technische universität dortmund

fakultät für informatik

TU Dortmund University
Department of Informatik, Informatik XII
Real-Time Group
Prof. Dr. Jian-Jia Chen

Master Thesis

Implementation and Evaluation of Real-Time Multiprocessor Scheduling Algorithms on $LITMUS^{RT}$

by

Jun Jie Shi

Supervised by:

Senior researcher: M. Ing. Wen Hung Kevin Huang

Examiner: Prof. Dr. Jian Jia Chen

Dortmund, DECEMBER 2016

Declaration of Authorship

I, Jun Jie Shi, declare that this thesis titled, 'Implementation and Evaluation of Real-Time Multiprocessor Scheduling Algorithms on $LITMUS^{RT}$ ' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift

Acknowledgements

I am gratitude to many people who has given me so much assistance during my thesis. First of all, I would like to thank my professor Prof. Chen who has kindly given me the chance to work on this wonderful topic. Meanwhile, I appreciate my supervisor Kevin who has guided me during the whole process of my thesis. I would also like to express my appreciate to Shuai Zhao (The University of York) for his guidance and advice during my implementation work. As well as Björn Brandenburg who is the maintainer of the $LITMUS^{RT}$ also gave me a lot of assistance throughout all the six months of my thesis. Last but not least, I would like to thank all those friends, Kuan-Hsun Chen and Wei Liu who has supported me during the writing of thesis.

Abstract

When concurrent real-time tasks have to access sharing resources, to prevent race conditions, no two concurrent accesses to one sharing resource are in their critical sections at the same time. Therefore, a mechanism for synchronization and resource access must be furnished.

To date, many protocols for task synchronizations are available, for instance, Multiprocessor Priority Ceiling Protocol (MPCP) with suspension-based and Multiprocessor resource sharing Protocol (MrsP) with spinning-based. Moreover, resource-oriented partitioned scheduling has been recently shown to be a particularly elegant approach for semi-partitioned systems. However, those protocols are developed upon the assumption that run time overhead is negligible. But, the fact is that the induced costs of overhead may come into play and vary from one protocol to another. Therefore, it is still arguable that whether there exists a preferable approach for resource sharing in multiprocessor systems when taking into account run time overhead.

In this thesis, we will evaluate what impact runtime overheads would have on different protocols based on the platform $LITMUS^{RT}$. Whether the overheads will destroy the essences of protocols in theory as well. We found that resource-oriented partitioned fixed priority scheduling is a pragmatically good algorithm in the sense that it has high performance in terms of schedulability and reasonably low run time overhead. Thus, the implementation and evaluation for all the resource synchronization protocols are based on P-FP scheduling algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the end of the last century, the original uni-processor system cannot meet the raising demand for capacity of computer from the users any longer. So the multiprocessor was developed by several companies. However, such multiprocessor chips have not been used in real-time system widely. When we design a real-time system, we need consider about lots of factors such as energy consuming and heat dissipation which may influence the whole performance, as well as the cost. In the first several years after the multiprocessors are available, the researchers of the real-time system haven't paid much attention to the multiprocessor environment for its high price, high power consumption and less performance improvement. That means, with the raising of the cost and energy consuming, there is no distinct improvement for the multiprocessor real-time system's performance. Until nowadays, benefited by the development of the chip technology, multiprocessor chips become more and more popular with the faster computing speed, lower price, lower power consumption as well as smaller volume. We believe that the application for the multiprocessor on the real-time system can definitely improve the performance with limited cost rise. Thus, people pay more attentions to such multiprocessor based real-time system, to find one optimal scheduling algorithm as well as resource synchronization protocol. Good scheduling algorithms can take full advantage of the multiprocessor which can reduce the average response time and increase the handling capacity of the system, bad algorithms are just wasting the utilization of the processors. Also, good resource synchronization protocol can improve the system's performance in some aspects. Of course, the criteria to judge whether one algorithm as well as one protocol is good or not are depend on different real-time applications.

Some systems are trying to reduce the average response time, some are strict to the deadline, thus we may apply different scheduling algorithms and protocols when facing to different systems. All in all, the researches are trying to exploit the potentialities of the multiprocessor's computing ability as much as possible when using it in the real-time system.

In the first place, we need talk about the attributes that one hard real-time system has. One of the most important properties that a hard real-time system should have is predictability. That is, based on the kernel features and on the information associated with each task, the syetem should be able to predict the evolution of the tasks and guarantee in advance that all critical timing constraints will be met [1]. A preemptive real-time operating system (RTOS) forms the fundamental of most embedded systems. If the preemption is not allowed, the optimal schedules may leave the processor idle to finish tasks with early deadlines arriving later. Meanwhile, when preemption is not allowed and tasks can have arbitrary arrivals, the problem of minimizing the maximum lateness and the problem of finding a feasible schedule become NP-hard [1]. To ensure rapid response time, an embedded RTOS can use preemption, in which a higher-priority task can interrupt a lower-priority task that is running. When the higher-priority task finishes running, the lower-priority task resumes executing from where it was interrupted. The use of preemption guarantees the worst-case response time, which enable use of the application in safety-critical situation. However, the situation will be different when it comes to sharing resources which are mutual exclusion. When a task has accessed one such sharing resource, it cannot be preempted by a higher priority task which want access to the same resource. Thus, the critical sections are used to prevent the concurrent accesses to the sharing resources. However, the use of the critical section will introduce other problems *e.g.*, deadlock and priority inversion. That is why we need resource synchronization protocols to solve the problems caused by the resources sharing. All those protocols are developed without considering the runtime overheads. However, the fact is that the overheads cannot be neglected in the real world implementation.

The fact leads to the motivations of our topic. What impact runtime over heads would have on different protocols' performance? Different protocol will suffer from different overheads. Will such difference destroy the advantages in theory of some protocols? Further more, will the overheads play different roles under different utilization? Will

the overheads perform different among light load system, medium load system and heavy load system? In our thesis, we will discuss all these problems in the following chapters.

1.1 Thesis Organization

In the Chapter 1, we will introduce the tasks scheduling algorithms for multiprocessor and the resource synchronization protocols for uni-processor. In the Chapter 2, several resource synchronization protocols for multiprocessor will be listed. Chapter 3 and 4 focus on the test-bed installation, verification and new protocols' implementation. In the Chapter 5, we will evaluate the performances for those resource synchronization protocols which we are interested in with taking the overheads into account. Lastly, we will draw the conclusions based on our current researches in Chapter 6.

1.2 Scheduling Algorithms for Multiprocessor

Actually, we do have several efficient scheduling algorithms for uni-processor system *e.g., Earliest Deadline First* (EDF) and *Rate Monotonic* (RM) Algorithm. Under the multiprocessor environment, we have two classes of the scheduling algorithms, *global* and *partitioned* [2]. But in our thesis, another type of scheduling algorithm will be taken into consideration named *semi-partitioned* scheduling [3]. Global based scheduling algorithms use a single scheduler for all processors. All the eligible tasks are stored in a single priority-ordered queue, the global scheduler select the highest priority tasks from this queue for execution. That means each task can be executed on any processor. Global scheduler allows both normal execution and critical section to migrate between processors. That makes it possible that a task can be preempted on one processor and resumed on another processor. Partitioned based scheduling algorithms start by partitioning tasks among processors. Each processor is associated with a separated ready queue for eligible tasks. Then, tasks within each processor are scheduled by uni-processor scheduling algorithms such as *fixed priority* (FP) scheduling or *Earliest Deadline First* (EDF). The most difference between partitioned and semi-partitioned scheduling is that, under pure partitioned scheduling, no migration is allowed, under semi-partitioned, the migration is allowed under some criteria, thereby improving schedulability. The most advantage of the partitioned based algorithms is that they reduce the multiprocessor

scheduling problem to a set of uni-processor scheduling. The Figure 1.1 has described the approaches how the global and (semi-)partitioned scheduling algorithms work visually.

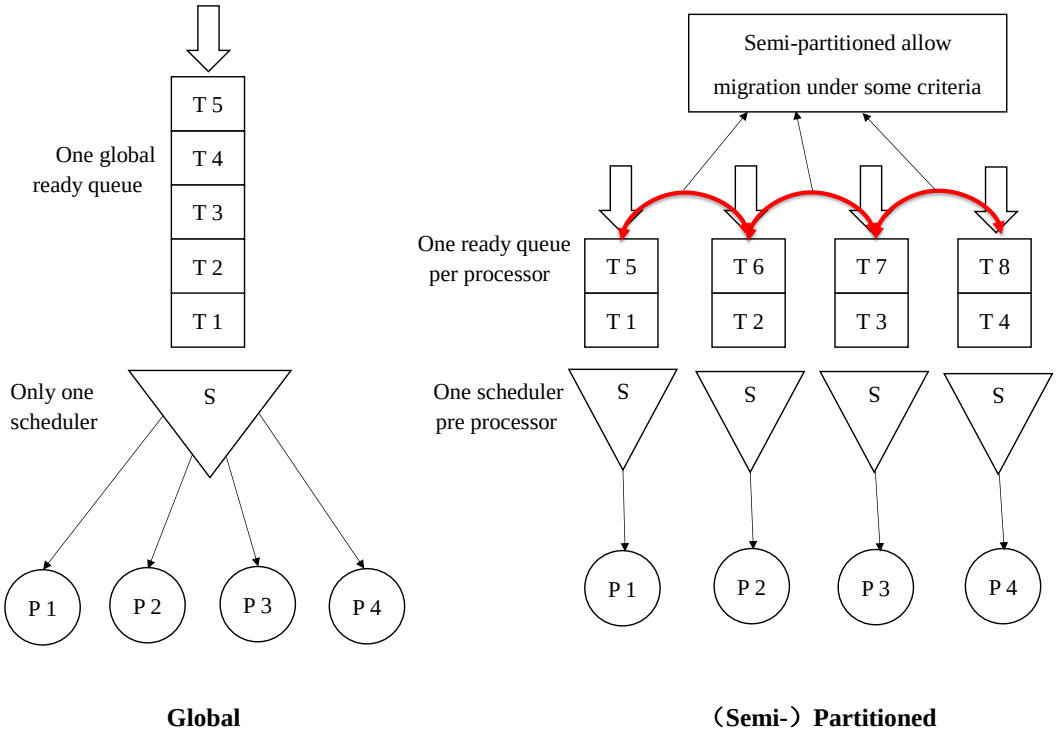

FIGURE 1.1: The approaches how the global and (semi-)partitioned scheduling algorithms work.

Every coin has its two sides. Both algorithms have their own advantages and disadvantages. The Table 1.1 has listed all the Pros and Cons for both scheduling algorithms.

There is another scheduling algorithm named hybrid scheduling as well. Such algorithm is neither pure partitioned nor pure global scheduling algorithm. For example, some kinds of tasks are forbidden to migrate between processors, but others are allowed to do the migration. One famous example of the hybrid scheduling algorithm is the *cluster-based* scheduling algorithm. Under cluster-based scheduling, processors are divided into different clusters. Tasks are statically allocated to different clusters, and in each cluster, tasks are scheduled by global scheduling algorithms [4].

In our thesis, we adopt the *Partitioned Fixed Priority* scheduling algorithm, which

	Global	Partitioned
Pros	Automatic load balancing Optimality possible Many elegant algorithms available More efficient reclaiming	Supported by automotive industry Limited migrations (semi-partitioned) Isolation between processors Simple to understand and implement Mature scheduling framework
Cons	Migration costs Inter core synchronization Loss of cache affinity Weak scheduling framework	Cannot exploit unused capacity Rescheduling not convenient NP-hard allocation (bin-packing problem)

TABLE 1.1: The advantages and disadvantages of global and partitioned scheduling algorithms.

has the following advantages: (i) Partitioned based scheduling suffers from lower overheads with limited migration comparing to the global scheduling. (ii) There are well-established theories support based on uni-processor. (iii) The P-FP also supports semi-partitioned scheduling. (iv) All those synchronization protocols that we are interested in can be implemented under P-FP.

1.3 The Necessity of Resource Synchronization

After confirming the scheduling algorithm that will be adopted in our thesis, we need get some basic knowledge about the resource synchronization. In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior so parts of the program where the shared resource is accessed is protected. This protected section is the *critical section*. It cannot be executed by more than one process. Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that does not allow multiple concurrent accesses [5]. The resource synchronization protocol is used to organized such critical section accesses from different tasks.

In the resource synchronization, there are two concepts that we need to keep in mind: *critical section* and *semaphore*. *Critical section* is a piece of code executed under mutual exclusion constraints that has been explained above. *Semaphore* is a synchronization

tool that can be used by tasks to build critical sections. A semaphore is a kernel data structure that, apart from initialization, can be accessed only through two kernel primitives, usually called *wait* and *signal*. When using this tool, shown in Figure 1.2 each exclusive resource R_k must be protected by a different semaphore S_k and each critical section operating on a resource R_k must begin with a $wait(S_k)$ primitive and end with a $signal(S_k)$ primitive [1].

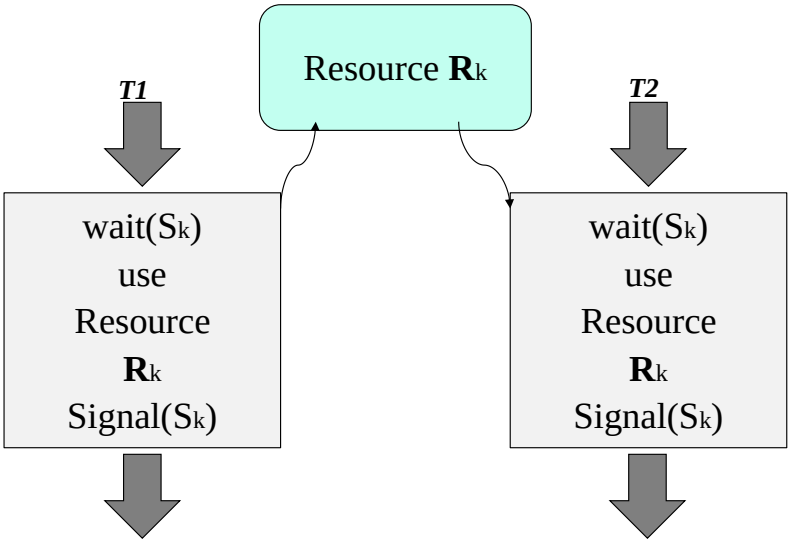

FIGURE 1.2: How the semaphore works to protect critical section.

Unfortunately, the need to share resources between tasks operating in a preemptive multitasking environment can create conflicts. Two of the most common problems are *deadlock* and *priority inversion*, both of which can result in application failure. Deadlock may cause the whole system crash, priority inversion may break the predictability of the real-time system, both of them can make our real-time system fail and cause some unexpected disasters. Deadlock, shown in Figure 1.3, is a special case of nested resource locks. Task A holds the resource 1 and is waiting for resource 2. Meanwhile, task B holds the resource 2 and is waiting for resource 1. That means, both tasks hold one resource and wait for another one. Thus, neither of these two tasks can execute. But, in our thesis, we don't consider about nested resources, which will make the task-set

construction and results analysis too complex. Therefore, we only need to take care about the priority inversion phenomenon when studying those protocols.

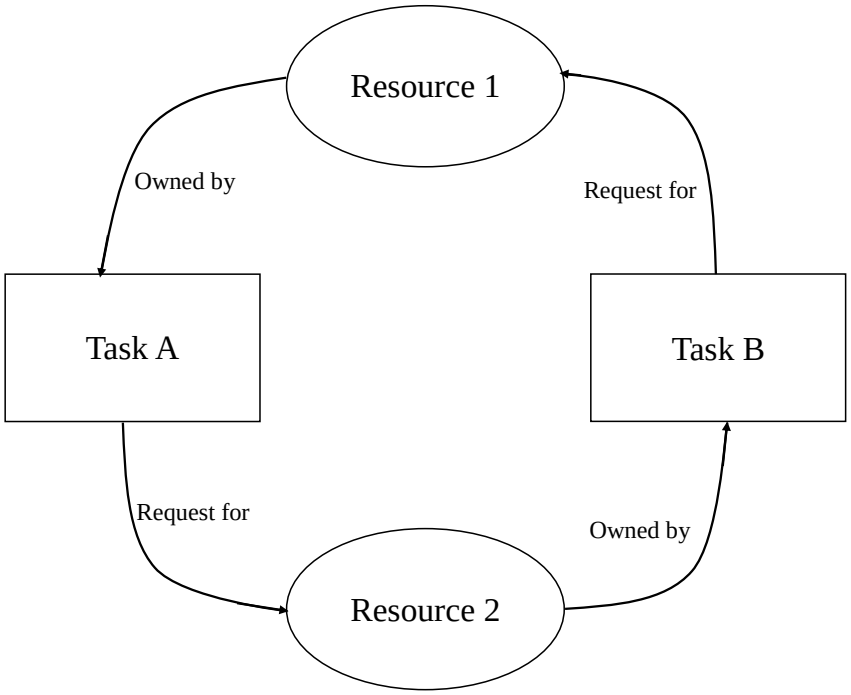

FIGURE 1.3: How the deadlock appears.

