate the design of the I-RSM, consider again Example 3.1, illustrated in Figure 3.4. Suppose at time $t = 6$ there exist m jobs (not shown) that do not utilize the locking protocol and 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

$$ic(J_i,t) = \underset{J_k \in B(J_i,t)}{\arg\min}\, ts(J_k). \tag{3.7}$$

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.

$$ICS(J_c,t) = \{J_i \mid \exists T_i \in \tau, ic(J_i,t) = J_c\}. \tag{3.8}$$

Example 3.3 (continued). 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:

$$p(J_i,t) = \max_{J_k \in \{J_i\} \cup ICS(J_i,t)} p(J_k,t). \tag{3.9}$$

Example 3.3 (continued). $ICS(J_1,6) = \{J_2,J_3,J_4\}$. Of these jobs J_3 has the highest priority, and thus J_1 inherits the priority of J_3 at time $t = 6$.

Lemma 3.5. *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 3.5 ensures that there are never two jobs executing with the same effective priority, which is equivalent to a task having two threads. This would break the assumptions of the sporadic task model, and thus the resulting system would not be analyzable using existing schedulability tests.

Lemma 3.6. *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 3.1, 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 3.7. *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 3.6, $ic(J_i,t)$ 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 $\mathcal{T}$. Depending upon the scheduler, analysis type, and token lock, $\mathcal{T}$ can be chosen to allow for increased parallelism or decreased worst-case pi-blocking. This issue is considered in Section 3.4.

3.3.5 Donation

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 effectively compared across clusters, as discussed in Chapter 2. 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 (Brandenburg and Anderson, 2011).

Example 3.4. In Example 3.1, 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 $\mathcal{T} \leq m$. This property is satisfied by a token lock employing priority donation such as the C-OMLP.

Lemma 3.8. *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. □

Analysis	Scheduler	Token Lock	$\mathcal{T}$	RSM	PMR Pi-blocking	Per-Request Pi-blocking
spin	Any	TTL	m	S-RSM	$O(m)$	$(m-1)L_{\max}$
s-aware	Partitioned	TTL	n	B-RSM	$O(n)$	$(n-1)L_{\max}$
	Clustered	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
	Global	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
		TTL	n	I-RSM	$O(n)^{\dagger}$	$(n-1)L_{\max}$
s-oblivious	Partitioned	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Clustered	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Global	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
		R^2DGLP	m	I-RSM	0	$(2m-1)L_{\max}$

† Applicable only under certain schedulers as discussed in Section 3.4.3.

Table 3.1: 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 2.1.

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.

3.4 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 3.1.

3.4.1 Spin $\mathcal{T}$-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, $\mathcal{T} = m$ under the TTL. For the remainder of this subsection, we assume $\mathcal{T} = m$.

Lemma 3.9. *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 3.2. *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 3.3. *The maximum duration of s-blocking per-request in the S-RSM is* $(m-1)L_{\max}$.

Proof. Follows from Theorem 3.1 and $\mathcal{T} = m$. □

3.4.2 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 (2011). 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 $\mathcal{T} = m$, 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 3.10. *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 (Brandenburg and Anderson, 2011) states that priority donation ensures that a resource-holding job is always scheduled. Therefore 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 3.8, 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 3.4. *The maximum duration of s-oblivious pi-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 (Brandenburg and Anderson, 2011), a token-holding job has sufficient priority to be scheduled. By Lemma 2 of (Brandenburg and Anderson, 2011) and the assumption that $\mathcal{T} = m$, 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 $(\mathcal{T}-1)L_{\max}$ time by Theorem 3.1. Since we assume $\mathcal{T} = m$, the maximum duration of pi-blocking caused by priority donation is thus $mL_{\max}$. □

Theorem 3.5. *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 3.1 and $\mathcal{T} = m$. □

Theorems 3.4 and 3.5 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 C-OMLP, as seen in Table 2.1. In combination with the $\Omega(m)$ pi-blocking lower bounds for s-oblivious and s-blocking discussed in Chapter 2, these results prove that the RNLP is also asymptotically optimal in these cases. 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.

3.4.3 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 $\mathcal{T} = n$ to allow for maximal concurrency, thus the TTL is used. From the TTL, we have the following lemma.

Lemma 3.11. *The TTL satisfies Properties T1 and T2.*

Proof. By the assumption that $\mathcal{T} = n$, and the fact that each job can hold at most one token, Property T1 is satisfied. Because there is a token per job, there is no blocking waiting for a token, and therefore Property T2 holds vacuously. □

Next we consider pairing the TTL with several of the RSMs previously presented.

B-RSM. We pair the TTL with the B-RSM under partitioned scheduling.

Theorem 3.6. *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 3.1 and $\mathcal{T} = n$. □

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

Theorem 3.7. *Let* n_p *be the number of tasks assigned to* J_i*'s partition. The worst-case s-aware progress-mechanism-related pi-blocking of job* J_i *caused by the boosting of other jobs in the B-RSM is* $(n_p - 1)L_{\max}$.

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^+ (Brandenburg, 2011) because both the B-RSM and the FMLP^+ employ unrestricted priority boosting. Thus, the bound follows directly from the bound for the FMLP^+ in Theorem 6.4 of (Brandenburg, 2011). □

Note that the progress-mechanism-related and per-request pi-blocking are both $O(n)$ (Theorems 3.7, and 3.1, respectively) for the pairing of the TTL and the B-RSM. Given the s-aware lower bound of $\Omega(n)$ discussed in Chapter 2, the RNLP with the TTL and B-RSM is asymptotically optimal for partitioned systems.

RSB-RSM. The RSB-RSM can be used to construct optimal RNLP variants under partitioned, clustered, and global scheduling when paired with the TTL and $\mathcal{T} = n$. Notably, when using the TTL, the timestamp of token acquisition is also the request-segment start time, because there is no blocking for tokens.

Theorem 3.8. *On partitioned, clustered, and globally scheduled systems, the maximum duration of s-aware pi-blocking per outermost resource request is $(n-1)L_{\max}$ under the RSB-RSM and TTL.*

Proof. In the worst case, all requests execute serially by Rule RSB1, because only one request can be boosted at a time. Thus, the bound follows directly from Theorem 3.1 and $\mathcal{T} = n$. □

The co-boosting required by Rule RSB2, which is derived from the FMLP$^+$ (Brandenburg, 2014), is used to ensure smaller progress-mechanism-related pi-blocking bounds. In particular, co-boosting ensures the following property.

Lemma 3.12. ***(Brandenburg, 2014, Lemma 13)*** *Let $[t_0, t_1]$ denote an independent segment of job J_i. During $[t_0, t_1]$, J_i incurs s-aware pi-blocking for the cumulative duration of at most one critical section per each other task in its cluster.*

Theorem 3.9. *The worst-case s-aware progress-mechanism-related pi-blocking per job caused by the TTL and the RSB-RSM under partitioned, clustered, or global scheduling is $O(n)$.*

Proof. Consider a job J_i in cluster k. Let n_k denote the number of tasks in cluster k. By Lemma 3.12, each independent segment may experience $n_k - 1$ critical sections of progress-mechanism-related pi-blocking. The number of independent segments per job J_i is a constant $(N_i + 1)$, and therefore the total s-aware progress-mechanism-related pi-blocking is $O(n)$. □

Priority inheritance. 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 (Brandenburg, 2011), as was discussed in Chapter 2. It is therefore preferable in most cases to use RSB under s-aware analysis instead of priority inheritance. However, under G-FP 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 JLFP schedulers in which each task's *relative priority point* does not exceed its period, priority inheritance yields an asymptotically optimal locking protocol.

Theorem 3.10. *The worst-case s-aware progress-mechanism-related pi-blocking per job caused by the TTL and I-RSM under either G-FP 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 (Brandenburg, 2011), the maximum s-aware pi-blocking caused by priority inheritance is $O(n)$ for G-FP and constrained, fixed priority-point global schedulers. □

Theorem 3.11. *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 3.1 and $\mathcal{T} = n$. □

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.

3.5 Dynamic Group Locks

In the preceding discussion, fine-grained locking was supported through nested locking. Next, we present *dynamic group locks* (*DGLs*), as an alternative technique to support fine-grained locking.

3.5.1 Dynamic Group Lock Rules

When using DGLs, a job may issue a request for a *set* of resources, denoted $\mathcal{D}_i$. DGLs can be integrated into the RSM rules of the RNLP with only minor modifications. As such, the rest of the mechanics of the RNLP remain, including the need to acquire a token before issuing the DGL request to the RSM. Once J_i has acquired its token, its request is enqueued in the resource queue for each resource in $\{\text{RQ}_a | \ \ell_a \in \mathcal{D}_i\}$. The DGL request is satisfied when it has acquired all resources in $\mathcal{D}_i$, at which point in time J_i is made ready. This can be expressed by replacing Rules Q3 and Rule Q4 with the following more general rules:

DGL1 When a job J_i issues a request $\mathcal{R}_i$ for a set of resources $\mathcal{D}_i$, for every resource $\ell_a \in \mathcal{D}_i$, J_i is enqueued in RQ_a in timestamp order.

DGL2 A job J_i with an outstanding resource request $\mathcal{R}_i$ for a subset of resources $\mathcal{D}_i \subseteq \mathcal{L}$ acquires all resources in $\mathcal{D}_i$ when **(i)** $\mathcal{R}_i$ is the head of every resource queue associated with a resource in $\mathcal{D}_i$, and **(ii)** there is no resource $\ell_a \in \{\ell_x \mid \ell_x \in \mathcal{L} \wedge \ell_y \in \mathcal{D}_i \wedge \ell_x \prec \ell_y\}$ such that $ts(hd(a)) < ts(J_i)$.

Note that if there are no nested requests and fine-grained locking is supported only via DGLs, condition (ii) of Rule DGL2 need not be checked; earlier-issued requests need not "reserve" resources for future nested requests. For example, if $\mathcal{R}_i$ held ℓ_a and may later issue a nested request for ℓ_b, condition (ii) of Rule DGL2 would prevent a later issued request $\mathcal{R}_k$ from acquiring ℓ_b. If instead, $\mathcal{R}_i$ issued a DGL request with $\mathcal{D}_i = \{\ell_a, \ell_b\}$, and never requested any other resources, then condition (i) of Rule DGL2 is sufficient to ensure that a request is never blocked by another later-issued request. Indeed, in the case of DGLs only, there is no need for a resource ordering at all. This greatly simplifies the RNLP implementation. The timestamp-ordered queues become simple FIFO queues, and there is no need for jobs to "reserve" a position in any queue, or "cut ahead" of later-issued requests. This is due to the fact that all enqueueing due to one request is done atomically. Thus, in this case, not only is the number of locking-protocol invocations reduced, but the execution time of any one call is likely lessened as well.

3.5.2 Dynamic Group Lock Analysis

This modified version of the RNLP that support DGLs has the same worst-case blocking bounds as those previously discussed for nested locking. Intuitively, such bounds do not change because a DGL request enqueues in multiple resource queues atomically when it is issued, instead of enqueueing in a single queue and essentially "reserving" slots in other queues for potential future nested requests. In the worst case, the set of blocking requests is the same in either case.

Example 3.5. Consider a request $\mathcal{R}_i$ that within its outermost request may access ℓ_a and ℓ_b. Using nested locking $\mathcal{R}_i$ first requests ℓ_a and then ℓ_b. In the RNLP, Rule Q3 prevents any request $\mathcal{R}_j$ issued after $\mathcal{R}_i$ from accessing ℓ_b. In this way, $\mathcal{R}_i$ has "reserved" ℓ_b for its nested request. Note that $\mathcal{R}_i$ may still be blocked when it issues its nested request for ℓ_b, but it will only be blocked by earlier-issued requests. Using DGLs, $\mathcal{R}_i$ atomically requests both ℓ_a and ℓ_b and may be blocked by the same set of requests as in the nested case. $\Diamond$

Given the intuition developed in the example, the following discussion of blocking follows analogously to the nested case (3.1)–(3.6).

Let $w(\mathcal{R}_i,t)$ be the set of resources upon which $\mathcal{R}_i$ is waiting at time t. We say J_i is *directly blocked* by all jobs with earlier timestamps in any of the resource queues in which J_i is enqueued.

$$DB(\mathcal{R}_i,t) = \{\mathcal{R}_x \in \mathrm{RQ}_a |\ \ell_a \in w(\mathcal{R}_i,t) \wedge ts(\mathcal{R}_x) < ts(\mathcal{R}_i)\} \tag{3.10}$$

Additionally, it is possible that a job J_i is blocked because a request with an earlier timestamp *could* make a nested request for the resource for which J_i is waiting. As before, this is also *indirect blocking*.

$$IB(\mathcal{R}_i,t) = \{\mathcal{R}_x \in \mathrm{RQ}_a |\ \ell_a \in \mathcal{L} \wedge w(\mathcal{R}_i,t) \cap \mathcal{L}_x \neq \emptyset \wedge ts(\mathcal{R}_x) < ts(\mathcal{R}_i)\} \tag{3.11}$$

Note that if there are no nested requests and fine-grained locking is supported wholly through DGLs, then there is never any indirect blocking.

The set of all jobs that J_i is blocked by at time t is the transitive closure of both direct and indirect blocking.

$$B(\mathcal{R}_i,t) = \bigcup_{\mathcal{R}_x \in DIB(\mathcal{R}_i,t)} DIB(\mathcal{R}_x,t) \tag{3.12}$$

where $DIB = DB(\mathcal{R}_i,t) \cup IB(\mathcal{R}_i,t)$. Note that $DB(\mathcal{R}_i,t) \cup IB(\mathcal{R}_i,t) \subseteq B(\mathcal{R}_i,t)$.

From the definition of $B(\mathcal{R}_i,t)$, similar to Lemma 3.1, we have the following.

Lemma 3.13. *For any request $\mathcal{R}_i$ and any time t, $\forall \mathcal{R}_x \in B(\mathcal{R}_i)$, $ts(\mathcal{R}_x,t) < ts(\mathcal{R}_i,t)$.*

To ensure a bounded duration of pi-blocking, we require that all jobs that are pi-blocked make progress. We thus require that Property P1 be upheld by any progress mechanism employed in an RSM supporting DGLs (Rules DGL2 and DGL1). All of the previous progress mechanisms can be applied directly to DGLs[4] while ensuring Property P1.

We then have the following, similar to Theorem 3.1.

Theorem 3.12. *A job can be blocked by at most $\mathcal{T}-1$ outermost requests within an RSM supporting DGLs (Rules DGL1 and DGL2).*

Proof. By Property P1 a job that is pi-blocked makes progress. By Lemma 3.13, a request can never be blocked by another request with a later timestamp. Because there are at most $\mathcal{T}$ jobs with tokens, a job can be request blocked by at most $(\mathcal{T}-1)$ requests with earlier timestamps. □

[4]The priority inheritance rules must apply the DGL-specific definition of $B(\mathcal{R}_i,t)$ from (3.12) instead.

Lemma 3.13 and Theorem 3.1 parallel Lemma 3.1 and Theorem 3.1, and thus the duration of pi-blocking for DGLs is the same as requesting the set of resources in a nested fashion. This is because the set of requests that block J_i is the same under either policy due to the non-greedy nature of the resource queues.

Progress mechanisms for DGLs. Property P1 can be directly satisfied using priority boosting (unrestricted or restricted) or priority donation under DGLs. However, under the sporadic task model, which most existing schedulability tests assume, two jobs cannot inherit the priority of one job simultaneously. Thus, if priority inheritance is used to ensure progress, then we must be careful to ensure that this property is not violated.

The priority inheritance rule of the RNLP (Rule I1), allows only the earliest timestamp job that blocks a job J_i to inherit J_i's priority. This inheritance rule was defined to allow a job that either transitively or indirectly blocks J_i to inherit J_i's priority, and ensure progress. However, there is a single job with the earliest timestamp that blocks J_i (because timestamps are totally ordered). Thus, a job's priority can be inherited by at most one job at a time, and hence, this inheritance rule also supports DGLs.

3.6 Fine-Grained Blocking Analysis

Our previous blocking analysis was sufficient to show the asymptotic optimality of the RNLP, and indeed produced coarse blocking bounds that could be safely used in practice. In this section, we refine that analysis to be more *fine-grained* in that more information about the length of each critical section, and the resources requested are incorporated into the analysis to further improve the blocking bounds. As the novel contribution of the RNLP that allows for fine-grained locking is the RSM logic, here we focus on request blocking that occurs within the RSM. Later, we will discuss how token blocking and progress-mechanism-related blocking can be bounded using the results of the RSM blocking analysis and the previous pi-blocking analysis techniques (Brandenburg, 2011, 2013b; Wieder and Brandenburg, 2013).

RSM blocking analysis. The analysis of the worst-case blocking for nested locking and DGLs are the same. Let $\mathcal{D}_{i,j,k}$ be the set of resources potentially accessed within the k^{th} critical section of $J_{i,j}$. In the case of nested locking, if ℓ_a is the resource requested by the outermost request, then $\mathcal{D}_{i,j,k} = \{\ell_b \mid \ell_b \in \mathcal{L} \wedge \ell_a \preceq \ell_b\}$. While this may seem pessimistic, it is reflective of the effect of Rule Q3, which can cause indirect blocking for all requests for resources later in the resource order. This has a similar effect to assuming that $\mathcal{R}_{i,j,k}$ may request any resource later in the resource ordering. In the case of DGLs without nesting $\mathcal{D}_{i,j,k}$ is simply the

set of resources requested by $\mathcal{R}_{i,j,k}$. We therefore assume in the remainder of this analysis that all fine-grained locking is supported through DGLs. Consequently, all blocking in the RSM is direct blocking.

By Lemmas 3.1 and 3.13, a request may only be blocked by other requests with earlier timestamps. Therefore, within the RSM, a request may be blocked by at most one outermost request per task (and thus job). For the purpose of analysis, like previous blocking analysis (Brandenburg, 2011), we assume that each job of the same task may issue the same set of requests,[5] and thus $\mathcal{D}_{i,j,k} = \mathcal{D}_{i,j+1,k}$. Therefore, in the remainder of this analysis, we need only consider conflicting requests from one job per task. We therefore omit the job index and let $\mathcal{R}_{i,k}$ denote the k^{th} request of an arbitrary job J_i of τ_i. Similarly, $\mathcal{D}_{i,k}$ is the set of resources requested by $\mathcal{R}_{i,k}$.

Definition 3.2. The *contention set* is the set of all requests that could potentially contend within the RSM.

$$C(\Gamma) = \bigcup_{J_x \in \Gamma} \{\mathcal{R}_{x,1}, \mathcal{R}_{x,2}, \ldots, \mathcal{R}_{x,N_i}\} \tag{3.13}$$

The request under analysis, $\mathcal{R}_{i,k}$, may not be blocked by all requests in $C(\Gamma)$; the contention set may include far more requests than available tokens $\mathcal{T}$. Our goal through this fine-grained blocking analysis is to determine the maximal subset of requests in the contention set that $\mathcal{R}_{i,k}$ may be blocked by in a given schedule.

Example 3.6. Consider a task system with four resource-requesting tasks, each of which issues one request. Then $C(\Gamma) = \{\mathcal{R}_{1,1}, \mathcal{R}_{2,1}, \mathcal{R}_{3,1}, \mathcal{R}_{4,1}\}$. $\Diamond$

Definition 3.3. A request $\mathcal{R}_{i,k}$ may be *directly blocked* by another request $\mathcal{R}_{x,y}$ if $\mathcal{D}_{i,k} \cap \mathcal{D}_{x,y} \neq \emptyset \wedge i \neq x$. Let $db(\mathcal{R}_{i,k}, \mathcal{R}_{x,y})$ be true if $\mathcal{R}_{i,k}$ can be directly blocked by $\mathcal{R}_{x,y}$.

Example 3.6 (continued). Let $\mathcal{D}_{1,1} = \{\ell_a\}$, $\mathcal{D}_{2,1} = \{\ell_a, \ell_b\}$, $\mathcal{D}_{3,1} = \{\ell_b, \ell_c\}$, and $\mathcal{D}_{4,1} = \{\ell_c, \ell_d\}$. Then $db(\mathcal{D}_{1,1}, \mathcal{D}_{2,1})$ is true but $db(\mathcal{D}_{2,1}, \mathcal{D}_{4,1})$ is false.

Note that additional insights pertaining to the exact system configuration can be used to rule out potential direct-blocking relationship. For example, in a spin-based, partitioned-scheduled system, $\mathcal{R}_{i,k}$ cannot be directly blocked by another requests $\mathcal{R}_{x,y}$ in the same partition.

[5] A job with conditional code may not actually issue every request assumed analytically, however, this only reduces blocking. Task models and schedulability analysis for tasks with more interesting per-job resource-access patterns are an interesting subject of future work.

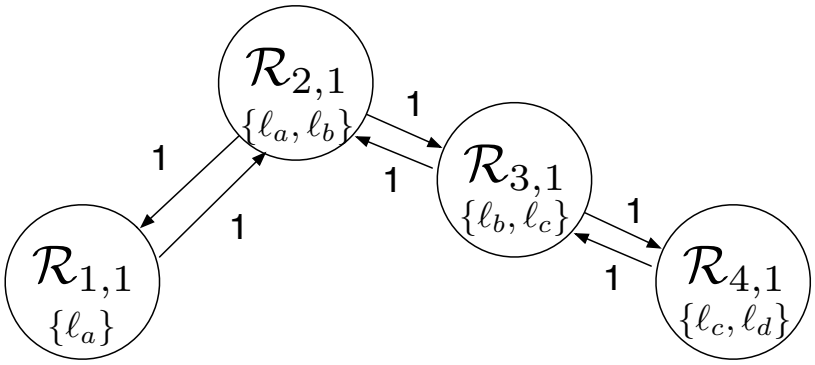


Figure 3.7: Depiction of the blocking graph for Example 3.6 where $L_{x,y} = 1$ for all $\mathcal{R}_{x,y}$.

By our assumption that all fine-grained locking is supported through DGLs, there is no indirect blocking. However, *processor blocking* may occur under both the B-RSM and the RSB-RSM when later-timestamp requests must block while a processor is priority boosted.

Definition 3.4. In the B-RSM and the RSB-RSM, a request $\mathcal{R}_{i,k}$ may be *processor blocked* by another request $\mathcal{R}_{x,y}$ if $\mathcal{R}_{i,k}$ and $\mathcal{R}_{x,y}$ are in the same cluster. Let $pb(\mathcal{R}_{i,k}, \mathcal{R}_{x,y})$ be true if $\mathcal{R}_{i,k}$ may be *processor blocked* by $\mathcal{R}_{x,y}$, and false otherwise.

For simplicity, let $ab(\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) = db(\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) \cup pb(\mathcal{R}_{i,k}, \mathcal{R}_{x,y})$ for the B-RSM and RSB-RSM, otherwise $ab(\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) = db(\mathcal{R}_{i,k}, \mathcal{R}_{x,y})$.

Definition 3.5. Let $G(\Gamma) = (V, E)$ be a *blocking graph*, which encodes all possible blocking.

$$
\begin{aligned}
&G = (V, E), \\
&V = C(\Gamma) \\
&E = \{(\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) \mid ab(\mathcal{R}_{i,k}, \mathcal{R}_{x,y})\}.
\end{aligned}
$$

The *edge weights* in the blocking graph are the critical-section lengths of the blocking requests.

$$
w(\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) = \begin{cases} L_{x,y} & \text{if } (\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) \in E, \\ 0 & \text{otherwise.} \end{cases}
$$

Example 3.6 (continued). The example graph for the running example in this section is depicted in Figure 3.7, assuming $L_{x,y} = 1$ for all $\mathcal{R}_{x,y}$.

Definition 3.6. A *k-path P* in a blocking graph $G(\Gamma)$ is a simple path through $G(\Gamma)$ of at most k edges with no repeated vertices.

Example 3.6 (continued). $\{\mathcal{R}_{1,1}, \mathcal{R}_{2,1}, \mathcal{R}_{3,1}\}$ is an example 2-path in $G(\Gamma)$.

Definition 3.7. The total request blocking $B(P)$ of a k-path P through a blocking graph is equal to the sum of the weights of the edges in P.

$$B(P) = \sum_{(\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) \in P} w(\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) \tag{3.14}$$

Example 3.6 (continued). For the purpose of the running example, let $L_{x,y} = 1$ for all $\mathcal{R}_{x,y}$. Then $B(\{\mathcal{R}_{1,1}, \mathcal{R}_{2,1}, \mathcal{R}_{3,1}\}) = 3$.

With these definitions in place, we can derive a fine-grained blocking bound for the RSM.

Theorem 3.13. *The worst-case request blocking for $\mathcal{R}_{i,k}$ in the RSM, denoted $b_{i,k}^{RSM}$, is upper bounded by the maximal total blocking of any $(\mathcal{T}-1)$-path beginning with $\mathcal{R}_{i,k}$ in $G(\Gamma)$.*

Proof. By Theorems 3.1 and 3.12, a request can be blocked by at most $\mathcal{T}-1$ outermost requests. By construction $G(\Gamma)$ encodes all possible direct blocking relationships, and thus any $(\mathcal{T}-1)$-path through $G(\Gamma)$ beginning from $\mathcal{R}_{i,k}$ is a possible blocking scenario. The total blocking of the maximal $(\mathcal{T}-1)$-path beginning with $\mathcal{R}_{i,k}$ therefore upper bounds any possible blocking scenario in the RSM for $\mathcal{R}_{i,k}$. □

Example 3.6 (continued). Consider the request $\mathcal{R}_{2,1}$ and $\mathcal{T} = 2$. The worst-case 1-path originating at $\mathcal{R}_{2,1}$ has a total request blocking of $\max(L_{1,1}, L_{3,1}) = 1$ (again assuming $L_{x,y} = 1$ for all $\mathcal{R}_{x,y}$). Note that $\mathcal{R}_{4,1}$ does not contribute to the blocking for $\mathcal{R}_{2,1}$ as it is not "reachable" given the limited number of tokens.

It is well known that the longest-path problem, *i.e.*, determining the maximum-length path in a graph, is NP-complete (Cormen et al., 2001). The maximum-length k-path problem, which must be solved to compute the bound given in Theorem 3.13, is related to the longest-path problem, and can be easily shown to be NP-complete as well, via reduction from the longest-path problem.

Given the complexity of the maximal-length k-path problem, we present a less-tight bound that can be computed more easily.

Corollary 3.1. *Let $Reach(\mathcal{R}_{i,k}, l, G)$ denote the set of requests that are reachable in $G(\Gamma)$ from $\mathcal{R}_{i,k}$ via an l-path. The worst-case request blocking for $\mathcal{R}_{i,k}$ in the RSM, $b_{i,k}^{RSM}$, is upper bounded by the sum of the longest $\mathcal{T}-1$ requests in $Reach(\mathcal{R}_{i,k}, \mathcal{T}-1, G)$.*

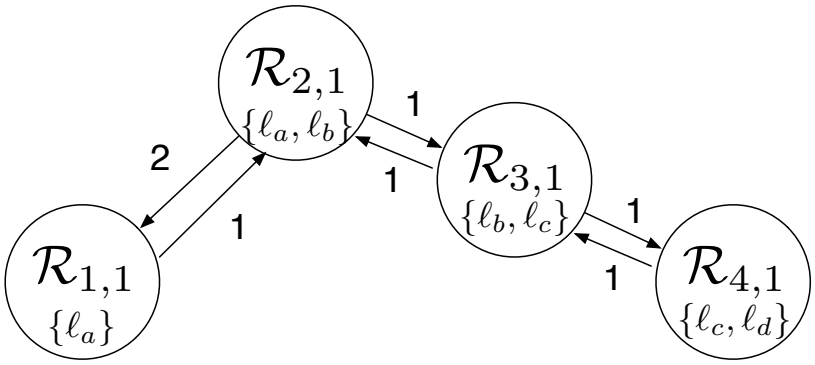


Figure 3.8: Depiction of the blocking graph for Example 3.6 where $L_{1,1} = 2$ and $L_{2,1} = L_{3,1} = L_{4,1} = 1$.

Example 3.6 (continued). Let $L_{1,1} = 2$ and $L_{2,1} = L_{3,1} = L_{4,1} = 1$ and $\mathcal{T} = 3$, as depicted in Figure 3.8. Using the tighter bound in Theorem 3.13, the two possible 2-paths originating from $\mathcal{R}_{2,1}$ in $G(\Gamma)$ have $B(\{(\mathcal{R}_{2,1}, \mathcal{R}_{1,1})\}) = B(\{(\mathcal{R}_{2,1}, \mathcal{R}_{3,1}), (\mathcal{R}_{3,1}, \mathcal{R}_{4,1})\}) = 2$. Using Corollary 3.1, $Reach(\mathcal{R}_{i,k}, 2, G(\Gamma)) = \{\mathcal{R}_{1,1}, \mathcal{R}_{3,1}, \mathcal{R}_{4,1}\}$. Therefore, the bound produced by Corollary 3.1 is $L_{1,1} + L_{3,1} = 3$, which is greater than that produced by Theorem 3.13.

The key difference between Theorem 3.13 and Corollary 3.1 is that the latter does not consider explicit blocking paths, only which requests are reachable through any $(\mathcal{T} - 1)$-path. As a result, the bound produced may include requests from different paths through the blocking graph.

With the results for the RSM pi-blocking, we are now prepared to discuss bounding token blocking and progress-mechanism-related pi-blocking.

Token lock pi-blocking. The modular nature of the RNLP, which separates the token lock from the RSM, allows for the reuse of previous pi-blocking analysis for the token-lock analysis. In particular, the pi-blocking analysis for the k-exclusion locking protocol can be applied directly by treating the critical-section length of each request $\mathcal{R}_{i,k}$ as $b_{i,k}^{RSM}$ instead of $L_{i,k}$.

Progress-mechanism-related pi-blocking. For the non-trivial token locks (non-TTL), progress-mechanism-related pi-blocking is also bounded through the token-lock analysis. For example, under the R^{2}DGLP, there is no progress-mechanism-related pi-blocking, and under the CK-OMLP, priority donation causes pi-blocking, which is analyzed in the CK-OMLP analysis.

Because the TTL does not itself cause any blocking (request or progress-mechanism-related), progress-mechanism-related pi-blocking must be considered separately. However, this analysis follows directly from previous analysis techniques, *e.g.*, Lemma 13 from (Brandenburg, 2014) for RSB.

Linear-programming-based fine-grained blocking analysis. In more recent work, Brandenburg (2013b) presented a framework for conducting pi-blocking analysis using a linear program (LP). In that framework, an LP is created for each job J_i, which enumerates all possible requests that could be issued while J_i is pending. The LP optimization function maximizes the blocking incurred by these requests, subject to constraints specific to each locking protocol that rule out potential schedules due to the behavior of the locking protocol. For example, for a FIFO-ordered spinlock, a request may only be directly blocked by at most one request per task. By adding constraints such as these to the LP, tighter blocking bounds result.

In this work, we did not specifically consider the use of LP-based blocking analysis for the RNLP. However, we conjecture it may be possible to add additional constraints for the RNLP to this LP-based analysis framework. For example, Corollary 3.1 could be used to add constraints that encode that within the RSM, requests may only be directly blocked by $\mathcal{T}-1$ requests in $Reach(\mathcal{R}_{i,k}, \mathcal{T}-1, G)$. Adding constraints that correspond to specific paths as in Theorem 3.13 would likely require integer variables.

3.7 Chapter Summary

In this chapter, we have presented the RNLP, a modular locking protocol composed of a k-exclusion token lock and a progress mechanism. Token locks and progress mechanisms are paired depending upon the scheduler, type of analysis, and how waiting is realized to achieve asymptotically optimal locking protocols corresponding to all prior asymptotically optimal multiprocessor real-time locking protocols. 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. Indeed, the RNLP predates RSB (Brandenburg, 2014), but RSB was easily be applied in the RNLP via the RSB-RSM to achieve asymptotic optimality in the clustered, s-aware case, as we discussed herein.

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 reduce both worst-case and observed runtime blocking. 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.

In the next chapter, we present extensions to the RNLP to support k-exclusion and reader/writer locking, as well as present an evaluation of the RNLP and these extensions.

CHAPTER 4: RNLP Extensions[1]

In Chapter 3, we considered a resource model in which all resources were assumed to be mutex resources. As described in Chapter 2, some resources have more relaxed sharing constraints such as reader/writer sharing or k-exclusion. In this chapter, we develop extensions to the RNLP that allow for fine-grained locking of resources with these alternative sharing constraints. To our knowledge, these extensions are the only locking protocols to support fine-grained reader/writer or k-exclusion locking.

Reader/writer. The first extension we consider in this chapter supports fine-grained reader/writer sharing, which allows read-only accesses to execute concurrently. We call this protocol that R/W RNLP. As we shall see, mixing read and write accesses in a fine-grained locking protocol gives rise to serious algorithmic and analytical challenges. The design of the R/W RNLP breaks new ground in several ways. For example, it is the first fine-grained multiprocessor real-time locking protocol that allows tasks to hold read locks and write locks simultaneously on different resources, and the first to allow read locks to be upgraded to write locks. We show that the R/W RNLP has worst-case pi-blocking no worse than previous coarse-grained reader/writer real-time locking protocols (Brandenburg and Anderson, 2010b; Brandenburg, 2011), which have been proven optimal with $O(m)$ writer blocking and $O(1)$ reader blocking.

Algorithmic and analytical challenges. The R/W RNLP was obtained by employing the concept of read and write "phases," as used in *phase-fair reader/writer* (*R/W*) *locks* (Brandenburg, 2011; Brandenburg and Anderson, 2009, 2010b, 2011), within the context of the RNLP (Ward and Anderson, 2012), which provides only mutex sharing. The RNLP effectively orders conflicting resource requests on a FIFO basis, *i.e.*, earlier requests are satisfied first. Thus, when abstractly considering behavior under the RNLP as a dynamically changing wait-for graph, an important *stability* property emerges: once a resource request is issued, its outgoing edge set, *i.e.*, the set of requests upon which it is waiting, does not change.

[1]Contents of this chapter previously appeared in preliminary form in the following papers:
Ward, B. and Anderson, J. (2013). Fine-grained multiprocessor real-time locking with improved blocking. In *Proceedings of the 21st Conference on Real-Time Networks and Systems*, pages 67–76.
Ward, B. and Anderson, J. (2014b). Multi-resource real-time reader/writer locks for multiprocessors. In *Proceedings of the 28th Parallel and Distributed Systems Symposium*, pages 177–186.

Phase-fair locks expressly violate this stability property. In order to enable $O(1)$ worst-case pi-blocking for read requests, phase-fair locks allow later-requested reads to "cut ahead" of earlier-requested writes. This is accomplished by alternating read and write phases; in a read (write) phase, the managed resource is accessed by *all* (*one*) issued read requests (write request). (Note that this is with respect to a *single* resource: prior work on phase-fair locks did not address the fine-grained sharing of multiple resources.) Because reads can "cut ahead" of writes, the outgoing edge set of write requests in the wait-for graph is not stable—in fact, it is not stable for any R/W locking protocol with $O(1)$ worst-case reader pi-blocking.

Dealing with this lack of stability was one of the main challenges we faced in designing the R/W RNLP since we desired $O(1)$ reader pi-blocking. One issue that arises on account of instability is what we call the *R/W ordering dilemma*. Consider a read request $\mathcal{R}_1^r$ that is waiting to access two resources, ℓ_a, which is read locked by request $\mathcal{R}_2^r$, and ℓ_b, which is write locked by request $\mathcal{R}_3^w$, as illustrated in Figure 4.1. Subsequently, a write request $\mathcal{R}_4^w$ is issued for the read-locked resource ℓ_a. Which request should be satisfied first, the waiting read $\mathcal{R}_1^r$ or the waiting write $\mathcal{R}_4^w$ (*i.e.*, where should $\mathcal{R}_4^w$ be inserted into the wait-for graph)? Phase-fair logic suggests that $\mathcal{R}_4^w$ be satisfied first (left side of Figure 4.1), as the resource for which it is currently waiting is read locked (*i.e.*, in a read phase, so a write phase should be next). However, this is problematic because it increases the blocking bound of the read request $\mathcal{R}_1^r$, which is already blocked by another writer ($\mathcal{R}_3^w$). Alternatively, if the read $\mathcal{R}_1^r$ is satisfied next (right side of Figure 4.1), then the write request $\mathcal{R}_4^w$ may be blocked by two read requests, leading to longer pi-blocking bounds than under a phase-fair lock.

