A Flexible Multiprocessor Resource Sharing Framework for Ada

Shiyao Lin

Submitted for the Degree of Doctor of Philosophy

University of York
Department of Computer Science

September 2013

Abstract

Lock-based resource sharing protocols for single processor systems are well understood and supported in programming languages such as Ada and the Real-Time Specification for Java, and in Real-Time Operating Systems, and those that conform to the Real-Time POSIX standard. In contrast, multiprocessor resource sharing protocols are still in their infancy with no agreed best practices, and yet current real-time programming languages and operating systems claim to be suitable for supporting multiprocessor applications. This thesis argues that, instead of supporting a single resource sharing protocol, a resource sharing framework should be provided that allows application-defined resource sharing protocols to be implemented. The framework should be flexible and adaptive so that a wide range of protocols with different design characteristics can be integrated and implemented effectively with minimum runtime overheads. The thesis reviews the currently available multiprocessor resource allocation policies and analyzes their applicability to the main industry standard real-time programming languages. It then proposes a framework that allows programmers to define and implement their own locking policy for monitor based concurrent control mechanisms. Instantiation of the framework is illustrated for the Real-Time Specification for Java and POSIX. A prototype implementation of the full framework for Ada is developed and evaluated.

Contents

List of Figures

Symbol	Description
P_i	Processor i
T_i	A specific task with id i
t_i	A specific global time i
M	The number of processors in the system
R_i	A General shared resource
G_i	A Global shared resource
S_i	A Short shared resource
L_i	A Long shared resource
α_i	PPCP threshold for priority level i
$Ceil(R)$	Ceiling of resource R
P_H	Starting priority of a global resource
FQ	FIFO queue
PQ	Priority queue

Table 1: **List of Symbols**

Acknowledgement

It would not have been possible to write this thesis without the help and support of the people around me, only some of whom it is possible to mention here:

I would like to express my deepest appreciation to Profession Andy Wellings: he has continually and convincingly conveyed the spirit of research, and the excitement of achieving goals with world-class academics. Without his guidance and persistent help, my research work and this dissertation would not have been possible.

The appreciation will also go to Dr. Thomas Richardson for the advice, unequivocal support and the valuable friendship which has been a great accelerator of the whole work. The welcomed distractions has been a great balance preventing the work from being too stressful.

In addition, a thank you to my friends Abdul Haseeb Malik, Seyeon Kim and many people in the RTS group for their feedbacks and opinions. These have been of critical importance in refining ideas and identifying opportunities.

Finally, I would like to thank my parents together with my sister for all their love and support throughout my life and never doubting that I would get to the end.

Declarations

I declare that the research work described in this thesis is original work unless otherwise indicated. The research was conducted by me with the supervision of Professor Andy Wellings. Certain parts of this thesis have appeared in previously published papers.

- In Chapter 3 and Chapter 4, the design and implementation of the framework is based on: *S. Lin, A. J. Wellings, and A. Burns. Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages. Concurrency and Computation: Practice and Experience, 2012.*

- In Chapter 4, supporting the framework in Ada is based on this publication: *S. Lin, A. Burns, and A. J. Wellings. Ada 2012: Resource sharing and multiprocessors. In Proceeding of the 15th IRTAW, volume XXXII, pages 32-44.ACM Letters, April 2013.*

- In Chapter 3, supporting the framework in RTSJ is based on: *A. J. Wellings, S. Lin, and A. Burns. Resource sharing in RTSJ and SCJ systems. In JTRESS, pages 11-19, 2011.*

Chapter 1

Introduction

1.1 Motivation

The term "real-time system" does not refer to a system with a super fast processing speed. It denotes the system with the ability to respond to real-world events in real-world time. The correctness of those systems is depending on the completion time of the tasks [4]. Producing a response later than the completion time, which is also known as deadline missing in real-time systems, will be erroneous and may have a significant impact to the system and its operating environment. As the system may need to react with their environment in a timely fashion, the real-time system is commonly distinguished into the hard and soft real-time systems. The hard real-time systems are very sensitive to the satisfaction of the deadlines of the tasks. The failure to meet a hard deadline may create catastrophic damage to the system or the surrounding environment. The soft real-time systems are less sensitive to deadline misses. Although strongly undesirable, they are tolerant of missing deadlines. This is because the application environment often applies mixed hard and soft real-time requirements to the system.

The categorization of hard and soft real-time systems inspires the idea of understanding more of its operating environments for a better system design. This idea soars not only in real-time systems but general systems engineering [65]. Solving real-time problems in multiprocessor systems is likely to benefit from having more considerations for the operating scenarios in the system design.

At the systems requirement level, real-time systems must satisfy the following

characteristics [15].

Large and Complex real-time systems have to support systems from the small single processor to the large multiprocessor distributed systems.

Extremely Reliable and Safe real-time systems are often deployed to mission-critical systems. The software must, therefore, be engineered to the highest integrity. The implementations of real-time systems should comprise minimum programming errors and the execution results must follow the expectations. Hence, the predictability of real-time system is emphasized.

Interaction with Hardware Interfaces The real-time systems are often sensitive of multiple infrastructure layers because of the overheads. It is often the case that real-time systems are deployed as embedded. It is therefore necessary for the real-time systems to have the ability to interact with the hardware and low level interfaces.

Efficient Implementation and Predictable Execution Environment Since real-time systems are time-critical, the implementation efficiency is more important than other systems. Being predictable to its execution is the key to the correctness of real-time systems.

The development of multiprocessor systems imposes a challenge to real-time systems. General purpose processor manufacturers have started to provide more processor cores on the same chip [17]. This is the industry's response to the limitations of processor clock speed and the exponential increasing cost of producing faster uniprocessor chips [41]. Modern processor technology is already edging towards its limits. It is clear that the standard uniprocessor architecture will not be able to support the need of computing power, even when taking into account the integrated circuit technology advances. Exploiting more Instruction Level Parallelism (ILP) has proven difficult with current process technologies [7].

Having more processing power to the real-time systems seems to be a significant improvement. As indicated by [71], allowing more than one processors to solve a real-time problem will exploit more parallelism in multiprocessor systems. It is a more natural reflection of the operating environment. For example, the sensors of an aircraft engine will be working independently and in true parallel to each other.

Reflecting the parallel nature of the systems will make the implementation more readable, maintainable and reliable.

However, having concurrent executions in multiprocessor systems endangers the satisfaction of the high level real-time system requirements. Parallel executing tasks have to coordinate their executions especially for the complex problems. Failure to do so will endanger the result of the execution and jeopardize the whole system. A typical phenomenon of having uncontrolled parallel execution is described by Section 1.4.1. Therefore, coordinating the execution of parallel tasks is essential and it may vary in the forms of synchronization and communication. These coordination, which is also known as resource sharing, in multiprocessor is complicated.

The characteristics of multiprocessor real-time scheduling algorithms have been described by [20]. The optimality and predictability are the two most important properties attracted the interest of this research. The predictability is important because it emphasizes on the real-time property of the tasks that even the worst-case scenario must be acceptable by the application scenario [51]. The resource sharing algorithms should therefore be considered for their worst case behaviour where the highest priority task suffers the maximum blocking. The other property is optimality. This property is crucial to this research as it is the fundamental observation and assumption of the work. According to the definition given in [20], an optimal algorithm should be able to schedule all of the task sets that can be scheduled by any other algorithms. Whenever such an algorithm exists, the resource sharing algorithm design can therefore be more focused. However, due to the complex nature of scheduling tasks in multiprocessor systems, there is no optimal multiprocessor resource sharing algorithm yet known to the academics.

The methods of multiprocessor resource sharing are still underdeveloped [23]. The performance optimality of multiprocessor resource sharing is much more complex to determine than its counterpart uniprocessor algorithms. The predictability property still has to be satisfied as a precedent in multiprocessor resource sharing protocol development. The maximum blocking time suffered by the highest priority task must therefore be determined and bounded by the deployment of resource sharing protocols, which then assures the predictability of the whole algorithm. The performance factors of multiprocessor resource sharing algorithms are still being studied and measured. There is a number of known factors, at the imple-

mentation level, which are critical to the performance of multiprocessor scheduling algorithms:

- Scheduling algorithm: Partitioned, Global or Hybrid
- Task allocation for Partitioned or Hybrid systems
- Mutual exclusion waiting mechanism: Spin or Suspension
- Queuing policy: FIFO or Priority Queue
- Preemptive or non-preemptive scheduling
- Resource accessing priority: original, ceiling or priority inheritance
- Degree of mutual exclusion: spin, Read/Write Lock or complete mutual exclusion

Theses factors will be discussed in the following subsections.

1.2 Processor Architecture

In multiprocessor systems, the performance of executing tasks is closely linked with the processor architecture. Processors in multiprocessor systems are often connected to each other in order to share resources. Shared memory multiprocessors consist of a number of processors connected with a shared memory area. This shared memory provides a means for processors to communicate [30].

There are two types of shared memory multiprocessor architecture, namely the uniform memory accecss (UMA) and the non-uniform memory access (NUMA). UMA is a multiprocessor architecture where all processors access the shared memory via a central switching mechanism [47]. It is often the case that all processors are connected to a common bus where the cost of the memory access is identical. The NUMA system, in contrast, does not have constant nor identical memory access costs [67]. Remote memory access normally incurs high latency.

Despite the difference in hardware connectivity, the NUMA system imposes no significant difference to the UMA system with regard to resource sharing in multiprocessors. The method in which the application-defined resource sharing

protocols integrates with programming languages should be unaffected at a high level. The processor architecture will only impact the low level interfacing and resource sharing algorithms design so as to incorporate the extra memory access cost. Therefore, in the interest of the resource sharing protocol framework design and evaluation, we are concerned only with the shared memory multiprocessor systems and assumed the underlying system architecture is UMA where all processors have an undeviating cost to access the shared memories.

1.3 Scheduling Algorithms

The multiprocessor resource sharing algorithm works in collaboration with the scheduling algorithms to intervene in the execution of tasks. The choice of the underlying scheduling algorithm will have a great impact on the type and performance of the resource sharing algorithms. The blocking time of the highest priority task will therefore have different result depending on those choices. Unfortunately, it is still a debate on which scheduling algorithm works the best for sharing resources in multiprocessor systems. In partitioned scheduling, tasks are statically assigned to dedicated processors for their entire execution. Tasks under global scheduling are not subject to such restriction and can migrate from one processor to other. At run time, the executing tasks may suspend from one processor and migrate to continue their remaining execution [44]. Renowned for its simplicity and predictability, partitioned scheduling has received more research attention than global scheduling [3]:

1. Partitioned scheduling appears to have an advantage of the best feasibility tests over global scheduling with respect to the statistical chance of being able to schedule an arbitrary hard-deadline task set.

2. Experience and algorithms developed in uniprocessor scheduling can be applied to each scheduling partition. The reason is that, under fully partitioned scheduling, processors are unified independently. Each processor, thus, has its own run queue. The environment for each processor is more or less the same as it was in the uniprocessor.

The fully partitioned scheduling algorithm fixes task execution affinity. The processor assignment of the tasks is known prior to the start of their execution.

The processor assignment is a difficult challenge with significant impact on the performance of the partitioned scheduling algorithms. The assignment process is known to be a typical NP-hard bin-packing problem [37]. A common solution is to deploy related heuristic functions to find a satisfactory assignment. However, each of the heuristic functions has its own advantages and disadvantages. The optimal situation is one where the affinity assignment uses the least number of processors [43]. Although we can measure how many more processors are needed to schedule a task set in the worst-case scenario, the processor capacity of each processor cannot be fully utilized as the bin packing solutions cannot guarantee the sum of allocated task utilization is a perfect match for the processor capacity. In this case, some of the processor capacity will be too little for any tasks to fit in, which will be wasted as well.

Global scheduling is appealing in the context of processor utilization. The advantages of global scheduling against partitioned scheduling are summarized as follows [20]:

1. Global scheduling is more suitable for open systems where tasks are capable of running on any processor in the system.

2. Since all tasks are managed by a single run queue with global scheduling, resource sharing is more manageable.

Davis and Burns [20] explained that, in general, multiprocessor scheduling algorithms are incomparable to each other. This is because some task sets, although schedulable by some algorithms, may not be schedulable by others. Through an empirical comparison, [5] explained that the worst-case performance of both fully partitioned and global scheduling is about the same. This leaves no advantage of either approach.

1.3.1 Hybrid Scheduling

The argument for the optimality of partitioned and global scheduling can be long and tedious. Hybrid scheduling was proposed to support more scalable operating architecture by scheduling tasks according to the similarity of their runtime requirements.

Tasks executed in multiprocessors may incur different costs to access shared memories. The cost of accessing memory can either be classified as Uniform Memory Access (UMA) costs or Non-uniform Memory Access (NUMA) costs depending on the remoteness of the data [62]. In an SMP system, the locality of the data is often ignored as memory is shared as a single entity across all processors in the system. The cost of accessing the memory is assumed to be constant for all processors; this is often referred as UMA access.

With this processor architecture, the main benefit of implementing hybrid scheduling, also known as clustering in distributed systems, is that high data locality can be achieved through distributing and replicating system services and data resources [63]. Also, a hierarchical (clustered) system can be easily adapted to different hardware configurations and architectures with the strength of scaling the underlying architecture. Hierarchical structure incorporates the principle of structuring from loosely-coupled and distributed systems [66]. The basic unit of structuring is a cluster. This is regarded as the basic unit of the system which provides a complete set of services as a part of the underlying system. Data and system services are expected to be shared by all processors and tasks in the same cluster. The grouping of clusters offers the mechanism of inter-cluster communication and integrate the consistent underlying clusters as a sub-system. Due to the nature of memory contention of shared resources, specific algorithm is required for inter-cluster communications.

Chandra and Prashant[18] offers the definition of hierarchical scheduling based on the above principles. Hierarchical scheduling is a scheduling framework that groups together processes, threads and applications to achieve aggregate resource partitioning. This framework enables the allocation of resources to collections of entities to benefit from the hierarchical underlying hardware structure.

1.4 Resource Sharing

1.4.1 Resources

Parallel programs can be non deterministic. This behaviour is hazardous to real-time systems as the result of execution is unpredictable [42]. Synchronization techniques are often deployed to ensure the integrity of the shared data between

tasks. Shared data access is often called a critical section, where execution is supposed to be atomic. Without proper synchronization, a critical section may be modified by other tasks so that data integrity is violated. The result of this program execution is non deterministic. Figure 1.1 exhibits the effect of race condition without proper synchronization:

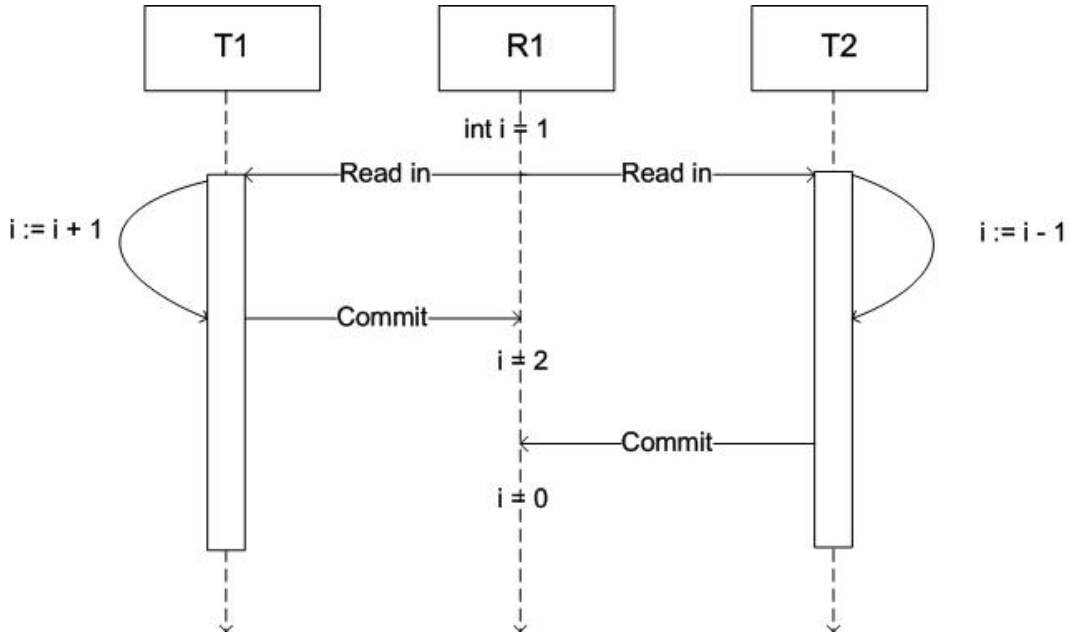

Figure 1.1: Race Condition

Let us assume we have two tasks T1 and T2 running parallel to each other on processor P1 and P2. A resource R_1 with initial value of 1 is shared between two tasks. T1 contains a single loop trying to increment R_1 by 1 and T2, in contrast, is trying to deduct 1 from R_1 on each loop. Each loop contains a read operation, that has a current value of R_1, and write operation, that commits the final value of the operation to R_1. Without proper synchronization, T1 and T2 can simultaneously read the initial value from R_1. Let us assume T1 commit the arithmetic result first. The value of R_1 will be 2 temporarily before T2 commit. Since T2 works on the initial value 1 instead of 2, the result committed to R_1 by T2 is 0. It is difficult to determine the final value of R_1 as it depends on the committing order of T1 and T2. With proper locking protocol, T2 can neither read nor write R_1 before T1 has committed. This will prevent such race condition and ensures that

the value of R_1 is always 1.

The resources mentioned in this thesis refer to the shared objects at the programming language level. At the low level, the resource primarily stands for the physical memory units. Although the physical memory has more details like clock and cache coherence considerations, both kinds of resources need mutual exclusion which is provided by semaphores and locks etc. They are both viewed as generic shared objects from the language perspective.

In this thesis, we consider three types of resource sharing primitives that differ with respect to their sharing constraint. The mutual exclusive resource requires serialized access where only one task is allowed to use the resource at any time. The resource holding task may conduct mixed read and write operations. The mutual exclusive lock ensures the integrity of the data is protected so that no other tasks can access the resources without holding the lock. Since the task bodies are sequentially executed, the internal usage of the resource is serialized.

The degrees of parallelism can be increased by allowing multiple tasks to have parallel read access to a shared resource at any one time. The reader-writer resource deploys two protocols to the acquiring task depending on the nature of the access. The write access of the task is exclusive where only one writer is allowed to execute the resource at any time. The reader can have simultaneous access to the resource as the shared data can only be changed by the writers (RW synchronization). The mutual exclusive and RW resource have diversified behavior under different scenarios. The mutual exclusive resources are likely to chain all waiting tasks that the blocking time for the tailing task can be quite high. The RW lock incorporate certain degrees of parallelism to the waiting tasks so that the average blocking time is likely to be lower. However, starvation could be an important issue where it is difficult to assign appropriate priorities to the read and write phase appropriately.

The spin lock is another fundamental synchronization primitive for resource sharing on SMP systems. It is usually implemented through atomic read-modify-write operations and busy waiting on a single word [69]. Since the tasks will be spinning and waiting for the lock instead of suspending, it is called a spin lock. When a task attempts to acquire a lock for a resource, the protocol appends its requests to the end of the spin queue. The tasks in the spin queue are often served in FIFO manner. The task at the head of the run queue after accessing the

resources, updates the spin variable for the next task in the queue so that the next task is capable of locking the resource. This is also called spin lock replenishment. [21] highlighted two advantages of spin locks: 1) Each task spinning variable can be chosen to be locally cached. This approach ensures waiting tasks do not generate excessive bus traffic. 2) Since the queue is FIFO ordered, the waiting time can be easily determined.

Whether it be spinning or suspension, the delaying approach is orthogonal to the type of locks being acquired. It is often the case that the RW lock implementation uses mutex as the lock to suspend the readers waiting from an unfinished writer. The readers in this case are not only limited to be suspended but spinning is also permitted depending on the implementation decisions of the developers. In this thesis, we consider only mutual exclusion locks, as these are basic mechanisms widely available in programming languages.

Resources can also be shared by using lock-free and wait-free algorithms. The motivation for considering alternative methods is that the cost of locking protocols can be significant when the length of critical section is short. The lock-free methods incorporate a retry loop on each task taking snapshots of the state of the shared data on each attempt [31]. The attempt at using the resource can either succeed or fail. A failed attempt will have no effect on the current status of the resource. In wait-free methods, shared resources are executed by carefully designed sequential code. Tasks with wait-free methods must be guaranteed to correctly finish their operations with bounded number of instructions [60]. Wait free methods are strengthened lock-free algorithms where the execution progress of every task in the system is guaranteed.

As a new alternative to the traditional lock based approach, the non-blocking methods possess certain advantages:

1. They avoid the priority inversion phenomenon which cause long blocking chains and complex scheduling analysis for locking methods.

2. They eliminate deadlocks as no locks are used.

3. If the codes are carefully designed, interference between different tasks is completely eliminated.

However, in order to be useful to real-time systems, resource sharing protocols must

be time efficient and predictable. Non-blocking methods often have high space demand and are hardware dependent [72]. Schedulability analysis of non-blocking methods is still underdeveloped. Therefore, this thesis concentrates mainly on lock based blocking algorithms.

1.4.2 Blocking

When locks are used, tasks issue resource requests to the system scheduler for mutual exclusive access. If such requests are not satisfied for various reasons, the issuing task is said to be blocked. Once approved, the issuing task can proceed to the critical section of the resource. Other issuing tasks will be blocked. The blocked task can have various effects on other executing or non-executing tasks in the system. The impact of the blocking effect can propagate to other processors and affect all tasks in the system. Depending on the significance of the effect, the blocking can be categorized as follows, if FIFO within priority scheduling is used:

Local Blocking Tasks are distributed to different processors to be executed in a multiprocessor system. Regardless of which scheduling algorithm is used, only one task can be executed on one processor at any time. The highest priority task can be blocked by a lower priority task on the same processor if the required shared resource is not available and is held by the lower priority task. When this happens on the same processor, the highest priority task has to wait for the lower priority task to finish the resource before being resumed for execution. The resource execution time of the lower priority task is therefore added to the blocking time suffered by the highest priority task.

Local blocking is depicted by Figure 1.2. Task T1 and T2 are both assigned to processor 1. T2 is released at t0 and soon acquires resource R_1 shared between T1 and T2. When T1 is released at t1, T2 is preempted because T1 has higher priority. T1 executes until t2 when shared resource R_1 is needed. The resource, however, is being held by T2. T2 is then resumed to finish its execution in R_1 and releases the resource at t3. Once the resource becomes available, T1 has the opportunity to lock the resource and finish its execution at t4. The blocked task and the resource holding task happen to be on the same processor.

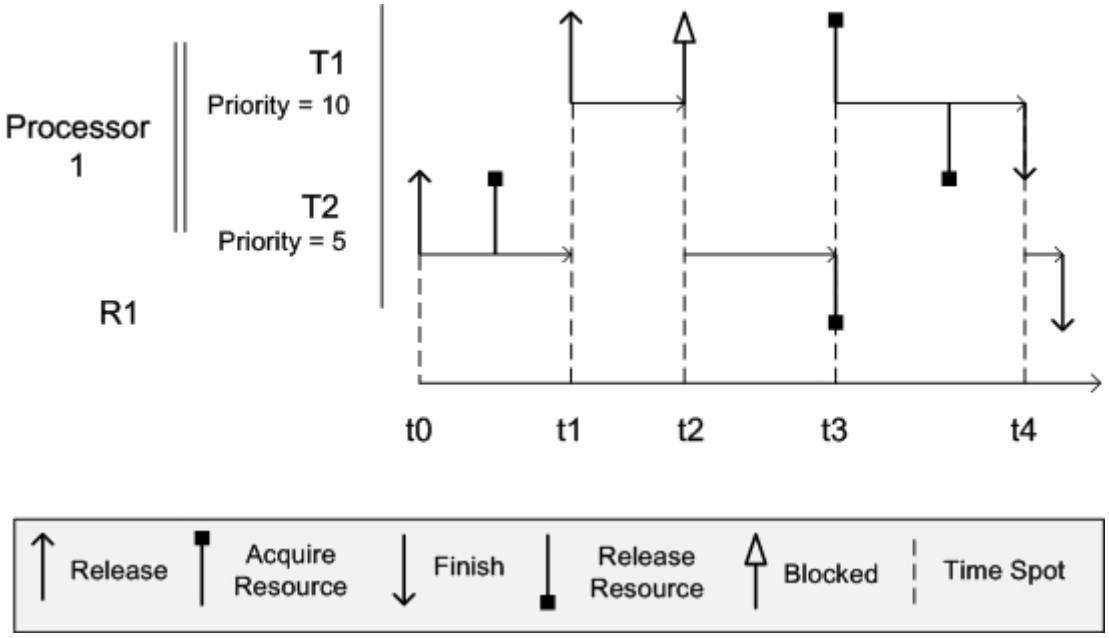

Figure 1.2: Direct Local Blocking

Remote Blocking In multiprocessor systems, resources may be shared between tasks running on different processors. Synchronization is therefore required on two processors where one processor must stop and wait for the other to finish execution first. The highest priority task being blocked on a remote processor can suffer extra blocking time on the processor remote to the resource. This is because the resource required by tasks on a remote processor may be held by a lower priority task on that processor. The following figure demonstrates a typical remote blocking scenario in detail:

In Figure 1.3, T1 and T2 are running on processor 2. T3 is running on processor 1. At t0, T3 and T2 are released on processor 1 and processor 2 respectively. All three tasks share the same resource R_1. R_1 firstly acquired by T2 at t1. At t2, T3 requires R_1 and will not be able to proceed as R_1 has already been locked by T2. T3, in this case, suffers blocking from T2 on processor 2. At t4, T1 attempts to lock R_1 but becomes blocked as R_1 is held by T2. The suspension length of T3 depends on the status of R_1 on processor 2. Although the resources are released by T2 on t5, T1 with higher priority will lock R_1 immediately. T3 has to wait till t6 when the resource is finally available. It can then proceed to the critical

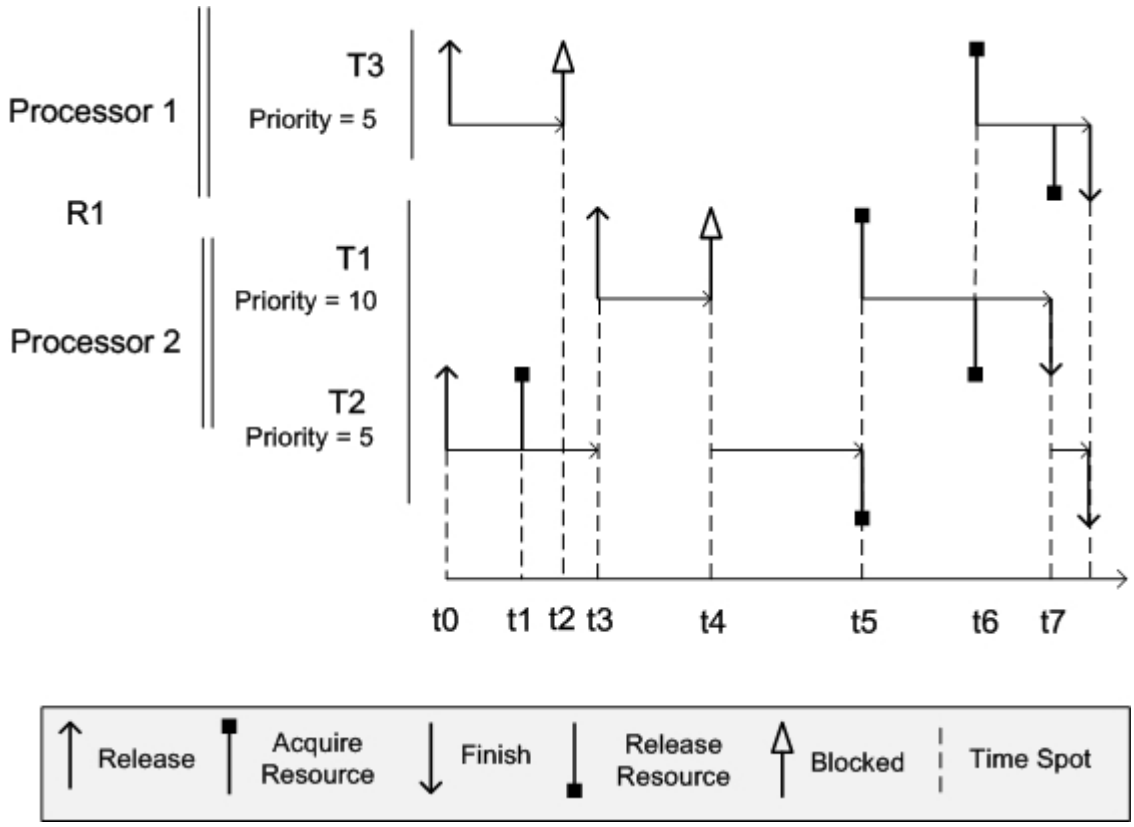

Figure 1.3: Remote Blocking

section of R_1 and finishes execution soon after t7. Without proper multiprocessor resource sharing protocol, T3 can suffer indefinite blocking from tasks on processor 2 with higher priority than T2, and therefore, it is highly likely that T3 will miss its deadline due to this blocking effect. The blocking suffered by T3 can be unbounded because T2 on processor 2 can be preempted by more higher priority tasks than T1. The execution time of all these tasks are added to the blocking time of T3. In order to maintain the predictability property, it is essential to bound this remote blocking with the use of a resource sharing protocol.

Transitive Blocking In a typical multiprocessor real-time system, there are multiple instances of shared resources. The interconnections between the different shared resources can be significant so that tasks waiting on different resources can be queued up in a single queue. No other tasks can proceed until the resource request of the non-preemptive task is satisfied. This is demonstrated by the following example:

Typical transitive blocking involves multiple shared resource. Each task in Figure 1.4 acquires two resources. T1 acquires R_1 then R_2. T2 acquires R_2 first then R_3. T3 acquires R_3 only. All tasks are assigned to independent processors. At t0, all tasks are released to their home processor. T1 acquires R_1 at t1. At the same time, T2 acquires R_2 on processor 2. At t2, T1 acquires R_2 within the critical section of R_1. However, R_2 is not available as it has been locked down by T2. At this moment, T3 locks R3 on processor 3. At t3, T2 acquires R_3 but fails because R_3 is being held by T3. Although T1 does not share any resources with T3, T1 indirectly joins the waiting queue headed by T3. T1 can only proceed with its execution when T3 finishes its execution in R_3 and T2 releases R_2 on processor 2. This kind of blocking is transitive; there could well be another task that acquires R_3 ahead of T3. In this case, T1 is blocked further.

1.4.3 Priority Inheritance - A Solution for Priority Inversion

The priority inheritance technique was designed to solve the priority inversion problem. Priority inversion is a typical resource sharing phenomenon where a high priority task can be blocked by a low priority task holding the resource being

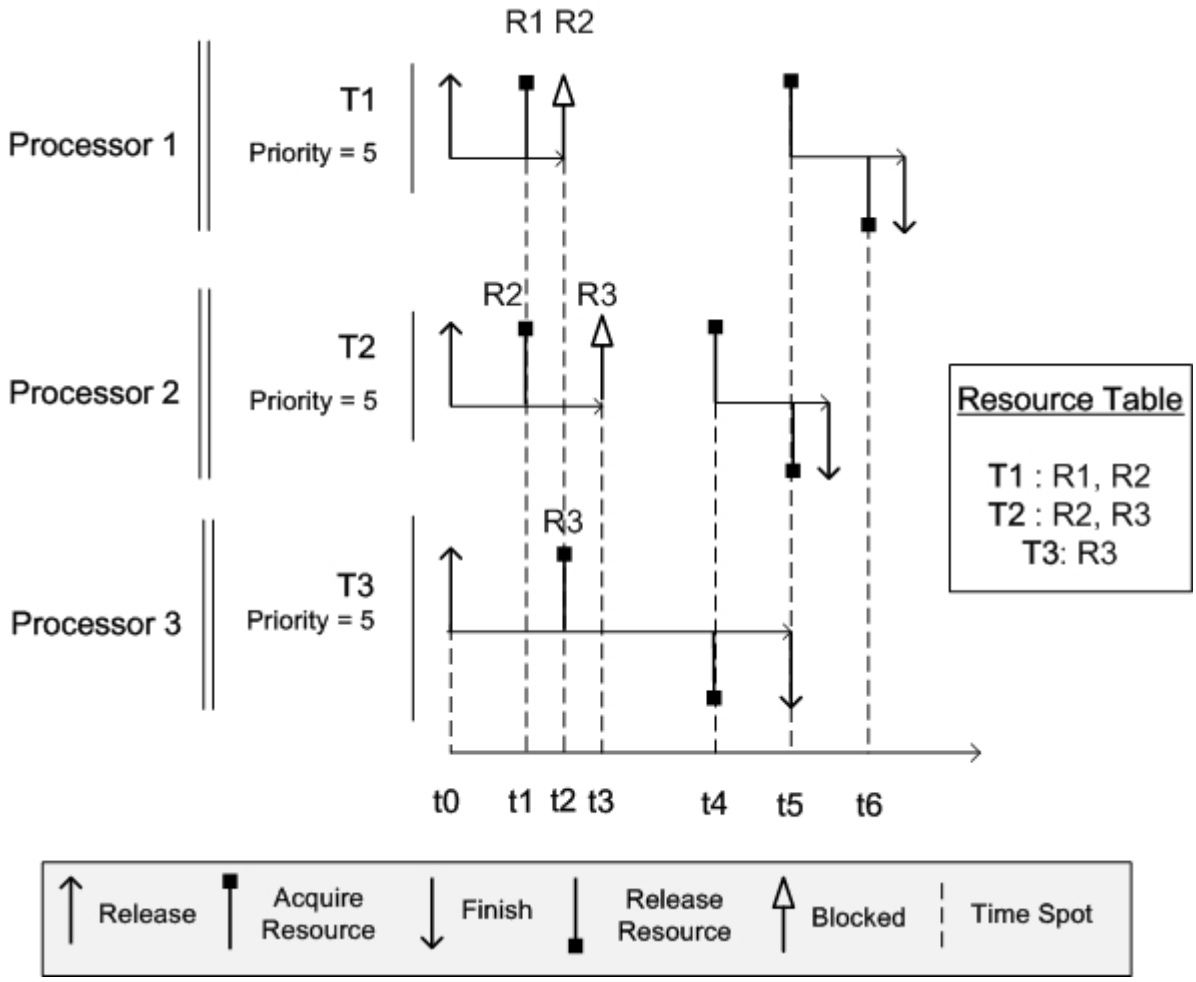

Figure 1.4: Transitive Blocking

required by the former. Although it has a higher priority, since the resource is non-preemptive, the task needs to wait for the low priority task to finish the resource before it can use it.

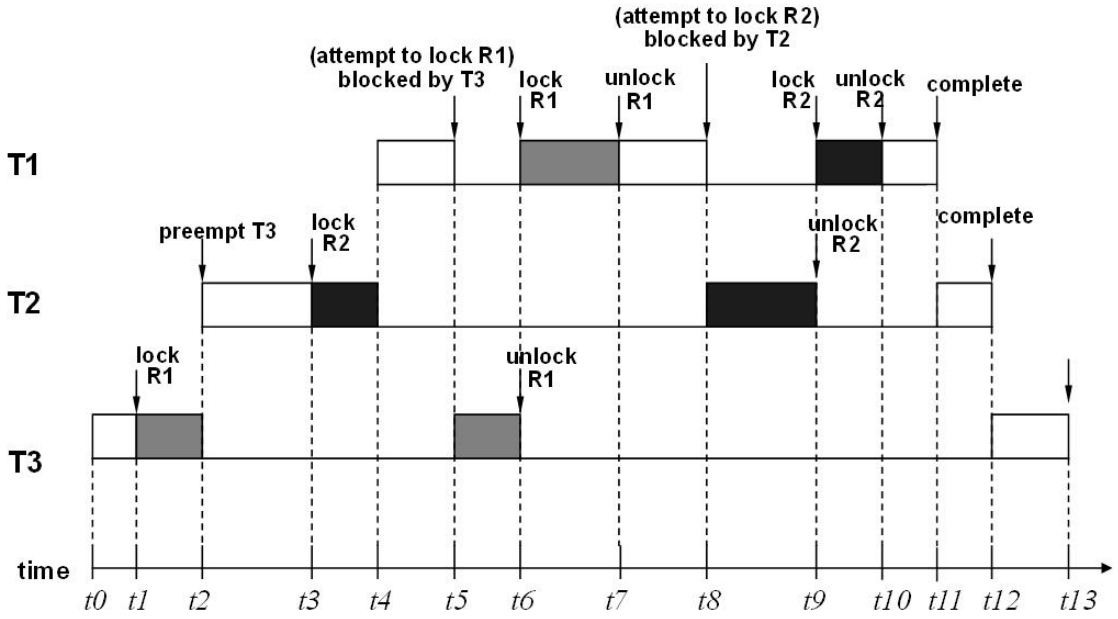

Figure 1.5: Priority Inheritance Blocking Chain

The priority inheritance protocol [58] was hence developed so that the low priority task in this case inherits the priority of the blocked high priority task in order to minimize blocking time. However, there is still a potentially long blocking chain where a high priority task may be blocked by low priority tasks for multiple times. For example, as shown in Figure 1.5, we have three tasks T1, T2 and T3 running in a single processor. They are assigned priority 2, 1 and 0 respectively. Let us assume that T3 requires resource R_1, and T2 requires resource R_2 and the highest priority task T1 requires both resource R_1 and R_2. At the beginning, T3 is released and locks resource R_1. While T3 is operating in the critical section of R_1, T2 is released. Because its priority is higher than T3, T2 preempts T3 and locks the free resource R_2. While T2 is holding R_2, T1 is released and starts to execute as the highest priority task in the system. It attempts to lock R_1 and finds that R_1 is held by T3. Therefore, T3 is resumed and inherits the priority of T1. At this point, T1 can only use R_1 after R_1 has been released by T3. In the

same manner, T1 attempts R2 later at t8 and T2 is resumed. In this scenario, we can see that T1 suffers blocking from T2 and T3 which forms a blocking chain potentially producing long blocking delay.

Also, [49] states that priority inheritance may produce deadlock. This can be easily illustrated by simplifying the above scenario. Let us assume there are only two tasks T1 and T2 sharing two resources R_1 and R_2 in a single processor system. T2 has a higher priority than T1. T1 starts its execution at time t0. It then locks R_2 at t1. T2 is released and preempts T1 (due to its higher priority) at t2. It then locks R_1 at t3. At t4, T2 attempts to lock R_2, but is blocked because T1 is holding it. At t5, according to the priority inheritance rule, T1 resumes execution by inheriting the priority of T2. Shortly after this, T1 attempts to lock R_1 but becomes blocked again due to the resource being held by T2. Both tasks at this moment cannot proceed and become deadlocked.

1.5 Thesis Hypothesis

The performance of a multiprocessor resource sharing algorithm can be affected by many factors. The environment, the choice of scheduling algorithm, the system architecture, the semantics of the application and the algorithm itself can all have an impact on the performance of the algorithm. The design of an optimal algorithm may not be available. This is because the performance of the resource sharing algorithms in multiprocessors varies in different operating scenarios. Other factors, including the locking primitives, the processor architecture and the underlying scheduling algorithms, are all having influencing impact. Therefore, the aim of this thesis is to investigate these related factors and propose a flexible resource sharing framework for multiprocessors where programmers can design and implement the best suited algorithm based on the operating application scenarios.

Sharing resources on multiprocessor platforms is inherently difficult. It involves using an appropriate scheduling algorithm, implementing the most suitable locking primitive, and utilising an appropriate locking algorithm.

The new algorithms are still under development to agree a best approach to estimate the execution time of tasks with shared resources in multiprocessors. [19] indicated that the conventional approach of modeling a global fixed priority scheduling queue inflates the worst-case execution time of every task to account

for the longest time that could be spent on spinning by that task. Such estimation sometimes appears to be too pessimistic. The simple queuing structure estimation accounts for too much blocking so that more tasks can interfere with the task under analysis at low priority levels. This inflation significantly limits the resource sharing performance of global scheduling algorithms that granted fewer tasks for execution.

David and Burns [20] states that an optimal scheduling algorithm should schedule all of the task sets that comply with the task model, and all deadlines of all possible sequences of jobs should be satisfied. Neither partitioned scheduling nor global scheduling is optimal as they both perform differently in different scenarios. There is no single algorithm that dominates all other algorithms in all scenarios. The multiprocessor resource sharing algorithms, which rely on the scheduling algorithms, are largely affect by the choice of scheduling algorithm. The operating scenario may have different preferences in scheduling algorithm choices, locking primitives, queuing policies and implementation specifications. Therefore, the optimality of the resource sharing algorithms is largely dependent on its operating scenario. This is demonstrated in detail in the following sections and chapters.

Consequently, the research hypothesis is:

The performance of a multiprocessor resource sharing protocol is largely dependent on the application semantics. This thesis contends that it is, therefore, inappropriate to introduce support for a particular multiprocessor resource sharing protocol into a language definition. Instead, a language should support a framework that allows a variety of protocols to be implemented (either by the programmer or via prewritten standard libraries). A flexible framework can be applied to a wide range of multiprocessor resource sharing protocols with small overheads.

1.6 Thesis Outline

The remainder of the thesis is organized in the following chapters:

Chapter 2 Explains the related theories and technologies associated with multiprocessor resource sharing.

Chapter 3 Presents the abstract structure of the multiprocessor flexible resource sharing framework

Chapter 4 Demonstrates the details of the proposed framework implementation in Ada.

Chapter 5 Evaluate the applicability of the framework to various protocols.

Chapter 6 Presents the conclusions and future work

Chapter 2

Literature Review

In order to meet the goal of this thesis, providing a multiprocessor flexible resource sharing framework for programming languages, it is necessary to discuss the underlying principles and theories before the actual introduction of the framework. These concepts are vital to the design and performance of the contribution. The following section explains the terminologies which are commonly used by later sections.

2.1 Single Processor Resource Sharing Algorithms

The blocking phenomenon introduces some critical factors to be considered in resource sharing algorithm design. The scheduling policy, priority assignment and resource accessing priority are all important and critical to the effectiveness and performance of resource sharing methods. The synchronization between tasks is adding an extra blocking time worsening the response time of the high priority tasks. Tasks under scheduling algorithms with shared resources are supposed to be schedulable where the worst case response time is less or equal to their deadlines. The highest priority real-time task should suffer the minimum blocking as the highest priority task, with conventional rate monotonic assignment, is more likely to least laxity to its deadline which means only minimum delay is allowed. Various resource sharing algorithms were designed with the above factors considered.

2.1.1 Priority Ceiling Protocol

In uniprocessor resource sharing schemes, the Priority Ceiling Protocol (PCP) [58] is a well known method for fixed priority scheduling systems. It imposes an extra condition from the priority inheritance protocol where a task can only acquire a new resource if its priority is higher than all priority ceilings of all the resources allocated to tasks other than itself. The priority ceiling is defined to be the highest priority of any task that may use the resource. In other words, a resource request from task T is denied if the resource is already allocated to another task or the priority of T is not higher than all priority ceilings for resources allocated to tasks other than T at the time.

There are two varieties of PCP called the Immediate and the Original Priority Ceiling Protocol (IPCP and OPCP). Both approaches have the same worst-case performance [58]. The main difference between these two approaches is the priority assigned to the acquiring task. The OPCP demands the resource acquiring task to be blocked if its active priority is not higher than the current system ceiling. The dynamic priority of the tasks under OPCP is the higher of its own priority and any priority it inherits from its blocked high priority tasks. The system ceiling is set to the priority of the resource having the highest ceiling among those locked by other tasks currently. Under IPCP, the task's priority is immediately raised to the ceiling of the resource being acquired instead of being raised at the time when the high priority task is blocked. This will block other tasks with the same or lower priority from executing whether acquiring resources or not. The main advantage of IPCP is that it reduces the number of context switches produced by OPCP.

Let us recall the example with IPCP from Figure 1.5. After T3 locks resource R1, its priority is raised to 2 as this is the ceiling of the resource. The ceiling of R1 is 2 because both T1 and T3 requires R1. But T1 is the highest priority task in the system which assigns the ceiling of R1 to 2. T2 later on cannot execute because its priority is lower than T3 which is running at the ceiling of the resource. Therefore, T1 only suffers blocking from T3 instead of both T2 and T3. The priority ceiling protocol improves the response time of high priority tasks with bounded maximum blocking delay by preventing transitive blocking. Since the priority of low priority task will be raised to the ceiling of the resource, the high priority task only needs to wait for one critical section of the underlying low priority task. The worst case

of this waiting time will take place in the situation where the underlying task has the longest critical section among all other tasks. Owing to this feature, after the highest priority task in the chain has the resource, the blocking chain is resolved gradually from the highest priority task to the lowest. In this way, the deadlocks are also prevented.

2.1.2 Stack-Based Resource Sharing Protocol

The other main uniprocessor resource sharing algorithm is the Stack-Based Resource Sharing Protocol (SRP) [4] for EDF scheduling and Rate Monotonic scheduling. The SRP is similar to IPCP in that it also blocks the tasks from execution rather than from accessing the resource. It is called stack based because the late coming task at the top of the stack with higher preemption level will preempt the tasks at the lower levels of the stack. This imposes a rule that only the task at the top of the stack can execute without suspension. The SRP manages the execution of the tasks according to their preemption levels. The preemption levels proposed, distinguished from priorities, is to enable the protocol to predict blocking, in the presence of dynamic scheduling schemes like EDF. The preemption levels are statically assigned to tasks inversely to their relative deadlines. The earliest deadline task is assigned with the highest preemption level. The essence of the preemption level imposes a constraint on the task execution that the lower preemption level task cannot preempt the tasks with higher preemption level. The preemption levels were designed and extended by SRP to restrict lower or equal preemption level tasks from execution. Once the execution begins, a task cannot be blocked from accessing shared resources. If a low preemption level resource holding task is preempted, the top executing task is consequently blocked where a deadlock will emerge. The execution eligibility rule for SRP is more strict than PCP where a task can only start execution if its preemption level is higher than the highest ceiling of all locked resources. With the example from above, T2 in this case will not start execution at t2 because its preemption level is not higher than the ceiling of resource R1 at this moment. When T3 finishes R1, T1 and T2 are eligible for execution. However, at this point, T1 has an earlier absolute deadline than T2. Therefore, T1 starts execution granted with resource R1. Later on, T2 can start execution if R1 is released by T1.

2.2 Multiprocessor Resource Sharing Algorithms

2.2.1 Motivation and Challenges

With the advent of multiprocessor resource sharing, [13] [3] compared different protocols and concluded the following:

1. Non blocking methods are generally preferred for small simple objects. Wait-free implementations are generally preferable for large or complex objects.
2. Wait free algorithms are preferable to lock-free algorithms.
3. Suspension based locking should be avoided under partitioned scheduling for global resources.
4. If a system spends at least 20% of its time in critical sections, the use of suspension based locking will not lead to better schedulability than spinning based locking.

These conclusions agree with [20] that different locking primitives have various performance under different scenarios. The small and efficient locks, such as spin locks are effective for small simple objects. This is because, if suspension based locking is used, most of the time spent in the critical section will be consumed by the locking overheads. Suspension based locks are not efficient for small and simple objects. However, if long and complex objects are shared between tasks, such overhead is relatively small compared to their whole execution length. The flexibility, where schedulers can inherently change the priority or the queuing order of the tasks, gained from deploying suspension based locks, seems to be worthwhile. This suggests that the interaction with the scheduler is complex and expensive. The suspension based locking can interact the schedulers to reduce the blocking time suffered by the highest priority task to a minimum. For example, the MPCP imposes an important property that the highest priority task only suffers at most one longest critical section of any lower priority task. When the blocking time saved becomes significant to the overhead spent, the suspension based lock becomes more efficient.

Therefore, based on the motivation that the best suitable resource sharing algorithm in multiprocessor is the one most suitable for a particular scenario, this thesis

proposed a flexible resource sharing framework to allow the application developers to integrate their own best resource sharing protocol with their applications at compile time. The framework will shield the developers from the hazard of interacting with various underlying primitives and structures.

There are many inspirations for multiprocessor resource sharing algorithm developments. Extending existing uniprocessor algorithms is a promising method as it can transform well understood knowledge from uniprocessors into multiprocessors. Multiprocessor resource sharing suffers additional impact from other factors than the uniprocessor algorithms. The extension of the underlying hardware introduces extra factors to be considered in resource sharing. In uniprocessor systems, tasks are dispatched to the same processor where all tasks are ordered by the single run queue. The execution and resource accessing eligibility can then be easily determined by comparing the priority of those tasks in the queue. This unified scheduling scheme is lost in some multiprocessor systems. As mentioned in Chapter 1, partitioned multiprocessor systems can have multiple run queues managing different tasks. The resource sharing, in which case, is becoming difficult because the priorities across different run queues are not comparable to each other. Also, the multiprocessor architecture possesses unique factor to be considered. The remote blocking is the effect that never existed in uniprocessors. The remote blocking is a significant issue to multiprocessor resource sharing because indefinite blocking may happen and worsen the response time of the highest priority task. The following algorithms describes the problem of multiprocessor resource sharing from different perspectives.

2.2.2 MPCP

With the disciplines from PCP in the uniprocessor world, [49] proposed the Multiprocessor Priority Ceiling Protocol (MPCP) for partitioned scheduling systems bringing resource sharing towards multiprocessor platforms. Since tasks are executing in parallel to each other, the resource sharing protocols in the multiprocessor world needs to be generalized to consider and minimize remote blocking.

The MPCP solves this problem by introducing a synchronization processor and the idea of a globalized critical section. It is proposed to isolate the blocking caused by local and global resources. A processor responsible for executing

a global critical section (global resource), which is the resource shared between tasks across different processors, is called the synchronization processor. All other processors are called application processors. In the event of attempting to access a global resource, the acquiring task can then be considered to be migrated to the synchronization processor. As for the local resource sharing, PCP is imposed and tasks remain in their original processor for execution.

As a variation of PCP, the priority ceiling is applied to global resources. In MPCP, the ceiling of a local resource is defined to be the highest priority task that will ever use the resource. The ceiling of global resource is defined by the following two rules:

1. The ceiling of a global resource must be higher than the highest base priority task in the system.

2. The ceiling of a global resource is the sum of the highest priority task accessing the resource and the base priority of the global resources.