Ideally, a high-priority task τ should be able to preempted lower priority jobs immediately upon τ's initiation. Priority inversion is the phenomenon where a higher priority task is blocked by lower priority tasks. A common situation arises when two tasks attempt to access shared resource. To maintain consistency, the access must be serialized. If the higher priority task gains access first then the proper priority order is maintained; however, if the lower priority task gains access first and then the higher priority task requests access to the shared resource, this higher priority taskis blocked until the lower priority task completes its access to the shared resource. Thus, *blocking* is a form of priority inversion where a higher priority task must wait for the processing of a lower priority task [6]. There are two types of priority inversion: bounded and unbounded priority inversion. In the bounded priority inversion, task τ_L with low-priority acquires the lock before the releasing of the task τ_H with high-priority. High-priority task τ_H is forced to wait the task τ_L to release the sharing resource. The worst-case wait time for Task τ_H is equal to the length of the critical section of Task τ_L. Bounded priority inversion is also called as direct blocking, which is the price we in essential have to pay for ensuring

mutual exclusion. Such predictable direct blocking won't generate any problem for the real-time system considering the worst case execution time of the high-priority task can be bounded. Unbounded priority inversion, shown in Figure 1.4, occurs when an intervening task extends a bounded priority inversion. Task τ_M with medium-priority can preempt task τ_L with low-priority in its critical section. Only when task τ_M completes its job, task τ_L can resume its critical section. Meanwhile, task τ_H with high-priority cannot execute until task τ_L release the resource. As a result, the worst-case wait time for Task τ_H is now equal to the sum of the worst-case execution times of Task τ_M and the critical section of Task τ_L. In some cases, priority inversion can occur without causing immediate harm the delayed execution of the high priority task goes unnoticed, and eventually the low priority task releases the shared resource. However, there are also many situations in which priority inversion can cause serious problems. If the high priority task is left starved of the resources, it might lead to a system malfunction or the triggering of predefined corrective measures, such as a watchdog timer resetting the entire system. Priority inversion can also reduce the perceived performance of the system. Because priority inversion results in the execution of a lower priority task blocking the high priority task, it can lead to reduced system responsiveness, or even the violation of response time guarantees.

1.4 Resource Synchronization Protocols in Uni-processor

To solve these problems mentioned above, several resource synchronization protocols have been developed and investigated deeply. In this section, four well-developed protocols in uni-processor environment will be introduced.

Firstly, the *non-preemptive protocol* (NPP) is the simplest protocol to prevent the unbounded priority inversion phenomenon. Under the NPP, once a task starts its execution for critical section, it cannot been preempted by any other tasks until it finishes the execution and releases the resource. The most advantage of the NPP is that the implementation of NPP is quite simple [7]. We only need to raise the task's priority to the highest priority that the system supports temporarily when it executes the critical section. Once it finishes the execution of the critical section, the priority will be lowered to the original one, in order to reduce the influence to other tasks with the higher priority.

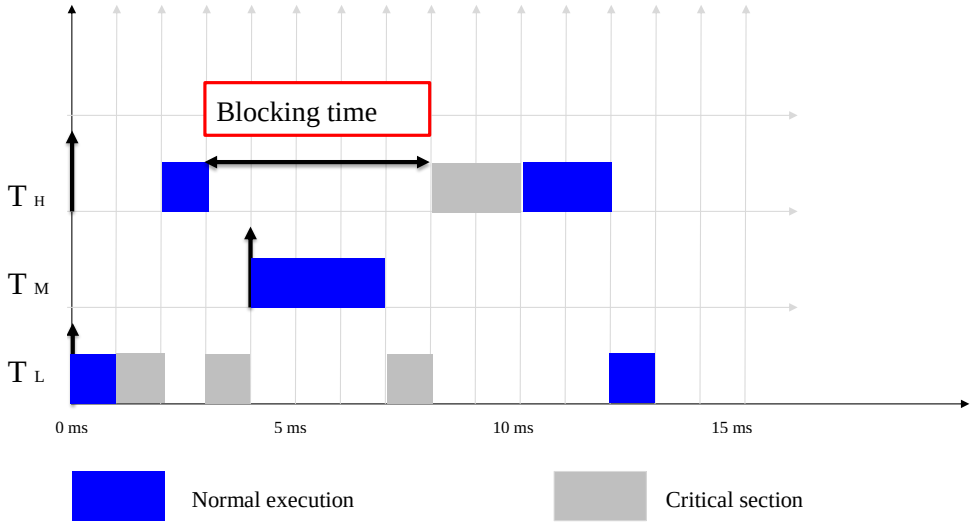

FIGURE 1.4: Priority inversion phenomenon.

Secondly, the *priority inheritance protocol* (PIP) is also a simple solution to the problem of unbounded priority inversion caused by resource constraints. Under the PIP, if a lower-priority task τ_L locks the resource and starts execute the resource at first, the task τ_H with the higer priority comes later, and it also want access to the same resource. Once τ_H requests to the same resource locked by τ_L, the priority of τ_L will be raised to the priority which is equal to the priority of τ_H. That means, after the resource requesting by τ_H, τ_L can not be preempted by any other tasks with the priority higher than τ_L but lower than τ_H. Thus, a high-priority task τ_H can be blocked for at most the duration of $min(n, m)$ critical sections. Where n is the number of lower-priority tasks that could block τ_H and m is the number of distinct semaphore that can be used to block τ_H. It bounds the maximum blocking time of each task [1].

Thirdly, the *Priority Ceiling Protocol* (PCP) extends the ideas of PIP to solve the problems of unbounded priority inversion, chain blocking and deadlocks, while at the same time minimizing inheritance-related inversions.[8] Each semaphore is assigned a priority ceiling equals to the priority of the highest priority task that can lock it. When a task acquires a shared resource, the task's priority is temporarily raised to the priority

ceiling of that resource. The priority ceiling must be higher than the highest priority of all tasks that can access the resource, in order to ensure that a task owning a shared resource won't be preempted by any other task attempting to access the same resource. When the resource holder releases the resource, the task is returned to its original priority level. Under the PCP, a high-priority job τ_H can be blocked for at most the duration of one critical section [6].

Lastly, the *stack resource policy* (SRP) is a resource allocation policy which permits processes with different priorities to share a single run-time stack. It is a refinement of the *priority ceiling protocol* (PCP), which strictly bounds priority inversion and permits simple schedulability tests. With or without stack sharing, the SRP offers the following improvements over the PCP: (1) it unifies the treatment of stack, reader-writer, multi-unit resources, and binary semaphores; (2) it applies directly to some dynamic scheduling policies, including *earliest deadline first* (EDF), as well as to static priority policies; (3) with EDF scheduling, it supports a stronger schedulability test; and (4) it reduces the maximum number of context switches for a job execution request by a factor of two. It is at least as good as the PCP in reducing maximum priority inversion [9].

All these four protocols applied in uni-processor environment are the fundamental of most resource synchronization protocols used in the multiprocessor, that we will discuss in the next chapter.

Chapter 2

Resource Synchronization Protocols for Multiprocessor

Unlike the uni-processors, in the multiprocessor environment, we should consider more elements which can affect the scheduling behavior. In the uni-processor environment, we only consider about the sequence when we allocate the tasks, meanwhile, all the critical section can only be executed on that processor. However, everything will be changed when it comes to multiprocessor situation. Under the P-FP multiprocessor scheduling algorithms, we should make the decision to allocate which task to which processor at first. Besides the allocation of tasks to different processors, the tasks' sequence within one processor can also raise new challenges because of the global resource existing. When dealing with the sharing resources, we should be careful of the resource's location and task's migration. That means we need to manage the global resource synchronization.

In this chapter, we will discuss different protocols of the resource synchronization, that we are interested in under the multiprocessor environment. Some of them have been implemented some where, some of them are not. We will only discuss the theories of them in this chapter, the implementation issues will be investigated in Chapter 4.

2.1 Resource Synchronization Protocols for Multiprocessor

The existing resource synchronization protocols can be categorized as suspension-based and spinning-based protocols. Under a suspension-based protocol, a task will suspend if the resource is locked by other tasks when it tries to lock the sharing resource. That means, the processor can be taken over by the lower-priority tasks, when the high-priority task is blocked by another task who is executing the critical section on other processor. Once the resource is released by other task, the high-priority task will resume to the processor and preempt the execution of the lower-priority tasks. Such suspension-based protocols can make the full use of the processor when the high-priority tasks are waiting for the release of the resources. However, under a spinning-based protocol, a task will keep the processor and perform spin-lock when the requesting resource is locked by others. What means, the lower-priority tasks can never occupy the processor if there is at least one higher-priority task hasn't finished its execution.

Under the multiprocessor environment, the location of resources should be taken into consideration. There are two major types of semaphore protocols. In *shared-memory* locking protocols, tasks execute critical sections *locally*, in the sense that each task accesses shared resources directly from the processor on which it is allocated. That means, for each task, it can access to the shared resource wherever it is located on the preocessors available on the system. For example, there is the case under the classic *Multiprocessor Priority Ceiling Protocol* (MPCP) [10, 11]. In contrast, in *distributed-based* locking protocols, each resource is accessed only from a designated *synchronization processor* (or *cluster*). Under such protocols, which derive their name from the fact that they could also be used in distributed systems (*i.e.*, in the absence of shared memory), require critical sections to be executed *remotely* if tasks access resources not local to their assigned processor [12]. In other word, each resource has been assigned to one specific processor, each request for the shared resource should be execute on that specific processor, which requires the migration is possible for each task which is requesting the resources locating on remote processors.

In this section, we will introduce four famous protocols under *Partitioned (or semi-partitioned) Fixed Priority* (P-FP) briefly. These four protocols have been developed

and investigated very well, meanwhile, have been implemented somehow and somewhere. Some of them will participate in our evaluation in Chapter 5.

2.1.1 Multiprocessor Priority Ceiling Protocol

Multiprocessor Priority Ceiling Protocol (MPCP) was proposed by Rajkumar in [11], which extends *Priority Ceiling Protocol* (PCP) to shared memory multiprocessor environment, hence allowing for synchronization of tasks sharing mutually exclusive resources using P-FP scheduling. Under the MPCP, resources are divided into local and global resources. The local resources are protected using a uni-processor synchronization protocol like PCP. All the tasks which will access to the local resources will be allocated to the processor where the resource is assigned on. Those tasks only access to the global resources can be allocated freely. We only focus on the global resources in our thesis. MPCP is suspension-based protocol, under which tasks waiting for a global resource suspend and are en-queued in an associated prioritized global wait queue. A task blocked by a global resource suspends and makes the processor available for the local tasks. Meanwhile, the priority of a task within a *global critical section* (gcs), in which it request a global resource, is boosted to the priority which is greater than the highest priority among all local tasks (also can be treated as normal execution). This priority is called remote ceiling. A *gcs* can only be preempted by other *gcs*s that have higher remote ceiling not by a non-critical section. That means, the execution of the critical section always has the higher priority than the normal execution on the local processor no matter how are the original priority levels they have.

As a consequence of the ceiling raising, the blocking time of a task, not only the local blocking, the remote blocking where a task is blocked by tasks with any priority executing on other processors should be included. However, the maximum remote blocking time of a task is bounded and is a function of the duration of other tasks' critical sections [11].

Global critical cannot be nested in local critical sections and vice versa. Global resource potentially lead to high blocking times, thus tasks sharing the same resources are preferred to be assigned to the same processor as long as possible. Thus, we have to figure

out an algorithm that attempts to reduce the blocking times by assigning tasks to appropriate processors [13]. Such partition problem is another complex issue that we will not involve in our thesis.

2.1.2 Distributed Priority Ceiling Protocol

Distributed Priority Ceiling Protocol (DPCP) [11, 14] extends PCP as well. Unlike the typical MPCP, under DPCP, resources are assigned to processors, a task accesses a resource via an *Remote Procedure Call* RPC-like invocation of an agent on the resource's processor that performs the access, which means the task need to migrate to a remote processor if it requests a resource is locating on another processor. Requests for global resources cannot appear in nested request sequences, which may cause the deadlock. Requests for such resources are ordered by priority and execute at elevated priority levels so that they complete more quickly. Initially, Rajkumar proposed that all global resources were assigned to a single synchronization processor. In that way, all the critical sections executed on that processor follow the PCP. This was then generalized in the same paper to allow multiple synchronization processors, but again each resource was assigned to one synchronization processor.

The DPCP provides one resource agent for each resource and each task. Resource agents are subject to priority boosting, which means that they have priorities higher than any regular task (thus execution for critical section cannot be preempted by any normal executions). Meanwhile, resource agents acting on behalf of higher-priority tasks may still preempt agents acting on behalf of lower-priority tasks [14].

A task that is trying to access to a global resource will migrate to the synchronization processor during the execution of that critical section. After the execution for critical section, the task will migrate back to the original processor where it is released at. Of course, tasks request the same resource on its synchronization processor perform the PCP similar to the situation in the uni-processor [15].

2.1.3 Flexible Multiprocessor Locking Protocol

Flexible multiprocessor locking protocol (FMLP) [16] can be applied to both global and partitioned scheduling algorithms. In FMLP, resources are categorized into short and

long resources, and whether a resource is short and long is specified by users. There is no limitation on nesting resource accesses, however that requests for long resources cannot be nested in requests for short resources [17].

Under FMLP, deadlock is prevented by grouping resources. A group includes either global or local resources, and two resources are in the same group if a request for one is nested in a request for the other one. Similar to other sigle resource, the requests for the group are also mutual exclusive. A group lock is assigned to each group and only one task can hold the lock of the group at any time. Meanwhile, the types of the locks within the FMLP are different. The tasks that are blocked on short resources perform busy-wait and are added to a FIFO queue. And once tasks that access short resources, they will hold the group lock and execute non-preemptively. To the contrary, tasks that access long resources hold the group lock and execute preemptively using priority inheritance (*i.e.*, is inherits the highest priority among all tasks blocked on any resources within the group). Tasks are blocked on long resources are also added to a FIFO queue. The main problem of the FMLP is that, there is no concrete solution on how to assign a global resource as long or short and it assumed to be user defined [17]. In an evaluation of partitioned FMLP [18], the authors differentiate between long FMLP and short FMLP where all global resources are only long and only short respectively. Thus, long FMLP and short FMLP are suspension-based and spinning-based synchronization protocols respectively. In both alternatives the tasks accessing a global resource executes non-preemptively and blocked tasks are added into a FIFO-based queue.

2.1.4 Distributed FIFO Locking Protocol

Distributed FIFO Locking Protocol (DFLP) [12] was inspired by the FMLP$^+$ [19], which relies on simple FIFO queue to avoid starvation of the resources. Under DFLP, conflicting requests for each serially-reusable resource l_q are ordered with a per-resource FIFO queue FQ_q. Requests for l_q are served by an agent A_q assigned to l_q's cluster $C(l_q)$. Resource requests are processed according to the following rules:

1. When J_i issues a request R for resource l_q, J_i suspends and R is appended to FQ_q. J_i's request is processed by agent A_q when R becomes the head of FQ_q.
2. When R is complete, it is removed from FQ_q and J_i is resumed.

3. Agent A_q is inactive when l_q's request queue FQ_q is empty and active when it is processing requests. Active agents are scheduled preemptively in order of increasing issue times with regard to each agent's currently processed request (*i.e.*, an agent processing an earlier-issued request has higher priority than one serving a later-issued request). Any ties in request times can be broken arbitrarily (*e.g.*, in favor of agents serving requests of lower-indexed tasks).

4. Agents have statically higher priority than jobs (*i.e.*, agents are subject to priority-boosting).

2.2 New Protocols under Consideration

All the protocols that have mentioned above have been proposed for many years. There are lots of investigations that have been finished both in theory and implementation aspects. In this section, two new protocol will be introduced, also the implementation issues will be covered in Chapter 4.