Because we desire $O(1)$ pi-blocking for read requests, we have no choice but to sometimes let read requests "cut ahead" of write requests when resolving the R/W ordering dilemma, as in phase-fair locks. As noted above, this "cutting ahead" inserts edges into the wait-for graph that are not in accordance with FIFO request ordering. This has an effect that is not just localized but system-wide: in the wait-for graph, entire *paths*, representing *transitive* blocking relationships, may be inconsistent with FIFO ordering. The resulting ***transitive*** *early-on-late pi-blocking* can be difficult to properly handle and analyze.

Perhaps the most significant problem that arises in the development of a fine-grained R/W locking protocol is that of *inconsistent phases*. For example, consider that both a read and a write request are waiting for two resources, one of which is read locked, while the other is write locked. Which request should be satisfied next? This problem is further complicated when considering transitive blocking relationships. From the perspective of both waiting read and write requests, the resources for which they are waiting are in

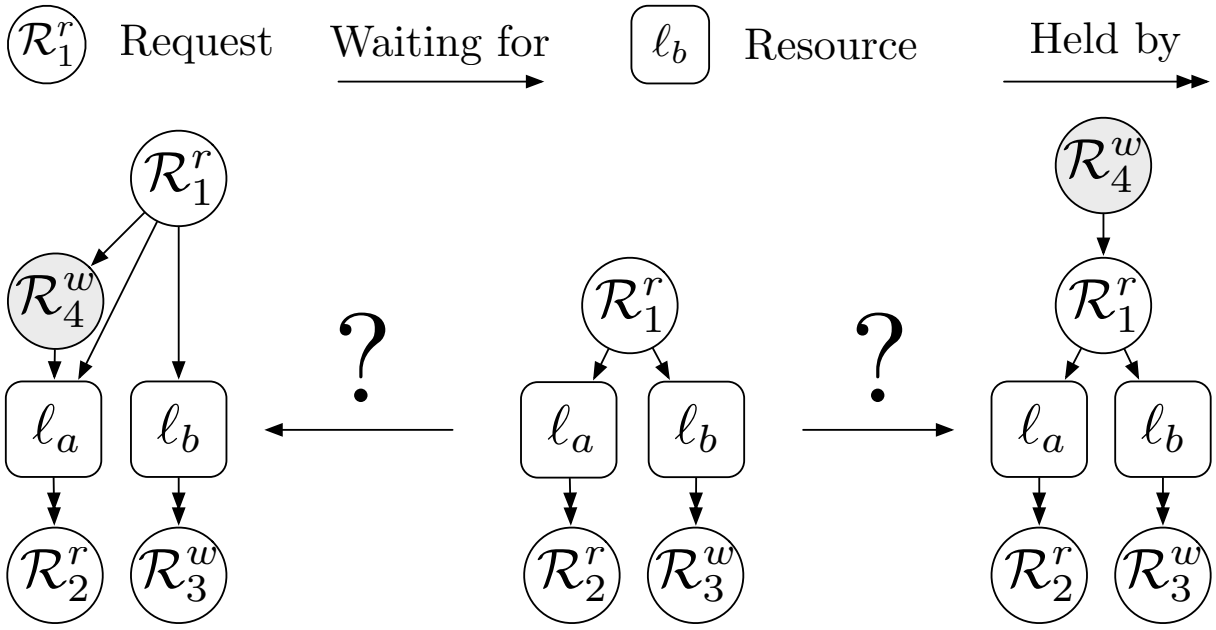


Figure 4.1: Illustration of the R/W ordering dilemma. The left and right side of the figure depict alternative places to insert a new write request $\mathcal{R}_4^w$ into the wait-for graph depicted in the middle.

inconsistent phases, in that one resource is in a *read phase*, while the other is in a *write phase*. This issue is unique to fine-grained locking protocols, as coarse-grained ones arbitrate access to all resources as one. To address this problem, we present the concept of *request entitlement*. The question of which request is satisfied next is decided in favor of the request that is deemed entitled.

The algorithmic challenges and their solutions in the R/W RNLP were motivated by analytical considerations. However, the analysis that proves the R/W RNLP to be optimal required insights in its own right. In particular, the analysis pertaining to the problem of (and solution to) inconsistent phases required novel insights, which can be seen both in the proof, as well as the definition of entitlement.

***k*-exclusion locking.** In many applications, it may be desirable to support fine-grained locking among both *k*-exclusion and mutex resources. For example, consider that a *k*-exclusion lock is used to arbitrate access to a GPU, and that data must be transferred to and/or from the GPU while it executes. If that data is guarded by a mutex lock, then the task may require simultaneous access to both the GPU and memory object(s).

This simple use case raises two research challenges. First, how can we support fine-grained locking among *k*-exclusion resources where the number of replicas may be different? To resolve this issue, the rules of the RNLP must be generalized.

In order to prevent deadlock, resources are ordered in the RNLP, and tasks must request resources in order. In order to maintain optimality, the RNLP enforces that a later-issued request never acquires a resource ℓ_b if an earlier-issued request holds a resource ℓ_a such that $\ell_a \prec \ell_b$. This rule ensures that an earlier-issued request can never be blocked by a later-issued request when it issues a nested request. Generalizing this rule to replicated resources is non-trivial. Do all replicas of resources later in the ordering need to be idled when earlier resources are locked? If not, how should replicas be "reserved" to ensure earlier-issued requests cannot block on later-issued requests?

As we will show in this chapter, this issue can be resolved by applying ideas from dynamic group locking. With DGLs, tasks request all potentially accessed resources, which allows replicas of later-ordered resources to be effectively "reserved." These reservations resolve the aforementioned issues, and allow for the construction of an asymptotically optimal fine-grained k-exclusion locking protocol.

Organization. The remainder of this chapter is organized as follows. We begin in Section 4.1 with a presentation of the R/W RNLP. In Section 4.2 we describe how the RNLP can be extended to support k-exclusion locking. In Section 4.3, we present schedulability studies conducted to evaluate the schedulability benefits of fine-grained locking with the RNLP presented in Chapter 3 and the extensions presented in this chapter.

4.1 R/W RNLP

Before describing the R/W RNLP, we begin by clarifying the models and assumptions used in this section.

Scheduling. We consider *clustered*-scheduled systems, in which the m processors are grouped into m/c clusters, each of size c. Note that these results can be applied to partitioned ($c = 1$) and global ($c = m$) scheduling as well. As before, we assume a job-level fixed priority scheduling algorithm.

Resource model. In this section, we assume that all resources in $\mathcal{L}$ are reader/writer (R/W) resources. Each resource indicated in a request is requested for either *reading* or *writing*. We say that a resource is *read* (*write*) *locked* if it is held by a request that reads (writes) it. We assume that each resource ℓ_a is subject to a *reader/writer sharing constraint*: writes of ℓ_a are mutually exclusive, but arbitrarily many reads of ℓ_a can be executed concurrently. Such a read is not allowed to modify ℓ_a. Two requests *conflict* if they include a common resource that is written by at least one of them.

In constructing the R/W RNLP, we endeavor, to the extent possible, to enable non-conflicting requests to be satisfied concurrently. Towards that end, we desire the following additional properties, in addition to fine-grained reader/writer locking.

- **R/W mixing**. Some resources may be read and others written in one critical section. Such critical sections can be satisfied concurrently if they do not conflict.
- **R-to-W upgrading.** A job that has acquired a resource for reading may *upgrade* its read to a write. For example, a job may read a resource, and based upon the value read, decide that it needs to write that resource.
- **Incremental locking.** The resources accessed by a job within a single critical section may be requested via a sequence of requests. For example, a job may request ℓ_a, read its value, and then execute some conditional code that requests ℓ_b.

However, to simplify exposition, we initially make the following simplifying assumption.

Assumption 4.1. All resources accessed within a single outermost critical section are requested via a single request, these resources are either all read or all written, and no read request may be upgraded.

We relax Assumption 1 later in this section and support the aforementioned three features. Until then (*i.e.*, while Assumption 1 is still in place), we use the following notation. We denote the set of resources that are needed in $\mathcal{R}_i$'s critical section as $\mathcal{N}_i$. By Assumption 1, each request can be categorized as either a *read request* or a *write request*, and each critical section as either a *read critical section* or a *write critical section*. For notational clarity, we often annotate read (respectively, write) requests as $\mathcal{R}_i^r$ (respectively, $\mathcal{R}_i^w$). Thus, $\mathcal{R}_1$ and $\mathcal{R}_1^w$ refer to the same request. We denote the longest read (respectively, write) critical section length as $L_{\max}^r$ (respectively, $L_{\max}^w$), and we let $L_{\max} = \max(L_{\max}^r, L_{\max}^w)$. Further, after showing the parallels between nested locking and DGLs in Section 3.5, for simplicity we also assume DGLs until we relax Assumption 1. All other terminology and notation from previous chapters persists.

Progress mechanisms. The R/W RNLP has stricter requirements on the progress mechanism employed. In particular, we require that a progress mechanism for the R/W RNLP satisfy the following two properties:

RWP1 A resource-holding job is always scheduled.

RWP2 At most m jobs may have incomplete resource requests at any time, at most c from each cluster.

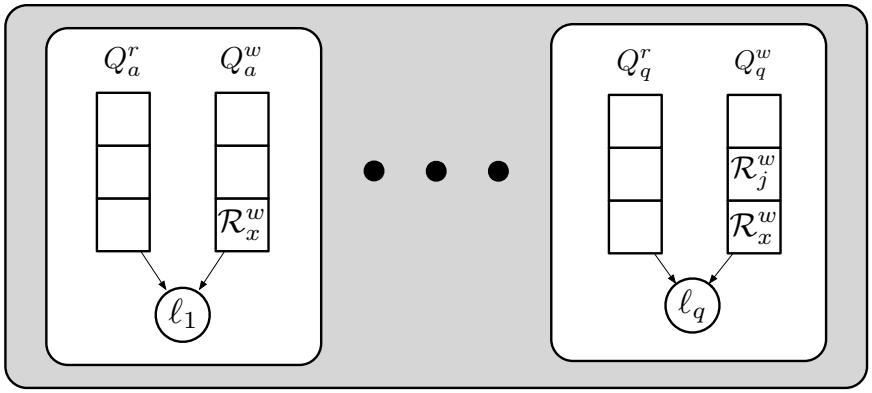


Figure 4.2: Queue structure in the R/W RSM. For each resource $\ell_a \in \{\ell_1, \ldots, \ell_q\}$, there is a read queue Q_a^r and a write queue Q_a^w.

The two progress mechanisms that satisfy these properties that we consider here are priority donation (Brandenburg and Anderson, 2011), and for spin-based locks, non-preemptive unconditional boosting. Both of these progress mechanisms trivially satisfy these two properties.

Token lock. Property RWP2 restricts the token lock to at most m-exclusion. Priority donation and spin-based non-preemptive boosting both realize an m-exclusion lock, and therefore, no token lock is required (*i.e.*, the TTL can be used).

We also note that to date no single-resource multiprocessor locking protocol supports s-aware pi-blocking that is $O(1)$ for reads and $O(n)$ for writes. It is for this reason, in this work we focus exclusively on spin-based and s-oblivious analysis. As discussed in Chapter 3, m-exclusion token locks are optimal in these cases, and indeed as we will show herein, are optimal for reader/writer fine-grained locking as well. S-aware reader/writer locking is a challenging open problem that warrants further attention.

4.1.1 R/W RNLP Architecture

The key new component in the R/W RNLP is the R/W RSM. In the R/W RSM, two queues are used per resource ℓ_a, a queue for readers, Q_a^r, and a queue for writers, Q_a^w, as depicted in Figure 4.2. We assume that each read (respectively, write) request is enqueued atomically in the read (respectively, write) queue of each resource it requests. The timestamp of the issuance of each request $\mathcal{R}_i$ is recorded and denoted $ts(\mathcal{R}_i)$. All writer queues are order by these timestamps, resulting in FIFO queueing. We denote the earliest timestamped incomplete write request for ℓ_a (*i.e.*, the head of Q_a^w) as $E(Q_a^w)$. Similar to phase-fair locks (Brandenburg and Anderson, 2010a,b), the queue from which requests are satisfied (Q_a^r or Q_a^w) alternates. The techniques

that govern such alternation, however, are quite different from traditional phase-fair locks due to the R/W ordering dilemma.

Example 4.1. As we explain the rules of the R/W RSM, we will reference relevant parts of the example schedule in Figure 4.3, which will later be explained in its entirety. In this running example, there are five tasks and a processor for each task, such that all pending jobs are scheduled. Additionally, these tasks share three resources, ℓ_a, ℓ_b, and ℓ_c. At time $t = 2$, when $\mathcal{R}_2^w$ is issued, $ts(\mathcal{R}_2^w) = 2$ is established. Also, since $\mathcal{R}_2^w$ requires all three resources and since it is the only write request waiting for any resource, $E(Q_a^w) = E(Q_b^w) = E(Q_c^w) = \mathcal{R}_2^w$. ◊

Before describing the techniques that govern when requests should be satisfied, we define relevant notation. We say that two resources ℓ_a and ℓ_b are *read shared*, denoted $\ell_a \sim \ell_b$,[2] if both ℓ_a and ℓ_b could be requested together as part of a single read request (*i.e.*, for some $\mathcal{R}_i^r$, $\{\ell_a, \ell_b\} \subseteq \mathcal{N}_i$). We call the set of all resources that are read shared with ℓ_a the *read set* of ℓ_a, denoted $S(\ell_a) = \{\ell_b |\ \ell_b \sim \ell_a\}$.

Example 4.1 (continued). In Figure 4.3, for $\mathcal{R}_5^r$, $\mathcal{N}_5 = \{\ell_a, \ell_b\}$. Thus, $\ell_a \sim \ell_b$ (and $\ell_b \sim \ell_a$). Since $\mathcal{R}_5^r$ is the only request for multiple resources, $S(\ell_a) = \{\ell_a, \ell_b\}$ and $S(\ell_c) = \{\ell_c\}$.

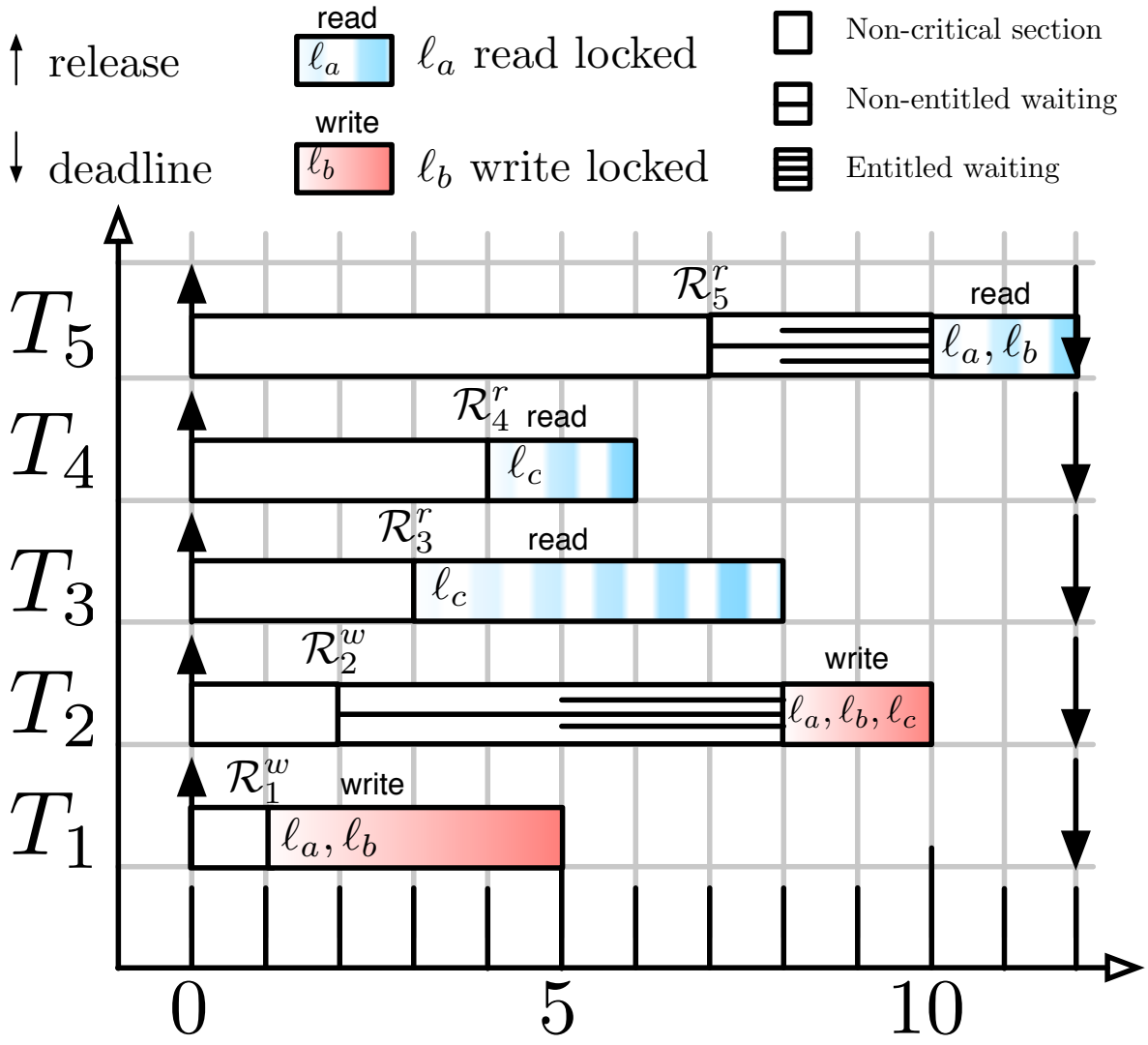
To avoid transitive early-on-late blocking, a write request may be forced to request additional resources besides those needed in its critical section. To reflect this, we let $\mathcal{D}_i$ denote the set of resources that $\mathcal{R}_i$ must actually request. For a read request $\mathcal{R}_i^r$, $\mathcal{D}_i$ is simply $\mathcal{N}_i$. However, for a write request $\mathcal{R}_i^w$, $\mathcal{D}_i = \bigcup_{\ell_a \in \mathcal{N}_i} S(\ell_a)$. While forcing write requests to acquire more resources than actually needed reduces runtime concurrency, it does not negatively affect worst-case pi-blocking. As we shall see, this expansion rule enables us to avoid transitive early-on-late blocking. Additionally, we shall show later that this expansion of write requests can be relaxed to enable additional concurrency on average.

Example 4.1 (continued). Suppose $\mathcal{R}_2^w$ in Figure 4.3 only needs $\mathcal{N}_2 = \{\ell_a, \ell_c\}$ in its critical section. Because $\ell_a \sim \ell_b$ and $\ell_a \in \mathcal{N}_2$, $\mathcal{R}_2^w$ actually requests $\mathcal{D}_2 = \{\ell_a, \ell_b, \ell_c\}$.

General rules. The first few rules of the R/W RSM are common to both readers and writers and describe the necessary actions that must be taken when a job either issues a request or completes a critical section.

G1 When J_i issues $\mathcal{R}_i$ at time t, the timestamp of the request is recorded: $ts(\mathcal{R}_i) := t$.

[2]Read sharing is reflexive and symmetric.



(a) Schedule in the running example.

time	**queue states**					
	Q_a^r	Q_a^w	Q_b^r	Q_b^w	Q_c^r	Q_c^w
$[0,2)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[2,7)$	$\emptyset$	$\{\mathcal{R}_2^w\}$	$\emptyset$	$\{\mathcal{R}_2^w\}$	$\emptyset$	$\{\mathcal{R}_2^w\}$
$[7,8)$	$\emptyset$	$\{\mathcal{R}_2^w\}$	$\{\mathcal{R}_5^r\}$	$\{\mathcal{R}_2^w\}$	$\emptyset$	$\{\mathcal{R}_2^w\}$
$[8,10)$	$\emptyset$	$\emptyset$	$\{\mathcal{R}_5^r\}$	$\emptyset$	$\emptyset$	$\emptyset$
$[10,12]$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

(b) Queue states over time in the schedule in (a).

Figure 4.3: Illustration of the running example.

G2 When $\mathcal{R}_i$ is satisfied, it is dequeued from either Q_a^r (if it is a read request) or Q_a^w (if it is a write request) for each $\ell_a \in \mathcal{D}_i$.

G3 When $\mathcal{R}_i$ completes, it unlocks all resources in $\mathcal{D}_i$.

G4 Each request issuance or completion occurs atomically. Therefore, there is a total order on timestamps, and a request cannot be issued at the same time that a critical section completes.

Example 4.1 (continued). At time $t = 8$, when $\mathcal{R}_3^r$ completes its critical section, $\mathcal{D}_3 = \{\ell_c\}$ is unlocked. This allows $\mathcal{R}_2^w$ to be satisfied (as explained later), and therefore $\mathcal{R}_2^w$ is dequeued from Q_a^w, Q_b^w, and Q_c^w.

The remaining read- and write-specific rules rely on the concept of *entitlement*, which we use to resolve the R/W ordering dilemma. Intuitively, a request becomes *entitled* once it is the next request to be satisfied (with respect to the resources for which it is waiting), and remains entitled until it is satisfied. While a request is entitled, it blocks all conflicting requests. Entitlement is defined differently for read and write requests. We begin with read requests, which are entitled if they are blocked only by satisfied (and not entitled) writes.

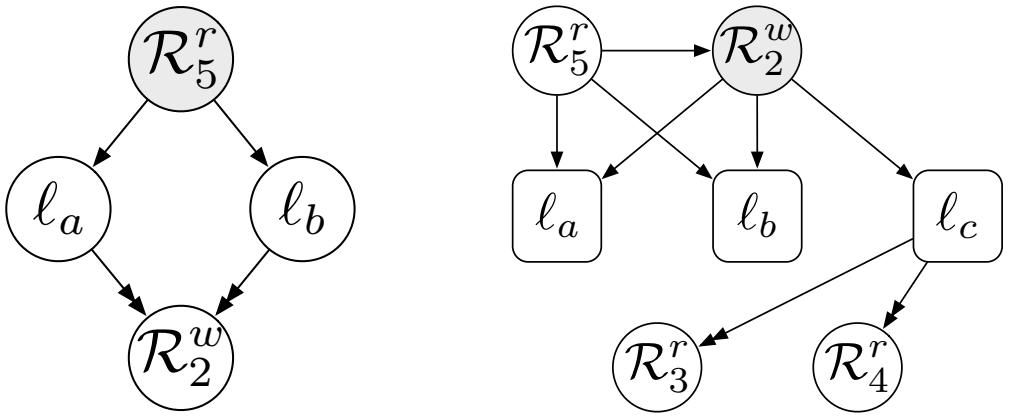
Definition 4.1. An unsatisfied read request $\mathcal{R}_i^r$ becomes *entitled* when there exists $\ell_a \in \mathcal{D}_i$ that is write locked, and for each resource $\ell_a \in \mathcal{D}_i$, $E(Q_a^w)$ is not entitled (see Definition 4.2).[3] (Note that $E(Q_a^w) = \emptyset$ could hold. In this case, we consider $E(Q_a^w)$ to be a "null" request that is not entitled.) $\mathcal{R}_i^r$ remains entitled until it is satisfied.

Of course, if a newly issued read request does not conflict with satisfied or entitled incomplete requests, then it is satisfied immediately (see Rule R1 below) and Definition 4.1 does not apply (only unsatisfied requests can be entitled).

Example 4.1 (continued). At time $t = 8$, $\mathcal{R}_5^r$ is blocked by $\mathcal{R}_2^w$, which holds ℓ_a, ℓ_b, and ℓ_c, as depicted in Figure 4.4 (a). By Definition 4.1, $\mathcal{R}_5^r$ becomes entitled at time $t = 8$ because ℓ_a and ℓ_b are write locked and $E(Q_a^w) = E(Q_b^w) = \emptyset$.

Next we consider the writer case. Intuitively, an entitled write is the head of all relevant write queues and not blocked by any entitled reads (but possibly satisfied reads).

[3] Entitlement is a property of a request, and Definitions 4.1 and 4.2 give conditions upon which a request becomes entitled in terms of the entitlement of other requests. Therefore, while Definitions 4.1 and 4.2 reference each other parenthetically to aid the reader, they are not in fact circularly defined.



(a) Reader entitlement. (b) Writer entitlement.

Figure 4.4: Illustrations of the wait-for graphs of entitled read and write requests. Inset (a) corresponds to $\mathcal{R}_5^r$ at time $t = 8$, and inset (b) corresponds to $\mathcal{R}_2^w$ at time $t = 7$ in Figure 4.3. Note that in inset (a), $\mathcal{R}_5^r$ is blocked by at least one satisfied write request, and in inset (b) $\mathcal{R}_2^w$ is blocked by at least one satisfied write request.

Definition 4.2. An unsatisfied write request $\mathcal{R}_i^w$ becomes *entitled* when for each $\ell_a \in \mathcal{D}_i$, $\mathcal{R}_i^w = E(Q_a^w)$, no read request in Q_a^r is entitled (see Definition 4.1),[4] and ℓ_a is not write locked. $\mathcal{R}_i^w$ remains entitled until it is satisfied.

Observe that an entitled write request $\mathcal{R}_i^w$ is only blocked by satisfied but incomplete read requests since according to Definition 4.2 no resource in $\mathcal{D}_i$ is write locked.

Example 4.1 (continued). At time $t = 7$, $\mathcal{R}_3^r$ holds ℓ_c, and blocks $\mathcal{R}_2^w$, which is waiting for ℓ_a, ℓ_b, and ℓ_c, as depicted in Figure 4.4 (b). Because $\mathcal{R}_2^w$ is the earliest timestamped writer waiting for any of the resources, and none is write locked, $\mathcal{R}_2^w$ becomes entitled. Note that, although $\mathcal{R}_2^w$ is entitled, it is still blocked. Prior to time $t = 5$, $\mathcal{R}_2^w$ was not be entitled because ℓ_a and ℓ_b were write locked by $\mathcal{R}_1^w$.

An entitled request (read or write) may be blocked by multiple requests, each holding different resources. We let $B(\mathcal{R}_i, t)$ be the set of satisfied requests that conflict with an entitled request $\mathcal{R}_i$ at time t (*i.e.*, the set of requests that block $\mathcal{R}_i$ at time t). Note that since read requests do not conflict with each other, $B(\mathcal{R}_i^r, t)$ only contains write requests. Analogously, an entitled write is only blocked by read requests, and thus $B(\mathcal{R}_i^w, t)$ only consists of read requests. This matches the phase-fair intuition that reads concede to writes, and writes concede to reads.

[4] See Footnote 3.

Example 4.1 (continued). At time $t \in [6,8)$, $\mathcal{R}_2^w$ is blocked by $\mathcal{R}_3^r$, thus $B(\mathcal{R}_2^w,t) = \{\mathcal{R}_3^r\}$. Earlier, at time $t \in [5,6)$, $\mathcal{R}_2^w$ is blocked by both $\mathcal{R}_3^r$ and $\mathcal{R}_4^r$, so $B(\mathcal{R}_2^w,t) = \{\mathcal{R}_3^r, \mathcal{R}_4^r\}$.

Reader rules. We next define reader-specific rules, which utilize the previously given definition of entitlement. These rules define the behavior of the R/W RSM, when a read request is issued and satisfied, respectively.

R1 When $\mathcal{R}_i^r$ is issued, for each $\ell_a \in \mathcal{D}_i$, $\mathcal{R}_i^r$ is enqueued in Q_a^r. If $\mathcal{R}_i^r$ does not conflict with any entitled or satisfied write requests, then it is satisfied immediately.

R2 An entitled read request $\mathcal{R}_i^r$ is satisfied at the first time instant t such that $B(\mathcal{R}_i^r,t) = \emptyset$.

Example 4.1 (continued). At time $t = 3$, $\mathcal{R}_3^r$ is issued and it is satisfied immediately by Rule R1. $\mathcal{R}_3^r$ is allowed to "cut ahead" of $\mathcal{R}_2^w$ in this case because $\mathcal{R}_2^w$ is not entitled, and ℓ_c is unlocked. Further, at time $t = 10$, $\mathcal{R}_5^r$ is satisfied by Rule R2. This is because $\mathcal{R}_5^r$ is entitled, and $\mathcal{R}_2^w$ completed its critical section and unlocked ℓ_a and ℓ_b.

Writer rules. The writer rules parallel the reader rules.

W1 When $\mathcal{R}_i^w$ is issued, for each $\ell_a \in \mathcal{D}_i$, $\mathcal{R}_i^w$ is enqueued in timestamp order in the write queue Q_a^w. If $\mathcal{R}_i^w$ does not conflict with any entitled or satisfied requests (read or write), then it is satisfied immediately.

W2 An entitled write request $\mathcal{R}_i^w$ is satisfied at the first time instant t such that $B(\mathcal{R}_i^w,t) = \emptyset$.

Example 4.1 (fully explained). At time $t = 1$, a write request $\mathcal{R}_1^w$ is issued for ℓ_a and ℓ_b, which is immediately satisfied (by Rule W1). At time $t = 2$, another write request, $\mathcal{R}_2^w$, is issued for ℓ_a, ℓ_b, and ℓ_c and is enqueued in Q_a^w, Q_b^w, and Q_c^w (by Rule W1). $\mathcal{R}_3^r$ is issued and satisfied immediately at time $t = 3$ by Rule R1, as previously described. Similarly, at time $t = 4$, $\mathcal{R}_4^r$ is issued and satisfied immediately (by Rule R1). Note that at time $t = 4$, both $\mathcal{R}_3^r$ and $\mathcal{R}_4^r$ have read locked ℓ_b, demonstrating reader parallelism. Further, at time $t = 4$, ℓ_a and ℓ_b are write locked while ℓ_c is read locked, a level of concurrency only possible with fine-grained locking. When $\mathcal{R}_1^w$ completes at time $t = 5$, $\mathcal{R}_2^w$ becomes entitled. At time $t = 7$, $\mathcal{R}_5$ is issued for ℓ_b and ℓ_c, but it is not satisfied because $\mathcal{R}_2^w$ is entitled to both resources. After $\mathcal{R}_3^r$ completes at time $t = 8$, $\mathcal{R}_2^w$ is satisfied (by Rule W2). Finally, after $\mathcal{R}_2^w$ completes at time $t = 10$, $\mathcal{R}_5^r$ is satisfied (by Rule R2).

This concludes the definition and introduction of the R/W RNLP. To summarize, the R/W RNLP implements phase-fairness, where reads concede to writes and writes concede to reads. To resolve the R/W ordering dilemma, we have introduced the concept of entitled blocking. Intuitively, an entitled request is

"next in line" with regard to its requested resources and only blocked by satisfied, but incomplete requests of the opposite kind.

4.1.2 Analysis

Similar to the analysis from Chapter 3, we begin with coarse-grained analysis, ultimately aimed at proving asymptotic optimality. In Section 4.1.2.3, we present fine-grained pi-blocking analysis, similar to that presented for the RNLP. For the purpose of asymptotic analysis, similarly to Chapter 3, we assume that m and n (the number of processors and tasks, respectively) are variables, and all other parameters are constants. Examples of such constants include critical-section lengths. Additionally, we assume that locking-protocol invocations take zero time and all other overheads are negligible (such overheads can be easily factored into the final analysis (Brandenburg, 2011, Chapters 3,7)).

4.1.2.1 Entitlement Analaysis

In this subsection, we present analysis showing a number of key properties about entitlement. In presenting this analysis, we let I denote an arbitrary invocation of the locking protocol (read or write issuance or read or write completion) that occurs at time t_I, and we let $t_I^- = \lim_{\varepsilon \to 0} t_I - \varepsilon$ be the time instant immediately prior to that invocation. We say that I *entitles* (*satisfies*) a request $\mathcal{R}_i$ if $\mathcal{R}_i$ becomes entitled (respectively, satisfied) as a result of I (*i.e.*, $\mathcal{R}_i$ is entitled (satisfied) after I but not before I).

Lemma 4.1. *The following properties of satisfaction and entitlement hold.*

E1 *If I satisfies $\mathcal{R}_i^r$, then I is either a read issuance or a write completion.*

E2 *If I satisfies $\mathcal{R}_i^w$, then I is either a write issuance, a read completion, or a write completion.*

E3 *If I satisfies $\mathcal{R}_i^r$ and I is the issuance of read request $\mathcal{R}_x^r$, then $\mathcal{R}_i^r = \mathcal{R}_x^r$.*

E4 *If I satisfies $\mathcal{R}_i^w$ and I is the issuance of write request $\mathcal{R}_x^w$, then $\mathcal{R}_i^w = \mathcal{R}_x^w$.*

E5 *If I satisfies $\mathcal{R}_i^w$ and I is the completion of a conflicting read request $\mathcal{R}_x^r$, then at time t_I^-, $\mathcal{R}_i^w$ is entitled, and $B(\mathcal{R}_i^w, t_I^-) = \{\mathcal{R}_x^r\}$.*

E6 *If I satisfies $\mathcal{R}_i^r$ and I is the completion of a conflicting write request $\mathcal{R}_x^w$, then at time t_I^-, $\mathcal{R}_i^r$ is entitled, and $B(\mathcal{R}_i^r, t_I^-) = \{\mathcal{R}_x^w\}$.*

E7 *If I satisfies $\mathcal{R}_i^w$ and I is the completion of a conflicting write request $\mathcal{R}_x^w$, then at time t_I^-, for each $\ell_a \in \mathcal{D}_i^w$, $\mathcal{R}_i^w = E(Q_a^w)$ and no read request in Q_a^r is entitled, and for each resource $\ell_a \in \mathcal{D}_i$, ℓ_a is either locked by $\mathcal{R}_x^w$, or unlocked.*

E8 *If I entitles $\mathcal{R}_i^r$, then I is a read issuance or a read completion.*

E9 *If I entitles $\mathcal{R}_i^w$, then I is a write issuance or a write completion.*

E10 *If $\mathcal{R}_i^w$ and $\mathcal{R}_x^r$ conflict, then they are not simultaneously entitled.*

Proof. We prove the stated properties in succession.

Proposition E1. If I is a write issuance, then it releases no resources for which $\mathcal{R}_i^r$ is waiting, and hence cannot cause $\mathcal{R}_i^r$ to become satisfied. On the other hand, if I is a read completion and $\mathcal{R}_i^r$ is not entitled prior to I, then by Rule R2, I cannot cause $\mathcal{R}_i^r$ to become satisfied. If I is a read completion and $\mathcal{R}_i^r$ is entitled (and hence blocked) prior to I, then $B(\mathcal{R}_i^r, t_I^-)$ contains at least one write request; I cannot cause this write request to complete, thus following I, $\mathcal{R}_i^r$ remains entitled (and hence blocked).

Proposition E2. Like the first case considered above, I cannot cause $\mathcal{R}_i^w$ to be become satisfied if it is a read issuance.

Proposition E3. If I is the issuance of read request $\mathcal{R}_i^r$, then it does not unlock any resources, and hence cannot cause any previously issued request to become satisfied. However, by Rule R1, I may cause $\mathcal{R}_i^r$ itself to become satisfied.

Proposition E4. If I is the issuance of write request $\mathcal{R}_i^w$, then it does not unlock any resources, and hence cannot cause any previously issued request to become satisfied. However, by Rule W1, I may cause $\mathcal{R}_i^w$ itself to become satisfied.

Proposition E5. By Rule W2, if I satisfies $\mathcal{R}_i^w$, then prior to I, $\mathcal{R}_i^w$ must have been entitled, and $\mathcal{R}_x^r$ must have been the only request that blocked $\mathcal{R}_i^w$.

Proposition E6. By Rule R2, if I satisfies $\mathcal{R}_i^r$, then prior to I, $\mathcal{R}_i^r$ must have been entitled, and $\mathcal{R}_x^w$ must have been the only request that blocked $\mathcal{R}_i^r$.

Proposition E7. By Rule W2, if I satisfies $\mathcal{R}_i^w$, then it must be entitled. However, because $\mathcal{R}_x^w$ is satisfied at time t_I^- and conflicts with $\mathcal{R}_i^w$, $\mathcal{R}_i^w$ is not entitled at time t_I^- by Definition 4.2. For $\mathcal{R}_i^w$ to be satisfied at time t_I, by Rule W2, it must become entitled at time t_I. By Definition 4.2, for $\mathcal{R}_i^w$ to be entitled at time t_I,

after $\mathcal{R}_x^w$ unlocks all resources in $\mathcal{D}_x$, for each $\ell_a \in \mathcal{D}_i$, $\mathcal{R}_i^w = E(Q_a^w)$, no read request in Q_a^r is entitled, and ℓ_a is not write locked. Furthermore, since $\mathcal{R}_i^w$ is satisfied at time t_I, all resources in $\mathcal{D}_i$ are unlocked after $\mathcal{R}_x^w$ completes. The proposition follows.