Under these rules, the priority ceilings of global resources are higher than the ceilings of local resources in the system and maintained in a priority order. In other words, tasks in global resources are always executed in preference to local resources. High priority tasks in global resources are executed in preference to low priority tasks in global resources. For example, A task T with base priority 4 accesses a global resource. Let us assume the base priority for the global resource is 5. The active priority of task T, when executing in the global shared resources, will be 9 (4+5). It will have a higher execution eligibility any task with priority lower than 9.

The MPCP is then defined by the following rules:

- Each application processor runs the normal PCP with resources shared locally.

- If a task requests a global resource, it will be migrated to the synchronization processor guarded by a global ceiling.

Figure 2.1 explains in detail how MPCP works with an example. The example was specifically set for demonstrating how global resources are shared in a mixed environment of local resource. In this example, we can see how the global

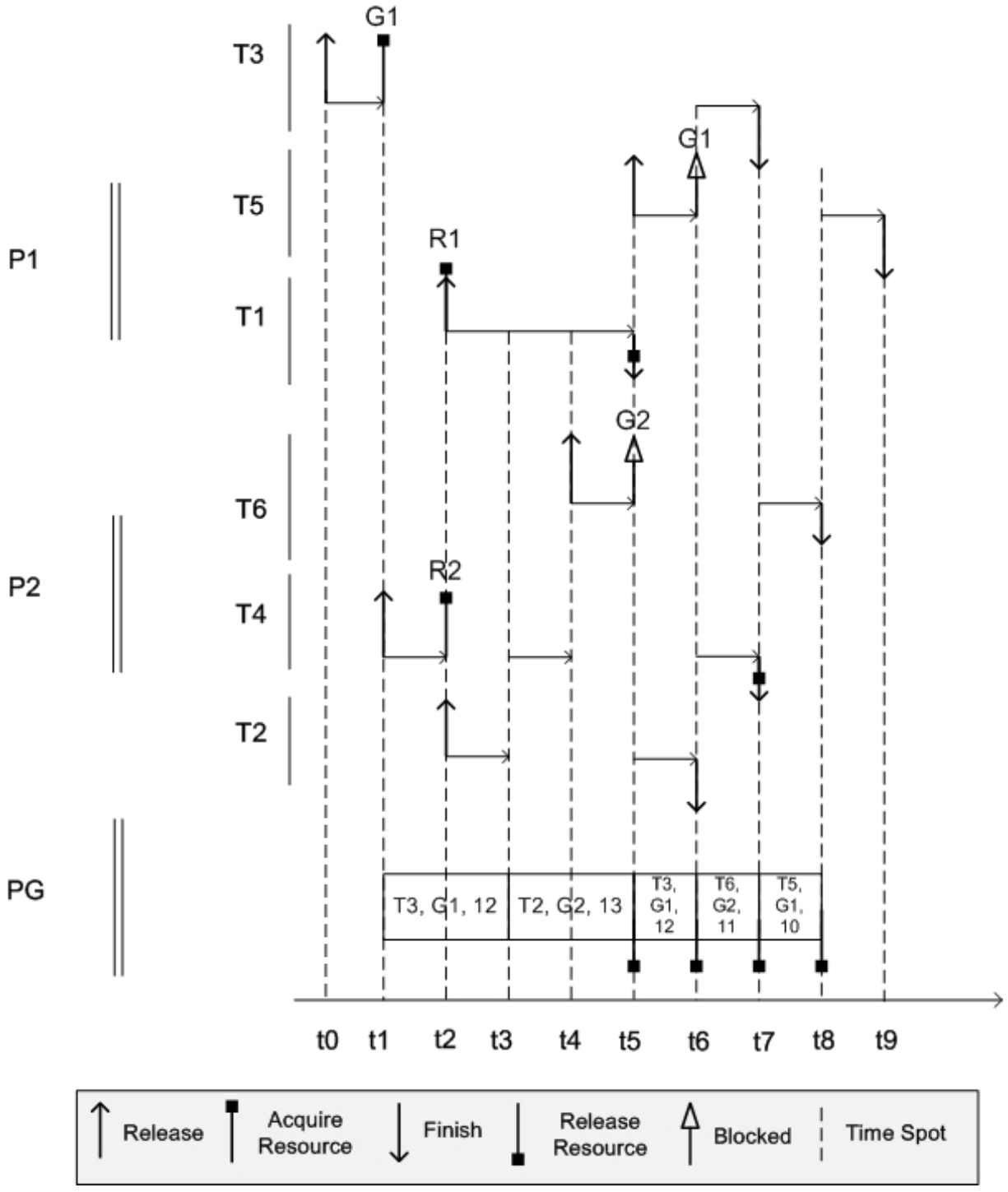

Figure 2.1: MPCP Illustration Example

synchronization processor minimizes the blocking suffered by the highest priority task. As shown in the figure, we have three processors P1, P2 and PG. PG is the synchronization processor. Task T1, T3, T5 are assigned to P1. The other tasks are assigned to P2. R_1 and R_2 are shared with other tasks which are not relevant to this scenario. Since the highest base priority is 6, the base priority of the global resources is 7. Table 2.1 illustrates the resource required by each task:

Task	**Base Priority**	**Requiring Resource**	**Affinity**
T1	1	R1	P1
T2	6	G2	P2
T3	5	G1	P1
T4	2	R2	P2
T5	3	G1	P1
T6	4	G2	P2

Table 2.1: MPCP Resource Table 1

Resource	**Ceiling**
R1	1
R2	2
G1	12
G2	13

Table 2.2: MPCP Resource Table 2

At t0, T3 is released at priority 5 on P1. It immediately start execution.

At t1, T4 becomes runnable on P2 at priority 2. It acquires R2 sooner after its execution has been started. During this time, T3 migrates to the synchronization processor PG and attempts to lock G1.

At t2, T2 is released to P2 at priority 6. Since it has a higher priority, its execution is started immediately. On P1, T1 starts execution and acquires R1 at priority 1.

At t3, T2, which started execution on t2, migrates to the synchronization processor PG and attempts to acquire global resource G2. It locks the resource with its active priority increased to the ceiling of the resource. With higher active priority (6+7=13) than the current executing task, T2 immediately preempts T3

and becomes runnable from t3. Since T2 has already been migrated to PG, T4 is resumed for execution on P2.

At t4, T6 is released to P2 at priority 4. T4 is preempted in this case because it has lower priority than T6.

At t5, T6 migrates to PG and attempts to lock G2. At the same time, T2 has finished its execution in G2. It releases the global resource and migrated back to P2. As the highest priority task on P2, it continues its execution. T4 is still suspended. The next highest priority task on PG is T3 with active priority 12. It is resumed for execution due to the fact that T2 has already left the processor.

At t6, T2 stops its execution on P2. It yields the processor to T4 which can continue its execution in R2. On PG, T3 has finished its execution in G1 and migrates back to P1. T6, as the next highest priority task, locks G2 and starts execution at priority 11.

At t7, the execution of T3 is finished and T3 stops on P1. Similarly, T4 stops execution on P2. T6 has finished its execution in G2 and migrates back to P2. As the only task runnable in P2, it is resumed for execution immediately.

Similarly, T5 has finished its critical section in G1 on PG and migrates to P1 to finish its remaining execution on P1.

The MPCP produces minimized blocking behavior similar to the PCP in the uniprocessor world. The paper indicated that an outermost global resource request can be blocked for the duration of at most one global critical section of all lower priority tasks on a synchronization processor. This can be illustrated by simplifying the above example. Let us assume there are only 3 tasks in the system which are T3, T4 and T2. All tasks wish to lock the same resource R1. The ceiling of the resource is then set to 13. If the tasks are released for execution in the same order as depicted by Figure 2.1, T4 will be migrated PG acquiring G1 once it has started execution at t1. It finds that G1 is not available and thus becomes suspended. T2 migrates to PG and attempts to lock G1 once it has been released for execution at t2. However, due to the fact that T3 is holding the resource, T2 is blocked. At t3, T2 will be granted for the access of the resource instead of T4 because it has the highest ceiling priority of the resource. T4, which is also waiting on locking G1, will be blocked again. No other tasks with lower priority than T2 will be granted the access to G1 at t3. This is an important property of MPCP that the blocking suffered by the highest priority task is bounded.

There are other versions of the Multiprocessor Priority Ceiling Protocol (MPCP). Initially, [49] proposed the synchronization processor approach. This was then generalized in the same paper to allow multiple synchronization processors, but again each global resource was assigned to one synchronization processor. A task that wishes to access a global resource migrates to the synchronization processor for the duration of its access. Each global resource is also assigned a ceiling equal to $P_H + 1 + \max_i\{\rho_i | T_i \ uses \ R^k\}$, where P_H is the highest priority of all tasks that are bound to that processor, τ_i is the priority of task T and R^k is a resource. The generalized protocol is defined below:

1. Tasks are partitioned to processors. PCP is used for local resource sharing.

2. When a task T_i is blocked by a global resource, it is added to a prioritized queue and is suspended. The resource holding task will continue its execution at the inherited highest priority of the tasks being blocked on that resource (if higher than its own priority).

3. If task T_i locks a free global resource, it will execute at its current active priority.

4. When task T_i leaves the resource, the next highest priority task in the queue (if any) will be granted access to the resource.

For the simplicity of implementation, [49] suggest that the priority when accessing a global resource can be raised to the ceiling immediately. We also note that with the basic MPCP there was a single synchronization processor and nested resource accesses were allowed. This is because, after migrating to the synchronization processor, the nested resource request can assigned non-decreasing priority. In which case, the inner nested resource request will have the highest execution eligibility where the deadlock is avoided. With the generalized MPCP, the execution priority of the nested resource request is only raised to the highest local priority level which is not high enough to avoid deadlock. This is the situation where two highest priority tasks may require the resource being hold by each other where the resource requests are essentially deadlocked. This is perhaps too strong a constraint. What is really required is that the group of resources involved in a nested chain is assigned to the same synchronization processor.

Later, [49, 48] renamed the MPCP as the Distributed Priority Ceiling Protocol (DPCP) and clarified the remote access mechanism. In order to access a global resource, a task must acquire a local agent first. A local agent is a task on the remote processor where the global resource is being held. Any remote request for the global resource must be accomplished through the agent. When a global request is granted through the agent, the agent executes at an effective priority higher than any of the normal tasks on that processor. Hence, the protocol was targeted at distributed shared memory systems. For globally shared memory systems, the need for remote agents is removed and all global resources can be accessed from all processors. This protocol is now generally referred to as the MPCP.

It is worth restating the important properties of MPCP here as the summary. MPCP deploys priority ceiling protocol on both normal and synchronization processors. The priority ceiling protocol avoids deadlocks on uniprocessor systems. As nested resource locking is prohibited, all locking transactions and requests are managed by the PCP protocol running on individual processors. It is therefore clear that the MPCP avoids deadlocks on multiprocessors. MPCP demands the ceiling of the global resource is absolutely higher than the highest priority of the task in the system. The blocking time for the highest priority task accessing the global shared resource is bounded for at most one longest critical section for any lower priority task. All other tasks waiting for accessing shared resources are ordered in priority queues.

2.2.3 MSRP

The Stack Resource Policy, proposed by [4], emphasizes that:"a job is not allowed to start executing until its priority is the highest among the active jobs and its preemption level is greater than the system ceiling." If allowed to lock any resource, the current executing job must be running at the ceiling of the resource. Once a job starts, all its resources will be available. With this approach, all the tasks in the system can use the same run-time stack (hence the name of the policy).

In order to preserve this property, [26] proposed the Multiprocessor Stack Resource Policy (MSRP) for partitioned EDF scheduling. Resources are divided into two groups: local and global. Local resources are only accessed by tasks that execute on the same processor. Global resources are those that can be accessed by

tasks running on different processors. Unlike SRP, global resources have different ceilings on each processor. The ceiling indicates the highest priority of the sharing tasks on that processor. Furthermore, each processor has its own system ceiling which is equal to the highest preemption level task dispatched to that processor. On processor k, tasks are only allowed to execute global resources at the processor ceiling priority, which is the highest preemption level of all the tasks on processor k.

The MSRP protocol is thus defined as:

Rule 1 For each individual processor, SRP is used for sharing local resources. Every tasks has its preemption level, and ceilings are given to local resources.

Rule 2 Nested resource access is allowed for local resources, and local resources can use a global resources nestedly. However, nested global resources are prohibited - in order to prevent deadlocks.

Rule 3 The systems ceiling on each processor is defined to be greater than or equal to the maximum preemption level of the tasks allocated on that processor.

Rule 4 When a task requires a global resource R from processor k, it sets k's system ceiling to resource R's ceiling for that processor. If the resource is not available, the tasks busy waits in a FCFS queue.

Rule 5 When a global resource is released by task T_i executing on processor k, the tasks at the head of the FCFS queue (if any) will be granted access to the resources. The system ceiling for k is restored to its previous value.

The following example demonstrates the MSRP protocol in detail:

Task	**Affinity**	**Requiring Resource**	**Preemption Level**
T1	P1	nil	3
T2	P1	R1(G1)	2
T3	P1	R1 (G1)	1
T4	P2	G1	1
T5	P2	nil	2

Table 2.3: MSRP Resource Table

The preemption level, as described in SRP section, is used to track the locking behaviour of tasks at run time and is assigned inversely to the relative deadlines of tasks. The earliest deadline task is assigned with highest preemption level. Consider a system consisting of two processors and five tasks sharing two resources together. Table 2.3 shows the details of the task assignment and resource requirements. The ceiling of R1 is 2. The system ceiling of processor P2 is 2 and processor P1 is 3. When accessing G1 on P1, the priority of the tasks accessing G1 will be raising to 3 (the system ceiling) instead of 2 (the resource ceiling) in order to become non-preemptive.

The scheduling detail is shown in Figure 2.2. At time t0, T3 is released on processor 1. It immediately locks R1 on t0 and G1 on t3. When T2 is released on processor 1 at time t2, it finds that its preemption level is not higher than current system ceiling and is therefore blocked. Similarly, T1 is blocked immediately when released to processor 1 at time t4 as its preemption level is not higher than the system ceiling. This is because, when G1 is locked by T3, the system ceiling is raised to 3 which is the maximum preemption level of the tasks on processor 1. When T4 is released on processor 2, it attempts to lock G1 at time t4. However, G1 is being held by T3 on processor 1. T4 therefore joins the global FCFS queue waiting for G1 to be released. Since G1 has ceiling 2 on processor 2, T5 is blocked immediately after its release since its preemption level is not higher than the system ceiling at time t4. At time t5, G1 is released by T3. T1, with preemption level 3, can proceed with its normal execution on processor 1. On processor 2, T3 is granted the access to G1 as it is the head of the FCFS queue. When G1 is released on processor 2, T4 is suspended and yield the processor to T5. It can only proceed with its execution starting from time t8 when the system ceiling is falls back to the lower level. On processor 1, T3 releases R1 at time t7. T2, with preemption level 2, is granted with the processor to execute. It then locks local resource R1 at time t8 and finishes execution at time t10. At this time, T3 can use the processor to finish its remaining non critical execution.

MSRP shares global resources across processors in an FCFS manner. In order to acquire a global resource, a task must be running at the processor ceiling which makes it non-preemptive. Of course, a task on another processor will be able to acquire another global resource. Hence, it is possible that a task, once executing, will find some of its global resources already allocated. In order to minimize the

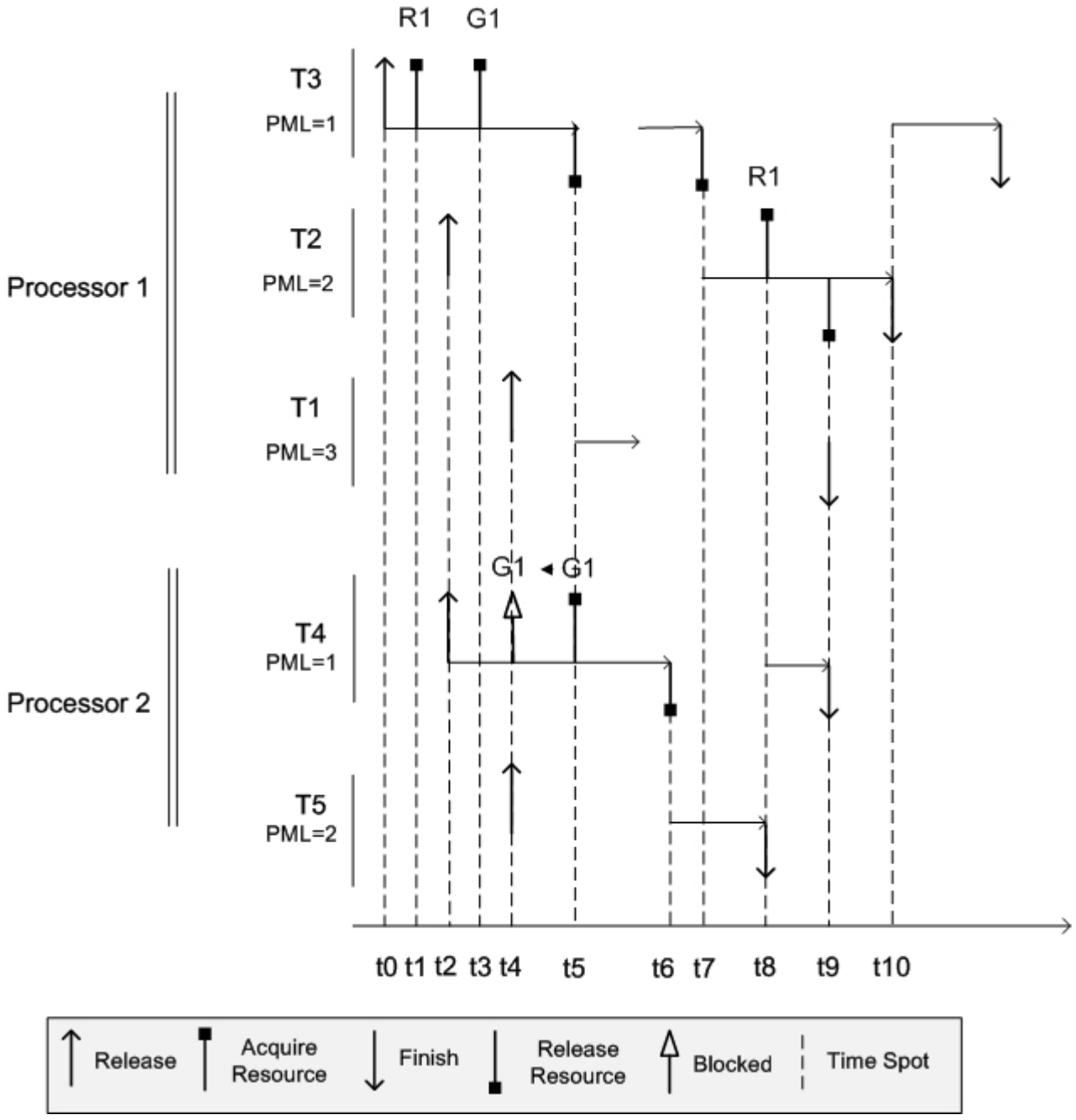

Figure 2.2: MSRP Example

blocking time suffered by the highest priority task, it is, therefore, necessary to non-preemptively busy-wait on the resource. The alternative approach would be to not allow the other tasks to execute (that is, implement a global system ceiling). This could result in processors potentially being kept idle.

In MPCP, the highest priority task can be suspended by having unavailable resources and resumed back for execution once the current resource holding task is finished. This results in a context switch on each resource access on the suspension based locking algorithms like MPCP. MSRP has an advantage of using the system ceilings of preemption levels so that the context switches is reduced to one [50]. However, wasting the processor cycles by spinning on unavailable resources is certainly the associated expense.

It is also worthwhile restating the property of the MSRP. All tasks on different processors are accessing global resources non-preemptively. If the resource is not available, the acquiring tasks are set to spinning in FIFO queue. Apart from the FIFO spinning time, the blocking time for any high priority task is bounded by MSRP to at most one critical section of any tasks with lower preemption level.

2.2.4 Flexible Multiprocessor Locking Protocol

The Flexible Multiprocessor Locking Protocol [9] proposed new ways to share resources in multiprocessor systems. The FMLP became the first resource sharing protocol to support nested resource, which can be applied to both global and partitioned scheduling.

With the inspiration of achieving a high degree of parallelism, the FMLP maintains a balance between busy-waiting from spin locks and blocks from suspensions locks with the following rules:

1. Resource Division: Depending on the duration they can be held, the resources are divided into long and short resources. Conventional suspension locks are used for guarding long resources. Spin locks are used for sharing short resources.

2. Minimize Short Busy-Waiting: As a feature of this protocol, the time of a task spent on waiting for a short resource should be minimized. The protocol ensures this by constraining the short resource requests. A short resource

request is non-preemptive. Also, long resource requests cannot be contained in short resource request so that the length of execution in short resources can be minimized.

3. Resource Grouping: Similar resources are grouped together in order to be dealt with efficiently. Each group contains only either long or short resources. The resources, required by nested requests, are grouped into the same group. The resource type of the group is determined by its outermost request. Therefore, a long resource group may contain requests for short resources. However, a short resource group may not contain requests for long resources.

After the resources have been grouped, the tasks must acquire the group lock first before locking the actual resource. For example, a short resource request S from a task T must require S's group lock first. Since any possible nested resource requests from T at this stage are short, T remains in a non-preemptable spinning state until the resource lock has been released. As for long resource requests, priority inheritance is used. Similar to the short resource requests, a task T must also acquire a long resource L's group lock first. If T blocks a higher priority task requesting a resource in the same group, it will inherit its priority and finish L as soon as possible. If another long resource is required inside L by T, the request will be granted immediately if the requested resource is in the same group. However, if the acquiring nested resource is short, the above mentioned short resource sharing protocol will be used.

The following Figure 2.3 illustrates the FMLP in a detailed example using partitioned scheduling.

In Figure 2.3, we have 4 tasks running on two processors P1 and P2. The tasks T1, T2, T3 and T4 have priority assigned 3, 2, 1 and 4 respectively. The resources shared between the tasks are listed in the following table:

Task	**Priority**	**Requiring Resource**
T1	3	S1
T2	2	S1 (S2), L1
T3	1	S2
T4	4	L1

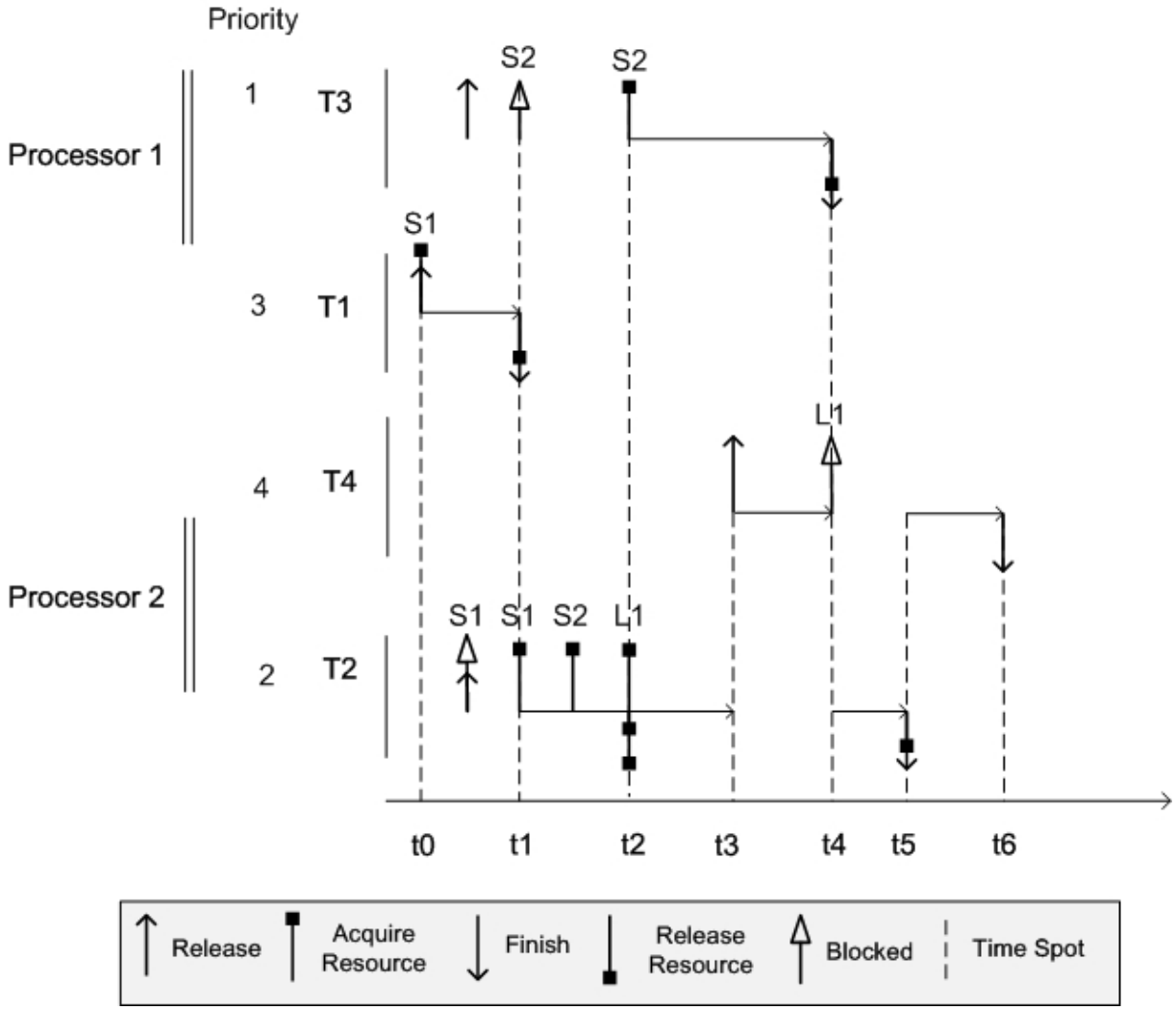

Figure 2.3: FMLP Example

S1 and S2 are two short resources grouped in one short resource group with group lock G1 because T2 acquires nested access to S1 and S2. L1 is a long resource and is therefore grouped individually in a long group with group lock G2.

In Figure 2.3, T1 is released to P1 at t0. It then successfully obtains G1 and locks S1.

Between t0 and t1, T2 is released to P2 and requires S1 immediately. However, it finds that G1 has already been granted to T1. Since T1 is in a non-preemptive state, T2 is therefore set spinning. T3 is released to P1 later on before t1. It cannot start execution since T1 is holding the processor P1 at this moment.

At t1, T1 finishes execution in S1 and releases G1. T2 is then granted with G1 and locks S1. At this point, T3 is granted with the processor P1 for execution and requires S2. However, since G1 has been granted to T2, T3 is blocked.

At t2, T2 finishes execution in S1, S2 and releases G1. It then locks the long resource group lock G2 and locks the free resource L1. Since G1 is free now, T3 can then obtains G1 and locks down S2.

At t3, T4 is released to P2. Since its priority is higher than T2, it starts execution immediately.

At t4, it attempts to lock L1 and finds G2 has already been locked by T2. T2 at this point is therefore resumed and inherits the priority from T4 running at priority 4. At this moment, T3 finishes execution in S2 and released both the lock for the resource and the group.

At t5, T2 finishes execution in L1 and releases the group lock G2 leaving L1 to T4.

In this scenario, since the resources are guarded by group locks, T3 is blocked by T1 although it requires a different short resource. Also, it is worthwhile noticing that T2 gets the short resource group lock G1 ahead of T3 because the spin lock queue is operated in FIFO manner.

The FMLP protocol promoted the classification of long and short resources depending on the critical section length of the shared resources. The long resources, whose scheduling expenses are relatively small to its total execution time, is granted to the highest priority task using mutex based lock. The short resources, with less execution time to spare for heavy scheduling expenses, is granted to the first waiting task in the FIFO spinning queue. This classification reduces the scheduling overheads and simplifies the schedulability analysis for multiproces-

sor resource sharing. The FMLP is significant also in supporting nested resource sharing. short resource requests are allowed to be nested within long resource requests. Restriction on scheduling is also lifted where both partitioned and global scheduling are permitted.

2.2.5 Parallel-PCP

We have seen the problem discovered in the priority inheritance protocol where a high priority task gets blocked multiple times by lower priority tasks. The PCP was developed to prevent this multiple blocking. The low priority tasks in this case have to wait until the resources have been released by high priority tasks. This imposes a high risk to low priority tasks so that they may need to wait a long time since the blocking chain starting from high priority tasks may be very long. [21] pointed out that the response time of a task depends on three parameters: 1) the execution time of the task itself; 2) the amount of time that the task needs to wait before being granted with resources; 3) the amount of execution time from other tasks that have higher priority. The more resources high priority tasks require the more response time delay is added to low priority tasks.

New global fixed-priority preemptive multiprocessor scheduling algorithm called Parallel-PCP (P-PCP) improve the average performance of the foreseen resource sharing protocols by allowing certain number of low priority tasks to block high priority tasks [23]. That is the low priority tasks to lock resources in some situations which are not permitted by PCP. In PCP, a high priority task only needs to wait for one longest critical section used by one underlying lower priority task in the worst case. This is not always true in P-PCP. A task T with priority P is allowed to lock a shared resource if the total number of tasks with base priority less than P, which refers to the lower priority tasks, and active priority greater than I, which refers to a subset of those tasks which are running at a higher priority with potentials to block high priority tasks, is at most α_i. α_i is a threshold defined for each priority level of the system. According to the rule, a low priority 3's α_i is not higher than a high priority 4's α_i. α_i's value is assigned artificially. A low value for α_i implies that fewer tasks with base priority lower than α_i are simultaneously allowed to execute at an active priority higher than α_i. The PCP is an extreme instance to P-PCP where α_i is set to one for all priority levels which means at most

one lower priority task may be able to block the execution of a high priority task. Although the response time of high priority tasks is improved, the rule in PCP reduces the parallelism in the system. The protocol therefore is looking forward to manipulate an appropriate value for α_i in order to gain a trade off between both parallelism and efficient response time for high priority tasks.

The scheduling rules of P-PCP are summarized as follows:

Rule 1 The principle rule is : a job is allowed to lock a shared resource if the total number of tasks with base priority less than i and active priority greater than i is at most α_i. α_i is an implementation defined value that determines the maximum length of the blocking chain of the conventional priority inheritance protocol. By setting α_i to 1, PPCP works similarly to PCP. If set to the total number of tasks in the system, the protocol will have the same semantics as the Priority Inheritance Protocol (PIP).

Rule 2 If there are any unassigned processors and a task T_i does not require any resource, the free processor is assigned to T_i. Otherwise, continue to Rule 3.

Rule 3 If the current executing task T_i requires a resource and the resource is locked, PIP applies. If the resource is free, Rule 1 applies and resource is either assigned to T_i or the task is suspended.

In order to show the advantage of P-PCP, the blocking suffered by T5 in the following two examples is demonstrated and analyzed. The examples in Figure 2.4 and Figure 2.6 comprise 8 tasks(T1..T8) and 3 processors. The tasks are assigned with decrementing priority where T1 has the highest and T8 has the lowest. The resources shared between the tasks are explained by the following table:

Resource	**Shared between Tasks**
R1	T8, T3
R2	T7, T4
R3	T5, T6, T2

Table 2.4: P-PCP Resource Table

With only priority inheritance in place, as seen in Figure 2.4, any lower priority task can block high priority tasks as long as it holds the resource being required by the latter.

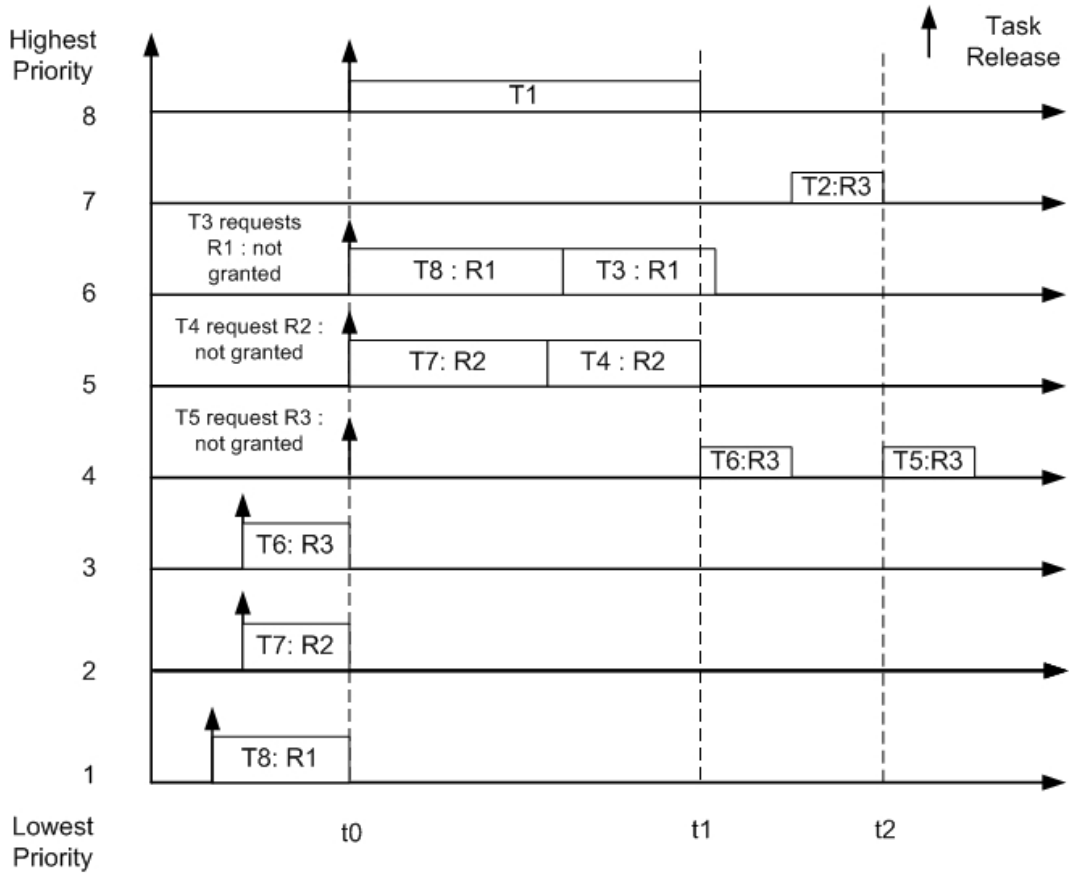

Figure 2.4: Scheduling Example of PIP

The system is globally scheduled and starts with T8 holding R1 on P1, T7 holding R2 on P2 and T6 holding R3 on P3.

At t0, T3 is released and finds that R1 has been locked by T8. Therefore, T8 inherits T3's priority and is raised to priority level 6. Similarly, T7 inherits T4's priority and is raised to priority level 5. The highest priority task T1 is released by preempting T6. At this point, only T1, T7 and T8 are executing.

Between t0 and t1, T7 and T8 finish R1 and R2. Theses two resources are then granted to T3 and T4 respectively.

At t1, as T4 finishes its execution making the processor idle, T6 holding R3 still, as a preempted task, resumes its execution. It is worthwhile noting that T5 cannot lock R3 here because R3 is still locked up by T6.

Between t1 and t2, T3 finishes R1. The processor is then yielded to a newly released task T2 which will lock R3 after it has been released by T6.

At t2, T5 is then finally eligible to lock R3.

Under pure priority inheritance, T5 suffers blocking from two lower base priority tasks (T7 and T8). The number of time units T5 suffers is analyzed in Figure 2.5

Figure 2.5 analyzes the response time of T5 in the scenario demonstrated by Figure 2.4. By assuming it is released at time 2, T5 in this situation suffers a delay from t2 to t9.

This is reduced in P-PCP by setting α to 3 for all priority levels greater than 4 and to 2 for priority levels less or equal to 4. The impact of Z is demonstrated by Figure 2.6

In Figure 2.6, we can see that the resource request to R2 from T7 at t0 is not granted. At each scheduling event, the scheduler will search through all tasks in the system seeking tasks with lower priority but higher effective priority holding shared resources for all priority levels. At this point, the scheduler finds that there are three tasks falling into this type for priority level 4. Since α_4 has been set to 2, which means at most 2 tasks with lower basic priority and higher effective priority than 5 are allowed to execute for priority level 5, and T8 has already released and locked R1, T7 is therefore suspended.

Since T8 has is already executing, T7's request is therefore denied. In this case, T8 is popped up to priority 2 for execution. T6 starts by holding R3.

Then at t1, T1 as the highest priority task in the system is released to the only

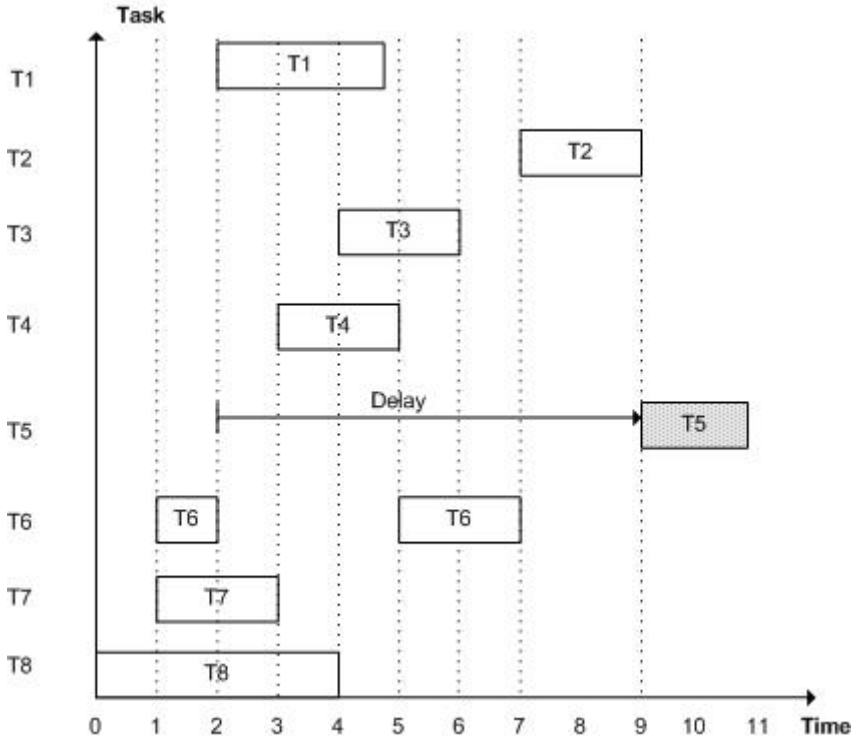

Figure 2.5: Response Time for T5 under PIP

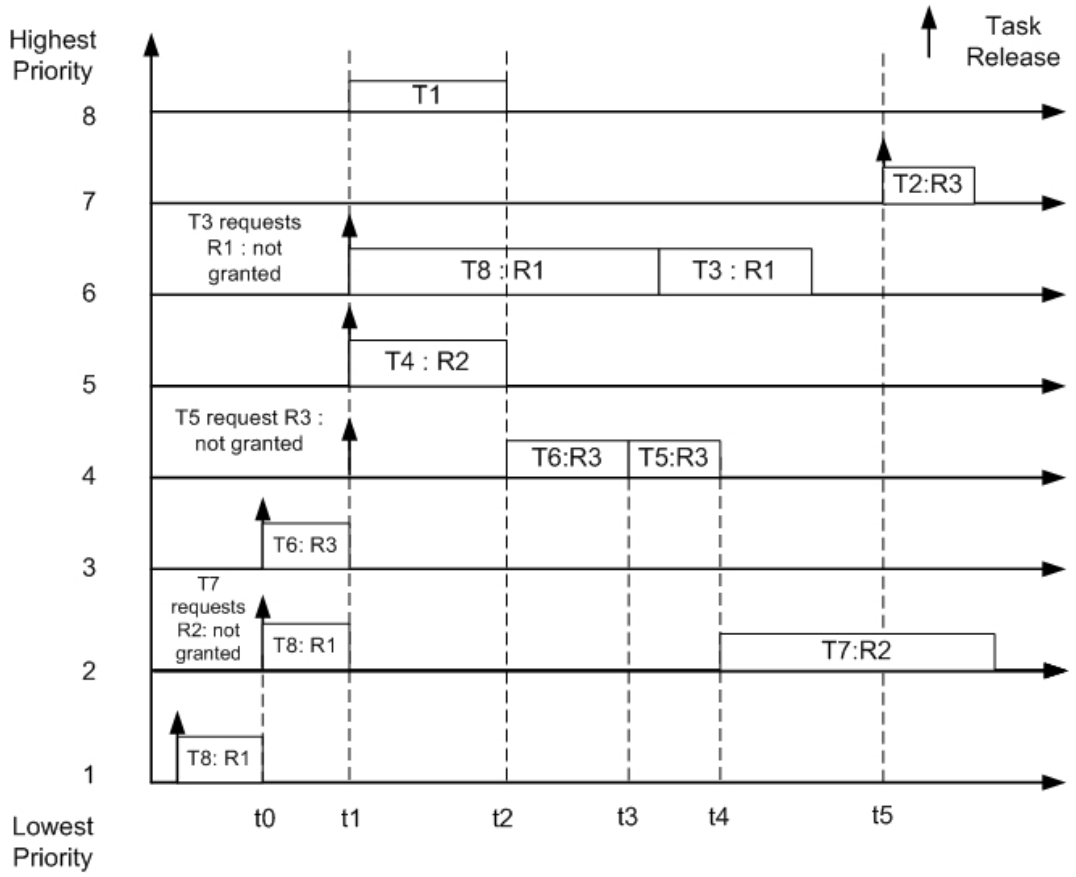

Figure 2.6: Scheduling Example of P-PCP

available processor. Then, T3 is released. Since R1 has already been locked up by T8, T3 is blocked and suspended. T8 inherits its priority running at priority level 6. Since T7 has been suspended, R2 is granted to the newly released task T4. T4 with a higher priority preempts the current executing task T6. T5 at this point is suspended as well because R3 has been locked up by T6. The three currently running tasks are T1, T8 and T4.

At t2, T4 finishes R2 which then yields the processor to the preempted task T6.

At t3, after T6 releases R3, T5 is granted with R3 and starts execution.

In this case, T5 suffers blocking only from T8 and T6 instead of T6, T7 and T8 in the previous case. This is owing to the tuning parameter α which suspends T7 at t0.

The number of units T5 suffers in this situation is shown in Figure 2.7

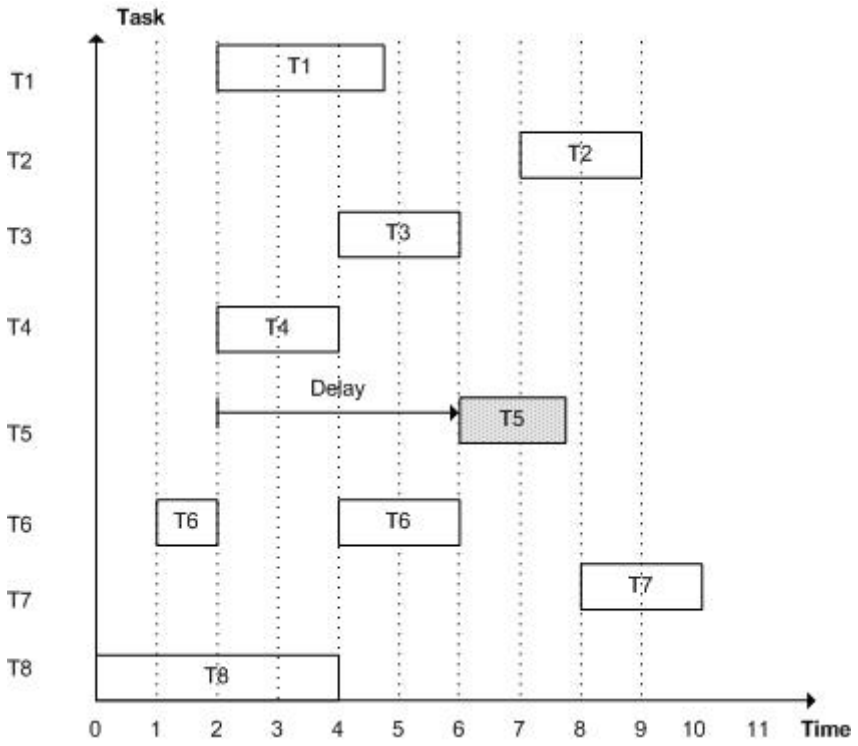

Figure 2.7: Response Time of T5 under P-PCP

All executions for tasks other than T7 are the same as they were in Figure 2.5.

Since T7 is suspended at time 1, the blocking introduced to T5 is reduced. As we can see in this figure, T5 only suffered delay from t2 to t6.

Since P-PCP was developed to reduce the blocking chain length to improve the general response time of high priority tasks, the paper conducted a response time analysis estimating the upper bound of the blocking time encountered by a task T under global scheduling. The response time of task T on multiprocessor can be estimated from the following factors:

- Execution time of task T itself
- The amount of time that task T needs to wait before being granted the resource
- The amount of execution by other tasks that have higher effective-priority than T.

Compared with previous resource sharing schemes, P-PCP reduces the interference suffered by the highest priority task by restricting the number of lower basic priority but higher effective priority task allowable for execution. By the time when the highest priority task is released, the longest waiting time suffered will be limited to the α set to the highest priority level. Also, the low priority tasks are benefited from this rule. By reducing the length of blocking chain, the waiting time of low priority tasks is reduced. The faster the low priority task processes the shared resource, the smaller chances the highest priority task gets blocked by unavailable resources. When less number of tasks are blocked on unavailable resources, more tasks can be executed in parallel which in turn increases the performance of the system. However, P-PCP didn't introduce support for nested resource sharing.

2.2.6 OMLP

The O(m) locking protocol (OMLP) [11] was introduced with an m-exclusion lock to reduce the negative effect of using priority inversion algorithms on long resources. Low priority resource requests are still delayed but the starving effect is prevented. The OMLP algorithm is proposed with an alternative design principle which abandons the resource participation method used by FMLP. Unlike the idea

of resource classification, OMLP separates the tasks into two queues based on the number of resource-competing tasks running at one time. When the total number of resource-contenting tasks is smaller than the number of processors in the system, all resource contending tasks are inserted into a dedicated FIFO queue where the shared resources are served in FIFO order. If there are at least M tasks requesting the same resource, the excessive tasks are inserted into the priority queue where all tasks are ordered by their active priority.

OMLP supports both partitioned and global scheduling. In partitioned scheduling, a global FIFO queue and a dedicated priority queue is needed for each resource on every processor. This is because tasks are not allowed to migrate to different processors in execution. A contention token is inserted into each processor acting as a binary semaphore representing the resource request in a dedicated global FIFO queue. The task holding the contention token waiting in the global FIFO queue is suspended until it becomes the head. In which case, it is granted the shared resource and all other tasks waiting on the same resource are suspended. The algorithm is explained in detail as follows:

Global OMLP Each global resource, k, is managed by two queues: a FIFO queue (FQ_k) and a Priority Queue (PQ_k). A resource is always granted to the head element of FQ_k. The FQ_k is restricted to contain M (the number of processors in the system) tasks waiting to access the resource. If the number of waiting task exceeds M - 1, the rest are placed in PQ_k. Tasks in PQ_k are priority ordered where the highest priority task will be the next task dispatched to FQ_k if the number of task in FQ_k at any time is less than M. When a task releases the resource k, it is dequeued from the FQ_k queue and the task at the new head of the queue is resumed at a priority which is equal to the highest priority of any task in FQ_k and PQ_k.

Partitioned OMLP Partitioned OMLP uses contention tokens to control access to global resources. There is one token per processor, which is used by all tasks on that processor when they wish to access a global resource. Associated with each token there is a priority queue PQ_m. There is only one queue per global resource, a FIFO queue, again of maximum length M. In order to acquire a global resource, the local token must be acquired first. If the token is not free, the requiring task is enqueued in PQ_m. If free, the token is

acquired, its priority is then raised to the highest priority on that processor, and the task is added to the global FQ_k and, if necessary, suspended. When the head of FQ_k finished with the resource, it is removed from the queue, releases its contention token, and the next element in the queue (if any) is granted the resource.

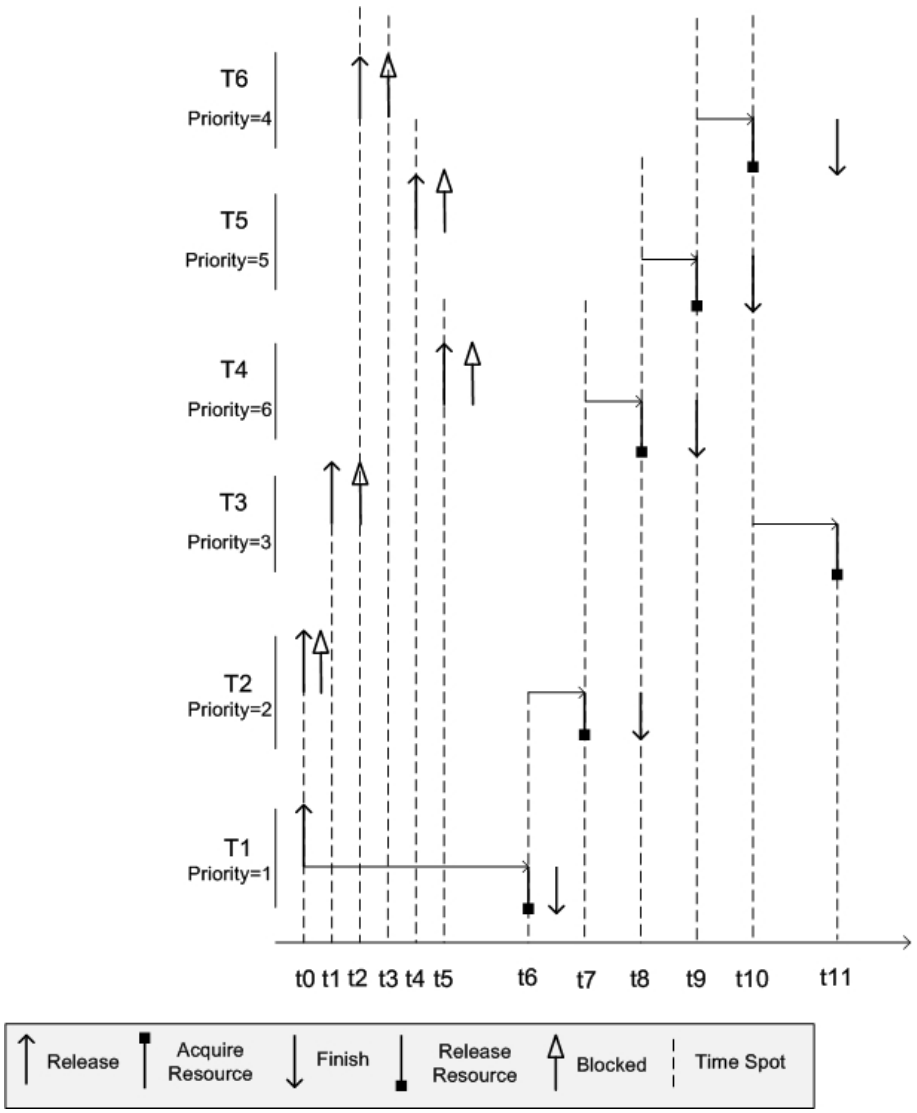

Figure 2.8: Global OMLP Example

Figure 2.8 demonstrates the Global OMLP multiprocessor resource sharing

algorithm in a detailed example. The scenario contains 6 tasks running on 2 processors. The release time of the tasks varies from t0 to t5. For demonstration purpose, only a single resource is shared between all tasks.