2.2.1 Distributed Non-preemptive Protocol

Distributed Non-preemptive Protocol (D-NP) is a simple protocol which is developed from the DPCP. Different to the DPCP, under the D-NP, once a task starts the execution for critical section, it cannot been preempted by any other tasks even those tasks with the higher priority and request differnt resources. Under thr typical DPCP, the task which is holding the resource can be preempted by those tasks with the higher ceiling priority, and resumes after all the higher-ceiling tasks finishing their execution. In the real implementation, the frequent preemption will cause frequent context switch which will create large overhead. We notice that such non-preemptive execution can reduce the context switch overhead obviously which can improve the holistic performance. For those task-set which all the tasks have tiny critical section length, D-NP will reduce the average execution time by eliminating several needless context switch overheads. In that situation, the overhead of the implementation may cause a significant influence to the performance of the protocol. However, every coin has its two sides, it may increase the direct blocking time of the task which has the highest priority. We construct one symbol to indicate the critical section for each task and resource as $S_{resource\_id,job\_number}$,

so $S_{3,1}$ means the length of the critical section which task 1 accesses to the resource 3. Similar to the PCP [20], under the DPCP, the worst case blocking time for the task T_h which has the highest priority can be defined in the Equation 2.1. That means the worst case blocking time for each resource access is equal to the longest of other task's critical section which accesses to the same resource, and the total blocking time is equal to the sum of all the single worst case blocking time within the task due to multi-resource accessing. Under the DNP, the worst case blocking time of the task T_h which has the highest priority has been shown in the in the Equation 2.2. In this equation, we can conclude that, the task can be blocked by any other tasks' critical section due to the non-preemptive execution of the critical section. Thus for each resource access, the worst case block time is equal to the longest critical section among all the tasks (no matter which resource it is accessing). So the total blocking time is equal to the number of critical section within the task multiple to the longest critical section.

$$\mathbf{B}_{DPCP} = \sum_{i=1}^{n} (\max \mathbf{S}_{n,j}) \tag{2.1}$$

$$\mathbf{B}_{DNP} = \mathbf{n} * (\max \mathbf{S}_{*,j}) \tag{2.2}$$

From those two equations, we can easily see that $B_{DPCP} <= B_{DNP}$. However, if such increase of the blocking time in theory can be eliminated by the reduce of the overheads caused by frequent context switch in the real implementation, we can still consider the DNP is better than DPCP under some specific situations. All the evaluation work will be covered on the Chapter 5.

2.2.2 Multiprocessor Resource Sharing Protocol

Multiprocessor Resource Sharing Protocol (MrsP) was proposed by A. Burns and A.J. Wellings in [15]. The distinctive nature of this protocol is that tasks waiting to gain access to a resource must service the resource on behalf of other tasks that are waiting for the same resource (but have been preempted) on the remote processor. MrsP is a variant of *Multiprocessor Stack Resource Policy* (MSRP) [21]. The basic idea of the protocol can be described as follows:

1. All resources are assigned a set of ceiling priorities statically, one per processor (for those processors that have tasks that use the resource), for processor p_k it is the maximum priority of all tasks allocated to p_k that use the resource.

2. An access request on any resource results in the priority of the task being immediately raised to the local ceiling for the resource.

3. All the tasks waiting for a resource are dealt with in a FIFO order (are added into a FIFO queue).

4. While waiting to gain access to the resource, and while actually using the resource, the task continues to be active and executes (possible spinning) with priority equal to the local ceiling of the resource (spin-lock).

5. Any task waiting to gain access to a resource must be capable of undertaking the associated computation on behalf of any other waiting task.

6. This cooperating task must undertake the outstanding requests in the original FIFO order.

To conclude, there are three distinct features of MrsP:

1. Requests for the same resource are ordered in FIFO queue.

2. Tasks are spinning with priority equal to the local ceiling of the resource while it is waiting to access the resource. (only when tasks are waiting for the resources as well as executing the critical section are spinning).

3. Help mechanism: the spin task who is waiting for one resource uses its 'wasted cycles' to help the resource holing task make progress. (when the task who is holding the resource is preempted on its processor). Meanwhile, the sequence to find the helper should follow the original requests FIFO order (traverse the FIFO queue).

The first rule ensures the fairness of the resource requests. The second rule can prevent the priority inversion phenomenon. The combination of first and second rule can bound the maximum blocking time of each task. Meanwhile, the third rule can reduce the average response time by taking full use of the processors' wasted cycles on the basis

of guarantee the time constraints (without increasing any other tasks' response time in theory). These three main features also are the main points that we need to take care about when we implement the MrsP in Chapter 4.

Chapter 3

Test-bed Installation and Testing

All the implementation and evaluation in this thesis are based on the $LITMUS^{RT}$, which is abbreviated from the LInux Testbed for MUltiprcessor Scheduling in Real-Time system [22, 23]. $LITMUS^{RT}$ was originally launched at the University of North Carolina at Chapel Hill under the direction of James H. Anderson. Now, The maintainer and main developer behind $LITMUS^{RT}$ is Björn Brandenburg of the Max Planck Institute for Software Systems (MPI-SWS).

As we all know, besides the $LITMUS^{RT}$, there are other real-time system test-bed available (*e.g.*, RTEMS [24], QNX Neutrino [25] and so on). The reasons why we choose the $LITMUS^{RT}$ as our experiment platform are as follows:

1. $LITMUS^{RT}$ is open source code, which can be downloaded from the official website for free.
2. It is a well-established evaluation platform in the real-time research community.
3. It contains several useful tools can be easily applied for feature tracing, which are significant for evaluation.
4. It has supported a variety of scheduling and resource synchronization protocols. There is no need for us to implement all the protocols that we are interested in from the very beginning.
5. It employs a plug-in based architecture, where different scheduling algorithms can be plugged activated and changed dynamically at run time.

6. It is well distributed, which makes it convenient for the new developer to implement their own algorithms as new plug-ins, implement a new protocol inside the existed plug-in as well.

3.1 Synopsis of $LITMUS^{RT}$

The $LITMUS^{RT}$ patch is a real-time extension of the Linux kernel with a focus on multiprocessor real-time scheduling and resource synchronization. The Linux kernel is modified to support the sporadic task model and modular scheduler plugins. Clustered, partitioned, and global scheduling are included, semi-partitioned scheduling is supported as well.

$LITMUS^{RT}$ provides a useful experimental platform for applied real-time systems research. Meanwhile, it services as a proof of concept, showing how algorithms such as *Pfair* schedulers and predictable multiprocessor locking protocols can be implemented on current hardware. All we talk about in this thesis is using the plugin P-FP, which means *partitioned fixed priority*, which also supports the *semi-partitioned scheduling*. All the resource synchronization protocols are implemented under that environment.

3.2 Installation and Modification

3.2.1 Installation

The general installation guide can be found on the official website of $LITMUS^{RT}$. But there are still several points needed to be noticed when installing the $LITMUS^{RT}$ to the Linux system (take Ubuntu 14.04 as example). First of all, in the latest version of $LITMUS^{RT}$ based on Linux kernel 4.1.3, however when we compile the kernel, there are two bugs in the file 'drivers\media\usb\as102\as102_usb_drv.c'. We need to add two '&' inside the line 117 and 135. Moreover, to use the trace tool included by *Feature Trace Tools*, we need to disable the option *Build a relocatable kernel* when we configure the kernel. Such option can not be disabled unless we disabled the other option named *EFI runtime service support* at first. Considering the demand for the feature tracing for each protocols, we should install the Ubuntu into a real computer rather than a virtual

machine. That because, virtualization is not transparent with regard to timing. Also, virtualized operating system cannot possibly offer better timing than the underlying hypervisor for the overheads occurs. While, what we are investigating are those protocols used for real-time system, which are sensitive to the timing. Thus, we should ensure the timer of our platform as accurate as possible.

Besides the main patch for the Linux kernel, there are also user space library and feature trace tools available on the web-page, which can make our work more efficient and go smoothly. There is one useful tasks allocation tool in the *liblitmus* named 'rtspin'. Meanwhile, schedule tracing tool and traced data drawing tool are also available in *ft_tools*.

3.2.2 Modification

To meet the requirement of our experiment, we need modify the excited tools at first. The original task allocation tool can set only one critical section within a task. And the position of that critical section is set randomly. Meanwhile, the critical section can only be assigned to the processor where the task is assigned. That means we cannot simulate the distributed-base protocols' behavior. In order to verify the existing protocols' behavior provided by the $LITMUS^{RT}$, we need set the pattern of task statically by our own. We should make it possible to define each chunk inside one task, including the chunk's length and the type of the chunk (normal execution or critical section) as well as the resource id (if needed). For the support of the distributed-based protocol, the critical section's migration option should be added. So, we need to modify the original task allocation tool 'rtspin' at first. We construct new task allocation format, to support multi-resources access within one task, and both normal execution and critical section's position have to be defined statically under that format. Now, the modified format of the tasks allocation tool is as follows:

```
rtspin -w -p 1 -N 3 -M 2 -X DPCP -D 0:10,1:40,0:10 -q 10 60 130 5 &
```

- ***-w*** wait for a synchronous release. The trailing & starts the process in the background and is useful for scripting the creation of multiple waiting tasks.
- ***-p*** task's partition, indicates which processor this task is assigned to initially.

- ***-N*** the number of chunks that this task contains.
- ***-M*** the processor where the resources are assigned to, in order to support for distributed protocol.
- ***-X*** resource synchronization protocol that will be adopted this time.
- ***-D*** the task's format. Each chunk has two parts, divided by colon, first part indicates the type, 0 means the normal execution, other number indicates the resource id (critical section naturally), and second part means the length of that chunk. Different chunks are divided by comma.
- ***-q*** task's priority (1 indicates the highest priority, 512 indicates the lowest priority)
- ***60 130 5*** last three numbers indicate the: $execution\_time \mid period \mid whole\_run\_time$

Following the manual above, we can explain the sample task allocation as follows: this task consists of three chunks, the normal execution will be allocated on processor 1; the critical section will execute on processor 2; these three chunks are connected in series, first is the 10 ms normal execution, second is the 40 ms for the critical section 1, last is another 10 ms normal execution; such task has priority 10; the whole execution time of this task is 60 ms; the period is 130 ms; the run time is 5 s.

Although in the distributed protocols, resources can be allocated in different processor. That means we need assign a specific processor for each chunk within one task. Maybe the format of one chunk should be set as type:length:processor. However, to reduce the migration cost, we assume all the resources (which one task may access) are assigned to the same processor. So, there is no need for us to add this processor option inside the task's format.

3.3 Testing

After the installation, we noticed that in the P-FP plugin, there are several resource synchronization protocols have been implemented. For the following two reasons, we designed some experiments to test these protocols. For one thing, we need to verify whether these protocols can operate as expected. For another, such experiments can

help us get familiar with the way how the test-bed works as soon as possible. In this section, the verification for PCP, MPCP and DPCP will be shown.

3.3.1 verification for PCP

It's known to all, the *Priority Ceiling Protocol* (PCP) is one of the most famous resource synchronization protocols used for uni-processor environment. It is applied widely in the partitioned multiprocessor environment. To simplify the verification, we only consider one resource shared by three tasks in one processor along with two pure normal execution tasks. The result has been shown in Figure 3.1. We assume T_1 has the highest priority P_1, T_2 and T_3 have same priority P_2, T_4 has the middle priority P_3, and T_5 has the lowest priority P_4.

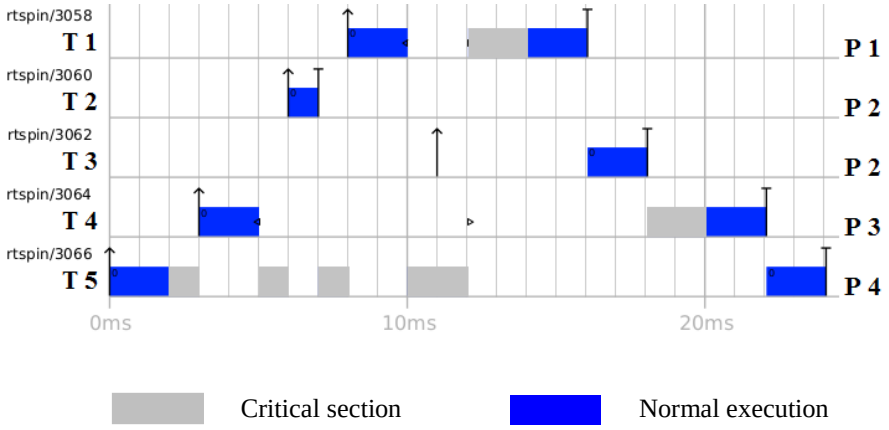

FIGURE 3.1: PCP's behavior for one resource.

Time	0 ms	2 ms	5 ms	10 ms	12 ms	14 ms	20 ms
Priority	P_4	P_4	P_3	P_1	P_4	P_4	P_4

TABLE 3.1: The priority of the task T_5 changes along with time.

From the result, we can see, on 2 ms, T_5 enter the *critical section*, on 5 ms, T_4 is blocked by T_5 for the same resource request, so T_5 inheritance the priority of T_4, but T_2 is released on 6 ms with out resource request, T_5 is preempted by high priority task. On 10 ms, T_1 is blocked by T_5, so T_5 inheritance the highest priority from T_1. Thus, when T_3 is released on 11 ms, it cannot preempt T_5 like what T_2 has done on 6 ms. From the behavior above, the *Priority Inheritance* (PI) has been proved. The priority of the

task T_5 has shown in Table 3.1, which is dynamic updated according to the resource request by other higher-priority tasks. When the T_5 finishes the critical section on 12 ms, T_1 enter the critical section first even though T_4 requests the resource before T_1 does. That because the wait queue for one resource is ordered by tasks' original priorities. Meanwhile, the resource ceiling property has shown in Figure 3.2 as well as Table 3.2. The only thing that we need care about is on 5 ms, T_3 wants enter its critical section, but it is blocked by T_5 due to the resource ceiling issue. After that we can draw the conclude that, the PCP has been implemented correctly.

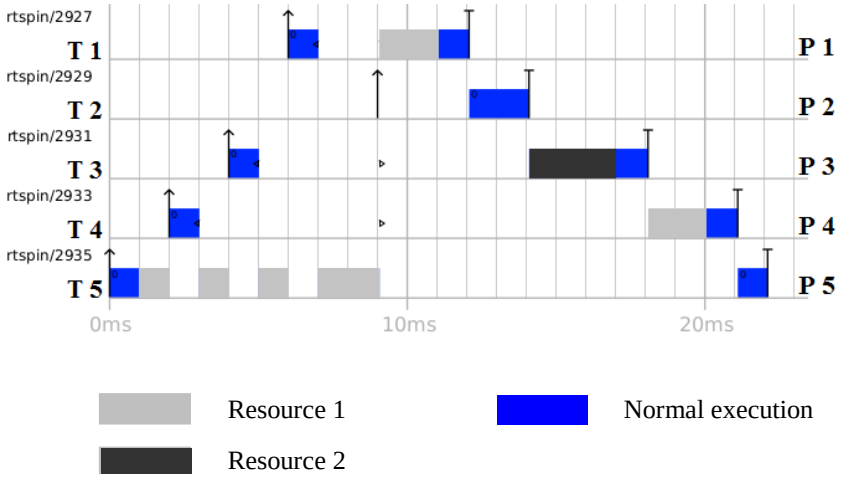

FIGURE 3.2: PCP's behavior for two resource.

Time	0 ms	1 ms	5 ms	11 ms
Ceiling for R1	NULL	NULL	P_3	P_3
Ceiling for R2	NULL	P_1	P_1	P_4

TABLE 3.2: The ceilings for both two resources.

3.3.2 verification for MPCP

The verification for MPCP will be a little bit complex, for the multiprocessor environment and multi-resources access(without nested). We consider a two-processors environment and there are three resources are available. The result is shown in Figure 3.3. T_1 to T_5 are assigned to processor 1, T_1 and T_2 have the highest priority, T_5 has the lowest priority on processor 1. T_6 to T_8 belong to processor 2. T_6 has the highest priority and T_8 has the lowest priority on that processor.