Proposition E8. By Definition 4.1, if $\mathcal{R}_i^r$ is unsatisfied and not entitled prior to I, *i.e.*, at time t_I^-, then it is blocked at t_I^- by an entitled write request, $\mathcal{R}_x^w$. Thus, by Definition 4.2, the following hold at time t_I^-: $\mathcal{R}_x^w$ is at the head of each write queue in which it is enqueued; no resource for which $\mathcal{R}_x^w$ is waiting is write locked; and $\mathcal{R}_x^w$ is not blocked by any entitled read request. Recall that entitled requests are, by definition, unsatisfied. Thus, $\mathcal{R}_x^w$ must be blocked by at least one *satisfied* read request at time t_I^-. Now, if I is a write issuance, then $\mathcal{R}_x^w$ clearly remains entitled at time t_I, and hence $\mathcal{R}_i^r$ is not entitled at time t_I. On the other hand, if I is a write completion, then it may cause certain entitled reads to become satisfied; however, it will not cause the satisfied read that blocks $\mathcal{R}_x^w$ to complete. Thus, as before, $\mathcal{R}_x^w$ remains entitled at time t_I, and hence $\mathcal{R}_i^r$ is not entitled at time t_I.

Proposition E9. By Definition 4.2, if $\mathcal{R}_i^w$ is unsatisfied and not entitled prior to I, *i.e.*, at time t_I^-, then it is blocked at t_I^- by either **(i)** at least one earlier-issued write request $\mathcal{R}_x^w$, or **(ii)** or some entitled read request $\mathcal{R}_y^r$ (or both).

Case (i). If at time t_I^-, $\mathcal{R}_i^w$ is blocked by the earlier-issued write request $\mathcal{R}_x^w$, then clearly if I is a read issuance, then $\mathcal{R}_x^w$ will continue to block $\mathcal{R}_i^w$ at time t_I. Hence, $\mathcal{R}_i^w$ is not entitled at time t_I. If I is a read completion, it may satisfy certain write requests, but $\mathcal{R}_x^w$ will remain incomplete, hence $\mathcal{R}_i^w$ is not entitled at time t_I.

Case (ii). If at time t_I^-, $\mathcal{R}_i^w$ is blocked by some entitled read request $\mathcal{R}_y^r$, and I is a read issuance, then $\mathcal{R}_y^r$ remains entitled, and hence $\mathcal{R}_i^w$ is not entitled at time t_I. If I is a read completion, then $\mathcal{R}_y^r$ remains entitled (recall entitled requests are not satisfied) as the satisfied write request upon which it is blocked remains satisfied. Hence, $\mathcal{R}_i^w$ is not entitled at time t_I.

Not that if both conditions (i) and (ii) hold simultaneously, the proof logic from either case is sufficient to show that the proposition holds.

Proposition E10. Definitions 4.1 and 4.2 preclude conflicting read and write requests from both becoming entitled due to *separate* invocations of the locking protocol. Propositions E8 and E9 preclude such requests from both becoming entitled due to the *same* invocation of the locking protocol. □

Next we show that once a write request $\mathcal{R}_i^w$ is entitled, no conflicting request $\mathcal{R}_x$ can be satisfied before it, which implicitly bounds how long it remains entitled.

Lemma 4.2. *If a write request $\mathcal{R}_i^w$ is entitled before and after I and $\mathcal{R}_x \in B(\mathcal{R}_i^w, t_I)$, then $\mathcal{R}_x \in B(\mathcal{R}_i^w, t_I^-)$.*

Proof. Suppose not. Then the mentioned request $\mathcal{R}_x$ (read or write) is satisfied by I, and by the definition of $B(\mathcal{R}_i^w, t_I)$, $\mathcal{R}_x$ conflicts with $\mathcal{R}_i^w$.

Assume that $\mathcal{R}_x^r$ is a read request. Then, by Proposition E1, I is a read issuance or a write completion. If I is a read issuance, then by Proposition E3, $\mathcal{R}_x^r$ is issued at t_I; however, by Rule R1, I cannot then satisfy $\mathcal{R}_x^r$ because $\mathcal{R}_i^w$ is entitled. If I is a write completion, then by Proposition E6, $\mathcal{R}_x^r$ is entitled at t_I^-; however, by Proposition E10, this implies that $\mathcal{R}_i^w$ is not entitled at t_I^-, contradicting the lemma statement.

Now assume that $\mathcal{R}_x^w$ is a write request. Then, by Proposition E2, I is a write issuance, read completion, or write completion. If I is a write issuance, then by Proposition E4, $\mathcal{R}_x^w$ is issued at t_I; however, by Rule W1, I cannot then satisfy $\mathcal{R}_x^w$ because $\mathcal{R}_i^w$ is entitled. If I is a read completion, then by Proposition E5, $\mathcal{R}_x^w$ is entitled at time t_I^-; however, by Proposition E10, this implies that $\mathcal{R}_i^w$ is not entitled at t_I^-, contradicting the lemma statement. If I is a write completion, by the statement of the lemma, it follows that $\mathcal{R}_i^w$ and $\mathcal{R}_x^w$ conflict and share some resource ℓ_c. Moreover, by Proposition E7, $\mathcal{R}_x^w = E(Q_c^w)$ holds at t_I^-. However, by Definition 4.2, this contradicts the assumption that $\mathcal{R}_i^w$ is entitled at t_I^-. □

Similar to Lemma 4.2, we next show that once a read request $\mathcal{R}_i^r$ becomes entitled, no conflicting request can be satisfied before it.

Lemma 4.3. *If a read request $\mathcal{R}_i^r$ is entitled before and after I and $\mathcal{R}_x^w \in B(\mathcal{R}_i^w, t_I)$, then $\mathcal{R}_x^w \in B(\mathcal{R}_i^w, t_I^-)$.*

Proof. Suppose not. Then, the mentioned write request $\mathcal{R}_x^w$ is satisfied by I, and by the definition of $B(\mathcal{R}_i^w, t_I)$, $\mathcal{R}_x^w$ conflicts with $\mathcal{R}_i^w$. Thus, by Proposition E2, I is either a write issuance, read completion, or write completion. If I is a write issuance, then by Proposition E4, I is the issuance of $\mathcal{R}_x^w$ itself; however, by Rule W1, I cannot satisfy $\mathcal{R}_x^w$, because $\mathcal{R}_i^r$ is entitled prior to I. If I is a read (respectively, write) completion, then by Proposition E5 (respectively, Proposition E7), $\mathcal{R}_x^w$ is entitled at t_I^-; however, by Proposition E10, this contradicts the assumption that $\mathcal{R}_i^r$ is entitled at t_I^-. □

The following two corollaries follow from Lemmas 4.2 and 4.3, respectively, and are crucial to our blocking analysis.

Corollary 4.1. *Suppose that the request $\mathcal{R}_i^w$ becomes entitled at time t_e and satisfied at time t_s. Then, no new requests may be added to $B(\mathcal{R}_i^w, t)$ at any time time $t \in [t_e, t_s)$.*

Proof. For contradiction, let $t_f \in [t_e, t_s)$ be the first time instant in which a new request $\mathcal{R}_x$ is added to $B(\mathcal{R}_i^w, t_f)$, *i.e.*, $\mathcal{R}_x \notin B(\mathcal{R}_i^w, t_f^-) \wedge \mathcal{R}_x \in B(\mathcal{R}_i^w, t_f)$. Then by Lemma 4.2 and the fact that $\mathcal{R}_i^w$ is entitled during $[t_e, t_s)$, $\mathcal{R}_x \in B(\mathcal{R}_i^w, t_f^-)$. Contradiction. □

Example 4.1 (continued). This corollary is demonstrated at time $t = 7$ in Figure 4.3, when $\mathcal{R}_5^r$ is issued. Because $\mathcal{R}_2^w$ is entitled at that time, $\mathcal{R}_5^r$ is forced to block until after $\mathcal{R}_2^w$ completes, even though the resources it requested are available.

Corollary 4.2. *Suppose that the request $\mathcal{R}_i^r$ becomes entitled at time t_e and satisfied at time t_s. Then, no new requests may be added to $B(\mathcal{R}_i^r, t)$ at any time $t \in [t_e, t_s)$.*

Proof. For contradiction, let $t_f \in [t_e, t_s)$ be the first time instant in which a new request $\mathcal{R}_x$ is added to $B(\mathcal{R}_i^r, t_f)$, *i.e.*, $\mathcal{R}_x \notin B(\mathcal{R}_i^r, t_f^-) \wedge \mathcal{R}_x \in B(\mathcal{R}_i^r, t_f)$. Then, by Lemma 4.3 and the fact that $\mathcal{R}_i^r$ is entitled during $[t_e, t_s)$, $\mathcal{R}_x \in B(\mathcal{R}_i^r, t_f^-)$. Contradiction. □

While Corollary 4.2 is not depicted in Figure 4.3, it is similar to Corollary 4.1. Next, we show that worst-case acquisition delay is $O(1)$ for readers and $O(m)$ for writers. Two additional lemmas are used in establishing these results.

4.1.2.2 Blocking Analysis

After establishing the above key properties of entitlement, especially Corollaries 4.1 and 4.2, we now proceed to leverage these properties in bounding worst-case pi-blocking.

Lemma 4.4. *A write request $\mathcal{R}_i^w$ experiences acquisition delay of at most $L_{\max}^r$ time units after becoming entitled.*

Proof. Suppose that $\mathcal{R}_i^w$ becomes entitled at time t_e and satisfied at time t_s. By Corollary 4.1, new requests are not added to $B(\mathcal{R}_i^w, t)$ at any $t \in [t_e, t_s)$. Moreover, by Definition 4.2, each request in $B(\mathcal{R}_i^w, t)$ is a read. By Property RWP1, every request in $B(\mathcal{R}_i^w, t_e)$ is scheduled, and therefore will complete in at most $L_{\max}^r$ time units. Thus, by time $t_e + L_{\max}^r$, $\mathcal{R}_i^w$ will not be blocked, and by Rule W2, will be satisfied. □

The following lemma is essential to show that transitive early-on-late blocking does not adversely affect the worst-case blocking bounds.

Lemma 4.5. *If $\mathcal{R}_i^w$ is the earliest timestamped write request among all incomplete write requests, then $\mathcal{R}_i^w$ is either satisfied or entitled.*

Proof. Suppose not. Then, by Definition 4.2, for some resource $\ell_a \in \mathcal{D}_i$, either **(i)** $\mathcal{R}_i^w \neq E(Q_a^w)$, **(ii)** some request $\mathcal{R}_x^r \in Q_a^r$ is entitled, or **(iii)** ℓ_a is write locked. By Rule W1, (i) and (iii) are not possible since the write queues are ordered by timestamp, and $\mathcal{R}_i^w$ is the earliest incomplete write. For (ii), assume $\mathcal{R}_x^r$ is entitled and $\ell_a \in \mathcal{D}_i \cap \mathcal{D}_x$. Then, by Definition 4.1, $\mathcal{R}_x^r$ is blocked by a satisfied write request $\mathcal{R}_h^w$. Recall that $\mathcal{R}_h^w$ must request all resources in the read sets of resources in $\mathcal{N}_h$. Further, ℓ_a must be in at least one of these read sets. Thus, $\ell_a \in \mathcal{D}_h \cap \mathcal{D}_i$, and $\mathcal{R}_h^w$ and $\mathcal{R}_i^w$ conflict. Thus, since $ts(\mathcal{R}_i^w) < ts(\mathcal{R}_h^w)$, $\mathcal{R}_h^w$ cannot be satisfied. □

Theorem 4.1. *The worst-case acquisition delay of a read request $\mathcal{R}_i^r$ is at most $L_{\max}^w + L_{\max}^r$ time units.*

Proof. We first show that if $\mathcal{R}_i^r$ is issued at time t_i, then it must become entitled or satisfied by time $t_i + L_{\max}^r$. Suppose not. Then, throughout the interval $[t_i, t_i + L_{\max}^r)$, $\mathcal{R}_i^r$ is blocked by a non-empty set W of conflicting entitled write requests, for otherwise, $\mathcal{R}_i^r$ would become entitled (by Definition 4.1) or satisfied (by Rule R1). By Property RWP1 and Lemma 4.4, each write request $\mathcal{R}_x^w \in W$ will be satisfied by time $t_i + L_{\max}^r$. Once all such write requests are satisfied, by Definition 4.1, $\mathcal{R}_i^r$ will become entitled or satisfied, a contradiction.

If $\mathcal{R}_i^r$ becomes satisfied by time $t_i + L_{\max}^r$, then its acquisition delay is at most $L_{\max}^r$ time units. Consider now the other possibility, *i.e.*, that $\mathcal{R}_i^r$ becomes entitled by some time $t_e \leq t_i + L_{\max}^r$. In this case, we show that $\mathcal{R}_i^r$ is satisfied by time $t_e + L_{\max}^w$, from which an acquisition delay of at most $L_{\max}^r + L_{\max}^w$ time units follows. By Corollary 4.2, the number of resource-holding write requests blocking $\mathcal{R}_i^r$ monotonically decreases until $\mathcal{R}_i^r$ is satisfied. By Property RWP1, each such blocking request completes in at most $L_{\max}^w$ time units. Thus, $\mathcal{R}_i^r$ is satisfied by time $t_e + L_{\max}^w$. □

Theorem 4.2. *The worst-case acquisition delay of a write request $\mathcal{R}_i^w$ is at most $(m-1)(L_{\max}^r + L_{\max}^w)$ time units.*

Proof. Suppose that the write request $\mathcal{R}_i^w$ is issued at time t_i and not satisfied immediately. Let $\mathcal{R}_x^w$ be the incomplete write request with the earliest timestamp at t_i ($\mathcal{R}_x^w$ could be $\mathcal{R}_i^w$). By Lemma 4.5, $\mathcal{R}_x^w$ is either entitled or satisfied at t_i. Suppose the latter is true, *i.e.*, $\mathcal{R}_x^w$ is satisfied at t_i. Then, by Property RWP1, $\mathcal{R}_x^w$

completes its critical section by time $t_i + L^w_{\max}$. By Property RWP2, there are at most $m-1$ incomplete write requests with timestamps earlier than that of $\mathcal{R}^w_i$ at t_i. Thus, by time $t_i + L^w_{\max}$, there are at most $m-2$ such requests. By Lemmas 4.4 and 4.5, the one with the earliest timestamp is satisfied by time $t_i + L^w_{\max} + L^r_{\max}$, and thus, by Property RWP1, completes its critical section by time $t_i + L^w_{\max} + L^r_{\max} + L^w_{\max}$. Continuing inductively, all earlier-timestamped write requests complete their critical sections by time $t_i + L^w_{\max} + (m-2)(L^r_{\max} + L^w_{\max})$. At that time, $\mathcal{R}^w_i$ has the earliest timestamp. Hence, by Lemma 4.4, it is satisfied by time $t_i + L^w_{\max} + (m-2)(L^w_{\max} + L^r_{\max}) + L^r_{\max}$, *i.e.*, $\mathcal{R}^w_i$'s acquisition delay is at most $(m-1)(L^r_{\max} + L^w_{\max})$ time units.

The remaining possibility to consider is that $\mathcal{R}^w_x$ is entitled at t_i. In this case, by Definition 4.2, $\mathcal{R}^w_x$ is blocked by some read request $\mathcal{R}^r_h$. Thus, by Property RWP2, there are at most $m-2$ incomplete write requests with timestamps earlier than that of $\mathcal{R}^w_i$ at t_i.

By Property RWP1, $\mathcal{R}^r_h$ will complete by time $t_i + L^r_{\max}$, and $\mathcal{R}^w_x$ is completed at time $t_i + L^r_{\max} + L^w_{\max}$. Again inductively, all earlier-timestamped write requests complete their critical sections by time $t_i + L^r_{\max} + (m-2)(L^r_{\max} + L^w_{\max})$. Thus, it follows that $\mathcal{R}^w_i$'s acquisition delay is at most $(m-2)(L^r_{\max} + L^w_{\max}) + L^r_{\max}$ time units. (Note that the blocking of $\mathcal{R}^w_x$ due to $\mathcal{R}^r_h$ is accounted for in this reasoning by Lemma 4.4.) □

For a spin-based lock, the worst-case acquisition delay for either reads or writes is the worst-case s-blocking (recall Definition 2.3). However, non-preemptive spinning can cause other jobs, even non-resource-using jobs, to be pi-blocked upon release. For example, if a high-priority job J_h is released that has sufficient priority to be scheduled, but a low-priority job J_l is spinning non-preemptively, then J_h is pi-blocked. The worst-case pi-blocking can easily be shown to be $O(m)$ through analysis similar to single-resource spin-based mutex or reader-writer locks (Brandenburg, 2011).

4.1.2.3 Fine-grained R/W RNLP Blocking Analysis

Similarly to the fine-grained blocking analysis for the mutex RNLP in Section 3.6, we also present fine-grained blocking analysis for the R/W RNLP. There are many similarities in these analyses, and we therefore use a similar analysis framework as that used in Section 3.6. We refer the reader back to Section 3.6 for pertinent examples that illustrate the concepts of the analysis framework.

In the analysis in Section 3.6, we observed that in the mutex RNLP a request may be blocked by at most one other request per task within the RSM. By similar logic, since writes are satisfied in FIFO order (with

respect to other writes) a request in the R/W RSM may be blocked by at most one write request per task. However, because read and write phases are interleaved, it is possible for multiple read requests to block to another write request. This observation affects the contention set.

Lemma 4.6. ***(Brandenburg, 2011, Lemma 5.1)*** *At most* $\left\lceil \frac{t+r_i}{p_i} \right\rceil$ *distinct jobs of a task* τ_i *can execute in any interval of length* t.

In applying this result to determine the number of contending requests, the response time of each task is initially assumed to be equal to its period. A fixed-point iteration can be applied to determine more precise blocking bounds. Specifically, after computing the response time including blocking bounds, the new response-time bounds can be applied to recompute the tighter blocking bounds. This process can proceed iteratively.

Definition 4.3. The *write contention set* of request $\mathcal{R}_{i,k}$ is the set of write requests that could potentially contend within the R/W RSM with $\mathcal{R}_{i,k}$.

$$C^w(\Gamma) = \bigcup_{J_x \in \Gamma} \{\mathcal{R}^w_{x,1}, \mathcal{R}^w_{x,2}, \ldots, \mathcal{R}^w_{x,N^w_i}\} \tag{4.1}$$

Definition 4.4. The *read contention set* of request $\mathcal{R}_{i,k}$ is the set of read requests that could potentially contend with the R/W RSM with $\mathcal{R}_{i,k}$ in an interval of length t.

$$C^r(\mathcal{R}_{i,k}, t) = \bigcup_{J_x \in \Gamma \setminus \{\tau_i\}} \bigcup_{y=1}^{\lceil \frac{t+r_x}{px} \rceil} \{\mathcal{R}^r_{x,y,1}, \mathcal{R}^r_{x,y,2}, \ldots, \mathcal{R}^r_{x,y,N^r_i}\} \tag{4.2}$$

Example 4.2. Consider a task system $\Gamma = \{\tau_1, \ldots \tau_5\}$ in which all tasks have the same period, p_Γ, and each job issues one resource request. Let $\mathcal{R}_1, \mathcal{R}_2$, and $\mathcal{R}_3$ be write requests and $\mathcal{R}_2$, and $\mathcal{R}_4$ be read requests. Also, let $\mathcal{D}_1 = \{\ell_a\}$, $\mathcal{D}_2 = \{\ell_a, \ell_b\}$, $\mathcal{D}_3 = \{\ell_b, \ell_c\}$, $\mathcal{D}_4 = \{\ell_b, \ell_c\}$, and $\mathcal{D}_5 = \{\ell_c\}$. The write contention set is $C^w(\Gamma) = \{\mathcal{R}_{1,1}, \mathcal{R}_{3,1}, \mathcal{R}_{5,1}\}$. The read contention set of $\mathcal{R}^w_1$ over its period, p_Γ, is $C^r(\mathcal{R}^w_1, p_\Gamma) = \{\mathcal{R}^r_{2,1}, \mathcal{R}^r_{1,2}, \mathcal{R}^r_{4,1}, \mathcal{R}^r_{4,2}\}$. By Lemma 4.6, assuming $r_i = p_i$, at most two distinct jobs of each other task can execute within the period of τ_1, hence there are two read requests in the read contention set of $\mathcal{R}^w_1$ for each task that issues read requests, τ_2 and τ_4. ◊

Definition 4.5. A request $\mathcal{R}_{i,k}$ may be directly blocked by another request $\mathcal{R}_{x,y}$ if $\mathcal{D}_{i,k} \cap \mathcal{D}_{x,y} \neq \emptyset$, $i \neq x$, and both $\mathcal{R}_{i,k}$ and $\mathcal{R}_{x,y}$ are not read requests. Let $db(\mathcal{R}_{i,k}, \mathcal{R}_{x,y})$ be true if $\mathcal{R}_{i,k}$ can be directly blocked by $\mathcal{R}_{x,y}$.

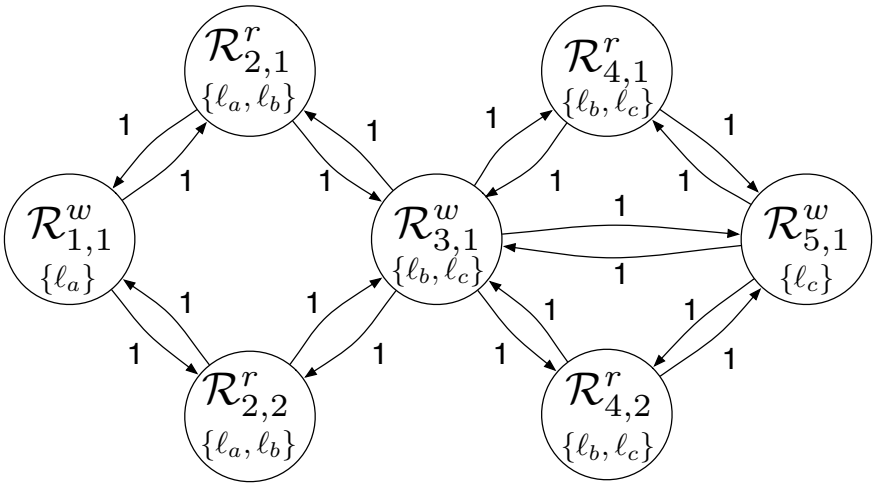


Figure 4.5: Depiction of the blocking graph $G(\Gamma, \mathcal{R}^w_{5,1}, p_\Gamma)$ corresponding to Example 4.2 assuming that for all $\tau_i \in \Gamma$, $L^r_i = L^w_i = 1$.

By this definition, a request may be directly blocked by any conflicting request from a different task. Next we define the blocking graph, similarly to the analysis in Section 3.6.

Example 4.2 continued. $\mathcal{R}^w_3$ may be directly blocked by $\mathcal{R}^r_2$, $\mathcal{R}^r_4$, and $\mathcal{R}^w_5$, because it is a write request, and requests a common resource with each of these other requests. However, $\mathcal{R}^r_2$ is not directly blocked by $\mathcal{R}^r_4$ because, while they both request ℓ_b, they are both read requests, and therefore do not conflict.

Definition 4.6. Let $G(\Gamma, \mathcal{R}_{i,k}, t) = (V, E)$ be a *blocking graph*, which encodes all possible blocking.

$$
\begin{aligned}
&G = (V, E), \\
&V = C^w(\Gamma) \cup C^r(\mathcal{R}_{i,k}, t) \\
&E = \{(\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) \mid db(\mathcal{R}_{i,k}, \mathcal{R}_{x,y})\}.
\end{aligned}
$$

The *edge weights* in the blocking graph are the critical-section lengths of the blocking requests.

$$
w(\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) = \begin{cases} L_{x,y} & \text{if } (\mathcal{R}_{i,k}, \mathcal{R}_{x,y}) \in E, \\ 0 & \text{otherwise.} \end{cases}
$$

Example 4.2 continued. The blocking graph $G(\Gamma, \mathcal{R}^w_5, p_\Gamma)$ corresponding to the running example is depicted in Figure 4.5.

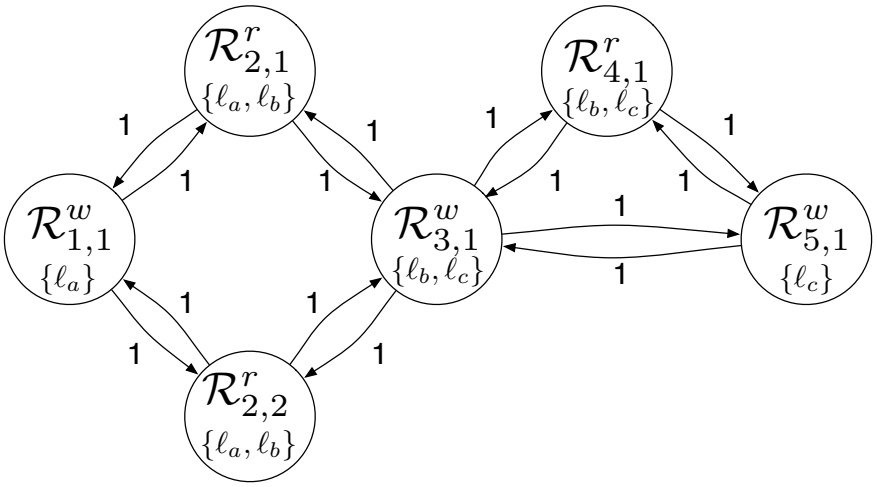


Figure 4.6: Depiction of the blocking graph $G(\Gamma, \mathcal{R}^r_4, p_\Gamma)$ corresponding to Example 4.2 assuming $L^r_i = L^w_i = 1$ for all $\tau_i \in \Gamma$.

Definition 4.7. A (r,w)-path P in a blocking graph $G(\Gamma, \mathcal{R}_{i,k}, t)$ is a simple path through $G(\Gamma, \mathcal{R}_{i,k}, t)$ of at most r read vertices and w write vertices, excluding the source node.

Example 4.2 continued. Consider the path $P = \{\mathcal{R}^w_{1,1}, \mathcal{R}^r_{2,1}, \mathcal{R}^w_{3,1}, \mathcal{R}^r_{4,1}, \mathcal{R}^w_{5,1}\}$. P is a $(2,2)$-path.

Theorem 4.3. *The worst-case request blocking for a read request $\mathcal{R}^r_{i,k}$ in the R/W RSM is upper bounded by the maximal total blocking of any $(1,1)$-path in $G(\Gamma, \mathcal{R}^r_{i,k}, r_i)$ beginning with $\mathcal{R}^r_{i,k}$.*

Proof. By Theorem 4.1, a request can be blocked by at most one read and one write phase. By construction, $G(\Gamma, \mathcal{R}_{i,k}, r_i)$ encodes all possible direct blocking relationships, and thus any $(1,1)$-path beginning with $\mathcal{R}^r_{i,k}$ is a possible blocking scenario. The total blocking of the maximal $(1,1)$-path beginning with $\mathcal{R}^r_{i,k}$ upper bounds any possible blocking scenario in the R/W RSM for $\mathcal{R}^r_{i,k}$. □

Example 4.2 continued. Consider the blocking graph $G(\Gamma, \mathcal{R}^r_{4,1}, p_\Gamma)$, shown in Figure 4.6. Assuming constant read and write critical-section lengths of one, the worst-case request blocking of $\mathcal{R}^r_{4,1}$ is $L^w_3 + L^r_2 = 2$. However, as shown in Figure 4.7, if $L_5 = 5$ the weight of the path $\{\mathcal{R}^r_{4,1}, \mathcal{R}^w_{5,1}\}$ outweighs the path $\{\mathcal{R}^r_{4,1}, \mathcal{R}^w_{3,1}, \mathcal{R}^r_{2,1}\}$ of weight two, even though it does not include a read request.

Theorem 4.4. *The worst-case request blocking for a write request $\mathcal{R}^w_{i,k}$ in the R/W RSM is upper bounded by the maximal total blocking of any $(m-1, m-1)$-path in $G(\Gamma, \mathcal{R}_{i,k}, r_i)$.*

Proof. By Theorem 4.2, a request can be blocked by at most $m-1$ read and $m-1$ write phases. By construction, $G(\Gamma, \mathcal{R}_{i,k}, r_i)$ encodes all possible direct blocking relationships, and thus any $(m-1, m-1)$-

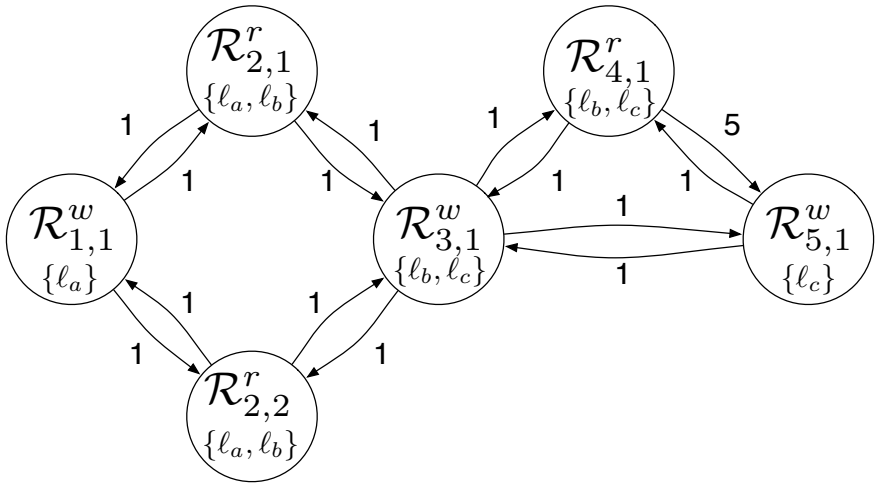


Figure 4.7: Depiction of the blocking graph $G(\Gamma, \mathcal{R}^r_{4,1}, p_\Gamma)$ corresponding to Example 4.2 assuming that for all $\tau_i \in \Gamma \setminus \{\tau_5\}$, $L^r_i = L^w_i = 1$, and $L^w_5 = 5$.

path beginning with $\mathcal{R}^w_{i,k}$ is a possible blocking scenario. The total blocking of the maximal $(m-1, m-1)$ beginning with $\mathcal{R}^w_{i,k}$ upper bounds any possible blocking scenario in the R/W RSM for $\mathcal{R}^w_{i,k}$. □

Example 4.2 continued. Consider the blocking graph of $\mathcal{R}^w_{5,1}$ depicted in Figure 4.5, and let $m = 2$. Assuming all critical sections have a length of one, a maximum-length $(2,2)$-path through $G(\Gamma, \mathcal{R}^w_{5,1}, p_\Gamma)$ is $\{\mathcal{R}^w_{5,1}, \mathcal{R}^r_{4,1}, \mathcal{R}^w_3, \mathcal{R}^r_{2,1}, \mathcal{R}^w_{1,1}\}$.

The blocking bound in Theorem 4.3 is more general than that in Theorem 3.13 (if there are no read requests, the RNLP and R/W RNLP behave the same). The problem of computing this bound, similarly to Theorem 3.13, is also NP-complete. (The problem of computing the bound in Theorem 4.3 is not NP-complete given the constant-length $(1,1)$-path computed.) Similarly to Section 3.6, we give a looser bound which can be computed more efficiently.

Corollary 4.3. *Let $Reach(\mathcal{R}^w_{i,k}, r, w, G)$ denote the set of requests that are reachable in G from $\mathcal{R}_{i,k}$ via an (r,w)-path. The worst-case request blocking for $\mathcal{R}_{i,k}$ in the R/W RSM, is upper bounded by the sum of the longest r read requests and w write requests in $Reach(\mathcal{R}_{i,k}, m-1, m-1, G(\Gamma, \mathcal{R}^w_{i,k}, r_i))$.*

As discussed in Section 3.6, using these bounds on request blocking, more detailed bounds on progress-mechanism-related pi-blocking can be computed using previous techniques (Brandenburg, 2011). Also, we conjecture that the linear-programming-based analysis of (Brandenburg, 2013b) could be applied to the R/W RNLP by adding further constraints based on the analysis presented herein.

4.1.3 R/W RNLP Optimizations

Next, we briefly summarize additional optimizations that can be incorporated into the R/W RNLP to improve average-case parallelism, and thus responsiveness in many cases. These optimizations do not affect the worst-case blocking bounds. We describe them independently for ease of exposition, but note that they can be combined in a real implementation. These optimization improve parallelism, and therefore average-case responsiveness, which is often desirable in practice.

4.1.3.1 Requesting Fewer Resources

Requiring write requests to lock an expanded set of resources enabled us to establish Lemma 4.5. This lemma can instead be established by utilizing *placeholders*, which allow for increased parallelism. Specifically, we require a write request $\mathcal{R}_i^w$ to enqueue a *placeholder* $\mathcal{R}_i^p$ in the queues of all non-needed resources that we earlier required $\mathcal{R}_i^w$ to request. In this case, the R/W RNLP functions as previously described with the following exceptions. A placeholder is never entitled or satisfied. Instead, each placeholder $\mathcal{R}_i^p$ is removed from the write queue in which it is enqueued when $\mathcal{R}_i^w$ becomes entitled or satisfied. Therefore, until $\mathcal{R}_i^w$ becomes entitled, its associated placeholders prevent later-issued write requests from becoming entitled or satisfied, thereby ensuring that Lemma 4.5 is not violated.

Example 4.1 (continued). Continuing from the example in Figure 4.3, suppose that $\mathcal{R}_1^w$ only needed $\mathcal{N}_1 = \{\ell_b\}$ and $\mathcal{R}_2^w$ only needed $\mathcal{N}_2 = \{\ell_a, \ell_c\}$. When $\mathcal{R}_1^w$ is issued, it would enqueue a placeholder in Q_a^w, but since it is satisfied immediately, the placeholder is removed. When $\mathcal{R}_2^w$ is issued, it enqueues in Q_a^w and Q_c^w, and enqueues a placeholder in Q_b^w. However, $\mathcal{R}_2^w$ is not blocked by any conflicting requests, since $\mathcal{R}_1^w$ only holds the lock on ℓ_b, so $\mathcal{R}_2^w$ can be satisfied immediately at time $t = 2$, thereby improving concurrency.

Using placeholders allows for additional concurrency. However, this parallelism is not reflected in the worst-case blocking bounds under our analysis assumptions.

4.1.3.2 R/W Mixing

Before we show how to relax Assumption 1 to allow jobs to issue *mixed* requests, we first extend our notation. We denote the set of resources that $\mathcal{R}_i$ needs read (respectively, write) access to as $\mathcal{N}_i^r$ (respectively, $\mathcal{N}_i^w$) and we let $\mathcal{N}_i = \mathcal{N}_i^r \cup \mathcal{N}_i^w$. If $\mathcal{N}_i^w = \emptyset$, then we say $\mathcal{R}_i$ is a read request, otherwise we say that $\mathcal{R}_i$ is a write request. With this notation, a mixed request is a write request $\mathcal{R}_i^w$ with $\mathcal{N}_i^r \neq \emptyset$ and $\mathcal{N}_i^w \neq \emptyset$. We

also adapt our definition of the read shared relation, $\sim$. Given two resources ℓ_a and ℓ_b, we say that ℓ_b is read shared with ℓ_a, if for some potential request $\mathcal{R}_i$, $\ell_a \in \mathcal{N}_i$, and $\ell_b \in \mathcal{N}_i^r$.[5]

The rules of the R/W RNLP support mixed requests with only a minor modification. Intuitively, a mixed request is treated almost exactly like an exclusively write request, though there are three key differences. First, an entitled mixed request can be satisfied if all resources for which it requires read access are either unlocked *or* read locked. Second, when a mixed request is satisfied, resources for which read-only access is needed are read locked, not write locked, which allows read requests to be satisfied concurrently. Third, with respect to writer entitlement (Definition 4.2), blocked write requests treat a resource that is read locked by a mixed request as if it were write locked.

Example 4.1 (continued). Consider the schedule depicted in Figure 4.3, without the placeholder optimization described previously. Assume that $\mathcal{R}_2^w$ is actually a mixed request that requires read access to $\mathcal{D}_2^r = \{\ell_a, \ell_b\}$ and write access to $\mathcal{D}_2^w = \{\ell_c\}$. Then when $\mathcal{R}_5^r$ is issued at time $t = 7$, because it does not conflict with $\mathcal{R}_2^w$ (both requests only require read access to ℓ_a and ℓ_b), $\mathcal{R}_5^r$ can be satisfied immediately by Rule R1.