T1 is the first task released in the system at t0. It starts execution and immediately acquires the shared resource R. At the same time, T2 is released but finds the shared resource R is not available. According to the global OMLP rule, T2 joins the global FIFO queue and waiting for R to be available again. At t1, T3 is released to processor 2. It is blocked as well since R is still being held by T1. However, T3 is not able to join the global FIFO queue because the length of the FIFO queue is limited to 2 in this scenario and there are already two tasks running in the FIFO Queue. T3 therefore joins the priority queue and waiting for a position to become available in the FIFO queue.

T6 is then released and acquires resource R. As R is still being held by T1, T6 is suspended and joins the priority queue of R for similar reason to T3. T5 and T4 are released on t3 and t4 respectively. They both join the priority queue and wait for R to become available. At this moment, there are 4 tasks waiting in the priority queue. T4 is the one with the highest priority. For this reason, it is positioned at the head of the priority queue.

When T1 releases R at t6, T2 is immediately granted with the resource and starts execution. T4, the head of the priority queue, is removed into the FIFO queue. It is positioned as the next task accessing the shared resource once R is released by T2. T6, T5 and T4 are migrated from the priority queue to the FIFO Queue sequentially and granted the shared resource R. When T3 joins the FIFO queue, there is no task waiting in the priority queue. It is granted with R at t10 and finishes execution at t11.

In partitioned OMLP, task organization is more complicated than in its counterpart global OMLP. In global OMLP, all tasks are managed by the pair of global FIFO and priority queue. In partitioned OMLP, tasks are managed by one global FIFO queue and a number of priority queues. This complex partitioned queue structure causes problems in the context of sharing more than one resources. That is, when a high priority task arrives at a partition or cluster, the resource holding a low priority task can be preempted. This can result in extra priority inversion and possibly cause deadline missing for the late arriving task.

The combination of FIFO and priority queues reveals the significance of OMLP

in obtaining the best from both FIFO and priority based queuing policies. The highest priority task in OMLP has to wait only at most M number of tasks waiting in the FIFO queue. Although not competitive to MPCP, where the highest priority task suffers only one longest critical section of any lower priority tasks, it is still positive that the blocking suffered by the highest priority task is predictable. However, further optimization can be considered to reduce the blocking time of highest priority task.

2.2.7 Priority Donation

The priority donation protocol is an extension of the OMLP for clustered based systems. The priority donation is a form of priority boosting where the resource holding tasks are forced to be unconditionally scheduled within its cluster. Let us firstly consider a simplified example of illustrating the motivation of priority boosting. T1 and T2 are the two tasks running in a cluster with only one processor. T3 is dispatched to a different cluster but sharing the same resource R with the other two tasks. If T1 is firstly released and locked the resource, the release of T2 will preempts the resource holding task T1 since it has a higher priority. T3, if becomes released and attempts R in a different cluster, will suffer extra blocking from T2. The priority boosting prevents this situation by forcing T1 to be scheduled. The original priority boosting has a potential side effect of preempting the same "victim" task. It is highly possible that one task can suffer more than one priority boosting during its execution cycle. The priority donation forms a special one to one relationship between the priority donor and the priority recipient task where only the task actually cause the priority inversion will have its priority donated. With priority donation, the "victim" task is predetermined at the releases and each task is preempted at most once.

The rules of priority donation are summarized as follows [12]:

1. A task T_d becomes a priority donor to T_i during t_a (t_a : the time period between a task issuing its resource request and the resource being released by the task) if :
   (a) T_i was the C^{th} highest priority pending task prior to T_d's release
   (b) T_d has one of the C highest base priorities

(c) T_i has issued a global request that is incomplete at period t_a

2. T_i inherits the priority of T_d during t_a.

3. If T_d is displaced from the set of the C highest priority tasks by the release of T_h , then T_h becomes T_i's priority donor and T_d ceases to be a priority donor.

4. If T_i is ready when T_d becomes T_i's priority donor, then T_d suspends immediately. T_i and T_d are never ready at the same time.

5. A priority donor may not issue resource requests. T_d suspends if it requires a resource while being a priority donor.

6. T_d ceases to be a priority donor as soon as either :

   - T_i completes its request
   - T_i's base priority becomes one of the C highest
   - T_d is being relieved by a later released priority donor task.

Let us consider the following example which demonstrates the priority donation in a greater detail:

Figure 2.9 depicts a fixed priority schedule where T1 and T4 donate their priority to lower priority tasks T3 and T6. The system contains two clusters running 3 tasks each. Two resources R1 and R2 are shared between all tasks in the system. Since there are two processors in the system, each cluster allows two tasks to be running parallel to each other.

T2 is firstly released to cluster 2 at t0. At the same time, T3 with the lowest priority 1 is released and immediately requires R1 at t1. T1 is released at t3 with the highest priority in cluster 2. By the definition of the used scheduling algorithm, T3 could have been preempted and yield the processor to T1. However, according to the rule of priority donation, T3, as a resource holding task should always be scheduled and, continues executing and receives priority from T1. It is then running at priority T1 and becomes a priority recipient. T1 in this case becomes a priority donor that is not allowed to require any resources when its priority is being donated. T1 ceases to be a priority donor from t4 and immediately starts

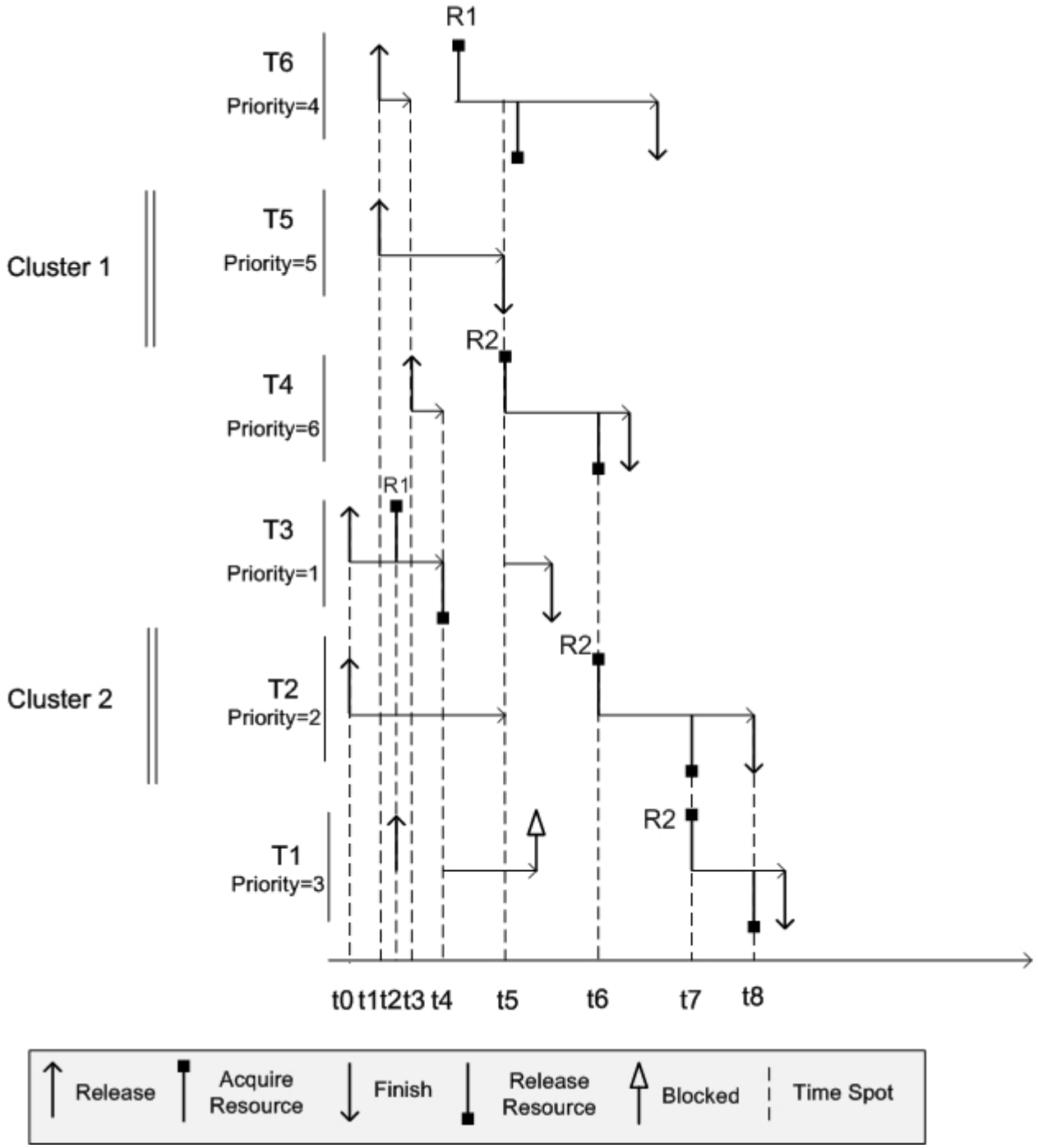

Figure 2.9: OMLP Priority Donation

execution blocked by T4 in cluster 1. It resumes execution at t7 when R2 is released by T2.

Similarly, when T4 is released at cluster 1 at t4, T6 has already issued the resource request of R1. T4 therefore donates its priority to T6 and is suspended when it becomes one of the two highest priority task in the system. This is happening at t5 when T5 finishes its execution in cluster 1. T4 in this case starts execution and acquires R2. T6, running at priority 6, acquires R1 shortly after t4 and finishes just after t5.

Without priority donation, a possible schedule could cause T6 to miss its deadline. That is, when T1 is released, T3 is immediately preempted. T6 has to wait both T1 and T3 to finish their execution which increases its blocking time in acquiring R1. With T1's priority being donated, T1 is not allowed to execute while being a priority donor. T3 will execute R1 without interference. In this case, T6 will be able to start execute once R1 is released from T3. The interference from T1 is removed.

2.2.8 SPEPP

Making resource holding tasks non-preemptive has an advantage of reducing the scheduling cost of all tasks. This is because the preemption operation itself is expensive [46]. With a FCFS-ordered queueing spin lock algorithm, in the case of being preempted, waiting tasks can be enqueued at the end of the waiting queue. This preempted task has to wait for other tasks to execute before beginning its reexecution. This expense, known as the preemption cost, must be minimized in order to for system scalability. The main motivation behind this consideration is that the interrupt handler and system interrupt are often assigned equal or higher priority than the highest priority task in the system. The interrupt rate in systems is normally very high [61]. The preemption cost often worsens the interrupt service time.

The SPEPP algorithm, proposed by [61], was designed to minimize the impact of preemption cost. Tasks under SPEPP are enqueued with an operation block on the waiting queues. An operational block contains all memory space for input and return values. With this structure, the resource requests appended in the queues are executed in strictly FCFS manner regardless of preemption. If it is the task's

turn to acquire the lock while it is preempted, the posted operation is executed by the non-preempted task at the head of the queue. Since SPEPP is more focused on low-level resource sharing details, it can be applied to either partitioned or global scheduling. The SPEPP algorithm is summarized as follows [61]:

- The atomic test and set primitive is used to update the shared variable indicating the preemption condition of the waiting task.
- Once a task acquires the lock, it executes all the operations in the queue without releasing the lock until it executes its own operation, except when an interrupt request is detected.
- The interrupt request is probed during doing test and set for the spin locks. However, pending interrupts must be served. The resource holding task must check the interrupt requests after serving the lock. If there are pended interrupt requests, it executes the corresponding interrupt handler for a bounded time before getting back to execute operations requested by the other tasks.
- The data structure for the spin lock and that for the operation queue are merged into a single data structure.

The pseudo code of the algorithm is given as below [61]:

```
1  shared var OpQueue;  // The operation queue;
2  shared var SpinLock; // Spin Lock
3
4  var opblock; // operation block
5
6  var op; // pointer to an operation block
7
8  // main routine
9  enqueue_tail(&opblock, OpQueue);
10 while the operation in opblock has not been executed do
11     acquire_lock(SpinLock);
12     op = dequeue_top(OpQueue);
13     execute(op);
14     release_lock(SpinLock);
15 end;
```

Listing 2.1: SPEPP synchronization

The head task will follow the main routine depicted by Listing 2.1 to execute all operations until either there are no operations appended in the queue or it has met the other request issued by itself. Once a critical section has been executed,

the corresponding shared variable of the waiting task will be updated. The waiting task will actively check on its own shared variable. If updated, the waiting task will exit the resource request routine and leave the spin queue.

The significance of the SPEPP algorithm is the delegation of execution rights. All previous multiprocessor resource sharing protocols assume that the task executes the critical section needs to be the owner task itself. The order of accessing a shared resource depends on the scheduling outcome of the owner tasks. The SPEPP algorithm offers an alternative perspective to solve the multiprocessor resource sharing problem through decoupling task scheduling and resource sharing.

2.2.9 Main Characteristics of Resource Sharing Protocols

The above multiprocessor resource sharing protocols are explained in detailed scenarios. Such examples are good for demonstration purposes but are less effective for revealing the main characteristics of the algorithms. Table 2.5 summarizes the main characteristics of the above protocols:

The algorithms, shown by Table 2.5, are all having different restrictions on scheduling algorithms, differentiating global and local resource, supporting of nested resources, accessing priority of shared resources and queuing policies. MPCP and MSRP only work with partitioned scheduling. Only FMLP and OMLP accept nested resources. The resource accessing priorities are under the dual impact of a resource sharing algorithm and the programming language restrictions. The choice of resource accessing priority has to satisfy the integrity of the resource sharing algorithm. Some algorithms are using ceiling priority as the accessing priority instead of priority inheritance. Alternative non preemptive accessing and priority donation mechanisms are also permitted. Similarly there is no single queueing policy that defines the waiting tasks operations agreed by all algorithms. Overall, no single protocol has emerged as being suitable for all resource sharing situations.

Protocol	Scheduling	Resources	Nested Resources	Access Priority	Queueing
MPCP	Partitioned	Yes	No	Ceiling	Suspends in a priority ordered queue
DPCP	Partitioned	Yes	Yes	Ceiling	Suspended in a priority ordered queue
MSRP	Partitioned	Yes	No	Non Preemptive	Spins in a FIFO queue
FMLP	Both	Yes	Group Lock	Non Preemptive for short; Suspension for long	Short: Spins in a FIFO queue; Long: Suspends in FIFO queue
PPCP	Global	No	No	Inheritance	Suspends in a Priority queue
OMLP	Both	Yes	Group Lock	Inheritance for global, Non preemptive for partitioned	Suspends in FIFO and Priority-ordered or Contention token queues
Clustered OMLP	Clustered	No	Group Lock	Priority Donation	Suspends in a FIFO queue
SPEPP	Both	No	No	Non Preemptive	Spins in a FIFO queue

Table 2.5: Summary of Multiprocessor Resource Sharing Protocols

2.3 Support for Multiprocessor Scheduling and Resource Control Protocols in Ada, RTS and Linux

The emergence of multiple processors imposes challenges on language abstractions. The facilities that the languages provide to exploit parallelism in multiprocessor architectures vary and there are no agreed standards. In order to benefit from this new architecture, tasks must be able to run in true parallel to each other on different processors. Tasks are usually assigned affinities to restrict the degree of migration between processors. The affinity settings also acts as a support mechanism for multiprocessing. The control of affinities is as important as the control of priorities in multiprocessor environment[16]. On top of this basic requirement, the predictability of the tasks is also important in real-time systems. The tasks sharing resources together are tightly coupled and their execution is normally modified by synchronization. The following sections investigate the support of multiprocessors in programming languages and operating systems in the context of processor affinity and available resource sharing mechanisms.

2.3.1 Multiprocessor Support in Ada

In the context of scheduling algorithms, Ada 2012 offers a wide range of scheduling options available for multiprocessor systems [24].

- Global preemptive priority-based scheduling
- Fully partitioned preemptive priority-based scheduling
- Global EDF scheduling
- Partitioned EDF scheduling

In addition to the above, Ada allows groups of processors to form "dispatching domains" (clusters), where each processor can only be part of one dispatching domain. Tasks can be globally scheduled within a single dispatching domain and it is also possible to fix a task to run on a single processor within a dispatching domain.

Coordination between tasks is required for resource allocation between competing tasks. In reality, external devices, files, shared memory spaces, buffers and protected algorithms are not physically transmitted between tasks. These devices often have shared information available to other devices through read and write operations[15]. Resource control protocols protect the integrity of the shared information. If the integrity of the data is compromised, for example when a task fails to update the data, it would be necessary to inform all involved tasks. Ada adopts the avoidance approach to support resource sharing at the language level. The avoidance approach relies on the guard or the acceptance of the conditions. Only the tasks satisfying the conditions under the acquiring request can be safely accepted and proceed. The condition is normally known as the guard (a barrier).

In conventional monitors, read and write operations are regarded as the same operation having a similar impact on the shared data. It offers a level of abstraction where both read and write operations on shared data require mutual exclusion. However, Ada offers a distinction between read and write appreciating the fact that read operation is not changing the underlying content of the data. Therefore, multiple read operations are allowed to run parallel to each other in Ada.

A protected object provides three types of accesses to its encapsulated shared data via protected entry, protected procedure and protected function. These protected actions ensure that the integrity of the encapsulated data is protected. A protected procedure provides complete mutual exclusive access to the data irrespective of the read or write nature of the operation. A protected function appreciates the nature of the read operations where concurrent read-only access to the data is allowed. A protected entry is similar to protected procedure with an extra boolean expression, also known as the barrier. In a call to a protected entry, the barrier is firstly evaluated. The caller task can only proceed inside the body of the entry if the barrier is evaluated to be true. If not, it is suspended until the barrier evaluates it to be true and no other tasks are currently active inside the protected object. The protected objects of Ada is explained with further detail in Chapter 4.

However, the Ada Reference Manual (ARM) and its annotated companion (AARM) do not fully define the access protocol for protected objects on a multiprocessor system. The following points summarize the current position.

1. Where there is contention for a protected object's lock, the possibility to use

spin-locks is a discussion point to a note.

> *"If two tasks both try to start a protected action on a protected object, and at most one is calling a protected function, then only one of the tasks can proceed. Although the other task cannot proceed, it is not considered blocked, and it might be consuming processing resources while it awaits its turn. There is no language-define ordering or queuing presumed for tasks competing to start a protected action - on a multiprocessor such tasks might use busy-waiting;" for monoprocessor considerations, see D.3, "Priority Ceiling Locking".*

2. It is implementation defined whether to spin non-preemptively, or, if not at what priority. Furthermore, it is not defined whether there are queues (FIFO or priority) associated with the spin-lock

   > *"It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy." AARM D.2.1. par 3*

3. The task which executes a protected action is not specified.

   > *"An implementation may perform the sequence of steps of a protected action using any thread of control; it need not be that of the task that started the protected action. If an entry_body completes without requeuing, then the corresponding calling task may be made ready without waiting for the entire protected action to complete. The reason for this is that these permissions are intended to allow flexibility for implementations on multiprocessors. On a monoprocessor, the thread of control that executes the protected action is essentially invisible, since the thread is not abortable in any case, and the "current_task" function is not guaranteed to work during a protected action seeC.*7.1*." AARM 9.5.3 pars 22 and 22.a*

4. The ceiling locking policy must be used with EDF scheduling.

   > *"If the EDF_Across_Priorities policy appears in a Priority_Specific _Dispatching pragma seeD.*2.2 *in a partition, then the Ceiling_Locking*

policy seeD.3 shall also be specified for the partition." AARM D.2.6 par 11.2

"The locking policy specifies the meaning of the priority of a protected object, and the relationships between theses priorities and task priorities". AARM D.3 6.2

5. The value of the ceilings can be altered by the implementation.

   Every protected object has a ceiling priority, which is determined by either a Priority or Interrupt_Priority pragma as defined in D.1, or by assignment to the Priority Attribute as described in D.5.2. The ceiling priority of a protected object orceiling, forshort is an upper bound on the active priority a task can have when it calls protected operations of that protected object." AARM D.3 par 8.2

   "The implementation is allowed to round all ceilings in a certain subrange of SystemṖriority or SystemİnterruptPriority up to the top of that subrange, uniformly."

In summary, Ada generally assumes the use of spinlocks but does not rule out other approaches.

The order of accessing protected objects is determined by the scheduling and dispatching algorithms. Since Ada 2005, Ada has supported different dispatching policies including fixed priority (FP) and EDF. EDF dispatching can be applied across the whole range of priorities or across a restricted range. In this way, EDF scheduling is integrated into a FP framework. Baker's stack resource policy [4] is also integrated with Ada's ICP protocol to support resource sharing using protected objects.

The approach that Ada adopted for EDF scheduling was novel, and the protocol's correctness was not formally verified. As a result the initial definition was found to contain errors [75] and had to be corrected. Although the protocol is now believed to be correct, its properties on a multiprocessor platform are unclear. This is partly because Ada does not completely define how protected objects are accessed in a multiprocessor environment. Also, according to [26], the SRP was

designed for single processor systems and cannot be directly applied to multiprocessors.

> *"The Stack Resource Policy has several interesting properties. It prevents deadlock, bounds the maximum blocking times of tasks, reduces the number of context switches and can easily extend to multi-unit resources. ... However, the SRP does not scale to multiprocessor systems."*

Table 2.5 captures the main characteristics of the resource sharing protocols reviewed in this thesis. The following summarizes whether the current protocols are compatible with Ada when busy-waiting is assumed.

MPCP No: MPCP suspends on an unavailable lock.

DPCP No: DPCP suspends on an unavailable lock.

MSRP Yes: if MSRP uses ceiling of ceilings for resources.

FMLP Partial: if FMLP supports short resources only.

PPCP No: PPCP suspends on an unavailable lock and no immediately inheritance.

OMLP No : OMLP suspends on an unavailable lock.

As can be seen, MPCP, DPCP, PPCP and OMLP are not a good match for Ada as they suspend on access to a resource. In a fully partitioned system, this might be enforced by a Ravenscar-like profile [22], as a slight variant of the MSRP can be used. That is tasks are set to spinning at the highest priority of the processor ceilings. If FMLP is constrained to short resources, it is essentially equivalent to using MSRP on the group lock and treating all resources as global. It perhaps should be noted as an aside, that use of the Ada rendezvous could be considered a long resource.

In Ada, nested resource access is allowed. In most of the protocols, as shown by Table 2.5, either nested resource accesses are disallowed, or the notion of groups is introduced (as in the FMLP) and a group lock must be obtained. This is to avoid deadlocks.

2.3.2 Multiprocessor Support in RTSJ

The Real-Time Specification for Java (RTSJ) Version 1.1 [10] provides more explicit support for multiprocessor issues than previous versions of the languages. However, issues surrounding the use of synchronized objects are still, to a large extent, unresolved.

In Java, a monitor is an object with the important property that the methods that are labeled as synchronized are executed atomically with respect to each other. This means that one synchronized method call cannot interfere with the execution of another synchronized method. The way this is achieved, in practice, is by ensuring that the calls are executed in mutual exclusion. There are several different ways of implementing this, for example, by having a lock associated with the monitor and requiring that each method acquires (sets) the lock before it can continue its execution.

The support that the RTSJ provides for multiprocessor systems is primarily constrained by the support it can expect from the underlying operating system. The following have had the most impact on the level of support that has been specified.

- The notion of processor affinity (or dispatching domains, as Ada calls them) is common across operating systems and has become the accepted way to specify the constraints on which processor a thread can execute. RTSJ directly supports affinities. A processor affinity set is a set of processors that can be associated with a Java task or RTSJ schedulable object. The internal representation of a set of processors in an AffinitySet instance is not specified, but the representation that is used to communicate with this class is a BitSet, where each bit corresponds to a logical processor ID. The relationship between logical and physical processors is not defined. Affinity sets allow cluster scheduling to be supported.

  Support is grouped together within the ProcessorAffinitySet class. The class also allows the addition of processor affinity support to Java threads without modifying the threads object's visible API.

- The range of processors on which global scheduling is possible is dictated by the operating system. RTSJ supports an array of predefined affinity sets.

Theses are implementation-defined. They can be used either to reflect the scheduling arrangement of the underlying OS or they can be used by the system designer to impose defaults for, say, Java threads, non-heap real-time schedulable objects etc. A program is only allowed to dynamically create new affinity sets with a cardinality of one.

- Many OSs give system operators command-level dynamic control over the set of processors allocated to a process. Consequently, the real-time JVM has no control over whether processors are dynamically added or removed from its OS process.

  Predictability is a prime concern of RTSJ. Clearly, dynamic changes to the allocated processors will have a dramatic, and possibly catastrophic, effect on the ability of the program to meet timing requirements. Hence, RTSJ assumes that processor set allocated to RTSJ threads does not change during its execution.

RTSJ Version 1.1 does not prescribe a particular implementation approach for supporting communication between schedulable objects running on separate processors.

In summary, RTSJ allows both globally scheduled, cluster and partitioned systems to be constructed but is silent on the details of communication between parallel schedulable objects - simply allowing priority inheritance or priority ceiling emulation. The problem is that the RTSJ/Java model is so flexible that an implementation does not have enough knowledge of what the application is doing in order to optimize its approach. Nested resource access must be assumed, and self suspension while holding a lock is possible. However, the soundness of these assumptions is based on the accordance of the semantics of the implementations. Of course, if an implementation only supports single processor pre-defined affinity sets then it is enforcing a fully-partitioned system. These implementation restrictions must be fully minded by the application developers.

2.3.3 Operating System Support

Although multiprocessors are becoming prevalent, there are no agreed standards on how best to address real-time demands. For example, the RTEM operating system

does not dynamically move tasks between CPUs. Instead it provides mechanisms whereby they can be statically allocated at a link time. In contrast, QNX's Neutrino [59] distinguishes between "hard thread affinity" and "soft thread affinity". The former provides a mechanism whereby the programmer can require that a task be constrained to execute only on a set of processors (indicated by a bit mask). With the latter, the kernel dispatches the task to the same processor on which it last executed (in order to cut down on preemption costs). Other operating systems provide similar facilities. For example, IBM's AIX allows a kernel thread to be bound to a particular processor [32].

POSIX POSIX is an abbreviation of the important standard known as Portable Operating System Interface for Computer Environments [34]. Since its first proposal in 1980 in AT&T's Unix system, POSIX was developed to a family of IEEE standards that supports portable programming. The first standard, introduced in 1990, defines the C language interface to operating system services [1]. The standards describes itself as:

> "[POSIX.1] defines a standards operating system interface and environment to support application portability at the source-code level. It is intended to be used by both application developers and system implementors."

Improving the application performance is always attracting the interests of the developers. Parallel processing, by having multiple processors running at the same time, is an applicable way of enhancing the computation power from the hardware. At the software side, multi-threading is essential to extract benefit from the parallel executing hardware or, at the bottom line, increase the throughput of applications on uniprocessor systems. A series of POSIX standards were proposed and authorized by IEEE technical committee on Operating Systems for tightly coupled multitasking environments. The specific functional areas covered are [27]:

Thread Management Creation, Control and Termination of Threads under a common shared address space

Synchronization Primitives Mutual exclusion and conditional variables, optimized for multitasking environments (e.g: Multiprocessors)

Harmonization Conforming to POSIX 1003.1 interfaces.

The POSIX 1003.1 and 1003.5b defines a range of scheduling algorithms including fixed priority scheduling policy, round robin and sporadic server policy. These scheduling algorithms are embedded into the POSIX compatible OS and accessible to application tasks running on those systems. It is possible for POSIX to incorporate dynamic new policies in addition to the existing ones. This is evidenced by that fact that the SCHED_OTHER policy in POSIX is implementation defined.

POSIX.1 defines a "Scheduling Allocation Domain" as a set of processors on which an individual thread can be scheduled at any given time. POSIX states that [33]:

- "For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR shall be used."

- "For application threads with scheduling allocation domains of size greater than one, the rules defined for SCHED_FIFO, SCHED_RR, and SCHED _SPORADIC shall be used in an implementation-defined manner."

- "The choice of scheduling allocation domain size and the level of application control over scheduling allocation domains is implementation-defined. Conforming implementations may change the size of scheduling allocation domains and the binding of threads to scheduling allocation domains at any time."

With this approach, it is only possible to write strictly conforming applications with real-time scheduling requirements for single-processor systems. If an SMP platform is used, there is no portable way to specify a partitioning between threads and processors.

Additional APIs have been proposed but currently these have not been standardized [57]. The approach has been to set the initial allocation domain of a thread as part of its thread-creation attributes. The proposal is only in draft and so no decision has been taken on whether to support dynamically changing the allocation domain.

There are no additional access protocols other than the standard priority inheritance and the immediate priority ceiling protocol (called the highest lock protocol by POSIX) supported by default in POSIX. However, POSIX does provide some primitive basic locking primitives for high level languages such as spin lock, mutex lock and rw lock.

As a basic conventional primitive of a POSIX shared variable, the mutex lock is granted to only one pthread at any one time. If several pthreads try to lock a mutex only one pthread will be successful. No other pthreads can proceed until the lock has been released by the owner pthread. The API is given below:

```
1 #include <pthread.h>
2 int pthread_mutex_lock (pthread_mutex_t* mutexlock)
3 int pthread_mutex_trylock (pthread_mutex_t* mutexlock)
4 int pthread_mutex_unlock (pthread_mutex_t* mutexlock)
```

A typical example using a POSIX mutex lock looks like the following pseudo-code:

```
1 // Thread 1
2 pthread_mutex_lock (mutexlock)
3 A = 2
4 pthread_mutex_unlock (mutexlock)
5
6 // Thread 2
7 pthread_mutex_lock (mutexlock)
8 A = A + 1
9 pthread_mutex_unlock (mutexlock)
```

As well being blocked on a mutex variable, POSIX provides a mechanism for pthreads to wait on conditional variables. This is particularly useful in asynchronous blocking where no predefined access order is agreed by the acquiring pthreads.

```
1 #include <pthread.h>
2 int pthread_cond_wait (pthread_cond_t* cond, pthread_mutex_t *mutex)
3 int pthread_cond_signal (pthread_cond_t* cond)
4 int pthread_cond_broadcast (pthread_cond_t* mutex)
```

The pthread_cond_wait(...) function blocks the calling task until the associated condition has been signaled. This function is called while the mutex is locked. This is to prevent the data of the function being corrupted by race condition. If the acquiring task is blocked on the condition, the associated mutex is released for the next task waiting outside the function. Once a signal is received by the conditional variable, the task is awakened and the mutex is automatically locked.

The task executing inside must release the mutex before leaving the critical section of the data. The phread_cond_signal takes the responsibility to signal the waiting conditional variable. If multiple conditional variable need to be signaled at one time, the pthread_cond_broadcast function should be used.

The pthread conditional variable should work closely with the mutex locks. A typical template should look like the following:

```
1 #include <pthread.h>
2 pthread_mutex mutex;
3 pthread_cond_t cond;
4 int count;
5
6 void *cond_write_procedure(void *t)
7 {
8     pthread_mutex_lock(&mutex);
9         printf("I am operating inside the critical section");
10        count ++;
11    pthread_cond_signal(&cond);
12    pthread_mutex_unlock(&mutex);
13 }
14
15 void *cond_read_procedure(void *t)
16 {
17    pthread_mutex_lock(&mutex);
18    pthread_cond_wait(&cond,&mutex);  /* The mutex will be
          automatically unlocked here */
19        printf("counter :%d", count);
20    pthread_mutex_unlock(&mutex);
21 };
```

From Linux, the additional support was found for mutex from POSIX. The read write lock with the pthread_rwlock_t type and its associated functions are introduced in the POSIX threads API [33]. With the read write lock, the synchronization between tasks is achieved in a way that the readers and writers are exclusively operating the shared data. With the reading lock, multiple readers are allowed to read the data at one time. Since multiple writing can corrupt the data, only one single task is allowed to write the data at one time and no readers are allowed to read when a write lock is being held by a writing task. In order to inhibit writer starvation, new readers are not allowed to obtain the lock once a new writer is waiting for access. This guarantees that the writers will be able to acquire the lock in a finite time.

```
1 #include <pthread.h>
2 int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
3 int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
4 int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

In addition to the API above, POSIX states that [33]:

- "If the Thread Execution Scheduling option is supported, and the threads involved in the lock are executing with the scheduling policies SCHED_FIFO or SCHED_RR, the calling thread shall not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise, the calling thread shall acquire the lock".

- "If the Threads Execution Scheduling option is supported, and the threads involved in the lock are executing with the SCHED_SPORADIC scheduling policy, the calling thread shall not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise, the calling thread shall acquire the lock".

- "If the Thread Execution Scheduling option is not supported, it is implementation -defined whether the calling thread acquires the lock when a writer does not hold the lock and there are writers blocked on the lock. If a writer holds the lock, the calling thread shall not acquire the read lock. If the read lock is not acquired, the calling thread shall block until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds a write lock".

With this approach, the choice of scheduling algorithm has a direct impact on the semantics of the read write lock. Again, writing strictly conforming applications in a multiprocessor environment is challenging. The behaviour of the read write lock, according to the above rule, is implementation defined. This open option means that there is no unified solution to the read write locks.

Linux and Unix In order to accommodate more real-time capabilities in multiprocessors, initial support for SMP systems was provided in kernel version 2.5.8 [38]. The Linux 2.6 kernel was introduced in 2003 with major features of SMP scalability and task affinity [64]. With the PREEMPT_RT patch, the new Linux kernel re-implemented locking and scheduling primitives. As a result, critical sections are protected with spin locks and RW locks. Priority inheritance has been built into the kernel becoming applicable for spin locks and semaphores.

In the context of scheduling, partitioning of user tasks and threads is obtained via the notion of CPU affinity. Each process in the system can have its CPU affinity set according to a CPU affinity mask [2]. A task's CPU affinity mask determines the set of CPUs on which its tasks are eligible to run. The affinity mask is actually a per-task attribute that can be adjusted independently for each of the task in a task group. The real-time scheduler of the PREEMPT_RT patchset uses cpusets to create a root domain. This is a subset of CPUs that does not overlap with any other subset. The scheduler adopts an active push-pull strategy for balancing real-time tasks. This refers to the tasks whose priority ranges from 0 to (Max_RT_RT_PRIO-1) across CPUs. Each CPU has its own run queue. The scheduler decides [28]:

1. where to place a task on wakeup
2. what action to take if a lower-priority task is placed on a run queue running a task of higher priority
3. what action to take if a low-priority task is preempted by the wake-up of a higher-priority task
4. what action to take when a task lowers its priority and thereby causes a previously lower-priority task to have the higher priority.

Essentially, a push operation is initiated in cases 2 and 3 above. The push algorithm considers all the run queues within its root domain to find one that has a lower priority task (than the task being pushed) at the head of its run queue. The task to be pushed then preempts the lower priority task.

A pull operation is performed for case 4. Whenever the scheduler is about to choose a task from its run queue that is lower in priority than the previous one, it checks to see whether it can pull tasks of higher priority from other run queues. If it can, at least one higher priority tasks is moved to its run queue and is chosen as the task to be executed. Only real-time tasks are affected by push and pull operations.

Linux does not provide additional support for mutexes. However, HP-UX (the Unix operating system implemented by HP) provides the following non-portable extensions to the pthread library [56]

```
1 #include <pthread.h>
2
3 int pthread_mutexattr_setspin_np( pthread_mutexattr_t *attr, int spin
    );
4 int pthread_mutexattr_getspin_np( const pthread_mutexattr_t *attr,
    int *spin);
5
6 int pthread_mutex_setyieldfreq_np( int yield );
7 int pthread_mutex_getyieldfreq_np (int *yield );
```

The possible values for the spin attribute are as follows:

- positive integer - the pthread_mutex_lock() function will busy-wait on the mutex lock for the specified number of iterations before blocking the task.
- PTHREAD_MUTEX_SPINONLY_NP - inhibits blocking on the mutex lock altogether.
- PTHREAD_MUTEX_SPINDEFAULT_NP - uses a built-in default value for the number of busy-wait iterations

The yield attributes specify how frequently the processor should be yielded during busy-waiting. The possible values for the yield attribute are as follows:

- positive integer - the busy-wait loop in the pthread_mutex_lock() will yield the processor after each specified number of iterations of the spin loop (where the total number of iterations is controlled by the per-mutex spin attribute).
- PTHREAD_MUTEX_YIELDNEVER_NP - inhibits yielding in the mutex lock altogether.
- PTHREAD_MUTEX_YIELDFREQDEFAULT_NP - uses a built-in default value for the frequency of yields in the busy-wait loop.

The major issue of incorporating application defined scheduling is the soundness of the integrated system and the compliance to the standards. There are suggestions about the application-defined scheduling algorithms to be programmed as a kernel module and integrated with the kernel schedulers. Running the application-defined algorithms at low level will prevent excessive interrupts from the user space. By running at the kernel space, the execution of the code will be running at higher

Event Code	Description	Additional Information
New_Task	A new task has requested attachment to the scheduler	None
Block	A task has been blocked	None
Lock_Mutex	A Task has invoked a "lock" operation on an availalbe application-scheduled mutex	pointer to the mutex
Unlock_Mutex	A task has released the lock of an application scheduled mutex	Pointer to the mutex

Table 2.6: Non-Exhaustive List of Registered Scheduling Events

priority. However, reprogramming kernel code is not only risky and time consuming, but a single error in code will corrupt the whole system. Therefore, the advent of supporting application-defined scheduling was more preferred to run the application-defined algorithms as a special thread [36]. In which case, the application scheduling thread is acting as a special POSIX signal handler for the corresponding POSIX event. For example, when a task is blocked due to unavailable resource, the kernel scheduler will be notified by the POSIX signal that the calling task is blocked. The kernel scheduler will check the background of the task and call the corresponding application scheduler thread which will make the correct scheduling decisions.

An approach along theses lines has been proposed in RTLinux [52], in which a two-level scheduler is used, where the upper level is implemented as a user task that maps application-defined server parameters to low-level attributes. An application scheduler can be created using the API depicted in Listing 2.2. The structure posix_appsched_scheduler _opts will contain pointers only to the functions used in that particular scheduler. When a registered system event occurs, as depicted by the Table 2.6, the system scheduler will check the corresponding function linked by the pointers. If assigned, the application-defined algorithm is called. If no application level algorithm has been defined, the pointers to the function will not be used nor defined.

```
1 posix_appsched_scheduler_create
2     (const posix_appscheduler_opts_t *sched_ops,
3      sizeof_t schedler_data_size,
4      void * arg,
5      size_t arg_size,
6      posix_appsched_scheduler_it_t *sched_id);
```

Listing 2.2: Application-Defined Scheduler

Changes have been proposed to low level POSIX compliant API:

```
typdef struct {

    void (*init) (void * sched_data, void *arg);

    void (*new_thread)
        (void *sched_data, pthread_t thread,
         posix_appsched_actions_t actions,
         struct timespec *current_time);

    void (*thread_block)
        (void *sched_data, pthread_t thread,
         posix_appsched_actions_t actions,
         struct timespec *current_time);

    void (*thread_unblock)
        (void *sched_data, pthread_t thread,
         posix_appsched_actions_t actions,
         struct timespec *current_time);

    ...  // Similar definitions of the scheduling event handlers
}
```

Listing 2.3: POSIX Compliant API for Integrating Application-Defined Scheduler

In Listing 2.3, the operations and methods defined in application schedulers are linked via the pointers to the primitives operations which are invoked by the system when a scheduling event occurs. For example, a new application task may be created with the underlying system using new_thread function. The application developers can redefine the behaviour of the method by associating a new customized function using the above interface.

The interactions between the scheduler and the POSIX is described by the pseudo code of the init_module() function:

```
int init_module (void) {
    pthread_attr_t attr;
    struct sched_param sched_param;
    pthread_t task;
    ... // other definitions
    posix_appsched_scheduler_id_t *edf_scheduler_id;

    posix_appssched_scheduler_ops_t edf_scheduler_ops =
        (...);

    // Scheduler creation
    posix_appsched_schduler_create (
```

```
13          &edf_scheduler_ops,
14          0, Null, 0,
15          &edf_scheduler_id);
16
17      // Application scheduled threads creation
18      ...
19  }
```

Listing 2.4: Init_module()

The use of resources may cause priority inversions or other blocking effects, it is necessary that the scheduler takes the resources into consider to establish its own protocols adapted to particular task set. POSIX has POSIX_THREAD_PRIO _INHERIT and POSIX_THREAD_PRIO _PROTECT built in as the default protocols for synchronization between tasks. However, the applications cannot change the priority of the tasks using the default protocols. Rivas and Harbour [53] defined a scheduling interface following the application-defined scheduling framework from above to incorporate the application-defined resource sharing protocols into the application schedulers. It classified the mutexes as below:

System-Scheduled Mutexes The default POSIX defined protocols can be used to access resource shared between application schedulers and the system schedulers.

Application-Scheduled Mutexes These protocols are only applicable to those threads created by the same scheduler. The behaviour of the protocol itself is defined by the application schedulers.

When a scheduling event, LOCK_MUTEX, UNLOCK_MUTEX, occurs with impact on any application scheduled task, the related application scheduler is notified through POSIX signals and the corresponding primitive operation will be invoked. The application scheduler will then decide what action to take in order to resolve the resource sharing confliction. Through the use of these functions, an application scheduler can reactivate or suspend its scheduled tasks for the purpose of resource sharing.

2.4 Summary

In order to benefit from multiprocessor architectures, the necessary updates are applied to conventional uniprocessor resource sharing algorithms and brand new

elements are introduced. The remoteness of the resource and affinity of the tasks are all having an impact on the predictability of real-time systems. Various multiprocessor resource sharing protocols have been designed to reduce the impact of remote blocking and incorporate new resource sharing principles. For example, MPCP tackles remote blocking by introducing a synchronization processor. P-PCP makes a trade-off between parallelism and response time of high priority tasks. OMLP utilizes the power of the priority queue reducing the waiting time of high priority task. Priority Donation means that preemption cost, which is expensive and common in multiprocessors, is reduced. However, there is little evidence to prove that there is an optimal solution available yet for resource sharing in multiprocessors. The main reasons behind this are:

- The locking primitives are mainly provided by the underlying hardware and the operating systems. The POSIX and Linux are still developing their incorporation of multiprocessor primitives.

- The requirements and scope of the multiprocessor resource sharing algorithms are all different from each other. For example, some support partitioned scheduling only. Some do not support nested resource.

- The multiprocessor resource sharing algorithm suffers impact from a wide range of factors (Task affinities, length of the resource, scheduling algorithm, underlying hardware primitives and nature of the tasks etc.). These factors are often affected largely by the application semantics. There are hardly any unified solutions available to solve this problem under all scenarios. This is further evidenced by [20] that there is no dominant scheduling algorithm yet available to multiprocessors.

Research efforts have emerged from different perspective to provide support for sharing resources in multiprocessors. The POSIX standard has provided three locking primitives. The Linux real-time patched 2.6 kernel has provided a preempted kernel, SMP support, real-time priorities and SCHED_FIFO schedulers. Programming languages have suggested that the effectiveness of the resource sharing algorithm is largely dependent on implementation (as it is implementation defined).

Finally, this chapter provides the evidence to support the thesis hypothesis, namely: **The performance of a multiprocessor resource sharing protocol is largely dependent on the application semantics. This thesis contends that it is, therefore, inappropriate to introduce support for a particular multiprocessor resource sharing protocol into a language definition.** In the following two chapters, the thesis explains how the application-defined protocols can be integrated into a programming language through the framework to achieve better resource sharing performance in multiprocessor systems.

Chapter 3

A Flexible Resource Sharing Framework

It is clear that techniques for multiprocessor scheduling and global resource access are still in their infancy. The new facilities that Ada and the RTSJ 1.1 provide for global, cluster-based and partitioned scheduling have added a great deal of flexibility into the languages. As reviewed in the previous chapter, the affinity and multiprocessor dispatching domains are all open to various implementation decisions. The implementation developers can define the best suitable multiprocessor resource sharing protocols according to their needs. According to [20], tuning the existing multiprocessor resource sharing protocols alone is not effective to obtain the optimal protocol. The optimality of a particular protocol, varies with the application scenarios. The programming language should therefore incorporate the facility to accept application-defined resource sharing protocols. This requires a flexible multiprocessor resource sharing framework that is integrated with the programming languages resource sharing model.

This chapter introduces a flexible resource sharing framework that incorporates the mechanism that the application developers can use to integrate their own multiprocessor resource sharing protocol at compile and run-time. With the framework, the application developers can instantiate a template from the framework and implement their own protocol by overloading the original procedures. At compile time, those procedures will be instantiated and linked with the runtime library. Whenever the specific resource is called, the application-defined procedures

will be dispatched instead of the default ones.

The framework is based on the monitor which is a widely accepted concept in popular programming languages and underlying systems. In this chapter, the concept of a monitor is explained first. It is followed by demonstrating the design of the framework together with its interactions with runtime libraries and other language facilities.

It is expected that the application-defined protocols can be integrated through standardized API with the framework at a reasonable low cost. The interactions between the framework and the underlying systems are then measured to estimate the overhead of the framework at the next chapter. It has been considered that a full pledged implementation is not feasible for the purpose of this research. A series of simulation implementations were deployed as an approximation of the full pledge code.

3.1 Basic Assumptions

Due to the complexity and variety of the fast changing underlying hardware components, the following assumptions are made:

1. It is assumed that the processors are executing the code at the same speed. All processors are identical to each other.

2. Two programs with the same assembly code should have the same semantics.

3. The main memory access cost is uniform and does not depend from which processor it is being accessed.

4. Tasks are the smallest scheduling entity within the system. The critical sections are executed by the tasks themselves.

5. The resources are software resources. That is a critical-section of code where mutual exclusive access is required to protect the data encapsulated from corruption. We do not consider read-write locks in this thesis.

3.2 Monitors

A monitor is a synchronization entity that enforces mutual exclusion [55]. The monitor was firstly introduced by Concurrent Pascal [29] as a synchronization mechanism to incorporate resource sharing between tasks. It is inherited by Java and has become widely accepted by popular programming languages. The definition of monitor varies between different language; In Ada, it takes the form of protected object. A general definition was given by [71]

> "A monitor is an encapsulation of a resource definition where all operators that manipulate the resource execute under mutual exclusion"

The resource under protection is typically referred to the data that is declared as private to the monitor. This data can only be accessed by the code listed inside the monitor. In the context of Ada, the resource hidden in the body of monitor can only be accessed via the procedures defined in the specification. Although, there can be multiple tasks calling the same method of a monitor at one time, it is guaranteed, by definition, that only one task is allowed to execute at a time. Other tasks must wait until the previous granted task has finished with the resource. A typical monitor is depicted by the pseudo code in Listing 3.1:

```
1  monitor class Account {
2
3      private int balance := 0;  // Shared resource
4
5      // Access method : withdraw
6      public method boolean withdraw (int amount)
7          precondition amount >= 0 // Assertion
8      {
9          if balance < amount then return false
10         else
11             {
12                 balance := balance - amount;
13                 return true;
14             }
15     }
16
17     // Access method : deposit
18     public method deposit (int amount)
19         precondition amount >= 0 // Assertion
20     {
21         balance := balance + amount;
22     }
23 }
```

Listing 3.1: A Monitor Template

The account is a monitor type. It contains one account balance to be shared between different tasks. The access to the balance is only granted through two methods (withdraw and deposit). Since declared as a monitor, the compiler normally assigns a semaphore with the deposit and withdraw methods to guarantee the mutual exclusion. In addition, conditional variable can be associated with the monitors. They act as a queue of the monitor with two extra procedures. Essentially, wait blocks the calling thread and places it at the queue; signal removes a thread from the queue and gives it access to the monitors.

The monitor is a flexible programming model that provides inter-process communication with distinction between synchronization and data communication. However, the monitor, is a passive entity. There is no mechanism in monitors that actively controls the scheduling of the tasks nor the order of accessing the shared data. The order of accessing shared resource in monitor is decided by the scheduler or in a predefined manner by the programmer. However, the synchronization methods incorporate a series of scheduling event that the programmers can intervene so that dynamic changes to the resource accessing order becomes possible. For example, when all resources are available, the control of the resource is passed back to the calling task. When a resource is released from an executing task, all tasks either waiting on semaphores or conditional variables are notified if the blocking condition is still true. If false, the waiting task becomes blocked.

3.3 Methodology

The idea of incorporating application-defined scheduling has attracted wide attention of scheduling researchers. The advantage of implementing application-defined scheduling is that the developers can have more flexibility to modify the scheduling policy in order to meet their application's needs without any changes to the kernel.

Linux has become one of the most popular OS in a short period since its release. Users have been attracted because of its flexibility and capability. [70] proposed a two level hierarchical scheduler framework to enhance its flexibility. The participation of application and system scheduler, namely the allocator and dispatcher, results in a mapping between the application and system tasks. The users can modify the attributes of new tasks and the routines of application sched-

ulers. The application schedulers are essentially high priority tasks collaborating with lower-level system schedulers. However, the predefined parameters of the tasks limit the scope of scheduling algorithms available to be implemented by the developers. Other algorithms requiring extra parameters (other than those that have been supported) is not applicable for implementation in their scheme.

This idea also promoted an alternative approach to implement the application scheduling algorithms as loadable kernel modules to Linux as proposed by [73]. At every scheduling event, the loadable kernel modules invoked by predefined APIs is executed to implement application-defined scheduling algorithms. It is fast and efficient but a bug in the application code can be catastrophic and the whole kernel can be affected.

Application-defined scheduling was developed further by [53] [54] to improve the reliability of the system and the ability to share resources between tasks. It is believed that task scheduling cannot be separated from task synchronization because the timing properties of the scheduling policy are closely linked to the synchronization protocols used[54]. In their proposal, application developers can implement their scheduler as a runnable task scheduled by the underlying system scheduler. There are two risks of running application-defined scheduler: 1) It is difficult to determine the relationship between the tasks priority managed by the application schedulers and the priority of the application scheduler itself. 2) The application schedulers are not free from race condition because multiple application schedulers may be running in parallel on different processors. The developers should obtain a systematic overview of the whole system and write well designed code to ensure the executions are all serializable. In the context of Ada, [53] extended the framework towards supporting application-defined resource sharing protocols. A similar strategy was adopted where two kinds of mutexes were introduced: System-scheduled mutexes and application scheduled mutexes. The system-scheduled mutexes are mainly used by application schedulers in terms of resource sharing. The application-defined protocols defines the behavior of the application-scheduled mutexes. However, the range of application-defined protocols is constrained by the current POSIX APIs: No_Priority_Inheritance, Highest_Ceiling_Priority (Immediate Ceiling Priority) or Highest_Blocked_Task (Basic Priority Inheritance).

Regardless of the various proposals available, the principle of incorporating

application-defined scheduling provides more flexibility from the system. The previous proposals all have constraints on the scope of protocols that the application developers can implement. The system can provide more flexibility by opening up lower system primitives to the application developers. However, direct interaction with low-level system system is risky because a little mistake of error in the user code can corrupt the whole system kernel.

The active application schedulers are troublesome to the extent that their interactions with the system schedulers must be carefully designed and verified. This thesis proposes to simplify the application-defined schedulers to be passive so that all scheduling decisions are passed onto the system scheduler. The system scheduler will drive all the interactions with different application-defined algorithms and should be reliable and effective. The motivation of incorporating application-defined scheduling is to encourage the application developers to implement their own algorithm through standardized APIs. The flexibility of the framework is greatly enhanced as no constraints are imposed on the range of protocols the users can implement. In the mean time, the reliability of the system is protected as no active decisions will be made by the application-defined protocols and only interactions through the standardized API are accepted.

3.4 A Framework for Multiprocessor Application-Defined Resource Control Protocols

The flexibility of the monitor can be enhanced to benefit resource sharing in multiprocessors. We adopt the approach supported by most real-time programming languages and operating systems shown as Figure 3.1. The first element of the approach requires a monitor control protocol to be associated with a particular monitor. This is already supported in the RTSJ and those OS supporting the POSIX pthread extensions. However, no application-defined protocols are allowed. The monitor control protocol will manipulate one or more locking primitives to achieve the required blocking semantics. Here, we define two such primitive locks, one where tasks spin waiting for the lock, the other where they are suspended.

The spin lock and queue lock are primitives provided by the underlying OS. They are all relied on a variable shared between tasks indicating the condition of

the lock. If the variable is set or the spin flag is raised, the spin lock is being held by the other task. The effectiveness of incorporating these primitives is evaluated in the next chapter.

On the programming language side, the concept of monitor is applicable to Ada, Java, C and the POSIX standard. The Ada protected object is essentially following the template of monitor depicted by Listing 3.1. The shared data is declared in the private section of the object specification. The protected function, entry and procedures are associated as the access methods. The synchronization of the methods are provided by the language semantics of the protected methods. RTSJ has similar facilities where the methods requiring mutual exclusion are marked with the keyword "synchronized". POSIX mutexes give a low-level API that supports monitors.

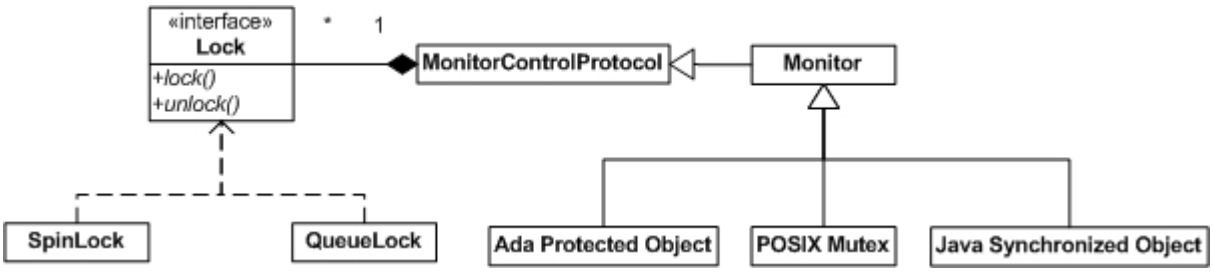

Figure 3.1: Basic Classes

Two primitive locks are provided to facilitate the implementation of the monitor control protocol:

Spin Lock There are two main characteristics of a spin lock. The priority at which the spinning occurs and the order at which the spinning tasks gain access.

Suspension Lock The main characteristic of a suspension lock is the queuing order. Here, we assume that no automatic priority inheritance is performed.

Whenever the run-time system (Ada runtime support system, Java virtual machine, or OS) needs to lock/unlock an object (an Ada protected object, Java monitor or an OS mutex) it calls out to the application and requests that it provides the lock protocol. This mechanism has to be integrated into the language

(rather than simply provided by an application abstract data type or class) as locking and unlocking are synchronization points and have defined effects on when updated memory locations become visible to an application. For example, both the Java and Ada memory models require that any shared data updated between locking and unlocking become visible to other threads/tasks in the program when unlocking completes. The details of the interface between the compiler/run-time and application will vary between languages, but the principles are the same. Figure 3.2 illustrates the approach.

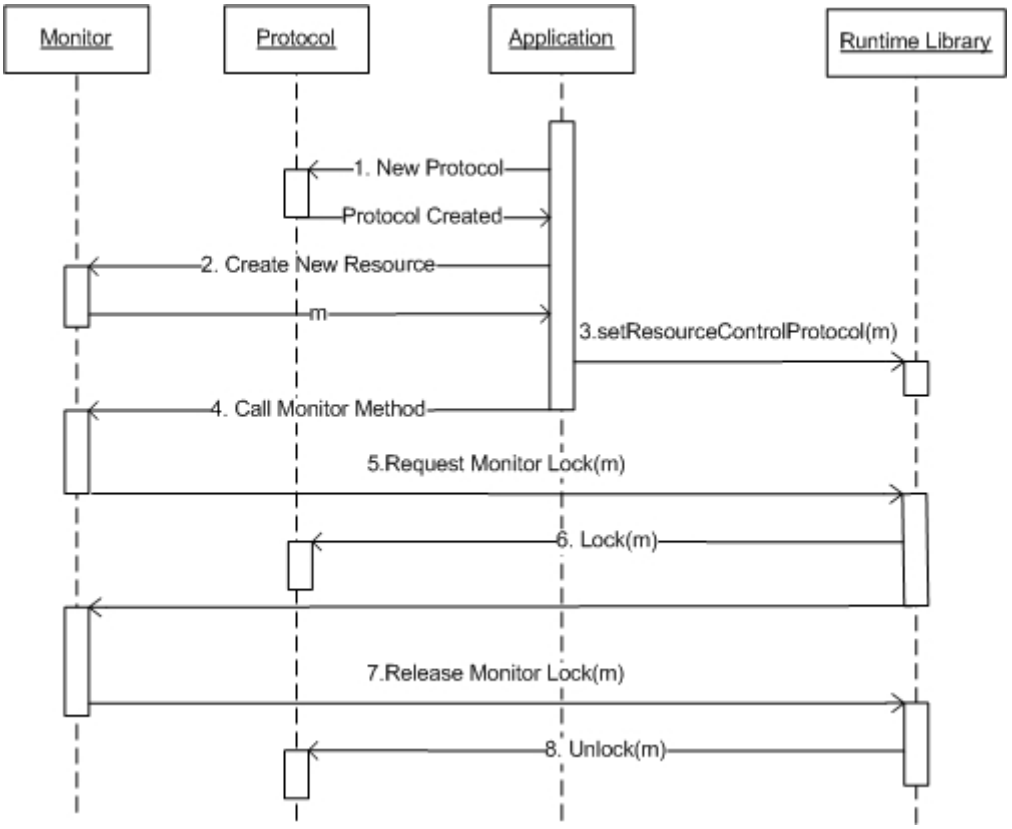

Figure 3.2: Integrating an Application-Defined Resource Control Protocol

1. The application creates an instance of its resource sharing protocol.

2. The application creates a resource encapsulated in a monitor

3. The application informs its runtime library support system that the created monitor should use the created resource control protocol

4. The application calls a method in the monitor

5. The monitor infrastructure code requests a lock from the runtime library

6. The runtime library forwards the call to the application's resource control protocol which provides the locking mechanism.

7. The monitor code indicates that it has finished and that the monitor should be unlocked.

8. The runtime library calls the unlock method in the applications resource control protocol.

When this framework is integrated with an OS (say a POSIX mutex) and used with C, then it is the responsibility of the applications programmer to ensure that any data shared between tasks that is accessed in the monitor code becomes globally visible to all threads that require access to it. This is because C [1] is a sequential language and its compiler is not aware of any multi-tasking.

3.5 Supporting the Framework in Ada

The typical control sequence of the framework is illustrated in Figure 3.3 for a dynamically created protected object. The model is essentially the same as that in Figure 3.2 except the compiler generates the initialization code.

The application in Figure 3.3 depicts the sequential code written by users. During the development phase, the users will write their resource sharing protocol by extending the Protected_Controlled type shown in Listing 3.2. When declaring a protected object, the users have to associate the application-defined protocol with the object. At compile time, the compiler will interpret the code and bound everything together with the protected object. At the run time, when a protected method is actually called by the application code, the compiled protected object

[1] This thesis assumes the use of C99. The recent varies of C11 [35] provided more support to encourage the use of multithreading and shared data.

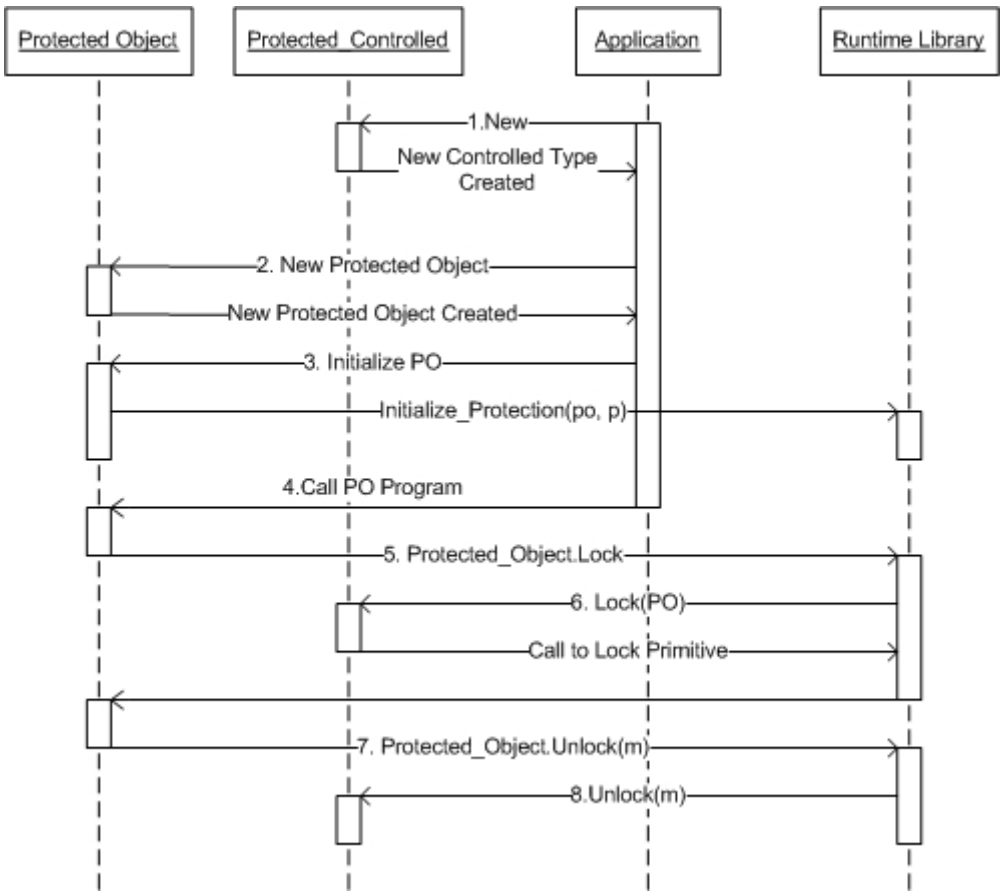

Figure 3.3: Integrating an Application-Defined Resource Control Protocol in Ada

code will pass on the call to the runtime library. The runtime library then redispatches the call to the application-defined method. The application-defined code can possibly use the original provided locking primitives. If this is the case, the OS lock API is called by the runtime library. If not, the application-defined methods will be called. When the lock is obtained, the control is passed back to the protected object where the critical section will be executed. Once finished, the unlock method of the protected object will be called. The runtime library will redispatch the call to the corresponding application-defined method automatically. In which case, the users can obtain the control over the next accessing task by manipulating the task order in the waiting queue. Similar to the locking methods, the users can make their own decisions on whether to use the OS unlocking primitives or not.

Following the design above, the following new language-defined package is introduced.

```
1  with Ada.Finalization; use Ada.Finalization;
2  with System; use System;
3  with Ada.Task_Identification; use Ada.Task_Identification;
4
5  package Ada.Protected_Object_Access is
6    type Lock_Type is (Read, Write);
7    type Lock_Visibility is (Local, Global);
8    type Protected_Controlled is new Limited_Controlled with private;
9
10   overriding procedure Initialize (C : in out Protected_Controlled);
11   overriding procedure Finalize (C : in out Protected_Controlled);
12
13   procedure Lock (       C : in out Protected_Controlled;
14                          L : Lock__Type;
15                          V : Lock_Visibility;
16                    Ceiling : Priority;
17                        Tid : Task_Id := Current_Task);
18   procedure Unlock (  C : in out Protected_Controlled;
19                      Tid : Task_Id := Current_Task);
20   private
21     -- implementation defined
22 end Ada.Protected_Object_Access;
```

Listing 3.2: Ada API for the Framework

The Ada application programmer can now create an instance of the Protected_Controlled type and write their own locking protocol. Ada provides a generic mechanism for mapping its high-level data types to the underlying machine level type. Of course, it provides default implementations of the abstractions, but it grants the programmer some control where it is necessary. These generic mechanisms are called aspect specifications. Here, we assume the introduction of a new

protected object aspect called Lock_Visibility; when set to true, this indicates that (i) the protected object can be accessed from more than one processor and (ii) the ceiling priority should be interpreted as an order requirement. A new locking policy, called User_Protected is also introduced. When this policy is enforced, every protected object can have an associated controller object which implements the access protocol.

When an object of Protected_Controlled is associated with a protected object, every time the Ada run-time system wants to lock/unlock the object, the associated lock/unlock procedure will be called. A protected object can be associated with the controller using the aspect specification. So, taking a simple PO type as an example, it would be written as follows:

```
1 protected type PO (PC : access Protected_Controlled) with
2     Locking_Policy => (User_Protected, Lock_Visibility => Global,
          Locking_Algorithm => PC) is
3     procedure P;
4     function F return Integer;
5 end PO;
```

Listing 3.3: Associate a Protected_Controlled type with a PO

where PC is a type that is derived from the Protected_Controlled type.

```
1 aspect_specification ::= with aspect_mark [=> aspect_definition] {,
    aspect_mark [=> aspect_definition] }
2
3 aspect_mark ::= aspect_identifier['Class]
4
5 aspect_definition ::= name | expression | identifier
```

Listing 3.4: Aspect Specification Grammar

The extension proposed by Listing 3.3 follows the Ada aspect specification grammar are shown by Listing 3.4. This extension must be included in the application source code at the compile time declaring the relationship between the protected object and its linked access methods. The with clause is an instruction to the compile to set up internal linkage to the compiled code so that the default links of lock and unlock methods are overwritten by the methods defined in Listing 3.2.

3.6 Supporting Framework in RTSJ

The RTSJ already directly supports the notion of a monitor control policy, encapsulated by the following class.

```
1  package javax.realtime;
2
3  public abstract class MonitorControl {
4
5      // constructors
6      protected MonitorControl();
7
8      public static MonitorControl getMonitorControl();
9      public static MonitorControl getMonitorControl(
10         Object monitor);
11     public static MonitorControl setMonitorControl(
12         MonitorControl policy);
13     public static MonitorControl setMonitorControl(
14         MonitorControl policy);
15     }
```

Listing 3.5: RTSJ Monitor Control Policy

This allows the programmer to specify whether a particular objects (monitor) lock should be subject to priority inheritance (the default) or priority ceiling emulation.

```
1  package javax.realtime;
2
3  public class PriorityInheritance extends MonitorControl {
4      public static PriorityInheritance instance ();
5      }
6
7  package javax.realtime;
8  public class PriorityCeilingEmulation extends MonitorControl {
9      public int getCeiling();
10     public static PriorityCeilingEmulation getMaxCeiling();
11     public static PriorityCeilingEmulation instance(int ceiling);
12 }
```

Listing 3.6: RTSJ Priority Inheritance and Priority Ceiling Emulation

To extend this model to support application-defined monitor resource control policies requires the introduction of lock-related primitive operations in the MonitorControl class, as illustrated below.

```
1  package javax.realtime;
2
3  public abstract class MonitorControl {
4      .. // as before
5
6      protected void lock();
```

```
7     protected void unlock();
```

Listing 3.7: RTSJ MonitorControl Class

Whenever the real-time JVM wishes to lock/unlock an application object, it calls the lock/unlock methods in its associated monitor control policy object. The default implementation of this method is a null operation. The overridden methods in the RTSJ-defined subclasses implement the appropriate policy. An application that wishes to perform is own lock policy can extend the MonitorControl class and override the methods. Internal JVM locks would not be delegated in this way.

Essentially, the aforementioned approach requires the JVM to delegate responsibility for implementing the application monitors of specified objects to the application. The sequence diagram for Java would not be fundamentally different from the Ada one. For example, when using a synchronized statement, the MonitorEnter byte code is executed. The JVM would delegate this operation to the application-defined code.

3.7 Supporting the Framework in a POSIX-Compliant OS

Processes were the schedulable entities before Linux 2.6. In order to be POSIX compatible and improve its threading facilities, Linux introduced new threading library to support full multi-threading at the kernel level. The NPTL (Native POSIX Thread Library) was firstly introduced in Red Had 9 with full implementations of synchronization primitives for inter-process communication [40].

Version 2.6 onwards Linux benefit from this property of futex and implemented new POSIX compliant blocking primitives. It is worthwhile restating the blocking primitives supported by POSIX:

```
1 // Pthread Mutex Lock
2 int pthread_mutex_init(pthread_mutex_t *mutex, const
      pthread_mutexattr_t *mutexattr);
3 int pthread_mutex_lock(pthread_mutex_t *mutex);
4 int pthread_mutex_unlock(pthread_mutex_t *mutex);
5 int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
6                 int protocol);
```

```
7
8  // Pthread RW Lock
9  int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
10                 const pthread_rwlockattr_t *restrict attr);
11 int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
12 int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
13 int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
14
15 // Pthread Spin Lock
16 int pthread_spin_destroy(pthread_spinlock_t *lock);
17 int pthread_spin_lock(pthread_spinlock_t *lock);
18 int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Listing 3.8: POSIX locking primitives API

As shown in Listing 3.8, the POSIX supports three locking primitives: Mutex, RW and Spin lock. The mutex is a suspension based locking where blocked tasks are suspended for unavailable resources. The RW lock participated the acquiring tasks into readers and writers. The shared resources are assigned to the two groups of tasks in turn. The spin lock sets the blocked tasks spinning for unavailable resources.

The pthread mutex type is defined as a struct in C. It is composed of various low level internal mutex variables including the __lock for futex, __count for reference counting and __owner to check the ownership of the mutex etc. All these variables are integer type whose bits are individually set and resit in the mutex operations. The application and system developers are allowed to declare the following kinds of mutex:

PTHREAD_MUTEX_NORMAL normal plain mutex.

PTHREAD_MUTEX_RECURSIVE recursive mutex where recursive locking is permitted. The owner thread must call pthread_mutex_unlock enough times to set the mutex free.

PTHREAD_MUTEX_ERRORCHECK mutex with error checking which will return error code EDADLK when recursive calling is made.

The developers can set the type of the mutex by passing the pthread attributes to the pthread_mutex_init function. At the initialization stage, the first two bits of __kind variable will be set indicating the specific type of the mutex. When the corresponding locking primitive is called for the mutex, the type of the mutex is firstly determined as follows:

```
1
2  int __pthread_mutex_lock (mutex)
3  pthread_mutex_t *mutex;
4  {
5   assert (sizeof (mutex->__size) >= sizeof (mutex->__data));
6
7   unsigned int type = PTHREAD_MUTEX_TYPE (mutex);
8
9   if (__builtin_expect (type & ~PTHREAD_MUTEX_KIND_MASK_NP, 0))
10  return __pthread_mutex_lock_full (mutex);
11
12  if (__builtin_expect (type, PTHREAD_MUTEX_TIMED_NP)
13  == PTHREAD_MUTEX_TIMED_NP)
14  {
15  simple:
16  /* Normal mutex. */
17  ...
18  }
19 else if (__builtin_expect (type == PTHREAD_MUTEX_RECURSIVE_NP, 1))
20  {
21  /* Recursive mutex. */
22  ...
23  return 0;
24  }
25
26
27  }
28  else if (__builtin_expect (type == PTHREAD_MUTEX_ADAPTIVE_NP, 1))
29  {
30    /* Adaptive Mutex */
31    ...
32  }
33 else
34 {
35  assert (type == PTHREAD_MUTEX_ERRORCHECK_NP);
36  /* Error Check Mutex */
37  ...
38  }
39   return 0;
40 }
```

Listing 3.9: pthread_mutex_lock structure for dealing with different mutexes

Within each condition of the if statement in Listing 3.9, the reference counter (__count variable) is incremented. The __lock is sent as the key to the futex functions to update the system so that future calls to pthread_mutex_lock to this unavailable resource will be suspended. When the resource has been finished, the corresponding pthread_mutex_unlock will be called where the reference counter and the futex will be reset. The implementation of the mutex was written in C. The definition and implementation of mutex incorporates the flexibility for exten-

sions. For example, some of the high end bits of __kind variable remain largely unused. It left a scope for the developers to define new mutex type which can be recognized by the "if statements" in the pthread_mutex_lock function. This scope of extension is also found at the resource sharing protocols side in POSIX.

Implementing the mutex alone is not enough. The POSIX defines two resource sharing protocols to avoid deadlocks by default:

PTHREAD_PRIO_NONE No resource sharing protocol is supported

PTHREAD_PRIOR_INHERIT Standard priority inheritance protocol is supported. The calling task should be executed at the highest priority of itself or the priority of the highest priority thread waiting on the shared resources initialized with this protocol.

PTHREAD_PRIO_PROTECT Priority ceiling protocol is supported. The calling task should be executed at the highest priority of all tasks initialized with this protocol.

The application or system developers can specify the use of resource sharing protocol by calling the pthread_mutexattr_setprotocol method. The function accepts a pointer to a particular mutex attribute and one of the above protocols as parameters. The protocol passed by the developer will be assigned to the mutex kind variable of the mutex attribute.

```
1  int pthread_mutexattr_setprotocol (attr, protocol)
2    pthread_mutexattr_t *attr;
3    int protocol;
4   {
5    /* Sanity checks are omitted */
6
7    struct pthread_mutexattr *iattr = (struct pthread_mutexattr *) attr
         ;
8
9    iattr->mutexkind = ((iattr->mutexkind & ~
         PTHREAD_MUTEXATTR_PROTOCOL_MASK)
10   | (protocol << PTHREAD_MUTEXATTR_PROTOCOL_SHIFT));
11
12   return 0;
13  }
```

Listing 3.10: Set the resource sharing protocol in POSIX

In the function depicted by Listing 3.10, the 28th and 29th bits of the mutexkind variable is set to indicate the specific algorithm used for resource sharing.

This setting takes effect in the pthread_mutex_full_lock function. The function shifts the 28th and 29th bits of the mutexkind variable and analyzes which algorithm was specified by the variable. The default setting is priority inheritance. However, if priority ceiling is specified, the correct priority assigned to the blocked task will be calculated by the __pthread_tpp_change_priority function at line 23 shown by Listing 3.11:

```
1  static int __pthread_mutex_lock_full (pthread_mutex_t *mutex)
2  {
3   int oldval;
4
5   switch (PTHREAD_MUTEX_TYPE (mutex))
6   {
7     case PTHREAD_MUTEX_ROBUST_RECURSIVE_NP:
8     /* More cases for mutex type*/
9       ...
10    case PTHREAD_MUTEX_PP_ADAPTIVE_NP:
11    {
12        int kind = mutex->__data.__kind & PTHREAD_MUTEX_KIND_MASK_NP;
13
14        oldval = mutex->__data.__lock;
15
16        int oldprio = -1, ceilval;
17        do
18        {
19            int ceiling = (oldval & PTHREAD_MUTEX_PRIO_CEILING_MASK)
20            >> PTHREAD_MUTEX_PRIO_CEILING_SHIFT;
21            /*Ceiling sanity checks are omitted*/
22
23            int retval = __pthread_tpp_change_priority (oldprio,
                  ceiling);
24            ceilval = ceiling << PTHREAD_MUTEX_PRIO_CEILING_SHIFT;
25            oldprio = ceiling;
26
27            /* Updated the lock depending on the specific protocol
                  used */
28
29            oldval
30            = atomic_compare_and_exchange_val_acq (&mutex->__data.
                  __lock,
31                #ifdef NO_INCR
32                 ceilval | 2,
33                #else
34                 ceilval | 1,
35                #endif
36                 ceilval);
37          ...
38        }
39        while ((oldval & PTHREAD_MUTEX_PRIO_CEILING_MASK) != ceilval)
              ;
40
41        /*Sanity checks are omitted*/
```

```
42              default:
43             /*default action*/
44          }
45          return 0;
46      }
```

Listing 3.11: POSIX implementation of PI and PCP

The pthread_mutex_full_lock is a comprehensive version of pthread_mutex_lock making all locking calls to the kernel space. The simplified version of pthread_mutex_full_lock utilizes user space reference counting to reduce the number of such calls to improve average system performance. The structures of these versions are the same. Mutexes are proceeded through different routines depending on their types. For example, as depicted by Listing 3.11, mutex declared as the pthread_mutex_pp_adaptive_np type is managed by resource sharing protocols. When this type of mutex is called on locking, the mutex lock and the pre-defined ceiling protocol are retrieved (line 12-20). These parameters will help the __pthread_tpp_change_priority function (line 23) to calculate the appropriate priority assigning to the calling task. In order to make the decision, the __pthread_tpp_change_priority scans through the tasks waiting in the blocking chain. If priority inheritance is used, the priority of the highest waiting priority task will be assigned. If priority ceiling is used, more scans will be made in order to determine the ceiling of the resource.

The implementation of the pthread mutex locks are well structured and flexible. The high end bits of the mutexkind variable of pthread_mutexattr_t are available for the application developers to incorporate new mutexes. The implementation of the POSIX library in glibc (implementation version of NPTL) is well designed and structured. The 32-bits integer mutexkind variable allows the application developers to extend for application-defined protocols. The application developers can set the 28th and 29th bit of mutexkind to specify an application-defined protocol. Also, a new application-defined mutex type can be introduced to the pthread mutex library. When the application-defined mutex is called within the pthread_mutex_lock function, the application-defined resource sharing protocol will be called where the priority management and scheduling can be directly intervened by application developers. Inspired by the RTLinux approach, the flexible resource sharing framework can introduce a pre-defined function pointer to the if statements of the pthread locking methods. The actual function implementing the

locking behaviours is only retrieved at the run-time when the corresponding function pointer is executed. Most of the interactions are remained in user space. The application developers can use the well designed locking primitives provided by the framework instead of making direct system calls to the kernel. The interaction of the multiprocessor flexible resource sharing framework is illustrated further by Figure 3.4:

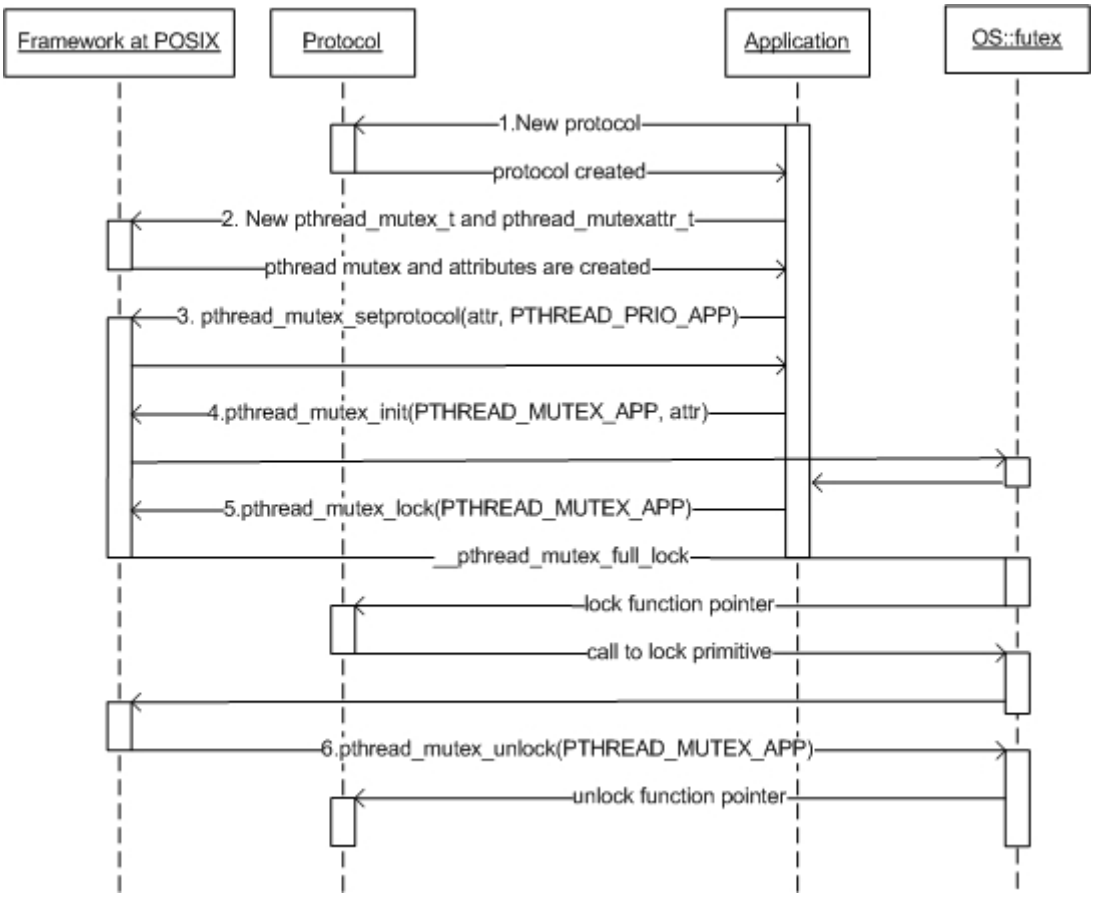

Figure 3.4: Supporting the Framework in POSIX

The semantics of the interactions is summarized as follows:

1. A new protocol is created from the framework.

2. The application declares a new pthread_mutex_t typed variable creating a mutex object and new pthread_mutexattr_t typed object.

3. The newly declared protocol is associated with the mutex by passing PTHREAD _PRIO _APP to the pthread_mutex_setprotocol.

4. The mutex object is initialized by calling the pthread_mutex_init function. PTHREAD_MUTEX_APP indicates this mutex is an application-defined mutex which should be managed by application-defined routines.

5. The pthread_mutex_lock is called on the application-defined mutex (PTHREA_MUTEX_APP). Since this is an application-defined mutex, the function pointer to the application-defined routines is invoked by the underlying OS so that the application-defined locking routines are retrieved and executed.

6. When the execution in the resource is finished and the pthread_mutex_unlock is called at the run-time, the application-defined unlock routines will be called through the associated function pointer.

If no application-defined routines are assigned, the framework can always use the default locking protocols available at the POSIX library. The above sequence diagram is only designed to illustrate the interactions in principle. The interactions are meant to show the applicability of the framework to the POSIX compliance systems. It is highly acceptable that the implementations will have different approaches in declaring the application-defined mutex, using the bits of mutex integers and implementing different locking primitives etc.

3.8 Supporting Application-Defined Condition Synchronization

The locks are implemented to prevent two tasks accessing the same shared resource at the same time. Only the task that has obtained the lock may proceed to executed the critical section. The waiting task will be blocked due to the fact that the lock has already been obtained by the other task. This single ownership of the lock is effective in protecting the integrity of shared resources from race condition. However, the synchronization between tasks sometimes relies on the condition of the shared resources.

Let us consider the following pseudo code example for condition synchronization:

```
1  Buffer::Consumer(Lock)
2  {
3      Lock->acquire();  // Acquire the lock first;
4      While (Lock->counter == 0)
5      {
6          Lock->release();
7          Lock.WaitOnNotify();
8          Lock->lock();
9      }; // Remove data from the buffer
10     Lock->counter--;
11     Lock.NotifyAll();
12     Lock->release();
13 };
14
15 Buffer::Producer(Lock)
16 {
17     Lock->acquire();
18     While (Lock->counter == MAXLMT)
19     {
20         Lock->release();
21         Lock.WaitOnNotify();
22         Lock->lock();
23     };  // Add data to the buffer
24     Lock->counter++;
25     Lock->NotifyAll();
26     Lock->release();
27 }
```

Listing 3.12: Pseudo Code for Buffer Operation

Listing 3.12 depicts two methods of a buffer. The consumer takes an element on each iteration from the buffer. if no element is left in the buffer, it will set itself suspended waiting for the notification from the producer for new elements. The producer acquires the permission to access the buffer and inserts a new element on each iteration. If the buffer is full, the producer will wait for the elements to be consumed and the corresponding notification from the consumers. The synchronization between the consumer and the producer does not only depend on the lock for the buffer but also the number of elements in the buffer.

This is a typical condition synchronization example. The waiters, namely the consumer in this case, acquires the lock to the shared resource first. It then checks the condition of the shared resource. If the current situation does not satisfy the requirement, it will be suspended for further notification from other tasks which have been granted the rights to change the condition of the shared resource. Before

suspension, the acquired lock will be released to grant the access to other tasks. If the resource condition has been changed and satisfies the requirements, the waiters will be resumed and the lock for the resource will be immediately acquired.

The flexibility of the framework incorporates the scope of supporting application-defined resource sharing algorithms in condition synchronization in Ada. Although Ada does not have direct implementation of the wait and notify methods, the condition synchronization is already provided by the protected entry facilities in Ada. It is worthwhile restating the principles of Ada protected entries. The protected entry is an access method to the data in protected objects. The protected entry is guarded by a boolean expression known as the barrier. The calling task must acquire the lock to the protected object first. If the access is granted, the task will evaluate the barrier. If the barrier is evaluated to be true, the task can therefore proceed into the critical section of the resource. Otherwise, the task is blocked on the barrier and the acquired lock of the object is released. The barrier is evaluated when there might have been changes imposed to the parameters of the barrier or a task has finished its execution in related protected methods with read/write locks. Following this design, the buffer example can be rewritten in Ada as protected entries:

```
1  protected body Buffer is
2      entry Consumer when counter > 0 is
3      begin  -- Remove data from the buffer
4          counter := counter - 1;
5      end Consumer;
6
7      entry Producer when counter < MAX_LMT is
8      begin -- Add data to the buffer
9          counter := counter + 1
10     end Producer;
11     ...
12 end Buffer;
```

Listing 3.13: Buffer with Protected Entries

The interpretation of the code in Listing 3.13 will have the barrier evaluation encapsulated by a pair of lock and unlock of protected objects. In normal situations, the default locking protocols are invoked at the run-time. If the application-defined protocol is specified at the aspect specification of the protected objects, the barrier evaluation will then be encapsulated by a pair of application-defined methods where necessary priority changes and task management can be implemented. At run-time, the application-defined methods, as methods of Protected_Controlled

type, will be executed by the compiler instead of the default ones.

3.9 Summary

Inspired by the literatures of resource sharing in multiprocessor systems, a flexible resource sharing framework is the key focus of this chapter. The framework is designed to lift some of the restrictions of resources sharing protocols on multiprocessor systems. The application developers are capable of defining a suitable algorithm for their application needs. This is motivated by the fact that the optimality of the resource sharing algorithm is based on individual application scenarios.

This principle of involving application participation in system operations is a common method of resolving the problem of adding new protocols to underlying OS. POSIX adopted a different but more concrete approach so that application defined scheduler procedures are directly passed to low level systems. The application-defined methods were called passively by the kernel schedulers and associated methods are invoked when the corresponding event occurred at the low level.

Although Ada is chosen to be the target language for the experiments of this research, the framework is widely applicable to different programming languages. The framework is designed to facilitate the monitor structure which is widely accepted by popular programming languages which support the monitors approach. Flexibility is a key concern of the framework where different locking primitives and resource sharing protocols can be integrated. The framework itself can work seamlessly with the underlying system scheduler to deliver the requirement of application-defined resource sharing protocols to lower systems. As we can see, the framework principles are widely accepted by underlying OS and programming languages. The framework itself is then adaptive to different languages and operating systems.

Chapter 4

Implementing the Ada Framework

This chapter discusses the prototype implementation of the framework. The expectation is that the framework should impose acceptable overheads to the applications in exchange for the flexibility for applications to implement the most suitable multiprocessor resource sharing protocol. The result should improve the application predictability and performance.

The implementation chooses Ada as the experimental language as it is the only ISO standard, object-oriented, concurrent, real-time programming language. Ada was specifically designed for large, long-lived embedded applications where reliability and efficiency are essential. It was developed in the 1980s to supersede hundreds of programming languages used by US Department of Defence. Because of its safety-critical features, it is now used not only for military applications, but in safety critical commercial projects [25]. For example, the European Train Control System (ETCS), a system for high speed train signal control, was written in Ada.

The framework implementation comprises a large number of elements. Firstly, the extension to the aspect specification should be introduced where the application developers can integrate their algorithm with the framework. Secondly, the runtime library should be refined and extended to be integrated with the framework. Finally, the implementation should follow the original language semantics and the definition of the protocols. The following sections implement the frame-

work in Ada with an estimation of the runtime overheads.

4.1 GNAT Structure

The implementation of the Ada framework comes in three parts: the aspect specification, the Protected_Object_Access type and the runtime integration. The pseudo code for the aspect specification and the introduction of Protected_Object _Access type is given in Chapter 3. The implementation is then focused on the integration of the framework with Ada. In order to do so, the Ada compiler (GNAT) must recognize the aspect specification depicted by Listing 3.3 where appropriate structure can be established for runtime use. Before explaining further, short introduction to the GCC GNAT is given:

The compiler used by this thesis is for Ada 2005 and its corresponding GNAT [1]. GNAT is an open source complete Ada compiler integrated into the GCC compiler system. A compiler is essentially a translator. It scans a set of instructions written in plain text and translates it into a set of machine instructions executable by the underlying hardware [6]. Such a translation should maintain an important property, platform independent execution. In order to do this, a series of interconnected steps must be followed. Peterson[45] offers an abstraction of the GCC compiling process:

1. Lexical analysis happens at the beginning of the process. It reads the characters and spaces in order to parse the meaningful terms (symbols, numbers and punctuation etc.) for later use.

2. The parser conducts further analysis of the output of the lexical analysis and determines the relationships between the terms. The output is represented in a tree structure and sent to the back end compiler.

3. This is a special feature of GCC for optimization purpose. The parse tree is translated to a pseudo assembly code called a Register Transfer Language (RTL).

[1] GNAT is available for download at: http://www.gnu.org/software/gnat/

4. After receiving the RTL output, the back end compiler starts the process of optimizing the code. Unreachable code, groupings of similar code and further optimizations are carried out at this stage.

5. The optimized code is translated into assembly language of the target machine.

6. The assembler is translated further into executable object code.

7. An executable program is created by combining the executable object code with the object files of the runtime libraries.

The compiling process of Ada is no different to the above abstract process except the Ada libraries are linked with the final executable code in the end. The API proposed by Listing 3.3 enforces new specification (Locking Policy, Locking_Visibility, Locking_Algorithm used in the with clause) to be accepted by Ada. This imposes changes to the whole compiler: 1) The lexical analysis and parser should be updated to recognize those key terms; 2) Distinct tree nodes must be established to represent the link of the data structure; 3) Rules of optimization must be refined; 4) The linker should recognize the updated runtime library so that necessary framework library code can be effectively linked at runtime.

As described above, the final executable code compiled by the GNAT is linked against the object code of the pre-implemented runtime library. In Ada, the GNU Ada Runtime Library (GNARL) is an implementation of the Ada tasking model which is closely linked to the performance of our framework. The GNARL collaborate with GNU Low-Level Interface (GNULLI) to reconciliate the Ada tasking model with the standard POSIX thread services. The layout of the GNARL is shown by Figure 4.1:

The GNULLI abstracts the implementation of the services that GNARL needs from the host operation system. It is not upward compatible because the GNULLI interface is changed when used by a different OS. For example, at the bootstrapping process of GCC, the auto configuration routine will detect the target machine's hardware. The GNULLI will be changed at the bootstrapping process depending on the underlying system (Linux, Sparcs or Solaris etc). The main responsibility of GNULLI is the abstraction of the POSIX interfaces. For example, the pthread

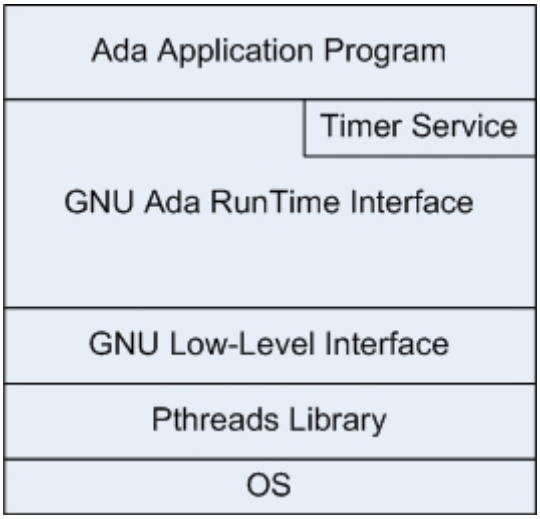

Figure 4.1: GNARL Components [68]

mutex type is an abstract of Task_Primitives.Lock which is an instantiation of the pthread_mutex_t type.

The intent of GNARL is to be an upward compatible extension of earlier attempts to converge on a common real time system interfaces for Ada83 [68]. It attempts to implement the Ada tasking model as directly as possible with the assistance from GNULLI. It isolates as much of the tasking semantics, thus reducing the interactions with application level code . The control flow of the GNARL is procedural. The compiler communicates with the tasking runtime system through function or procedure calls without direct reference to the data structures of the tasking runtime system.

4.2 How GNAT Implements Protected Object

4.2.1 The Semantics of Protected Objects

Protected Objects is a synchronization mechanism of Ada. It provides synchronization based on a data object rather than between tasks. A protected type provides safe resource sharing to its encapsulated data. Protected procedures allow mutual exclusive access to the data encapsulated in the protected object so

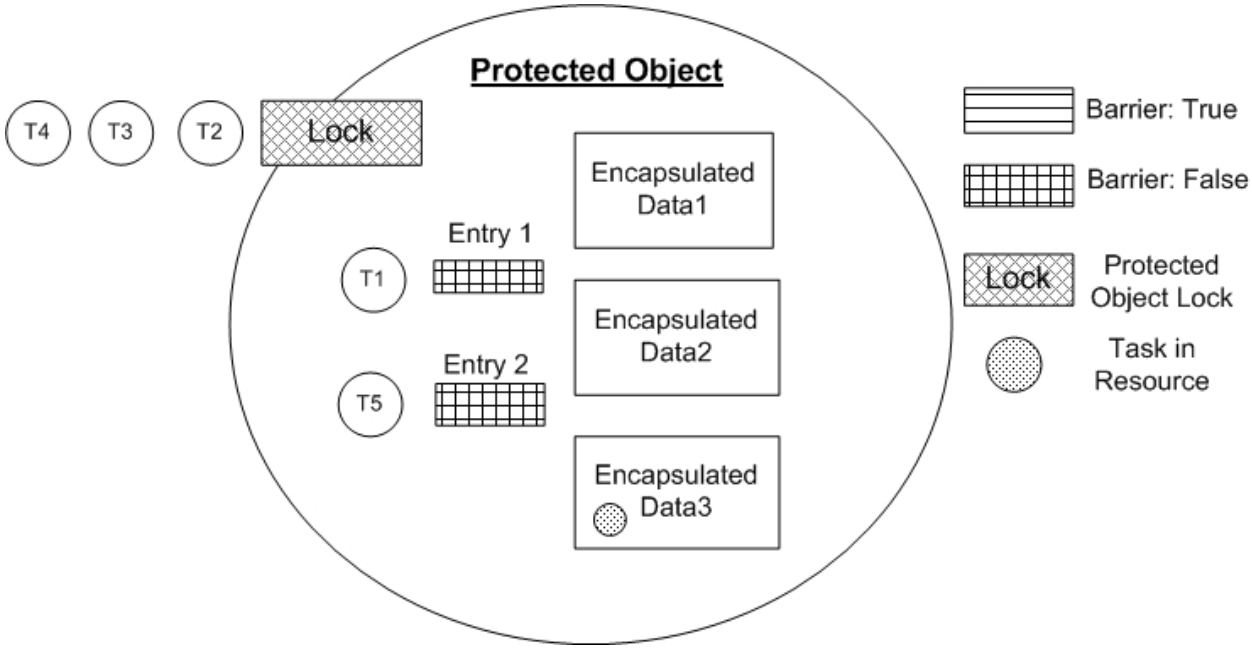

Figure 4.2: The Protected Object of Ada

that only one task is allowed to access the data at once. Protected functions provide concurrent read-only access to the encapsulated data. The protected entry is similar to the protected procedure with an extra barrier. The calling task must evaluate the boolean barrier to true in order to proceed to the protected data. If the boolean barrier is evaluated to false, the calling task is blocked.

Figure 4.2 illustrates the structure of a protected object in Ada. The large circle represents the protected object with its associated lock. Tasks acquiring the encapsulated data need to obtain the lock. If there is a procedure or entry call inside, no other tasks are allowed to enter. If the protected object is occupied by a function, other function calls can proceed in. This is because protected functions can provide read-only parallel access to its shared data. T1 and T5 are not yet able to access the data because they have to evaluate the Entry boolean barrier to true. Due to the state of the encapsulated data, the barrier of Entry 1 is evaluated to false. Therefore, T1 is blocked. T1 can only be resumed if the barrier is evaluated to true made by another procedure or Entry call on the same object. The tasks blocked on the entries are organized in the corresponding Entry queues.

Depending on the queuing policy configured by the programmer, such tasks can either be priority or FIFO ordered.

The following example explains the protected object further:

```
1  with Ada.Text_IO;use Ada.Text_IO;
2
3  procedure Crossroads is
4  ------All Specifications-------
5  task type Train is
6  end Train;
7
8  protected type Junction is
9      entry Approach;
10     procedure Passed;
11     function Status return Boolean;
12 private
13     Bar_Up: Boolean:= True;
14 end Junction;
15
16 ------All Implementations--------
17 protected body Junction is
18     entry Approach(item: integer) when Bar_Up is
19     begin
20         Bar_Up:=False;
21         Put_Line("A train is approaching");
22     end;
23
24     procedure Passed is
25     begin
26         Put_Line("The junction is free again");
27         Bar_Up:=True;
28     end;
29
30     function Status return Boolean is
31     begin
32         return Bar_Up;
33     end;
34 end Junction;
35
36 TrainA, TrainB, TrainC: Train;  -- Three Customers
37 AJunction : Junction;          -- One Waiter
38
39 task body Train is
40 begin
41       AJunction.Approach;
42       if AJunction.Status = True then
43           Put_Line("Collision!");
44       Passed;
45 end Train;
46
47
48 begin
49 end Crossroads;
```

The code illustrates a scenario where three trains are approaching the same junction. Only one train can cross the junction at one time. On approach, a train should send a message to the junction indicating its desire for using the junction. If denied, the train should stop and wait on the junction to be free. If the junction is free, which means the bar is up, the train can proceed directly to the junction and the bar is to prevent other trains entering the junction. The bar will be lifted again if the current approaching train has gone through the junction, this signals that the junction is ready for the next approaching train. Whenever a train is in doubt of the condition of the junction, the status function is always available. Multiple trains can consult the status of the junction at any one time.

Although safe mutual exclusive access to shared data is provided, deadlock is not prevented by using protected objects alone. Let us consider the following example. We have two tasks T1 and T2 sharing two protected objects R1 and R2. At first, T1 and T2 lock R1 and R2 respectively. If T1 continues to Lock R2, the two tasks become deadlocked. This is because R2 will not be released until T2 finishes executing R1. Also, R1 will not be released until T1 finishes executing in R2. At this point, neither T1 nor T2 can proceed. The two tasks are therefore deadlocked. This problem can be avoided by using resource sharing algorithms (e.g: Accessing the Protected Object at the ceiling priroity).