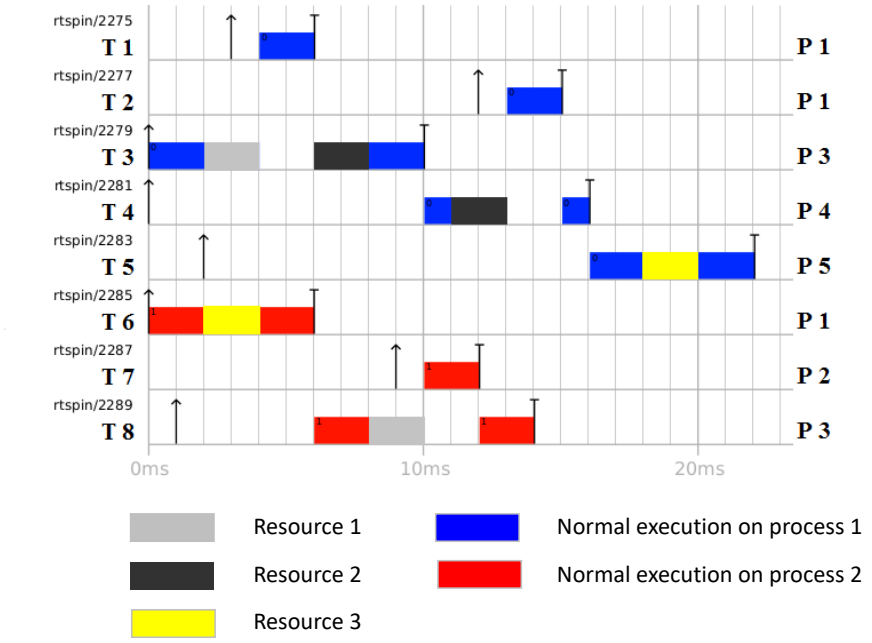

FIGURE 3.3: MPCP's behavior on two processors for three resources (a).

From the result shown in Figure 3.3, we can notice, once a task enters the critical section, it cannot be preempted by the normal execution. We can see, on 3 ms and 12 ms, two normal execution tasks T_1 and T_2, who have the highest priority on processor 1 are released. But they cannot interrupt the critical section executed by T_3 and T_4. The same situation happens on processor 2. The feature that the critical section always having the higher priority than normal execution no matter what the original priorities they have. This experiment only proof the behavior within each processor, to verify the behavior of requesting global resource between processors, we designed another experiment, the result is shown in Figure 3.4. Now we have two sharing resource between two processors. T_1 has the higher priority on processor 1 and T_3 is the only one task assigned to processor 2.

From the result, we can see, although the normal execution task cannot interrupt the critical section, but the critical section can be preempted by other critical section when it has the higher ceiling than the task under executing. Thus, on 5 ms, when T_3 releases the resource, T_1 resumes to execute, T_2 is preempted by T_1. Such preemption can happen because the T_1's critical section has the higher ceiling than T_2's. Now, we need to design

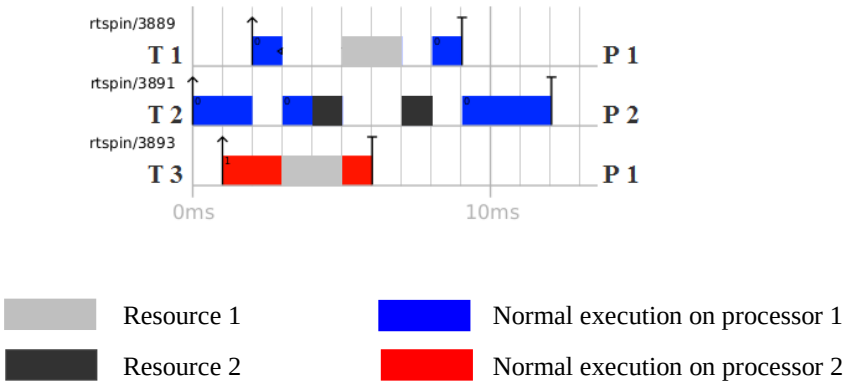

FIGURE 3.4: MPCP's behavior on two processors for three resources (b).

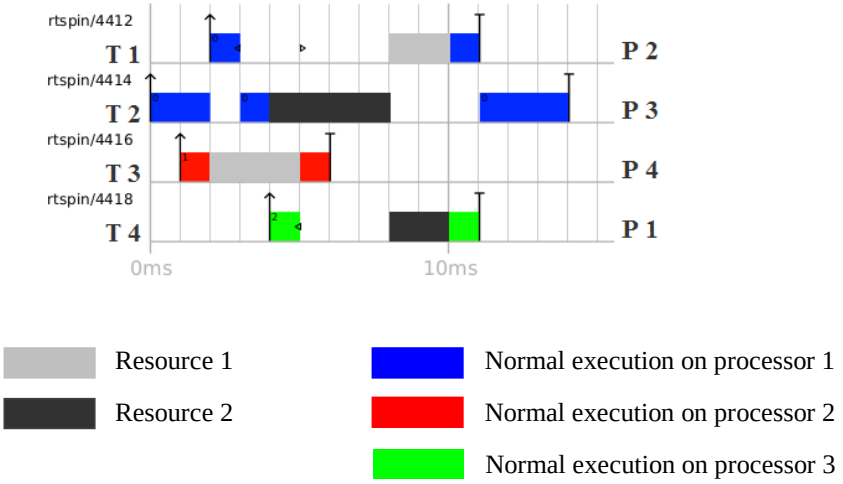

FIGURE 3.5: MPCP's behavior on two processors for three resources (c).

another experiment to verify how the ceiling works. The result is shown in Figure 3.5. Similar to last experiment, but in this experiment, we have another task T_4 which is assigned to processor 3 and has the highest priority among all the tasks.

Unlike to the last experiment, this time the T_1 cannot interrupt the execution of T_2, that because there is another task T_4 also wants access to the resource 2. That means

resource 2 has the ceiling priority equals to P_1, which is the highest priority among all the tasks. Once T_2 enters the critical section, it has the resource ceiling priority. Thus, T_1 cannot interrupt the T_2 any more, both resources's ceilings have been shown in Table 3.3. Now, all the properties of MPCP have been verified. For one thing, normal execution cannot preempt the critical section. For another, the resource's ceiling priority works for multiprocessor environment to prevent the priority inversion phenomenon.

Time	0 ms	2 ms	3 ms	4 ms	5 ms
Ceiling for R_1	NULL	P_2	P_2	P_2	P_2
Ceiling for R_2	NULL	NULL	NULL	P_1	P_1

TABLE 3.3: The ceilings for both resources under MPCP.

3.3.3 verification for DPCP

We assume that all the resources are allocated to one specific processor. Thus, the behavior of DPCP in the processor where the resources are assigned to is the same to PCP under uni-processor environment. The most different feature of DPCP is that the normal execution and the critical section are executed on different processors. Thus the migration within a task will be introduced. Considering the behavior of PCP has been verified in advance, the main purpose of this experiment is to verify the migration, and the behavior after migration. The result has been shown in Figure 3.6. T_1 has the highest priority, T_4 has the lowest priority. Only T_2 accesses the resource 2, other three tasks only access to resource 1.

The result has shown in Figure 3.6 contains two periods, the first period used to present the resource access, the second one is to present the processor where the normal execution and critical section execute on. In the first period, T_2 cannot preempt T_5 on 5 ms, because T_5 has the higher resource ceiling due to T_1 is requesting the same resource. But T_1 can preempt the T_2 on 8 ms for the higher resource ceiling it has. Meanwhile, the access to the same resource ordered by priority like what PCP should act as. In the second period, comparing to the first period, we can easily see, all the critical section have been migrated to processor 2 where is different to the normal execution. Thus, the DPCP has been verified on both the PCP-like performance aspect and the migration aspect.

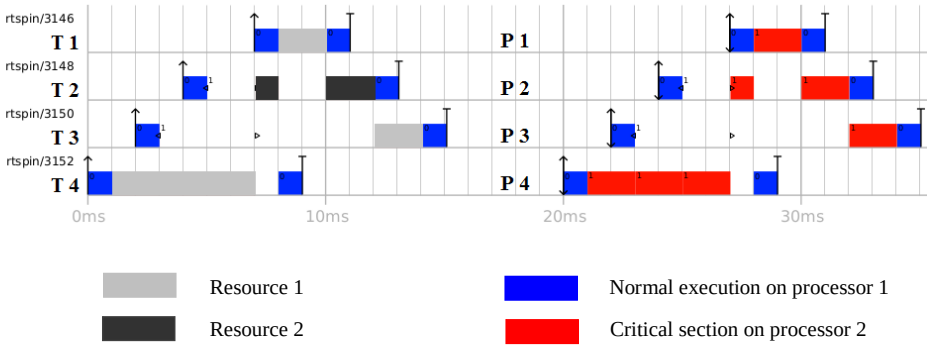

FIGURE 3.6: DPCP's behavior on two processors for two resources.

Until now, all those PCP based protocols implemented on the $LITMUS^{RT}$ have been verification, and we are familiar with the test bed as well.

Chapter 4

Implementation for New Protocols

After the verification of the existing protocols which have been implemented on the $LITMUS^{RT}$, we need to implement two resource synchronization protocols which haven't been implemented on the test-bed. That because the evaluation for these existing protocols have been made by others. That is meaningless for us to repeat other's job. Thus we need to implement several new protocols which we are interested in on the $LITMUS^{RT}$ in order to construct a new evaluation. There are two protocols that we want implement in this thesis: *Distributed Non-Preemptive* (D-NP) protocol and *Multiprocessor Resource Sharing Protocol* (MrsP). The theories of them have been discussed in Chapter 2, we only focus on the implementation issues on this chapter.

4.1 Introduction for the Process Scheduling in the kernel

Before we beginning the real implementation, we need to get familiar with procedure how the process scheduling works in the kernel. We start with the original Linux kernel. The *state* field of the process descriptor describes the current condition of the process (see Figure 4.1). Each process on the system is in exactly one of five different states. This value is represented by one of five flags: [26]

1. *TASK_RUNNING*: The process is runnable, it is either currently running or on a runqueue waiting to run.

2. *TASK_INTERRUPTIBLE*: The process is sleeping (that is, it is blocked), waiting for some condition to exist.

3. *TASK_UNINTERRUPTIBLE*: This state is identical to TASK_INTERRUPTIBLE except that it does not wake up and become runnable if it receives a signal.

4. *__TASK_TRACED*: The process is being traced by another process, such as a debugger, via *ptrace*.

5. *__TASK_STOPPED*: Process execution has stopped; the task is not running nor is it eligible to run.

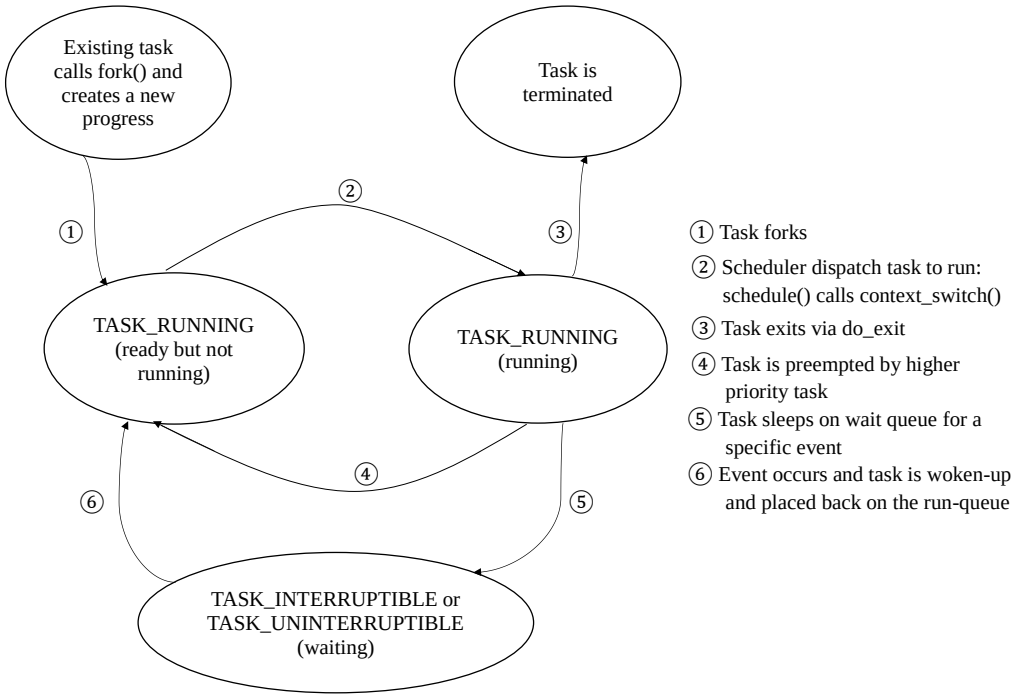

FIGURE 4.1: Flow chart of process states in the original Linux kernel.

Figure 4.1 has explained the process states of original Linux kernel clearly. Due to the resources sharing, the Linux kernel provides the semaphore to do the resource synchronization just like we have mentioned in Chapter 1, all the tasks want to access a resource which is locked by other task under the suspension-based locking will be added to the

associating wait queue. So now we come to the issue how a task sleeps and is woken up. The Figure 4.2 depicts the relationship between each scheduler state. [26].

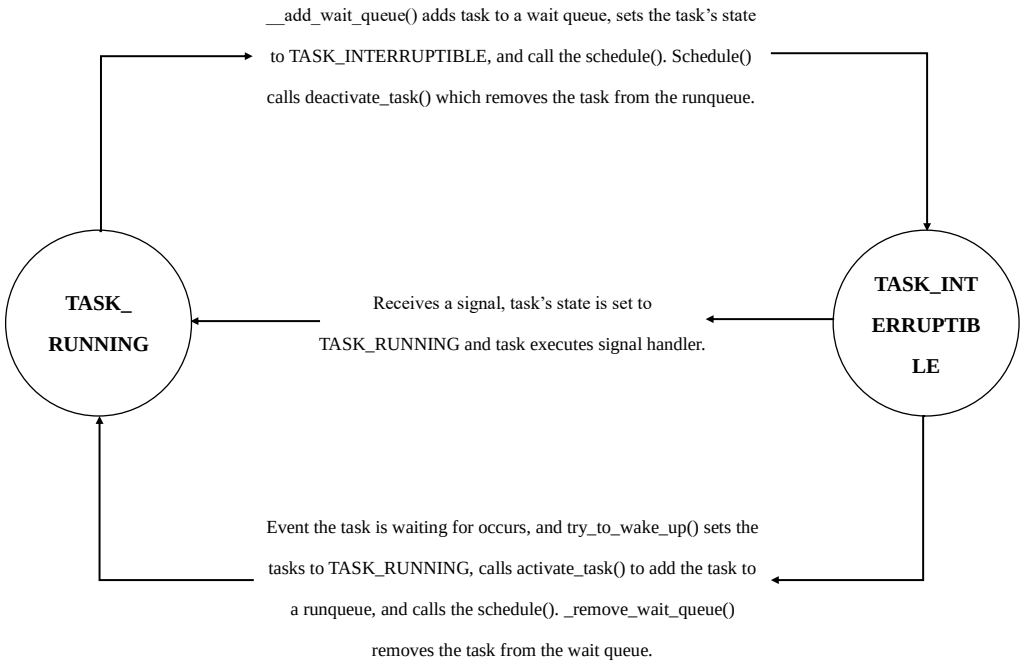

FIGURE 4.2: Sleep and waking up of a task.

After introducing the procedure how the preemptible Linux kernel works, now it comes to the real-time system. Actually, the process states are similar to to the Linux kernel, but it just simplifies the diagram and makes it clear for the reader to understand all the states of one task in the real-time kernel. In this section, we describe the structure of a small real-time kernel, called DICK (*DIdactic C Kernel*)[1]. The essential functions of the kernel have been shown in Figure 4.3. The possible states in which a task (or process) can be during its execution as well as how a transition from one state to another can be performed have been described. In any kernel that supports the execution of concurrent activities, there are at least three states in which a task can enter: *RUN*, *READY* and *WAIT*. But in our $LITMUS^{RT}$ kernel, the periodic tasks as well as synchronized tasks releasing are supported. Thus the states *IDLE* and *SLEEP* should be mentioned. Also, the state named *ZOMBIE* may be entered by a task. Those six states have been described in detail as follows: [1]

1. *RUN*: A task enters this stats as it starts execution on the processor.