4.1.3.3 R-to-W Upgrading

We call a read request that can be upgraded to a write request, as previously described, an *upgradeable* request, which we denote as $\mathcal{R}_i^u$. Intuitively, we treat an upgradeable request as a write request that can optimistically execute read-only code while its needed resources are read-locked to determine if write access is necessary. Since the blocking bounds of a write request assume that it will be blocked by other read requests, the optimistic execution of the read-only section essentially executes for free. Thus, an upgradeable request has the same worst-case blocking bounds as a write request, but may offer additional concurrency if the write segment of the critical section is not required.

To support this behavior in the R/W RNLP, we treat $\mathcal{R}_i^u$ as two separate logical requests, a read request,[6] $\mathcal{R}_i^{u_r}$ and a write request $\mathcal{R}_i^{u_w}$, which can cancel each other if necessary. (With respect to Property RWP2, an upgradeable request is only one request; logical requests are not counted separately.) When $\mathcal{R}_i^u$ is issued, $\mathcal{R}_i^{u_r}$ is enqueued as a read request and $\mathcal{R}_i^{u_w}$ is enqueued as a write request. If $\mathcal{R}_i^{u_w}$ is satisfied before $\mathcal{R}_i^{u_r}$, then $\mathcal{R}_i^{u_r}$

[5]The read sharing relation may not be symmetric with mixed requests. For example, let $\mathcal{N}_i = \{\ell_a, \ell_b\}$ and $\mathcal{N}_i^r = \{\ell_b\}$. Then $\ell_a \sim \ell_b$, but $\ell_b \not\sim \ell_a$.

[6]We assume the worst-case execution time of the read-only segment of the upgradeable request finishes in $L_{\max}^r$ time.

is canceled and removed from all read queues. If $\mathcal{R}_i^{u_r}$ is satisfied first, it executes its critical section, and upon completion or realization that upgrading is not necessary, $\mathcal{R}_i^{u_w}$ is canceled and removed from all write queues in which it is enqueued. If $\mathcal{R}_i^{u}$ must be upgraded, then when the read-only segment of its critical section completes, all resources are unlocked. Later, when $\mathcal{R}_i^{u_w}$ is satisfied, the job can execute the write segment of its critical section. Note that the state of any read objects may change between $\mathcal{R}_i^{u_r}$ completing and $\mathcal{R}_i^{u_w}$ being satisfied. Thus, $\mathcal{R}_i^{u_w}$ may need to re-read data. If this behavior is unacceptable for a given application, a write request should instead be issued for all resources that could be written.

4.1.3.4 Incremental Locking

Next, we show how the R/W RNLP can be adapted to allow jobs to incrementally request resources they use within a critical section, as described earlier. We assume that it is known *a priori* the set of all resources that could possibly be requested in this incremental fashion. For example, if resources are ordered, it may be assumed that a request $\mathcal{R}_i$ may issue a nested request while holding ℓ_a for any resource ℓ_b such that $\ell_a \prec \ell_b$. Alternatively, the R/W RNLP could be implemented such that an outermost request informs the locking protocol which resources it may request in a nested fashion.

To support this functionality, we initially treat $\mathcal{R}_i$ as if it were a request for all of the resources for which it could potentially lock incrementally. From Corollaries 4.1 and 4.2, after $\mathcal{R}_i$ becomes entitled, no conflicting request can be satisfied before $\mathcal{R}_i$. Thus, if $\mathcal{R}_i$ only initially requires access to some subset $s \subseteq \mathcal{D}_i$, it can be granted access as soon as it is entitled and each resource $\ell_a \in s$ is not locked by a conflicting request. $\mathcal{R}_i$ remains entitled to all other resources in $\mathcal{D}_i \setminus s$, thereby preventing other conflicting requests from acquiring those resources. If $\mathcal{R}_i$ later needs some additional resource(s) $s' \subseteq \mathcal{D}_i \setminus s$, then it waits until each $\ell_a \in s'$ is not locked by a conflicting request. However, because $\mathcal{R}_i$ is entitled to all resources in $\mathcal{D}_i$, the total duration of acquisition delay across all incremental requests is at most the worst-case acquisition delay previously proven in Theorems 4.1 and 4.2.

Note that entitlement serves a similar purpose as priority ceilings (Sha et al., 1990), since it prevents later-issued requests from acquiring resources that may be incrementally requested.

Example 4.1 (continued). Consider again the example schedule depicted in Figure 4.3. If $\mathcal{R}_2^w$ were to incrementally lock ℓ_a and/or ℓ_b, then $\mathcal{R}_2^w$ could begin executing on either or both of these resources starting at time $t = 5$ when $\mathcal{R}_1^w$ unlocks ℓ_a and ℓ_b. If $\mathcal{R}_2^w$ were to incrementally request ℓ_c, to which it is entitled

beginning at time $t = 5$, then it would block and wait until time $t = 8$ when $\mathcal{R}_3^r$ completes its read critical section. Note that through this incremental locking, some of the critical section of $\mathcal{R}_2^w$ executed early, serving only to improve concurrency and reduce blocking.

4.1.4 Implementation

In this section, we present pseudocode for a spin-based implementation of the R/W RNLP for partitioned-scheduled systems. This implementation does not support any of the extensions described in Section 4.1.3. The implementation uses bitmasks to encode the state of each of the resources. These bitmasks are protected by a phase-fair lock. Importantly, read requests only ever need read access to this lock. Write requests are queued in shared per-resource FIFO queues, and a mutex lock is used to protect accesses and to ensure that enqueues into multiple write queues are effectively atomic.

Listing 1 Shared variables in the R/W RNLP implementation.

```
1: Phase-fair PFLock
2: Mutex MLock
3: bitmask Unavailable, WEntitled, WLocked initially 0
4: Request struct WQueue[0..q − 1][0..m − 1] initially NULL
5: int WHead[0..q − 1], WTail[0..q − 1] initially 0
6: int Entry[0..m − 1] initially 0
7: int Exit[0..m − 1] initially 0
8: Request struct Requests[0..m − 1] initially NULL
```

The implementation contains many shared-state variables, which are depicted in Listing 1. The mutex lock *MLock* is used to ensure that enqueueing into the proper wait queues is atomic. The phase-fair lock *PFLock* protects the state variables that encode that status of the resources. The variables *Unavailable*, *WEntitled*, and *WLocked* are bitmasks where each bit corresponds to a single resource. *Unavailable* encodes which resources are not available to be locked, *WEntitled* encodes whether or not there is an entitled write request waiting for a given resource, and *WLocked* encodes whether a resource is write locked or not. Write requests are stored in circular buffers, *WQueue*, and *WHead* and *WTail* store pointers to the head and the tail of each resource queue. The *Entry* and *Exit* variables are used in a lock-free fashion to allow waiting requests to detect when a blocking request has completed. The details of this logic will be described later. Finally, *Requests* stores all of the outstanding requests, one per processor.

The data structure corresponding to each individual request is shown in Listing 2. This structure encodes what resources have been requested, the status of the request (*i.e.*, whether it is waiting, entitled, or acquired), the type of the request (*i.e.*, read or write), and the processor from which the request was issued.

Listing 2 Request-struct members.

```
1: bitmask resources
2: status ∈ {WAITING, ACQUIRED, ENTITLED}
3: type ∈ {READ, WRITE}
4: int proc // the partition from which the request was issued.
```

Listing 3 Pseudocode for the R/W RNLP read lock function.

```
1:  procedure READ LOCK(resources)
2:      r ← Requests[proc]
3:      r.resources ← resources
4:      r.type ← READ
5:      r.status ← WAITING
6:      Entry[proc] ← Entry[proc] + 1
7:      read-lock PFLock
8:      if (r.resources & Unavailable) = 0 then
9:          r.status ← ACQUIRED
10:     end if
11:     unlock PFLock
12:     if r.status ≠ ACQUIRED then
13:         while r.resources & WEntitled ≠ 0 do
14:         end while
15:         r.status ← ENTITLED
16:         while r.resources & WLocked ≠ 0 do
17:         end while
18:     end if
19: end procedure
```

We begin by describing the read-lock code in Listing 3. To begin, the request struct r is initialized, and $Entry[proc]$ is incremented. This indicates to other processors that another request has been issued by processor $proc$. Next a check is performed to determine if the request can be satisfied immediately. This check can be performed efficiently using bitmasks, though it must be done in a read critical section after acquiring *PFLock*. In the case that the request is not satisfied immediately, then it must wait for conflicting entitled and/or satisfied write requests to complete, as in Lines 12–18.

The read unlock logic, shown in Listing 4, is quite simple. The per-processor *Exit* counter is incremented, which may trigger waiting write requests to proceed, as seen later, and the request is removed from the *Requests* array.

Listing 4 Pseudocode for the R/W RNLP read unlock function.

```
procedure READ UNLOCK(resources)
    Exit[proc] ← Exit[proc] + 1
    Requests[proc] ← NULL
end procedure
```

Listing 5 Pseudocode for the write lock function.

```
 1: procedure WRITE LOCK(resources)
 2:     req ← requests[proc]
 3:     req.resources ← resources
 4:     req.type ← WRITE
 5:     req.status ← WAITING
 6:     Entry[proc] ← Entry[proc] + 1
 7:     Lock MLock
 8:     for r ∈ resources do
 9:         WQueue[r][wtail[r]] ← req
10:         wtail[r] ← (wtail[r] + 1) % m
11:     end for
12:     Unlock MLock
13:     for r ∈ resources do
14:         while WQueue[r][WHead[r]] ≠ req do
15:         end while
16:     end for
17:     Write lock PFLock
18:     Unavailable ← Unavailable | req.resources
19:     req.status ← ENTITLED
20:     Write unlock PFLock
21:     for i ∈ {0,...,m−1} \ {req.processor} do
22:         start ← Entry[i]
23:         end ← Exit[i]
24:         tmp ← Requests[i]
25:         if tmp.resources & req.resources = 0 ∨ start = end ∨ tmp.type = WRITE ∨ tmp.status =
    WAITING then
26:             continue
27:         end if
28:         while Exit[i] < start do
29:         end while
30:     end for
31:     write-lock PFLock
32:     WEntitled & req.resources // Zero the bits in WEntitled corresponding to each requested resource
33:     wlocked ← wlocked | req.resources
34:     unlock PFLock
35: end procedure
```

Listing 6 Pseudocode for the R/W RNLP write unlock function.

```
procedure WRITE UNLOCK(resources)
    let req = requests[proc]
    exit[proc] = exit[proc] + 1
    requests[proc] ← NULL
    write-lock PFLock
    WLocked & req.resources // Zero the bits in WLocked corresponding to each requested resource
    Unavailable & req.resources // Zero the bits in Unavailable corresponding to each requested resource
    unlock PFLock
    Lock MLock
    for r ∈ resources do
        WQueue[r][WHead[r]] ← NULL
        WHead[r] ← (WHead[r] + 1) % m
    end for
    Unlock MLock
end procedure
```

The write lock logic is shown in Listing 5. In our discussion of this procedure, we assume that τ_i has issued $\mathcal{R}_i^w$ and is executing Listing 5. After initializing the request struct, in Lines 7–12, *req* is enqueued into all necessary wait queues. This enqueueing is protected by the *MLock* mutex lock, so as to ensure that enqueueing in all queues is effectively atomic. Next, in Lines 13–16, τ_i busy waits until it is the head of each wait queue in which it is enqueued. At this point, τ_i becomes entitled, and all requested resources are marked as unavailable, which prevents the resources from being acquired by a later-issued request. In Lines 21–30, τ_i waits for each satisfied read request to complete. τ_i detects that a conflicting read request has completed when it increments the per-processor *Exit* counter to be equal to the per-processor *Entry* counter. Finally, before $\mathcal{R}_i^w$ is finally satisfied, the resources it requested are marked as locked in *WLocked* and no longer entitled in *WEntitled*.

The last procedure in our R/W RNLP implementation is write unlock, which is shown in Listing 6. This procedure is predominately bookkeeping. The per-processor *Exit* counter is incremented, and the per-processor request struct is nullified. The unlocked resources are reflected in both *WLocked* and *Unavailable*, and the head of each queue is incremented.

4.2 Multi-Unit Multi-Resource Locking

In this section, we turn our attention to showing how to support replicated resources within the RNLP. We do this by leveraging recent work on asymptotically optimal real-time k-exclusion protocols (Elliott,

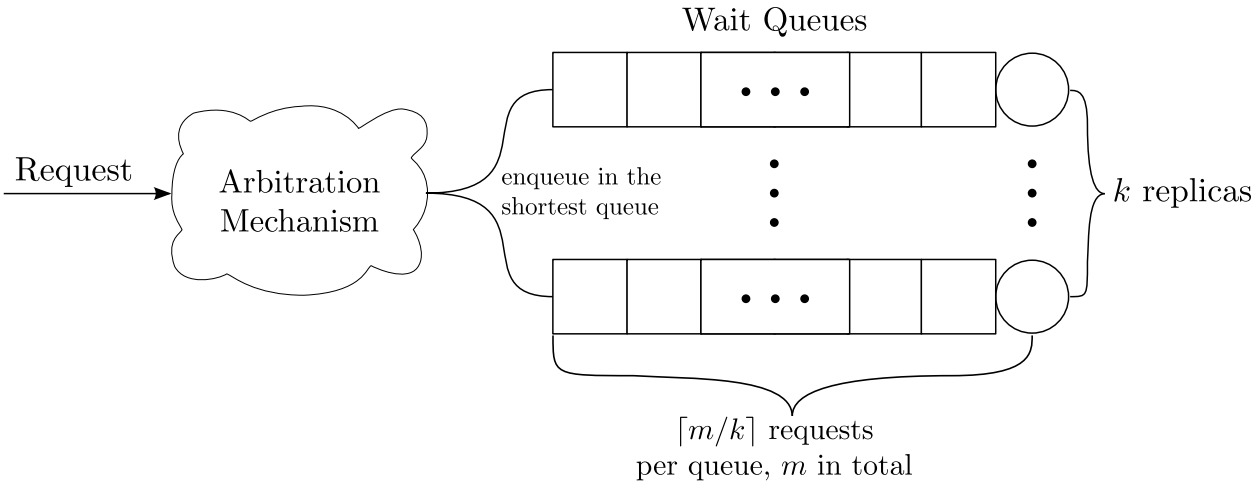


Figure 4.8: Figure illustrating the basic queue structure used in previous k-exclusion locking protocols. The arbitration mechanisms in these protocols behave similar to the token lock of the RNLP.

2015; Elliott and Anderson, 2013; Ward et al., 2012). Such protocols provide a limited form of replication: they enable requests to be performed on k replicas of a *single* resource. We desire to extend this functionality by allowing tasks to perform multiple requests simultaneously on replicas of *different* resources.

To motivate our proposed modifications to the RNLP, we consider three k-exclusion protocols, namely the O-KGLP (Elliott and Anderson, 2013), the k-FMLP (Elliott, 2015), and the R^{2}DGLP (described later in Chapter 5), which function as depicted in Figure 4.8. In these protocols, each replica is conceptually viewed as a distinct resource with its own queue. An "arbitration mechanism" (similar to our token lock) is used to limit the number of requests concurrently enqueued in these queues. In the case of s-aware (respectively, s-oblivious) analysis, the arbitration mechanism is configured to allow up to n (respectively, m) requests to be simultaneously enqueued. A "shortest queue" selection rule is used to determine the queue upon which a given request will be enqueued. This rule ensures that in the s-aware (respectively, s-oblivious and spin-based) case, each queue can contain at most $\lceil n/k \rceil$ (respectively, $\lceil m/k \rceil$) requests. From this, a pi-blocking bound of $O(n/k)$ (respectively, $O(m/k)$) can be shown. Both bounds are asymptotically optimal.

Suppose now that we have two such replicated resources, as shown in Figure 4.9, and that we wish to be able to support requests that involve accessing two replicas, one per resource, simultaneously. If the enqueueing associated with such a request is done by the arbitration mechanism atomically, then this is simple to do: as a result of processing the request, it is enqueued onto the shortest queue associated with each resource *at the same time*. This simple generalization of the aforementioned k-exclusion algorithms retains their optimal pi-blocking bounds.

Note that the functionality just described is provided by DGLs. Thus, to support multiple replicas when simultaneous lock holding is done only via DGLs (and not nesting), we merely need to treat each replica as a single resource and use a "shortest queue" rule in determining the replica queues in which to place a request. If each resource is replicated at least k times then it is straightforward to show that the earlier-stated pi-blocking bounds of $O(m/k)$ and $O(n/k)$ for s-oblivious and s-aware analysis, respectively, still apply. As before, both bounds are asymptotically optimal.

If simultaneous lock holding is done via nesting, then the situation is a bit more complicated. This is due to the RNLP's conservative resource acquisition rule (Rule Q3), which enables a request with a lower timestamp to effectively "reserve" its place in line within any queue of any resource it may request in the future. This rule causes problems with replicated resources. Consider again Figure 4.9. Consider an outermost request $\mathcal{R}_i$ for ℓ_a that *may* make a nested request for ℓ_b. Which replica queue for ℓ_b should hold its "reservation?" If a specific queue is chosen by the "shortest queue" rule when $\mathcal{R}_i$ receives its timestamp, and if $\mathcal{R}_i$ does indeed generate a nested request for ℓ_b later, then the earlier-selected queue may not still be the shortest for ℓ_b when the nested request is made. If a queue is not chosen until the nested request is made, then since $\mathcal{R}_i$ had no "reservation" in any queue of ℓ_b until then, it could be the case that requests with later timestamps hold all replicas of ℓ_b when the nested request is made. This violates a key invariant of the RNLP.

Our solution is to require $\mathcal{R}_i$ to conceptually place a reservation in the shortest replica queue for each resource that may be required in the future. The idea is to enact a "DGL-like" request for $\mathcal{R}_i$ when it receives a token that enqueues a "placeholder" request for $\mathcal{R}_i$ on one replica queue, determined by the "shortest queue" rule, for each resource it *may* access. Such a placeholder can later be canceled if it is known that the corresponding request will not be made. Thus, as before, nesting and DGLs are equivalent from the perspective of worst-case asymptotic pi-blocking.

4.3 Evaluation

In this section, we present experimental results for the RNLP extensions presented in this chapter.

4.3.1 R/W RNLP Evaluation

To evaluate the practicality of the R/W RNLP, we implemented the spin-based variant described in Section 4.1.4, and conducted a *schedulability* study, in which we applied a schedulability test to tens of

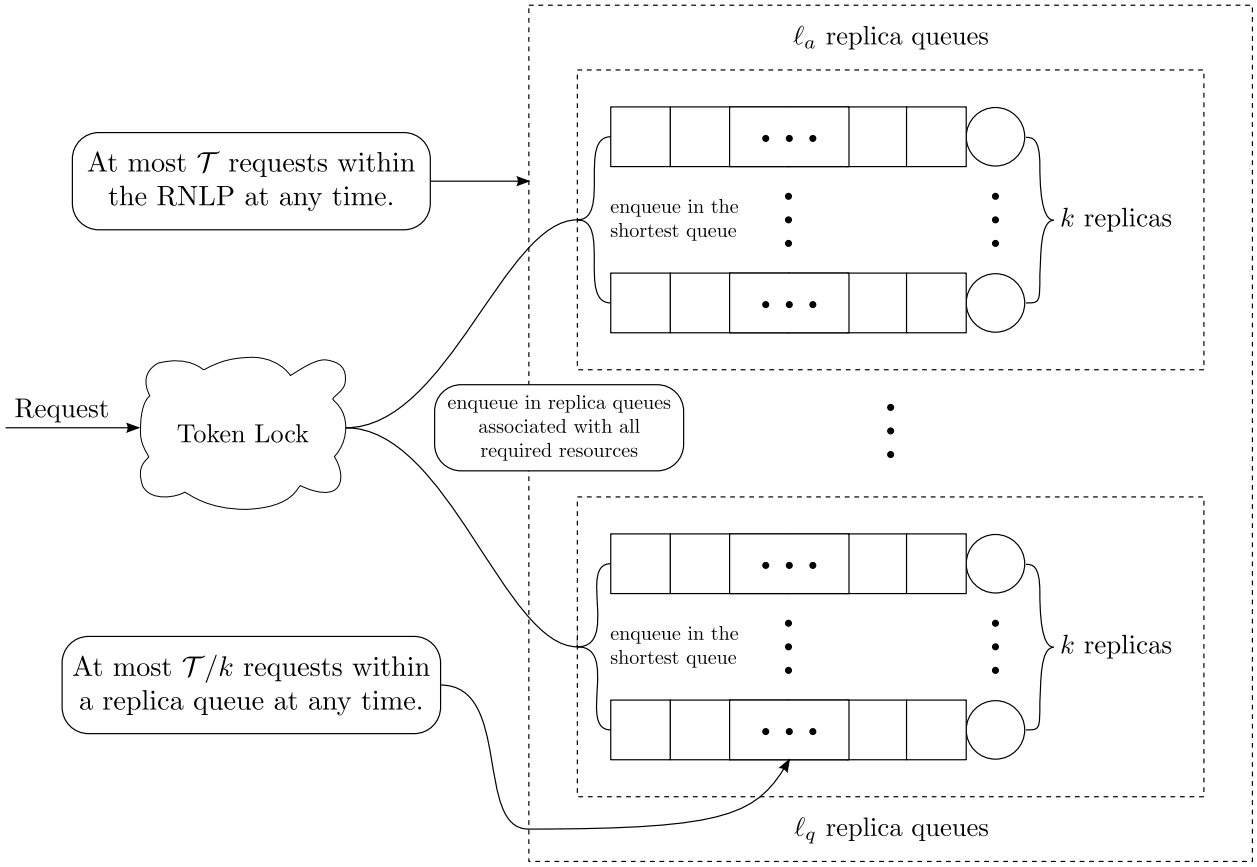


Figure 4.9: Figure illustrating how DGL can be used to request replicas of different resources.

Procedure	Median (μs)	Worst (μs)
read lock	0.106	0.168
read unlock	0.048	0.124
write lock	0.478	0.626
write unlock	0.129	0.215

Table 4.1: Lock and unlock overheads for read and write requests. The worst-case overhead reported is the 99^{th} percentile, to filter the effects of interrupts and other spurious behavior.

thousands of randomly generated task systems to determine the fraction of systems for which it could be shown that no deadlines are missed.

We implemented the R/W RNLP based on the pseudocode presented earlier. The R/W RNLP was run in user-space on top of LITMUS$^{\text{RT}}$ (LITMUS$^{\text{RT}}$ Project, 2016), a real-time extension of Linux. Our implementation was designed for a partitioned scheduler ($c = 1$); the P-EDF scheduler was used in our evaluations.

We evaluated our implementation on a 2.67Ghz quad-core Intel Core i7-920 processor. We measured the *overhead* of the lock and unlock procedures used in the implementation, where such overhead is defined to be the total procedure runtime minus any time spent busy waiting for other requests to complete. In total, we measured overheads for 18 task-system configurations, corresponding to task-system configuration used in the schedulability study below. These task systems were chosen to ascertain how the implementation behaves under high contention, and under different ratios of read to write requests. Each task set was executed for two minutes. The largest median- and worst-case overheads observed across all task sets are reported in Table 4.1. These overheads are sufficiently small to demonstrate that the R/W RNLP can be practically implemented. Furthermore, read requests have smaller overheads than writes, which is desirable for a R/W locking protocol that is best used when reads are more common than writes.

Schedulability. Next, we present an evaluation of the R/W RNLP on the basis of schedulability. These experiments are intended to show the effects that blocking bounds have on schedulability, and do not include overheads.

We randomly generated sporadic task systems using a similar experimental design as previous studies (*e.g.*, (Brandenburg, 2011)). We assume that tasks are partitioned onto $m = 4$ processors, and scheduled in EDF order. We generated task systems with a total system utilization in $\{0.1, 0.2 \ldots, 4.0\}$. Per-task utilizations in a given task system were chosen to be *medium* or *heavy*, which correspond to uniformly distributed utilizations in the range $[0.1, 0.4]$ or $[0.5, 0.9]$, respectively. The periods of all tasks were chosen

uniformly from either $[3,33]$ ms (*short*) or $[50,250]$ ms (*long*). All tasks were assumed to access shared resources, but only $\mathcal{P}^r \in \{50,70,90\}\%$ of the tasks issue read requests. Each read (write) request was configured to access $N^r \in \{1,2,4\}$ (respectively, $N^w \in \{1,2,4\}$) of 50 resources. Read and write critical-section lengths for each job were exponentially distributed with a mean of either $10\,\mu$s (*small*) or $1000\,\mu$s (*large*).

For each generated task set, we evaluated schedulability using four different real-time locking protocols, OMLP mutex group locks (Brandenburg and Anderson, 2010a), the RNLP (Ward and Anderson, 2012), Phase Fair (PF) R/W group locks (Brandenburg and Anderson, 2009, 2011), and the R/W RNLP presented herein. Blocking bounds under each protocol were evaluated using fine-grained analysis similar to that in (Brandenburg, 2011). For the RNLP and the R/W RNLP, additional optimizations were also included, which are based on evaluating possible transitive blocking relationships.

While our experiments generated hundreds of schedulability graphs,[7] here we present in Figure 4.10 a selection that depict relevant trends. In Figure 4.10, the curves denoted NOLOCK depict schedulability assuming no resource requests, while the remaining curves depict schedulability using the locking protocol as labeled. The protocol with a curve closest to NOLOCK provides the best schedulability.

Observation 4.1. In all observed cases, schedulability under the fine-grained locking protocols, the RNLP and the R/W RNLP, was no worse than schedulability using the corresponding coarse-grained locking protocols, the OMLP and phase-fair R/W locks, respectively.

This observation is supported by insets (a) and (b) of Figure 4.10. In inset (a), the R/W RNLP is roughly the same as phase-fair R/W locks, while the RNLP significantly outperforms the OMLP. However, in many cases, such as in inset (b), the fine-grained RNLP and R/W RNLP offer improved schedulability over their coarse-grained counterparts.

Observation 4.2. For read-dominated workloads, *i.e.*, those with larger $\mathcal{P}^r$, phase-fair R/W locks, and the R/W RNLP perform comparatively better. The R/W RNLP performs comparatively better than phase-fair locks when N^r is small.

This observation is supported by inset (a) of Figure 4.10, in which read critical-section lengths are large and write critical-section lengths are small, and 90% of tasks issue read requests. Additionally, in inset (b), in which $N^r = 1$, schedulability under the R/W RNLP is better than under phase-fair locks. Note that the

[7] Available online at `http://www.cs.unc.edu/~bcw/diss/`

gap between phase-fair locks and the R/W RNLP is smaller for larger N^r on account of write requests being forced to request unneeded resources.

Observation 4.3. In some cases in which there are a comparatively large number of write requests, the RNLP offers slightly improved schedulability over the R/W RNLP.

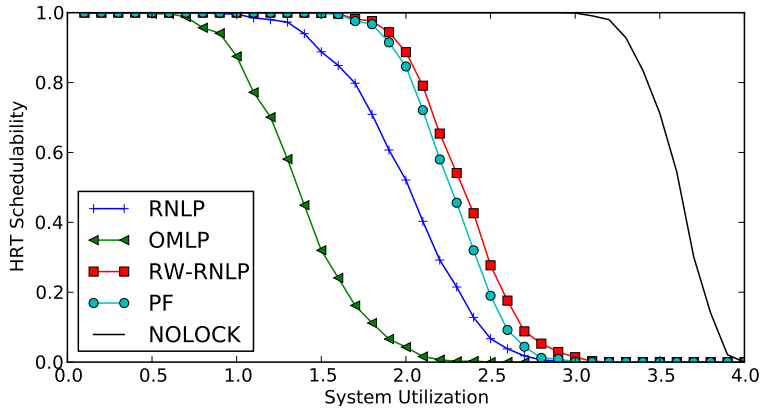
This observation is supported by inset (c) of Figure 4.10, in which 50% of tasks issue write requests. Under the R/W RNLP, writer blocking is $((m-1)(L^r_{\max}+L^w_{\max})$ instead of $(m-1)L_{\max}$ for the RNLP) to allow for $O(1)$ reader blocking. When writes are comparatively common, the benefits of $O(1)$ reader blocking in some cases do not outweigh the cost of increased writer blocking, and thus the RNLP may outperform the R/W RNLP by a small margin.

These schedulability results, in conjunction with our measured overheads, demonstrate that fine-grained mutex and R/W locks are practically implementable, and offer improved schedulability over coarse-grained alternatives.

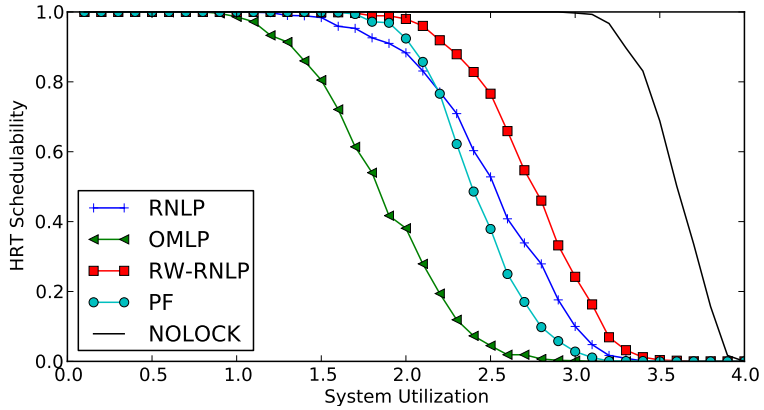
4.3.2 *k*-exclusion RNLP Evaluation

Next we present an experimental evaluation of fine-grained locking via the RNLP and *k*-exclusion RNLP through a schedulability study. In this study, we evaluated the schedulability of randomly generated task systems, and report the fraction that are schedulable. These experiments were designed to depict the effect of blocking bounds on schedulability, and therefore do not include overheads. We note the the RNLP and *k*-RNLP have been implemented and been proven useful in the context of the aforementioned shared-cache (Ward et al., 2013b) and GPU (Elliott et al., 2013) use cases, respectively.

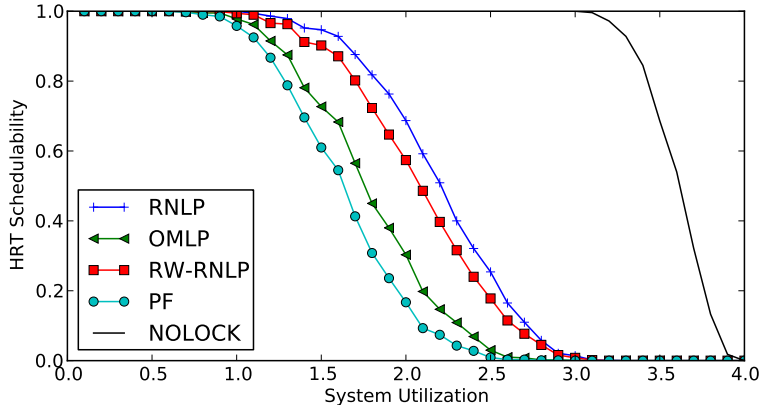
We randomly generated task systems using a similar experimental design as previous studies, similarly to the the previous subsection. We assume that tasks are partitioned onto $m = 8$ processors, and scheduled with EDF priorities. We also assume that all tasks have implicit deadlines ($d_i = p_i$). We generated task systems with total system utilizations in $\{0.1, 0.2, \ldots, 8.0\}$. The per-task utilizations where chosen uniformly from the range $[0.1, 0.4]$ or $[0.5, 0.9]$, denoted, *medium* or *heavy*, respectively. The period of each task was chosen uniformly from either $[3, 33]$ ms (*short*) or $[50, 250]$ ms (*long*). All tasks were assumed to access $N \in \{2, 4, 8\}$ of 16 shared resources. The duration of each critical section was exponentially distributed with a mean of either $10\,\mu$s (*small*) or $1000\,\mu$s (*large*).



(a) $\mathcal{P}^r = 90\%$, $N^r = 2$, $N^w = 4$, large read and small write critical sections.



(b) $\mathcal{P}^r = 50\%$, $N^r = 1$, $N^w = 2$, large read and small write critical sections.



(c) $\mathcal{P}^r = 50\%$, $N^r = 2$, $N^w = 4$, large read and write critical sections.

Figure 4.10: Schedulability results.

For each generated task set, we evaluated HRT schedulability under four different locking protocols: two coarse-grained protocols, the mutex OMLP and the clustered k-exclusion variant of the OMLP (Brandenburg

and Anderson, 2011), the CK-OMLP, and two fine-grained protocols, the RNLP (Ward and Anderson, 2012) and the k-exclusion RNLP variant presented herein (denoted K-RNLP). We also evaluated the schedulability of the task system assuming no critical sections (denoted NOLOCK). For the RNLP variants, fine-grained blocking analysis presented in Sections 3.6 and 4.1.2.3 is applied, specifically the bounds from Corollaries 3.1 and 4.3. We present a subset of our generated graphs in Figures 4.11–4.13, in which all critical-section lengths are large.[8]

Observation 4.4. Schedulability is no worse using a fine-grained locking protocol than a similar coarse-grained one.

This observation is supported by Figure 4.11, which depicts the schedulability of two different system configurations. Inset (a) depicts a system in which fine-grained locking provides little if any schedulability benefit over coarse-grained locking for either mutex or k-exclusion locks. Inset (b), on the other hand, depicts a system in which fine-grained locking provides more significant schedulability benefits owing to the additional analysis optimizations. We note that the blocking bounds for the coarse-grained locking protocols upper bound the worst-case blocking for the fine-grained protocols, and thus the fine-grained protocols will perform no worse than the coarse-grained ones.

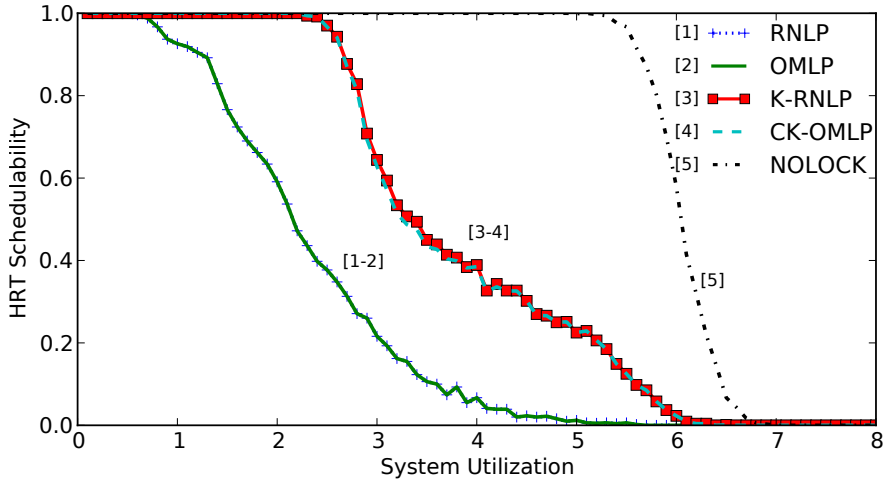
Observation 4.5. Resource replication improves schedulability.

This observation is supported by Figure 4.12, which depicts the schedulability of a given system under different degrees of resource replication. When resources are more highly replicated, more requests can be satisfied concurrently, which decreases blocking bounds. The reduced blocking made possible by resource replication can be reflected in the worst-case blocking bound, which is $O(m/k)$. This improved blocking bound results in improved schedulability, as is seen in Figure 4.12.

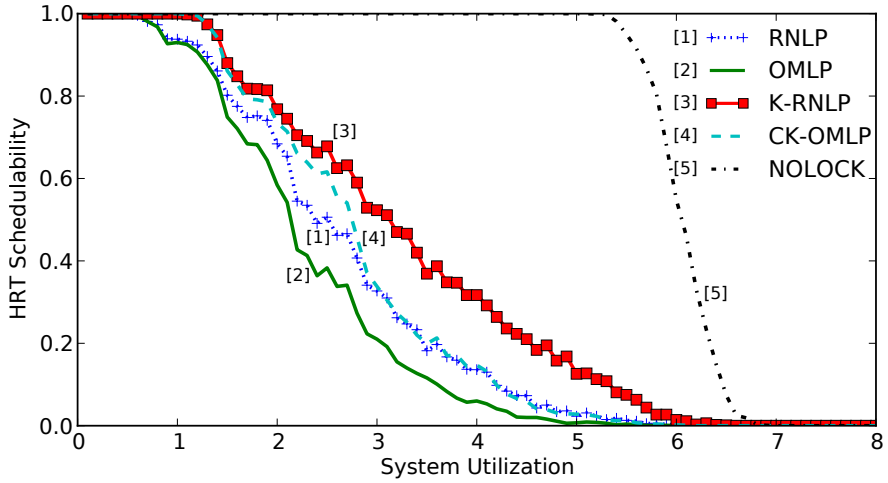
Observation 4.6. Fine-grained locking improves schedulability over coarse-grained locking most when the number of resources accessed within an outermost critical section is small.