4.2.2 GNAT Implementation of Protected Objects

Protected Objects present a complex problem in implementation since there are various forms of locking behaviour which all have different semantics. The protected entry grants the access to the calling task depending on the barrier evaluation and the states of the private data of the protected object. Ada requires this to be checked after each protected procedure or entry call (AARM 9.5.3 pars 22 and 22.a). This requires extra routines to be implemented so that the barriers of the protected entries are checked for any passed tasks. If so, the corresponding bodies of these tasks are executed. This is also known as the proxy model which allows the tasks to continue with other waiting tasks on open barriers [68]. The compiler normally translates all barrier expressions into boolean functions and all entry bodies into procedures. The following simplified protected objects demonstrates the translation of the protected objects in a layering manner.

```
1  procedure Entrytest is
2      protected type PO is
3            entry Wait;
4            entry Signal;
5            procedure ident;
6            function Is_Open return Boolean;
7         private
8             ...
9         end PO;
10
11        protected body PO is
12
13           entry Wait when True is
14           begin
15             null;
16           end Wait;
17
18           entry Signal when Is_Open is
19           begin
20             null;
21           end Signal;
22
23           procedure Ident is
24           begin
25             null;
26           end Ident;
27
28           function Is_Open return Boolean is
29           begin
30              return True;
31           end Is_Open;
32
33        end PO;
34
35        obj : PO;
36
37    begin
38        obj.Wait;
39        obj.Signal;
40        obj.Ident;
41    end;
```

Listing 4.1: A Simple Protected Object

Listing 4.1 describes a simple protected object comprising four access methods. The following Listing 4.2 is a schematic intermediate representation of protected object and its associated access methods which might be generated at the GNARL level. This code is to be extended and instantiated by GNULLI.

```
1  type PO is record
2      Object : Protection(Num_Entries => N);
3      ...
4      -- additional states of object
5      ...
6      end record;
7
8  procedure Ident (
9      obj : in out PO;
10     ...
11     --additional parameters for uninterpereted data etc.
12     ...
13 ) is
14     Dummy : Boolean;
15 begin
16     Defer_Abortion;
17     Lock(obj.Object);
18     ... -- in critical section
19     Service_Entries(obj, Dummy);
20     Unlock(obj.Object);
21     Underfer_Abortion;
22 end Ident;
23
24 function Is_Open (
25     obj : PO;
26     ...
27     --additional parameters for uninterpereted data etc.
28     ...
29 ) is
30     temp : Boolean;
31 begin
32     Defer_Abortion;
33     Lock_Read_Only(obj.Object);
34     ... -- Read Only operations in critical section
35     temp := expression;
36     Unlock(obj.Object);
37     Undefer_Abortion;
38     return temp;
39 end Is_Open;
40
41 procedure Signal (
42     obj : PO;
43     ...
44     --additional parameters for uninterpereted data etc.
45     ...
46 ) is
47     Dummy : Boolean;
48     Pending_Serviced : Boolean;
49 begin
50     Defer_Abortion;
51     Lock(obj.Object);
52     Protected_Entry_Call (obj.Object, 1, Uninterperted_Data,
           Simple_Call);
```

```
53          -- The barriers are evaluated in this function
54      Service_Entries (obj.Object, Pending_Serviced);
55      Unlock (obj.Object);
56      -- Other House keeping routines;
57      end Signal;
58
59  -- This is the routine to serve the waiting entry calls on open
        barriers
60  procedure Service_Entries (
61      ID  : Task_Id;
62      obj : in out PO;
63      Pending_Servicesd : out Boolean
64  ) is
65      Uninterpreted_Data : System.Address;
66      subtype PO_Entry_Index is Protected_Entry_Index range 0..2;
67      Barriers : Barrier_Vetor (1..2);
68      E : PO_Entry_Index;
69      PS : Boolean;
70      Cumulative_PS : Boolean := False;
71  begin
72      Queuing.Selected_Protected_Entry_Call (Id, obj, obj.Entry_Call);
73      -- Queue the current entry call to the queue first
74
75      loop
76          Barriers(1) := ... -- Barrier expression;
77          Barriers(2) := ... -- Barrier expression;
78          Next_Entry_Call (obj.Object, Barriers, Uninterpreted_Data, E)
                ;
79          case E is
80              when Null_Protected_Entry => exit;
81              when 1 =>
82                  ... -- sequence of statements
83                  Complete_Entry_Body(PS);
84              when 2 =>
85                  ... -- sequence of statments
86                  Complete_Entry_Body(PS);
87              end case;
88              ... -- Other housekeeping routines
89          end loop;
90      end Serivce_Entries;
```

Listing 4.2: PO intermediate interpretation at GNARL

Listing 4.2 is an abstraction of the actual code of GNARL. It is not possible to fully reprint the full code of the protected object library. The code was abstracted from the actual implementation to demonstrate the collaboration between GNARL and GNULLI in the context of implementing the high level language semantics. At the high language semantics level, ARM (Ada Reference Manual) requires that the barriers are to be evaluated before the actual protected entry call. In GNARL, the barriers are evaluated in the Protected_Entry_Call function where the barrier

function address is retrieved from the uninterpreted_data and executed. Also, Ada dictates that the protected entry cannot terminate if there are entries waiting on open barriers. As mentioned in previous section, Ada adopted the proxy model for barrier evaluation as shown by Service_Entries. When a protected entry is called, the calling task is firstly appended to the protected entry queue. Depending on the queuing policy, the next entry call will be chosen. The index of the entry, together with the address of the pass in parameters, is retrieved to execute the entry call of the next waiting task on the open barrier. When the body of the entry is complete, the waiting task will be notified by GNARL calling the Complete_Entry_Body function. The waiting task will be waked up. These tasks can then continue their execution right after the entries. The Service_Entry routine repeats this progress until the Next_Entry_Call cannot find an eligible entry body for execution. The integration of Ada protected entries is discussed in a greater detail in Chapter 6 .

All these protected objects operations rely on the comprehensive support of GNULLI. The blocked tasks at closed barriers are suspended and a context switch occurs for those tasks whose entry body has been executed by the operating task. All tasks in Ada are managed by the Ada Task Control Block (ATCB) which is mapped to the Thread Control Block (TCB) by GNULLI to POSIX. Similarly, GNARL locks the the protected procedure to prevent race condition. The GNULLI function Write_Lock will be called with the associated protected object lock retrieved. The Write_Lock function will call the pthread_mutex_lock function to instruct the OS to lock the resource down.

4.3 Framework Implementation

The previous sections have explained the composition of GNAT, where the framework will be implemented. This includes the runtime redispatching at the GNARL level to integrate application-defined calls and APIs to application developers in Listing 4.2. Also, instrumentation at the GNULLI level is necessary in order to incorporate the application-defined locking primitives. A full implementation would also involve reprogramming the front end compiler. As explained in Section 4.1, the front end compiler recognizes the lexical words with a parser and sets up a tuple tree with lexical analysis. Necessary lexical tokens, as explained later in this section, will be required to be recognized by the front end compiler. This work

volume is large and beyond the scope of this thesis. It is therefore not feasible to support a full implementation.

As an alternative to the full implementation, a simulation approach assumes that the lexical tokens introduced by the framework were recognizable by the compiler. As discussed, the GCC compiler translates all source code into executable object file and linked with runtime library. This is irrespective of the changes to the hardware platform and the sort of programming language used. The last readable text before the object file is the assembler code. It represents the optimized final execution order of the code in terms of explicit function calls and register moves are shown. The simulation method adopted for the framework implementation follows the expectation that two code with identical compiled assembly code has the same semantics at high language level. Therefore, one code can be an excellent approximation of the other, which is also known as simulation. The implementation of the framework is actually simulated with application code collaborating with the extended runtime library so that the front end compiler remains intact. The simulation code is expected to generate identical assembly code as if it was fully implemented and the front end compiler was instrumented.

4.3.1 Effectiveness of the Simulation Method

The following experiments were conducted to check the effectiveness of the simulation. It is expected that the test application code will generate the same assembly code as the original Ada runtime library for protected objects. It is worthwhile restating the example code shown in Listing 4.1. Each method represents a particular category which is expected to have different assembly code at the lower level. When compiled, the actual assembly code generated is as follows (only function calls are shown):

```
1  entrytest__po__identN.3182
2
3    system_soft_links_abort_defer(%rip), %rax
4    -- set the calling task so that it cannot be aborted during the
         call;
5
6    system_tasking_protected_objects_entries_lock_entries
7    -- locks the protected procedure ident
8
9    -- the code of the actual protected entry is placed here
10
```

```
11    system_tasking_protected_objects_operations_service_entries
12    -- unlocks the protected procedure indent
13
14    system_soft_links_abort_undefer(%rip), %rax
15    -- set the calling task so that it can be aborted during again
```

Listing 4.3: Pseudo Assembly Code for procedure ident in code 4.1

```
1  entrytest__po__signal_B6b.2371:
2
3      __gnat_rcheck_CE_Access_Check
4      -- sanity check
5
6      entrytest__po__is_openN.3188
7      -- call the barrier function
8
9      system_standard_library_abort_undefer_direct
10     -- set the calling task so that it can be aborted
11
12     system_tasking_protected_objects_operations_communication_blockIP
13     -- handles the communication with underlying system
14
15     system_tasking_protected_objects_operations_protected_entry_call
16     -- call the runtime library routine to lock the PO
17
18     -- the code of the actual protected entry is placed here
19
20     system_tasking_protected_objects_operations_complete_entry_body
21     -- unlocks the PO
```

Listing 4.4: Pseudo Assembly Code for Entry Signal in code 4.1

Following the introduced semantics, a protected object procedure in Listing 4.1 is translated into a series of assembly code leading by entrytes_po_identN3̇182. The assembly code comprises a pair of lock and unlock procedure calls to the System.Tasking.Protected_Objects package. The actual operation in the critical section of the resource is thus protected. Before this, the calling task is set to be non-abortable. It is reset after the resource has been released. The protected entry calls was compiled with no significant difference to the protected procedures. The corresponding assembly code is shown by Listing 4.4. The entry barrier clause is interpreted as a function call to entrytest_po_is_open. When the barrier is cleared, the assembly code follows the pattern of protected procedure calls encapsulating the critical section execution with lock and unlock calls to System.Tasking.Protected_Objects package.

The interpretation is inline with the semantics of the language defined in the Ada reference manual and the GNARL library. At the end of the test, the actual

compilation of the test application code generates the same assembly code as the original Ada library.

4.3.2 Framework API Proposal

The full implementation of the resource sharing framework proposed in this thesis is deployed at the runtime library of Ada. A new Ada package is introducd to be instantiated and reloaded by the application developers in their source code. The lock and unlock functions were designed to define the semantics of the application-defined resource sharing protocols.

```
1  protected body Ada.Protected_Object_Access is
2      overriding procedure Initialize (C : in out Protected_Controlled)
           is
3      begin
4          -- initialize the data structure for the controller
5      end;
6
7      overriding procedure Finalize (C : in out Protected_Controlled)
           is
8      begin
9          -- perform any necessary finalization
10     end;
11
12     procedure Lock ( C : in out Protected_Controlled;
13                      L : Lock_Type;
14                      V : Lock_Visibility;
15                Ceiling : Priority;
16                Task_Id := Current_Task ) is
17     begin
18         -- default locking protocol here
19     end;
20
21     procedure Unlock ( C : in out Protected_Controlled;
22                        Tid : Task_Id := Current_Task ) is
23     begin
24         -- default unlocking protocol here
25     end;
26 end package Ada.Protected_Object_Access;
```

Listing 4.5: Ada.Protected_Object_Access

The application defined lock and unlock procedures are integrated with the original Ada protected objects. The runtime library is instrumented to call the correct routine associated with the calling Protected_Controlled object. The System.Tasking.Protected_Object package is re-defined as follows.

```
1  with Ada.Finalization; use Ada.Finalization;
2  with System; use System;
3  with Ada.Task_Identification; use Ada.Task_Identification;
4  with Ada.Protected_Object_Access; use Ada.Protected_Object_Access;
5
6  package System.Tasking.Protected_Object is
7
8      -- The compiler will generate code to
9      -- the class-wide locking/unlocking procedures
10
11     procedure Lock ( C : access Protected_Controlled'Class;
12                      L : Lock_Type;
13                      V : Lock_visibility;
14                      Ceiling : Priority;
15                      Tid : Task_Id);
16     procedure Unlock (C : access Protected_Controlled'Class;
17                       Tid : Task_Id);
18     private
19         type Protected_Controlled is new Limited_Controlled with null
               record;
20 end System.Tasking.Protected_Object;
```

Listing 4.6: Ada.Protected_Object

The body of this package is illustrated below:

```
1  package body System.Tasking.Protected_Object is
2
3      procedure Lock ( C : access Protected_Controlled'Class;
4                       L : Lock_Type;
5                       V : Lock_Visibility;
6                       Ceiling : Priority;
7                       Tid : Task_Id := Current_Task ) is
8      begin
9          -- perform any generic processing here;
10         -- for example, checking for ceiling violations
11
12         Lock (C.all, L, V, Ceiling, Tid);
13         -- redispatches to the appropriate locking routine.
14
15         -- perform any generic processing here
16     end;
17
18     procedure Unlock ( C : access Protected_Controlled'Class;
19                        Tid : Task_Id := Current_Task ) is
20     begin
21         -- perform any generic processing here
22
23         Unlock (C.all, Tid);
24         --redispatches to the appropriate unlocking routine
25
26         -- perform any generic processing here
27     end;
28
```

```
29 end System.Tasking.Protected_Object;
```

Listing 4.7: Ada.Protected_Object Body

In the body of System.Tasking.Protected_Object package, all lock and unlock functions are redispatched at line 12 and 23. The compiler will search the corresponding lock and unlock routine with the same set of parameters and the invisible tag. Those routines are then called instead of the original ones. However, if the application developers have not defined their own routines. The calls will be redispatched to the original Ada routines.

4.4 Framework Overheads

In order to determine the overheads of using the framework, experiments were performed with the Ada Core GNAT system.

4.4.1 Ada Real-Time Clock Facilities

In the following tests, all measurements use the Ada.Real_Time package. In order to coordinate the program's execution with the natural time of the environment, the Ada.Real_Time package provides access to a hardware clock that approximates the passage of real time. The package implements a monotonic regular clock for real time application needs. The package supports long term application execution up to 50 years. The minimum tick value of the underlying clock is restricted to be less or equal to 1 millisecond. The smallest amount of time representable by the time unit must be no greater than 20 milliseconds. The high resolution time is returned by the function Clock(). The return of the Clock() is a duration of the time [74] shown as follows:

```
1 function Monotonic_Clock return Duration is
2     TS : aliased timespec;
3     Result : Interfaces.C.int;
4 begin
5     Result := clock_gettime
6         (clock_id => CLOCK_REALTIME, tp => TS'Unchecked_Access);
7     pragma Assert (Result = 0);
8     return To_Duration (TS);
9 end Monotonic_Clock;
```

Listing 4.8: System.Task_Primitives.Operations.Monotonic_Clock

The To_Duration function converts the underlying C compatible time to Ada program accessible types on each returns of the call. This is the inaccuracy of the measurement which will reduce the effectiveness of the time stamping. In our experiment, in absence of interrupts and preemption, the execution time of the code is measured by sampling the start and the end time. A possible time drifting is demonstrated by the following diagram:

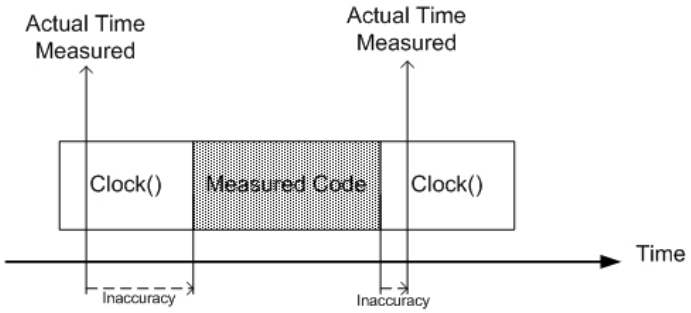

Figure 4.3: Time Drifting of Clock()

The inaccuracy at the right of Figure 4.3 represents the time cost to convert the high resolution time value returned by the underlying POSIX API. The inaccuracy at the left of Figure 4.3 indicates the elapse of time to the time when the actual underlying hardware clock is stamped. It is expected that the overall drift suffered by the measurement code is at most one entire Clock() function. To get more precise estimation of the measurement code, the affect of time drift must be minimized from the measurement. It is left over to the later implementation and evaluation work mainly because it imposes less significant difference to the overall performance figure.

4.4.2 Original Ada Protected Object Overhead

The previous sections explained the interactions and details for a full implementation of the framework. However, such an implementation requires enormous amount of work and it is beyond the scope of this research. The purpose of this simulated implementation, is to model the existing Ada facilities and show that the code is effective in multiprocessor systems, is to obtain an estimate on the order of magnitude of the overhead introduced by the framework into the application code.

Therefore, estimating the execution time of the code instead of acquiring exact cost of introducing the API is the concern of this research. Although cumbersome implementation details is avoided, the simulation implementation still requires a full implementation of the framework at the runtime library level. The interactions are then changed and addressed in the following sections.

Consider the following simple Ada program, which calculates the time taken to make 1000 null protected procedure calls.

```
1  with Text_IO; use Text_IO;
2  with Ada.Real_Time; use Ada.Real_Time;
3
4  procedure Main is -- Original Ada program
5      T0 : Time;
6      Elapsed : Time_Span;
7      type Elapsed_Array is array (1..1100) of Time_Span;
8      Execution_Time : Elapsed_Array;
9
10     protected PO is
11         procedure Enter;
12     end PO;
13
14 begin
15     for i in 1..1000 loop
16         T0 := Clock;
17             for j in 1..1000 loop
18                 PO.Enter;        -- Make the protected call to PO
19             end loop;
20         Execution_Time (i) := Clock - T0;
21     end loop;
22
23     for i in 1..1000 loop
24         Put_Line("Elapsed "&Duration'Image(To_Duration(Execution_Time
               (i))));
25     end loop;
26 end Main;
```

Listing 4.9: Protected Object Full Implementation Test Code

The evaluation platform is a 2.66 GHz Intel Core 2 Duo processor with 4GB 1067 MHz memory and Linux ubuntu 2.6.38-11 SMP kernel. As with any Linux application, the evaluation application suffers from system interrupts and other system overheads. The measurement will attempt to exclude the incidental data and obtain an approximation to the true value of the execution cost of the code.

In order to achieve this, the protected object call is encapsulated in a "for" loop call, which contains 1000 iterations. A pair of time snapshots is only taken for every 1000 iterations. This is because the accuracy of the data is too small to

IRQ Name	CPU1	CPU2	Comment
RES	68453	74467	Rescheduling interrupts
CAL	23503	5418	Function call interrupts
TLB	48526	49659	TLB shootdowns

Table 4.1: */proc/interrupts* before 10 seconds testing period

IRQ Name	CPU1	CPU2	Comment
RES	68455	74472	Rescheduling interrupts
CAL	23505	5419	Function call interrupts
TLB	48527	49661	TLB shootdowns

Table 4.2: */proc/interrupts* after 10 seconds testing period

sustain from the impact of the time drifting. Therefore, the execution time of a large number of iterations is taken to minimize the effect of such impact. Also, the expense of interrupts when a common issue for measuring application performance in Linux. The interrupts are sent by the underlying system in the context of changing scheduling and timing requests. It is random in terms of maintaining routine services at a lower level of the system. The Linux provided the reference facilities to count the total number of interrupts since installation. The counter file is used as an approximation to estimate the impact of system interrupts on the evaluation program, using "*cat/proc/interrupts*; *sleep*10; *cat/proc/interrupts*". This command counts the number of system interrupts received by the kernel in the period of 10 seconds with the reference to the interrupts statistics. The result is shown by table 4.1 and table 4.2:

In the 10 seconds period, there are 7 rescheduling interrupts, 3 function call interrupts and 3 multiprocessor memory interrupts received by the kernel. The associated interrupt handlers are then invoked to preempt the current executing task. This process sometimes can be expensive. For example, the TLB shootdowns break the mapping between the virtual and physical memory of local processors, which, will invalidate the mappings of remote processors in turn. The interrupt handlers of TLB shootdowns is expensive and has a wide effect on all processors in the system [8]. Owing to its nature, the execution of interrupt handlers can be easily identified as a spike in the execution time curve.

Program	Median Time for 1000 protected procedure in milliseconds	Inter-Quartile Range in miliseconds
Original Ada Program	0.777	0.001

Table 4.3: Overhead of the Original Ada Protected Object Call

It is not of the concern of this research to accurately estimate the execution time of the interrupt handlers. The mandate is to reduce the impact of interrupt handling and obtain an effective estimation of the measured code. The noise data can then be identified and excluded. The inter-quartile range is an effective statistical method for this purpose. It measures the statistical dispersion of the sample data, being equal to the difference between the upper and the lower quartiles. A smaller number indicates a less dispersed data set. The performance of the original Ada protected object depicted by Listing 4.9 is shown by Table 4.3:

As the measurement code showed, an average time taken to complete 1000 protected object entry call is 0.78 millisecond. The inter-quartile range indicates that the data set is converged. There is only a few pieces of deviated data arriving at an exceptional large number. These numbers put into a perspective to be compared against the overhead of the framework.

4.4.3 A Simulated Framework Implementation

The overhead imposed by the simulated framework is twofold: 1) The simulation cost 2) The actual overhead of executing the framework.

The evaluation and implementation of the framework is firstly set to estimate the cost of simulation. Certain changes have been imposed to the original design of the framework:

In Figure 4.4, the application is solely acting as the event driver. When compared with Figure 3.3, the simulation eliminates the Protected_Controlled type. It interacts with the Protected_Object package and the runtime library directly. This interaction is a simplified version of the simulated approach because the runtime redispatching is not included in application-defined protocols. The application program directly calls the default lock method through the Protected_Object.lock interface. Control is passed back to the application once the lock is obtained.

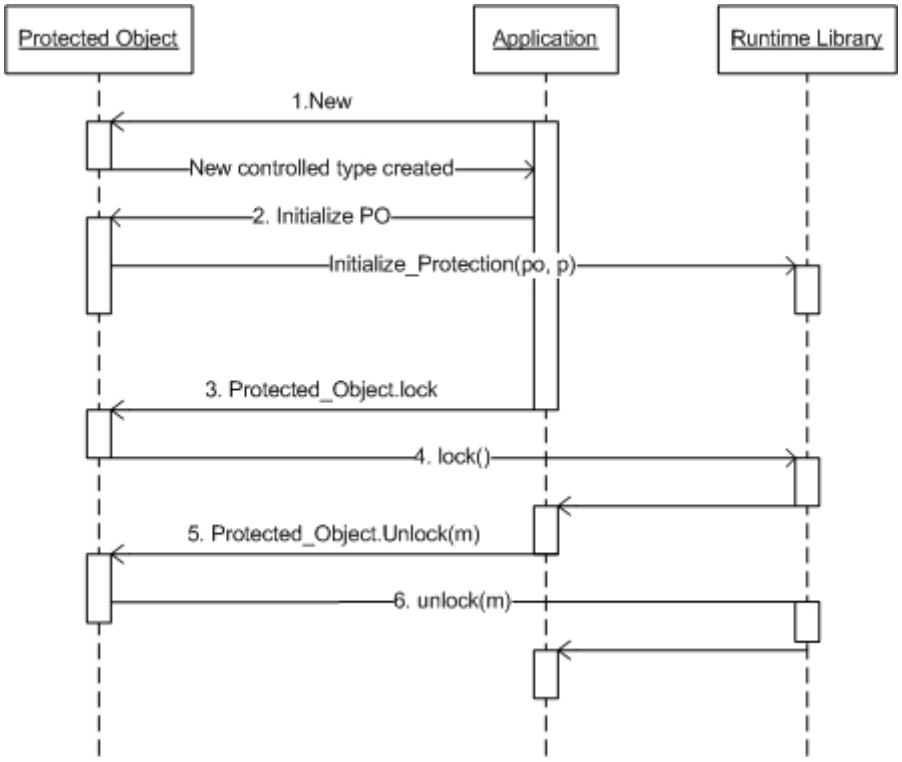

Figure 4.4: Simulated Implementation Interaction

When finished, the application calls the unlock interface in the Protected_Object package. The interface invokes the unlock method provided by the underlying run-time library. This design estimate the execution time of the simulation method. The source code can be found in Listing 4.10:

```
1  with Text_IO; use Text_IO;
2  with System.Tasking.Initialization; use System.Tasking.Initialization
       ;
3  with System.Soft_Links; use System.Soft_Links;
4  with System.Tasking.Protected_Objects;
5  use System.Tasking.Protected_Objects;
6  with Ada.Real_Time; use Ada.Real_Time;
7  with Ada.Task_Identification; use Ada.Task_Identification;
8
9  procedure Main is  -- hand compiled code
10     package SPO renames System.Tasking.Protected_Objects;
11
12     T0 : Time;
13     Elapsed : Time_Span;
14     type Elapsed_Array is array (1..1100) of Time_Span;
15     Execution_Time : elapsed_array;
16
17     obj : aliased SPO.Protection;
18
19 begin
20     SPO.Initialize(Obj'Unrestricted_Access);
21
22     for i in 1..1000 loop
23         T0 := Clock;
24         for j in 1..1000 loop
25             Abort_Defer.all;
26             SPO.Lock(Obj'Unrestricted_Access);
27             null;
28             SPO.Unlock(Obj'Unrestricted_Access);
29             Abort_Undefer.all;
30         end loop;
31         Execution_Time(i) := Clock - T0;
32     end loop;
33     ...
34 end Main;
```

Listing 4.10: Protected Object Simulation Code

The object on line 17 is the data representation of the protected object, which is initialized on line 20. The call to the protected procedure is mapped to the calls on lines 25-29. The results of executing the hand compiled program is listed in Table 4.4:

The simulated code is slightly more efficient than the original code. There is some, essentially, null code that is generated by the compiler for house keeping purposes. Since the interactions with the Ada default protocol are eliminated,

Program	Median Time for 1000 protected procedure in milliseconds	Inter-Quartile Range in miliseconds
Hand Compiled Program	0.670	0.002

Table 4.4: Overhead of the Simulated Ada Protected Object Call

the surrounding maintenance code on the entry and exit of protected procedure calls are not present in the simulation code. The simulation, however, offers an approximation of the original Ada protected object call. This approximation is an effective baseline measurement to be compared with the performance of the refactored program with the framework integrated.

The refactored program imposes necessary changes to Ada in order to incorporate application-defined resource sharing protocols through our framework. The concept of polymorphism is a powerful tool for this purpose. Ada supports the idea of polymorphism by introducing the concept of class. For each tagged type T, there is an associated type T'Class. T'Class, also called "the class wide type of T", comprises the union of all types in the tree of derived types rooted at T [39]. Owing to this routine, a subprogram can define one or more parameters as a type of the form T'Class. Different types are thus created under the same type but different parameters. Dynamic dispatching is the process provided by Ada at runtime to determine which routine to call. Ada will check the parameters of the incoming calls and searches the invisible tag associated of the T'Class for a perfect match. The implementation of the framework utilized this concept by passing in a class-wide type of Protected_Controlled. The compiler will determine at runtime which lock routine to call in terms of this protected controlled object. In which case, the application-defined lock and unlock methods will be automatically called upon its associated protected object.

The full simulated measurement is based on the above refactored Ada library. The updated library will redispatch each incoming lock and unlock calls to its associated routine instead of directly using the original primitives. Extra execution expense is expected to emerge in this routine. However, the application-defined resource sharing protocols can benefit from this low cost integration. In order to maintain the consistency of the measurement data, same measurement code is used in this case (as shown in code 4.10). The test result is illustrated by Table

Program	Median Time for 1000 protected procedure in milliseconds	Inter-Quartile Range in miliseconds
Hand Compiler Program	0.700	0.002

Table 4.5: Overhead of Simulated Ada Refactored Protected Object Call

4.5:

In general, the data obtained shows that the average cost of completing an protected object call to a refactored Ada program is 0.03 milliseconds greater than the hand compiled program. It is inline with the expectation that the redispatching facility will impose extra cost to the runtime library. Although the measurement data suggested that the refactored program is more efficient than the original Ada program, the refactored program, in fact, has longer execution time than the original program. As mentioned in previous paragraphs, the refactored program follows the simulation principle where certain house keeping routines are omitted. Taken into account the extra 0.1 milliseconds saved, the refactored program his 0.03 milliseconds execution time longer than its equivalent original Ada program for 1000 protected object calls.

4.5 Summary

Following the previous discussions, this chapter examines the framework through practical implementation in Ada. Due to the fact that programming at low level involves different components, the framework implementation is likely to vary from the scope of the modification and the efficiency of the code execution. Those components, like GNAT and protected objects of Ada, were examined at the beginning of this chapter.

The complete implementation of the resource sharing framework requires the full support of the GNAT compiler so that the new specification introduced by the framework need to be recognized by the parser and the semantics imposed by the framework can be implemented by the runtime library. The framework was fully implemented at the library level using the simulation approach. This will not mitigate the benefit of extra flexibility imposed by the framework with minor runtime overheads.

As any application developer would expect, the interface must be light and efficient. The framework is evaluated and measured from its efficiency against the original Ada protected objects. The implementation evaluation comprises three experiments found that the overhead imposed by the framework is acceptable. More specifically, a 0.03 milliseconds increase for every 1000 protected object calls. This is equivalent to a 4% increase of the total execution time of the call. We can therefore conclude that the added flexibility of allowing application defined resource sharing protocols justifies the small increase in runtime overheads.

Chapter 5

Expressive Power of the Ada Framework

In this chapter, we present various multiprocessor resource sharing protocols and their implementations to demonstrate the expressive power of the flexible multiprocessor resource sharing framework in Ada. These protocols were presented and explained with concrete examples in Chapter 2. It is worth reproducing the summary of multiprocessor resource sharing protocols in Table 5.1 to remind us of their properties.

The multiprocessor resource sharing algorithms were characterized by different design principles. Some protocols, like MPCP, are designed to minimize the remote blocking by using suspension locks. Some protocols emphasize on efficiency by using the spin locks instead of suspension locks. Other protocols are more closely linked to the underlying hardware structure of multiprocessor systems by deploying hierarchical queues for resource sharing. Perhaps, it is the diverging views on the solution of sharing resources in multiprocessor systems which have caused the segregated solutions. The algorithms considered in this section are summarized by Table 5.1.

It becomes a challenge for the framework to support all of these protocols. With well diversified requirements, the protocols do not have a unified resource acquiring model. They may attempt to lock at the start of their execution as well as at any random time point. They may require multiple instances of different queuing policy on different processors. It is also highly possible that they may be

Protocol	Scheduling	Resources	Nested Resources	Access Priority	Queueing
MSRP	Partitioned	Yes	No	Non Preemptive	Spins in a FIFO queue
FMLP	Partitioned	Yes	Group Lock	Non Preemptive for short; Suspension for long	Short: Spins in a FIFO queue; Long: Suspends in FIFO queue
MPCP	Partitioned	Yes	No	Ceiling	Suspends in a priority ordered queue
OMLP	Global	Yes	Group Lock	Inheritance for global	Suspends in FIFO and Priority-ordered queues
Clustered OMLP	Clustered	No	Group Lock	Priority Donation	Suspends in a FIFO queue

Table 5.1: Resource Sharing Protocols Evaluated in this Chapter

interested in modifying scheduler decisions at runtime. This chapter, therefore, evaluates the framework against the possible requirements of the protocols with designated operating scenarios. It is expected that the application developers can implement application defined protocols by purely using our framework and the programming language facilities.

The structure of this chapter is as follows:

In section 5.1, we summarize our overall evaluation approach. In section 5.2, we consider MSRP, which requires tasks to spin while blocked on a global resource. This is the approach suggested by the Ada Reference Manual. However, our framework allows approaches based on suspension locks as well. In section 5.3, we consider MPCP which is a simple suspension based protocol. In section 5.4, we consider FMLP, which combines suspension with spin. Finally, we consider two versions of OMLP. The basic OMLP is described in section 5.5. The clustered OMLP is a more complex algorithm has more interactions with the scheduler than the other algorithms. This protocol is the most challenging to support in our proposed framework.

5.1 Evaluation Approach

The prototype implementation of the framework was not designed to understand the effectiveness of the underlying resource sharing algorithm, but to investigate the expressive power of the framework. It is expected that the framework can operate effectively enough to support different multiprocessor resource sharing algorithms at various levels in adaptive ways. For example, different resource sharing algorithms may be viewed as a black box which requires different combination of input and output. The framework should be adaptive in itself so that it can be integrated with the algorithm and the application program seamlessly. When the application developers modifies the input to the framework, the framework should be adaptive so that no recompilation is needed. The evaluation therefore should enumerate as many combinations as possible to test the features of the framework in various scenarios. The expected result of the evaluation is that the framework can provide enough support for the application-defined protocol candidates in the context of Ada. The application developers should be able to implement their algorithm using only the framework and the high level language facilities.

Evaluation Task Model The task model used for the evaluation work is important. It determines the behaviour of the evaluation tasks which will lead to different behaviour of the resource sharing algorithms at run time. In order to maintain the objectivity of the test, the following general tasking model is used:

An evaluation task can have as many as five states. At the creation stage, a task will be created and its parameters will be initialized. However, it is not runnable as it has not been officially released to the scheduler. After creation, it can be released to the scheduler and enters the running state at some late time according to its scheduling eligibility. Once the execution starts, an evaluation task can acquire shared resources at any time and execute in the critical section of the resource after obtaining the lock. When its execution in the resource is finished, the task will release the resource and continue its normal execution for some time. It will finally enter the termination state once all its execution has finished.

It is worthwhile mentioning that an evaluation task can spend an indefinite amount of time between any two adjacent states. This is quite common in the real

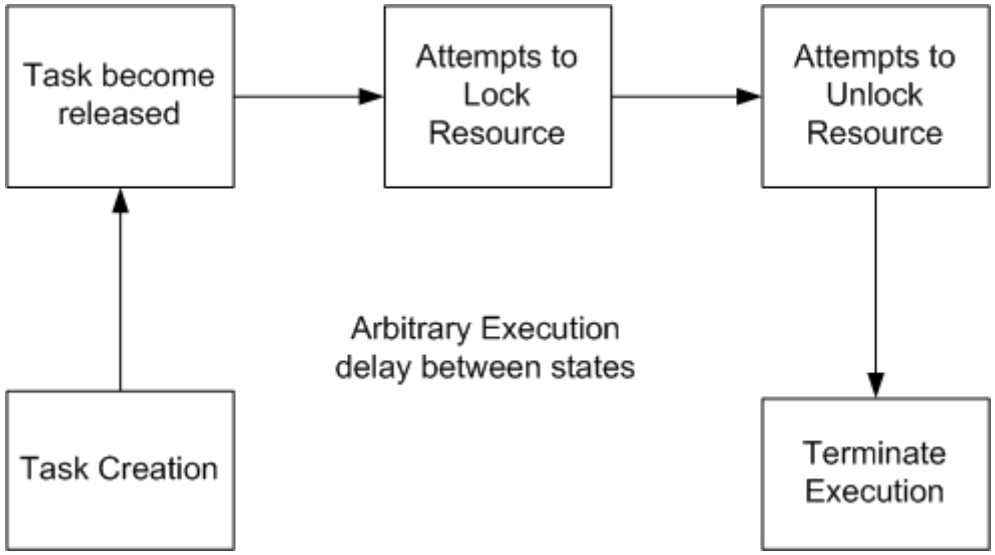

Figure 5.1: Evaluation Task Model

world where a task, for example, involves a heavy math computation but only uses shared resource when it wants to write the result to the output. Alternatively, when a task operates heavily in memory operations, it may spend a minimum amount of time spent in normal calculations and incur minimum execution delays between any of the non critical states. Adjusting the execution delays between the states can dynamically simulate different semantics of the application environments demanding different corresponding behaviours of the framework and the resource sharing algorithms.

In order to improve the presentation quality, the main test drive program of the following protocols are not individually presented as they all have the same structure. An example test drive program is listed as below to illustrate how the task model is implemented for evaluations:

```
1  pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
2  with Some_Locking_Protocols; use Some_Locking_Protocols;
3  procedure Main is
4
5      Start_Time : Time;
6
7      procedure ComputationWork (c : integer) is
8              temp : integer :=0;
9          begin
10             for j in 1..c loop
```

```
11                  for i in 1..100000 loop
12                      temp := temp + 1;
13                  end loop; temp := 0;
14              end loop;
15          end;
16      pragma inline(ComputationWork);
17
18  -- ================== Resource Definition ==================
19  protected type Resource (pri:Priority; V:Locking_Visibility; PC:
        access Some_Locking_Protocol)
20      with Priority => pri, Locking_Visibility => V, Locking_Algorithm
          => PC is
21          procedure Access_Resource Access_Resource(t1, t2, t3:Integer;
                R:access Resource);
22          procedure Nested_Access(t1, t2, t3:Integer; R:access Resource
                );
23  end Resource;
24
25  protected body Resource is
26      procedure Access_Resource(t : Integer) is
27      begin
28          ComputationWork(t);
29      end;
30
31      procedure Nested_Access(t1, t2, t3:Integer; R:access Resource) is
32      begin
33        ComputationWork (t1);
34        R.Access_Resource(t2);
35        ComputationWork (t3);
36      end;
37  end Resource;
38
39  -- ================== Task Definition ==========================
40  -- Task acquiring no resource
41  task type No_Resource_Worker (
42      pri : Integer;
43      res : access Resource;
44      affinity : Integer;
45      d   : Integer) is
46      pragma Priority(pri);
47  end No_Resource_Worker;
48
49  task body No_Resource_Worker is
50  begin
51      STPO.setaffinity(STPO.self, affinity); -- affinity
52      delay until Start_Time;
53      ComputationWork(d);
54      stop;
55  end
56  -- Task acquiring single resource
57  task type Single_Resource_Worker (
58      pri : Integer;
59      res : access Resource;
60      affinity : Integer;
```

```
61      d1, d2, d3: Integer) is
62      pragma Priority(pri);
63    end Single_Resource_Worker;
64
65    task body Single_Resource_Worker is
66    begin
67      STPO.setaffinity(STPO.self, affinity); -- affinity
68      delay until Start_Time;
69      ComputationWork(d1);
70      res.Access_Resource(d2);
71      ComputationWork(d3);
72      stop;
73    end Single_Resource_Worker;
74
75    -- Task acquiring nested resource
76    task type Nested_Resource_Worker (
77      pri : Integer;
78      res1, res2 : access Resource;
79      affinity : Integer;
80      d1, d2, d3, d4, d5: Integer) is
81      pragma Priority(pri);
82    end Nested_Resource_Worker;
83
84    task body Nested_Resource_Worker is
85    begin
86      STPO.setaffinity(STPO.self, affinity); -- affinity
87      delay until Start_Time;
88      ComputationWork(d1);
89      res1.Nested_Access(d2, d3, d4, res2);
90      ComputationWork(d5);
91      stop;
92    end Nested_Resource_Worker;
93
94    R1 : aliased Resource(Some_Locking_protocol'access);
95
96    procedure driver is
97      T1 : Single_Resource_Worker (1, R1'access, 1, 2, 2, 3);
98    begin
99      null;
100   end driver;
101
102 begin
103     Start_Time := Clock + milliseconds(1000);  -- Set the start up
            time.
104     driver; -- Driver procedure to start the tasks
105 end Main;
```

Listing 5.1: Original Evaluation Main Program

Listing 5.1 shows the structure of the test programs. The actual programs are controlled so that tasks are released in an appropriate order for the purpose of evaluation. In Chapter 2, each protocol was illustrated with a particular scenario. The test program were engaged to follow theses scenarios.

As mentioned in Chapter 4, the full implementation of the framework require modification to the front end Ada compiler which is beyond the scope of this research. The simulation approach was used instead. The evaluation test follows this principle and derived the compiled evaluation drive program as follows:

```
1  pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
2  -- with necessary packages
3
4  procedure Main is
5      package STPO renames System.Task_Primitives.Operations;
6
7      Start_Time : Time;
8
9      procedure ComputationWork (c : integer) is
10             temp : integer :=0;
11         begin
12             for j in 1..c loop
13                 for i in 1..100000 loop
14                     temp := temp + 1;
15                 end loop; temp := 0;
16             end loop;
17         end;
18     pragma inline(ComputationWork);
19
20   R1 : aliased Global_MPCP(3, false, 1, 1);  -- for example
21
22   -- ================== Resource Definition ==================
23   type Resource (PC : access Protected_Controlled'Class) is
24       null
25   record;
26
27   resourceR1 : aliased Resource(R1'access);
28
29   procedure Access_Resource(t : Integer) is
30   begin
31     ComputationWork(t);
32   end;
33
34   procedure Nested_Access(t1, t2, t3:Integer; R:access Resource) is
35   begin
36     ComputationWork (t1);
37     Lock(R.PC.all);
38     ComputationWork(t2);
39     Unlock(R.PC.all);
40     ComputationWork (t3);
41   end Nested_Resource;
42
43   -- ================== Task Definition ======================
44   -- Task acquiring no resource
45   task type No_Resource_Worker (
46     pri : Integer;
47     res : access Resource;
48     affinity : Integer;
```

```
49      d    : Integer)  is
50          pragma Priority(pri);
51    end No_Resource_Worker;
52
53    task body No_Resource_Worker is
54    begin
55      STPO.setaffinity(STPO.self, affinity); -- affinity
56      delay until Start_Time;
57      ComputationWork(d);
58      stop;
59    end No_Resource_Worker;
60
61    -- Task acquiring single resource
62    task type Single_Resource_Worker (
63      pri : Integer;
64      res : access Resource;
65      affinity : Integer;
66      d1, d2, d3: Integer)  is
67          pragma Priority(pri);
68    end Single_Resource_Worker;
69
70    task body Single_Resource_Worker is
71    begin
72      STPO.setaffinity(STPO.self, affinity); -- affinity
73      delay until Start_Time;
74      ComputationWork(d1);
75      Lock(res.PC.all);
76      Access_Resource(d2, res);
77      Unlock(res.PC.all);
78      ComputationWork(d3);
79      stop;
80    end Single_Resource_Worker;
81
82      -- Task acquiring nested resource
83    task type Nested_Resource_Worker (
84      pri : Integer;
85      res1, res2 : access Resource;
86        affinity : Integer;
87        d1, d2, d3, d4, d5: Integer)  is
88        pragma Priority(pri);
89    end Nested_Resource_Worker;
90
91    task body Nested_Resource_Worker is
92    begin
93      STPO.setaffinity(STPO.self, affinity); -- affinity
94      delay until Start_Time;
95      ComputationWork(d1);
96      Lock(res1.PC.all);
97      Nested_Access(d2, d3, d4, res2);
98      Unlock(res1.PC.all);
99      ComputationWork(d5);
100     stop;
101   end Nested_Resource_Worker;
102
```

```
103   --  ================== Driving Procedure ==================
104   procedure driver is
105
106     --  Prioriy, resource, partition,  affinity, delay 1 - 3
107     T1 : Single_Resource_Worker (1, resourceR1'access, 1, 2, 2, 3);
108
109   begin
110     null;
111   end driver;
112
113 begin
114     Start_Time := Clock + milliseconds(1000);  -- Set the start up
            time.
115     driver; -- Driver procedure to start the tasks
116 end Main;
```

Listing 5.2: Compiled Evaluation Main Program

Listing 5.2 demonstrates the actual evaluation main program for the protocol tests. Application programmers can define their application-defined protocols as shown by the example at line 20. If the evaluation task acquires only one shared resource, the template single resource worker will be instantiated. However, if multiple shared resources are required at run time, the nested resource worker template will be used. It will call the nested access procedure at line 34 first. Within this procedure, another simulated resource attempt will be made at line 37. This is to simulate the calling sequence of the nested resource requirements.

QueueLock package The System.Multiprocessors.QueueLock package (often renamed as SQL in the following codes) was specifically created for the evaluation work and for common support for all protocols. It was introduced as an interface and tools package between the evaluation code and the low level programming language for channeling low level functionalities to the application codes. The Ada compiler GCC 4.6.1 used for the evaluation is strict about the test program using internal GNAT functions. Some crucial functions required by the implementation of the resource sharing algorithms at the application level are not available. This package was therefore created as an aggregate of functionalities including system locks, scheduling functions, task operations and type casts etc. These procedures were fully wrapped and tested before being used by the evaluation code. The full implementation is listed in Appendix B.

Time Stamping and Status Message The following evaluation code often prints the current time point associated with some brief messages explaining the current state of the executing task. This is an effective evaluation method to identify the current state of the calling task. The brief message will not only verify that if the current calling task was executed as expected, but also reveals the situation of all other tasks. For example, a low priority task will not print anything if it has been preempted by a high priority task. The time stamping will check the order of the events ensuring the protocol and framework were all making expected decisions in a correct order.

5.2 Multiprocessor Stack Resource Policy - MSRP

Tasks in MSRP are blocked at the release stage. Once released, the execution will not be blocked due to unavailable resource. Protocols like MSRP use spin based locking. This imposes a challenge to the framework because it involves early interaction and has to dynamically adjust the system ceilings to release the blocked tasks.

As shown by Table 5.1, the Multiprocessor Stack Resource Policy requires partitioned scheduling and FIFO spinning to unavailable global resource. In the Ada context, the preemption level of the task is directly related to the Ada task priority. This can be set at the task creation using an Ada pragma.

The partitioning of the task is achieved using the Ada 2012 CPU aspect. Since we are using Ada 2005, we have to call the set task affinity function explicitly. MSRP is a ceiling protocol, so when a task is inside a protocol object, the task runs at the priority of the protected object ceilings. Ada supports preemptive priority based scheduling and no preemption will occur once a task is inside a protected object and another task wants to use the same protected object. Hence, the current system ceiling is implicitly the ceiling of the locked protected object with the highest ceiling value. The scheduling scenario, depicted by Figure 2.2, assigns 5 tasks on two processors. The resource allocation and preemption level assignments are given in Table 5.2:

With the shared resource specified [1], the evaluation test was conducted to verify

[1] T1 is released first followed by T2 and T4

Task	Affinity	Requiring Resource	Preempt Level	Release Order
T1	P1	nil	3	1
T3	P1	R1 (G1)	2	2
T2	P1	R1 (G1)	1	3
T4	P2	G1	1	2
T5	P2	nil	2	3

Table 5.2: Resource Table of MSRP

the behaviour of the MSRP protocol in the framework. If the scheduling outcome including the intermediate states and use of the fundamental primitives matches the expected results, the spin lock based protocols like MSRP can be supported by the framework.

The implementation of the MSRP can be found at Listing 5.3:

```
1  -- with necessary packages
2
3  package MSRP is
4
5     package SQL renames System.Multiprocessors.QueueLock;
6     type Global_MSRP(Num_Of_Tasks:Natural; V:Locking_Visibility; ID:
          Integer; Ceil:Integer) is new Protected_Controlled with private;
7
8     overriding procedure Initialize (C : in out Global_MSRP);
9     overriding procedure Finalize (C : in out Global_MSRP);
10    overriding procedure Lock(C : in out Global_MSRP);
11    overriding procedure Unlock(C : in out Global_MSRP);
12
13    power_on : Time;
14
15 private
16
17    type Global_MSRP(Num_Of_Tasks:Natural; V:Locking_Visibility; ID:
          Integer; Ceil:Integer) is new Protected_Controlled  with
18    record
19      Global_Lock : aliased SQL.Spin_Lock;
20    end record;
21
22 end MSRP;
```

Listing 5.3: MSRP Package Specification

A shared resource under MSRP is a new protected controlled type acceptable by the framework. The implementation overrides four methods of the framework to specify the behaviours of tasks in shared resources. The package specification defines types and records needed for the implementation.

```
1  package body MSRP is
2
3     package STPO renames System.Task_Primitives.Operations;
4
5     procedure stop is
6     begin
7        -- reduce system ceiling
8        Print_Time;
9        Put_Line(" => "&image(current_task)(1..2)&" stops execution");
10    end;
11
12    overriding procedure Initialize (C : in out Global_MSRP) is
13    begin
14      SQL.Initialize(C.Global_Lock);
15    end;
16
17    overriding procedure Finalize (C : in out Global_MSRP) is
18    begin
19      null;
20    end;
21
22    overriding procedure Lock(C : in out Global_MSRP) is
23      P : integer;
24    begin
25      if C.V then  -- Global Resource
26        Print_Time;
27        Put_Line(" => " &image(current_task)(1..2) &" attempts to lock
              resource G"&Integer'image(C.ID));
28        Set_Priority(C.Ceil + SQL.Get_PML);
29        SQL.SimpleSpinLock(C.Global_Lock);
30        Print_Time;
31        Put_Line(" => "&image(current_task)(1..2)&" Got the Global Lock
              and acquired G"&Integer'image(C.ID));
32      else  -- Local Resource
33        Print_Time;
34        Put_Line(" => " &image(current_task)(1..2) &" attempts to lock
              resource R"&Integer'image(C.ID));
35        Set_Priority(C.Ceil);
36        Print_Time;
37        Put_Line(" => "&image(current_task)(1..2)&" Got the Local Lock
              and acquired R"&Integer'image(C.ID));
38      end if;
39    end Lock;
40
41    overriding procedure Unlock(C : in out Global_MSRP) is
42      P : integer;
43    begin
44        if C.V then -- Global Resource
45          Print_Time;
46          Put_Line(" => "&image(current_task)(1..2)&" unlocked G"&
                Integer'image(C.ID));
47          SQL.SimpleSpinUnlock(C.Global_Lock);
48          Set_Priority(SQL.Get_PML);
49        else -- Local Resource
```

```
50            Print_Time;
51            Put_Line(" => "&image(current_task)(1..2)&" unlocked R"&
                  Integer'image(C.ID));
52            Set_Priority(SQL.Get_PML);
53            end if;
54    end Unlock;
55
56 end MSRP;
```

Listing 5.4: MSRP body

Listing 5.4 explains the implementation details of the MSRP protocol within the framework. The test program ensures the release follows the sequence given in Figure 2.2. The protocol starts when a task becomes runnable. Since MSRP is fully partitioned, the released task is firstly assigned to its allocated processor following the design of the evaluation task model. The preemption levels of the tasks are reflections of their priorities. The evaluation test program is set to use FIFO within priority scheduling. The tasks with low preemption level will not be released for execution by the scheduler because there is a higher priority task executing in the system. This feature facilitates the assertions of the MSRP algorithm so that only the tasks with higher preemption level than the system ceiling will be able to start execution. During its execution cycle, the released tasks may access the global/local shared resource through the lock and unlock procedures. If the resource is global, its priority is immediately raised to the corresponding highest and becomes non-preemptive at line 28. It appends itself to the global FIFO queue waiting for the precedence tasks to finish. When the resource is granted, the calling task will be returned from the SimpleSpinLock function. It will finish the lock procedure and continues its execution in the critical section. This implementation relies on the pthread spin lock functions to provide the FIFO queuing facilities [2] If the resource is local, starting from line 32, the calling task will raise its priority and return. There is no lock requests necessarily required by the local resources because the protocol itself guarantees that the released task will not be blocking once released. In the unlock procedure, the calling task will delete itself from the global FIFO, if the resource is global, and yield the resource to the next waiting

[2]The implementation of the pthread spin lock may vary in the distribution versions of Linux. The version used by the evaluation tests is using the kernel 3.6.11x86_64 #1 SMP PREEMPT. The implementation utilizes a 32bits integer, depending on the implementation, as the FIFO queue for the spin locks. It always append the new tasks to the higher bits and releases the tasks at the lower bits.

task.