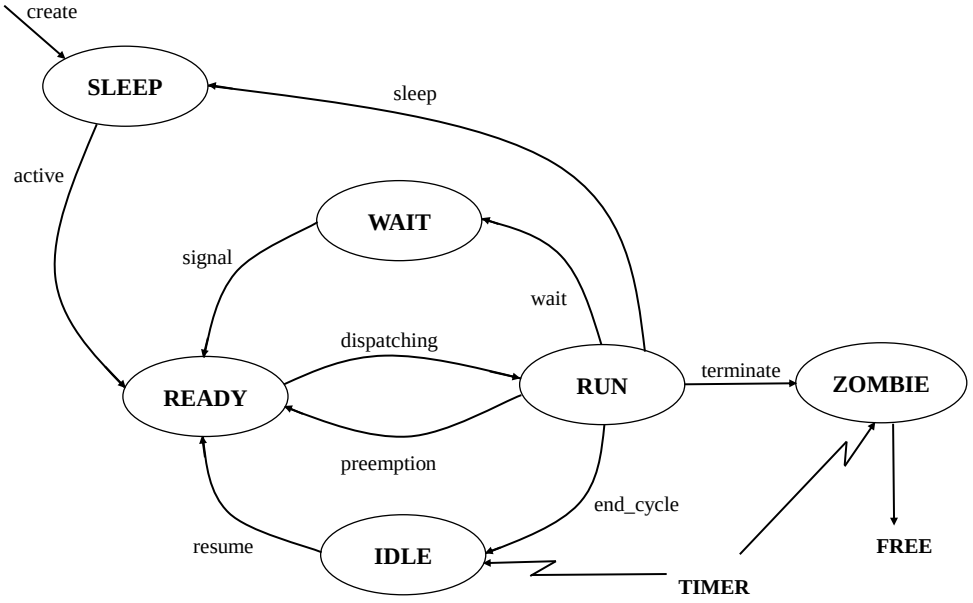

FIGURE 4.3: State transition diagram in DICK.

2. *READY*: This is the state of those tasks that are ready to execute but cannot be executed because the processor is assigned to another task (due to the priority is lower than the task which is scheduled on). All tasks that are in this condition are maintained in a queue, called the *ready queue* (ordered by the priority mostly).

3. *WAIT*: A task enters this state when it executes a *synchronization* primitive to wait for an event. When using semaphores, this operation is a *wait* primitive on a locked semaphore. In this case, the task is inserted in a queue associated with the semaphore (ordered either by priority or FIFO). The task at the head of this queue is resumed once the semaphore is unlocked by another task that executed a *signal* on that semaphore. When a task is resumed, it is inserted in the ready queue at first.

4. *IDLE*: A periodic job enters this state when it completes its execution and has to wait for the beginning of the next period. In order to be awakened by the timer, a periodic job must notify the end of its cycle by executing a specific system call *end_cycle*, which puts the job in the *IDLE* state and assigns the processor to another ready task. At the right time, each periodic job in the *IDLE* state will be

awakened by the kernel and inserted in the ready queue. This operation is carried out by a routine activated by a timer.

5. *SLEEP*: This state is introduced to handle the activation and suspension of a-periodic tasks. Meanwhile, task creation and activation are separated in DICK. The creation primitive *create* allocates and initializes all data structures needed by the kernel to handle the task. However, the task is not inserted in the ready queue, but it is left in the *SLEEP* state, until an explicit activation is performed.

6. *ZOMBIE*: A task can be said in that state when it is in the interval of time between the abort operation and the end of its period, since it does not exist in the system, but it continues to occupy processor bandwidth.

The reason why we introduce the basic knowledge about the tasks' states and the transition between states is that the essential of the resource synchronization protocol is to control the tasks' transition followed by some specific rules. The implementation of the protocol is to build such rules.

4.2 Debug

As we all known, any improper operation in the kernel may cause the whole system down. Thus we need to build one appropriate develop environment as well as debug environment to make the our work more efficient. There are two approaches that can be used to debug for the $LITMUS^{RT}$ kernel. For one thing is the trace and log tool provided by the developer of the $LITMUS^{RT}$. For another, there is a common kernel debug tool named *Kernel GUN DeBugger* (KGDB) [27].

4.2.1 Tracing and Logging

The debug tracing mechanism is available in $LITMUS^{RT}$, that exposes one device file named *litmus_log*. This trace contains text message created by the *TRACE()* that convey information useful for debugging. There is one global *litmus_log* buffer for the whole system. The *litmus_log* facility is essentially a replacement for *printk()*, which cannot be invoked from scheduling code without risking deadlock. Debug tracing must

be enabled at compile time. Note that debug tracing creates significant overhead because string formatting takes place. The *litmus_log* buffer can be read by simply opening the file and reading its contents using the following command:

```
cat /dev/litmus/log > my_debug_log
```

Kill the cat process to stop recording debug messages. Note that messages may appear in an order that differs from the sequence of events at runtime. If order is important (for example when debugging race conditions), then recorded messages can be sorted offline with the help of the sequence number at the start of each recorded message. All the information is available on [28].

```
sort -n my_debug_log > my_sorted_debug_log
```

However, the 'tracing and logging' can only work for the condition that the kernel is runnable. If there is something wrong appears in the very beginning, we even don't have the chance to use the tool. Meanwhile, that is impossible for us to add the trace command for every kernel operations which can make the trace file too large to read, large lateness of the whole system as well. We believe such tool is useful when we debug for the problems on the inner logic as well as the operation sequences.

4.2.2 KGDB

We need one more powerful tool to debug from the very beginning of the implementation. KGDB is a debugger for the Linux kernel. It requires two machines that are connected via a serial connection. The serial connection can be built via the UDP/IP network protocol. The target machine (the one being debugged) runs the patched kernel and the other (host) machine runs gdb. The gdb remote protocol is used between the two machines (build the connection). Before we can use the KGDB, we need to build the environment for these two machine. We use *QEMU* to run a *Kernel-based Virtual Machine* (KVM) which installed the $LITMUS^{RT}$ kernel using the following code:

```
#!/bin/bash
qemu-system-x86_64 \
       -enable-kvm \
       -cpu host \
```

```
-smp 2 \
-hda litmus.qcow \
-m 1024 \
-name "ubuntu-qemu-cjk" \
-gdb tcp::12345 \
-net nic \
-net user,hostfwd=::2222-:22,smb=/home/litmus/shared
```

We use the disk image based on *xubuntu* which has the $LITMUS^{RT}$ kernel default that is available on: http://www.litmus-rt.org/tutor16/litmus-2016.1.qcow.gz. Now, our host (Ubuntu system) and guest (xubuntu with the $LITMUS^{RT}$ kernel) can be connected in serial via tcp network protocol on port: 12345.

Further more, besides building the virtual machine, we need to build another powerful communication between the *host* and *guest* to transport the files. We use the *Server Message Block* (SMB) to build that transport communication. The KGDB on the *host* need to read the copy of one file named *vmlinux* located on the *guest*, where contains the Linux kernel in one of the object file formats. Such file will be rewritten every time after recompiling of the kernel. Thus, after rewriting the kernel file on the *guest* and compiling, we need to update the *vmlinux* file for the *host*. Such file is almost $200mb$, we should find an efficient way to update the file between *host* and *guest*. That is the reason why we need the SMB to communicate between the *host* and *guest*. After install the SMB, we need to configure some file so that SMB can work as expect. The processes are as follows:

```
// come to the following path in host to add new configuration:
// etc/samba/smb.conf
[qume]
   comment = xubuntu space with litmus-rt
   path = /home/litmus/shared
   read only = no
   available = yes
   force user = litmus
   public = yes
   writable = yes
   browseable = yes
   create mask = 0644
```

```
   directory mask = 2777

// set the folder in that path as a 'shared' folder

// do the following command to restart the server
/sbin# service smbd restart

// now this folder is shared between the host and guest,
// you can access this folder in the following path:
smb://10.0.2.2/shared/
```

All the preparation has been finished at present, we can come to the real implementation in the next section.

4.3 Implementation of Distributed Non-Preemptive Protocol

Similar to the DPCP, under the *Distributed Non-Preemptive* (DNP) scheduling, the resources are assigned to fixed processors. Each task want access to the specific resource, should migrate to the processor where the resource is assigned to. However, the behavior in the critical section is different to the DPCP. Once a task has locked the resource starts to execute the critical section, it cannot be preempted by any other tasks until it finishes the section even by those tasks which have the higher priorities. Meanwhile, there is another important attribute that, no normal execution will be allowed to execute when there is at least one critical section is assigned to that processor and waiting for other task to release the processor. In the other word, the critical section always has the higher priority than the normal execution no matter what the base priorities of them are.

Thanks to the $LITMUS^{RT}$, the DPCP has been implemented inside the P-FP plug-in. Because the migrations for the the critical sections are the same between DPCP and DNP protocol. Thus, the only issue that we need consider about is how to implement the non-preemptive part. As far as we are concerned, there are two options to implement the non-preemptive execution. For one thing, the non-preemptively execution is similar to the virtual spin lock of MPCP. The processor is occupied by one task, all the tasks come

later than that task will be added into a *wait queue* which is associated to the processor. That means each processor has their own *wait queue* used to realize the non-preemptive virtual spin lock. Fortunately, the $LITMUS^{RT}$ provides such non-preemptive virtual spin lock as well. In that sense, our work is to combine both two attributes into a new protocol. For another, we can raise the task's priority to 1, which is the highest priority among all the tasks, when the task begins its critical section. After the rise, no task can preempt it since it gets the highest priority. Further more, the user space library *liblitmus* provides a pair of flags named '*enter_np()*' and '*exit_np()*' to bound the non-preemptive execution fragment, but that choice can only handle the non-preemptive attribute rather than the relationship between normal execution and critical section. However, if we choose the option that raise the priority may cause some problem if the race condition occurs, when a task is enter to the critical section and the priority is raised to the highest priority, at the same time, another another task wants execute another critical section which has the highest priority is migrated to the same processor. Under that situation, the scheduling decision may be different for each period. Thus, we choose the option which applies the non-preemptive virtual spin lock to implement the non-preemptive execution.

The flowchart of one task is migrated to access the resource on the specific processor is shown in Figure 4.4. The first step of the task when it wants to enter the critical section is *boost* the priority of itself. Which can ensure the task cannot be preempted by the normal execution task which has the higher priority. The *spin_task* is a task structure acts as a container for the task which is occupying the processor right now. Before occupying the processor, each task need to check whether the container is empty or not. If the container is empty, then write itself task structure to the container, and starts the execution. If the container is not empty, that means there is one task is occupying the processor. To guarantee the non-preemptive execution, the new task will be added into a wait queue which is ordered by task's priority. When one task finished its execution, it will clean the container, and send a *signal* to the wait queue. That *signal* means the task which occupies the processor has finished its execution, after receiving that signal, the task which has the highest priority in the wait queue will be woken up and come to the conditional judgment again as a new coming task. Such mechanism can ensure that there is only one task can occupy the task to execute its critical section without preemption. Other tasks come later will be added into the wait queue before they can

preempt the task under executing.

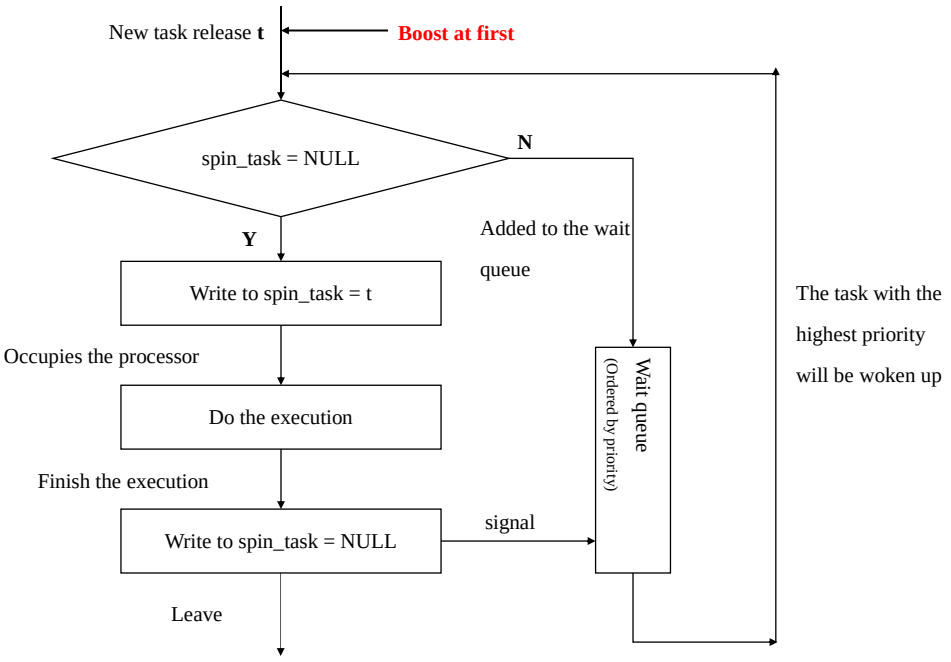

FIGURE 4.4: The flowchart when a task enters its critical section under DNP protocol.

After implementing and debugging, we need to verify such new protocol by running simulated task-set. The result has been shown in Figure 4.5. Task T_1 has the highest priority among all the tasks, and it is the only one task that only has the normal execution and assigned to the process 2. All the other tasks contain both normal execution and critical section, and normal executions are only assigned to processor 1, critical sections are only assigned to processor 2. From the Figure 4.5 we can see, all the critical sections are executed non-preemptively and the normal execution can not be executed before the critical section even though it has the higher priority. But, we still notice that there are several gaps on the task T_5's critical section, and each gap occurs when a new critical section is migrated to that processor. That means, each time when a new task is released to the processor, the processor will allocate some time to handle that new coming task, to execute the commands in the kernel. After that, continues to execute the previous one. Although such *suspend* and *resume* operations may not cause large overhead, we still try to reduce it. So, we add the *enter_np()* and *exit_np()* for each critical section, which can guarantee all the critical sections within these two labels are executed non-preemptively that means there is no gaps due to the release of new tasks. Remarkable, that solution doesn't eliminate these gaps, but just delays all

the operations for the new coming tasks until the current task is finished. The result has been shown in Figure 4.6. Such solution to collect all the operation together, can reduce the overhead of the context switch. That is why we choose this plan as the final implementation to do the evaluation in Chapter 5.

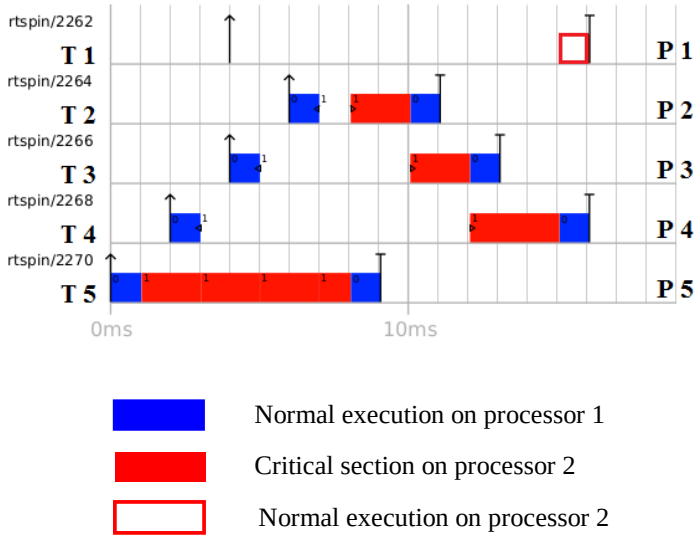

FIGURE 4.5: The result of the first attempt of the DNP protocol.

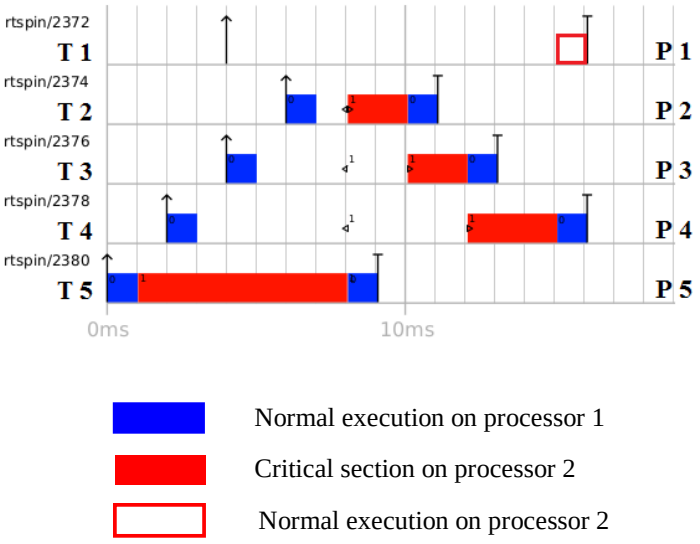

FIGURE 4.6: The result of the second attempt of the DNP protocol.

4.4 Implementation of Multiprocessor Resource Sharing Protocol

We also need to implement the second protocol which is more complex and different to the existing protocol on the $LITMUS^{RT}$. Just like we have discussed in Chapter 2. There are three main features of the MrsP.