This observation is corroborated by Figure 4.13, which depicts the schedulability under the K-RNLP of a given system with tasks requesting different numbers of resources, N. In that particular system, the schedulability when $N = 2$ is considerably better than when $N > 2$, but the benefits of fine-grained nesting diminish with larger N. This is because when the number of resources accessed within a critical section is small, fine-grained locking is more likely to allow non-conflicting requests, which would have

[8]The remainder of the generated graphs are available online `http://www.cs.unc.edu/~bcw/diss/`.



(a) $k = 4$, short periods, $N = 8$, heavy per-task utilizations.



(b) $k = 2$, short periods, $N = 2$, heavy per-task utilizations.

Figure 4.11: Sample schedulability results. Inset (a) demonstrates that fine-grained nesting, in some cases provides little if any advantage over coarse-grained nesting. Inset (b) demonstrates that in other cases, fine-grained nesting can provide more significant schedulability benefits over coarse-grained nesting.

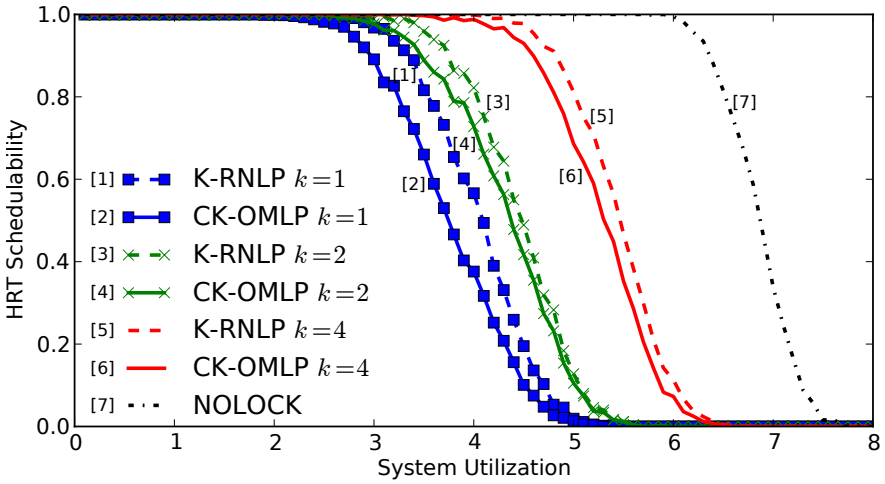


Figure 4.12: Illustration of the improved schedulability made possible with a higher degree of resource replication. In this figure, periods are long, per-task utilizations are medium, and $N = 2$.

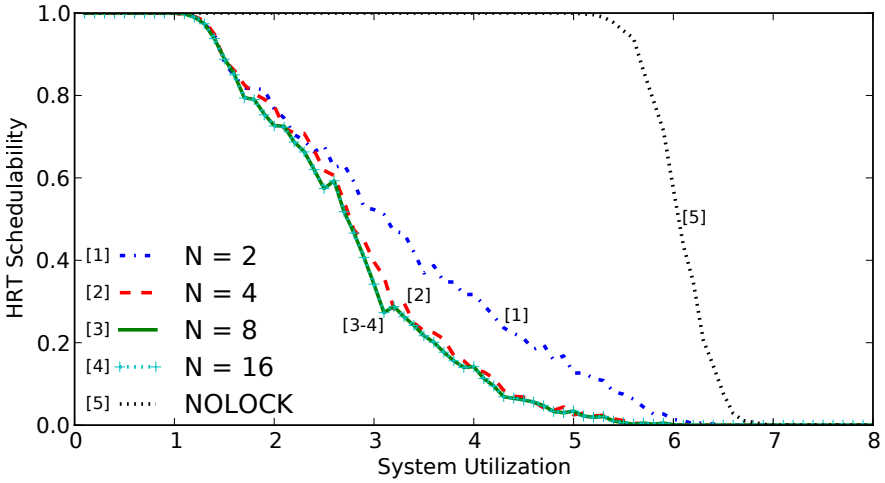


Figure 4.13: Illustration of the effect of the number of resources accessed within an outermost critical section on schedulability. In this figure, periods are short, per-task utilizations are heavy, and $k = 2$.

been serialized under coarse-grained locking, to be satisfied concurrently. In many cases, as is seen in Figure 4.13, this parallelism can be reflected in the blocking analysis (though it does not affect blocking bounds asymptotically). Note also the number of resources accessed within an outermost critical section are often small in practice (Brandenburg and Anderson, 2007). Thus, the cases in which fine-grained locking performs best are the most common in practice.

From these results, we conclude that fine-grained locking protocols offer improved schedulability over coarse-grained ones. Furthermore, we note that even in cases in which fine-grained locking provides no analytical benefit, it is still preferable in practice as it may lead to improved response times and therefore safety margins and responsiveness.

4.4 Chapter Summary

We have presented the R/W RNLP, which is the first fine-grained real-time multiprocessor locking protocol that supports reader/writer sharing. Having to support two different operations on resources—reads and writes—introduces considerable difficulty in designing a fine-grained reader/writer real-time locking protocol. The R/W RNLP resolves the R/W ordering dilemma using the concept of entitled waiting. The R/W RNLP also prevents transitive early-on-late blocking that would increase worst-case pi-blocking bounds.

In addition, we also presented the first fine-grain multi-unit multi-resource locking protocol in the *k*-RNLP. The *k*-RNLP has been applied in practice in the context of locking individual ways of the shared cache in a multicore processor (Ward et al., 2013b), enabling more fine-grained control of resource allocation than would have otherwise been possible.

CHAPTER 5: Synchronization Algorithms for Shared Hardware Resources[1]

In the preceding two chapters, we developed the RNLP family of multiprocessor real-time locking protocols, the first such protocols to support fine-grained locking. While the RNLP can be applied to any type of resource, its design was primarily targeted at shared data objects. Later, the RNLP was applied to control access to cache resources (Ward et al., 2013b). In this chapter, we focus our attention on synchronization algorithms that are primarily motivated by the need to control access to shared hardware resources. While there may be other applications for these algorithms, they were conceived as solutions to hardware-management issues.

As described in Chapter 2, modern computer architectures, particularly multicore systems, include shared hardware resources such as GPUs, caches, and interconnects that introduce timing-interference channels. Unmanaged access to such resources can adversely affect the execution time of other tasks, and lead to unpredictable execution times and associated analysis pessimism that can entirely negate the benefits of a multicore processor. To mitigate such effects, accesses to shared hardware resources should be managed, for example, by a real-time locking protocol. However, the synchronization requirements of these hardware resources may be slightly different than other resources for which real-time locking protocols are traditionally designed. In this chapter, we design several real-time locking protocols motivated by these new synchronization requirements.

The first use case we consider in this chapter is that of shared GPUs. As described in Section 2.3.2.3, real-time locking protocols have been applied to arbitrate access to GPUs (Elliott, 2015). An important characteristic of the resource-usage pattern of GPUs is that GPU critical sections can be very long: typically tens of milliseconds, but even as great as several seconds, in length. These lengths are orders of magnitude greater than those normally associated with shared data structures (Brandenburg, 2011), a more typical

[1]Contents of this chapter previously appeared in preliminary form in the following papers:
Ward, B., Elliott, G., and Anderson, J. (2012). Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 280–289.
Ward, B. (2015). Relaxing resource-sharing constraints for improved hardware management and schedulability. In *Proceedings of the Real-Time Systems Symposium*, pages 153–164.

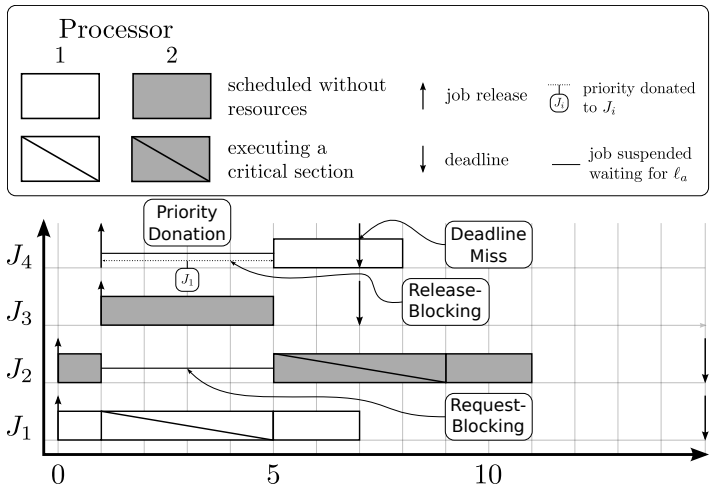


Figure 5.1: Example of the effects of release-blocking under JRPD on a G-EDF-scheduled system with $m = 2$ processors. J_4 is release-blocked at time $t = 1$ while it donates its priority to J_1. This release-blocking causes J_4 to miss its deadline. If instead, J_4 had preempted J_1, J_1 and J_2 would finish three time units later, but still meet their deadlines. Also, at time $t = 1$, J_2 is request-blocked by J_1, which holds the shared resource.

locking-protocol application. These long critical-section lengths motivate the design of locking protocols that minimize or eliminate progress-mechanism-related pi-blocking, which affects all tasks. For example, under priority donation (hereafter referred to as *job-release priority donation* (*JRPD*) for clarity), a latency-sensitive application with a short period may be forced to donate its priority to a long GPU critical section, which may not complete until after the deadline of the latency-sensitive application. This is depicted in Figure 5.1. Clearly, for task systems containing very long critical section, an independence-preserving locking protocol (recall from Section 2.2.3.8, an independence-preserving locking protocol has zero progress-mechanism-related pi-blocking) is necessary. In this chapter, we present a new progress mechanism called *replica-request priority donation* (*RRPD*), which has many parallels to priority donation as described in Section 2.2.3.8, but is independence preserving. Using this new progress mechanism, we construct an independence preserving k-exclusion locking protocol for globally scheduled systems called the *replica-request donation global locking protocol* (*R²DGLP*).[2] Since its inception (Ward et al., 2012), the R²DGLP has been applied in GPUsync (Elliott, 2015) to control access to GPUs.

Accesses to other hardware resources may be managed with more relaxed *sharing constraints* than mutual exclusion or k-exclusion, while still mitigating timing-interference channels. In this chapter, we presents two new classes of sharing constraints, *preemptive mutual exclusion*, and *half-protected exclusion*,

[2]Previously called the I-KGLP in (Ward and Anderson, 2012) and presented in an unpublished online-only appendix.

which are motivated by the sharing constraints of buses and caches, respectively. Synchronization algorithms are presented for both sharing constraints, where applicable, on both uni- and multi-processor systems. A fundamentally new analysis technique called *idleness analysis* is presented to account for the effects of blocking in globally scheduled multiprocessor systems. Experimental results suggest that these relaxed synchronization requirements and improved analysis techniques can improve schedulability by up to 250%. Furthermore, idleness analysis can be applied to existing locking protocols to improve schedulability in many cases.

Organization. In Section 5.1, we present RRPD and the R^{2}DGLP, as well as experimental results pertaining to this new locking protocol. In Section 5.2, we present the preemptive mutual exclusion sharing constraint, as well as a simple synchronization algorithm that realizes that sharing constraint. Idleness analysis is also presented in Section 5.2. In Section 5.3, we present the half-protected exclusion sharing constraint, and present uni- and multi-processor synchronization algorithms that realize half-protected sharing. In Section 5.4, we present an experimental evaluation of preemptive mutual exclusion and half-protected exclusion. Finally, we summarize in Section 5.5

5.1 k-exclusion

In this section, we present RRPD and the R^{2}DGLP. First, we formalize our resource model and assumptions.

5.1.1 Resource Model

In this section, we assume that each resources ℓ_a is a multi-unit resource, with k_a serially reusable units called *replicas*. Therefore, at most k_a requests for ℓ_a may be satisfied at a time, one for each replica. We assume that a job can only request one replica at a time. Note that in the case that $k_a = 1$, the resource is an ordinary serially reusable (mutex) resource. We assume that replica requests are *non-nested*, *i.e.*, a job holding a replica of ℓ_a cannot issue another replica request for another replica of any resource until its current critical section is completed. Fine-grained locking can be supported via the k-exclusion variant of the RNLP, as described in Section 4.2. The R^{2}DGLP is also useful as a token lock in this case.

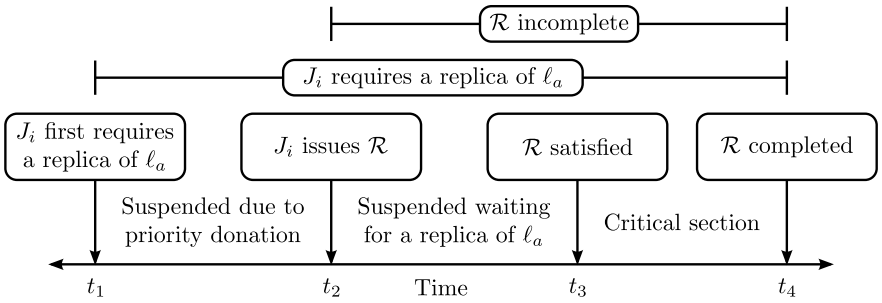


Figure 5.2: Phases of a replica acquisition under RRPD.

In the locking protocol presented in this section, the issuance of $\mathcal{R}$ can be deferred from the first instant at which J_i *requires* a replica of ℓ_a. All previous locking-related definitions persist (*e.g.*, satisfied, incomplete, completed). The phases of replica acquisition considered in this section are depicted in Figure 5.2.

5.1.2 Replica-Request Priority Donation

Job-release priority donation (JRPD) was designed as a progress mechanism for clustered systems (Brandenburg and Anderson, 2011). In a clustered-scheduled system, comparing priorities across clusters is not very useful, because a high-priority job with respect to one cluster may have a relatively low priority when compared to jobs in another cluster. To ensure progress, JRPD ensures that all resource-holding jobs are scheduled by forcing high-priority jobs to suspend and donate their priority upon release to prevent preemptions of resource-holding jobs. However, this donation can cause release-blocking for any job in the system, not just those that engage in the locking protocol. We demonstrate such behavior in Figure 5.1, in which, at time $t = 1$, J_4 is released, and is forced to suspend and donate its priority to J_1, so that J_1 remains scheduled.

In a globally scheduled system, priorities can be compared among all tasks, which allows us to adapt the rules of priority donation such that jobs that do not ever require shared resources are never pi-blocked. Thus, there is no release-blocking. To do so, we modify priority donation such that a job donates its priority on replica request instead of job release. Elliott and Anderson (2013) proposed a similar request-time donation mechanism, however their mechanism resulted in increased blocking bounds as compared to $\mathsf{R^2DGLP}$ we present here. We call this new definition of priority donation, *replica-request priority donation* (*RRPD*). Note that RRPD on its own is not sufficient to ensure progress, but when coupled with a progress mechanism such as priority inheritance, as we will demonstrate with respect to the $\mathsf{R^2DGLP}$ in Section 5.1.3, yields

desirable worst-case pi-blocking bounds. The rules of the RRPD will be demonstrated later in an example of the R^{2}DGLP.

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 replica request. Let t_3 and t_4 be the times that J_i's request is satisfied and completed, respectively. These times are depicted in Figure 5.2. Additionally, let J_d be a priority donor of J_i. Also let T_x be the time that J_d first requires a replica of ℓ_a. J_d suspends to let J_i complete its replica request.

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

D2 J_d becomes J_i's priority donor a time t_x if **(i)** J_d has one of the m highest base priorities among jobs that currently require a replica of ℓ_a, **(ii)** J_i is the lowest effective-priority job[3] with an incomplete request for a replica of ℓ_a at time t_x, and **(iii)** there are m jobs with an incomplete request for a replica of ℓ_a.

D3 J_i assumes the priority of its donor (if any) J_d 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 highest base-priority jobs 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 **(i)** J_i completes its critical section (*i.e.*, at time t_4), or **(ii)** J_d is relieved by Rule D4.

Note that Rules D1-D6 on their own do not necessarily ensure progress. For example, consider a system with two processors, and one replica of ℓ_a, in which two lower-priority jobs J_{l_1} and J_{l_2} have incomplete requests for a replica of ℓ_a, and two higher-priority jobs J_{h_1} and J_{h_2} are the priority donors of J_{l_1} and J_{l_2}, respectively. If J_i is released and has one of the highest m effective priorities in the system, then only one of J_{h_1} and J_{h_2} has sufficient priority to be scheduled. If J_{h_1} has a higher priority than J_{h_2}, but J_{h_1} is not donating to the replica holder, than progress is not ensured. For this reason, we require the rules of a locking protocol that utilizes RRPD to also ensure the following progress property.

[3]Ties are broken according to the scheduler's tie-breaking rules.

RRP1 A job J_i with an incomplete replica request makes progress (*i.e.*, the replica-holding job for which J_i is waiting is scheduled) if J_i has sufficient effective priority to be scheduled.

In the R^{2}DGLP, this property is satisfied through the use of priority inheritance.

Analysis. We next analyze several qualities of RRPD that will later be used to bound the duration of pi-blocking for the R^{2}DGLP.

Lemma 5.1. *A job J_i with an incomplete request for a replica of ℓ_a has one of the m highest effective priorities among jobs that require a replica of ℓ_a (those that have issued a request, as well as those that are suspended waiting to issue a request, as seen in Figure 5.2).*

Proof. By contradiction. Assume J_i has an incomplete request for a replica of ℓ_a but does not have one of the m highest effective priorities among the jobs that require a replica of ℓ_a. Thus there are at least m jobs of higher effective priority than J_i that require a replica of ℓ_a. 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 currently require a replica of ℓ_a issued their replica requests before J_i, then J_i would not have been allowed to issue its request by Rule D1.

Consider a job J_d that is one of the m jobs of higher effective priority than J_i, which is suspended and waiting to issue its request for a replica of ℓ_a. J_d is not suspended by Rule D1, because it has one of the highest m effective priorities among jobs that currently require a replica of ℓ_a. Thus J_d must be suspended by Rule D2, and is thus a priority donor. Therefore J_d has no effective priority by Rule D3, because its priority is being donated to a job with an incomplete request for ℓ_a (*e.g.*, J_i). □

Lemma 5.2. *There are at most m jobs with an incomplete request for a replica of ℓ_a at any time.*

Proof. Assume for contradiction that there are more than m jobs with an incomplete request for a replica of ℓ_a. Let J_i be the job that issued the $(m+1)^{st}$ request for a replica of ℓ_a. By Rule D1, when J_i issued its request, it was one of the m jobs of highest effective priority with an incomplete request for a replica of ℓ_a. However, there were also m jobs with incomplete requests for a replica of ℓ_a, that, by Lemma 5.1, had the highest m effective priorities of jobs that required a replica of ℓ_a. This contradicts the assumption that J_i was allowed to issue a replica request. □

Lemma 5.3. *A job J_i that has one of the highest m base priorities among jobs that currently require a replica of ℓ_a also has one of the highest m effective priorities (with respect to priority donation only) among jobs that currently require a replica of ℓ_a.*

Proof. The only way for a job's effective priority to be increased is through priority donation via Rule D2. Priority donation forms a one-to-one relationship between donor and recipient, in which the effective priority of the recipient is elevated while the effective priority of the donor is reduced to zero by Rule D3. Thus, the highest m effective priorities are equal to the highest m base priorities among jobs that currently require a replica of ℓ_a. Therefore, if a job J_i has one of the highest m base priorities, then J_i is neither a priority donor nor the recipient of priority donation from another job J_d, and thus also has one of the highest m effective priorities (with respect to priority donation only) among jobs that currently require a replica of ℓ_a. □

Lemma 5.4. *Under RRPD, if a job J_i that requires a replica of ℓ_a is pi-blocked waiting for a replica of ℓ_a it either has an incomplete request for a replica of ℓ_a or it is a priority donor.*

Proof. Assume for contradiction that a job J_i is pi-blocked, does not have an incomplete request for a replica of ℓ_a, and is not a priority donor. By Definition 2.2, if J_i is pi-blocked, then it has one of the highest m base priorities in the system, and by Lemma 5.3 is among the set of the highest m effective-priority jobs that need a replica of ℓ_a. Thus, by Rule D1, J_i would issue a request for a replica of ℓ_a. □

Lemma 5.5. *A priority donor J_d can be pi-blocked during priority donation for at most the maximum duration of time that a job can be pi-blocked with an incomplete request for a replica of ℓ_a (i.e., the time between t_2 and t_4 in Figure 5.2), plus one critical section.*

Proof. If J_d is pi-blocked while it is a priority donor, then the recipient of its priority donation J_i has sufficient effective priority to be scheduled. By Property RRP1, J_i makes progress. Thus J_d can be pi-blocked while it is donor for at most the maximum duration of time J_i can be pi-blocked waiting for its request to be satisfied, plus J_i's critical section length. □

The rules of RRPD facilitate the design of a simple, optimal k-exclusion locking protocol, which we present next.

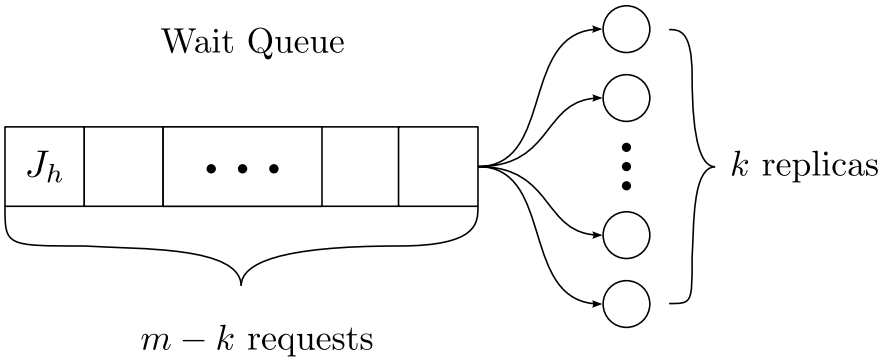


Figure 5.3: Queue structure used by the CK-OMLP, which is suboptimal when used with RRPD. Under JRPD, the progress mechanism employed by the CK-OMLP, all replica-holding jobs are scheduled, and thus the total pi-blocking is at most $\lceil (m-k)/k \rceil L_{\max}$. However, under RRPD, if J_h is the only job with sufficient effective priority to be scheduled, then only one replica-holding job will be scheduled. Thus J_h must wait in the wait queue for $\max((m-k-1)L_{\max}, 0)$ time, which is suboptimal.

5.1.3 R^{2}DGLP

The clustered k-exclusion OMLP (CK-OMLP) (Brandenburg and Anderson, 2011), which employs JRPD, uses a single FIFO-ordered queue to order the acquisition of replicas. Under JRPD, every replica-holding job is scheduled, and thus all requests make progress and the maximum duration of pi-blocking is $O(m/k)$. This design does not extend to RRPD. Under RRPD, a replica-holding job is not guaranteed to be scheduled (if higher-priority work is present), and thus if only one job J_i with an incomplete replica request has sufficient priority to be scheduled, only one replica-holding job would be scheduled. Thus it is possible for all requests to be serialized on a single resource, which results in an $\max(0, (m-k-1)L_{\max})$ blocking bound, which is suboptimal, as shown in Figure 5.3. Instead the R^{2}DGLP employs a similar queue structure to the O-KGLP (Elliott and Anderson, 2013) and the k-FMLP (Elliott, 2015), in which there are k_a queues for each resource ℓ_a, one per replica.

Structure. In the R^{2}DGLP, access to each replica of a resource ℓ_a is arbitrated by an individual FIFO ordered replica queue denoted KQ_x. Within each replica queue, priority inheritance is used to ensure progress. As will be proven later, this design limits the maximum queue length to $\lceil m/k \rceil$, and thus the maximum duration of pi-blocking is $O(m/k)$, which is optimal given the known lower bound of $\Omega(m/k)$ (Brandenburg, 2011, Lemma 6.13). The queue structure of the R^{2}DGLP is shown in Figure 5.4. In the following rules and analysis, we consider, without loss of generality, only a single resource ℓ_a, with k replicas.

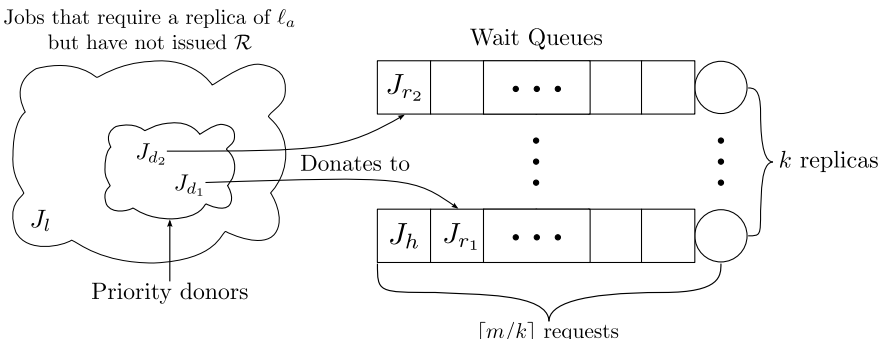


Figure 5.4: Queue structure of the R^{2}DGLP.

K1 J_i is enqueued on the shortest KQ_x when it issues $\mathcal{R}$. J_i suspends until $\mathcal{R}$ is satisfied (if KQ_x was non-empty).

K2 $\mathcal{R}$ is satisfied when J_i becomes the head of KQ_x. A resource-holding job is ready.

K3 The head of KQ_x inherits the highest effective priority (which could be a donated priority) of any job in KQ_x.

K4 J_i is dequeued from KQ_x when $\mathcal{R}$ is completed. The new head of KQ_x, if any, acquires replica x.[4] J_i's priority donor (if any) may then issue a replica request subject to Rule D1.

Example 5.1. Consider a G-EDF-scheduled system on $m = 4$ processors with a single resource ℓ_a with $k_a = 2$ replicas as depicted in Figure 5.5. At time $t = 0$, jobs J_1 and J_2 are released with deadlines of $d_1 = 10$ and $d_2 = 14$. At time $t = 1$, J_3 and J_4 are released with deadlines $d_3 = 15$ and $d_4 = 13$. Also at time $t = 1$, both J_1 and J_2 request and acquire a replica of ℓ_a. At time $t = 2$, J_3 requests a replica of ℓ_a, and is enqueued in KQ_2 and suspends by Rule K1. Then at time $t = 3$, J_5 and J_6 are released with deadlines of $d_5 = d_6 = 12$. At this time, J_1, J_4, J_5, and J_6 have the four highest effective priorities, and therefore are scheduled, even though J_2 is holding a replica of ℓ_a. At time $t = 4$, J_4 requests a replica of ℓ_a, enqueues in KQ_1 and suspends by Rule K1. Also at time $t = 4$, J_5 requests a replica of ℓ_a and because there are $m = 4$ jobs with incomplete replica requests, by Rule D2, J_5 must donate to J_3, the job with the lowest effective priority among the jobs with incomplete requests for a replica of ℓ_a. J_2 then inherits J_3's effective priority, or J_5's priority, by Rule K3. At time $t = 5$, J_1 releases its replica of ℓ_a, allowing J_4, the next job in KQ_1 to acquire a replica of ℓ_a. Also at time $t = 5$, J_6 requests a replica of ℓ_a, and is enqueued in KQ_1. J_4 also inherits J_6's priority by Rule K3.

[4] As an implementation optimization, if KQ_x is left empty after J_i dequeues, a request from another KQ_y can migrate to KQ_x and acquire replica x to reduce average-case pi-blocking. For example, the highest effective-priority job could be chosen to migrate.

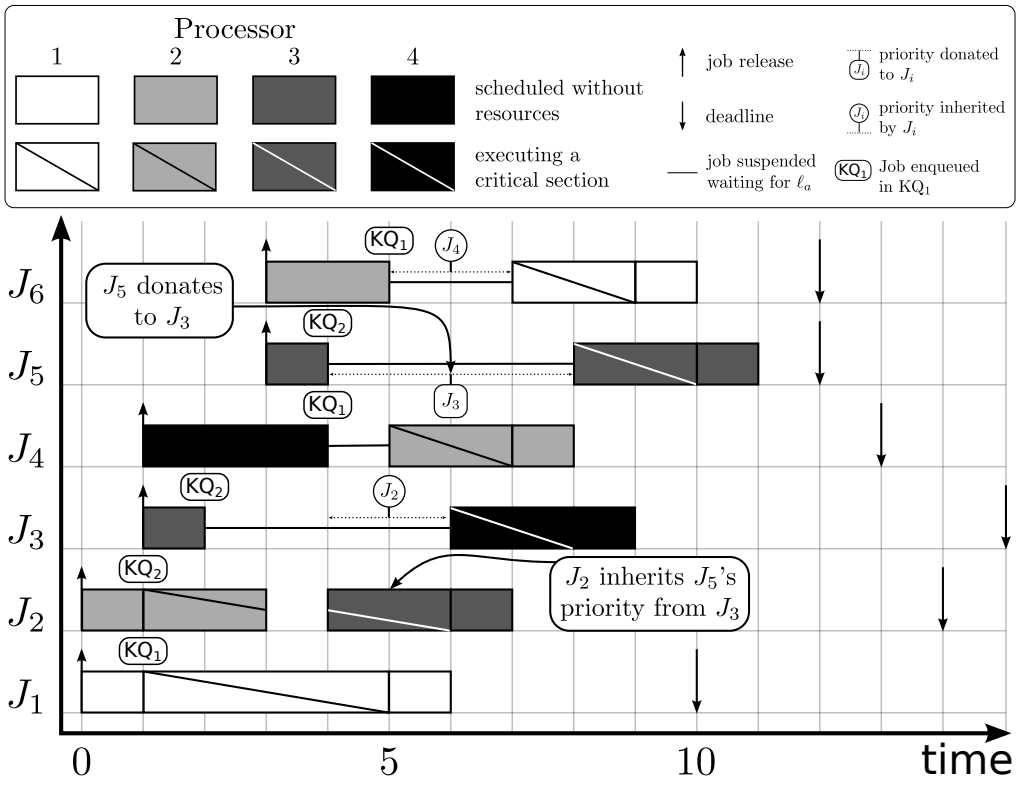


Figure 5.5: Figure depicting the task system in Example 5.1. In this example, $k = 2$ and $m = 4$.

At time $t = 6$, J_2 releases its replica of ℓ_a, and J_3 acquires it. At time $t = 7$, J_4 releases its replica, which allows J_6 to begin its critical section. At $t = 8$, J_3 finishes its critical section, and J_5's donation obligation is finally completed by Rule D6, and it therefore is allowed to request a replica of ℓ_a. At this time, J_5 can immediately acquire its replica and begin its critical section. Finally, at times $t = 9$ and $t = 10$, J_6 and J_5 respectively complete their critical sections and the example returns to ordinary G-EDF scheduling. ◊

Analysis. Next, we analyze the worst-case pi-blocking of the R^{2}DGLP. By Lemma 5.4, if a job is pi-blocked it either has an incomplete replica 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 replica request. We analyze each of these times separately.

Lemma 5.6. *The maximum length of* KQ$_x$ *is* $\lceil m/k \rceil$.

Proof. By Lemma 5.2, there are no more than m jobs with incomplete replica requests. 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 $\lceil m/k \rceil$, otherwise there would have been a shorter queue on which J_i would have enqueued. □

Lemma 5.7. *Rule K3 ensures Property RRP1.*

Proof. If a job J_i with an incomplete request in KQ_x has sufficient effective priority to be scheduled, then by Rule K3, the job at the head of KQ_x inherits J_i's effective priority. Thus the replica holder is scheduled, and J_i makes progress. □

Lemma 5.8. *A job J_i can be pi-blocked for $(\lceil m/k \rceil - 1)L_{\max}$ in KQ_x.*

Proof. By Lemma 5.7, a job that is pi-blocked makes progress. By Lemma 5.6, there are at most $\lceil m/k \rceil - 1$ jobs that are enqueued ahead of J_i in KQ_x. Thus, the maximum duration of pi-blocking in KQ_x is $(\lceil m/k \rceil - 1)L_{\max}$. □

Lemma 5.9. *A job J_d can be pi-blocked for a maximum duration of $\lceil m/k \rceil L_{\max}$ while it is a priority donor.*

Proof. Follows from Lemmas 5.5 and 5.8. □

Theorem 5.1. *The maximum duration of pi-blocking a job J_i can experience waiting for a replica per request is $(2\lceil m/k \rceil - 1)L_{\max}$.*

Proof. Follows from Lemma 5.4, 5.8, and 5.9. □

Note that the O-KGLP (Elliott and Anderson, 2013), the only other known asymptotically optimal k-exclusion locking protocol under s-oblivious analysis that does not cause release-blocking, has a worst-case pi-blocking bound of $(2\lceil m/k \rceil + 2)L_{\max}$. Thus, the locking protocol we present has a worst-case blocking bound that improves upon the O-KGLP by $3L_{\max}$. These blocking bounds can be seen in Table 5.1. As we show next in Section 5.1.4, this improvement can be quite significant when critical sections (*i.e.*, $L_{\max}$) are long.

Additionally, note that when $k = 1$, the blocking bound is $(2m-1)L_{\max}$, which is the same as that of the global OMLP (Brandenburg and Anderson, 2010a). The R^{2}DGLP is therefore a more flexible locking protocol in that it can be used either for a mutex lock, or a k-exclusion lock, both with good blocking bounds.

	Release-blocking	Request-blocking
R^{2}DGLP	0	$(2\lceil m/k \rceil - 1)L_{\max}$
O-KGLP	0	$(2\lceil m/k \rceil + 2)L_{\max}$
CK-OMLP	$\lceil m/k \rceil L_{\max}$	$(\lceil m/k \rceil - 1)L_{\max}$
k-FMLP	$\lceil n/k \rceil L_{\max}$	$(\lceil n/k \rceil - 1)L_{\max}$

Table 5.1: Blocking bounds of several k-exclusion suspension-based locking protocols.

5.1.4 R^{2}DGLP Evaluation

To better understand the schedulability properties of the R^{2}DGLP, we randomly generated task sets with varying characteristics. Soft real-time schedulability under G-EDF scheduling was determined, as described in (Erickson, 2014) for tasks with relative deadlines equal to periods ($d_i = p_i$). We focus our attention on soft real-time schedulability since global schedulers (the only type the R^{2}DGLP supports) are capable of ensuring bounded deadline tardiness in sporadic task systems with no utilization loss. Schedulability was also tested under different locking protocols for comparison. These were the k-FMLP (Elliott, 2015), the CK-OMLP (Brandenburg and Anderson, 2011), and the O-KGLP (Elliott and Anderson, 2013).

Experimental design. The task set characteristics varied by per-task utilization, number of replicas k, critical section length, and number of resource-using tasks in a task set. In all experiments, the system contained a single k-exclusion resource, and each task's period was selected from the range $[3ms, 33ms]$, a common range for multimedia applications. *Utilization intervals* determine the range of utilizations for individual tasks and were $[0.01, 0.1]$ (*light*), $[0.1, 0.4]$ (*medium*), and $[0.5, 0.9]$ (*heavy*). The number of replicas in each case varied among $k \in \{2, 4, 6, 8\}$. *Critical section intervals* determine the range of critical section lengths for resource-using tasks and were $(0\%, 2\%]$ (*very short*), $(0\%, 10\%]$ (*short*), $[10\%, 25\%]$ (*moderate*), and $[50\%, 75\%]$ (*long*), where critical section length is a percentage of e_i. The moderate and long intervals are inspired by GPU-usage patterns in which there are k GPUs (Elliott and Anderson, 2012) (our motivating use case) while very short and short intervals may be common for other shared resources. *Resource usage percentage intervals* determine the number of tasks in a task set that use a resource protected by the k-exclusion lock and vary in increments of 10% from 0% to 100%. Each combination of these four parameters resulted in an experimental scenario. Each scenario was used to evaluate schedulability under each locking protocol on an eight-CPU system. For example, one such scenario tested schedulability for task sets with

light utilizations, $k = 4$ replicas, *short* critical section intervals, where 50% to 60% of tasks required the use of a replica of the shared resource. A total of 432 experimental scenarios were run.