As shown by Table 5.2, there are two resources shared between 5 tasks in the scenario. R1 is the local resource shared between T2 and T3 on processor 1. G1 is a global resource required by both T3 and T4. Since MSRP supports nested resources, T3 have the nested resource requests of R1 and G1. In order to verify the effect of the algorithm in various scenarios, the tasks are set to be released at different times. Table 5.2 demonstrates the relative release sequence of the tasks. T3 is the first task to be released on processor 1 in the system. T2 and T4 were then released followed by T1 and T5. This release sequence will examine the global blocking mechanism of and other different important properties of MSRP. The output of the evaluation driving program is as follows:

```
1  [ 1.00031] => T3 starts execution
2  [ 1.01070] => T3 attempts to lock resource R1
3  [ 1.01073] => T3 Got the Local Lock and acquired R1
4  T3 => Got R1
5  [ 1.02083] => T4 starts execution
6  [ 1.03083] => T3 attempts to lock resource G1
7  [ 1.03085] => T3 Got the Global Lock and acquired G1
8  T3 => Got G1
9  [ 1.04132] => T4 attempts to lock resource G1
10 [ 1.05044] => T3 unlocked G1
11 [ 1.05046] => T4 Got the Global Lock and acquired G1
12 T4 => Got G1
13 [ 1.05048] => T1 starts execution
14 [ 1.06056] => T1 stops execution
15 [ 1.06065] => T4 unlocked G1
16 [ 1.06069] => T5 starts execution
17 T5 => Executing!
18 [ 1.08009] => T5 stops execution
19 [ 1.08012] => T3 unlocked R1
20 [ 1.08014] => T3 stops execution
21 [ 1.08016] => T2 starts execution
22 [ 1.08017] => T2 attempts to lock resource R1
23 [ 1.08019] => T2 Got the Local Lock and acquired R1
24 T2 => Got R1
25 [ 1.09031] => T2 unlocked R1
26 [ 1.09033] => T4 stops execution
27 [ 1.10029] => T2 stops execution
```

Listing 5.5: MSRP Evaluation Results

The evaluation result shown by Listing 5.5 ordered the outputs by their stamped times. These time stamps can be mapped against the time points in the original scenario which is fully demonstrated by the following table:

According to Table 5.3, all scheduling outcomes took place as expected by the

Evaluation Stamped Time	Original Scenario Time	Event
1.00031	t0	T3 starts
1.01070	t1	T3 locks R1
1.02083	t2	T4 starts
1.03083	t3	T3 locks G1
1.04132	t4	T4 attempts G1
1.05046	t5	T4 locks G1
1.06065	t6	T4 unlocks G1
1.08012	t7	T3 unlocks R1
1.08017	t8	T2 locks R1
1.09031	t9	T2 unlocks R1
1.10029	t10	T2 stops

Table 5.3: Mapping time stamps to Figure 2.2

original scenarios. The evaluation output is analyzed further as follows:

Tasks Execution Order In MSRP, tasks are released for execution in the order of their preemption levels. A high preemption level task once released will not be blocked until it has finished its execution or its preemption level is not the highest. Our test result in Listing 5.5 demonstrates this effect by showing, for example, T3 has been released and executes all the way through till it has released R1 and terminates execution at time 1.08014. The effectiveness of the preemption level is also demonstrated by the preemption of T4. At time 1.06065, T4 unlocked G1 and returns its preemption level to 1. T5 with a higher preemption level immediately starts execution at time 1.06069. T4 only get a chance to continue its work and terminates execution at time 1.09033.

Global FIFO Blocking The MSRP protocol demands the global resources shared in FIFO spin queue. At time 1.03085, T3 has successfully obtained the global resource G1. On processor 2, T4 attempts to lock G1 at time 1.04132. It becomes blocked and spinning in a global FIFO queue until T3 releases G1 at time 1.05044. As the next waiting task in the global FIFO queue, T4 was granted the resource at time 1.05046.

Execution Tasks are never blocked This is an important property of MSRP because violation of this rule will lead to the failure of the protocol and deadlock of the underlying tasks. In our test scenario, all tasks are released and executed straight to the end. For example, T3, acquiring most of the shared resources, starts execution at time 1.00031 and finished at time 1.08014. Its execution are never blocked due to unavailable resources. Similarly, T2 released at time 1.08016 has successfully obtained R1 at time 1.08019 because its release has been postponed by the protocol. If it starts its execution before time 1.08012, it will be blocked immediately because T3 will be holding R1.

The evaluation output matches the expected scheduling results of the original MSRP protocol. The implementation of MSRP was carried out using purely the framework and Ada language facilities. It demonstrate the ability of the framework of incorporating spin lock based protocols, providing release state blocking, differentiating global and local resources, supporting non-preemptive spinning wait for shared resources and nested locking with the permission of the protocol.

5.3 Multiprocessor Priority Ceiling Protocol - MPCP

MPCP is a typical resource sharing algorithm used for applications with suspension based locks. It is based on partitioned scheduling where tasks are statically assigned to processors before execution. The suspension lock is heavily used because the critical sections of these applications are often long and requires lengthy computations. The scheduling and resource sharing overheads are often small to the total execution time of the long critical section.

Resources are labeled as either global or local. Global resources are shared within a global ceiling priority range which is absolutely higher than any local priorities. Local resources are shared at the ceiling of local priorities with the original ceiling priority protocol.

```
1 package MPCP is
2
3   type Global_MPCP(SynchIndex : Integer; Ceil:Integer; ID:Integer; V:
        Visibility) is new Protected_Controlled with private;
```

```
4
5    overriding procedure Initialize (C : in out Global_MPCP);
6    overriding procedure Finalize (C : in out Global_MPCP);
7    overriding procedure Lock(C : in out Global_MPCP);
8    overriding procedure Unlock(C : in out Global_MPCP);
9
10   -- Internal procedures were omitted
11 private
12
13  type Global_MPCP(SynchIndex : Integer; Ceil:Integer; ID:Integer; V:
        Visibility) is new Protected_Controlled  with
14   record
15      Llock   : aliased SQL.Suspension_Lock;
16      DCeil   : Integer;
17      T       : Task_ID;
18   end record;
19
20   power_on : Time;
21
22 end MPCP;
```

Listing 5.6: MPCP Package Specification

As shown by Listing 5.6, the Global_MPCP type is a new protected_controlled type extending the framework with four new overriding methods. It accepts the parameters including the index of the synchronization processor, the ceiling of the resource, ID of the resource and the visibility indicating the resource type. Inside the resource type record, there is the actual suspension lock and the dynamic ceiling of the local resource. The lock and unlock methods were overridden to implement the behaviour of MPCP. When the corresponding protected object is called, the runtime system dynamically detects the associated resource sharing algorithm. If it is MPCP, the overridden initialize, finalize, lock and unlock methods are called instead of the original ones.

```
1 package body MPCP is
2
3    package STPO renames System.Task_Primitives.Operations;
4
5    overriding procedure Initialize (C : in out Global_MPCP) is
6    begin
7       SQL.SimpleSuspensionInit(C.Llock);
8       power_on := Clock;
9    end;
10
11   overriding procedure Finalize (C : in out Global_MPCP) is
12   begin
13      null;
14   end;
15
```

```
16    overriding procedure Lock(C : in out Global_MPCP) is
17      p : Integer;
18    begin
19      if C.V then -- global resource
20        Print_Time;
21        Put_Line(" => "&image(current_task)(1..2)&" attempts to lock
              resource G"&Integer'image(C.ID));
22        Print_Time;
23        Put_Line(" => "&image(current_task)(1..2)&" migrates to PG");
24        STPO.setaffinity(STPO.self, C.SynchIndex);
25        SQL.SimpleSuspensionLock(C.Llock);
26        Set_Priority(SQL.Get_PML+C.Ceil);
27      else -- local resource
28        -- implementation of Original Priority Ceiling Protocol
29    end Lock;
30
31    overriding procedure Unlock(C : in out Global_MPCP) is
32      p : Integer;
33    begin
34        C.T := Null_Task_Id;
35        p := SQL.GetAffinity;
36        SQL.SimpleSuspensionUnlock(C.Llock);
37        if C.V then -- global resource
38          Print_Time;
39          Put_Line(" => "&image(current_task)(1..2)&" unlocked G" &
                Integer'image(C.ID));
40          Print_Time;
41          Put_Line(" => " &image(current_task)(1..2)&" migrates back to
                processor"&Integer'Image(affinityset(Integer'value(image(
                current_task)(2..2)))));
42          STPO.setaffinity(STPO.Self, affinityset(Integer'value(image(
                current_task)(2..2))));
43          Set_Priority(SQL.Get_PML); -- restore the original priority
44        else -- local resource
45          -- implementation of Original Priority Ceiling Protocol
46        end if;
47    end Unlock;
48
49  end MPCP;
```

Listing 5.7: MPCP Package Body

Listing 5.7 demonstrates the details of MPCP in the framework. The lock procedure is called when an evaluation task attempts to lock a shared resource. If the resource being acquired is global, the current calling task will be migrated to the synchronization processor as shown by line 24. Its priority will be immediately raised to the corresponding global ceiling at line 26 after obtaining the resource. As depicted by [48], the global critical section is always executed at the global ceiling priority which is higher than the assigned priority of the highest priority task in the system. If the resource is local, the task will obtain the resource according

to the original priority ceiling protocol. Its priority will only be raised if actual blocking takes place.

When the unlock is called, the underlying suspension lock is firstly released. The tasks on synchronization processor have to migrate back to their original processors when their global resource execution is finished. This is accomplished at line 42 and reset the tasks to its original priority at line 43. If the resource is local, the task will restore its previous priority and yield the resource to any waiting tasks. The implementation of unlock local resources were following the semantics of original priority ceiling protocol.

The evaluation program declares two local resources (R1, R2) and two global resources (G1, G2). The resource requirements are depicted by the following table:

Task	**Affinity**	**Requiring Resource**	**Base Priority**	**Release Sequence**
T1	P1	R1	1	3
T2	P2	G2	6	3
T3	P1	G1	5	1
T4	P2	R2	2	2
T5	P1	G1	3	5
T6	P2	G2	4	4

Table 5.4: MPCP Resource Table

Since the global ceiling must be absolutely higher than all local ceilings, G1 and G2 are assigned ceilings 12 (7+5) and 13 (7+6) respectively. After being declared at line 8, the resources are associated with Ada protected objects within our evaluation task model. At this point, the framework was configured with application defined resource sharing protocols so that when the compiled protected objects are called the application lock and unlock procedures are dispatched instead. As shown by Table 5.4, there are 6 tasks running in the system. The affinity of the tasks are statically determined at compile time. Following our evaluation task model, all tasks are released to the system at the same time (1000 milliseconds after the system startup). After a variable initialization delay, all tasks are resumed from startup delay and attempt to run. During their executions, the tasks can access the shared resources through lock/unlock procedures. Once started, the tasks executions are independent to each other and delays between states are effective in creating requests with different timings. The same test scenario with

different delays may result in distinguishing protocol behaviour.

In order to verify the correctness of the implementation, tasks are set with various release times. T3 is firstly released on P1. T4 is then released for acquiring R2 on P2. T2 and T1 are released at the same time after T4 has obtained the lock. Before T3 releases G1 on the synchronization processor, T6 and T5 are released in sequence. The test program ensures such release follows the sequence given in Figure 2.1. The output of the evaluation is collected and analyzed by Listing 5.8:

```
1  [ 1.00027] => T4 attempts to lock resource R 2
2  T4 => Got R2
3  [ 1.00032] => T3 attempts to lock resource G 1
4  [ 1.00035] => T3 migrates to PG
5  T3 => Got G1
6  [ 1.00052] => T1 attempts to lock resource R 1
7  T1 => Got R1
8  [ 1.00072] => T2 attempts to lock resource G 2
9  [ 1.00074] => T2 migrates to PG
10 T2 => Got G2
11 [ 1.00112] => T6 attempts to lock resource G 2
12 [ 1.00115] => T6 migrates to PG
13 [ 1.00122] => T2 unlocked G 2
14 [ 1.00123] => T2 migrates back to processor 2
15 [ 1.00138] => T5 attempts to lock resource G 1
16 [ 1.00140] => T5 migrates to PG
17 [ 1.00147] => T1 unlocked R 1
18 [ 1.00148] => T1 stops execution
19 [ 1.00148] => T2 stops execution
20 [ 1.00156] => T3 unlocked G 1
21 [ 1.00157] => T3 migrates back to processor 1
22 T6 => Got G2
23 [ 1.00178] => T4 unlocked R 2
24 [ 1.00179] => T4 stops execution
25 [ 1.00181] => T3 stops execution
26 [ 1.00181] => T6 unlocked G 2
27 [ 1.00183] => T6 migrates back to processor 2
28 T5 => Got G1
29 [ 1.00207] => T5 unlocked G 1
30 [ 1.00207] => T5 migrates back to processor 1
31 [ 1.00208] => T6 stops execution
32 [ 1.00233] => T5 stops execution
```

Listing 5.8: MPCP Evaluation Output

The number surrounded by a pair brackets indicates the spot time of the message printed. The following message explains the current state of the executing task. For example, at time 1.00027 second, T4 has become released and attempts to lock R2. It is then granted the access to R2 and printed the message "$T4 => Got\ R2$".

In order to verify the MPCP implementation, the time stamps of the evaluation output are mapped to the time stamps of Figure 2.1 as follows:

Evaluation Stamped Time	**Original Scenario Time**	**Event**
1.00027-1.00032	t1	T3 locks G1, T4 releases and locks R2
1.00052	t2	T1 releases and locks R1
1.00072	t3	T2 releases and locks G2
1.00072-1.00112	t4	T6 releases and attempts G2
1.00112	t5	T2 unlocks G2
1.00138	t6	T5 releases and attempts G1
1.00178-1.00183	t7	T4 unlocks R2, T6 unlocks G2
1.00207	t8	T5 unlocks G1
1.00233	t9	T5 stops execution

Table 5.5: Mapping time stamps to Figure 2.1

According to Table 5.5, all scheduling outcomes were take place as expected by the original scenarios. The evaluation output is analyzed further as follows:

Minimizing Remote Blocking The MPCP scales up the global resource ceiling to reduce the remote blocking phenomenon on accessing global resources. At time 1.00032, T3 attempts to lock G1. It migrates to the synchronization processor and continues execution. However, at time 1.00072, T2 becomes released and attempts to lock G2. It therefore migrates to the synchronization processor and immediately preempts T3 because T3 has lower priority. T3 will only be resumed if T2 has finished its execution and migrated back to P2. The events at time 1.00156 to 1.00157 testifies that T3 was resumed as expected. The preemption of T3 justifies the fact that T2 was actually set running at higher priority. The remote blocking is minimized in this case because, if T2 was set running at a lower priority, it could suffer blocking from other high priority tasks which may or may not require G2.

Local Ceilings R2 is used by T4 and shared between some other tasks with a lower priority and not relevant to this test. T4 was the second earliest task released to the system and acquired R2 at time 1.00027. Although running

at the local ceiling, T4 was preempted by higher priority task T2 and T6. Its execution is finally finished at time 1.00179 before T6. This matches the theorem scheduling outcome of processor 2 in the literature review.

Effective of Priority Scheduling and Task Allocation There is at most one task execution at any processor at any time. The highest priority task is always scheduled for execution.

5.4 Flexible Multiprocessor Locking Protocol - FMLP

The FMLP is a protocol with the concept of group locking. The previously mentioned protocols are all treating the individual tasks as the smallest scheduling entity to share the resources on a stand alone basis. The group locks of FMLP differentiate the shared resources in terms of the length of their critical sections. Resources with similar critical section length are grouped together so that nested resource requests are allowed by holding a group lock.

The evaluation test follows the original FMLP proposal and introduces 4 evaluations tasks running on 2 processors. The test was designed to evaluate the ability of the framework in supporting resource division, deploying multiple locking primitives as well as resource grouping. It is expected that the evaluation outcome will match the theorem scheduling results of the original proposal. Since both global and partitioned scheduling are supported by FMLP, partitioned scheduling was chosen as the scheduling algorithm for the evaluation test. This is because partitioned FMLP has no task migration which is more effective for demonstration purposes.

The resource requests of the tasks are depicted by the following table:

S1 and S2 are two short resources grouped in one short resource group with group lock G1. T2 requires nested access to S1 and S2 by holding G1. L1 is a long resource and individually grouped to G2. T1 is firstly released on P1 followed by T2 on P2. T3 is then released on P1 before T1 releases S1. T4 is released before T2 releases L1 on P2. The implementation of FMLP is explained by Listing 5.9

Task	Affinity	Requiring Resource	Priority	Release Sequence
T1	P1	S1	3	1
T2	P1	S1 (S2), L1	2	2
T3	P2	S2	1	3
T4	P2	L1	4	4

Table 5.6: FMLP Resource Table

```
1  package FMLP is
2
3      package SQL renames System.Multiprocessors.QueueLock;
4
5      type Resource_Type is (short, long);
6      type Group_Lock is private;
7      type FMLP (r:Resource_Type; GL:Group_Lock; index:Integer; id:
           Integer) is new Protected_Controlled with private;
8
9
10   overriding procedure initialize (C : in out FMLP);
11   overriding procedure Finalize (C : in out FMLP);
12   overriding procedure Lock(C : in out FMLP);
13   overriding procedure Unlock(C : in out FMLP);
14
15   power_on : Time;
16
17 private
18
19   type FMLP  (r:Resource_Type; GL:Group_Lock; index:Integer; id:
       Integer)  is new Protected_Controlled with
20     record
21         slock : SQL.Spin_Lock;
22         dlock : SQL.Spin_Lock;
23         llock : SQL.Suspension_Lock;
24     end record;
25
26     type long_record is record
27       lock : SQL.S_Object;
28       pri : Integer;
29     end record;
30
31   type short_arr_t is array (0..SQL.Num_Of_Task) of SQL.Spin_Lock;
32     type long_arr_t is array (0..SQL.Num_Of_Task, 0..SQL.Num_Of_Task)
           of long_record;
33     type owner_arr_t is array (0..SQL.Num_Of_Task) of Task_Id;
34     type ceil_arr_t is array (0..SQL.Num_Of_Task) of integer;
35
36     type Group_Lock is
37     record
38         SQ : short_arr_t;
39         LQ : long_arr_t;
```

```
40          LQN: ceil_arr_t;
41          data_lock : SQL.Spin_Lock;
42          total : integer := 0;
43          owner : owner_arr_t;
44          ptr, DPrio : ceil_arr_t;
45      end record;
46
47  end FMLP;
```

Listing 5.9: FMLP Package Specification

A resource in FMLP can be either short or long. The type of resource is defined at line 5. An FMLP resource is a new protected controlled type acceptable by the framework. It accepts three parameters. The Resource_Type indicates whether this is a short or a long resource. The Group_Lock is instantiated and passed in by the application developers. The application developers may group multiple FMLP resources into the same group by passing the same group lock to the instances of these resources. The group lock type must be consistent with its outer most enclosing resource. If the enclosed resource is short, the group lock must be short typed. Otherwise, the group lock is long. The id assigns an unique ID to each resource. This is especially useful of understanding the internal states of the resource and debugging.

As shown by line 21-23 Listing 5.9, FMLP resource is implemented with a spin lock and a suspension lock. The group lock record is visible to all tasks in the system containing both spin and suspension locks. The application developers can instantiate a group lock and associate it with an FMLP object following the type definition at line 7. The implementation will differentiate the type of the FMLP protocol by applying corresponding actions. If the resource is short, the spin lock at line 21 will be used. If the resource is long, the suspension lock at line 23 will be deployed instead.

The implementation of FMLP protocol is depicted by Listing 5.10

```
1  package body FMLP is
2
3    package STPO renames System.Task_Primitives.Operations;
4
5    overriding procedure initialize (C : in out FMLP) is
6    begin
7      SQL.Initialize(C.dlock);
8      SQL.Initialize(C.slock);
9      SQL.SimpleSuspensionInit(C.llock);
10     -- other initialize routines including assign UID to C.index etc.
11   end;
```

```
12
13    overriding procedure Finalize (C : in out FMLP) is
14    begin
15      null;
16    end;
17    overriding procedure Lock(C : in out FMLP) is
18    begin
19      SQL.SimpleSpinLock(C.dlock); -- data lock
20      if C.r = short then
21        Print_Time;
22        Put_Line(" => "&image(current_task)(1..2) & " attempts to lock
              Short S"&Integer'image(C.id));
23
24        if C.GL.owner(C.index)/=current_task or C.GL.owner(C.index)=
              Null_Task_Id then
25          append(index, owner, C.GL.owner(C.index)); -- FIFO task
26          append(index, priority, Get_Priority(Current_Task)); -- FIFO
                priority
27          Set_Priority(System.Any_Priority'Last);
28          SQL.SimpleSpinUnlock(C.dlock); -- data lock
29          SQL.SimpleSpinLock(C.GL.SQ(C.index)); -- acquire Group Lock
30
31          SQL.SimpleSpinLock(C.dlock); -- data lock
32          Print_Time;
33          Put_Line(" => "&image(current_task)(1..2)&" obtained [Group
                Lock, Short] " &Integer'image(C.index));
34          C.GL.owner(C.index) := current_task;
35          C.GL.DPrio(C.index) := System.Any_Priority'Last;
36          SQL.SimpleSpinUnlock(C.dlock); -- data lock
37        else
38          Set_Priority(System.Any_Priority'Last);
39          SQL.SimpleSpinUnlock(C.dlock); -- data lock
40        end if;
41
42        SQL.SimpleSpinLock(C.slock); -- acquire resource lock
43        Print_Time;
44        Put_Line(" => "&image(current_task)(1..2)&" obtained Spin Lock
              for S"&Integer'image(C.id));
45        C.GL.ptr(C.index) := C.GL.ptr(C.index) + 1 ;
46      else
47        Print_Time;
48        Put_Line(" => "&image(current_task)(1..2)&" attempts to Lock L"
              &Integer'image(C.id));
49        if (C.GL.DPrio(C.index)<Get_Priority(Current_Task) and C.GL.
              owner(C.index)/=Null_Task_Id) then
50          append(index, priority, Get_Priority(C.GL.owner(C.index)));
                -- FIFO
51          C.GL.DPrio(C.index) := Get_Priority(Current_Task);
52          Set_Priority(Get_Priority(Current_Task), C.GL.owner(C.index))
                ;
53        end if;
54
55        if C.GL.owner(C.index) /= current_task and C.GL.owner(C.index)
              /=Null_Task_Id then
```

```
56          C.GL.LQN(C.index) := C.GL.LQN(C.index) + 1;
57          C.GL.LQ(C.index, C.GL.LQN(C.index)).pri := Get_Priority(
                Current_Task);
58          SQL.SimpleSpinUnlock(C.dlock); -- data lock
59          SQL.Suspend_Untill_True(C.GL.LQ(C.index, C.GL.LQN(C.index)).
                lock); -- Group Lock
60          SQL.SimpleSpinLock(C.dlock); -- data lock
61          Print_Time;
62          Put_Line(" => "&image(current_task)(1..2)&" obtained [Group
                Lock, Long] " &Integer'image(C.index));
63        else
64          C.GL.owner(C.index) := current_task;
65        end if;
66        SQL.SimpleSpinUnlock(C.dlock); -- data lock
67        SQL.SimpleSuspensionLock(C.llock);     -- Resource Lock
68        Print_Time;
69        Put_Line(" => "&image(current_task)(1..2)&" obtained Suspension
              Lock for L"&Integer'image(C.id));
70        C.GL.ptr(C.index) := C.GL.ptr(C.index) + 1;
71      end if;
72    end Lock;
73
74    overriding procedure Unlock(C : in out FMLP) is
75      temp : Integer;
76    begin
77      SQL.SimpleSpinLock(C.dlock);
78      if C.r = short then
79        C.GL.ptr(C.index) := C.GL.ptr(C.index) - 1 ;
80        Print_Time;
81        Put_Line(" => "&image(current_task)(1..2)&" releases Spin Lock
              for S"&Integer'image(C.id));
82        SQL.SimpleSpinUnlock(C.slock); -- release resource lock
83
84        if C.GL.ptr(C.index) = 0 then
85          C.GL.owner(C.index) := Null_Task_Id;
86          Print_Time;
87          Put_Line(" => "&image(current_task)(1..2)&" releases [Group
                Lock, Short] "&Integer'image(C.index));
88          C.GL.owner(C.index) := get(C.index, owner);
89          C.GL.DPrio(C.index) := get(C.index, priority);
90          temp := C.GL.DPrio(C.index);
91          SQL.SimpleSpinUnlock(C.dlock);  -- data lock
92          SQL.SimpleSpinUnlock(C.GL.SQ(C.index)); -- Release the group
                lock
93          Set_Priority(temp);
94        else
95          SQL.SimpleSpinUnlock(C.dlock);
96        end if;
97      else
98        C.GL.ptr(C.index) := C.GL.ptr(C.index) - 1 ;
99        Print_Time;
100       Put_Line(" => "&image(current_task)(1..2)&" releases Suspension
              Lock for L"&Integer'image(C.id));
101       SQL.SimpleSuspensionUnlock(C.llock);  -- Resource Lock
```

```
102         if C.GL.ptr(C.index) = 0 then
103           C.GL.owner(C.index) := NULL_Task_Id;
104           Print_Time;
105           Put_Line(" => "&image(current_task)(1..2)&" releases [Group
                  Lock, Long] "&Integer'image(C.index));
106           SQL.Set_True(C.GL.LQ(C.index, find_max(C.GL, C.index)).lock);
                  -- Group Lock
107           SQL.SimpleSpinUnlock(C.dlock); -- data lock
108           Set_Priority(get(C.index, priority));
109         else
110           SQL.SimpleSpinUnlock(C.dlock); -- data lock
111         end if;
112       end if;
113     end Unlock;
114
115 end FMLP;
```

Listing 5.10: FMLP Body

The implementation extends the four methods of the framework to implement the FMLP protocol. The lock and unlock procedures are overriding. When a short FMLP resource is called, it firstly check whether the calling task is holding the group lock at line 24. If not, it attempts to save its current priority and obtain the group lock. According to protocol rules, tasks waiting or using the short resources should be executed non-preemptively until the lock has been relinquished. Therefore the priority of the calling task is raised to the highest at line 27 and reset accordingly at line 93. By setting the priority to the highest and using FIFO within priority scheduling, no other tasks will be able to preempt the spinning task until it voluntarily reduces its priority to a lower value. The current calling task is then safe to acquire the actual lock of the resource at line 42. It is worthwhile mentioning here that only short resources are allowed to be nested. Since the priority has been raised to the highest and the group lock is obtained, any nested short resource requests will be satisfied immediately. Tasks acquiring inner resource will jump through the ownership check and proceed to the resource lock directly. The priority of the calling task will only be reset back to its original value once all locked resources have been released. This is confirmed by checking the nesting levels of the call. If the nesting level is equal to zero, the calling task must have released all resources it has previously locked. Therefore, it is safe to restore its priority to its previous value. The previous priority of the task is saved to a FIFO queue at line 25-26. This is happening before the priority of the calling task has been set to non-preemptively high. If the calling task has locked some other

group lock before acquiring this short group lock, its priority must be decreased gradually to the previous value. It is highly possible that the priority of the calling task was inherited from a task which was blocked on another long resource group lock.

All resource requests for the long resources must obtain the group lock as well. Any high priority task blocked on the group lock should have its priority inherited by the low priority owner task at line 52. Similar to the short resources, the priority of the long resource group owner task must be saved to a FIFO queue at line 50 for priority resetting. After registering its priority, the long resource acquiring task is then checked if it has the ownership of the group lock at line 55. If it is not the owner, it will suspend itself at the group lock at line 59. After obtaining the group lock, it will attempt to acquire the actual resource lock at line 67. Once obtained the resource lock, it will increase the nesting level at line 70. On the unlocking phase, the current calling task will reduce its nesting level followed by releasing the shared resource. If it has released all its locked resources, the ownership is reset and the highest priority task waiting on the long group lock will be released. After releasing the data lock, its priority is reduced to its previous value which was saved at the FIFO queue at line 50.

The test program is configured to release the tasks following the sequence given in Figure 2.3. The evaluation output of the above implementation is shown as follows:

```
1  [ 1.00020] => T1 attempts to lock Short S 1
2  [ 1.00022] => T1 obtained [Group Lock, Short]  1
3  [ 1.00023] => T1 obtained Spin Lock for S 1
4  T1 => Got S1
5  [ 1.00090] => T2 attempts to lock Short S 1
6  [ 1.00093] => T1 releases Spin Lock for S 1
7  [ 1.00095] => T1 releases [Group Lock, Short]  1
8  [ 1.00096] => T1 stops execution
9  [ 1.00096] => T2 obtained [Group Lock, Short]  1
10 [ 1.00096] => T2 obtained Spin Lock for S 1
11 T2 => Got S1
12 [ 1.00107] => T3 attempts to lock Short S 2
13 [ 1.00139] => T2 attempts to lock Short S 2
14 [ 1.00139] => T2 obtained Spin Lock for S 2
15 T2 => Got S2
16 [ 1.00177] => T2 releases Spin Lock for S 2
17 [ 1.00178] => T2 releases Spin Lock for S 1
18 [ 1.00178] => T2 releases [Group Lock, Short]  1
19 [ 1.00179] => T3 obtained [Group Lock, Short]  1
20 T3 => Got S2
21 [ 1.00179] => T2 attempts to Lock L 1
```

```
[ 1.00181] => T2 obtained [Group Lock, Long]  2
[ 1.00182] => T3 obtained Spin Lock for S 2
[ 1.00184] => T2 obtained Suspension Lock for L 1
T2 => Got L1
[ 1.00252] => T4 attempts to Lock L 1
[ 1.00253] => T3 releases Spin Lock for S 2
[ 1.00254] => T3 releases [Group Lock, Short]  1
[ 1.00255] => T3 stops execution
[ 1.00299] => T2 releases Suspension Lock for L 1
[ 1.00300] => T2 releases [Group Lock, Long]  2
[ 1.00301] => T2 stops execution
[ 1.00301] => T4 obtained [Group Lock, Long]  2
[ 1.00336] => T4 obtained Suspension Lock for L 1
T4 => Got L1
[ 1.00337] => T4 releases Suspension Lock for L 1
[ 1.00337] => T4 releases [Group Lock, Long]  2
[ 1.00342] => T4 stops execution
```

Listing 5.11: FMLP Evaluation Results

The full effect of FMLP is illustrated and evaluated through comparing the evaluation results to the original scenario. In order to do so, the evaluation time stamps are mapped to the time stamps of the original scenario in Table 5.8.

Evaluation Stamped Time	**Original Scenario Time**	**Event**
1.00020-1.00023	t0	T1 releases and locks S1
1.00090-1.00096	t1	T1 releases S1 and the group lock, T2 resumes and locks S1
1.00177-1.100179	t2	T2 releases S2 and S1 with their group locks
1.00252	t3	T4 attempts L1 but blocked
1.00299-1.00300	t4	T2 releases L1 and its group lock
1.00301	t5	T2 stops
1.00342	t6	T4 releases L1 and stops

Table 5.7: Mapping time stamps to Figure 2.3

According to Table 5.8 and Figure 2.3, the scheduling events were take place as expected. The evaluation output is analyzed further against the original scenario of FMLP:

Group Locking Effect This evaluation demonstrates various blocking phenomenon as the result of group locking. At time 1.00090, T2 attempts to acquire resource S1 and gets blocked due to the group lock of S1 is being held by T1.

T2 only gets the group lock after it has been released at time 1.00096. T2 then continues its execution and subsequently locks S2. When T3 becomes released and attempts to lock S2, it is blocked at time 1.00107 due to the unavailable group lock G1 even though the S2 is free at that time. T3 only gets the resource after G1 has been released by T2 at time 1.00178.

Nested Resource Requests In FMLP, a task may issue nested resource request if it is holding the corresponding group lock. T2, in this case, acquires S1 and S2 in a nested way after obtaining the group lock at time 1.00096. By holding the group lock, it prevents other tasks from entering the resource group and suffers minimum blocking from accessing nested resource. This is evidenced by blocking T3 from acquiring S2.

Incorporating different locking primitives FMLP requires both spin and suspension lock to operate the execution of the whole protocol. At time 1.00181, T2 obtained the long group lock for L1. T4 was therefore suspension blocked by the group lock at time 1.00252. At this moment, T2 inherits T4's priority and continue execution. T4 was resumed back and grants the resource at time 1.00301.

Customize Resource Configuration FMLP requires shared resources to be associated with certain group locks. The framework should therefore take extra input parameters passed in at run time and be adaptive. The evaluation shows that the framework is flexible and sufficient to take such customized resources. The implementation verifies the ability of the framework to take online configuration with the flexibility to manage different types of shared resources.

5.5 O(m) Locking Protocol - OMLP

The previously examined protocols have an implicit assumption that each shared resource is guarded by a particular queuing policy. This constraint is efficient if the application developers are interested in using one queuing policy for a single resource. However, when the application requires multiple different queuing policies, this constraint must be lifted. The OMLP protocol is a typical complex

protocol which uses multiple queuing policies on one single resource. It supports both global scheduling and partitioned scheduling. In global OMLP, as shown by this evaluation test, each resource is associated with an M long FIFO queue and a priority queue. The head of the FIFO queue gets the shared resource. Tasks join the FIFO queue waiting for the resource. Since the FIFO queue is limited to M length, the other tasks join the priority queue. Only the highest priority task is released to the FIFO queue when there is an available position.

The definition of the types and procedures of the implementation of Global OMLP in the framework is depicted by Listing 5.12

```
1  package OMLP is
2
3    package SQL renames System.Multiprocessors.QueueLock;
4
5    type Global_OMLP(FQMax : integer) is new Protected_Controlled with
         private;
6
7    overriding procedure initialize (C : in out Global_OMLP);
8    overriding procedure Finalize (C : in out Global_OMLP);
9    overriding procedure Lock(C : in out Global_OMLP; L:Lock_Type; V:
         Lock_Visibility; Ceiling:Priority; Tid:Task_Id );
10   overriding procedure Unlock(C : in out Global_OMLP; Tid:Task_Id);
11   power_on : Time;
12
13 private
14     type Q_record is
15         record
16             id : Task_Id;
17             pri: Integer;
18             lock : SQL.S_Object;
19         end record;
20
21     type Q_Arr_t is array (1..SQL.Num_of_Task) of Q_record;
22
23     type Global_OMLP(FQMax : integer) is new Protected_Controlled
           with
24         record
25             FQ, PQ : Q_Arr_t;
26             FQH, FQL, PQH, PQL: integer :=1;
27             data_lock : aliased SQL.Spin_Lock;
28         end record;
29
30 end OMLP;
```

Listing 5.12: OMLP Header

An OMLP resource is defined as a new protected controlled type with an extra customized configuration parameter specifying the length of the FIFO queue. The FQ and PQ are two arrays of Q_record. The S_Object is a specially implemented

conditional wait atomic synchronization object. The state of an S_Object can be changed via the access method. The benefit of having the S_Object in the implementation is to obtain the direct control of the tasks on suspension given in Appendix B. The task suspended in a false state S_Object will only wake up once the state has been turned to true by the other tasks. Having an array of S_Objects will provide us such direct control so that the application-defined protocol can resume the target task directly without affect the other tasks. Each OMLP resource contains one FIFO queue and one priority queue. The indexes of the FIFO queue and the priority queue are associated with the S_Objects. Tasks waiting in both queues are suspended in their allocated indexed S_Objects. The application developers then have the absolute control over the sequence of the tasks being released from these queues if they choose to have discretionary management over their queues instead of using the OS primitives. The data_lock is used as the mutual exclusion mechanism to protect the heavy queue operations.

The four virtual method of the framework were overridden by Listing 5.13 to implement the OMLP protocol.

```
1  package body OMLP is
2
3      package STPO renames System.Task_Primitives.Operations;
4      package ATI renames Ada.Task_Identification;
5      package ADP renames Ada.Dynamic_Priorities;
6
7    overriding procedure initialize (C : in out Global_OMLP) is
8    begin
9      -- initialization work
10   end;
11
12   overriding procedure Finalize (C : in out Global_OMLP) is
13   begin
14     null;
15   end;
16
17   overriding procedure Lock(C : in out Global_OMLP; L:Lock_Type; V:
         Lock_Visibility; Ceiling:Priority; Tid:Task_Id ) is
18     index : Integer;
19   begin
20     SQL.SimpleSpinLock(C.data_lock); -- data lock
21
22     if (C.FQ(C.FQL).id/=Null_Task_Id and C.FQ(C.FQL).pri<Get_Priority
           (current_task)) then
23       Set_Priority(Get_Priority(current_task), C.FQ(C.FQL).id);
24     end if;
25
26     if ((C.FQH-C.FQL) >= C.FQMax) then
27         index := append_PQ(C);
```

```
28            C.PQ(index).id := current_task;
29            C.PQ(index).pri := Get_Priority(current_task);
30            SQL.SimpleSpinUnlock(C.data_lock); -- data lock
31            SQL.Suspend_Until_True (C.PQ(index).lock); -- PQ lock
32            SQL.SimpleSpinLock(C.data_lock);  -- data lock
33            delete_PQ(C, index);
34            Print_Time;
35            Put_Line(image(current_task)(1..2)&" joins FQ from PQ");
36        end if;
37        if C.FQ(C.FQL).id/=Null_Task_Id then
38          if C.FQ(C.FQH).id/=current_task then
39             C.FQH := C.FQH + 1;
40          end if;
41          SQL.SimpleSpinUnlock(C.data_lock); -- data lock
42          SQL.Suspend_Until_True (C.FQ(C.FQH).lock); -- FQ Lock
43          SQL.SimpleSpinLock(C.data_lock); -- data lock
44        end if;
45        C.FQ(C.FQL).id := current_task;
46        if find_max(C)>Get_Priority(current_task) then
47          C.FQ(C.FQL).pri := find_max(C);
48          Set_Priority(C.FQ(C.FQL).pri);
49        else
50          C.FQ(C.FQL).pri := Get_Priority(current_task);
51        end if;
52        SQL.SimpleSpinUnlock(C.data_lock); -- data lock
53    end Lock;
54
55    overriding procedure Unlock(C : in out Global_OMLP; Tid:Task_Id) is
56    begin
57        SQL.SimpleSpinLock(C.data_lock); -- data lock
58        C.FQL := C.FQL + 1;
59        if ((C.FQH-C.FQL)<C.FQMax and find_max(C.PQ)/=-1) then
60          SQL.Set_True(C.PQ(find_max(C.PQ)).lock);
61          C.FQH := C.FQH + 1;
62          C.FQ(C.FQH).id := C.PQ(find_max(C.PQ)).id;
63        end if;
64        SQL.Set_True(C.FQ(C.FQL).lock);
65        SQL.SimpleSpinUnlock(C.data_lock); -- data lock
66        Set_Priority(SQL.Get_PML);
67    end Unlock;
68
69
70  end OMLP;
```

Listing 5.13: OMLP Body

When the protected object associated with OMLP protocol is called, the lock and unlock procedure in Listing 5.13 will be dispatched. A task will call the lock procedure whenever it wants to lock an OMLP resource. It will check if the resource is being held by some other tasks and donates its priority to the resource holding task when its priority is higher than the resource holding task. It will then examine the size of the FIFO queue. If there is more than M number of

task already waiting, it will append itself to the priority queue and suspend on the priority queue lock at line 30. Each task has its dedicated S_Object in both priority and FIFO queue. Whenever a position in the FIFO queue becomes available, the OMLP protocol will only wake up the highest priority waiting task in the queue. Other normal priority task will remain suspended. When a task is resumed by the protocol from the priority queue, it firstly delete its index from the priority queue at line 32 and insert itself to the FIFO queue at line 39. Once released from the priority queue, the calling task will check if the resource is being hold by any other tasks. If so, it will set itself suspended at the end of the FIFO queue. Once resumed back from suspension in FIFO queue, it will check its current priority is the highest in both queues. If not, it will inherit the priority from the highest priority task in both queues.

Since the OMLP protocol requires the number of CPU available to make provisions for the data structure of the FIFO queue, the customized configuration parameter, C.FQMax, is used to provide such information to the framework. The CPU count was passed into the framework to configure the FIFO queue length. R, the shared resource in the evaluation scenario, is declared as an OMLP resource with a FIFO queue of 2 tasks. In this evaluation test, 6 tasks are assigned to 2 processors with global scheduling. All our previous evaluations were using partitioned scheduling. This evaluation was specifically designed for global scheduling. It is worthwhile mentioning that the release procedure in Listing 5.13 is not setting the affinity of the calling tasks anymore. The test driving program will not set the affinity of the tasks either. In this case, the underlying system will dispatch the executing tasks globally across its available processors.

In order to check the queue operation of OMLP, all tasks are released before T1 has released the resource. T1 and T2 are firstly released by granting resource to T1. T2 therefore will be blocked due to unavailable resource. All tasks are attempting to lock resource immediately after release. The rest tasks, T3, T6, T5 and T4, are released in sequence before R has been unlocked by T1. The output of the evaluation program is listed as follows:

```
1 [ 1.00005] => T1 released at priority: 1
2 [ 1.00005] => T2 released at priority: 2
3 [ 1.00017] =appendFQ= : inserting T1
4 T1 => Got R
5 [ 1.00018] =appendFQ= : inserting T2
6 [ 1.00048] => T3 released at priority: 3
```

```
 7 [ 1.00070] => T6 released at priority: 4
 8 [ 1.00075] => T3 Lock: append_PQ : 1
 9 [ 1.00095] => T6 Lock: append_PQ : 2
10 [ 1.00116] => T5 released at priority: 5
11 [ 1.00138] => T4 released at priority: 6
12 [ 1.00140] => T5 Lock: append_PQ : 3
13 [ 1.00162] => T4 Lock: append_PQ : 4
14 [ 1.00224] =deleteFQ= : releasing T1
15 [ 1.00227] => T1 Unlock: release_HQ :T2
16 [ 1.00229] T2 joins FQ from PQ
17 T2 => Got R
18 [ 1.00229] => T1 stops execution
19 [ 1.00230] =appendFQ= : inserting T4
20 [ 1.00233] => T2 Unlock: delete_PQ : 2
21 T4 => Got R
22 [ 1.00275] =deleteFQ= : releasing T4
23 [ 1.00277] => T4 Unlock: release_HQ :T5
24 [ 1.00278] T5 joins FQ from PQ
25 [ 1.00280] => T5 Unlock: delete_PQ : 3
26 [ 1.00280] =appendFQ= : inserting T5
27 [ 1.00301] => T2 stops execution
28 [ 1.00301] =deleteFQ= : releasing T4
29 [ 1.00303] => T4 Unlock: release_HQ :T6
30 T5 => Got R
31 [ 1.00304] T6 joins FQ from PQ
32 [ 1.00307] => T6 Unlock: delete_PQ : 2
33 [ 1.00308] =appendFQ= : inserting T6
34 [ 1.00326] => T4 stops execution
35 [ 1.00327] =deleteFQ= : releasing T5
36 [ 1.00330] => T5 Unlock: release_HQ :T3
37 T6 => Got R
38 [ 1.00331] T3 joins FQ from PQ
39 [ 1.00335] => T3 Unlock: delete_PQ : 1
40 [ 1.00335] =appendFQ= : inserting T3
41 [ 1.00353] => T5 stops execution
42 [ 1.00356] =deleteFQ= : releasing T6
43 T3 => Got R
44 [ 1.00382] => T6 stops execution
45 [ 1.00383] =deleteFQ= : releasing T3
46 [ 1.00386] => T3 stops execution
```

Listing 5.14: OMLP Evaluation Results

In order to fully verify the correctness of the queue operation, the evaluation results must be linked back to the original scenario in Figure 2.8. The evaluation time stamps are mapped back to the time stamps of the original scenario in Table 5.8.

Following the time mappings, the whole output is analyzed against the original proposal as follows:

Time[1.00005] The first two released and resource acquiring tasks T1 and T2

Evaluation Stamped Time	Original Scenario Time	Event
1.00005	t0	T1 T2 released
1.00048	t1	T3 released
1.00075	t2	T3 attempts to lock R
1.00095	t3	T6 attempts to lock R
1.00116	t4	T5 released
1.00138	t5	T4 released
1.00224	t6	T1 unlocks R
1.00233	t7	T2 unlocks R
1.00277	t8	T4 unlocks R
1.00280	t9	T5 unlocks R
1.00307	t10	T6 unlocks R
1.00335	t11	T3 unlocks R

Table 5.8: Mapping time stamps to Figure 2.8

are appended to the FIFO queue successfully.

Time[1.00005-1.00138] T3 and T6 have also become released and immediately acquire resource R. Although having higher priority than T1 and T2, they are inserted at the priority queue. Similarly T5 and T4 are released and appended to the priority queue. It is expected that T4 will be the first task released from the priority queue as it has the highest priority 6.

Time[1.00227-1.00277] T1 has released resource R. T2, as the next waiting task in the FIFO queue, is granted the resource. This is evidenced by the print out of T2. At the same time, T4 was released from the priority queue to the FIFO queue at time 1.00277.

Time[1.00280-1.00386] T5, T6 and T3 are gradually released from the priority queue to the FIFO queue in order. This is because T5 has higher priority than T6 and T3. The release order was correct because T3 is the last finished task and it has the lowest priority 3.

The output of the evaluation test matches the original proposal. This reveals the ability of the framework in supporting multiple queuing policy at one shared resource.

5.6 Priority Donation - Clustered OMLP

The priority donation imposes extra challenges to the framework. It requires a closer interaction with the scheduler to control the priority donation tasks. Whenever a task displaces a resource sharing low priority task from executing on the processors, the priority donation mechanism will suspend the priority donor task and resumes the resource holding task for execution. The priority donor task will have to be automatically resumed when the priority donation is finished. All previous evaluated protocols are only dealing with the current executing task and no interactions with suspended tasks are required on behalf of the resource sharing protocols except those already handled by Ada. It also needs the support of cluster scheduling and the tracking of the C [3] highest executing tasks in the cluster.

Since priority donation is a very complex algorithm, its rules are recalled here [12]:

1. A task T_d becomes a priority donor to T_i during t_a (t_a : the time period between a task issuing its resource request and the resource being released by the task) if :
   (a) T_i was the C^{th} highest priority pending task prior to T_d's release
   (b) T_d has one of the C highest base priorities
   (c) T_i has issued a global request that is incomplete at period t_a
2. T_i inherits the priority of T_d during t_a.
3. If T_d is displaced from the set of the C highest priority tasks by the release of T_h , then T_h becomes T_i's priority donor and T_d ceases to be a priority donor.
4. If T_i is ready when T_d becomes T_i's priority donor, then T_d suspends immediately. T_i and T_d are never ready at the same time.
5. A priority donor may not issue resource requests. T_d suspends if it requires a resource while being a priority donor.

[3] C : the number of processors allocated to a cluster

6. T_d ceases to be a priority donor as soon as either :

   - T_i completes its request
   - T_i's base priority becomes one of the C highest
   - T_d is being relieved by a later released priority donor task.

Implementing the priority donation algorithm in Ada requires tasks to be stopped from running when they are released. Although Ada allows a task to be suspended (using the Ada.Synchronous_Task_Control package), these tasks have to be explicitly identified. With priority donation, the donating task can not be explicitly identified before it is released. Hence, with the current framework it is not possible to implement the priority donation algorithm. Burns and Wellings [14] have recognized that Ada provides a set of low-level mechanisms that needs to be combined to implement the real-time programming abstractions such as periodic, aperiodic and sporadic tasks. They present a group of these high-level abstractions including a release manager, which can handle the various release patterns along with deadline miss and cost overrun detection facilities. Task templates are then given that utilize the release managers so relieving the tasks of having to use the low-level mechanisms. The following (taken from [14]) is an example of these templates.

```
1  package Real_Time_Tasks is
2    task type Simple_Real_Time_Task(S : Any_Task_State; R :
         Any_Release_Mechanism; Init_Prio : Priority) is
3      pragma Priority(Init_Prio);
4    end Simple_Real_Time_Task;
5    ...
6  end Real_Time_Tasks;
7
8  with Ada.Task_identification; use Ada.Task_Identification;
9  package body Real_Time_Tasks is
10
11   task body Simple_Real_Time_Task is
12   begin
13     S.Initialize;
14
15     loop
16       R.Wait_For_Next_Release;
17       S.code;
18     end loop;
19   end Simple_Real_Time_Task;
20 end Real_Time_Tasks;
```

Listing 5.15: Release Manager Template

The real time tasks can be released by the passage of time or via certain events. The tasks, as depicted by Listing 5.15, are registering themselves to the release manager via calling Wait_For_Next_Release. The calling task will be waiting at the function until its release timer elapsed or certain events have occurred.

The main goal of the framework is that the application only indirectly interacts with the framework when it requests and releases a resource. This goal has to be relaxed in order to give the required flexibility. The overall approach taken in this section is that tasks, which wish to use the framework with resource allocation algorithm that are tightly coupled with the scheduler, must register themselves with the framework immediately once they are released. The proposed interaction between the main components of the priority donation protocol is summarized by Figure 5.2. The tasks executing in the systems need the support from both the release manager and the framework. At the release stage, the release manager needs to receive the timing event from the epoch timer in order to release the tasks at the correct time. The priority donation manger in the framework needs to authorize the release of the tasks that are not violating the priority donation rules. Similarly, at the lock, unlock and complete stages, the tasks needs to interact with the framework closely in order to proceed its execution.