1. FIFO queue for the requests of the resources over different processors.
2. Tasks are spinning at local ceiling priority on the processor only when they are waiting for some resources.
3. Help mechanism between the semaphore's owner and waiting tasks.

All these three attributes will be discussed in detail in this section. Meanwhile, implementations for all of these three attributes contribute for the complete implementation of MrsP.

4.4.1 FIFO Queue

FIFO queue for the resource requests is the most easy part among all these three features. Although here are two protocols which have been implemented on the P-FP in the $LITMUS^{RT}$ used the FIFO queue to organize the resource access requests. We cannot use such structure directly for all the protocols implemented on P-FP currently are suspension-based. However, our MrsP's implementation is based on the spin lock. Thus, to implement such FIFO queue, we add a task structure named *next* for each task's parameter that points to the next task which requests the same resource. After one task finishing its critical section, the semaphore will be allocated to the next task which the parameter *next* points to. This *next* structure will be useful when we implement the help mechanism as well. Meanwhile such FIFO queue for one resource only works for the requests from the different processors, and those requests from the same processor will be managed in a priority based queue.

The way to implement the FIFO spin lock named $ticket - basedspinlock$. Briefly, the *ticketlock* [29] consists of two words, a *ticket* variable and a *grant* variable. Arriving

threads atomically $fetch-and-increment$ the ticket and then spin, waiting for grant variable to match the value returned by the $fetch-and-increment$ primitive. At that point the thread is said to own the lock and may safely enter the critical section. The Yan Solihin's pseudo code example [30] has shown as follows:

```
ticketLock_init(int *next_ticket, int *now_serving)
{
   *now_serving = *next_ticket = 0;
}

ticketLock_acquire(int *next_ticket, int *now_serving)
{
   my_ticket = fetch_and_inc(next_ticket);
   while(1) {
      *now_serving == my_ticket;
      break;
   }
}

ticketLock_release(int *now_serving)
{
   *now_serving++;
}
```

The most advantage of the ticket lock is that it is fair which can ensure the FIFO queue comparing to other spin locks. Also, such spin lock is suitable for our implementation for MrsP's FIFO queue.

4.4.2 Spinning at Local Ceiling

To achieve the object that we have mentioned in the last subsection (there are at most m tasks are waiting in the FIFO queue), we need to raise the task's priority to the resource ceiling in the current processor where the task can request the resource. We have explained in the Chapter 2, the resource ceiling for each processor is equals to the highest priority of the task among all the tasks in that processor which will access that resource. That means once a task gets the semaphore and executes the critical section,

it cannot be preempted by other tasks which request the same resource. Such resource ceilings are set in advance by the users.

To explain how it works, we take a simple example combining the FIFO queue and resource ceiling. We use $T_{p,j}$ to indicate the tasks, p indicates the processor where the task is allocated to, j indicates the task's number as well as the priority of each task (the bigger the number is the lower the priority it has). For example, there are 5 tasks and 2 processors $T_{1,1}$, $T_{1,3}$, $T_{1,5}$, $T_{2,2}$, $T_{2,4}$, so the resource ceiling on processor p_1 is 1 on p_2 is 2. On t_0, $T_{1,5}$ is released, it raises its priority to 1 on processor 1, and get the semaphore. On t_1, $T_{2,4}$ is released, it raises its priority to 2, and is spinning on processor 2. On t_2, all the other tasks are released, however all these three will be added to the ready queue corresponding to processor where they are allocated to without getting any ticket. That because one task can get the ticket, it should be allowed to execute some normal execution for the kernel commands on that processor. Unfortunately, due to the ceiling raising, no tasks can execute that commands in our example. On t_3, $T_{1,5}$ releases the resource and lowers its priority, so $T_{1,1}$ is woken up and tries to lock the resource, but it comes latter than $T_{2,4}$, now $T_{2,4}$ gets the semaphore and $T_{1,1}$ is spinning on processor 1. On t_4, $T_{2,4}$ releases the semaphore, $T_{1,1}$ gets the semaphore, $T_{2,2}$ is woken up and is spinning on processor 2. On t_5, $T_{1,1}$ releases the semaphore, $T_{2,2}$ gets the semaphore and $T_{1,3}$ is spinning on processor 1. On t_6, $T_{2,2}$ releases the semaphore, and $T_{1,3}$ gets the semaphore. All the ticket information has shown in Table 4.1, Only when *sem.ticket* is equal to the task's ticket, the task can get the semaphore and execute the critical section. From that simple example, we can notice that, only m tasks can get the tickets

	$T_{1,1}$	$T_{1,3}$	$T_{1,5}$	$T_{2,2}$	$T_{2,4}$	sem.ticket
t_0	NULL	NULL	0	NULL	NULL	0
t_1	NULL	NULL	0	NULL	1	0
t_2	NULL	NULL	0	NULL	1	0
t_3	2	NULL	NULL	NULL	1	1
t_4	2	NULL	NULL	3	NULL	2
t_5	NULL	4	NULL	3	NULL	3
t_6	NULL	4	NULL	NULL	NULL	4

TABLE 4.1: The ticket table for FIFO spin lock.

from the semaphore structure where m is the number of all the processors that have the

tasks will access to that semaphore. Thus, the task with the highest priority comes later will be blocked at most m times of the critical sections caused by the tasks with lower priorities (but request the resource earlier).

4.4.3 Help Mechanism

Actually, the help mechanism is the most challenging part. The flowchart of the help mechanism has been shown in Figure 4.7. And there are several points that need to be underlined. First, only those tasks are spinning at their own processors can help the semaphore's owner which they are waiting for. Second, when the semaphore's owner is helped by another task, it should migrate to the processor belonged to this task who does the help. Third, the task should migrate back to the original processor if it is helped by others. Last but not least, when the task which is preempted while waiting becomes the next semaphore owner, it also deserves the help by others if possible.

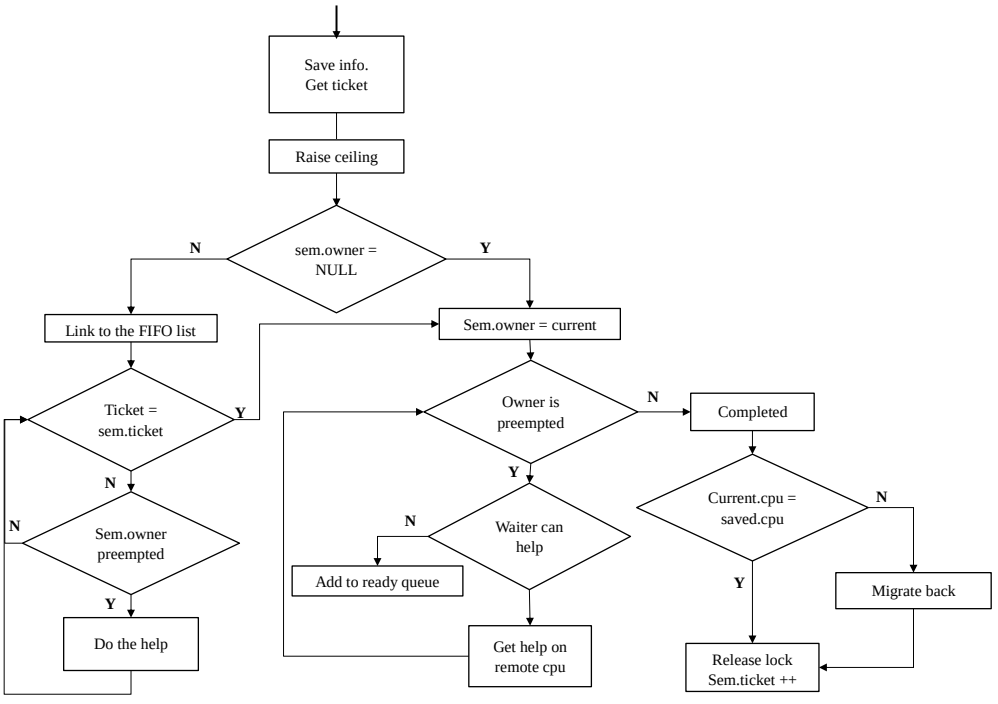

FIGURE 4.7: The flowchart for the help mechanism.

From the Figure 4.7, we can see, before entering the lock progress, all the information of the task will be stored. So, they can come back to the original state after the execution

for critical section. We add a variable named *sem_owner_preempted* in the semaphore to indicate the semaphore owner's current state (running or preempted). All the tasks waiting for the semaphore will read this variable to decide whether the owner needs help. Meanwhile, we have another variable named *preempted_while_waiting* for each task to indicate the task's state while it is waiting for the owner releases the semaphore, in order to make some scheduling decision after getting the semaphore.

Based on the P-FP plugin, we need to modify three main parts to support the help mechanism. Firstly, we need to modify the function named *pfp_schedule()* which makes the scheduling decision. The original function is used to find the proper *next* task, the previous task will be added to the ready queue. But in our implementation, if the previous task is the owner of one semaphore, some specific steps will be taken. If the function notices that the previous task is the semaphore's owner, the variable *cpu* in this task's *task_structure* will be set to -511. Once the system detects the *cpu* in the task's *task_structure* is different to the current processor, the function named *finish_switch ()* will be called. In this function, the semaphore's owner will try to find the help task via task's parameter *next* that we have discuss in the subsection *FIFO Queue*. If the owner can find one helper, it will be migrated to the helper's processor. The owner's priority will be set as the helper's priority minus one, in order to preempt the helper which is spinning on its processor. If the owner cannot find any helper, it will be requeued to the *saved_cpu*'s ready queue and the variable *sem_owner_preempted* will be set to 1. Also, in the function *pfp_schedule()*, if the previous task is waiting for the semaphore and added to the ready queue, the variable *preempted_while_waiting* will be set to 1. Secondly, the semaphore lock function will be modified. In the spinning while loop, the variable *sem_owner_preempted* will be read and check whether the owner is preempted by others at present. If the owner is preempted, the spinning task can let the owner migrate to the current processor and modify the priority to take over the current processor. Thirdly, we will modify the semaphore unlock. Just like what we have mentioned, if the semaphore's next owner is preempted while waiting, when the previous owner unlock the semaphore, it should set the *sem_owner_preempted* to 1. In that way, the spinning task can notice the semaphore owner is preempted, and needs help. The new semaphore owner will get help if possible once it gets the semaphore.

4.4.4 verification for the implementation

After implementation all of those three attributes, we need to verify all of them as well to check whether the MrsP has been implemented correctly as expected. In order to analyze the results clearly and easily, we suppose each task can only access to one resource at most.

Firstly, the FIFO queue will be verified. The result has been shown in Figure 4.8. From the figure, we can see, task T_2 with the higher priority comes later than T_3 with the lower priority. Both of them are spinning on their own processor, because the resource is occupied by the task T_1. After T_1 releasing the resource, T_3 gets the resource rather than T_2. Although T_2 has the higher priority than T_3, but T_2 comes latter, thus T_3 can get the resource at first. That is the FIFO queue.

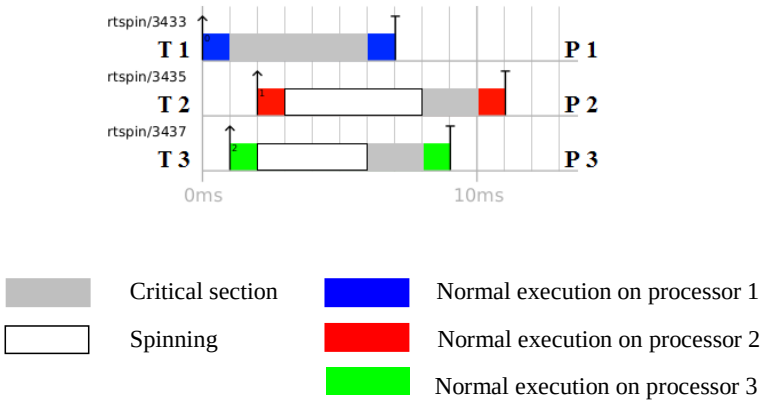

FIGURE 4.8: The verification for the FIFO queue.

Secondly, the attribute 'spinning at the local ceiling' will be verified. We only have one resource on this experiment. The resource ceiling on processor 2 is set as 2. From the Figure 4.9, we can see, when the task T_5 is released with the original priority, there is no other tasks here, so T_5 occupies the processor. But the resource is occupied by task T_1, thus $T5$ is spinning at the processor 2. To verify whether T_5 is spinning at the local ceiling, we release a new task T_4 with the priority higher than T_5 but lower than the resource ceiling. We notice that, T_4 cannot preempt the T_5, for T_5 has the higher

priority than T_4 currently. Thus, the task will spin at the local ceiling when the resource is occupied by other task on other processor.

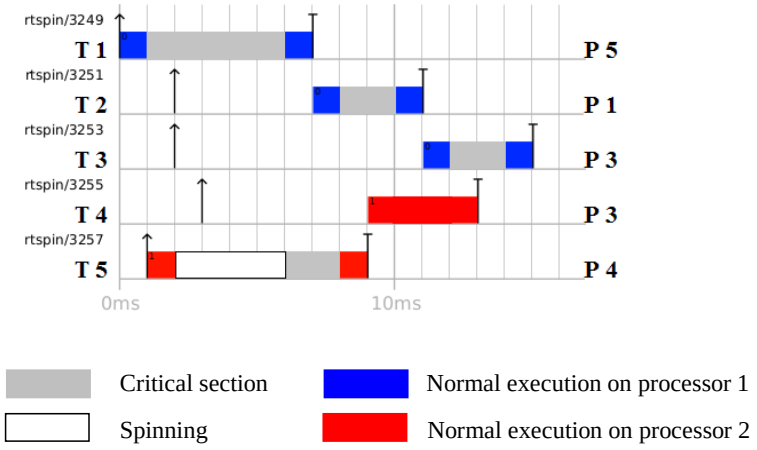

FIGURE 4.9: The verification for the spinning.

Thirdly, we need to verify the behavior of the task with the highest priority but comes later.The result has been shown in Figure 4.10. From the figure we can see, the task T_2 has the highest priority. When T_2 is released on process 1, the T_1 has locked the resource, and the priority of T_1 has been raised to the resource ceiling which is equal to the priority of T_2. Thus, T_2 cannot preempt T_1 to execute its normal execution, and is put into the ready queue directly. Before the T_2 can try to lock the resource and get the ticket, there is another task T_4 on processor 2 has get the next ticket in advance. When T_1 finishes its critical section, the priority will come back to the original one, so the T_2 can preempt it to execute the normal execution. But the resource is occupied by the T_4 on the processor 2, T_2 is spinning and waiting for the resource. Thus, the highest priority task can be blocked by m times (m equals to the total number of processors where the tasks can access to the same resource).

Lastly, the help mechanism need to be verified. Actually, the help operation will be taken under two situation. For one thing, there is a waiting task spinning on its processor, when the semaphore owner is preempted by others. For another, one task is released and trying to get the semaphore and spinning on its processor after the semaphore owner has been preempted. In the first situation, the semaphore will migrate to the

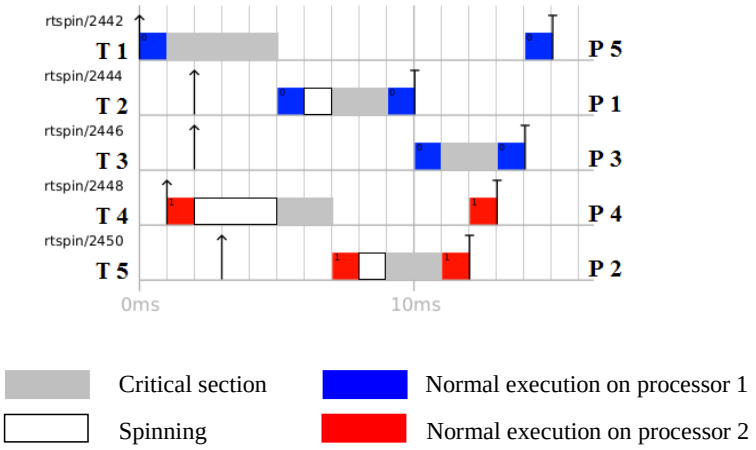

FIGURE 4.10: The direct blocking for the task with the highest priority.

helper's processor by itself using the function *finish_switch*. In the second situation, the semaphore owner has been en-queued to its ready queue, that means it cannot migrate to the helper's processor by itself. Thus, in that situation, the spinning helper task will get the run-queue lock of the semaphore owner, and help the owner to migrate to the proper processor. From the Figure 4.11, we can see, when task T_1 is preempted by the task T_3 with the higher priority on processor 1, there is a task T_2 is spinning for waiting the resource on processor 2. Then, T_1 can migrate to the processor 2 and continue execute the critical section directly. However, on the Figure 4.12, there is no task spinning on other processor when the owner of the semaphore T_1 is preempted. Luckily, before the blocking is ended, there is one task T_2 on processor 2 also need to the same resource as T_1. When T_2 is trying to lock the resource, it notices the resource is occupied by T_1, but T_1 is not under execution. So T_2 helps the T_1 to migrate to the processor 2 for reducing the waiting time of T_2 its own.