We generated random task sets for each experimental scenario in the following manner. First, we selected a total system utilization cap uniformly in the interval $(0, 8]$ capturing the possible system utilizations on a platform with eight CPUs. We then generated tasks by making selections uniformly from the intervals in each scenario. Per-task utilization was selected from the scenario's utilization interval. Execution times were derived from the selected utilization and period. We added the generated tasks to a task set until the set's total utilization exceeded the utilization cap, at which point the last-generated task was discarded. Next, we designated some tasks to use the shared resource; we determined the number of resource-using tasks by selecting a percentage from the resource usage percentage interval of the scenario. A critical section length for each resource-using task was selected from the scenario's critical section interval. Bounds on pi-blocking were computed using detailed analysis similar to that presented in (Brandenburg, 2011) for each tested locking protocol. As per s-oblivious analysis, task execution times were inflated by their respective pi-blocking bounds (*i.e.*, $e_i^{inflated} = e_i + b_i$, where b_i is the pi-blocking bound of τ_i), prior to performing the soft real-time schedulability test. Tardiness bounds were computed using the method developed by Erickson *et al.* since it offers the tightest know tardiness bounds for G-EDF schedulers (Erickson et al., 2014). These tardiness bounds were incorporated into fixed-point iterative schedulability tests. Fixed-point tests are necessary because tardiness can affect bounds on pi-blocking, which in turn can increase tardiness. Thus, tight tardiness bounds can improve soft real-time schedulability analysis.

Results. A selection of results that demonstrate observable trends across all scenarios is presented here.[5] We found that trends were most clearly expressed in scenarios using *light* utilizations since this resulted in task sets with more tasks.

Observation 5.1. Schedulability is poor under the C-OMLP when critical section lengths are long and when there are relatively few resource-using tasks.

Figure 5.6 depicts schedulability under a scenario where a small percentage of tasks use a resource, yet each of these have long critical sections. Under the C-OMLP, all non-resource-using tasks experience release-blocking from $\lceil m/k \rceil$ replica requests. This negatively affects schedulability in all cases, but is particularly harmful when critical section lengths are long, as is the case in Figure 5.6. Here, only 20%

[5] Additional schedulability graphs are available online: `http://www.cs.unc.edu/~bcw/diss/`.

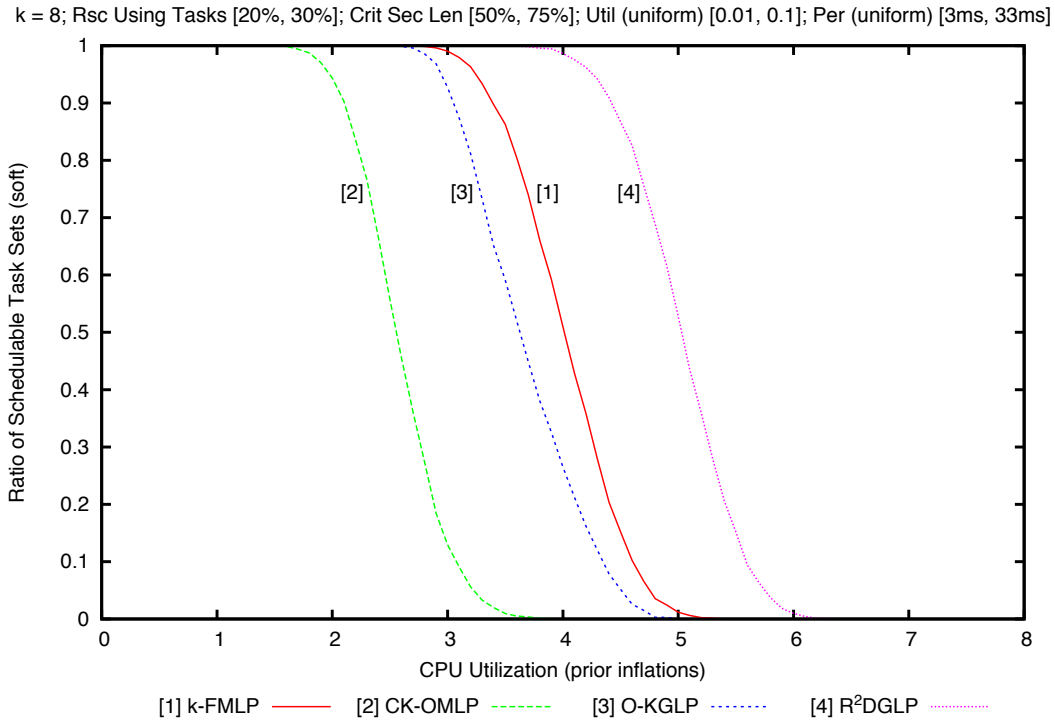


Figure 5.6: The R^{2}DGLP dominates both the CK-OMLP and O-KGLP. This scenario highlights that release-blocking affects negatively affect the CK-OMLP.

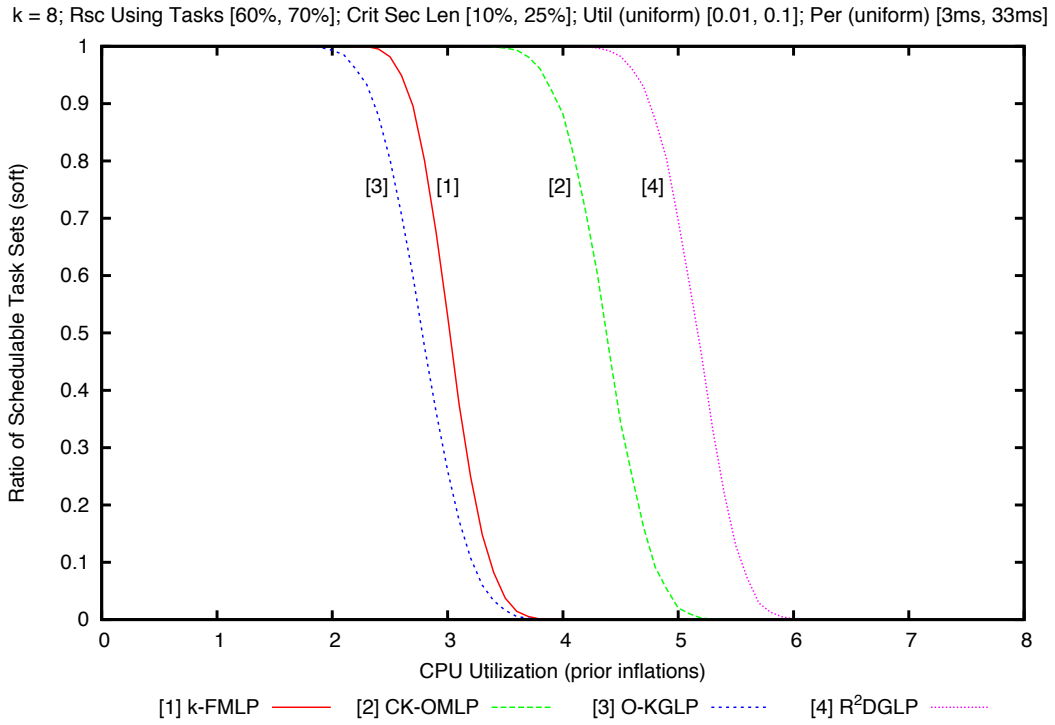


Figure 5.7: The R^{2}DGLP dominates both the CK-OMLP and O-KGLP. The O-KGLP performs poorly due to additional blocking for resource-using tasks

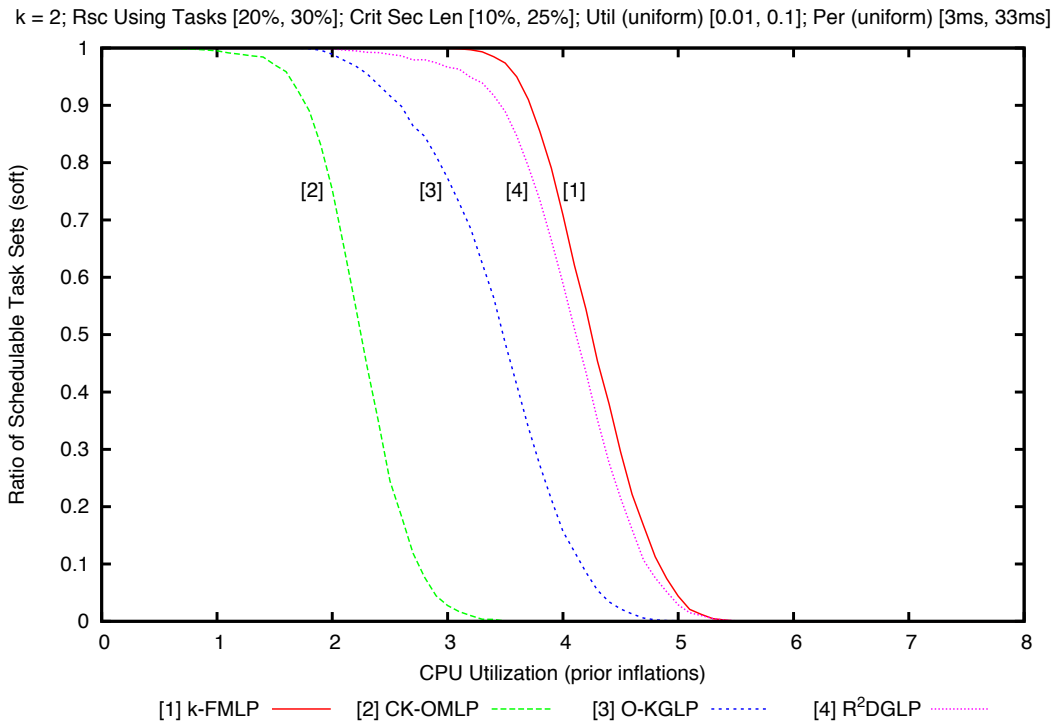


Figure 5.8: The R^{2}DGLP dominates both the CK-OMLP and O-KGLP, and the O-KGLP dominates the CK-OMLP. However, the k-FMLP outperforms the R^{2}DGLP (occurred in about 14% of tested scenarios).

to 30% of tasks in every task set use a resource, but critical section lengths are long, ranging from 50% to 75% of execution time. Roughly 50% of task sets with utilizations of 5.0 are schedulable under the R^{2}DGLP. In contrast, approximately 50% of task sets with utilizations of 2.5 (half that of the R^{2}DGLP) are schedulable under the CK-OMLP. Further, no task sets with utilizations greater than 4.0 are schedulable under the CK-OMLP.

Observation 5.2. The R^{2}DGLP improves upon the O-KGLP, especially when k is large.

Figure 5.6 also depicts schedulability under a scenario where k is large. Though both the R^{2}DGLP and O-KGLP are asymptotically optimal, the R^{2}DGLP significantly outperforms the O-KGLP. Under the R^{2}DGLP, resource-using tasks can experience request-blocking from $2\lceil m/k \rceil - 1$ other replica requests (Theorem 5.1). In comparison, these tasks experience request-blocking from $2\lceil m/k \rceil + 2$ other replica requests under the O-KGLP; three more requests than the R^{2}DGLP. When k is large, the addition of three requests to the blocking term has a stronger affect on schedulability. For example, in the scenario depicted in Figure 5.6, $m = 8$ and $k = 8$, so under the R^{2}DGLP, requests are pi-blocked by only a single request versus four requests under the O-KGLP.

Observation 5.3. Schedulability under the C-OMLP can be better than the O-KGLP when there are many resource-using tasks.

Figure 5.7 illustrates a scenario where the number of resource-using tasks is relatively high and the C-OMLP can outperform the O-KGLP. Here, we begin to see trade-offs between release-blocking and request-blocking between these two protocols. Like the R^{2}DGLP, resource-using tasks can experience pi-blocking from $2\lceil m/k \rceil - 1$ replica requests under the C-OMLP, though non-resource-using tasks also experience pi-blocking (Observation 5.1). Resource-using tasks experience pi-blocking from three additional requests under the O-KGLP, but non-resource-using tasks experience no pi-blocking. When the relative number of resource-using tasks is high, the blocking effects from non-resource-using tasks is decreased, while the blocking effects from resource-using tasks are magnified. Thus, neither the C-OMLP or O-KGLP dominates the other in all scenarios. However, the R^{2}DGLP dominates both of these in all scenarios.

Observation 5.4. The R^{2}DGLP strictly dominates the C-OMLP and O-KGLP when jobs only issue a single request.

When jobs only make one replica request, the R^{2}DGLP offers the best schedulability of known optimal k-exclusion locking protocols for globally-scheduled JLFP systems. The dominance of the R^{2}DGLP over the C-OMLP and O-KGLP in this case can be observed in Figures 5.6, 5.7, and 5.8. The R^{2}DGLP exhibits the best aspects of both the C-OMLP and O-KGLP: resource-using jobs can experience pi-blocking from $2\lceil m/k \rceil - 1$ replica requests, and non-resource-using jobs experience no pi-blocking.

Note that if individual jobs issue many requests for shared resources than the C-OMLP may have better schedulability. In the C-OMLP, jobs can be blocked upon release, however, subsequent requests can only be blocked for $(\lceil m/k \rceil - 1)L_{\max}$ instead of $(2\lceil m/k \rceil - 1)L_{\max}$. Therefore, if jobs issue many short requests, the C-OMLP may be favorable to the R^{2}DGLP.

Observation 5.5. The k-FMLP sometimes outperforms the R^{2}DGLP.

Figure 5.8 illustrates a scenario where the k-FMLP outperforms the R^{2}DGLP, despite not being asymptotically optimal. This occurred in about 14% of the tested scenarios. While the R^{2}DGLP is asymptotically optimal, there are cases where the k-FMLP can offer better schedulability. This is due to two aspects of the k-FMLP. First, resource-using tasks can experience pi-blocking from $\lceil n_a/k \rceil$ requests, where n_a is the number of tasks that may request ℓ_a. Thus, the k-FMLP can outperform the R^{2}DGLP when n_a is sufficiently small. Second, due to the FIFO ordering of all requests, a task can be pi-blocked by at most one request per task

under the k-FMLP. Due to donation mechanisms, a task can be pi-blocked by at most two requests per task under the $\mathsf{R}^2\mathsf{DGLP}$, even though the total number of requests that may pi-block a task is $2\lceil m/k \rceil - 1$. A task may experience less pi-blocking under the k-FMLP if there is a high degree of variance in critical section lengths.

Minimum response-time constraints. As discussed at the beginning of the chapter, release-blocking imposes a minimum response time on all tasks, including those that do not use resources. This minimum response time is on the order of several of the longest critical sections. As a result, non-resource-using tasks must have similar response-time constraints to those of resource-using tasks. For implicit-deadline systems, this means that all tasks, resource-using and non-resource-using, must have similar periods. This limits system flexibility since not all potential applications have this characteristic. In order to highlight this limitation, we selected a scenario from our prior experiments and scaled the execution times and periods of *non*-resource-using tasks by a scaling factor, $s \in \{1/2, 1/4, 1/8\}$. Thus, task utilization remained constant, yet response-time constraints on non-resource-using tasks become more stringent with smaller scaling factors. The results of this are depicted in Figure 5.9. It is easily observed that release-blocking negatively affects schedulability of the CK-OMLP. In contrast, the remaining protocols (which only incur request-blocking) are unaffected.

This concludes our discussion of k-exclusion locking and the $\mathsf{R}^2\mathsf{DGLP}$.

5.2 Preemptive Mutual Exclusion

In this section, we present the preemptive mutual exclusion sharing constraint, a synchronization algorithm realizing this constraint, as well as idleness analysis. First, we introduce relevant models, assumptions, and definitions.

5.2.1 Resource Model

In this section, we assume there to be n_r non-processor shared resources, $\ell_1, \ldots, \ell_{n_r}$. Later, we formalize the preemptive mutual exclusion sharing constraints for these resources. A job safely accessing a resource given the assumed sharing constraints is said to be in a *critical section*. We assume that jobs occupy a processor while executing critical sections. For simplicity, unless otherwise stated, critical sections are assumed to be non-nested, *i.e.*, only one resource may be accessed at a time. (In Section 5.3, we briefly

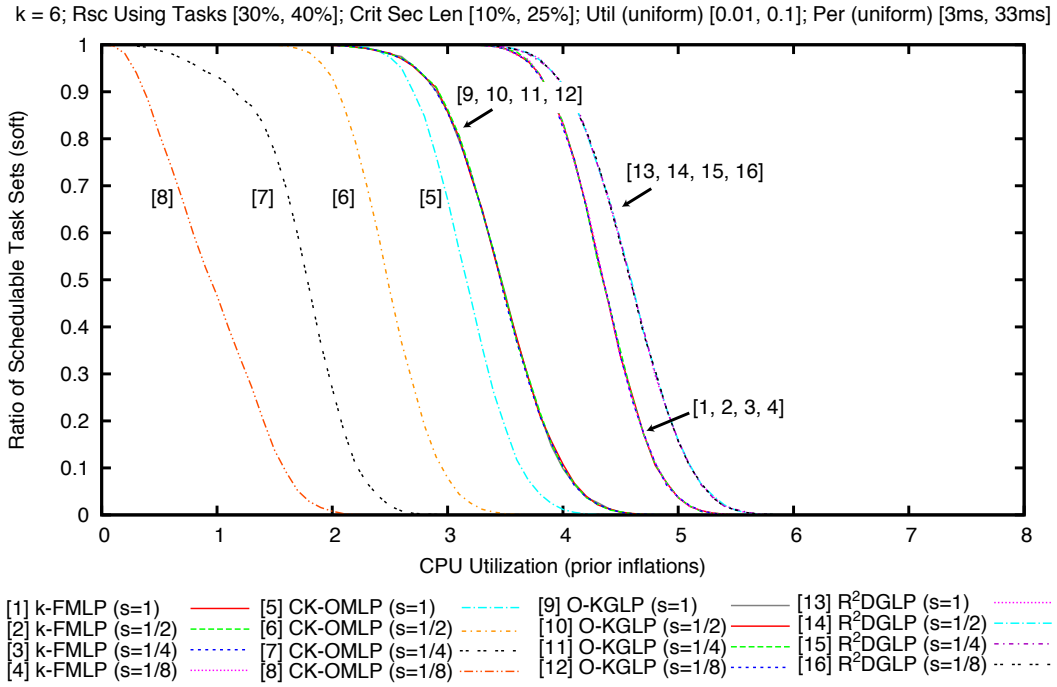


Figure 5.9: Release-blocking imposes a minimum response time on all tasks. This negatively affects schedulability of tasks with varying response-time constraints.

discuss how our analysis could be extended to nested requests.) A job is composed of alternating critical sections and non-critical sections in which only a processor is used. As in previous chapters, a job of τ_i that requires access to ℓ_q must issue a resource request to a locking protocol before it can execute its critical section. A request may be blocked, or forced to wait, by the locking protocol to ensure that the sharing constraint is not violated. The total length of all critical sections for ℓ_q by a job of task τ_i is given by $L_{i,q}$. We make no other assumptions in the remainder of this chapter as to the frequency or duration of critical sections within each job. Critical-section execution times are incorporated in e_i. A task τ_i's *per-resource utilization* of ℓ_q is defined as $u_i^{\ell_q} = L_{i,q}/p_i$, and its *total resource utilization* is given by $u_i^R = \sum_{q=1}^{n_r} u_i^{\ell_q}$. The total resource utilization of all tasks is $U^R = \sum_{\tau_i \in \Gamma} u_i^R$. Also, L_Σ denotes the sum of the $m-1$ largest $\sum_{q=1}^{n_r} L_{i,q}$. Formally,

$$L_\Sigma = \sum_{k=1}^{m-1} \phi_k, \tag{5.1}$$

where ϕ_k is the k^{th} largest value of $\sum_{q=1}^{n_r} L_{i,q}$ for all $\tau_i \in \Gamma$. Also, let $L_{all} = \sum_{\tau_i \in \Gamma} \sum_{q=1}^{n_r} L_{i,q}$. Also let e_Σ denote the sum of the $m-1$ largest per-task execution times, and e_{all} be the sum of all per-task execution times.

Similarly, L_Σ denotes the sum of the $m-1$ largest per-task total critical-section lengths, and L_{all} denotes the sum of all the per-task critical-section lengths.

We also formalize the following definitions.

Definition 5.1. Job $J_{l,j}$ is *pending* at time t if $a_{l,j} \leq t \leq f_{l,j}$.

(Recall from Chapter 2, $a_{l,j}$ is the release time of $J_{l,j}$, and $f_{l,j}$ is the completion time of $J_{l,j}$.)

Definition 5.2. Job $J_{l,j}$ is *ready* at time t if it is pending, and it is not blocked waiting for a shared resource.

Definition 5.3. Job $J_{l,j}$ is *CPU-preempted* at time t if it is ready, but it does not execute at t.

Preemptive mutual exclusion and half-protected exclusion (described in Section 5.3) sometimes allow a job executing a critical section to be preempted with respect to the resource it is accessing, thereby allowing a higher-priority task to access the shared resource. Such a preemption is both a CPU preemption, as the job must suspend execution, as well as a *resource preemption*.

Preemptive mutual exclusion. Locking protocols are often used to arbitrate access to non-processor shared resources such as memory objects. Such protocols guarantee that each critical section executes entirely with exclusive access to the locked resource before any other access is allowed. We call such a sharing constraint *non-preemptive mutual exclusion*. In this section, we consider a weaker sharing constraint called *preemptive mutual exclusion*, which is applicable to *preemptive resources*.

A preemptive resource is one for which at most one task can access the resource at any time t, but accesses to that resource may be preempted, or paused, and later resumed. Preemptive resources differ from non-preemptive ones in that non-preemptive mutex resources require that no other job accesses the resource between the start and end of each critical section. Preemptive resources are motivated by bus scheduling, in which multiple tasks may need to transmit data on a bus such as the memory bus, PCIe bus, or a network link. Such transmissions can be "paused" to allow higher-priority transmissions to access the bus. We note that there may be systems-level considerations associated with pausing a transmission, just as there are systems-level considerations associated with scheduling real-time tasks on any physical processor. In this work we build a formal foundation for preemptive non-processor resources.

5.2.2 Preemptive Mutual Exclusion on Uniprocessors

On a uniprocessor platform, optimally synchronizing access to preemptive resources is trivial—the currently executing task implicitly has exclusive access to all preemptive resources. If a task executing a

preemptive critical section is processor preempted, it is implicitly preempted from all preemptive resources. Therefore, preemptive resources on uniprocessors can be supported with zero pi-blocking (s-oblivious or s-aware), as they are trivially supported by existing uniprocessor scheduling algorithms.

5.2.3 Preemptive Mutual Exclusion on Multiprocessors

In contrast to the uniprocessor case, preemptive resources must be explicitly synchronized on multiprocessor platforms. If two tasks executing concurrently on different processors both need access to the same preemptive resource, only one can execute at a time; the other task(s) must wait. We demonstrate this behavior in the following example.

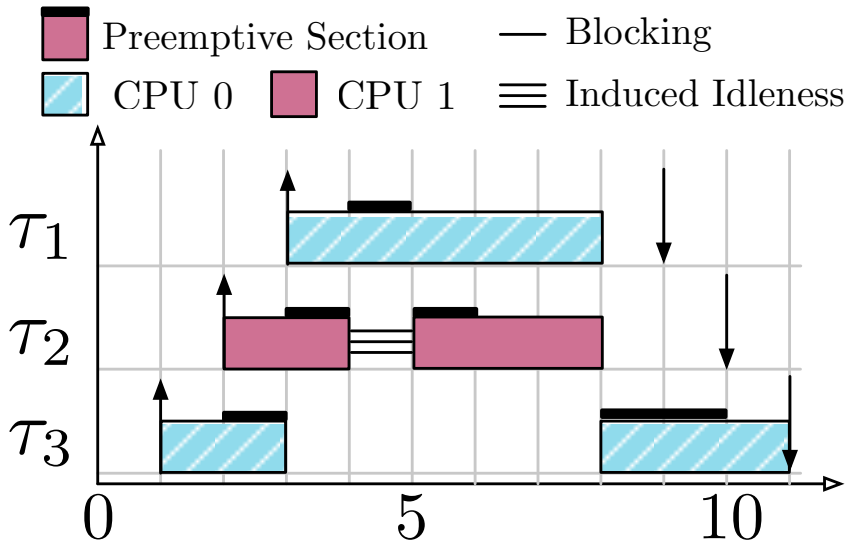
Example 5.2. Consider three EDF-scheduled tasks on two processors and one EDF-prioritized preemptive resource, ℓ_p, as shown in Figure 5.10(a). At time $t = 1$, τ_3 is released, and at time $t = 2$, it acquires ℓ_p. At time $t = 2$, τ_2 is released. At time $t = 3$, τ_2 acquires ℓ_p, thereby preempting τ_3 from both the processor as well as ℓ_p. Also at time $t = 3$, τ_1 is released. At time $t = 4$, τ_1 preempts τ_2 with respect to ℓ_p. Note that during the interval $[4,5)$, ℓ_p *induces idleness* on CPU 1, as there are two suspended tasks waiting to access ℓ_p. At time $t = 5$, τ_1 completes its preemptive critical section, and τ_2 resumes its access of ℓ_p. At time $t = 6$, τ_2 completes its preemptive critical section, but τ_3 does not resume its preemptive critical section until time $t = 8$, when τ_1 completes, relinquishing CPU 0 to τ_3. $\Diamond$

Access to preemptive resources can be arbitrated by a non-preemptive mutex locking protocol, as the sharing constraint of a preemptive resource is weaker than a non-preemptive mutex. However, preemptive sharing can reduce blocking, and therefore improve schedulability. This is demonstrated in Figure 5.10(b), where the same task set shown in Figure 5.10(a) is scheduled using a non-preemptive locking protocol,[6] and τ_1 misses a deadline on account of excessive blocking.

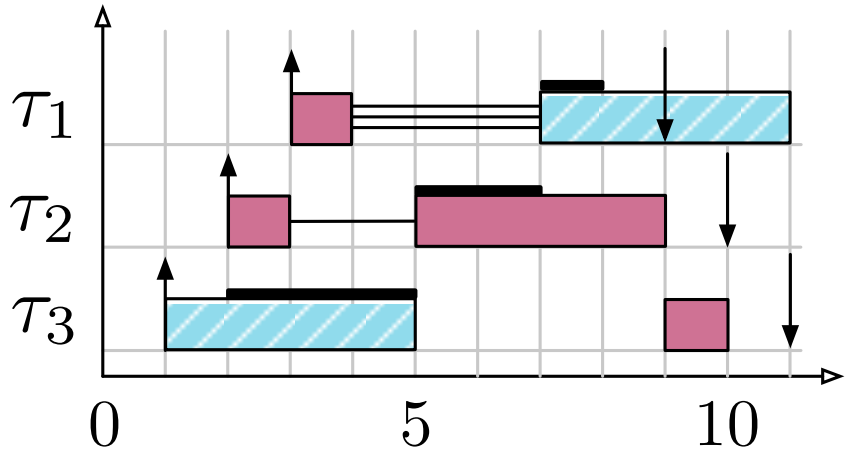
5.2.4 Schedulability Analysis

We next provide schedulability analysis for scheduling with EDF-prioritized preemptive resources. This analysis is based on a novel technique called *idleness analysis* for accounting for the effects of blocking in schedulability analysis. In contrast to either s-oblivious or s-aware blocking analysis described in Chapter 2, in this section, we take a different approach entirely—we apply idleness analysis in place of blocking analysis.

[6] This schedule would result from using either the FMLP$^+$ (Brandenburg, 2014) or the global OMLP (Brandenburg and Anderson, 2014), which are optimal under s-aware and s-oblivious analysis, respectively.



(a) Fully preemptive resource example.



(b) Schedule from (a) if ℓ_p were non-preemptive.

Figure 5.10: Comparison of non-preemptive and preemptive scheduling of a shared resource. Induced idleness (Definition 5.6) occurs when a processor is idle and there are at least m pending jobs.

Instead of determining the worst-case per-request pi-blocking, we instead determine the maximum amount of induced idleness that can occur in an interval on account of synchronization. We then analytically treat idleness as demand, similar to s-oblivious analysis.

Our analysis builds upon the previous G-EDF schedulability analysis of Baruah (Baruah, 2007) and Liu and Anderson's (Liu and Anderson, 2013) extension to self-suspending tasks. We begin with several definitions.

Definition 5.4. A processor is *busy* if it is executing a job, and *idle* if it is not. A time instant t is said to be *busy* if all processors are busy, and *idle* if at least one processor is idle.

Definition 5.5. An idle instant t is *truly idle* if there are at most $m-1$ pending tasks, and *effectively busy* if there are m or more pending tasks.

Definition 5.6. Idleness that occurs in an effectively busy instant is said to be *induced* by blocking. We call such idleness *induced idleness*.

Example 5.2 (continued). As depicted in Figure 5.10(a), the system is busy at time $t = 7$, as both processors are in use, and idle at time $t = 4.5$ and $t = 10$. At time $t = 4.5$, all three tasks are pending, but there is idleness induced by blocking, so $t = 4.5$ is effectively busy. By comparison, at time $t = 10$, τ_3 is the only ready job, and therefore $t = 10$ is truly idle.

Next, we derive sufficient conditions that ensure that no task misses any deadline. Suppose that $J_{l,j}$ is the first job of τ_l to miss a deadline, and let $t_d = d_{l,j}$. Let $t_a = a_{l,j}$ denote the arrival time of $J_{l,j}$. $J_{l,j}$ necessarily misses its deadline if it executes for strictly less than e_l time units over the interval $[t_a, t_d)$. Job $J_{l,j}$ may be prevented from executing if it is preempted or blocked by higher-priority work. Therefore, we disregard all jobs with deadlines later than t_d, as they do not affect the scheduling of $J_{l,j}$. (Note that with alternative critical-section prioritizations, discarding lower-priority jobs is not safe, as will be considered in Section 5.3.) Consequently, any blocking of $J_{l,j}$ is induced idleness.

Definition 5.7. Let t_o denote the last truly idle time instant before t_a. Let $[t_o, t_d)$ be the *analysis interval*, and let $A_l = t_a - t_o$.

Over the interval $[t_o, t_a)$, $J_{l,j}$ is not pending, and therefore cannot execute. By construction, the entire interval $[t_o, t_a)$ is effectively busy.

Definition 5.8. Let Θ denote a collection of (possibly non-contiguous) intervals contained within $[t_a, t_d)$, in which $J_{l,j}$ does not execute. If the cumulative length of Θ exceeds $d_l - e_l$, then $J_{l,j}$ may miss its deadline.

By Definition 5.8, the total length of the intervals in $[t_o, t_a) \cup \Theta$ is at most $A_l + d_l - e_l$ otherwise $J_{l,j}$ is unschedulable. At any time instant $t \in [t_o, t_a) \cup \Theta$, all processors are effectively busy, *i.e.*, either executing work, or idle on account of blocking. Let $W(\tau_i)$ denote the higher-priority work contributed by τ_i to the interval $[t_o, t_a) \cup \Theta$. Similarly, let $I_q(\tau_l)$ denote the total amount of idleness induced in $[t_o, t_a) \cup \Theta$ due to resource ℓ_q. When quantifying idleness, we sum across all processors, *e.g.*, if idleness is induced on two processors during $[0, 1]$, this is two units of idleness, not one. At any time instant $t \in [t_o, t_a) \cup \Theta$, the m processors can either be busy executing demand or idle on account of induced idleness. Thus, in order for $J_{l,j}$ to miss a deadline, it is necessary that the total amount of work and induced idleness in the interval $[t_o, t_a) \cup \Theta$ satisfies

$$\sum_{q=1}^{n_r} I_q(\tau_l) + \sum_{\tau_i \in \Gamma} W(\tau_i) > m(A_l + d_l - e_l). \tag{5.2}$$

To ensure that no task will ever miss a deadline, Condition (5.2) must not hold for any τ_l or value of A_l. The general proof framework codified by Condition (5.2) is depicted in Figure 5.11. We next derive a corresponding schedulability test. Specifically, we review previous work that considers the work from higher-priority jobs, $W(\tau_i)$ in Section 5.2.4.1, and then present new analysis for induced idleness $I_q(\tau_l)$ in Section 5.2.4.2.

5.2.4.1 Demand

This subsection is largely a review of previous results, and therefore proofs are omitted. Similar to previous work (*e.g.*, (Baruah, 2007)), there are two types of tasks to consider, those with *carry-in* work, *i.e.*, demand that had been released before t_o, and those without carry-in work. First, we review the classic demand-bound function (Baruah et al., 1990).

Lemma 5.10. *The maximum cumulative execution requirement by jobs of a sporadic task τ_i that both arrive in, and have deadlines within any interval of length t is given by the demand-bound function*

$$DBF(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) e_i\right) \tag{5.3}$$

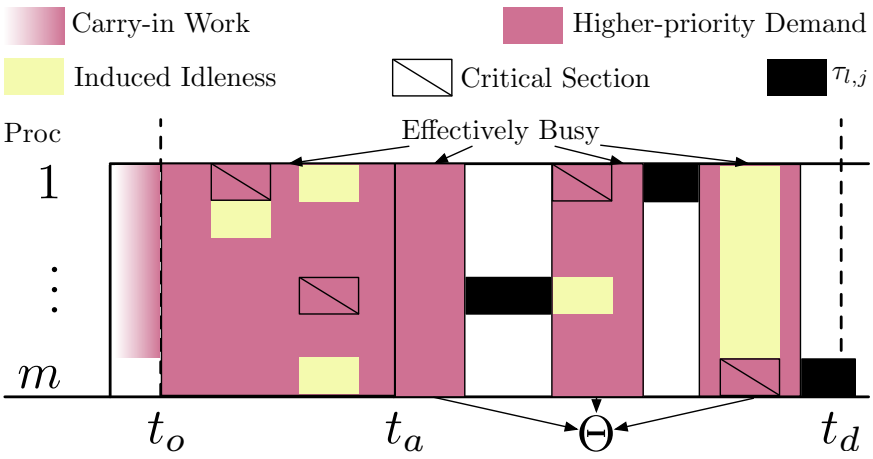


Figure 5.11: Depiction of the overall proof framework.

We denote the workload contributed by τ_i over the interval $[t_o, t_a) \cup \Theta$ assuming no carry-in job as $W_{nc}(\tau_i)$. Using the demand-bound function, we can derive the following bound on the work contributed by τ_i over $[t_o, t_a) \cup \Theta$.

Lemma 5.11. ***(Baruah, 2007).***

$$W_{nc}(\tau_i) = \begin{cases} \min(DBF(\tau_i, A_l + d_l), A_l + d_l - e_l) & i \neq l \\ \min(DBF(\tau_i, A_l + d_l) - e_l, A_l) & i = l \end{cases} \tag{5.4}$$

For tasks with carry-in jobs, the demand-bound function is modified, to account for carry-in demand.

$$DBF'(\tau_i, t) = \left\lfloor \frac{t}{p_i} \right\rfloor e_i + \min(e_i, t \bmod p_i) \tag{5.5}$$

Using (5.5), we can derive the workload $W_c(\tau_i)$ contributed to the interval $[t_o, t_a) \cup \Theta$ by a carry-in task τ_i.

Lemma 5.12. ***(Baruah, 2007)***

$$W_c(\tau_i) = \begin{cases} \min(DBF'(\tau_i, A_l + d_l), A_l + d_l - e_l) & i \neq l \\ \min(DBF'(\tau_i, A_l + d_l) - e_l, A_l) & i = l \end{cases} \tag{5.6}$$

Next, we determine the maximal total demand $\sum_{\tau_i \in \Gamma} W(\tau_i, t)$ by considering which tasks have carry-in work and which do not. In previous analysis of independent task system (Baruah, 2007), t_o is chosen to be the last idle instant before t_a, and therefore at most $m-1$ tasks can have carry-in work. Similarly, by our

assumption that t_o is truly idle, at most $m-1$ tasks can have carry-in work. Note that this choice of t_o limits carry-in work to $O(m)$, as compared to the best known global s-aware schedulability test (Liu and Anderson, 2013), which has $O(n)$ carry-in work.