In order to interact with these tasks, as shown by Listing 5.16, the release manager is defined as a protected type. It defines the semantics of the release action and how to execute the release at appropriate times. Following the definition at Figure 5.2, the release manager uses Ada timing events facility to release the pending tasks at predetermined times. The first time calling task therefore will have its release procedure registered with its release time by setting up a coordinated timing event. The associated event handler will be called automatically once the timing event has occurred, since the task should already be waiting on the Wait_For_Next_Release function, opening the barrier will set the task released immediately. The release procedure, defines how the release behaviours of the tasks, provides an interaction with the framework where application defined protocols may call back to influence the task releases at this level. Certain release events may cause a task to be released. The application defined protocols, like priority donation, may have specific controls on which tasks can be released. For example, a task may not be released if its priority is to be donated to some other task. In this case, the release manager checks with the framework if the

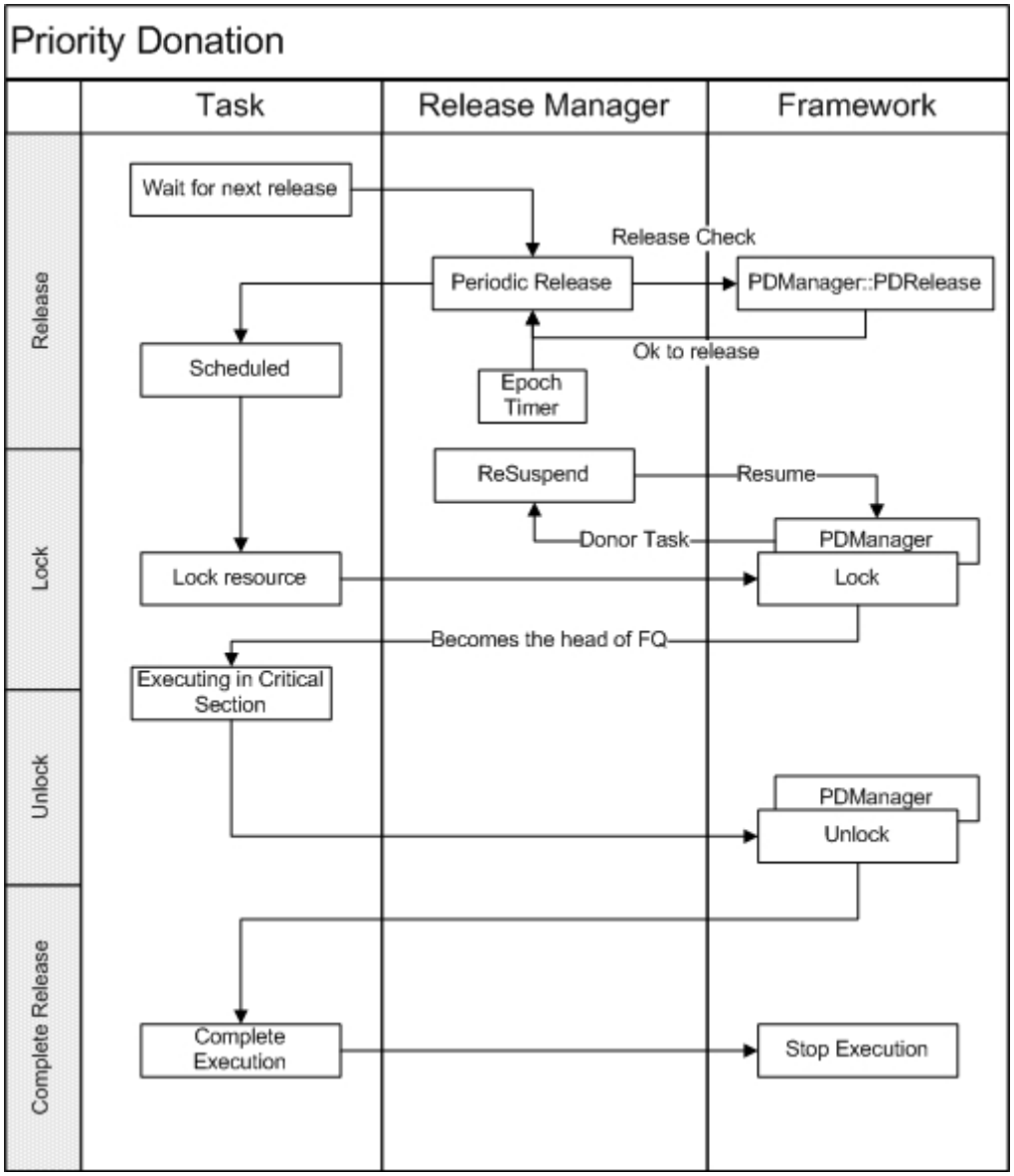

Figure 5.2: Priority Donation Main Components Interaction

current task can be released after setting up the time event. If so, the guard of the Wait_For_Next_Release entry will be open and the task will be released. The tasks will remain blocked on the guard otherwise.

```
1  package Release_Mechanisms.Periodic is
2    protected type Periodic_Release(S: Any_Periodic_Task_State) is new
         Release_Mechanism with
3      entry Wait_For_Next_Release;
4       pragma Priority(System.Priority'Last);
5      procedure Release(TE : in out Timing_Event);
6      procedure Allow_Release;
7      procedure ReSuspend;
8    private
9      Event : Timing_Event;
10     Next : Time;
11     New_Release : Boolean := True;
12     First: Boolean := True;
13   end Periodic_Release;
14 end Release_Mechanisms.Periodic;
```

Listing 5.16: Periodic Release Manager Specification

```
1  package body Release_Mechanisms.Periodic is
2    protected body Periodic_Release is
3      entry Wait_For_Next_Release when New_Release is
4      begin
5        if First then
6          First := False;
7          Epoch_Support.Epoch.Get_Start_Time(Next);
8          Next := Next + S.Period;
9          Event.Set_Handler(Next, Release'Access);
10         New_Release := False;
11         requeue Periodic_Release.Wait_For_Next_Release;
12       else
13         New_Release := False;
14       end if;
15     end Wait_For_Next_Release;
16
17     procedure Release(TE : in out Timing_Event) is
18     begin
19       Next := Next + S.Period;
20       TE.Set_Handler(Next, Release'Access);
21       if ReleaseCheck(Allow_Release'Access) then -- callback to the
             framework
22         New_Release := True;
23       end if;
24     end Release;
25
26     procedure Allow_Release is
27     begin
28      New_Release := True;
29     end;
30   end Periodic_Release;
31 begin
```

```
32    null;
33 end Release_Mechanisms.Periodic;
```

Listing 5.17: Periodic Release Manager Body

Given this approach, we are now able to provide full controls of the tasks releases through interacting with the Ada periodic release manager. The full implementation is given in Appendix A. At the abstraction level, as shown by Listing 5.18 and Figure 5.3, the implementation of the priority donation must make decisions on whether the current task can be released and pass that decision to the release manager. If the current task can be released, the "ReleaseCheck" function will return true. At every call to the "ReleaseCheck", the "Allow_Release" procedure is passed onto the framework. Since every task in the system has its individual release manager, the framework keeps track of the matching between the tasks and the "Allow_Release" procedures. In this way, the framework may release a particular task by making a callback to the low level release manager through its associated "Allow_Release" procedure.

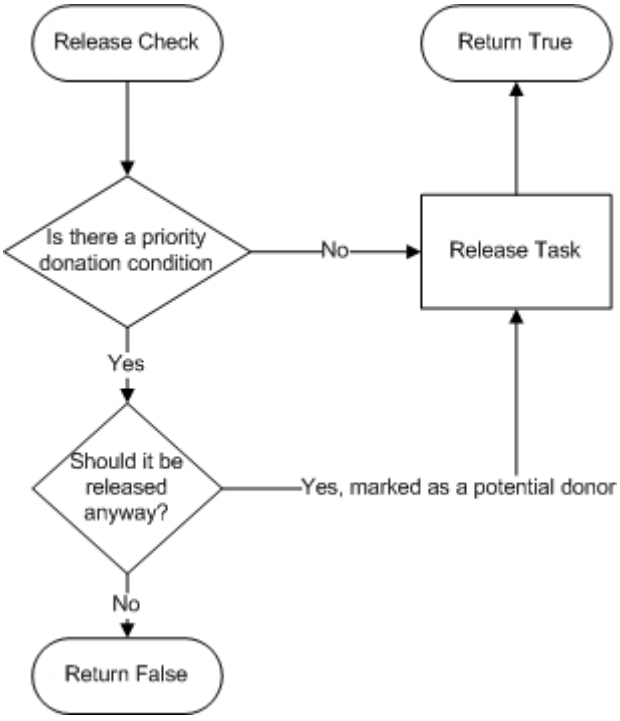

Figure 5.3: Priority Donation Release Check

Figure 5.3 demonstrates the release check routines of the framework. When the framework is called upon a release check, the calling task will be tested if it is a priority donor task. If so, the framework will return false. The calling task will therefore not be released. If it is a potential donor task, it is marked with special flag and released. However, this task may not issue any resource request. A potential donor task will call the "ReSuspend" procedure to get itself suspended in the resource locking procedure. Using the Ada synchronous task control package, the implementation of the "ReSuspend" procedure is able to actively control the release sequence of these suspending tasks. The task is released normally otherwise.

The priority donation protocol was implemented as a protected type and assigned one per each cluster. As it is only used by tasks in the same cluster, it can have a different protected object or protocol. The priority donation manager may intercept the execution of a task at various stages. At the release stage, the "PDRelease" called via the "ReleaseCheck" function will check the calling task against the priority donation rules. The function will return false to the underlying Ada facilities if the task has failed the test. As mentioned in previous chapters, the implementation of protected function is the same as the protected procedures in GNAT. Mutual exclusion is implemented for the accesses of the "PDRelease". The "ReDonate" implements priority donation at every release of shared resources. This procedure was specially designed for enforcing the next resource holding task to be scheduled if it may be preempted by the release of a higher non resource holding priority task. Whenever the priority donation is finished, the "StopDonation" procedure will be called and the DonationLog will be updated.

```
1  package OMLP is
2
3     Num_of_Clusters : constant Integer := 2;
4     Num_of_Proc : constant Integer := 2;
5
6     type Global_OMLP is new Protected_Controlled with private;
7
8     overriding procedure initialize (C:in out Global_OMLP);
9     overriding procedure Finalize (C:in out Global_OMLP);
10    overriding procedure Lock(C:in out Global_OMLP; L:Lock_Type; V:
          Lock_Visibility; Ceiling:Priority; Tid:Task_Id );
11    overriding procedure Unlock(C:in out Global_OMLP; Tid:Task_Id);
12
13    type Release_CallBack is access protected procedure;
14    function ReleaseCheck (CallBack:Release_CallBack) return Boolean;
15    procedure stop;
16    power_on : Time;
```

```
17
18 private
19   -- see appendix
20 end OMLP;
```

Listing 5.18: Priority Donation Specification

Figure 5.4 demonstrates the implementation routines of the priority donation lock procedure where tasks are calling for locking shared resources. When a task attempts to lock a shared resource, it is firstly evaluated in order to determine whether it is a donor task. If so, according to the priority donation rules, it is not eligible for acquiring resources and should be suspended immediately. When it passes the test, it checks if the resource is already locked. If the resource is free, it acquires the resource and returns. If some other task is holding the resource, it becomes blocked at an appropriate position either in the priority queue (PQ) or FIFO queue (FQ). It will only return from the lock procedure if it becomes the head of the FQ with the resource locked.

In the unlock procedure, as shown by Figure 5.5, since the resource request is finished, any task donating its priority to this priority recipient should be resumed and released. After that, according to the rules of cluster OMLP, the highest priority task from the PQ should be released and migrated into the FQ. If there is no other task waiting for the resource, the unlocking task will simply return. If there is some other task waiting, the calling task will check if the next resource holding task can be resumed successfully. If not, there must be a higher priority task released and that task should be suspended with its priority donated. The unlock procedure will swap the priority of these tasks making sure the high priority released task will not preempt the resource holding task when it is scheduled for execution.

The evaluation test comprises 6 tasks running in 2 clusters with two processors each. R1 and R2 are two priority donation shared resources. It specifies that the resource is shared in a system with 2 clusters and each cluster should contain 2 processors. The resource allocation between tasks are depicted by the following table:

The evaluation scenario was specifically designed to check the priority donation mechanisms. Low priority tasks are set with large execution time so that, when high priority task becomes released, the low priority resource acquiring task will be the priority recipient and blocks the high priority task. Each evaluation task

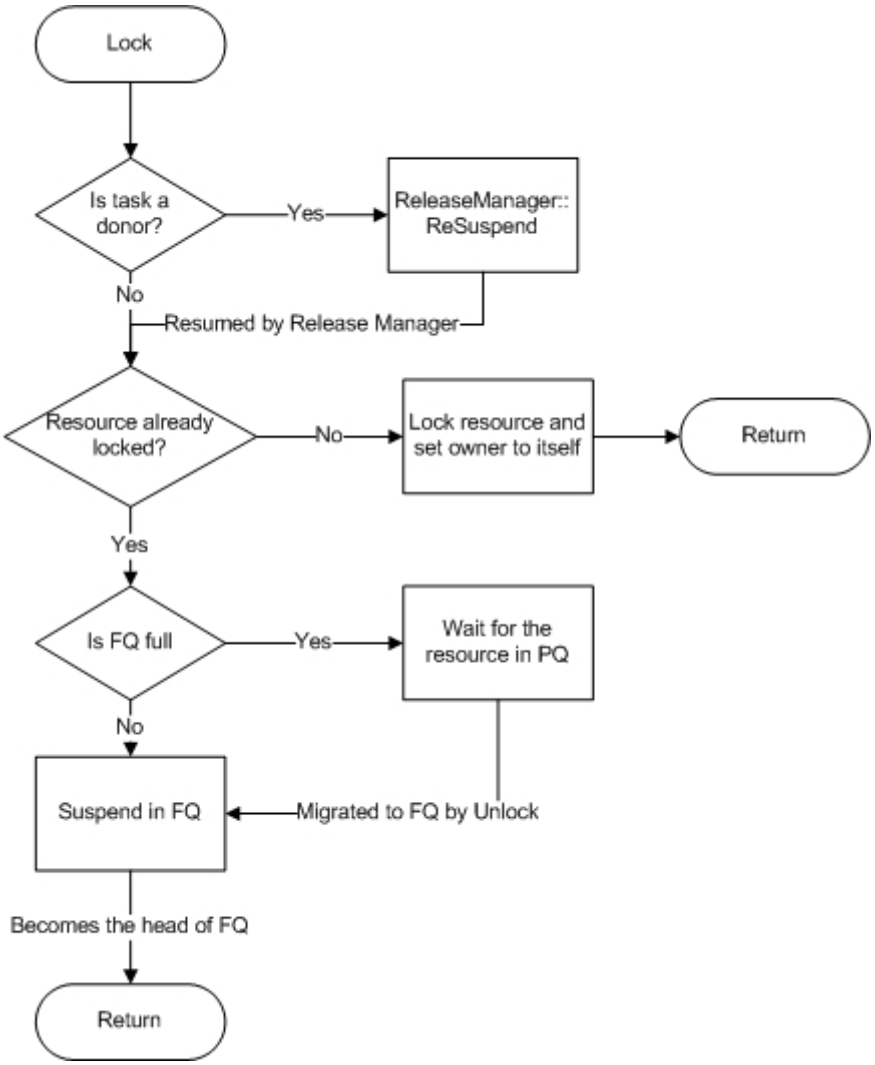

Figure 5.4: Priority Donation Lock Routine

Task	Cluster Affinity	Requiring Resource	Priority	Release Sequence
T1	1	R2	3	3
T2	1	nil	2	1
T3	1	R1	1	1
T4	2	R2	6	4
T5	2	nil	5	2
T6	2	R1	4	2

Table 5.9: Priority Donation Resource Table

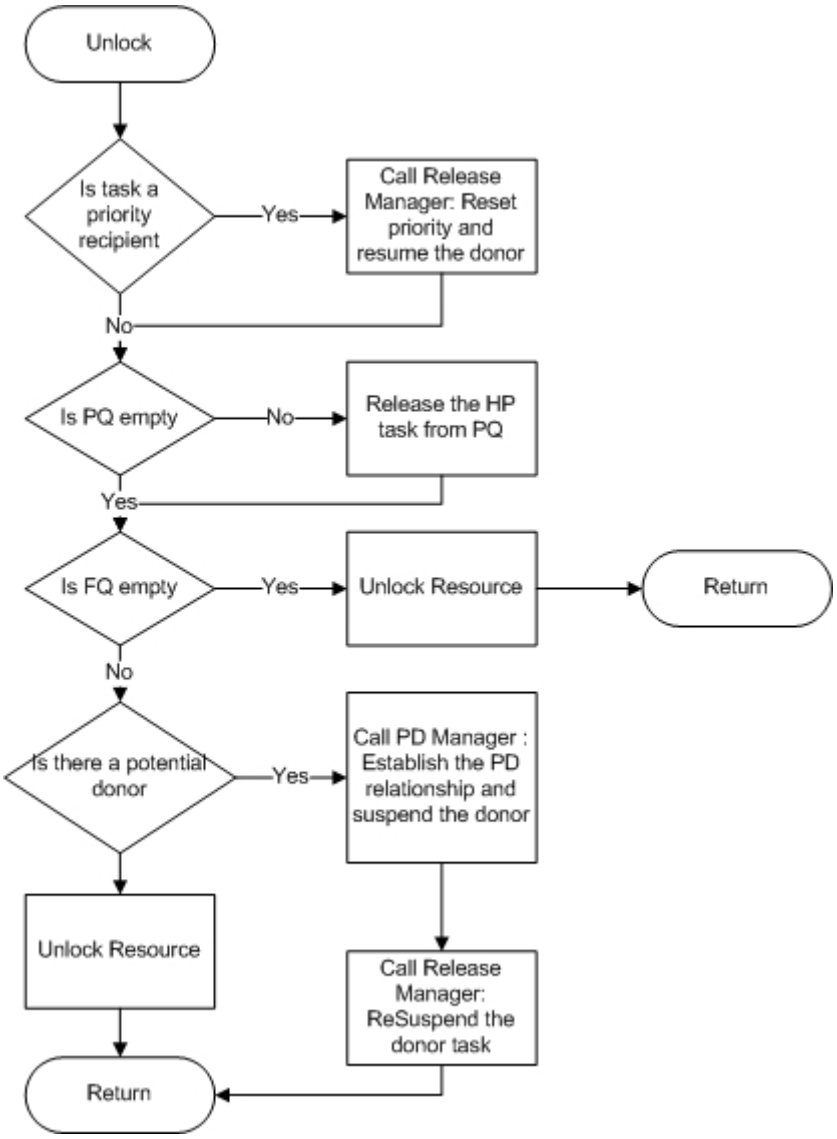

Figure 5.5: Priority Donation Unlock Routine

in the test calls the setCluster procedure before being released. This sets the cpu mask of the calling task to the range specified by the passed in parameters. The tasks is then allowed to migrate between the allowed processors.

Tasks are set to be released at various times. The setting was designed to promote one priority donation on each individual cluster. T3 and T2 are firstly released on cluster 2 followed by T6 and T5 on cluster 1. T1 is released at the same time as T3 attempts to lock R1. T4 is released to cluster 1 after T3 has obtained R1 on cluster 2. The latter two releases are specially designed to trigger the priority donating between tasks. The output of the evaluation program is as follows:

```
1  [ 1.00071] =Lock= : T3 joins FQ
2  [ 1.00076] =appendFQ= : inserting T3
3  T3 => Got R1
4  T5 => Executing!
5  [ 1.00082] =Lock= : T6 joins FQ
6  [ 1.00082] =appendFQ= : inserting T6
7  [ 1.00084] =PriorityDonation= : T4 donating its priority to T6
8  [ 1.00086] =PriorityDonation= : T1 donating its priority to T3
9  T6 => Got R1
10 [ 1.00117] =deleteFQ= : releasing T3
11 [ 1.00118] =PriorityDonation= : T1 ceases to be Priority Donor
12 [ 1.00120] =stop= : T5 stops execution
13 [ 1.00121] =Lock= : T1 joins FQ
14 [ 1.00123] =appendFQ= : inserting T1
15 [ 1.00124] =PriorityDonation= : T4 ceases to be Priority Donor
16 [ 1.00125] =Lock= : T4 joins FQ
17 [ 1.00125] =appendFQ= : inserting T4
18 [ 1.00128] =deleteFQ= : releasing T6
19 T4 => Got R2
20 [ 1.00135] =Lock= : T2 joins FQ
21 [ 1.00135] =appendFQ= : inserting T2
22 [ 1.00151] =stop= : T3 stops execution
23 [ 1.00152] =deleteFQ= : releasing T4
24 [ 1.00154] =stop= : T4 stops execution
25 T2 => Got R2
26 [ 1.00165] =deleteFQ= : releasing T2
27 T1 => Got R2
28 [ 1.00176] =stop= : T2 stops execution
29 [ 1.00176] =stop= : T6 stops execution
30 [ 1.00178] =deleteFQ= : releasing T1
31 [ 1.00179] =stop= : T1 stops execution
```

Listing 5.19: OMLP Evaluation Results

The priority donation protocol is a complex algorithm where priority donating mechanism only take place in certain scenarios. The test program is configured to release the tasks in a specific order agreed by the original scenario in Figure 2.9 to

ensure the protocol is tested and verified. The evaluation time are mapped to the time stamps of the original scenarios in Table 5.10

Evaluation Stamped Time	**Original Scenario Time**	**Event**
1.00000-1.00071	t0-t2	T2 and T3 get released on cluster 2 T5 and T6 get released on cluster 1 T3 attempts R1
1.00082	t3	T6 attempt to lock R1
1.00109-1.00117	t4	T3 unlocks R1
1.00135	t5	T2 attempts to lock R2
1.00152-1.00154	t6	T4 unlocks R2 and stops execution
1.00165	t7	T2 unlocks R2
1.00179	t8	T1 unlocks R2 and stops execution

Table 5.10: Mapping time stamps to Figure 2.9

With the time stamps mapped to the original scenario, the implementation is credited because the scheduling outcome were inline with the expectation. The whole output is analyzed further against the original proposal as follows:

Priority Donating At time 1.0084, T4 has been released to the release manager for scheduling. It checks the cluster record and found T5 and T6 are already executing. Both tasks have lower priority. The lowest priority task, T6, has issued resource request at time 1.00082. It therefore decided to donate its priority to T6 and suspend from execution. Similarly, T1 donates its priority to T3 which is the lowest priority task in cluster 1. Therefore, T6 was able to carry forward its resource request and obtained R1 at 1.00084. At time 1.00117, T3 has finished R1 and informed the priority donation manager the priority donation relationship should be ceased. T1 was therefore resumed by the priority donation manager at time 1.00121. As the highest priority task, it immediately acquires R2 and obtained the resource at time 1.00165. T4, the priority donating task in cluster 2, was resumed by the unlocking of T6 at time 1.00128. It successfully obtained R2 and finished execution at time 1.00154.

Nested Protected Object The implementation of the priority donation protocol incorporates two internal protected objects. The priority donation

manager is associated with a series of protected entries and procedures to facilitate the release mechanism of the priority donation protocol respectively. The evaluation scenarios rely on the original Ada protected object shared protocol to order the tasks waiting to be released on the release entry families. However, the application developer may want to introduce their own protocol to regulate the release order of tasks in the release manager. The application developer can create a new resource sharing protocol using the framework and attaching it to the protected objects. The framework therefore will dispatch the application defined routines to schedule the release waiting tasks. All other tasks operating outside the release manager will execute as normals.

The priority donation protocol of OMLP is a complex algorithm but it is effective in testing the flexibility and effectiveness of the framework. It involves closer interaction with the scheduler compared with previously examined protocols. The tasks involve blocking from the start to the finish of their executions. Multiple queue policies are also required for the normal execution of the protocol. However, the evaluation test argues that the framework together with the language facilities provided by Ada is sufficient in providing all above services to the application developers. The application developers may even have nested protected objects with different resource sharing protocol by using the framework.

5.7 Summary

This chapter evaluates the flexibility of the framework in supporting different resource sharing protocols by verifying their prototype implementations. The evaluation test starts with MSRP. MSRP is a spin based protocol with resource waiting tasks spinning on unavailable global resources. Since the MSRP prototype implementation can generate the same scheduling result as expected by the original MSRP semantics, the implementation demonstrates that spin lock based protocols (like MSRP) can be fully supported by the protocol. The evaluation tests are then carried forwards to suspension lock based protocols like MPCP. By having all tasks suspending on different locks, the MPCP implementation demonstrates the ability of the framework of having explicit control of task ordering and different locking

policy on different resources. The FMLP protocol evaluates the framework further by imposing restrictions and differentiations on the shared resources. Since the framework accepts customized configuration parameters, the application developers can pass in the type of the resource at runtime. The application developers can also collect different shared resources into one resource group. By holding the group lock, protocols like FMLP can support nested locks in the framework. The OMLP protocol is sophisticated because of its complex queue operation. The evaluation of the framework demonstrates that the application developers may deploy different queuing policies on a single resource in the framework. The framework is then evaluated against a more complex algorithm requiring close interaction with the underlying scheduler (clustered OMLP with priority donation). The priority donation protocol requires release blocking, tracking of the highest executing priority task and cluster based scheduling. Priority donation was implemented with the framework by having direct control of the tasks on release. Priority donating tasks are suspended from release until their priority recipients have finished execution and returned their priority back.

Chapter 6

Conclusions and Future Work

This chapter concludes the thesis by summarizing the key contributions of the research. The thesis goals, hypothesis and contributions are reviewed. The future directions of this research are also discussed in conjunction with the limitations of the research.

6.1 A Summary of the Key Thesis Contributions

The performance of multiprocessor systems with resource sharing is affected by many factors. As described in Chapter 1, the scheduling algorithms may have a distinctive impact on the system performance. Globally scheduled systems have better processor utilization in comparison to partitioned systems. Partitioned scheduling has an advantage in having better schedulability analysis results by implementing well studied uniprocessor algorithms on the partitioned processors. For a system with large number of processors, a cluster-based scheduling may give better scalability.

Sharing resources in multiprocessor systems is never a simple task. A shared resource in multiprocessor systems may block the tasks in many different ways: direct blocking, remote blocking, transitive blocking and priority blocking etc. Furthermore, the tasks in multiprocessor systems are likely to suffer more blocking than in uniprocessor systems. The multiprocessor resource sharing algorithms were designed to reduce the blocking suffered by the highest priority task. As described in Chapter 2, MPCP, a partitioned suspension based protocol, deploys

the global resource sharing tasks to an absolute higher priority range to prevent them from being blocked by local tasks. MSRP, a partitioned spin based locking protocol, implements spin locks on tasks waiting for global resources. More advanced mechanisms were introduced by later algorithms. The group lock was introduced by FMLP where tasks are allowed to acquire nested resource if they obtained the group lock first. OMLP introduces the possibility of having dual queuing policies for one shared resource and priority donation to overcome the drawbacks of priority inheritance in cluster scheduled systems. As shown in Table 2.5, all these algorithms have different characteristics in terms of the choice of scheduling algorithms, resource categorization, nested resource support, resource accessing priority and the choice of task queuing policy. The significance of those choices will have a great impact on the performance of the algorithms. For example, suspension based protocols will have better performance with long critical sections where the suspension overhead is fractional to the length of the whole critical section. Although spin based protocols are more efficient in short critical sections, high priority tasks may suffer more blocking than if guarded by the suspension based protocols [13]. The performance of the multiprocessor resource sharing algorithms is largely dependent on the application scenarios. It is therefore inappropriate to introduce a particular protocol to be used for all applications.

Therefore, the thesis was motivated to introduce a flexible resource sharing framework as recalled by the thesis hypothesis:

> **The performance of a multiprocessor resource sharing protocol is largely dependent on the application semantics. This thesis contends that it is, therefore, inappropriate to introduce support for a particular multiprocessor resource sharing protocol into a programming language definition. Instead, a language should support a framework that allows a variety of protocols to be implemented (either by the programmer or via pre-written standard libraries). A flexible framework can be applied to a wide range of multiprocessor resource- sharing protocols with relatively small overheads.**

This thesis provides three key contributions. The first contribution is the proposal of the flexible resource sharing framework. The proposal defines the interac-

tions between the systems and the application developers on how the application defined resource sharing protocols can be integrated with the systems. Different systems were considered for this feasibility of applying the framework.

The second contribution of the thesis is to investigate the details of incorporating the framework with Ada. Integrating the framework with any system is challenging because the framework should not violate the rules and semantics of the original facilities. It should provide the application developers with the confidence that, by following the framework, they are protected from the low level interactions hazards. The protected objects in Ada may have Ceiling_Locking, a predefined protocol of Ada, where tasks cannot have a higher priority than the protected object ceiling when accessing the resource. The Real-Time Systems Annex of the Ada reference manual allows implementations to define a new relationship between the priority of the tasks and the priority of the protected objects (AARM D.3 par 6.2). This offers the scope for our framework to introduce application-defined protocols to work with Ada's original routines. The application-defined protocols can therefore be integrated with the Ada language through our framework while the original Ada runtime library routines remains largely intact. This contribution revisits the semantics of Ada according to the Ada reference manual and the prototype implementations of the framework. The work defines transactions of the framework with Ada runtime in flow diagrams. The evaluation programs investigates the overheads of the framework by running the prototype implementations in a simulated environment. The overhead incurred at the expense of running the framework is fractional to the total execution overheads.

The third contribution, perhaps the most challenging one, is to argue the framework is capable of supporting with different multiprocessor algorithms. The challenge of working with different resource sharing algorithms is that they all have different interaction requirements for the framework and the underlying systems. The framework was evaluated against the prototype implementation of MPCP for supporting the suspension based locking protocols. It has also tested against the spin based protocols like MSRP. The challenge of having nested resource requests is met by testing against the implementation of FMLP. Supporting sophisticated queuing policy and sharing resources with cluster scheduling is verified by implementing the OMLP family protocols. By running through all these prototyped evaluation tests, the thesis argues that the framework is flexible enough to coop-

erate with all these algorithms.

6.2 Limitations of this Work

The limitation of this work is constrained by the assumptions of the framework. As depicted by the implementations of the protocols evaluated in Chapter 5, the interaction with the framework is assumed to occur only on resource accesses. This assumption is valid for protocols like MPCP where tasks are scheduled in normal priority order when executing outside of their critical sections. The priority inversion normally occurs at the event of locking when the resource holding task blocks high priority tasks. The conventional resource sharing protocols are then deployed here to avert this priority inversion. However, when sophisticated protocols are required, such as blocking tasks outside the critical sections on a precautionary basis, this assumption must be relaxed for the framework to provide such control to the application developers. The implementation of the priority donation algorithm illustrates how this assumption is relaxed.

The tasks scheduled by the framework are also assumed to execute their own critical sections. This assumption is inherited from the literatures introduced in earlier chapters where schedubility analysis is carried out on per task basis. However, at the OS level, a task may contain memory blocks, resource requirements, operation blocks and many other decomposable parts. It is possible that its critical section may be executed by other tasks as long as global memory blocks access is allowed. New algorithms like SPEPP are imposing this challenge to the framework. The framework needs to be adapted to more closely linked with the scheduler to incorporate such changes.

The other important limitation of this work is the evaluation approach. As mentioned in Chapter 4, the framework introduces new specifications to the aspect specification to associate the application-defined protocols. The full implementation of this will require an update to the front end compiler from the parser to the RTL language. This work is entensive and beyond the scope of this research. Therefore, the simulation approach was adopted instead. This prototype evaluation approach is then focused on verifying the flexibility of the framework instead of obtaining the optimal performance figure of the framework. However, with a full implementation, where the front end compiler is changed, the performance of

the framework can be measured more accurately.

6.3 Future Work

The main future research work of this thesis is to relax the second assumptions of the supported algorithms as mentioned earlier. That is the tasks may be allowed to execute the critical sections on behalf of other tasks. This enables the framework to support new algorithms like SPEPP [61] (SPEPP stands for "Spinning Processor Executes for Preempted Processors"). Task in SPEPP decouples a memory block when it attempts to access a shared resource. The memory block contains the preserved memory for the input and output parameters used when accessing the resource. All memory blocks from different tasks are ordered in a FIFO queue with a spin lock. Once queued, the resource holding task at the head of the queue executes all its critical sections of the other tasks in the queue in sequence using the appropriate memory block until it has reached its own memory block or it is interrupted. The other tasks which have inserted memory blocks into the queue will periodically check the status flag in the queue to see whether their memory blocks have been executed or not. The algorithm is fully explained in Chapter 2.

A possible implementation of SPEPP would evaluate the feasibility of integrating of the framework with Ada protected entries. This is because the semantics of SPEPP closely match those of Ada protected entries. Evidenced by the assembly code shown in Chapter 4, the protected entry is compiled as a protected procedure if the barrier was always set to true. In essence, the protected entry is a special form of protected procedure with extra initial function calls for barrier evaluation and different service routines. The unique implementation of Ada protected entries offers an opportunity for the framework to be introduced in a similar manner to the protected procedures. Although Ada has implemented the protected entries using the proxy model internally, no such functionality is available to the end users. This also imposes a challenge to the framework to support interactions with low language level details.

```
1 with Ada.Finalization; use Ada.Finalization;
2 with System; use System;
3 with Ada.Task_Identification; use Ada.Task_Identification;
4 with System.Tasking.Protected_Objects;
5
6 package Ada.Protected_Object_Access is
```

```
7    type Lock_Type is (Read, Write);
8    type Lock_Visibility is (Local, Global);
9    type Protected_Controlled is new Limited_Controlled with private;
10
11   type Subprogram_Body_Id is private;
12
13   overriding procedure Initialize (C : in out Protected_Controlled);
14   overriding procedure Finalize (C : in out Protected_Controlled);
15
16   procedure Lock (C:in out Protected_Controlled; L:Lock__Type; V:
         Lock_Visibility; Ceiling:Priority; Tid:Task_Id:=Current_Task);
17   procedure Unlock (C:in out Protected_Controlled; Tid:Task_Id:=
         Current_Task);
18
19   procedure Lock(C:in out Protected_Controlled; Ceiling:Priority; Tid
         :Task_Id:=Current_Task; Body_Id: in out Subprogram_Body_Id);
20   procedure Unlock (C:in out Protected_Controlled; Tid:Task_Id:=
         Current_Task; Body_Id: in out Subprogram_Body_Id);
21
22   function Evaluate_Barrier (Body_Id:Subprogram_Body_Id) return
         boolean;
23   procedure Execute_Body(Body_Id:Subprogram_Body_Id);
24
25 private
26    type Subprogram_Body_Id is
27     record
28      Index : System.Tasking.Protected_Objects.Protected_Entry_Index;
29      O     : System.Address;
30      P     : System.Address;
31      B     : System.Tasking.Protected_Objects.Entry_Body;
32     end record;
33    -- implementation defined
34 end Ada.Protected_Object_Access;
```

Listing 6.1: Ada API for the Framework

Providing such information at the framework is essential for the framework to support Ada protected entry like protocols such as SPEPP where the resource holding task may execute others critical sections. This is owing to the fact that the application defined protocols may need to access the entry barriers and bodies of other tasks. The "Subprogram_Body_Id" is proposed for this purpose that low level function pointers to be encapsulated in this type. The application defined protocols are shielded from handling the low level details but saving this record from the framework. A "Subprogram_Body_Id" will be instantiated with three pointers (function pointer for the protected entry, barrier function and the body) and the index of its position in the Ada entry queue. Once created, the application programmer will only need to concern the maintenance of the entry queue. Two essential auxiliary functions are provided to evaluate the barrier and execute the

body. These function are provided at the framework so that the barrier and the body of the entries could be easily executed with its "Subprogram_Body_Id".

```
1  package SPEPP is
2
3    type SPEPP_Protocol is new Protected_Controlled with private;
4
5    type SPEPP_Record is private;
6    type SPEPP_Record_Access is access SPEPP_Record;
7
8    procedure Initialize (SP : in out SPEPP_Protocol);
9    procedure Finalize (SP : in out SPEPP_Protocol);
10
11   procedure Lock (C:in out SPEPP_Protocol; L:Lock__Type; V:
         Lock_Visibility; Ceiling:Priority; Tid:Task_Id);
12   procedure Unlock (C:in out SPEPP_Protocol; Tid:Task_Id);
13
14   procedure Lock (SP: in out SPEPP_Protocol; Ceiling:Any_Priority;
         Tid:Task_Id; Body_Id : in out Subprogram_Body_Id);
15   procedure Unlock (SP: in out SPEPP_Protocol; Tid:Task_Id; Body_Id:
         in out Subprogram_Body_Id);
16
17  private
18   type spin_queue is array (1..50) of Spin_Lock;
19
20   type SPEPP_Protocol is new Protected_Controlled  with
21     record
22       SLock : Spin_Lock;
23       RQ    : spin_queue;
24       PO_Executed : spin_queue;
25       RQ_Index : integer;
26       RQ_total : integer;
27       Entry_Head : SPEPP_Record_Access;
28       global : integer;
29       tid : Task_Id;
30     end record;
31
32   type SPEPP_Record is
33     record
34       bid : Subprogram_Body_Id;
35       next : SPEPP_Record_Access;
36     end record;
37
38 end SPEPP;
```

Listing 6.2: SPEPP Specification

Following the updates to the framework, the SPEPP algorithm can be implemented as shown by Listing 6.2. The SPEPP specification extends the framework by instantiating the newly defined lock and unlock procedures. The protected entry queue is defined in the SPEPP_Protocol record to keep track of the waiting tasks for the resource. With this specification, all tasks waiting for the shared

resources are spinning to a dedicated lock in the queue. The resource holding task holds all the lock and may release a particular task if its critical section has been executed.

The other main direction of the future research work is to provide deadlock free guarantees to the application developers. The framework offers the freedom for the application developers to implement any locking protocols by inheriting the framework. At the moment, the framework relies on the integrity of the protocols to ensure they are deadlock free. There is a possibility that the application codes may contain deadlocks which will be critical to the systems. The framework may be extended with deadlock prevention techniques to handle such situation. A deadlock can occur if the following four conditions are satisfied:

Mutual Exclusion only one task can use a resource at once.

Hold and Wait there must exists tasks which are holding resources while waiting for others.

No Preemption A resource can only be released voluntarily by a task

Circular Wait A circular chain of tasks must exist for tasks to cross waiting for each other's resource.

The framework may introduce a deadlock prevention approach to dynamically detect if the deadlock condition exit. If so, the framework should reject the resource request of the task and prevent the deadlock from happening. For example, the framework may impose compulsory priority orders for the nested resource in order to prevent circular waiting. If a task using an application-defined resource sharing protocol attempts to acquire a nested resource with lower priority than its first resource, the attempt should be rejected by the framework.

6.4 Final Words

The contribution of this thesis enables Ada application developers to implement the most suitable multiprocessor resource sharing algorithms for their operating scenarios. In the development of the traditional uniprocessor real time applications, only the predefined resource sharing algorithms are available to the application developers. This approach was convenient for the uniprocessor systems as

the research community has identified the optimal algorithm for their specific scenarios. However, this constraint was broken in multiprocessor systems. Currently multiprocessor resource sharing algorithms are still in their infancy and there is no optimal solution. A flexible resource sharing framework therefore seems to a practical solution for the application developers.

The real time community still maintains widely diverging views as to how resources should be shared in multiprocessors. The amount of interest in non-blocking and non-locking methods is good evidence of this. However, it is widely accepted that the success of a multiprocessor resource sharing protocol depends on resources being shared as efficiently as possible with satisfaction to all real time requirements. It is still unknown whether an optimal algorithm will be proposed for sharing resources in multiprocessor systems. At the moment, divide and conquer seems to be a good strategy.

Appendix A

Priority Donation Implementation

This appendix give the full prototype implementation of the priority donation algorithm discussed in Chapter 5. The code given in Chapter 5 was an abbreviation from this full implementation for the demonstration purposes. Due to the complexity of the program and for the presentation purposes, the full implementation is only given in this appendix. The package specification was given by Listing A.1.

```
1  package OMLP is
2
3     package SQL renames System.Multiprocessors.QueueLock;
4     package ART renames Ada.Real_Time.Timing_Events;
5
6     Num_of_Clusters : constant Integer := 2;
7     Num_of_Proc : constant Integer := 2;
8
9     type Global_OMLP is new Protected_Controlled with private;
10
11    overriding procedure initialize (C:in out Global_OMLP);
12    overriding procedure Finalize (C:in out Global_OMLP);
13    overriding procedure Lock(C:in out Global_OMLP; L:Lock_Type; V:
          Lock_Visibility; Ceiling:Priority; Tid:Task_Id );
14    overriding procedure Unlock(C:in out Global_OMLP; Tid:Task_Id);
15
16    type Release_CallBack is access protected procedure;
17    function ReleaseCheck (CallBack:Release_CallBack) return Boolean;
18    procedure stop;
19    power_on : Time;
20
21 private
22       data_lock : aliased SQL.Spin_Lock;
23       type Q_record is
24           record
```

```
25              lock : SQL.S_Object;
26              id : Ada.Task_Identification.Task_Id;
27              pri: Integer;
28          end record;
29
30    type Q_Arr_t is array (1..SQL.Num_of_Task) of Q_record;
31    type PQ_Arr_t is array (1..2, 1..SQL.Num_of_Task) of Q_record;
32    type N_Arr_t is array (1..SQL.Num_of_Task) of integer;
33      Global_FQ : Q_Arr_t;
34      Global_FQH, Global_FQL : integer;
35
36    type Global_OMLP(CSize: integer; proc : integer) is new
          Protected_Controlled with
37      record
38        FQ : Q_Arr_t;
39        PQ : PQ_Arr_t;  -- one PQ per cluster
40        FQ_Lock : SQL.Spin_Lock;
41        FQH, FQL: integer :=1;
42        PQH, PQL: N_Arr_t := (others=>0);
43      end record;
44
45    type clusterset_t is array (1..10) of integer;
46    clusterset : clusterset_t :=(2,2,2,1,1,1,0,0,0,0);
47    Cluster_Lock : aliased SQL.Spin_Lock;
48    type cluster_record_t is
49      record
50          id : Ada.Task_Identification.Task_Id;
51          lock : SQL.S_Object;
52          req : boolean;  -- issued resource request?
53      end record;
54    type cluster_t is array (1..CSize, 1..proc) of cluster_record_t;
55    cluster : cluster_t;
56    c1, c2 : integer := 1;
57
58  end OMLP;
```

Listing A.1: Priority Donation Specification

Listing A.2 demonstrates the implementation details of the priority donation algorithm. The callback functions, ReleaseCheck and Release_Now, are defined at line 4 to 22. The priority donation manager is given at line 25-58. The conventional framework methods are given from line 100 onwards.

```
1  package body OMLP is
2
3      -- Release Manager Interaction Facilities
4      function ReleaseCheck (CallBack:Release_CallBack) return Boolean
           is
5          cid: integer;
6          id : Task_Id;
7          CB : Release_CallBack;
8      begin
9         SQL.SimpleSpinLock(data_lock); -- data lock
```

```
10          id := current_task;
11          append_CB(id, CallBack);
12          cid := (SQL.GetAffinity / 3) +1;
13          return PDManager(cid).PDRelease(id, cid);
14       end;
15
16       procedure Release_Now (id:Task_Id) is
17       begin
18         index := Find_Task_Index_In_DQ(cid, id);
19         cluster(cid, index).exe := True;
20         cluster(cid, position).donor := False;
21         Get_CB(id).CB.all;
22       end;
23
24    -- Priority Donation Manager
25    protected body PriorityDonationManager is
26
27       function PDRelease (id:Task_Id; cid:Integer) return Boolean is
28           position, index : integer;
29       begin
30         position := append_cluster_DQ(cid, current_task);
31         cluster(cid, position).exe := True;
32         -- Note: At most Num_of_Proc tasks exe flag will be true at any
               time.
33
34         if Find_avail_Processor(cid)=-1 then -- if there is no
              available processor
35           index := index_of_Cth_HP_task_in_cluster(cid).DQ;
36           if All_Resource_Flag(cid)=True and cluster(cid, index+1).id/=
                Null_Task_Id and cluster(cid, index+1).req=True then
37           -- all previous tasks have issued resource request and the C
                +1 highest priority task
38
39             if cluster(cid, index+1).exe=True then -- if the C+1
                  highest priority task is executing
40               Print_Time;
41               Put_Line(" =PriorityDonation= : "&image(id)(1..2)&"
                    donating its priority to "&image(cluster(p, pid).id)
                    (1..2));
42               cluster(cid, position).donor := True;
43               cluster(cid, position).exe := False;
44               DonationLog(LogLength).recipient := cluster(p, index+1).
                    id;
45               DonationLog(LogLength).donor := id;
46               LogLength := LogLength + 1;
47               Set_Priority(DonationLog(LogLength).recipient,
                    Get_Priority(id));
48               SQL.SimpleSpinUnlock(C.data_lock); -- data lock
49               return False;
50             else -- if it is suspended
51               cluster(cid, position).donor := True;
52             end if;
53           end if;
54         end if;
```

```
55        Release_Now(current_task);
56        SQL.SimpleSpinUnlock(C.data_lock); -- data lock
57        return True;
58      end;
59
60      procedure StopDonation (id : Task_Id; cid : Integer) is
61         p : integer;
62        begin
63        SQL.SimpleSpinLock(data_lock); -- data lock
64        Print_Time;
65        -- reset exe flag in DQ
66        for i in DonationLog'range loop
67          if DonationLog(i).recipient = id then
68             Set_Priority(DonationLog(LogLength).donor, Get_Priority(
                  DonationLog(LogLength).donor));
69             cluster(cid, DonationLog(i).pid).id := Null_Task_Id;
70             DonationLog(LogLength).recipient := Null_Task_Id;
71             Release_Now(DonationLog(LogLength).donor);
72             DonationLog(LogLength).donor := Null_Task_Id;
73             SQL.SimpleSpinUnlock(C.data_lock); -- data lock
74             return;
75          end if;
76        end loop;
77        SQL.SimpleSpinUnlock(C.data_lock); -- data lock
78      end;
79
80      procedure ReDonate is
81        tid1, tid2, tid3 : Task_Id;
82        cid, pri1: Integer;
83       begin
84        tid1 := C.FQ(C.FQL).id;
85        cid := C.FQ(C.FQL).ccid;
86
87        tid2:=the_HP_EXE_task_with_no_resource_request_in_the_cluster;
88        tid3:=the_LP_EXE_task_in_cluster_other_than_tid1_tid2;
89        -- tid3 might be null or the next task scheduled for execution.
90        pri2 := Get_Priority(tid2);
91        if tid1/= Null_Task_Id and tid2/=Null_Task_Id and Get_Priority(
             tid2)>Get_Priority(tid1) and tid3/=Null_Task_Id and
             Get_Priority(tid1)>=Get_Priority(tid3) then
92          Set_Priority(tid1, Get_Priority(tid2));
93          Set_Priority(tid2, C.FQ(C.FQL).pri);
94          C.FQ(C.FQL).pri := Get_Priority(tid2);
95          -- update Donation Log
96          -- set the req flag of tid2 of its home cluster DQ
97        end if;
98       end ReDonate;
99    end PriorityDonationManager;
100
101   overriding procedure initialize (C : in out Global_OMLP) is
102   begin
103     -- all initialization work here
104     end loop;
105
```

```
106   overriding procedure Finalize (C : in out Global_OMLP) is
107   begin
108     null;
109   end;
110
111     -- Clustered OMLP
112   overriding procedure Lock(C : in out Global_OMLP; L : Lock_Type; V
          : Lock_Visibility; Ceiling : Priority; Tid :  Task_Id ) is
113         cid, DQindex, PQindex, temp : integer;
114   begin
115     SQL.SimpleSpinLock(data_lock);
116     cid := (SQL.GetAffinity / 3) +1;
117
118     DQindex := Find_Task_Index_In_DQ(cid, id);
119     if cluster(cid, DQindex).donor=True then
120       -- priority lowered
121     end if;
122     cluster(cid, DQindex).req := True;
123     if (C.FQH-C.FQL)>=Num_of_Clusters then
124       cluster(cid, DQindex).exe := False;
125       Print_Time;
126       Put_Line(" =Lock= : "&image(current_task)(1..2) & " joins PQ");
127       PQindex := append_PQ(C, cid);
128       C.PQ(p, PQindex).pri := Get_Priority(current_task);
129       C.PQ(p, PQindex).ccid := cid;
130       SQL.SimpleSpinUnlock(C.data_lock); -- data lock
131       SQL.Suspend_Until_True(C.PQ(p, PQindex).lock); -- PQ lock
132       SQL.SimpleSpinLock(C.data_lock);  -- data lock
133       delete_PQ(C, p, PQindex);
134       Print_Time;
135       Put_Line(image(current_task)(1..2)&" joins FQ from PQ");
136     end if;
137     if C.FQ(C.FQL).id/=Null_Task_Id then
138       if C.FQ(C.FQH).id/=current_task then
139         C.FQH := C.FQH + 1;
140       end if;
141       cluster(cid, DQindex).exe := False;
142       -- update FQ record
143       SQL.SimpleSpinUnlock(C.data_lock); -- data lock
144       SQL.Suspend_Until_True(C.FQ(C.FQH).lock); -- FQ Lock
145       cluster(cid, DQindex).exe := True;
146       SQL.SimpleSpinLock(C.data_lock); -- data lock
147     end if;
148     Print_Time;
149     Put_Line(" =Lock= : "&image(current_task)(1..2) & " joins FQ");
150     C.FQ(C.FQL).id := current_task;
151     SQL.SimpleSpinUnlock(C.data_lock); -- data lock
152   end Lock;
153
154   overriding procedure Unlock(C : in out Global_OMLP; Tid :  Task_Id)
          is
155     cid, DQindex : integer;
156   begin
157     SQL.SimpleSpinLock(data_lock);
```

```
158       C.FQL := C.FQL + 1;
159       releasePQ(C); -- release the highest priority task from PQ
160       deleteFQ(C); -- release the next task in FQ and increase C.FQH
161       cid := (SQL.GetAffinity / 3) +1;
162       DQindex := Find_Task_Index_In_DQ(cid, id);
163       cluster(cid, DQindex).req := False;
164       PDManager(cid).StopDonation(current_task, SQL.GetAffinity);
165       PDManager(cid).deRegister(current_task);
166       PDManager(cid).ReDonate;
167       SQL.SimpleSpinUnlock(data_lock);
168       Set_Priority(SQL.Get_PML);
169    end Unlock;
170
171 end OMLP;
```