After all the verification experiments above, we believe that the protocol MrsP has been implemented correctly. However we have no idea about how is the performance under huge tasks comparing to other protocols. We will discuss the performance of several protocols that we are interested in in the next chapter.

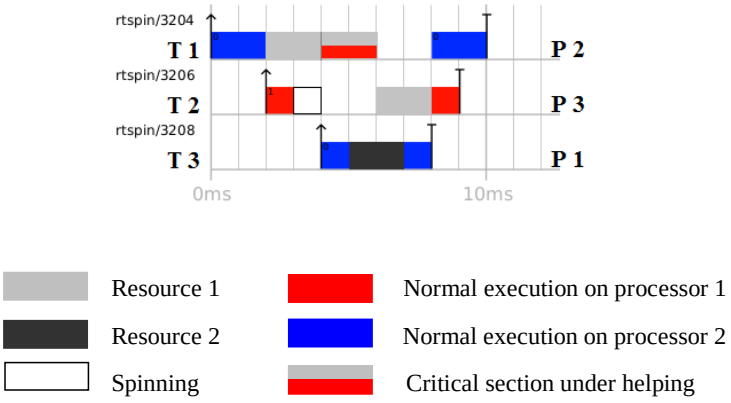

FIGURE 4.11: The verification for the help mechanism (a).

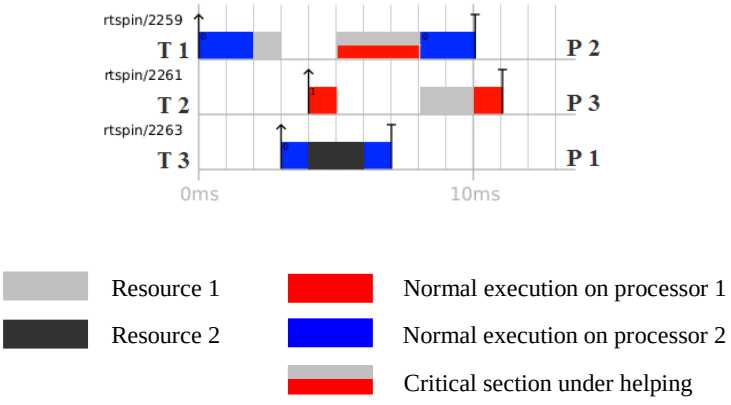

FIGURE 4.12: The verification for the help mechanism (b).

Chapter 5

Evaluation

The hardware platform used in our experiment is a cache-coherent SMP consisting of four 64-bit AMD Phenom(TM) processors running at 2.3 GHz, with 256K L1 instruction and data caches, and a 512K L2 cache per processor, and 3.5 GB of main memory. Before we can do the evaluation among all the candidate protocols, we should construct several reasonable task set at first. The performance of those protocols should be reflected by the ability to handle the tasks. Different choice of task-sets may cause different result. Meanwhile, there are two main indicators to show the performance of protocols in practice: overheads and response time. In the following sections, we will discuss these three parts in detail.

5.1 Task-set construction and verification

Of course, under our partitioned-based scheduling algorithm, the performance of protocols dependent on the task-set and the partitioned. We need task sets that are neither too simple nor too complicated, so that our problem models can highlight the algorithm benefits and won't be impossible to analyze.[31]

5.1.1 Construction

In our thesis, we apply the task-set from [32]. Such task-set is modeled from the real world real-time system, which can make our evaluation more reasonable and acceptable.

The *average execution times* (ACET), *best case execution times* (BCET) and *worst case execution times* (WCET) can be described in the Table 5.1. After acquiring the execution time, we need to find the some kind of distribution to generate the complete task set. Although, the execution times showing on the table may not fulfill any kind of classic distribution, we just assume that we can use *Gaussian distribution* to generate our task set. Actually, the *Gaussian distribution* only need one variable: the ACET, thus the BCET and WCET act as the lower bound and upper bound for the distribution. After that, we need come to the sharing resources allocation. The whole resource part will occupy the 5%-15% of the actual execution time for each task. We start from only one resource available, that means all the task will access to the unique resource.

Period	Average Execution Time in us		
	Min.	Avg.	Max.
1 ms	0,34	5,00	30,11
2 ms	0,32	4,20	40,69
5 ms	0,36	11,04	83,38
10 ms	0,21	10,09	309,87
20 ms	0,25	8,74	291,42
50 ms	0,29	17,56	92,98
100 ms	0,21	10,53	420,43
200 ms	0,22	2,56	21,95
1000 ms	0,37	0,43	0,46

TABLE 5.1: The runnable average execution times.

5.1.2 Verification

After finishing all the preparation, we tried to run one task set on two processors to verify our experiment design. We select one task for each periodic with only one sharing resource. Unexpected performance of DPCP occurs, which has shown in Figure 5.1 with red box. Within some time interval, all the tasks disappear. Except the first one, all the following wired performance will repeat every 50 ms with the same shape like the Figure 5.2 has shown. Such failure period won't change according to different task sets, but the length of the failure interval will decrease along with the decrease of the number of tasks. Meanwhile, the period of the failure will be 200 ms, when we change the timer

frequency from 1000 Hz to 250 Hz. We consider such periodic failures are caused by the frequent migration with some time quantum. The reason why we make such assumption is because MPCP has no such failure problem and all the distributed-based protocols are suffered by that problem.

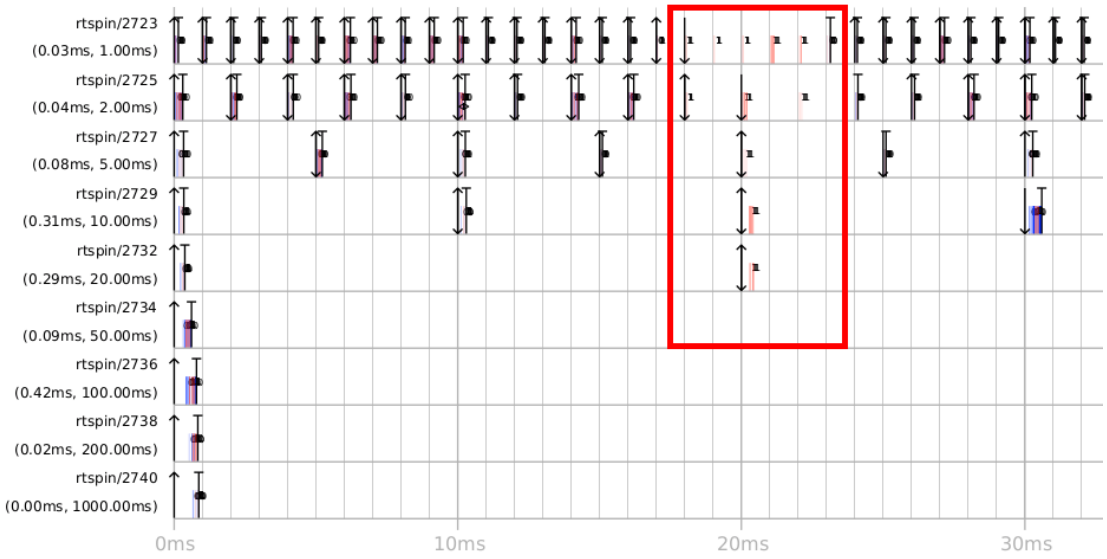

FIGURE 5.1: The fist interval of the failure under DPCP.

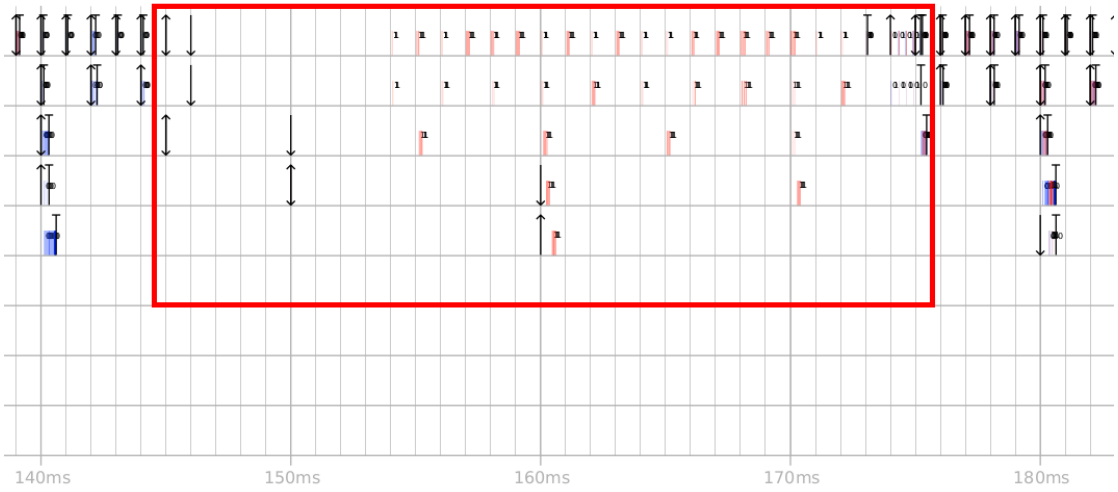

FIGURE 5.2: The rest of the failure's performance under DPCP.

Finally, we find out the reason why such periodic failure occurs is rely on the limited size of the tracing buffer. To decrease the overhead which is caused by the tracing tool, the $LITMUS^{RT}$ uses the buffer to store the tracing events temporarily. These buffers

are statically allocated as per-CPU data. Too large buffers can cause issues with the per-cpu allocator (and waste memory). Too small buffers can cause scheduling events to be lost which has shown above. After increase the buffer size in the configuration file, such periodic problem can be eliminated.

However, we still face to another problem. In the task set provided previous subsection, the utilization for each task is too small, that means we need a huge number of tasks to achieve the high utilization. However, if there are too much tasks, we still will suffer from the periodic failure, although we have increase the buffer size. We need make a compromise between the buffer size and tasks number. To solve the problem caused by the conflict, we have no choice but to increase the WCET for each task which can raise the utilization of each task. Meanwhile, considering to the limited size of the hard disk, it is impossible for us to run the task set for a long time(over one hour). The files generated by the tracing tool are very big. In order to reach the worst case as soon as possible, we increase the ACET to close to the WCET. After the modification, our task sets are shown in the Table 5.2.

Period	Average Execution Time in us		
	Min.	Avg.	Max.
1 ms	0,34	58,00	60,11
2 ms	0,32	178,20	180,69
5 ms	0,36	393,04	420,38
10 ms	0,21	830,09	860,87
20 ms	0,25	1728,74	1741,42
50 ms	0,29	1752,56	1769,98
100 ms	0,21	1775,53	1784,43
200 ms	0,22	1785,56	1790,95
1000 ms	0,37	1760,43	1770,46

TABLE 5.2: The task-sets used for evaluation.

We will build several experiments, the number of different tasks chosen for those experiments are different to reach different processor utilization. Meanwhile, when we construct the real experiments, we will vary the WCET and ACET a little bit for those tasks have the same period. Also, we have two kind of sharing resources assumptions: (i) There is only one resource available, each task will request for that resource for 15% of

it's actual execution time. (ii) There are 5 resources available, each task will request 3 of them in order, the total critical section still contains 15% of the whole actual execution time.

5.2 Overheads evaluation

Protocols are designed upon the assumption that run-time overheads are negligible. However, in practice, this is very unlike to be the case, because the kernel will spend several cpu cycles to do the operations that are introduced by the protocol (*e.g.*, en-queue one task to the ready queue, migrate one task from one processor to another and so on). Some of these overheads may waste too much time that can impact the performance of some protocols. Thus, we need to figure out how much the overhead can affect the performance of those protocols under $LITMUS^{RT}$. At first, we need to understand what is the overhead, and how's the overhead occurs. In the beginning, the overhead is defined as those costs required to run a business, but which cannot be directly attributed to any specific business activity, product, or service in the business field. After that, we introduce the overhead to the computer since field. In computer science, overhead is any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to attain a particular goal. In our thesis, the overhead is defined as the time cost by the system between the task should be executed and the task is real executed.

We should consider the percentage of the overheads in the whole response time, otherwise, the major influence factor for the performance of each protocols will be the overheads rather than the protocols' scheduling decisions. So we need try to minimize the effects from the overheads and focus on the protocol itself. The feature trace tools inside the $LITMUS^{RT}$ provides the main overheads' tracing, the symbols can be defined as follows:

- *SEND_RESCHED*: inter processor interrupt latency.
- *SCHED*: make a scheduling decision (scheduler to find the 'next').
- *CXS*: context switch (schedule-¿next to the real current).

- *SCHED2*: perform post-context-switch clean up and management activities (deal with the schedule-¿prev staff).
- *RELEASE*: time spent to enqueue a newly released job in a ready queue.
- *RELEASE_LATENCY*: the difference between when a timer should have fired and when it actually did fire.
- *PLUGIN_SCHED*: only the time spent by the active scheduling plugin (use the plugin to find the next).

Because all the resource synchronization protocols that we are interested in are implemented based on the P-FP plugin, the most of the overheads are the same, such as *RELEASE* and *RELEASE_LATENCY*. What we care about are those overheads which may cause significant impact to the performance between different protocols. There are four main overheads that should be taken into consideration are shown as follows:

1. The migration overhead for DPCP, DNP and MrsP, which is MPCP never has.
2. The IPI overhead for MPCP (to check whether the remote resource is available).
3. The added overhead when make a scheduling decision caused by the help mechanism for MrsP.
4. The context switch overhead where DNP has its advantage over DPCP.

The overheads traced by the tool have been shown on the Table 5.3. For each overhead, we have both the worst case and average case to indicate the results, and the *SCHED(MrsP)* only indicates the more overhead introduced by the help mechanism comparing to other protocols.

	Migration	IPI(MPCP)	SCHED(MrsP)	CXS
Worst case	26 us	5.2 us	NON	9.56 us
Average case	8 us	1.3 us	NON	1.3 us

TABLE 5.3: The different overheads caused by protocols.

From the results shown on the table, we can conclude several points. Firstly, the migration overhead is the most drawback for those distributed-based protocols. Secondly,

the MPCP has the lowest overheads comparing to other protocols. Thirdly, the more scheduling decision overhead caused by the help mechanism can be neglected. Lastly, the context switch overhead may not affect the protocols' performance largely.

5.3 Response times evaluation

After knowing all the external influencing factors to the performance of our protocols, we need to construct the real experiments. As we all known, the most important indicator of one protocol's performance is the *worst case response time* in the real-time system. That because, in the hard real-time system, the miss of deadline can cause a disaster. Thus, the worst case response time plays the most important role on the setting of task's deadline. The worst case response time is the only criterion during our evaluation.

In our thesis, we are interested in these four protocols: *Multiprocessor Priority Ceiling Protocol* (MPCP), *Distributed Priority Ceiling Protocol* (DPCP), *Distributed Non-preemptive Protocol* (DNP), and *Multiprocessor Resource Sharing Protocol* (MrsP). So we will evaluate those four protocols' performance using the same specific task-set. However, it is impossible for us to analyze each task over the dozens of tasks, we only monitor the performance of the task with the highest priority.

At first, we have built 6 experiments for double processors environment and 6 experiments for quad processors environment with different processors' utilization and resource sharing assumptions. To reach different utilization for different number of feasible processors, we need select different number of tasks from the tasks warehouse showing in Table 5.2. The number of each task with different period for different utilization has been shown in Table 5.4, for each utilization, we have two kinds of sharing resources assumptions what we have mentioned above.