Let $W_d(\tau_i) = W_c(\tau_i) - W_{nc}(\tau_i)$ be the amount of carry-in work τ_i could contribute. Then the total demand contributed by all higher-priority tasks τ_i is given by

$$\sum_{\tau_i \in \Gamma} W(\tau_i) = \sum_{\tau_i \in \Gamma} W_{nc}(\tau_i) + \sum_{\text{the } (m-1) \text{ largest}} W_d(\tau_i). \tag{5.7}$$

This concludes our review of prior demand-based analysis. Next, we consider the idleness induced within $[t_o, t_a) \cup \Theta$, which is the novel aspect of our analytical contributions.

5.2.4.2 Idleness

At any time instant $t \in [t_o, t_a) \cup \Theta$ in which a processor is not busy executing higher-priority work, idleness must be induced. Here we bound the amount of such idleness.

Traditionally, pi-blocking, which can cause idleness, is accounted for by computing the maximum duration of pi-blocking for each request, and incorporating that into a compatible schedulability test. Such approaches suffer from analysis pessimism in both the blocking analysis, as well as the associated schedulability test. In this work, we take a more holistic analysis approach, and incorporate the critical-section behavior into the schedulability analysis itself. We do so by effectively inverting the analysis logic—instead of analyzing the worst-case blocking a request may experience, we instead analyze the worst-case idleness that the same request can induce. To do so, we first consider the demand for shared resources within the analysis interval, using similar analysis techniques as presented for processor demand.

Lemma 5.13. *The maximum cumulative critical-section execution requirement by jobs of a sporadic task τ_i for resource ℓ_q that both arrive in and have deadlines within any interval of length t is given by the critical-section-bound function*

$$CSBF_q(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) L_{i,q}\right). \tag{5.8}$$

Proof. We consider only jobs that are released and have deadlines within an interval of length t. The total demand can be bounded by the scenario in which some job $J_{i,k}$ has a deadline at the end of the interval, and

jobs are released periodically. There can be at most $\lfloor \frac{t-d_i}{p_i} \rfloor$ jobs released before $J_{i,k}$ that also have deadlines within the interval. Each job can execute for at most $L_{i,q}$ time units within all of its critical sections for ℓ_q. The lemma follows. □

At the beginning of the analysis interval, carry-in jobs may also carry in critical sections that must be executed during the analysis interval. We must account for those critical sections as well, and again, we can apply similar reasoning as in the ordinary demand arguments. For the carry-in case, at most $L_{i,q}$ units of critical-section workload carry-in to the analysis interval, similar to (5.5).

Lemma 5.14.

$$CSBF'_q(\tau_i, t) = \left\lfloor \frac{t}{p_i} \right\rfloor L_{i,q} + \min(L_{i,q}, t \bmod p_i) \tag{5.9}$$

Using Lemmas 5.13 and 5.14, we can quantify demand for each resource within the analysis interval. Next, we bound idleness induced by such critical sections.

Lemma 5.15. *The maximum total amount of induced idleness with respect to τ_l within $[t_o, t_a) \cup \Theta$ by requests for resource ℓ_q is*

$$I_q(\tau_l) = (m-1) \left(\sum_{\tau_i \in \Gamma} CSBF_q(\tau_i, A_l + d_l) + \sum_{\substack{\text{the } (m-1) \\ \text{largest}}} \delta_i \right) \tag{5.10}$$

where $\delta_i = CSBF'_q(\tau_i, A_l + d_l) - CSBF_q(\tau_i, A_l + d_l)$

Proof. There can be at most $m-1$ carry-in jobs, each of which contributes at most δ_i to the demand for resource ℓ_q in $[t_o, t_a) \cup \Theta$. All tasks contribute at most $CSBF_q(\tau_i, A_l + d_l)$ non-carry-in demand for ℓ_q. For each time unit of critical-section demand within $[t_o, t_a) \cup \Theta$, at most $m-1$ processors can have idleness induced on account of blocking for ℓ_q. □

From the preceding discussions and lemmas, we have the following Theorem.

Theorem 5.2. *A task system Γ is G-EDF schedulable on m processors sharing n_r preemptive resources if for all tasks $\tau_l \in \Gamma$ and for all $A_l \geq 0$,*

$$\sum_{q=1}^{n_r} I_q(\tau_l) + \sum_{\tau_i \in \Gamma} W(\tau_i) \leq m(A_l + d_l - e_l). \tag{5.11}$$

5.2.4.3 Run-Time Complexity

Similar to previous tests, (5.11) can be evaluated in time polynomial in n for each combination of τ_l and A_l. The following theorem, demonstrates that only a pseudo-polynomial number of values of A_l need to be evaluated to determine if (5.11) holds for all A_l.

Theorem 5.3. *If $(U+(m-1)U^R) < m$ and (5.11) is violated for any A_l, then it is violated for some A_l satisfying the following condition:*

$$A_l \leq \frac{\xi}{m-(U+(m-1)U^R)} \tag{5.12}$$

where $\xi = \sum_{\tau_i \in \Gamma}(d_l - d_i)((m-1)u_i^R + U) + (m-1)(L_\Sigma + L_{all}) + e_\Sigma + e_{all} - m(d_l - e_l)$.

Proof. For notational convenience, let $T = A_l + d_l$. Observe that $W_{nc}(\tau_i) \leq DBF(\tau_i, T)$, and $W_d(\tau_i) \leq e_i$. Also, a carry-in job can carry in at most one job's worth of critical sections, so $CSBF'_q(\tau_i, A_l + d_l) - CSBF_q(\tau_i, A_l + d_l) \leq L_{i,q}$.

If Condition (5.11) is violated we have

$$\sum_{q=1}^{n_r} I_q + \sum_{\tau_i \in \Gamma} W(\tau_i) > m(T - e_l)$$

$$\Rightarrow \text{Substituting for } \sum_{q=1}^{n_r} I_q \text{ and } \sum_{\tau_i \in \Gamma} W(\tau_i) \text{ given the above observations.}$$

$$(m-1)(L_\Sigma + \sum_{\tau_i \in \Gamma}\sum_{q=1}^{n_r} CSBF_q(\tau_i, T)) + \sum_{\tau_i} DBF(\tau_i, T) + e_\Sigma > m(T - e_l)$$

$$\Rightarrow \text{By Equations Equation (5.3) and Equation (5.8)}$$

$$(m-1)(L_\Sigma + \sum_{\tau_i \in \Gamma}\sum_{q=1}^{n_r}(\lfloor \frac{T-d_i}{p_i} \rfloor + 1)L_{i,q} + \sum_{\tau_i \in \Gamma}(\lfloor \frac{T-d_i}{p_i} \rfloor + 1)e_i + e_\Sigma > m(T - e_l)$$

$$\Rightarrow \text{Removing the floors}$$

$$(m-1)(L_\Sigma + \sum_{\tau_i \in \Gamma}\sum_{q=1}^{n_r}((T-d_i)u_i^{\ell_q} + L_{i,q})) + \sum_{\tau_i \in \Gamma}((T-d_i)u_i^P + e_i) + e_\Sigma > m(T - e_l)$$

$$\Rightarrow \text{Rearranging}$$

$$(m-1)(L_\Sigma + L_{all}) + e_\Sigma + e_{all} + \sum_{\tau_i \in \Gamma}((T-d_i)u_i^P) + (m-1)\sum_{\tau_i \in \Gamma}\sum_{q=1}^{n_r}(T-d_i)u_i^{\ell_q} > m(T - e_l)$$

$$\Rightarrow \text{Substituting } T \text{ and Rearranging}$$

$$\sum_{\tau_i \in \Gamma}((d_l - d_i)((m-1)u_i^R + U) + (m-1)(L_\Sigma + L_{all})) + e_\Sigma + e_{all} - m(d_l - e_l) > A_l(m - (U + (m-1)U^R))$$

$$\Rightarrow \text{Dividing and substituting}$$

$$A_l < \frac{\xi}{m - (U + (m-1)U^R)}$$

provided $m - (U + (m-1)U^R) < m$. □

Discussion. Since idleness analysis is a radically different approach to accounting for the effects of blocking in schedulability, it is useful to compare and contrast it to previous methods to better understand its merits. Under traditional blocking analysis, either s-oblivious or s-aware, blocking is quantified *horizontally*, or over time with respect to the blocked task. For example, under the global OMLP (Brandenburg and Anderson, 2014) a request can be s-oblivious pi-blocked by $2m-1$ other requests. This blocking is incorporated into the task as additional time it must wait for the request to be satisfied. In contrast, in idleness analysis the effects of blocking are quantified *vertically*, or with respect to processors. A request may delay higher-priority work, but if it does not induce idleness, that blocking is analytically *irrelevant*—the processors are busy executing other higher-priority demand. If a critical section is executing, it is occupying one processor, and in the worst case idleness is induced on all of the remaining $m-1$ processors. By analyzing the impacts of critical sections vertically instead of horizontally, we have essentially reduced the analytical impact of each critical section from $2m-1$ (OMLP) or n (FMLP) to $m-1$ via idleness analysis and relaxed sharing constraints. (In practice, constant factors make this comparison more difficult, but this high-level comparison gives insight into the differences.)

Under s-aware schedulability analysis, blocking does not contribute any demand to the analysis interval. However, the main source of pessimism in prior G-EDF s-aware schedulability analysis (Liu and Anderson, 2013) is in bounding carry-in work. Any suspending (resource-requesting) task may contribute carry-in work, instead of only $m-1$ as is possible using either s-oblivious or idleness analysis. In fact, Brandenburg (Brandenburg, 2014) showed that in some cases treating s-aware blocking bounds as demand and using s-oblivious schedulability analysis yielded far greater schedulability than s-aware schedulability tests, likely due to the issue of carry-in work. Idleness analysis allows for the analysis interval to be extended to a point where by definition only $m-1$ jobs carry-in work, which can significantly reduce carry-in workload.

Under idleness analysis, the utilization loss due to a critical section is inflated by a factor of m. This is similar to a bus-arbitration policy such as TDMA that fairly shares bandwidth among cores. However, in practice many commercial off-the-shelf (COTS) memory controllers do not implement a fair arbitration

policy, and therefore modeling bus accesses as preemptive critical sections can be used to achieve similar theoretical results on a less predictable hardware platform. Furthermore, in practice, lower-priority jobs could execute during induced idleness thereby improving response times, or idled processor could lead to power savings.

5.3 Half-Protected Exclusion

In this section, we consider a weaker sharing constraint called *half-protected*, which is motivated by managing caches. During program execution, a job may access many cache blocks, only some of which are reused. Blocks that can be shown to be reused are called *useful cache blocks* ($UCBs$), and the remaining accessed blocks are called *evicting cache blocks* ($ECBs$) (Lee *et al.*, 1998). By managing cache accesses, we want to ensure that UCBs are *protected*, or not evicted before they are reused. However, regions of code in which evicting cache blocks may be accessed may be *unprotected*, and only prevented from interfering (co-scheduling or preempting) with a protected section. Importantly, unprotected sections can be interrupted at any time with no negative consequences, since they do not reuse cache blocks.

To illustrate this concept, consider the example schedule depicted in Figure 5.12, corresponding to Example 5.3 described in greater detail later. In that example, consider that task τ_2 executes a tight loop during its protected section during time $t \in [5,6)$, in which it reuses many cache blocks, and enjoys a significant performance benefit from the cache. On some platforms, the latency difference between the last-level cache and main memory is a factor of four or more. Also consider that the unprotected sections of τ_1 (during time $[2,3)$ and $[7,8)$) and τ_3 (during time $[4,5)$ and $[6,7)$) are regions of code in which data that is never reused is accessed. For example, if these tasks were writing results or executing a search algorithm, there may be little, if any, cache reuse. In such cases, we need not protect such data from being evicted as it will not be reused. For this reason, the preemption at time $t = 5$ by τ_2 is acceptable. It may evict the cached data of task τ_1 and τ_3, but because that data is not reused, such evictions are acceptable. However, if τ_1 and τ_2 were co-scheduled, τ_1 may evict the useful data of τ_2 with accesses to data that it will never reuse.

In the remainder of this section, we formalize the half-protected sharing constraint, and present algorithms for both uniprocessors and multiprocessors.

Problem description. There are two classes of requests for half-protected resources: *protected requests*, which require non-preemptive mutual exclusion, and unprotected requests, which have no sharing constraints.

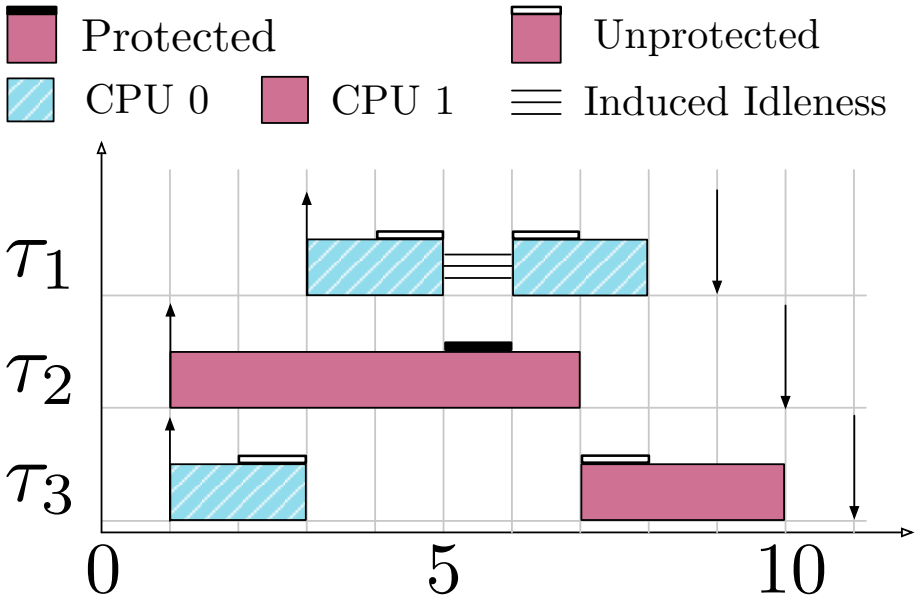


Figure 5.12: Example schedule depicting half-protected sharing.

Therefore, a protected request can interrupt an unprotected request at any time, but the converse is not true. On a multiprocessor, multiple unprotected sections can execute concurrently, but only one protected request can execute at a given time, and it must execute non-preemptively with respect to the resource.

To better understand the half-protected sharing constraint, we compare it to reader/writer sharing. Under reader/writer sharing, there are also two classes of requests: reads, which can execute concurrently or processor-preempt one another, and writes, which have the same sharing constraint as protected sections. However, both reads and writes are non-preemptive with respect to the resource. Therefore, a write request cannot be satisfied until all incomplete reads have completed their critical sections. In half-protected sharing, protected requests are identical to writes, while unprotected requests are a weaker version of reads. Unprotected requests are effectively preemptive read requests.

Example 5.3. Consider the example in Figure 5.12 in which three tasks are scheduled on two processors sharing a half-protected resource. At time $t = 1$, both τ_2 and τ_3 are released and begin executing. At time $t = 2$, τ_3 begins an unprotected section, and is processor-preempted by the release of τ_1 at time $t = 3$. At time $t = 4$, τ_1 also begins an unprotected section. Note that τ_3, while preempted is still in an unprotected section at $t = 4$. With protected sections or non-preemptive sharing, two critical sections cannot be satisfied concurrently. At time $t = 5$, τ_2 begins a protected section, which resource-preempts τ_1 and τ_3. This induces

idleness during the interval $[5,6)$. At time $t = 6$, τ_1 resumes its unprotected section. The remainder of the example follows G-EDF scheduling. $\Diamond$

Scheduling upon uniprocessors. Half-protected resources can be supported with extensions to the Priority Ceiling Protocol (PCP) (Rajkumar, 1991) or the Stack Resource Policy (SRP) (Baker, 1991). Interestingly, unlike preemptive resources discussed previously, half-protected resources pose a new synchronization problem on uniprocessor platforms.

In a priority-ceiling-based synchronization protocol, the *current priority ceiling* $\hat{\Pi}(t)$ at time t is equal to the highest ceiling associated with any active critical section. For the simple case of non-preemptive mutual exclusion, the ceiling of each resource is defined to be the highest-priority task that accesses that resource. If a task does not have a priority higher than the current priority ceiling $\hat{\Pi}(t)$, then it is forced to block unless it is executing the critical section responsible for setting the current priority ceiling. Priority inheritance is used to ensure resource-holder progress if a higher-priority task is blocked waiting for a lower-priority one.

To realize any benefits of a more relaxed sharing constraint in a priority-ceiling-based protocol, the rules for setting the current priority ceiling need to be relaxed to lower the ceiling and reduce blocking. For example, for reader/writer sharing, the ceiling of a reader/writer resource ℓ_r during a read critical section is set to the highest priority task that may read ℓ_r, while during a write critical section, the ceiling of ℓ_r is set to the highest priority task that may read or write ℓ_r (Rajkumar, 1991). By reducing the ceiling for read requests, some higher-priority read requests can processor-preempt tasks executing read critical sections, instead of blocking.

The resource ceiling Π_h of a half-protected resource ℓ_h depends upon whether it is currently in a protected or unprotected section, similar to reader/writer sharing. The *protected ceiling* or the ceiling during a protected section, is set to the priority of the highest-priority task that accesses ℓ_h. This ceiling value ensures that a protected section runs non-preemptively with respect to ℓ_h. The *unprotected ceiling* or the ceiling during an unprotected section, is set to the lowest priority in the system. This low ceiling ensures that any task can processor- or resource-preempt a job in an unprotected section, therefore preventing any task from blocking on an unprotected section. However, an unprotected section may still block on a protected section. Notably, similar to other priority-ceiling-based protocols, protected and unprotected sections can be nested using this approach.

When these resource-ceiling definitions are applied in the context of the SRP, there is an additional benefit: system calls are not necessary to enter or exit unprotected sections, thereby reducing overheads. In the SRP, all blocking is incurred before a job begins, and therefore all resources can be accessed immediately without blocking. System calls are required for protected sections, as the current priority ceiling may be increased to prevent higher-priority jobs from beginning execution when resources they may need are currently locked. However, unprotected sections will never increase the current priority ceiling, and therefore do not need to issue system calls. In fact, in the SRP no run-time knowledge of unprotected sections is necessary. Such knowledge is only necessary to set the protected-ceiling values offline. In contrast, in the PCP, entering and exiting unprotected sections must be handled at runtime to ensure that unprotected sections block on protected ones.

Scheduling upon multiprocessors. On a multiprocessor platform, half-protected resources pose new issues. Protected sections must execute non-preemptively with respect to the half-protected resource, as well as non-concurrently with any other protected or unprotected sections. In contrast, unprotected sections can execute concurrently, and be arbitrarily resource preempted. In fact, one protected request can resource-preempt multiple unprotected sections.

We consider the following half-protected synchronization policy. Protected requests are statically prioritized over unprotected ones, *i.e.*, a protected request will always preempt an unprotected request. Unprotected requests execute whenever they are not preempted. Protected requests are assumed to be prioritized against one another by non-preemptive EDF. Additionally, we assume that priority inheritance is used to ensure the progress of protected requests, *i.e.*, a protected request inherits the priority of the highest-priority protected or unprotected request that it blocks. Given this synchronization policy, we next extend our idleness analysis and derive an associated schedulability test.

Unprotected sections. Idleness analysis, as compared to blocking analysis, shifts the analysis burden from blocked requests to satisfied requests. This analysis technique is particularly well suited to unprotected sections.

Lemma 5.16. *Unprotected sections do not induce idleness.*

Proof. A satisfied request induces idleness if it blocks another request leaving a processor idle. Unprotected requests can be satisfied concurrently, and therefore do not block one another. By assumption, unprotected

requests are statically prioritized lower than protected requests, and do not block protected requests. Therefore, unprotected requests do not block other requests and thus do not induce idleness. □

Given this result, we can safely ignore unprotected requests in idleness analysis. Therefore, to avoid notational clutter, for the rest of this section, we assume that $L_{i,q}$ is the total length of τ_i's ℓ_q protected sections. Notably, these results also apply to non-preemptive resources also.

Carry-up sections. In idleness analysis for preemptive resources (Section 5.2.4.2), all critical sections and jobs with deadlines later than the analyzed job $J_{l,j}$ were ignored as they could not delay the execution of $J_{l,j}$. When considering non-preemptive resources, this is no longer true. A lower-priority job than $J_{l,j}$ may issue a request while $J_{l,j}$ is blocked or scheduled on another core that may subsequently non-preemptively block other higher-priority requests and induce idleness. We define a *carry-up request* to be a request that can execute in the analysis interval but that has a deadline after the analysis interval. In comparison to carry-in critical sections, which are issued by higher-priority jobs that were released before the analysis interval, a carry-up critical section can be released at any point (even before t_o) but has a lower priority than the analyzed job $J_{l,j}$. This behavior is depicted in Example 5.3 and Figure 5.12. Observe that when analyzing τ_1, τ_2 has a deadline after d_1. However, τ_2 still induces idleness during time $[5,6)$. In this case, the protected section of τ_2 is a carry-up section.

Carry-up critical sections must be accounted for in idleness analysis for protected sections to account for the additional sources of induced idleness. Importantly, such analysis requires more carefully considering jobs with deadlines after the end of the analysis interval, t_d. We therefore clarify several previous assumptions and definitions in this context.

Recall from Definition 5.7 that t_o is defined to be the last truly idle time instant before t_a, the release of $J_{l,j}$. However, we had previously discarded all jobs with deadlines after t_d. We use the same definition here, *i.e.*, t_o is the last truly idle time instant before t_a with respect to to jobs that have deadlines at or before t_d. Thus, at time t_o, there are at most $m-1$ pending jobs with deadlines at or before t_d, which are the *carry-in jobs*. Note that there could be additional jobs released before t_o but with deadlines after t_d that can contribute carry-up critical sections. Such critical sections are considered carry-up critical sections, rather than carry-in.

Lemma 5.17. *Let $[t_0,t_1)$ be an interval of length t, and τ_i be a task without a carry-in job. The maximum cumulative execution requirement of protected sections of a half-protected resource ℓ_q by jobs of a sporadic*

task τ_i that are pending at any time during $[t_0, t_1)$ is given by the protected-section bound function

$$PSBF_q(\tau_i, t) = \left\lceil \frac{t}{p_i} \right\rceil L_{i,q}. \tag{5.13}$$

Proof. There are two classes of tasks without carry-in work, those with all job releases after t_0 (Case 1), and those released before t_0, and with deadlines after t_1 (Case 2). Those with releases before t_0 and deadlines before t_1, by definition have a carry-in job.

Case 1. The maximum execution time of protected sections generated by jobs of τ_i released after t_0 is bounded by the case in which the first job of τ_i is released at t_0, and jobs are released periodically thereafter. There are at most $\left\lceil \frac{t}{p_i} \right\rceil$ such jobs that could be released before t_1, each of which executes protected sections for at most $L_{i,q}$ time.

Case 2. If a job of τ_i is released strictly before t_0, and has a deadline strictly after t_1, then $t < d_i \leq p_i$. Thus, there can be only one job of τ_i that is pending during $[t_0, t_1)$, and it executes protected sections for at most $L_{i,q} = \left\lceil \frac{t}{p_i} \right\rceil L_{i,q}$. □

Note that $PSBF_q(\tau_i, t)$ has many commonalities with the well-studied *request-bound function*, which bounds the execution requirement of jobs that can be released within an interval of length t. Similarly, we consider the protected sections of all jobs that can be released in an interval of length t, as even jobs with priority lower than $\tau_{l,j}$ can contribute carry-up sections, which can induce idleness.

Lemma 5.18. *Let $[t_0, t_1)$ be an interval of length t, and τ_i be a task with a carry-in job. The maximum cumulative execution requirement of protected sections of a half-protected resource ℓ_q by jobs of a sporadic task τ_i that are pending at any time during $[t_0, t_1)$ is given by the* protected-section bound function

$$PSBF'_q(\tau_i, t) = \left\lceil \frac{t + d_i}{p_i} \right\rceil L_{i,q}. \tag{5.14}$$

Proof. By Lemma 5.1 of (Brandenburg, 2011), there are at most $\lceil \frac{t+R_i}{p_i} \rceil$ distinct jobs of a task τ_i that can execute in any interval of length t, where $R_i \leq d_i$ is the response time of τ_i. Thus, $\left\lceil \frac{t+d_i}{p_i} \right\rceil$ total jobs may execute within $[t_0, t_1)$, and each may execute protected sections for at most $L_{i,q}$. □

Any non-protected sections (*i.e.*, unprotected or non-critical) of jobs with deadlines after t_d that execute within the analysis interval execute either concurrently with $J_{l,j}$ or during blocking of higher-priority jobs.

Therefore, such demand does not delay $J_{l,j}$ and need not be analyzed further. We next bound the total induced idleness, similarly to Lemma 5.15, using the results from the previous two lemmas.

Lemma 5.19. *The maximum total amount of induced idleness with respect to τ_l within $[t_o, t_a) \cup \Theta$ by protected requests for a half-protected resource ℓ_q is given by*

$$I_q^h(\tau_l) = (m-1)\left(\sum_{\tau_i \in \Gamma} PSBF_q(\tau_i, A_l + d_l) + \sum_{\substack{\text{the } (m-1) \\ \text{largest}}} \delta_i\right) \tag{5.15}$$

where $\delta_i = PSBF'_q(\tau_i, A_l + d_l) - PSBF_q(\tau_i, A_l + d_l)$.

Carry-up sections contribute demand to the analysis interval, as well as induce idleness. If a carry-up section is executed during the analysis interval, priority inheritance ensures that it is scheduled if it would induce idleness. However, because carry-up sections are contributed by lower-priority jobs, that demand is not accounted for by the workload of higher-priority jobs.

Lemma 5.20. *The maximum cumulative protected-section execution time of all carry-up requests with respect to τ_l for a half-protected resource ℓ_q that run in $[t_o, t_a) \cup \Theta$ is given by*

$$C_q(\tau_l) = \sum_{\tau_i \in \Gamma} (PSBF_q(\tau_i, t) - CSBF_q(\tau_i, t)) + \sum_{\text{the } (m-1) \text{ largest}} \alpha_i - \sum_{\text{the } (m-1) \text{ largest}} \beta_i \tag{5.16}$$

where $t = A_l + d_l$, $\alpha_i = PSBF'_q(\tau_i, t) - PSBF_q(\tau_i, t)$, and $\beta_i = CSBF'_q(\tau_i, t) - CSBF_q(\tau_i, t)$.

Proof. The total length of all protected requests, including carry-up requests, that execute within $[t_o, t_a) \cup \Theta$ is given by $\sum_{\tau_i \in \Gamma} PSBF_q(\tau_i, t) + \sum_{\text{the } (m-1) \text{ largest}} \alpha_i$. However, some of that demand is not carry-up demand, but instead higher-priority demand. This higher-priority demand is quantified by Lemmas 5.13, and 5.14, and totals $\sum_{\tau_i \in \Gamma} CSBF_q(\tau_i, t) + \sum_{\text{the } (m-1) \text{ largest}} \beta_i$. By subtracting the later term from the former term, we have the total amount of carry-up demand that executes during $[t_o, t_a) \cup \Theta$. □

From these results and discussion, we have the following schedulability test.

Theorem 5.4. *A task system Γ is global EDF schedulable on m processors sharing n_r half-protected resources if for all tasks $\tau_l \in \Gamma$ and for all $A_l \geq 0$,*

$$\sum_{q=1}^{n_r} (C_q(\tau_l) + I_q^h(\tau_l)) + \sum_{\tau_i \in \Gamma} W(\tau_i) \leq m(A_l + d_l - e_l). \tag{5.17}$$

Again, we can bound the maximum number of testing points required to check schedulability.

Theorem 5.5. *If $(U + (m-1)U^R) < m$ and (5.17) is violated for any A_l, then it is violated for some A_l satisfying the following condition:*

$$A_l < \frac{\phi}{m - (U + (m-1)U^R)} \tag{5.18}$$

where $\phi = m(L_{all} + L_\Sigma) + (m-1)d_l U^R + e_\Sigma - m(d_l - e_l) + \sum_{\tau_i \in \Gamma}(d_l - d_i)u_i^P + e_{all}$.

Proof. For notational convenience, let $T = A_l + d_l$. Observe that $W_{nc}(\tau_i) \leq DBF(\tau_i, T)$, and $W_d(\tau_i) \leq e_i$. Also, a carry-in job can carry in at most one job's worth of critical sections, so $CSBF'_q(\tau_i, A_l + d_l) - CSBF_q(\tau_i, A_l + d_l) \leq L_{i,q}$, and $PSBF'_q(\tau_i, A_l + d_l) - PSBF_q(\tau_i, A_l + d_l) \leq L_{i,q}$.

A task can only contribute carry-up critical sections from one job, thus $PSBF_q(\tau_i, T) - CSBF_q(\tau_i, T) \leq L_{i,q}$, and $PSBF'_q(\tau_i, T) - CSBF'_q(\tau_i, T) \leq L_{i,q}$. Therefore, $\sum_{q=1}^{n_r} C_q \leq L_{all} + L_\Sigma$. If Condition (5.17) is unsatisfied, we have

$$\sum_{q=1}^{n_r} (C_q + I_q^h) + \sum_{\tau_i \in \Gamma} W(\tau_i) > m(T - e_l)$$

$$\Rightarrow \text{Substituting for } \sum_{q=1}^{n_r} (C_q + I_q^h) \text{ and } \sum_{\tau_i \in \Gamma} W(\tau_i)) \text{ as described above}$$

$$L_{all} + L_\Sigma + (m-1)\sum_{q=1}^{n_r} \sum_{\tau_i \in \Gamma} \left(\left\lceil \frac{T}{p_i} \right\rceil L_{i,q} + L_\Sigma \right) + \sum_{\tau_i \in \Gamma} \left(\left\lfloor \frac{T - d_i}{p_i} \right\rfloor + 1 \right) e_i + e_\Sigma > m(T - e_l)$$

$$\Rightarrow \text{Removing floors and ceilings}$$

$$L_{all} + L_\Sigma + (m-1)\sum_{q=1}^{n_r} \sum_{\tau_i \in \Gamma} \left(\left(\frac{T}{p_i} + 1 \right) L_{i,q} + L_\Sigma \right) + \sum_{\tau_i \in \Gamma} \left(\frac{T - d_i}{p_i} + 1 \right) e_i + e_\Sigma > m(T - e_l)$$

$$\Rightarrow \text{Rearranging and substituting}$$

$$m(L_{all} + L_\Sigma) + (m-1)TU^R + e_\Sigma + e_{all} + \sum_{\tau_i \in \Gamma} (T - d_i)u_i^P > m(T - e_l)$$

$$\Rightarrow \text{Rearranging and substituting } T$$

$$m(L_{all}+L_{\Sigma})+(m-1)d_l U^R+e_{\Sigma}-m(d_l-e_l)+\sum_{\tau_i\in\Gamma}(d_l-d_i)u_i^P+e_{all}>A_l(m-(U+(m-1)U^R))$$

$$\Rightarrow \text{Rearranging and substituting}$$

$$A_l<\frac{\phi}{m-(U+(m-1)U^R)},$$

provided $m-(U+(m-1)U^R)<m$. □

Discussion. Perhaps surprisingly, the manner in which protected requests are prioritized against one another does not affect schedulability under our idleness-analysis framework, so long as a progress mechanism such as priority inheritance ensures that a carry-up request that induces idleness at time t is scheduled at time t. This is due to how idleness analysis accounts for the effects of blocking vertically rather than horizontally. Even a random prioritization among protected requests could not induce idleness more than $m-1$ processors during the execution of a critical section, though the worst-case blocking bound would be quite pessimistic. Thus, the above idleness analysis can be applied to existing non-preemptive locking protocols such as the global OMLP (Brandenburg and Anderson, 2014), or the FMLP (Block et al., 2007) by treating all lock requests as protected requests. Furthermore, nested locking protocols could also be supported at the expense of more verbose notation, provided the protocol supported transitive priority inheritance and prevented deadlock.

5.4 Evaluation

We next present evaluations of our proposed algorithms and analysis with respect to hard real-time schedulability. We considered two alternative schemes for randomly generating task systems, one motivated by a preemptive bus or interconnect, and another similar to previous studies on real-time locking protocols (Brandenburg, 2014). These two generation schemes provide insights into the schedulability gains afforded by relaxed sharing constraints and idleness analysis.

Bus synchronization. We first consider a random task-system generation process that varies both the total processor utilization and the total resource utilization of one preemptive resource. We considered schedulability on $m\in\{2,4,8\}$ processor systems. We generated task systems for each processor utilization in $\{0.1,0.2,\ldots,m\}$. The total resource utilization of the generated tasks was selected from $\{0.1,0.2,0.3,0.4\}$. The per-task processor utilizations were chosen from one of two exponential distributions with mean 0.1 (*light*) and 0.25 (*medium*), and restricted to $[0,1]$. The period of each task was uniformly chosen from

$[10\,\text{ms}, 100\,\text{ms}]$. The resource utilization of each task was uniformly random, and normalized across all tasks to sum to the specific total resource utilization. We assume one critical section per job. For each combination of parameters, task systems were randomly generated until the schedulability ratio could be estimated with 95% confidence to within 0.05.

For each generated task system, we considered schedulability under the global OMLP (Brandenburg and Anderson, 2014); the FMLP$^+$ (Brandenburg, 2014) assuming either Liu and Anderson's s-aware analysis (Liu and Anderson, 2013) (FMLP-A), or by treating s-aware blocking bounds as s-oblivious ones and applying Baruah's (Baruah, 2007) s-oblivious test (FMLP-O); preemptive sharing with idleness analysis (P-I); and non-preemptive sharing with idleness analysis (NP-I). Note that NP-I follows from treating all lock requests as protected requests. As a basis for comparison, we also plot schedulability without any synchronization (NOLOCK), which upper bounds all other considered cases. An example schedulability graph from this study is shown in Figure 5.13(a).[7]

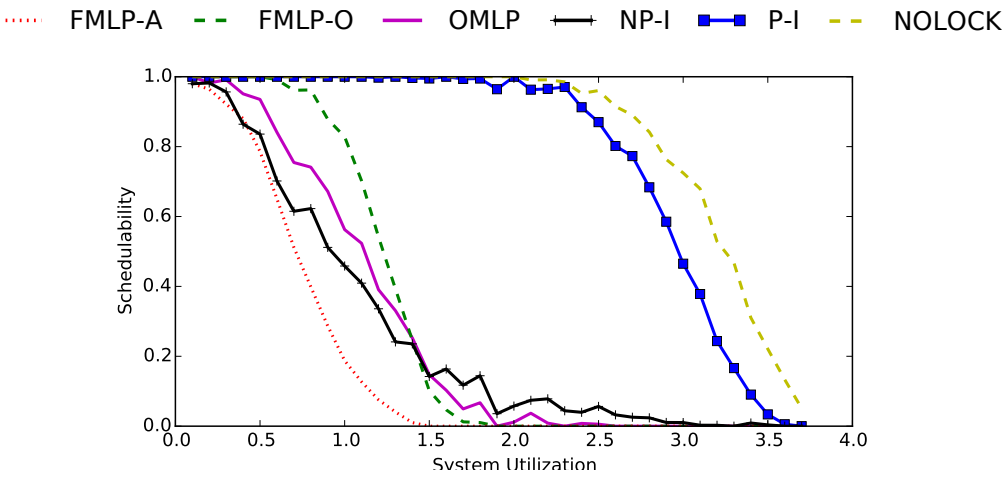
Observation 5.6. Preemptive sharing and idleness analysis can greatly improve schedulability over non-preemptive sharing.

This observation is supported by Figure 5.13(a). In particular, with preemptive sharing most task systems with utilizations up to approximately 2.5 are schedulable, while most task systems with utilization greater than 1.0 are unschedulable by all considered non-preemptive approaches. This demonstrates how more relaxed sharing constraints can be exploited to improve schedulability.