Listing A.2: Priority Donation Implementation

Appendix B

QueueLock Package

The System.Multiprocessors.QueueLock package was specifically created for the prototype evaluation implementations. It channels low level functionalities to the high level for the convenience of protocol implementations. It provides crucial functionalities including suspension, spinning and atomic test and set operations etc. The full implementation can be found as follows:

```
1  with System;
2  with System.Linux;
3  with System.Tasking;
4  with System.Task_Primitives;
5  with System.OS_Interface;
6  with Interfaces.C;
7  with Interfaces;
8  with System.Task_Primitives.Operations;
9
10 package System.Multiprocessors.QueueLock is
11    ------------------------------------
12    -------- Priority Patch Routines -----
13    ------------------------------------
14
15    procedure Set_Priority (pri : Integer);
16    -- default scheduling is SCHED_FIFO
17    function GetAffinity return integer;
18    -- Spin Locks --
19    type Queue_Order is (FIFO_Ordered, Priority_Ordered);
20    type Spin_Lock is private;
21    type RWLockT is private;
22
23    type pthread_spinlock_t is new Interfaces.C.int;
24    pragma Convention (C, pthread_spinlock_t);
25    type pthread_spinlock_status is limited private;
26    type Suspension_Lock is limited private;
27
28    ------------------------
```

```
29    ---- MSRP Routines ----
30    ------------------------
31    -- MAX PRIORITY 49;
32    -- MIN PRIORITY 1;
33    Num_of_Processors : constant Integer := 32;
34    Num_of_Task : constant Integer := 20;
35
36    -- Scheduler routine protection locks
37    MSRPQ : aliased pthread_spinlock_t;
38    type Ceiling_t is array (0..Num_of_Processors) of Integer;
39    type MSRP is record
40       Lock : Spin_Lock;
41       Ceiling : Ceiling_t;
42       Global : Boolean;
43    end record;
44
45 --  procedure Initialize;
46
47    procedure Release (CPU : Integer);
48    pragma Inline (Release);
49
50    procedure Lock (res : in out MSRP);
51    pragma Inline (Lock);
52
53    procedure Unlock  (res : in out MSRP);
54    pragma Inline (Unlock);
55
56    procedure Set_PML (P : Integer);
57    pragma Inline (Set_PML);
58
59    function Get_PML return Integer;
60    pragma Inline (Get_PML);
61
62    function GetSpinIndex return integer;
63    pragma Inline (GetSpinIndex);
64
65    procedure SetSpinIndex (i : integer);
66    pragma Inline (SetSpinIndex);
67
68    function GetSusIndex return integer;
69    pragma Inline (GetSusIndex);
70
71    procedure SetSusIndex (i : integer);
72    pragma Inline (SetSusIndex);
73
74    procedure setProc (CPU : Integer; C : Integer);
75    pragma Inline (setProc);
76
77    procedure Stop;
78    pragma Inline (Stop);
79
80    procedure MSRP_Init;
81
82    ----------------------------
```

```
83      ---- Assembler Routines ----
84      ---------------------------
85      function Sync_Val_Compare_And_Swap
86        (Destination : access Interfaces.Unsigned_32;
87         Comparand   : Interfaces.Unsigned_32;
88         New_Value   : Interfaces.Unsigned_32)
89         return Interfaces.Unsigned_32;
90
91      ---------------------------
92      ---- RWLocks Routines ----
93      ---------------------------
94      procedure RWInitialize(RWLock : in out RWLockT);
95
96      procedure RLock (RWLock : in out RWLockT);
97      procedure RUnlock (RWLock : in out RWLockT);
98
99      procedure WLock (RWLock : in out RWLockT);
100     procedure WUnlock (RWLock : in out RWLockT);
101
102     -------------------------
103     -- Spin Lock Routines --
104     -------------------------
105
106     procedure Initialize (SPL : in out Spin_Lock);
107     pragma Inline (Initialize);
108
109     procedure Acquire (SPL : in out Spin_Lock);
110     pragma Inline (Acquire);
111
112     procedure Release (SPL : in out Spin_Lock);
113     pragma Inline (Release);
114
115     function H_Queued_Priority (SPL : in out Spin_Lock)
116        return System.Any_Priority;
117
118     -----------------------------
119     -- Suspension Lock Routines --
120     -----------------------------
121     procedure Initialize (
122        SL : in out Suspension_Lock;
123        Ceil : System.Any_Priority;
124        QOrder : Queue_Order);
125     procedure Release (SL : in out Suspension_Lock);
126     pragma Inline (Release);
127
128     procedure Acquire (SL : in out Suspension_Lock);
129     pragma Inline (Acquire);
130
131     function H_Queued_Priority (SL : in out Suspension_Lock)
132           return System.Any_Priority;
133
134     function Get_Priority (Id : System.Tasking.Task_Id)
135        return System.Any_Priority;
136     procedure Set_Priority (
```

```
137             Id : System.Tasking.Task_Id;
138             Prio : System.Any_Priority);
139       function Current_Task
140          return System.Tasking.Task_Id;
141       function Head_Task (SL : Suspension_Lock)
142          return System.Tasking.Task_Id;
143       function Head_Task (SPL : Spin_Lock)
144          return System.Tasking.Task_Id;
145       function Is_Equal(Tid1 : System.Tasking.Task_Id; Tid2 : System.
             Tasking.Task_Id)
146        return boolean;
147       pragma Inline(Is_Equal);
148
149       ------------------------------
150       -- Simple Spin Lock Routines --
151       ------------------------------
152       procedure SimpleSpinUnlock (SPL : in out Spin_Lock);
153       procedure SimpleSpinLock (SPL : in out Spin_Lock);
154
155       ------------------------------
156       -- Simple Mutex Lock Routines --
157       ------------------------------
158       procedure SimpleSuspensionInit (SPL : in out Suspension_Lock);
159       procedure SimpleSuspensionLock (SPL : in out Suspension_Lock);
160       procedure SimpleSuspensionUnlock (SPL : in out Suspension_Lock);
161       ------------------------------
162       ------- PO Entry Routines -----
163       ------------------------------
164
165       --  ASSUMPTION: NO NESTED RESOURCE
166
167       -- Mutex imports
168        ---- C data type
169       subtype int             is Interfaces.C.int;
170       subtype long            is Interfaces.C.long;
171       subtype unsigned_char   is Interfaces.C.unsigned_char;
172       subtype unsigned_long   is Interfaces.C.unsigned_long;
173       type unsigned_long_long_t is mod 2 ** 64;
174       type pthread_mutexattr_t is record
175          mutexkind : int;
176       end record;
177       pragma Convention (C, pthread_mutexattr_t);
178
179       type pthread_mutex_t is new System.Linux.pthread_mutex_t;
180       type pthread_cond_t is array (0 .. 47) of unsigned_char;
181       pragma Convention (C, pthread_cond_t);
182       for pthread_cond_t'Alignment use unsigned_long_long_t'Alignment;
183
184       type pthread_condattr_t is record
185          dummy : int;
186       end record;
187       pragma Convention (C, pthread_condattr_t);
188
189       type time_t is new long;
```

```
190     type timespec is record
191        tv_sec  : time_t;
192        tv_nsec : long;
193     end record;
194     pragma Convention (C, timespec);
195     Mutex_Attr    : aliased pthread_mutexattr_t;
196     Cond_Attr     : aliased pthread_condattr_t;
197
198     -- Linux error return
199     EINTR      : constant := System.Linux.EINTR;
200     ENOMEM     : constant := System.Linux.ENOMEM;
201
202     -- Suspension Object
203     type S_Object is record
204        State : Boolean;
205        pragma Atomic (State);
206        Waiting : Boolean;
207        L : aliased pthread_mutex_t;
208        CV : aliased pthread_cond_t;
209     end record;
210
211
212     procedure Initialize (S : in out S_Object);
213     procedure Set_False (S : in out S_Object);
214     procedure Set_True (S : in out S_Object);
215     procedure Suspend_Until_True (S : in out S_Object);
216
217
218  private
219     Length_Limit : constant Integer := 50;
220
221     type Task_Id is new System.Tasking.Task_Id;
222
223     type pthread_spinlock_status is new Interfaces.C.int;
224     pragma Convention (C, pthread_spinlock_status);
225
226     type Node;
227     type Node_Ptr is access Node;
228
229     type Node is record
230        Id : System.Tasking.Task_Id;
231        Ppri : System.Any_Priority;
232     end record;
233
234     type NodeQueue is array (1 .. Length_Limit) of Node;
235
236     -- Based on imported spin lock primitives @ s-osinte-linux.ads --
237     -- NOTE : Underlying spin lock is STRICTLY FIFO based --
238     type Spinning_Priority is (Active_Priority_of_Task,
           Non_Preemptively);
239     type Spin_Lock_Array_t is array (0 .. Length_Limit) of
240              aliased pthread_spinlock_t;
241     type Spin_Lock_Flag_t is array (0 .. Length_Limit) of Integer;
242
```

```
243     type Spin_Lock is
244        record
245           Lock_Entry : Spin_Lock_Array_t;
246           Lock_Flag  : Spin_Lock_Flag_t;
247           QueueLock  : aliased pthread_spinlock_t;
248           Length : Integer := 0;
249           Length_Limit : Integer := 10;
250           Num_of_Wait : Integer := 0;
251           Order  : Queue_Order;
252           At_Pri : Spinning_Priority := Non_Preemptively;
253           SPQ    : NodeQueue;
254        end record;
255     -- FIX ME: QLock is for internal use only. Realy shouldn't be here
          --
256     PTHREAD_PROCESS_SHARED : pthread_spinlock_status := 1;
257     PTHREAD_PROCESS_PRIVATE : pthread_spinlock_status := 0;
258     test_spin : Spin_Lock;
259     -- Assume Mutex Lock is required for suspension lock --
260
261     type Sus_Cond_Array_t is array (0 .. Length_Limit) of
262              aliased System.OS_Interface.pthread_cond_t;
263     type Sus_Array_t is array (0 .. Length_Limit) of
264              aliased System.OS_Interface.pthread_mutex_t;
265     type Queue_Rec is array (0 .. Length_Limit) of Integer;
266     type Suspension_Lock is
267        record
268           Lock_Flag : Boolean;
269           Routine   : Boolean;
270           Lock   : aliased System.OS_Interface.pthread_mutex_t;
271           QLock  : aliased System.OS_Interface.pthread_mutex_t;
272           PLock  : aliased pthread_spinlock_t;
273           ACond  : aliased System.OS_Interface.pthread_cond_t;
274           Queue  : Queue_Rec;
275           LockSeq: Sus_Array_t;
276           Cond   : Sus_Cond_Array_t;
277           Length : Integer := 0;
278           Order  : Queue_Order;
279           Ceiling : System.Any_Priority;
280           SSQ    : NodeQueue;
281        end record;
282
283     ---------------------------
284     ---- MSRP Routines ----
285     ---------------------------
286
287     type Tas;
288     type Tas_Ptr is access Tas;
289
290     type Tas is record
291         T_id: System.Tasking.Task_Id;
292         PML : Integer;
293     end record;
294
295     -- Suspension Lock Foundamentals
```

```
296     Task_Queue : array (0..Num_of_Processors, 0..Num_of_Task) of Tas;
297     Task_Queue_Length : array (0..Num_of_Processors) of Integer;
298        -- Processor System Ceiling
299     Ceiling_Array : array (0..Num_of_Processors, 0..Num_of_Task) of
            Integer;
300     Ceiling_Array_Length : array (0..Num_of_Processors) of Integer;
301
302     ------------------------------
303     ---- RWLocks Routines ----
304     ------------------------------
305     type RWLockT is record
306         rin, rout : Interfaces.Unsigned_32;  -- rin(0):=PHID; rin(1):=
                PRES
307         rinQ, routQ: Spin_Lock;
308         win, wout : Interfaces.Unsigned_32;
309         winQ, woutQ: Spin_Lock;
310         WriterLock : Spin_Lock;
311     end record;
312
313
314 end System.Multiprocessors.QueueLock;
```

Listing B.1: Queue Lock Package Specification

```
 1 with Ada.Dynamic_Priorities;
 2 with System.Tasking; use System.Tasking;
 3 with System.Task_Primitives;
 4 with System.Task_Primitives.Operations;
 5 with Ada.Task_Identification;
 6 with Ada.Unchecked_Conversion;
 7 with Ada.Unchecked_Deallocation;
 8 with Interfaces.C;
 9 with System.OS_Interface;
10 with Text_IO; use Text_IO;
11 with System.OS_Primitives;
12 with System.Soft_Links;
13 with System.Machine_Code;  use System.Machine_Code;
14 with Interfaces; use Interfaces;
15
16 package body System.Multiprocessors.QueueLock is
17
18     package STPO renames System.Task_Primitives.Operations;
19     package ADPR renames Ada.Dynamic_Priorities;
20     package ATID renames Ada.Task_Identification;
21     package OSI renames System.OS_Interface;
22      package SSL renames System.Soft_Links;
23
24     use System.Task_Primitives.Operations;
25     use Interfaces.C;
26     use Interfaces;
27
28     ------------------------------------
29     -------- Priority Patch Routines ------
30     ------------------------------------
31     procedure Set_Priority (pri : Integer) is
```

```
32          Attributes : aliased OSI.pthread_attr_t;
33          Result : Interfaces.C.int;
34          Param : aliased OSI.struct_sched_param;
35       begin
36          Param.sched_priority := int(40);
37          Result :=
38               OSI.pthread_setschedparam
39               (STPO.Get_Thread_Id(STPO.Self), OSI.SCHED_FIFO, Param'
                     Access);
40          pragma Assert (Result = 0);
41       end Set_Priority;
42
43       function GetAffinity return integer is
44       begin
45         return Integer(STPO.Self.Common.Base_CPU);
46       end GetAffinity;
47
48       function GetSpinIndex return integer is
49       begin
50         return STPO.Self.Common.SpinQ_index;
51       end;
52
53       procedure SetSpinIndex (i : integer) is
54       begin
55         STPO.Self.Common.SpinQ_Index := i;
56       end;
57
58       function GetSusIndex return integer is
59       begin
60         return STPO.Self.Common.SuspensionQ_Index;
61       end;
62
63       procedure SetSusIndex (i : integer) is
64       begin
65         STPO.Self.Common.SuspensionQ_Index := i;
66       end;
67
68       ------------------------
69       -- Local Subprograms --
70       ------------------------
71       procedure OrderQueue (SL : in out Suspension_Lock);
72       --  Conversion functions between different forms of Task_Id
73       procedure Free is new Ada.Unchecked_Deallocation (
74             Object => Node, Name => Node_Ptr);
75       function FindPos (SPL : in out Spin_Lock) return Integer;
76       function FindPos (SL : in out Suspension_Lock) return Integer;
77       function FindNext (SL : in out Suspension_Lock;
78          start : Integer) return Integer;
79
80       --------------------
81       -- Current_Task --
82       --------------------
83       function Current_Task return System.Tasking.Task_Id is
84       begin
```

```
85          return System.Task_Primitives.Operations.Self;
86       end Current_Task;
87
88
89       function Is_Equal(Tid1 : System.Tasking.Task_Id; Tid2 : System.
             Tasking.Task_Id)
90          return boolean
91         is
92         begin
93             if Tid1 = Tid2 then
94                 return True;
95             else
96                 return False;
97             end if;
98       end Is_Equal;
99
100      ------------------------
101      -- Reorganize Queue --
102      ------------------------
103      -- Only Priority Queue needs this --
104      procedure OrderQueue (SL : in out Suspension_Lock) is
105         i : Integer;
106         temp : Node;
107      begin
108         for i in 1 .. (SL.Length - 1) loop
109             if (STPO.Get_Priority (SL.SSQ (i).Id) <
110                STPO.Get_Priority (SL.SSQ (i + 1).Id)) then
111                temp := SL.SSQ (i);
112                SL.SSQ (i) := SL.SSQ (i + 1);
113                SL.SSQ (i + 1) := temp;
114             end if;
115         end loop;
116      end OrderQueue;
117
118      function Get_Priority (Id : System.Tasking.Task_Id)
119         return System.Any_Priority
120      is
121      begin
122         return STPO.Get_Priority (Id);
123      end Get_Priority;
124
125      procedure Set_Priority (
126         Id : System.Tasking.Task_Id;
127         Prio : System.Any_Priority)
128      is
129      begin
130         STPO.Set_Priority (Id, Prio);
131      end Set_Priority;
132
133      function Head_Task (SL : Suspension_Lock)
134             return System.Tasking.Task_Id
135      is
136      begin
137         return SL.SSQ (1).Id;
```

```
138     end Head_Task;
139
140     function Head_Task (SPL : Spin_Lock)
141        return System.Tasking.Task_Id
142     is
143     begin
144        return SPL.SPQ (1).Id;
145     end Head_Task;
146     ------------------------------------------
147     -- Pthread Spinlock Platform dependent --
148     ------------------------------------------
149
150     function pthread_spin_init
151        (spin_t : not null access pthread_spinlock_t;
152         spin_share : pthread_spinlock_status) return int;
153     pragma Import (C, pthread_spin_init, "pthread_spin_init");
154
155     function pthread_spin_destroy
156        (spin_t : not null access pthread_spinlock_t) return int;
157     pragma Import (C, pthread_spin_destroy, "pthread_spin_destroy");
158
159     function pthread_spin_lock
160        (spin_t : not null access pthread_spinlock_t) return int;
161     pragma Import (C, pthread_spin_lock, "pthread_spin_lock");
162
163     function pthread_spin_unlock
164        (spin_t : not null access pthread_spinlock_t) return int;
165     pragma Import (C, pthread_spin_unlock, "pthread_spin_unlock");
166
167     function pthread_spin_trylock
168        (spin_t : not null access pthread_spinlock_t) return int;
169     pragma Import (C, pthread_spin_trylock, "pthread_spin_trylock");
170
171     function pthread_cond_broadcast
172        (cond : access OSI.pthread_cond_t) return int;
173     pragma Import (C, pthread_cond_broadcast, "pthread_cond_broadcast"
           );
174
175     ------------------------
176     -- Spin Lock Routines --
177     ------------------------
178     procedure Initialize (SPL : in out Spin_Lock) is
179        Result : Interfaces.C.int;
180        i  :  Integer;
181     begin
182
183        for i in SPL.SPQ'Range loop
184           SPL.SPQ (i).Id := null;
185           SPL.SPQ (i).Ppri := System.Any_Priority'First;
186        end loop;
187
188        for i in 0 .. SPL.Length_Limit loop
189           SPL.Lock_Flag (i) := 0;
190           Result := pthread_spin_init (
```

```
191                         SPL.Lock_Entry (i)'Access, PTHREAD_PROCESS_SHARED
                                );
192            pragma Assert (Result = 0);
193         end loop;
194
195         Result :=
196           pthread_spin_init (SPL.QueueLock'Access,
                 PTHREAD_PROCESS_SHARED);
197         pragma Assert (Result = 0);
198
199         SPL.Length := 0;
200      end Initialize;
201
202      ------------------
203      -- Acquire Lock --
204      ------------------
205      function FindPos (SPL : in out Spin_Lock)
206         return Integer
207      is
208         i : Integer;
209      begin
210         for i in 1 .. SPL.Length loop
211            if SPL.Lock_Flag (i) = -1 then
212               return i;
213            end if;
214         end loop;
215         SPL.Length := SPL.Length + 1;
216         return SPL.Length;
217      end FindPos;
218      pragma Inline (FindPos);
219
220      procedure SimpleSpinLock (SPL : in out Spin_Lock) is
221          Result : Interfaces.C.int;
222      begin
223          Result := pthread_spin_lock(SPL.QueueLock'Access);
224      end SimpleSpinLock;
225
226      procedure SimpleSpinUnlock (SPL : in out Spin_Lock) is
227          Result : Interfaces.C.int;
228      begin
229          Result := pthread_spin_unlock(SPL.QueueLock'Access);
230      end SimpleSpinUnlock;
231
232
233      procedure Acquire (SPL : in out Spin_Lock) is
234         Result : Interfaces.C.int;
235         T_Priority : System.Any_Priority;
236         Posi   :  Integer;
237      begin
238         Result := pthread_spin_lock (SPL.QueueLock'Access);
239         pragma Assert (Result = 0);
240         -- Init --
241         if Current_Task.Common.Spinning = False then
242            Posi := FindPos (SPL);
```

```
243            pragma Assert (Posi >= 1);
244
245            SPL.SPQ (Posi).Id := Current_Task;
246            SPL.SPQ (Posi).Ppri := STPO.Get_Priority (Current_Task);
247            SPL.Num_of_Wait := SPL.Num_of_Wait + 1;
248            Current_Task.Common.SpinQ_Index := Posi;
249            SPL.Lock_Flag (Current_Task.Common.SpinQ_Index) :=
250               SPL.Num_of_Wait;
251            -- If the queue is FIFO ordered --
252            Current_Task.Common.Spinning := True;
253         end if;
254         -- Init Done --
255         Result := pthread_spin_unlock (SPL.QueueLock'Access);
256         pragma Assert (Result = 0);
257
258         loop
259            Result :=
260                pthread_spin_lock (SPL.QueueLock'Access);
261            pragma Assert (Result = 0);
262
263            Result :=
264                pthread_spin_lock (
265                    SPL.Lock_Entry (Current_Task.Common.SpinQ_Index)'
                           Access);
266            pragma Assert (Result = 0);
267
268            Result := pthread_spin_unlock (SPL.QueueLock'Access);
269            pragma Assert (Result = 0);
270
271            if SPL.Lock_Flag (Current_Task.Common.SpinQ_Index) = 1 then
272               Result :=
273                    pthread_spin_unlock (
274                        SPL.Lock_Entry (
275                            Current_Task.Common.SpinQ_Index)'Access);
276               pragma Assert (Result = 0);
277               exit;
278            else
279               Result :=
280                  pthread_spin_unlock (
281                       SPL.Lock_Entry (
282                            Current_Task.Common.SpinQ_Index)'Access);
283               pragma Assert (Result = 0);
284            end if;
285         end loop;
286      end Acquire;
287
288      procedure Release (SPL : in out Spin_Lock) is
289         Result : Interfaces.C.int;
290         i, j : Integer;
291         Index : Integer;
292      begin
293         Result := pthread_spin_lock (SPL.QueueLock'Access);
294         pragma Assert (Result = 0);
295
```

```
296         for i in 1 .. SPL.Length loop
297            Result := pthread_spin_lock (
298                    SPL.Lock_Entry (SPL.SPQ (i).Id.Common.SpinQ_Index)'
                          Access);
299            pragma Assert (Result = 0);
300            Index := SPL.SPQ (i).Id.Common.SpinQ_Index;
301            SPL.Lock_Flag (Index) :=
302                   SPL.Lock_Flag (Index) - 1;
303
304            if SPL.Lock_Flag (Index) = 0 then
305               SPL.SPQ (Index).Id := null;
306               SPL.SPQ (Index).Ppri := -1;
307               SPL.Lock_Flag (Index) := -1;
308               SPL.Num_of_Wait := SPL.Num_of_Wait - 1;
309               Current_Task.Common.SpinQ_Index := -1;
310               Current_Task.Common.Spinning := False;
311            end if;
312            Result := pthread_spin_unlock (
313                 SPL.Lock_Entry (Index)'Access);
314            pragma Assert (Result = 0);
315         end loop;
316
317         Result := pthread_spin_unlock (SPL.QueueLock'Access);
318         pragma Assert (Result = 0);
319      end Release;
320
321      function H_Queued_Priority (SPL : in out Spin_Lock)
322         return System.Any_Priority is
323         Result : Interfaces.C.int;
324         i  :  Integer;
325         temp : Node_Ptr;
326         t_Prio : System.Any_Priority := System.Any_Priority'First;
327      begin
328         Result := pthread_spin_lock (SPL.QueueLock'Access);
329         for i in SPL.SPQ'Range loop
330            if t_Prio < STPO.Get_Priority (SPL.SPQ (i).Id) then
331               t_Prio := STPO.Get_Priority (SPL.SPQ (i).Id);
332            end if;
333         end loop;
334         Result := pthread_spin_unlock (SPL.QueueLock'Access);
335         return t_Prio;
336      end H_Queued_Priority;
337
338      ------------------------------
339      -- Suspension Lock Programs --
340      ------------------------------
341      procedure Initialize (
342         SL : in out Suspension_Lock;
343         Ceil : System.Any_Priority;
344         QOrder : Queue_Order)
345      is
346         MutexAttr : aliased OSI.pthread_mutexattr_t;
347         CondAttr  : aliased OSI.pthread_condattr_t;
348         Result : Interfaces.C.int;
```

```
349         i      : Integer;
350      begin
351         SL.Length := 0;
352         SL.Ceiling := Ceil;
353         SL.Order := QOrder;
354         Result := OSI.pthread_mutexattr_init (MutexAttr'Access);
355         pragma Assert (Result = 0);
356
357         Result := OSI.pthread_mutex_init (
358               SL.Lock'Access, MutexAttr'Access);
359         pragma Assert (Result = 0);
360
361         Result := OSI.pthread_mutex_init (
362               SL.QLock'Access, MutexAttr'Access);
363         pragma Assert (Result = 0);
364
365         Result := pthread_spin_init (
366                        SL.PLock'Access, PTHREAD_PROCESS_SHARED);
367            pragma Assert (Result = 0);
368
369         Result := OSI.pthread_condattr_init (CondAttr'Access);
370         pragma Assert (Result = 0);
371
372         Result := OSI.pthread_cond_init (
373               SL.ACond'Access, CondAttr'Access);
374         pragma Assert (Result = 0);
375
376         for i in SL.SSQ'Range loop
377            SL.SSQ (i).Id := null;
378            SL.SSQ (i).Ppri := System.Any_Priority'First;
379
380            Result := OSI.pthread_cond_init (
381               SL.Cond (i)'Access, CondAttr'Access);
382            pragma Assert (Result = 0);
383
384            SL.Queue (i) := -1;
385         end loop;
386
387         for i in 1 .. Length_Limit loop
388               Result := OSI.pthread_mutex_lock (
389                  SL.LockSeq (i)'Access);
390               pragma Assert (Result = 0);
391         end loop;
392         SL.Lock_Flag := False;
393         SL.Routine  := False;
394      end Initialize;
395
396      function FindPos (SL : in out Suspension_Lock)
397         return Integer
398      is
399         Result : Interfaces.C.int;
400         i : Integer;
401      begin
402         for i in 1 .. SL.Length loop
```

```
403             if SL.Queue (i) = -1 then
404                return i;
405             end if;
406          end loop;
407          return -1;
408       end FindPos;
409       pragma Inline (FindPos);
410
411       function FindNext (
412          SL : in out Suspension_Lock;
413          start : Integer) return Integer
414       is
415          Result : Interfaces.C.int;
416          i : Integer;
417       begin
418          for i in start .. SL.Length loop
419             if SL.Queue (i) /= -1 then
420                return i;
421             end if;
422          end loop;
423          return -1;
424       end FindNext;
425       pragma Inline (FindNext);
426
427       procedure Insert (SL : in out Suspension_Lock) is
428          Index : Integer;
429          Result : Interfaces.C.int;
430       begin
431          SL.Length := SL.Length + 1;
432          Index := FindPos (SL);
433          SL.Queue (Index) := 1;
434          pragma Assert (Index > -1);
435          Current_Task.Common.SuspensionQ_Index := Index;
436          SL.SSQ (SL.Length).Id := Current_Task;
437          SL.SSQ (SL.Length).Ppri := STPO.Get_Priority (Current_Task);
438
439       end Insert;
440
441       procedure Remove_PRIO (SL : in out Suspension_Lock) is
442          Index, i, Next : Integer;
443          T_P : System.Any_Priority := System.Any_Priority'First;
444          Result : Interfaces.C.int;
445       begin
446
447          Next := 2;
448          pragma Assert (Next >= 1);
449          for i in 2 .. SL.Length loop
450             if T_P < STPO.Get_Priority (SL.SSQ (i).Id) then
451                Next := i;
452                T_P := STPO.Get_Priority (SL.SSQ (i).Id);
453             end if;
454          end loop;
455
456          SL.Queue (Current_Task.Common.SuspensionQ_Index)
```

```
457            := -1;
458         SL.SSQ (1).Id := SL.SSQ (Next).Id;
459         SL.SSQ (1).Ppri := SL.SSQ (Next).Ppri;
460
461         for i in Next .. SL.Length loop
462            SL.SSQ (i).Id := SL.SSQ (i + 1).Id;
463            SL.SSQ (i).Ppri := SL.SSQ (i + 1).Ppri;
464         end loop;
465
466         SL.Length := SL.Length - 1;
467      end Remove_PRIO;
468
469      procedure Remove_FIFO (SL : in out Suspension_Lock) is
470         Index : Integer;
471         Result : Interfaces.C.int;
472      begin
473         for i in 1 .. (SL.Length) loop
474            SL.SSQ (i).Id := SL.SSQ (i + 1).Id;
475            SL.SSQ (1).Ppri := SL.SSQ (i + 1).Ppri;
476         end loop;
477         SL.Queue (Current_Task.Common.SuspensionQ_Index)
478            := -1;
479         SL.Length := SL.Length - 1;
480      end Remove_FIFO;
481
482      procedure SimpleSuspensionInit (SPL : in out Suspension_Lock) is
483           MutexAttr : aliased OSI.pthread_mutexattr_t;
484         CondAttr  : aliased OSI.pthread_condattr_t;
485         Result : Interfaces.C.int;
486         i      : Integer;
487      begin
488         Result := OSI.pthread_mutexattr_init (MutexAttr'Access);
489         pragma Assert (Result = 0);
490
491         Result := OSI.pthread_mutex_init (
492               SPL.Lock'Access, MutexAttr'Access);
493         pragma Assert (Result = 0);
494       end SimpleSuspensionInit;
495
496      procedure SimpleSuspensionLock (SPL : in out Suspension_Lock) is
497        Result : Interfaces.C.Int;
498      begin
499         Result := OSI.pthread_mutex_lock (SPL.Lock'Access);
500         pragma Assert (Result = 0);
501      end SimpleSuspensionLock;
502
503      procedure SimpleSuspensionUnlock (SPL : in out Suspension_Lock) is
504        Result : Interfaces.C.Int;
505      begin
506           Result := OSI.pthread_mutex_unlock (SPL.Lock'Access);
507         pragma Assert (Result = 0);
508       end SimpleSuspensionUnlock;
509
510      procedure Acquire (SL : in out Suspension_Lock) is
```

```
511         T_Priority : System.Any_Priority;
512         Index, i :  Integer;
513         Result, test : Interfaces.C.int;
514      begin
515         Result := OSI.pthread_mutex_lock (SL.QLock'Access);
516         pragma Assert (Result = 0);
517         Insert (SL);
518         if SL.Lock_Flag = True then
519            Result := OSI.pthread_mutex_unlock (SL.QLock'Access);
520            pragma Assert (Result = 0);
521            Result := OSI.pthread_mutex_lock (
522               SL.LockSeq (Current_Task.Common.SuspensionQ_Index)'Access
                     );
523            pragma Assert (Result = 0);
524         else
525            SL.Lock_Flag := True;
526            Result := OSI.pthread_mutex_unlock (SL.QLock'Access);
527            pragma Assert (Result = 0);
528         end if;
529      end Acquire;
530
531      procedure Release (SL : in out Suspension_Lock) is
532         temp : Integer;
533         i, Next, Index  :  Integer;
534         T_P : System.Any_Priority := System.Any_Priority'First;
535         Result : Interfaces.C.int;
536      begin
537         Result := OSI.pthread_mutex_lock (SL.QLock'Access);
538         pragma Assert (Result = 0);
539         if SL.Order = Priority_Ordered then
540            Remove_PRIO (SL);
541         else
542            Remove_FIFO (SL);
543         end if;
544
545         if SL.Length > 0 then
546            Result := OSI.pthread_mutex_unlock (SL.QLock'Access);
547            pragma Assert (Result = 0);
548            Index := SL.SSQ (1).Id.Common.SuspensionQ_Index;
549       --     Put_Line ("**** Releasing " & Integer'Image (Index));
550            Result := OSI.pthread_mutex_unlock (SL.LockSeq (Index)'
                  Access);
551            pragma Assert (Result = 0);
552         else
553             SL.Lock_Flag := False;
554             Result := OSI.pthread_mutex_unlock (SL.QLock'Access);
555             pragma Assert (Result = 0);
556         end if;
557      end Release;
558
559      function H_Queued_Priority (SL : in out Suspension_Lock)
560         return System.Any_Priority is
561         i : Integer;
562         Result : Interfaces.C.int;
```

```
563         T_P, T_Priority : System.Any_Priority := System.Any_Priority'
                First;
564      begin
565         Result := OSI.pthread_mutex_lock (SL.QLock'Access);
566         pragma Assert (Result = 0);
567         for i in 1 .. SL.Length loop
568            if T_Priority < STPO.Get_Priority (SL.SSQ (i).Id) then
569               T_Priority := STPO.Get_Priority (SL.SSQ (i).Id);
570            end if;
571         end loop;
572         Result := OSI.pthread_mutex_unlock (SL.QLock'Access);
573         pragma Assert (Result = 0);
574         return T_Priority;
575      end H_Queued_Priority;
576
577      ----------------------------
578      ---- MSRP Routines ----
579      ----------------------------
580      function pthread_mutex_init
581        (mutex : access pthread_mutex_t;
582         attr  : access pthread_mutexattr_t) return int;
583      pragma Import (C, pthread_mutex_init, "pthread_mutex_init");
584
585      function pthread_mutex_destroy (mutex : access pthread_mutex_t)
             return int;
586      pragma Import (C, pthread_mutex_destroy, "pthread_mutex_destroy");
587
588      function pthread_mutex_lock (mutex : access pthread_mutex_t)
             return int;
589      pragma Import (C, pthread_mutex_lock, "pthread_mutex_lock");
590
591      function pthread_mutex_unlock (mutex : access pthread_mutex_t)
             return int;
592      pragma Import (C, pthread_mutex_unlock, "pthread_mutex_unlock");
593
594      function pthread_condattr_init
595        (attr : access pthread_condattr_t) return int;
596      pragma Import (C, pthread_condattr_init, "pthread_condattr_init");
597
598      function pthread_condattr_destroy
599        (attr : access pthread_condattr_t) return int;
600      pragma Import (C, pthread_condattr_destroy, "
             pthread_condattr_destroy");
601
602      function pthread_cond_init
603        (cond : access pthread_cond_t;
604         attr : access pthread_condattr_t) return int;
605      pragma Import (C, pthread_cond_init, "pthread_cond_init");
606
607      function pthread_cond_destroy (cond : access pthread_cond_t)
             return int;
608      pragma Import (C, pthread_cond_destroy, "pthread_cond_destroy");
609
```

```
610     function pthread_cond_signal (cond : access pthread_cond_t) return
            int;
611     pragma Import (C, pthread_cond_signal, "pthread_cond_signal");
612
613     function pthread_cond_wait
614       (cond  : access pthread_cond_t;
615        mutex : access pthread_mutex_t) return int;
616     pragma Import (C, pthread_cond_wait, "pthread_cond_wait");
617
618     function pthread_cond_timedwait
619       (cond    : access pthread_cond_t;
620        mutex   : access pthread_mutex_t;
621        abstime : access timespec) return int;
622     pragma Import (C, pthread_cond_timedwait, "pthread_cond_timedwait"
           );
623
624
625      procedure Initialize (S : in out S_Object) is
626        Result : Interfaces.C.int;
627     begin
628        S.State := False;
629        S.Waiting := False;
630        Result := pthread_mutex_init (S.L'Access, Mutex_Attr'Access);
631        pragma Assert (Result = 0 or else Result = ENOMEM);
632        if Result = ENOMEM then
633           raise Storage_Error;
634        end if;
635        Result := pthread_cond_init (S.CV'Access, Cond_Attr'Access);
636        pragma Assert (Result = 0 or else Result = ENOMEM);
637        if Result /= 0 then
638           Result := pthread_mutex_destroy (S.L'Access);
639           pragma Assert (Result = 0);
640           if Result = ENOMEM then
641              raise Storage_Error;
642           end if;
643        end if;
644     end Initialize;
645
646     procedure Set_False (S : in out S_Object) is
647        Result : Interfaces.C.int;
648     begin
649        SSL.Abort_Defer.all;
650        Result := pthread_mutex_lock (S.L'Access);
651        pragma Assert (Result = 0);
652        S.State := False;
653        Result := pthread_mutex_unlock (S.L'Access);
654        pragma Assert (Result = 0);
655        SSL.Abort_Undefer.all;
656     end Set_False;
657
658     procedure Set_True (S : in out S_Object) is
659        Result : Interfaces.C.int;
660
661     begin
```

```
662         SSL.Abort_Defer.all;
663         Result := pthread_mutex_lock (S.L'Access);
664         pragma Assert (Result = 0);
665         if S.Waiting then
666            S.Waiting := False;
667            S.State := False;
668            Result := pthread_cond_signal (S.CV'Access);
669            pragma Assert (Result = 0);
670         else
671            S.State := True;
672         end if;
673         Result := pthread_mutex_unlock (S.L'Access);
674         pragma Assert (Result = 0);
675         SSL.Abort_Undefer.all;
676      end Set_True;
677
678       procedure Suspend_Until_True (S : in out S_Object) is
679         Result : Interfaces.C.int;
680      begin
681         SSL.Abort_Defer.all;
682         Result := pthread_mutex_lock (S.L'Access);
683         pragma Assert (Result = 0);
684
685         if S.Waiting then
686            Result := pthread_mutex_unlock (S.L'Access);
687            pragma Assert (Result = 0);
688            SSL.Abort_Undefer.all;
689            raise Program_Error;
690         else
691            if S.State then
692               S.State := False;
693            else
694               S.Waiting := True;
695               loop
696                  Result := pthread_cond_wait (S.CV'Access, S.L'Access);
697                  pragma Assert (Result = 0 or else Result = EINTR);
698
699                  exit when not S.Waiting;
700               end loop;
701            end if;
702
703            Result := pthread_mutex_unlock (S.L'Access);
704            pragma Assert (Result = 0);
705
706            SSL.Abort_Undefer.all;
707         end if;
708      end Suspend_Until_True;
709
710      procedure Set_PML (P : Integer) is
711      begin
712          STPO.Self.Common.PML := P;
713      end Set_PML;
714
715   function Get_PML return Integer is
```

```
716     begin
717         return STPO.Self.Common.PML;
718     end Get_PML;
719
720     procedure setProc (CPU : Integer; C : Integer) is
721     begin
722         Ceiling_Array_Length(CPU) := Ceiling_Array_Length(CPU) + 1;
723         Ceiling_Array(CPU, 1) := C;
724     end setProc;
725
726     procedure MSRP_Init is
727          Result : Interfaces.C.int;
728     begin
729        -- initialize protection lock
730        Result :=
731          pthread_spin_init (MSRPQ'Access, PTHREAD_PROCESS_SHARED);
732        pragma Assert (Result = 0);
733   end MSRP_Init;
734
735     procedure Lock (
736         res : in out MSRP)
737     is
738         temp, c_len, s_len, C, i  : Integer;
739         Result : Interfaces.C.int;
740     begin
741         null;
742
743     end Lock;
744
745
746     procedure Unlock (
747         res : in out MSRP)
748     is
749         temp, c_len, s_len  : Integer;
750         C  : Integer;
751         comp, comp_index : Integer;
752         Result : Interfaces.C.int;
753         released : Boolean := False;
754     begin
755         null;
756     end Unlock;
757
758     procedure Stop is
759         C, comp, comp_index : Integer;
760         c_len, s_len  : Integer;
761         released : Boolean := False;
762         Result : Interfaces.C.int;
763     begin
764         null;
765     end Stop;
766
767     function Sync_Val_Compare_And_Swap
768       (Destination : access Unsigned_32;  -- the target number
769        Comparand   : Unsigned_32;     -- was it still the old value
```

```
770         New_Value    : Unsigned_32)   -- the new value
771         return Unsigned_32
772      is
773         Prior_Value : Unsigned_32;
774         pragma Suppress (All_Checks);
775      begin
776         Asm("lock cmpxchg %1, %2;",
777           Inputs  => (Unsigned_32'Asm_Input ("r", New_Value),         --
                 %1
778                       Unsigned_32'Asm_Input ("m", Destination.all), --
                          %2
779                       Unsigned_32'Asm_Input ("a", Comparand)),
780           Outputs => (Unsigned_32'Asm_Output ("=a", Prior_Value)),   --
                 %0
781
782           Clobber => "memory, cc",
783           Volatile => True);
784         --  return %eax
785
786         return Prior_Value;
787      end Sync_Val_Compare_And_Swap;
788   end System.Multiprocessors.QueueLock;
```

Listing B.2: Queue Lock Package Implementation

Bibliography

[1] IEEE standard for information technology - portable operating system interface (posix). shell and utilities. *IEEE Std 1003.1, 2004 Edition The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001 Cor 1-2002 and IEEE Std 1003.1-2001 Cor 2-2004. Shell*, pages 0_1–, 2004.

[2] *Linux Manual Page, http://www.linuxmanpages.com/*, Aug, 2013.

[3] B. B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *RTCSA '00: Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, page 337, Washington, DC, USA, 2000. IEEE Computer Society.

[4] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1):67–99, April 1991.

[5] T. P. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. *In International Conf. on Real-Time and Network Sys.*, pages 119–127, 2005.

[6] W. A. Barrett, R. M. Bates, D. A. Gustafson, and J. D. Couch. *Compiler construction: theory and practice (2nd ed.)*. SRA School Group, USA, 1986.

[7] N. L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The m5 simulator: Modeling networked systems. *Micro, IEEE*, 26(4):52 –60, july-aug. 2006.

[8] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency: a software approach. In *Proceedings of the third inter-*

national conference on Architectural support for programming languages and operating systems, ASPLOS III, pages 113–122, New York, NY, USA, 1989. ACM.

[9] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56, Washington, DC, USA, 2007. IEEE Computer Society.

[10] G. Bollella and J. Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[11] B. B. Brandenburg and J. H. Anderson. Optimality results for multiprocessor real-time locking. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 49 –60, 2010.

[12] B. B. Brandenburg and J. H. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, pages 1–66, 2012.

[13] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353, Washington, DC, USA, 2008. IEEE Computer Society.

[14] A. Burns and A. J. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, New York, NY, USA, 3rev ed edition, 2007.

[15] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages (Fourth Edition)*. Addison-Wesley Longmain, 2009.

[16] A. Burns and A. J. Wellings. Dispatching domains for multiprocessor platforms and their representation in ada. In *Proceedings of the 15th Ada-Europe international conference on Reliable Software Technologies*, Ada-Europe'10, pages 41–53, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] J. Casazza. Intel core i7-800 processor series and the intel core i5-700 processor series based on intel microarchitecture (nehalem). *Intel White Paper, Intel Corporation, USA*, 2009.

[18] S. Chandra, A. and Prashant. Hierarchical scheduling for symmetric multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 19(3):418–431, 2008.

[19] Y. Chang, R. Davis, and A.J. Wellings. Reducing Queue Lock Pessimism in Multiprocessor Schedulability Analysis. In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, pages 99–108, Toulouse, France, November 2010.

[20] R. Davis and A. Burns. A survey of hard real-time scheduling algorithms and schedubility analysis techniques for multiprocessor systems. 2009.

[21] U. C. Devi, H. Leontyev, and J. H. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 75–84, Washington, DC, USA, 2006. IEEE Computer Society.

[22] B. Dobbing and A. Burns. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 annual ACM SIGAda international conference on Ada*, SIGAda '98, pages 1–6, New York, NY, USA, 1998. ACM.

[23] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. *Real-Time Systems Symposium, IEEE International*, 0:377–386, 2009.

[24] S. Edmond. Towards Ada 2012: An interim report. In Jorge Real and Tullio Vardanega, editors, *Reliable Software Technologiey Ada-Europe 2010*, volume 6106 of *Lecture Notes in Computer Science*, pages 238–250. Springer Berlin Heidelberg, 2010.

[25] M. B. Franklin. Who's Using Ada? http://www.seas.gwu.edu/m̃feldman/ada-project-summary.html. Technical report, Feburary 2012.

[26] P. Gai, G. Lipari, and M. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *In Proceedings of*

the 22nd IEEE Real-Time Systems Symposium, pages 73–83. Society Press, 2001.

[27] F. Garcia and J. Fernandez. POSIX thread libraries http://dl.acm.org/citation.cfm?id=348120.348381. *Linux J.*, 2000(70es), February 2000.

[28] A. Garg. Real-time linux kernel scheduler.

[29] P. B. Hansen. Monitors and concurrent pascal: a personal history. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 1–35, New York, NY, USA, 1993. ACM.

[30] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, 2007.

[31] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[32] IBM. *IBM AIX Systems Manual, http://www-03.ibm.com/systems/power/software/aix/about.html*, 2012.

[33] The IEEE and The Open Group. *The Open Group Base Specifications Issue 7 – IEEE Std 1003.1, 2007 Edition*. IEEE, New York, NY, USA, 2007.

[34] J. Isaak. Standards-the history of posix: a study in the standards process. *Computer*, 23(7):89–92, 1990.

[35] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[36] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Trans. Software Engineering*, 30:629, 2004.

[37] J. Liebeherr, A. Burchard, Y. Oh, and Sang H. Son. New strategies for assigning real-time tasks to multiprocessor systems. volume 44, pages 1429–1442, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[38] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wsowski. Evolution of the linux kernel variability model. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 136–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] J. Miranda, E. Schonberg, and G. Dismukes. The implementation of ada 2005 interface types in the gnat compiler. In Tullio Vardanega and Andy Wellings, editors, *Reliable Software Technology ?Ada-Europe 2005*, volume 3555 of *Lecture Notes in Computer Science*, pages 208–219. Springer Berlin Heidelberg, 2005.

[40] I. Molnar. The native posix thread library for linux, http://www.cs.utexas.edu/witchel/372/lectures/POSIX_Linux_Threading.pdf. Technical report, Tech. Rep., RedHat, Inc, 2003.

[41] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82 –85, Jan. 1998.

[42] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.

[43] Y. Oh and S. H. Sang. Fixed-Priority Scheduling of Periodic Tasks on Multiprocessor Systems, http://www.ncstrl.org:8900/ncstrl/servlet/searchformnamez=detail. Charlottesville, VA, USA, 1995. University of Virginia.

[44] U. Omar, P. Zapata, and P. M. Alvarez. Edf and rm multiprocessor scheduling algorithms: Survey and performance evaluation. 08 2010.

[45] R. Petersen. *Linux: The Complete Reference*. McGraw-Hill Professional, 4th edition, 2000.

[46] G. F. Pfister and V. A. Norton. "hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, 34(10):943–948, 1985.

[47] M. J. Quinn. *Parallel computing (2nd ed.): theory and practice*. McGraw-Hill, Inc., New York, NY, USA, 1994.

[48] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116 –123, June 1990.

[49] R. S. Rajkumar, L. Lehoczky, and J.P.Lehoczky. Real-time synchronization protocols for multiprocessors. *Real-Time Systems Symposium Proceedings.*, pages 259–269, 12 1988.

[50] J. Ras and M. K. Cheng, A. An evaluation of the dynamic and static multiprocessor priority ceiling protocol and the multiprocessor stack resource policy in an smp system. In *RTAS '09: Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, pages 13–22, Washington, DC, USA, 2009. IEEE Computer Society.

[51] H. Rhan and J. W. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 162–171.

[52] I. Ripoll, A. Crespo, and P. Balbastre. A new application-defined scheduling implementation in rtlinux. In *Sixth Real-Time Linux Workshop*, pages 175–181, 2004.

[53] M. A. Rivas and M. G. Harbour. A POSIX-Ada interface for application-defined scheduling. In *Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '02, pages 136–150, London, UK, UK, 2002. Springer-Verlag.

[54] M. A. Rivas and M. G. Harbour. Application-defined scheduling in ada. In *Proceedings of the International Real-Time Ada Workshop (IRTAW-2003*, pages 77–84, 2003.

[55] S. Robbins. Starving philosophers: experimentation with monitor synchronization. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, SIGCSE '01, pages 317–321, New York, NY, USA, 2001. ACM.

[56] P. Rogers and A. J. Wellings. Openada: Compile-time reflection for ada 95. In Albert Llamos and Alfred Strohmeier, editors, *Reliable Software Technologies - Ada-Europe 2004*, volume 3063 of *Lecture Notes in Computer Science*, pages 166–177. Springer Berlin Heidelberg, 2004.

[57] B. Senouci, A. Bouchhima, F. Rousseau, F. Petrot, and A. Jerraya. Fast prototyping of posix based applications on a multiprocessor soc architecture: "hardware-dependent software oriented approach". In *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on*, pages 69–75, 2006.

[58] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[59] QNX Software. *QNX neutrino RTOS 6.3.2, http://www.qnx.org.uk/products/neutrino-rtos/neutrino-rtos.html*, 2007.

[60] H. Sundell and P. Tsigas. Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 433 –440, 2000.

[61] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 134–143, Dec.

[62] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 47–58, New York, NY, USA, 2007. ACM.

[63] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 176–186, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[64] L. A. Torrey, J. Coleman, and B. P. Miller. A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler. *Softw. Pract. Exper.*, 37(4):347–364, April 2007.

[65] A. Tripathi. Challenges designing next-generation middleware systems. *Commun. ACM*, 45(6):39–42, June 2002.

[66] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: a structure for scalable multiprocessor operating system design. *J. Supercomput.*, 9(1-2):105–134, 2005.

[67] T. V. Verghese. *Resource management issues for shared-memory multiprocessors.* PhD thesis, Stanford, CA, USA, 1998. AAI9837261.

[68] Giering E. W. and T. P. Baker. The gnu ada runtime library (gnarl). In *Proceedings of the eleventh annual Washington Ada symposium & summer ACM SIGAda meeting on Ada*, WADAS '94, pages 97–107, New York, NY, USA, 1994. ACM.

[69] C. Wang, H. Takada, and K. Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. *2nd International Symposium on Parallel Architectures, Algorithms and Networkds*, pages 70–76, 1996.

[70] Y. C. Wang and K. J. Lin. Implementing a general real-time scheduling framework in the red-linux real-time kernel. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 246–255.

[71] A. J. Wellings. *Concurrent and Real-Time Programming in Java.* John Wiley & Sons, Inc., 2004.

[72] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):266–277, June 2007.

[73] V. Yodaiken. The RTLinux Manifesto, http://citeseer.ist.psu.edu/yodaiken99rtlinux.html. In *Proc. of The 5th Linux Expo, Raleigh, NC*, March 1999.

[74] J. Zamorano, J. Ruiz, and J.A. Puente. Implementing ada.real time.clock and absolute delays in real-time kernels. In Dirk Craeynest and Alfred Strohmeier, editors, *Reliable SoftwareTechnologies Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*, pages 317–327. Springer Berlin Heidelberg, 2001.

[75] A. Zerzelidis, A. Burns, and A. J. Wellings. Correcting the edf protocol in ada 2005. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 18–22, 2007.