Besides the total utilization set and the tasks chosen, the partition can have a significant impact in the performance of those protocols. In our evaluation, we are trying to disperse the utilization to each processor. However, according to the distributing character, under DPCP and DNP, we assign all the tasks to the first three processors, and all the critical sections on the processor 4. In the DPCP and DNP, there are only three processors available for the normal partition. In the MPCP and MrsP, we have whole four processor for partitioning. So, the DPCP and DNP share the same partition, MPCP and MrsP

P.	Ut.	1 ms	2 ms	5 ms	10 ms	20 ms	50 ms	100 ms	200 ms	1000 ms
2	60%	1	1	1	1	2	2	1	1	1
	80%	1	1	1	2	3	2	3	1	1
	100%	1	1	1	2	4	4	4	3	1
4	120%	1	1	1	3	5	5	4	3	1
	160%	1	1	1	5	7	6	5	3	1
	195%	1	1	2	6	8	8	6	2	2

TABLE 5.4: The tasks chosen from the task-set.

share one partition scheme as well. Now we take the 195% utilization with 4 processors for example, which has 36 tasks. The first thing that we need to do before the partition, is to sort all these 36 tasks and give them the priority according to its period, the smaller the period is the higher the priority it has. Under this rule, task T_1 has the smallest period and highest priority, T_36 has the longest period and the lowest priority. Due to the load sharing principle, our partition can be shown in Table 5.5.

	DPCP and DNP	MPCP and MrsP
Processor 1	T_1-T_8	T_1-T_6
Processor 2	T_9-T_{16}	T_7-T_{12}
Processor 3	T_{17}-T_{36}	T_{13}-T_{18}
Processor 4	critical sections	T_{19}-T_{36}

TABLE 5.5: The partition for all the tasks.

Under our task-set, the results (in *us*) of the experiments have been shown in Table 5.3, 5.4, 5.5 and 5.6. Due to some technical problems, we do not have the results for MrsP under the four processor and five available resources case.

According to the theory that we have mentioned and the included overheads, the WCRT in practice should be larger than the WCRT in theory. But But that does not seem to be the case in our results. For one thing, under the distributed-based protocols, all the critical sections should be executed on the same processor, which makes the executions tight. Thus, it easy to reach the *worst case* within limited time. For another, the *worst cases* under the MPCP and MrsP are harder to reach in a short time. That because the executions for critical sections are allocated on different processors, which makes the

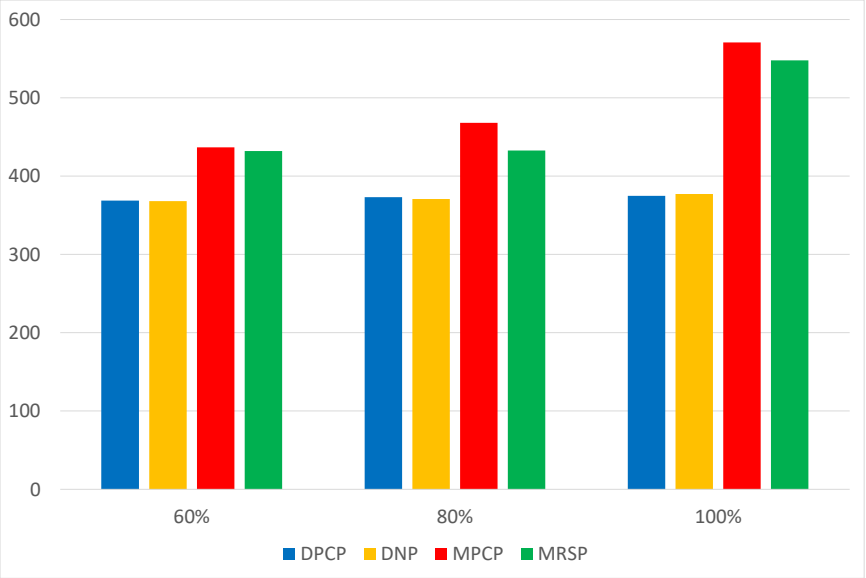

FIGURE 5.3: The result with two processors and only one resource.

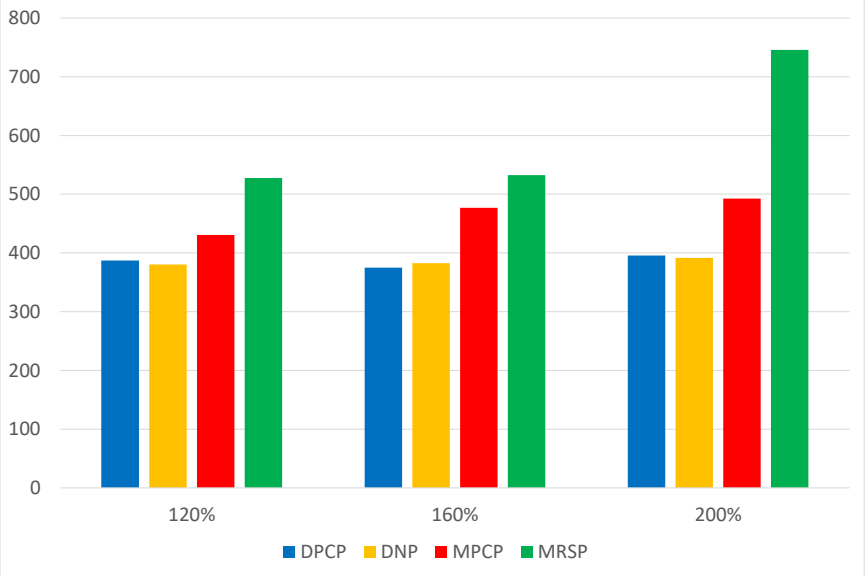

FIGURE 5.4: The result with four processors and only one resource.

executions loose. It needs tasks on all the four processors meet some critical conditions at the same time to reach the worst case response time in theory.

When the response time and the overheads has been confirmed, we can measure what is the proportion of overheads in whole response time. We choose the case (two processors and one sharing resource) which the overheads occupies the largest share of whole

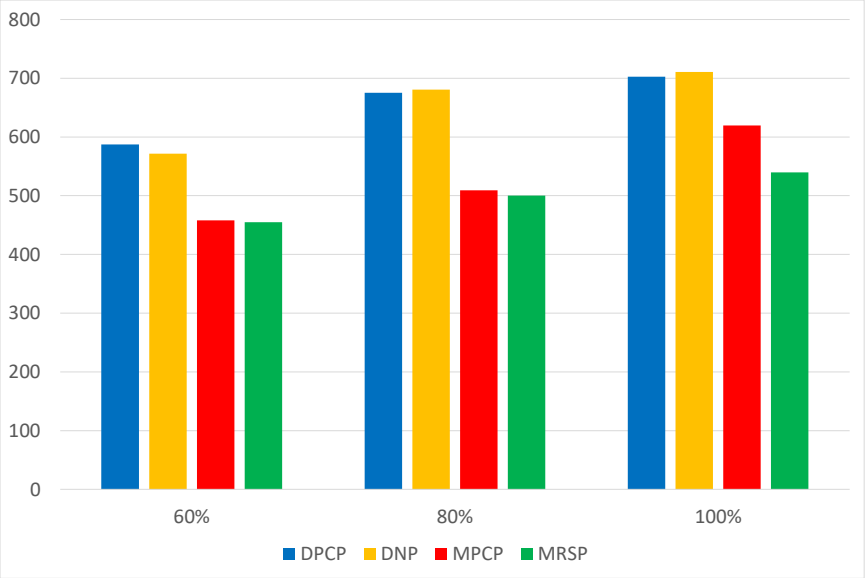

FIGURE 5.5: The result with two processors and five resource available.

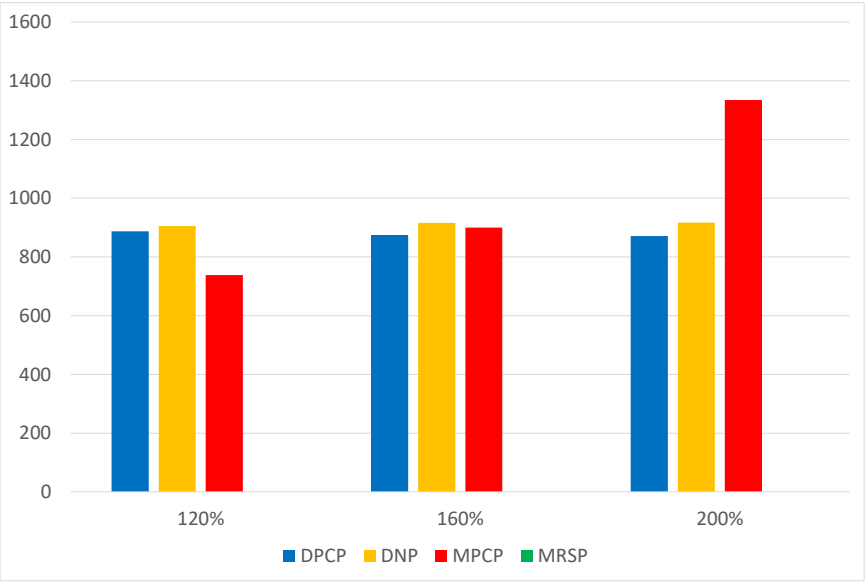

FIGURE 5.6: The result with four processors and five resource available.

response time. The results have been shown, DPCP in Figure 5.7 and MPCP in Figure 5.8.

After the evaluation, all the analysis and the conclusions will be discussed in Chapter 6.

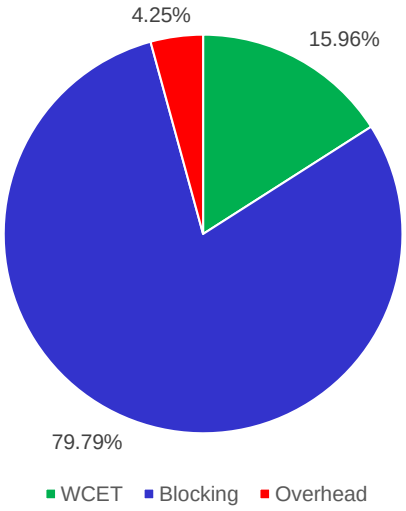

FIGURE 5.7: The pie chart for migration overhead of DPCP.

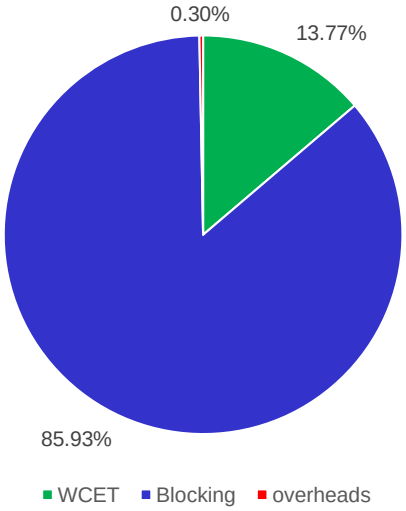

FIGURE 5.8: The pie chart for IPI overhead of MPCP.

Chapter 6

Conclusion

Known to all, the performance of one protocol depends on both the theory itself and the implementation. Besides the way how the protocol is implemented, the platform where the protocol is implement on and the machine where the implementation run can affect the performance as well. Thus, all the conclusions that we will draw on this chapter are based on the current implementation on the current platform running on our machine.

6.1 Results based on this current implementation

According to the evaluation that we have in Chapter 5 as well as the analysis for the theories in Chapter 2, we can arrive at several conclusions.

Firstly, the overheads caused by the $LITMUS^{RT}$ kernel can be neglected comparing to the big number of execution time as well as blocking time.Including of the overheads will not destroy the protocols' essences. Although the MPCP has less overheads than DPCP, MPCP still cannot perform better than DPCP under heavy load system, due to the blocking time is much lager than the benefit from the less overheads. For the task with the highest priority, under MPCP, it can be preempted by any other critical section on the local processor as well as the same resource accessing (direct blocking) on the remote processor. However, under DPCP, because of the distribution, the task with the highest priority cannot be preempted by any other of the tasks on the local processor. It can only be blocked once on the remote processor due to the mutual exclusion for each resource access. Thus, the MPCP can never perform better than DPCP in theory.

Under the real evaluation, the conclusion may not come to easily under low utilization for per processor situation which is treated as light load system. Considering the low percentage of the largest overheads of DPCP (less than 5%), we still consider that, the overheads caused by implementation cannot broke the protocol's theory.

Secondly, the high utilization can reflect the protocols' essences preferably in our evaluation. From the results of our experiments, we notice, those experiments with high utilization can demonstrate the protocol's essence preferably. To the contrary, those experiments with low utilization per processor, due to the executions for critical sections are loose, they can behave in a different way. The experiment for two processors with five resources available, the MPCP can perform better than DPCP, which is impossible in theory appears on our result. That because the limited of our design for the total execution time. Under the loose situation, the worst case in theory is hard to reach.

Last but not least, the summary based on observation is that the MPCP is sensitive to the utilization and the MrsP is sensitive to the number of processors.From the theories of those two protocols, the blocking time under MPCP mostly caused by other tasks' critical sections on the local processor, MrsP is mainly caused by the FIFO access to the resource on the remote processors. For the MPCP, if there are huge of critical sections on the local processor, the task with the highest priority will have very long blocking time. For the MrsP, if there are a lot of processors are available, and all of them contain the tasks will access to the resource as same as the task with the highest priority, it will be blocked m times (m equals to the number of the processor available).

6.2 Further Development

Although we have drawn some conclusions, there are still several aspects under consideration.

First of all, due to the average of the utilization, MPCP has the better performance in the average response time as well as those tasks with lower priority. That means it is valuable for the soft real-time system. The partition of the MPCP will have a significant influence to the performance. Thus, one efficient partition rule is pregnant.

Secondly, DNP may perform better than DPCP if the lengths of critical sections for each task are similar, and each critical section is reasonable short. Such condition can enlarge the effect caused by the context switch overhead. Also the situation if the task with the highest priority also request the longest critical section also need to be verified.

Thirdly, in the help mechanism under MrsP, it also need to migration, which will suffer from large overhead. So do we have any efficient partition algorithm to reduce such helping situation which can also reduce the overhead for whole system. Also, if one task which is helped by others, is preempted once it migrates to the remote processor. Under that situation, the help mechanism is just contribute to the overheads. Thus, if we set the task is non-preemptive once it is helped, can we reduce the possibility of more overheads?

Bibliography

[1] Giorgio Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[2] Andrea Bastoni, Bjorn B Brandenburg, and James H Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 14–24. IEEE, 2010.

[3] Andrea Bastoni, Bjorn B Brandenburg, and James H Anderson. Is semi-partitioned scheduling practical? In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 125–135. IEEE, 2011.

[4] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*, pages 181–190. IEEE, 2008.

[5] Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.

[6] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[7] Wen-Hung Huang, Maolin Yang, and Jian-Jia Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share?

[8] Rajib Mall. *Real-time systems: theory and practice*. Pearson Education India, 2009.

[9] Theodore P Baker. A stack-based resource allocation policy for realtime processes. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 191–200. IEEE, 1990.

[10] Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116–123. IEEE, 1990.

[11] Ragunathan Rajkumar. Synchronization in real-time systems: A priority inheritance approach. 1991.

[12] B Brandenburg. A note on blocking optimality in distributed real-time locking protocols,", 2012.

[13] Farhang Nemati, Moris Behnam, and Thomas Nolte. Multiprocessor synchronization and hierarchical scheduling. In *2009 International Conference on Parallel Processing Workshops*, pages 58–64. IEEE, 2009.

[14] Ragunathan Rajkumar, Lui Sha, and John P Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, volume 88, pages 259–269, 1988.

[15] Alan Burns and Andy J Wellings. A schedulability compatible multiprocessor resource sharing protocol–mrsp. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 282–291. IEEE, 2013.

[16] Robert I Davis and Alan Burns. Resource sharing in hierarchical fixed priority preemptive systems. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 257–270. IEEE, 2006.

[17] Farhang Nemati. Resource sharing in real-time systems on multiprocessors. 2012.

[18] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 149–160. IEEE, 2007.

[19] Björn B Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, Citeseer, 2011.

[20] John B Goodenough and Lui Sha. *The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks*, volume 8. ACM, 1988.

[21] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 73–83. IEEE, 2001.

[22] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. Litmus^ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126. IEEE, 2006.

[23] Linux testbed for multiprocessor scheduling in real-time systems ($LITMUS^{RT}$). URL https://www.litmus-rt.org/.

[24] Real-time executive for multiprocessor systems (RTEMS). URL http://www.rtems.org/.

[25] QNX Neutrino RTOS. URL http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html.

[26] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[27] D Grothe. Kgdb: linux kenrel source level debugger, 2001.

[28] Tracing-litmus. URL https://wiki.litmus-rt.org/litmus/Tracing.

[29] David Dice. Brief announcement: a partitioned ticket lock. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 309–310. ACM, 2011.

[30] Yan Solihin. *Fundamentals of parallel computer architecture*. Solihin Publishing and Consulting LLC, 2009.

[31] Scuola Superiore S Anna. Reality check: the need for benchmarking in rts and cps.

[32] S Kramer, D Ziegenbein, and A Hamann. Real world automotive benchmark for free. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*, 2015.