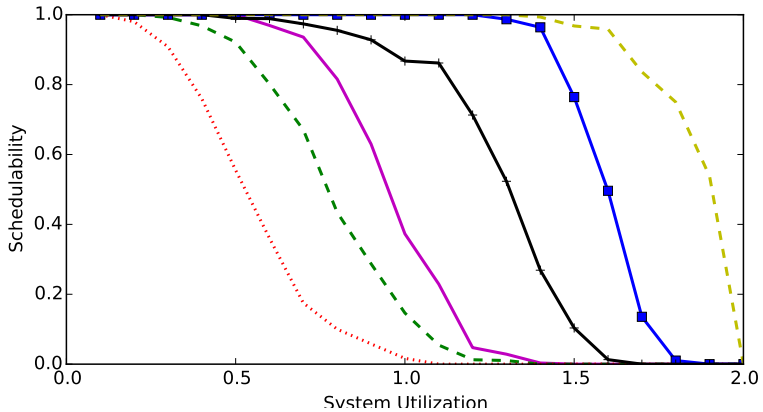
Locking study. As we discussed previously, idleness analysis can be used in place of blocking analysis to analyze schedulability for many existing real-time locking protocols, including the OMLP and the FMLP. We therefore considered an alternative task-system generation scheme similar to previous studies to evaluate the difference between idleness analysis and blocking analysis. In this study, tasks were generated as described above with the following exceptions. Total resource utilization was not directly controlled, instead, critical-section lengths were chosen uniformly among $[1\,\mu\text{s}, 15\,\mu\text{s}]$ (*short*), $[1\,\mu\text{s}, 100\,\mu\text{s}]$ (*moderate*), or $[5\,\mu\text{s}, 1280\,\mu\text{s}]$ (*long*). Each task had a uniform probability $p^{acc} \in \{0.1, 1.0\}$ of accessing one shared resource. Example schedulability graphs from this study can be seen in insets (b) and (c) of Figure 5.13.[8]

[7] All other generated graphs are available online `http://www.cs.unc.edu/~bcw/diss/`.

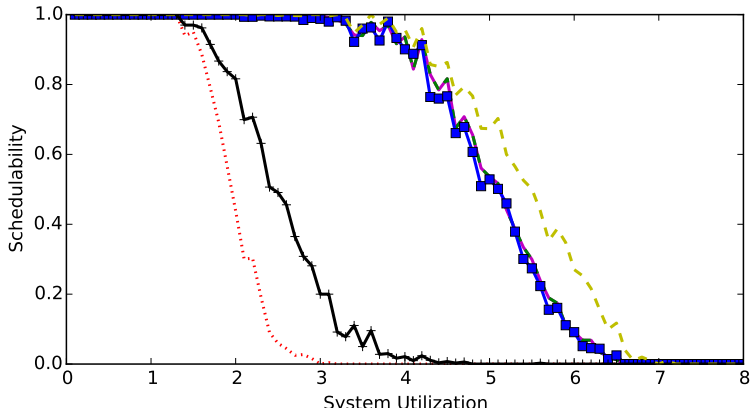
[8] All other generated graphs are available online `http://www.cs.unc.edu/~bcw/diss/`.



(a) $m = 4$, $U^R = 0.1$.



(b) $m = 2$, $p^{acc} = 1.0$.



(c) $m = 8$, $p^{acc} = 0.1$.

Figure 5.13: Three example schedulability graphs. (a) is drawn from the bus synchronization experiment, and (b) and (c) are from the locking experiment. Task utilizations are light in all cases. Long critical sections are assumed in (b) and (c).

Observation 5.7. Idleness analysis can improve schedulability over both s-aware and s-oblivious blocking analysis for existing real-time locking protocols.

This observation is supported by Figure 5.13(b), in which the curve NP-I has greater schedulability than all of the existing locking protocols. As discussed previously, idleness analysis effectively reduces the effect of blocking to $m-1$ from $2m-1$ or $O(n)$ for the OMLP and FMLP+, respectively. In comparison to s-aware schedulability analysis, by explicitly analyzing induced idleness, we are able to bound carry-in work to $m-1$ tasks instead of $O(n)$. Consequently, idleness analysis can provide improved schedulability. This is significant as the blocking bounds for the FMLP+ and the OMLP have been proven asymptotically optimal under s-aware and s-oblivious analysis assumptions, while idleness analysis applies to a very general class of mutex locking protocols, including random request prioritizations, which have very large worst-case blocking bounds.

Observation 5.8. Idleness analysis is incomparable with both s-aware and s-oblivious blocking analysis.

This observation is supported by Figure 5.13(c), in which $p^{acc} = 0.1$. In task systems in which the number of resource-using tasks is small, or the number of processors is large, idleness bounds are most pessimistic. In these cases, s-aware and s-oblivious blocking analysis can yield better schedulability. Idleness analysis is therefore incomparable to blocking analysis.

Observations 5.7 and 5.8 demonstrate that there may be interesting tradeoffs to explore at the intersection of idleness analysis and blocking analysis. For example, perhaps protocol-specific knowledge could be used to tighten idleness bounds.

5.5 Chapter Summary

In this chapter, we have presented three algorithms and analyses motivated by the needs of shared-hardware-management applications for GPUs, caches, and shared buses. In so doing, we have developed a new progress mechanism in RRPD, which was used to construct the $\mathsf{R}^2\mathsf{DGLP}$, an asymptotically optimal independence-preserving k-exclusion locking protocol for global scheduling. The $\mathsf{R}^2\mathsf{DGLP}$ improves upon existing k-exclusion locking protocols such as the O-KGLP and the CK-OMLP, as we have demonstrated. These improvements are particularly significant for applications in which critical sections are long, as is the case when using a locking protocol to arbitrate access to shared I/O devices such as GPUs. Notably, the

R^2DGLP has been applied in GPU-management applications (Elliott, 2015), as well as in the RNLP as a token lock, as described in Chapter 3.

We also formalized two new weaker sharing constraints, preemptive mutual exclusion, and half-protected exclusion, that enable reduced blocking in many cases. These two sharing constraints are motivated by the desire to more predictably shared bus traffic, and caches, respectively. We have also presented simple algorithms to realize these sharing constraints. To analyze the effects of blocking on schedulability of these algorithms, as well as other synchronization algorithms, we have developed a novel analysis technique called idleness analysis. Idleness analysis is used in place of blocking analysis in demand-based multiprocessor schedulability tests. Instead of analyzing the delays due to blocking, idleness analysis quantifies the idleness induced in the analysis interval, which can significantly reduce analysis pessimism. Experimental results show up to 250% increase in schedulability as a result of our new analysis techniques and relaxed sharing constraints.

CHAPTER 6: CONCLUSION

The main objective of the research presented in this dissertation was to develop synchronization algorithms for multiprocessor real-time systems that enable increased platform utilization. This objective was accomplished by the development of several new multiprocessor real-time locking protocols, each motivated by different use cases and resource-access patterns. By leveraging the semantics of the resource-access patterns, these protocols in many cases offer improved platform utilization over previous real-time locking protocols. In the remainder of this dissertation, we summarize the results presented herein (Section 6.1), briefly discuss other contributions to the field of real-time computing not included in this dissertation (Section 6.2), and describe relevant open questions and avenues for future work (Section 6.3).

6.1 Summary of Results

In the following, we review the key results of the work presented in this dissertation.

Fine-grained locking in multiprocessor real-time systems. In Chapter 3, we presented the RNLP, the first multiprocessor real-time locking protocol to support fine-grained locking. The problem of nested locking in multiprocessor real-time systems stood open for over twenty years. Prior to the RNLP, nested locking of resources shared among multiple processors in real-time systems was only supported through coarse-grained locking, in which resources are grouped and treated as a single lockable entity. Fine-grained locking can be realized through nested locking, or a new fine-grained locking technique called dynamic group locking. In the latter case, a request atomically requests a set of resources, which may be a subset of a larger group of resources that would be treated as a single lockable entity under coarse-grained locking.

The RNLP has a modular architecture, composed of a k-exclusion token lock and an RSM. In Chapter 3, several RSMs that support mutex resources were presented, and in Chapter 4, RSMs that support reader/writer sharing and k-exclusion were presented. We showed that there exist pairings of token locks and RSMs that are optimal for all platform configurations in which optimal coarse-grained locking protocols are known. Interestingly, at the time the RNLP was originally presented (Ward and Anderson, 2012), there was no known

optimal mutex locking protocol for clustered systems under s-aware analysis. Since then, RSB has been proposed (Brandenburg, 2014), we incorporated RSB into a new RSM presented herein, that is optimal in that case as well. This demonstrates the utility of the modular architecture of the RNLP.

In this dissertation, we presented the first fine-grained blocking analysis of the RNLP. (In previous publications (Ward and Anderson, 2012, 2014b), fine-grained blocking analysis was omitted due to space constraints.) This blocking analysis is tighter than previous pi-blocking bounds for the RNLP, at the expense of increased computational complexity. However, these tighter bounds results in improved platform utilization.

Independence-preserving *k*-exclusion locking protocol. In Chapter 5, we presented the R^{2}DGLP, an asymptotically optimal, independence-preserving *k*-exclusion locking protocol for globally scheduled systems. The R^{2}DGLP is especially useful for managing access to multiple GPUs, and indeed has been applied in such applications (Elliott et al., 2013; Elliott, 2015). In designing the R^{2}DGLP, we also developed a new progress mechanism, RRPD, which is similar to priority donation (Brandenburg and Anderson, 2011), but is applied at the time of request issuance instead of job release. By shifting the time of donation to request issuance, it is possible to construct an independence preserving locking protocol for globally scheduled systems. RRPD is similar to a priority-donation technique presented by Elliott and Anderson (2013) for the O-KGLP, but enables improved blocking bounds.

While the design of the R^{2}DGLP was motivated by the need to manage access to GPUs, it is also useful as a token lock in the RNLP. When combined with the I-RSM on globally scheduled systems, the resulting RNLP variant is also independence preserving, and has a request-blocking bound no worse than the global OMLP (Brandenburg and Anderson, 2010a).

Synchronization algorithms for shared hardware resources. In Chapter 5, we also defined two new resource-sharing constraints motivated by the need to more predictably manage shared hardware resources such as caches and buses. The goal of such shared-hardware management is to reduce or eliminate the effects of timing interference caused by concurrently executing tasks, thereby improving timing predictability. In turn, the improved predictability may offset the cost of shared-hardware management, leading to improved schedulability.

Towards this goal, we presented *preemptive mutual exclusion*, which is motivated by the need to manage access to a shared communication bus such as the memory bus, and *half-protected exclusion*, which

is motivated by the need to manage access to cache resources. We considered simple, straightforward algorithms that realize these sharing constraints, and provided analysis and experimental results showing the schedulability improvements they may enable.

Idleness analysis. Locking protocols cause blocking as tasks are forced to wait until resources are available, or to ensure resource-holder progress. This blocking must be somehow incorporated into schedulability analysis in real-time systems so as to ensure that jobs do not miss deadlines on account of blocking. Traditionally, blocking is analyzed via *blocking analysis*, the results of which are incorporated in schedulability analysis. In Chapter 5, we presented a new technique, called idleness analysis, for incorporating the effects of blocking into schedulability analysis, that does not require any blocking analysis. In idleness analysis, idleness induced by blocking is analyzed, instead of the duration of blocking. This "flips the analysis" from asking the question "how long can this request be blocked?" to instead asking "how much idleness can this request cause?"

Idleness analysis and blocking analysis are theoretically incomparable with respect to schedulability, *i.e.*, neither dominates the other. In Section 5.4, we presented the results of schedulability studies that were conducted to investigate in which cases one is favorable to the other. Idleness analysis is often favorable in systems with smaller core counts, as fewer processors can be idled as a result of synchronization.

6.2 Other Related Work

During the course of my graduate education, I have contributed a number of results outside of the scope of this document. This section briefly summarizes these results, which appeared in ten other peer-reviewed papers. These contributions fall into three broad categories: **(i)** the analysis and management of shared hardware components, **(ii)** multiprocessor synchronization, and **(iii)** soft-real-time schedulability and tardiness analysis.

6.2.1 Shared-Hardware Management and Analysis[1]

This dissertation has described several synchronization algorithms that are motivated by the application of managing shared hardware devices. In work outside of the scope of this dissertation, I have co-authored several other publications in which shared-hardware-management techniques have been applied in practice. Many of these publications even apply algorithms described herein.

GPUs. In collaboration with Elliott and Anderson (Elliott et al., 2013), I helped design and analyze GPUsync, which is a framework for managing GPUs in multi-GPU multiprocessor real-time systems. My contributions to this work were predominantly targeted at the design and analysis of GPUsync, which leverages the R^2DGLP described in Chapter 5, and Glenn Elliott implemented, debugged, and evaluated the entire system.

GPUsync takes a synchronization-oriented approach to GPU management and applies locking protocols to GPUs, as well as hardware components therein. GPUsync was designed with flexibility, predictability, and parallelism in mind. Specifically, it can be applied under either static- or dynamic-priority CPU scheduling; can allocate CPUs/GPUs on a partitioned, clustered, or global basis; provides flexible mechanisms for allocating GPUs to tasks; enables task state to be migrated among different GPUs, with the potential of breaking such state into smaller "chunks"; provides migration cost predictors that determine when migrations can be effective; enables a single GPU's different engines to be accessed in parallel; properly supports GPU-related interrupt and worker threads according to the sporadic task model, even when GPU drivers are closed-source; and provides budget policing to the extent possible, given that GPU access is non-preemptive. No prior real-time GPU management framework provides a comparable range of features.

[1]This subsection summarizes results from the following papers:
Elliott, G., Ward, B., and Anderson, J. (2013). GPUSync: A framework for real-time GPU management. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, pages 33–44.
Ward, B., Herman, J., Kenna, C., and Anderson, J. (2013b). Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 157–167.
Ward, B., Thekkilakattil, A., and Anderson, J. (2014). Optimizing preemption-overhead accounting in multiprocessor real-time systems. In *Proceedings of the 22nd Conference on Real-Time Networks and Systems*, pages 235–243
Chisholm, M., Ward, B., Kim, N., and Anderson, J. (2015). Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Proceedings of the 36th Real-Time Systems Symposium*, pages 306–316.
Kim, N., Ward, B., Chisholm, M., Fu, C.-Y., Anderson, J., and Smith, F. (2016). Attacking the one-out-of-*m* multicore problem by combining hardware management with mixed-criticality provisioning. In *Proceedings of the 22nd Real-Time Embedded Technology and Applications Symposium*, pages 149–160.
Chisholm, M., Kim, N., Ward, B., Otterness, N., Anderson, J., and Smith, F. (2016). Reconciling the tension between hardware isolation and data sharing in mixed-critical, multicore systems. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (to appear)*.

Multiprocessor cache-related preemption delay analysis. Preemptions (and migrations, in which a task is scheduled on one processor and moved to another processor), can increase the execution time of tasks due to the interference caused with respect to the cache. When a task is preempted, another task executes instead and may evict cache lines that the preempted task otherwise would have reused. When the preempted task resumes, it must reload previously cached data from memory that had it not been preempted, would have been cached and therefore accessed more quickly. There are a number of techniques known for quantifying this effect, known as *cache-related preemption delay* (*CRPD*) or *cache-related preemption and migration delay* (*CPMD*) analysis. Broadly, these two classes of techniques can be classified as either *preemption-centric*, in which the additional delays are analytically accounted for by inflating the execution time of the preempting task, or *task-centric*, in which the additional delays are analytically accounted for by inflating the execution time of the preempted task. We presented a new CRPD analysis technique that combined these two approaches to CRPD analysis into a linear program, which can be solved to minimize the utilization loss associated with CRPDs (Ward et al., 2014).

Dynamic shared-cache management In collaboration with Herman, Kenna, and Anderson (Ward et al., 2013b), I helped design shared-cache management techniques designed to improve predictability with respect to shared-cache accesses, thereby improving analytical platform utilization. In that work, we developed and analyzed several techniques for minimizing or eliminating interference among concurrently executing tasks through the shared cache. These two techniques are called *cache locking*, and *cache scheduling*. Cache locking applies a variant of the *k*-exclusion RNLP described in Chapter 4 to individual ways of shared-cache colors. Cache scheduling applies a scheduling algorithm, such as EDF, to the ways of cache colors. These techniques are also applied in a mixed-criticality scheduling framework.

Mixed-criticality shared-cache and memory-bank management. In a series of papers (Chisholm et al., 2015; Kim et al., 2016; Chisholm et al., 2016), we developed support in the *mixed-criticality on multicore* (MC^2) scheduling framework for shared-cache and DRAM-bank partitioning. Extensive experimental evaluations were conducted to quantify the effects of isolation vs. sharing with respect to both the shared cache as well as the DRAM banks. Furthermore, these evaluations were conducted to reflect the analysis assumptions used at different criticality levels—higher-criticality tasks are provisioned more pessimistically, while lower-criticality tasks may be provisioned based on observed average-case performance. Based on the results of these experimental evaluations, a linear-programming-based optimization framework was

developed to allow shared-cache and DRAM-bank partitions to be optimized to improve the overall platform utilization within the context of the mixed-criticality analysis framework. This optimization balances the cache and memory-bank allocation among the criticality levels, based on tradeoffs observed in experimental evaluations of the execution time of tasks given different cache and memory-bank allocations. This work combined recent advances in mixed-criticality scheduling, with work on hardware management, and showed how these two orthogonal approaches can be used together to realize better overall platform utilization.

More recently in this line of work, we have considered challenges raised by data sharing among tasks in the context of both mixed-criticality scheduling and hardware management. Hardware management techniques have predominantly focused on providing isolation with respect to hardware resources, but data sharing must fundamentally break some forms of isolation, thereby causing capacity loss, or a reduction in schedulability. To address this issue, we developed inter- and intra-criticality data-sharing mechanisms, and evaluated the extent to which they reduce data-sharing-related capacity loss.

6.2.2 Contention-Sensitive Locking[2]

While the RNLP enables fine-grained locking, there exist pathological resource-access patterns in which the worst-case blocking behavior is the same or quite similar to existing coarse-grained locking protocols. One such problematic, pathological case occurs when there is a high degree of transitive blocking. For example, consider that $\mathcal{R}_1$ requests $\mathcal{D}_1 = \{\ell_a, \ell_b\}$, $\mathcal{R}_2$ requests $\mathcal{D}_2 = \{\ell_b, \ell_c\}$, $\mathcal{R}_3$ requests $\mathcal{D}_3 = \{\ell_c, \ell_d\}$, *etc.* In this case, a long *transitive blocking chain* can arise, that serializes all requests, even though some requests in the chain do not conflict with one another. To address this issue, we developed the C-RNLP (Ward and Anderson, 2014a; Jarrett et al., 2015), which allows some requests to "cut ahead" of others, so long as it does not increase the blocking of earlier-timestamped requests. In the previous example, it may be possible for $\mathcal{R}_3$ to cut ahead of $\mathcal{R}_2$, and be satisfied concurrently with $\mathcal{R}_1$. By supporting such cutting ahead, we showed that the worst-case blocking bound for each request is then *contention sensitive*, *i.e.*, a function of only the number of requests with which it directly conflicts.

[2]This subsection summarizes results from the following papers:
Jarrett, C., Ward, B., and Anderson, J. (2015). A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*, pages 3–12.
Ward, B. and Anderson, J. (2014a). A contention-sensitive multi-resource locking protocol for multiprocessor real-time systems. In *Proceedings of the 35th IEEE Real-Time Systems Symposium Work-in-Progress Session*, pages 11–12

6.2.3 Improved Soft Real-Time Tardiness Bounds[3]

As discussed briefly in Chapter 2, G-EDF, or other G-EDF-like schedulers, can schedule soft-real-time systems with no capacity loss and bounded deadline tardiness. The tightest known bounds on deadline tardiness are derived through *compliant vector analysis* (*CVA*), which derives per-task tardiness bounds. Using CVA, Erickson and Anderson (2012) showed that by setting the priority point (or the "effective deadline" used as the scheduling priority) differently from the actual deadline as in G-EDF, it is possible to create schedulers with tardiness less than that of G-EDF as computed via CVA. Based on this work, we showed how to formulate CVA as a linear program, which in turn allows for priority points to be determined based on different optimization objectives and constraints (Ward et al., 2013a; Erickson et al., 2014). For example, we considered minimizing the maximum per-task lateness, and *maximum relative lateness*, or l_i/p_i.

6.3 Future Work

Next we discuss future research directions that could improve or build upon the results presented in this dissertation.

6.3.1 GPUs

As described previously, real-time locking protocols can be applied to manage GPUs (Elliott et al., 2013; Elliott, 2015). Task interactions with GPUs can be decomposed into three phases, memory transfer to the GPU, execute on the GPU, and transfer data back to system memory. Current GPUs require that GPU computations execute non-preemptively, and therefore are amenable to being managed by a non-preemptive mutex. However, for the purpose of predictable execution times, memory transfers to the GPU *copy engine*(s) may be modeled using preemptive mutual exclusion. (In practice, due to implementation concerns, *chunks* of data must be transferred non-preemptively.) A potential area of future research may be to manage copy engines as preemptive resources, and to apply idleness analysis to a GPU-management framework such as GPUSync (Elliott et al., 2013; Elliott, 2015).

[3]This subsection summarizes results from the following papers:
Ward, B., Erickson, J., and Anderson, J. (2013a). A linear model for setting priority points in soft real-time systems. In *Proceedings of Real-Time Systems: The past, the present, and the future–A conference organized in celebration of Alan Burns's sixtieth birthday*, pages 192–205.
Erickson, J., Anderson, J., and Ward, B. (2014). Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. *Real-Time Systems*, 50(1):5–47.

LP-based blocking analysis. Recently, Brandenburg (2013b) has pioneered a new methodology for expressing and computing blocking bounds through linear programming. This analysis technique has been shown to result in tighter blocking bounds in some cases. A promising direction for continued research on the RNLP family of locking protocols would be to develop LP-based blocking analysis for different RNLP variants. As briefly discussed in Sections 3.6 and 4.1.2.3, some of the insights in the fine-grained blocking analysis presented herein may be able to be leveraged in a linear program.

Extending idleness analysis. Idleness analysis has been presented herein as a technique for globally scheduled systems. An interesting avenue of future research is to consider how idleness analysis may be extended to partitioned or clustered systems, and whether or not idleness analysis is at all advantageous for such platform configurations.

Also, recently, Yang et al. (2015) presented a new analysis framework in which the effects of blocking are incorporated directed into an LP-based schedulability analysis. This work shares a similar motivation to idleness analysis; traditional blocking analysis can be pessimistic because it assumes all requests experience the worst-case blocking, when in fact only some requests may experience such severe blocking. An interesting avenue for future work is to try to combine these two analysis techniques, perhaps by incorporating ideas or concepts from idleness analysis into the LP-based response-time analysis.

Tight lower bound on synchronization-related utilization loss. In all previous approaches to accounting for synchronization in schedulability, blocking results in utilization loss. For s-oblivious schedulability tests, there exist locking protocols with $O(m)$ worst-case s-oblivious pi-blocking per task. It is possible all n tasks request resources, so the total utilization loss could be $O(nm)$. In the s-aware case, previous work has produced optimal locking protocols with $O(n)$ worst-case blocking per task. Liu and Anderson (2012) presented an s-aware soft real-time schedulability test with $O(m)$ utilization loss, assuming constant suspension lengths. By combining that analysis with an $O(n)$ s-aware blocking bound, the total utilization loss is also $O(nm)$. Finally, the schedulability test for idleness analysis (Ward, 2015) explicitly incorporates an $O(nm)$ utilization-loss term, even for preemptive resources.

Given that each of these three research directions has arrived at similar results, it seems that $\Omega(nm)$ utilization loss may be a tight lower bound for synchronization-related utilization loss. While pieces of this puzzle have been proven optimal, for example, the s-oblivious and s-aware blocking bounds, no lower-bound

results exist for synchronization-related utilization loss. A potential avenue for future research is to show than $\Omega(nm)$ synchronization-related utilization loss is fundamental.

Real-time lock-based transactional memory. Transactional memory (TM) is a paradigm for synchronization that dates back over 20 years (Herlihy and Moss, 1993), but that has seen significant recent interest. In a TM system, programmers need only specify regions of code that must execute effectively atomically, and the TM system resolves all conflicts, and ensures transactions are executed safely. In comparison to the lock-based synchronization mechanisms described previously in Chapter 2, TM has the benefit of being much simpler to use—developers need not concern themselves with coarse- vs. fine-grained synchronization, or which objects to lock and unlock when and where. This is particularly advantageous in safety-critical real-time systems, as parallel algorithms are notoriously difficult to both develop, as well as prove correct. TM has the potential of significantly easing the certification process of parallel algorithms in safety-critical systems, provided a certifiable real-time TM system can be developed.

An interesting avenue for future research is to apply the RNLP in the context of a lock-based real-time TM system, thereby enabling strong run-time parallelism, in addition to improved blocking bounds in comparison to coarse-grained approaches. The most significant hurdle in this line of research is in the static-analysis tools necessary to automatically convert transactional semantics into the appropriate lock/unlock calls to the previously presented locking protocols. To use the RNLP in particular, static-analysis tools must be able to determine at compile time all objects that may potentially be accessed within the transaction.

The results of static analysis are necessary in order to determine which resources to lock and unlock, as well as compute blocking bounds for the RNLP. However, if determining the set of potentially accessed resources is too difficult or pessimistic, less-pessimistic locking protocols (recall the RNLP is not work conserving, in order to attain asymptotic optimality) could be used in conjunction with idleness analysis, which was discussed in Section 5.2.4.2. Bounding the amount of idleness a particular transaction can induce in the system may be less pessimistic than bounding the amount of blocking it may experience, particularly if the set of accessed resources must be grossly over-approximated. Furthermore, idleness analysis may enable the use of less-complex, lower-overhead locking protocols to be used instead of the RNLP. However, great care must be taken to avoid deadlock, which is a significant concern when not leveraging a deadlock-free algorithm such as the RNLP.

BIBLIOGRAPHY

Altmeyer, S., Davis, R., and Maiza, C. (2012). Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526.

Altmeyer, S., Douma, R., Lunniss, W., and Davis, R. (2014). Evaluation of cache partitioning for hard real-time systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 15–26.

Baker, T. (1991). Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99.

Baruah, S. (2007). Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 119–128.

Baruah, S., Bertogna, M., and Buttazzo, G. (2015). *Multiprocessor Scheduling for Real-Time Systems*. Embedded Systems. Springer.

Baruah, S., Cohen, N., Plaxton, C., and Varvel, D. (1996). Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625.

Baruah, S., Mok, A., and Rosier, L. (1990). Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190.

Bertogna, M. and Baruah, S. (2011). Tests for global EDF schedulability analysis. *Journal of Systems Architecture*, 57(5):487–497.

Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56.

Brandenburg, B. (2011). *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC.

Brandenburg, B. (2013a). A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 292–302.

Brandenburg, B. (2013b). Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 141–152.

Brandenburg, B. (2014). The FMLP$^+$: an asymptotically optimal real-time locking protocol for suspension-aware analysis. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 61–71.

Brandenburg, B. and Anderson, J. (2007). Feather-trace: A light-weight event tracing toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 61–70.

Brandenburg, B. and Anderson, J. (2009). Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 184–193.

Brandenburg, B. and Anderson, J. (2010a). Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60.

Brandenburg, B. and Anderson, J. (2010b). Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87.

Brandenburg, B. and Anderson, J. (2011). Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and *k*-exclusion locks. In *Proceedings of the 11th ACM International Conference on Embedded Software*, pages 69–78.

Brandenburg, B. and Anderson, J. (2014). The OMLP family of optimal multiprocessor real-time locking protocols. *Design automation for embedded systems*, 17:277–342.

Bui, B., Caccamo, M., Sha, L., and Martinez, J. (2008). Impact of cache partitioning on multi-tasking real time embedded systems. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 101–110.

Burks, A., Goldstein, H., and von Neumann, J. (1946). Preliminary discussion of the logical design of an electronic computing instrument. Report to the U.S. Army Ordnance Department. Also appears in Computer Structures: Readings and examples., McGraw-Hill Inc., 1971, 92–120.

Burns, A. and Davis, R. (2016). Mixed criticality systems – a review. Technical report, Department of Computer Science, University of York.

Burns, A. and Wellings, A. (2013). A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 282–291.

Certification Authorities Software Team (CAST) (2014). Position paper CAST-32 multicore processors. `https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf`.

Chen, J.-J., Nelissen, G., Huang, W.-H., Yang, M., Brandenburg, B., Bletsas, K., Liu, C., Richard, P., Ridouard, F., Audsley, N., Rajkumar, R., and Niz, D. (2016). Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, Department of Computer Science, TU Dortmund. (Status: Preprint).

Chisholm, M., Kim, N., Ward, B., Otterness, N., Anderson, J., and Smith, F. (2016). Reconciling the tension between hardware isolation and data sharing in mixed-critical, multicore systems. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (to appear)*.

Chisholm, M., Ward, B., Kim, N., and Anderson, J. (2015). Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Proceedings of the 36th Real-Time Systems Symposium*, pages 306–316.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

Dertouzos, M. and Mok, A. (1989). Multiprocessor scheduling in a hard real-time environment. *IEEE Transactions of Software Engineering*, 15(12):1497–1506.

Devi, U. (2006). *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC.

Devi, U. and Anderson, J. (2005). Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341.

Elliott, G. (2015). *Real-Time Scheduling of GPUs, with Applications in Advanced Automotive Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC.

Elliott, G. and Anderson, J. (2012). Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48:34–74.

Elliott, G. and Anderson, J. (2013). An optimal *k*-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170.

Elliott, G., Ward, B., and Anderson, J. (2013). GPUSync: A framework for real-time GPU management. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, pages 33–44.

Erickson, J. (2014). *Managing Tardiness Bounds and Overload in Soft Real-Time Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC.

Erickson, J. and Anderson, J. (2012). Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 3–11.

Erickson, J., Anderson, J., and Ward, B. (2014). Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. *Real-Time Systems*, 50(1):5–47.

Erickson, J., Kim, N., and Anderson, J. (2015). Recovering from overload in multicore mixed-criticality systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium*, pages 775–785.

Faggioli, D., G., L., and Cucinotta, T. (2010). The multiprocessor bandwidth inheritance protocol. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 90–99.

Faggioli, D., Lipari, G., and Cucinotta, T. (2012). Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems*, 48(6):789–825.

Fernàndez, M., Gioiosa, R., Quiñones, E., Fossati, L., and Cazorla, F. (2012). Assessing the suitability of the NGMP multi-core processor in the space domain. In *Proceedings of the 10th ACM International Conference on Embedded Software*, pages 175–184.

Fisher, N., Goossens, J., and Baruah, S. (2010). Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2):26–71.

Gai, P., Lipari, G., and Di Natale, M. (2001). Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 73–83.

Gai, P., Natale, M. D., Lipari, G., Ferrari, A., Gabellini, C., and Marceca, P. (2003). A comparison of MPCP and MSRP when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 189–198.

Hennessy, J. and Patterson, D. (2007). *Computer architecture: A quantitative approach*. Morgan Kaufman.

Herlihy, M. and Moss, Moss, J. (1993). Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300.

Hohmuth, M. and Peter, M. (2001). Helping in a multiprocessor environment. In *Proceedings of the Second Workshop on Common Microkernel Systems Platforms*.

Holman, P. and Anderson, J. (2006). Locking under Pfair scheduling. *ACM Transactions of Computer Systems*, 24(2):140–174.

Hong, K. and Leung, J. (1988). On-line scheduling of real-time tasks. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 244–250.

Jarrett, C., Ward, B., and Anderson, J. (2015). A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*, pages 3–12.

Joseph, M. and Pandya, P. (1986). Finding response times in real-time systems. *The Computer Journal*, 29(5):390–395.

Jouppi, N. (2016). Google supercharges machine learning tasks with TPU custom chip. `https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html`.

Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y., and Rajkumar, R. (2011a). RGEM: a responsive GPGPU execution model for runtime engines. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 57–66.

Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. (2011b). TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Annual Technical Conference*, pages 17–30.

Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. (2012). Gdev: First-class GPU resource management in operating systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 401–412.

Kessler, R. and Hill, M. (1992). Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359.

Kim, H., Kandhalu, A., and Rajkumar, R. (2013). A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 80–89.

Kim, N., Ward, B., Chisholm, M., Fu, C.-Y., Anderson, J., and Smith, F. (2016). Attacking the one-out-of-*m* multicore problem by combining hardware management with mixed-criticality provisioning. In *Proceedings of the 22nd Real-Time Embedded Technology and Applications Symposium*, pages 149–160.

Kirk, D. (1989). SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 229–237.

Krishnapillai, Y., Wu, Z., and Pellizzoni, R. (2014). ROC: A rank-switching, open-row DRAM controller for time-predictable systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 27–38.

Lakshmanan, K., Niz, D., and Rajkumar, R. (2009). Coordinated task scheduling, allocation and synchronization on multiprocessors. pages 469–478.

Lee *et al*., C.-G. (1998). Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computing*, 47(6):700–713.

LITMUS^{RT} Project (2016). http://www.litmus-rt.org/.

Liu, C. (2013). *Efficient Design, Analysis, and Implementation of Complex Multiprocessor Real-Time Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC.

Liu, C. and Anderson, J. (2012). An $O(m)$ analysis technique for supporting real-time self-suspending task systems. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium*, pages 373–382.

Liu, C. and Anderson, J. (2013). Suspension-aware analysis for hard real-time multiprocessor scheduling. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, pages 271–281.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61.

Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M., and Pellizzoni, R. (2013). Real-time cache management framework for multi-core architectures. In *Proceedings of the 19th IEEE Real-Time Embedded Technology and Applications Symposium*, pages 45–54.

McKenney, P. (2009). "real time" vs. "real fast": How to choose? In *Proceedings of the 11th OSADL Real-Time Linux Workshop*, pages 1–12.

McVoy, L. and Staelin, C. (1996). lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–294.

Mok, A. (1983). *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA.

Mueller, F. (1995). Compiler support for software-based cache partitioning. In *Proceedings of the ACM Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 125–133.

Nellisen, G. (2012). *Efficient Optimal Multiprocessor Scheduling Algorithms for Real-Time Systems*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium.

Rajkumar, R. (1990). Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123.

Rajkumar, R. (1991). *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers.

Rajkumar, R., Sha, L., and Lehoczky, J. (1988). Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 259–269.

Regnier, P., Lima, G., Massa, E., Levin, G., and Brandt, S. (2011). RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 104–115.

Ridouard, F., Richard, P., and Cottet, F. (2004). Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proceedings of the 25th Real-Time Systems Symposium*, pages 47–56.

SchedCAT (2016). `http://www.mpi-sws.org/~bbb/projects/schedcat`.

Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: an approach to real-time synchronization. *Transactions on Computers*, 39(9):1175–1185.

Srinivasan, A. and Anderson, J. (2002). Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198.

Stigge, M. and Yi, W. (2015). Graph-based models for real-time workloads: a survey. *Real-Time Systems*, 51(5):602–636.

Valsan, P., Yun, H., and Farshchi, F. (2016). Taming non-blocking caches to improve isolation in multicore real-time systems. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 161–172.

Ward, B. (2015). Relaxing resource-sharing constraints for improved hardware management and schedulability. In *Proceedings of the Real-Time Systems Symposium*, pages 153–164.

Ward, B. and Anderson, J. (2012). Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 223–232.

Ward, B. and Anderson, J. (2013). Fine-grained multiprocessor real-time locking with improved blocking. In *Proceedings of the 21st Conference on Real-Time Networks and Systems*, pages 67–76.

Ward, B. and Anderson, J. (2014a). A contention-sensitive multi-resource locking protocol for multiprocessor real-time systems. In *Proceedings of the 35th IEEE Real-Time Systems Symposium Work-in-Progress Session*, pages 11–12.

Ward, B. and Anderson, J. (2014b). Multi-resource real-time reader/writer locks for multiprocessors. In *Proceedings of the 28th Parallel and Distributed Systems Symposium*, pages 177–186.

Ward, B., Elliott, G., and Anderson, J. (2012). Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 280–289.

Ward, B., Erickson, J., and Anderson, J. (2013a). A linear model for setting priority points in soft real-time systems. In *Proceedings of Real-Time Systems: The past, the present, and the future–A conference organized in celebration of Alan Burns's sixtieth birthday*, pages 192–205.

Ward, B., Herman, J., Kenna, C., and Anderson, J. (2013b). Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 157–167.

Ward, B., Thekkilakattil, A., and Anderson, J. (2014). Optimizing preemption-overhead accounting in multiprocessor real-time systems. In *Proceedings of the 22nd Conference on Real-Time Networks and Systems*, pages 235–243.

Wasly, S. and Pellizzoni, R. (2014). Hiding memory latency using fixed priority scheduling. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86.

Wieder, A. and Brandenburg, B. (2013). On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, pages 45–56.

Xu, M., Mohan, S., Chen, C., and Sha, L. (2016). Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 123–134.

Yang, M., Chen, J.-J., and Huang, W.-H. (2016). A misconception in blocking time analyses under multiprocessor synchronization protocols. Private communication.

Yang, M., Wieder, A., and Brandenburg, B. (2015). Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *Proceedings of the 36th IEEE Real-Time Systems Symposium*, pages 1–12.

Yun, H., Mancuso, R., Wu, Z., and Pellizzoni, R. (2014). PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 155–166.

Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., and Sha, L. (2013). MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–64.