# AN INVESTIGATION
# of
# REAL-TIME SYNCHRONISATION


**Akira Nakamura**

Wolfson College
University of Cambridge
Computer Laboratory


**A dissertation submitted for the degree of
Doctor of Philosophy in the University of Cambridge**

**December 1993**

# Abstract

Real-time mechanisms are important for supporting user interfaces, multi-media, and network processing on workstations. Current operating systems provide such real-time mechanisms. However workstations do not yet fully exploit such strong real-time properties and have only recently begun to partially use them for user interfaces, multimedia and network processing.

In real-time environments, *priority inversion* is a well known phenomenon in which lower priority tasks block higher priority tasks because of semaphore blocking. It has been realised more recently that priority inversion is a serious problem when a real-time system uses task synchronisations for sharing resources. In order to solve the priority inversion problem, the *priority inheritance* protocol was invented about 20 years ago. The priority inheritance protocol was finally exploited in the standardisation of IEEE POSIX.

Our primary goal in the design of new real-time synchronisation algorithms will be as follows:

1. The blocking duration of the higher task which is blocked on semaphores held by the lower tasks should be greatly reduced, ideally to no more than one *working unit*.

2. The achievement of real-time synchronisations will avoid deadlock.

The first of these goals is the more important, and we might well be able to create a deadlock avoidance procedure to achieve the second goal which does not impact the first.

We developed the dependency protocol for uniprocessors. Moreover, we developed the *plug-in* method, colour semaphores, preemptive semaphores and the dependency semaphores for multiprocessors. This research has the goal of designing suitable real-time synchronisation algorithms for both uniprocessor and multiprocessor systems at the lowest system level for hard real-time scheduling.

# Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

# Acknowledgements

I would like to thank my supervisor, Professor Roger Needham, Head of the Computer Laboratory. He has taught me a great deal about doing research into operating systems, encouraged me to challenge new subjects and discussed new ideas many times. I would also like to thank Professor Maurice Wilkes for his encouragement and advice; without his encouragement I could not continue my study. Professor David Wheeler read the draft and gave me precious comments. The discussions with Dr Ragunathan Rajkumar and Dr Hideyuki Tokuda were helpful.

I am grateful for the help and friendship of numerous members of the Computer Laboratory, particularly members of the System Research Group. In particular, I would to thank Martin Coen, Paul Jardetzky, Mike Hinchey, Narm-Hee Lee, Mark Lomas, Cormac Sreenan and Tim Wilson.

For reading and commenting on drafts of this dissertation, I thank Roger Needham, Mark Lomas, Tim Wilson, Kevin Nute, Cormac Sreenan and Bridget Louise.

My deepest appreciation is reserved for my parents, Kazuya and Kaneyo Nakamura, who have given enormous support for these three years. I would like to thank my son, Tomohiko and my wife, Nobue for their help in my family life at Cambridge.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

This research has the goal of designing suitable real-time synchronisation protocols for both uniprocessor and multiprocessor systems at the lowest level in operating systems for real-time scheduling.

In this thesis, we investigate real-time synchronisation protocols, which are the basic mechanisms for supporting real-time systems with preemptive mechanisms for both uniprocessors and multiprocessors.

Most of the literature to date describes real-time properties for uniprocessor systems, but we should also look carefully at real-time properties for multiprocessor systems. In order to design real-time scheduling algorithms for such systems, the nature of *priority* must be closely investigated. We propose that there are two types of priority:

1. A measure for allocation of processor time.

2. A measure for managing semaphore blocking.

What type of priority would be needed in an environment with a large number of CPUs? In the case of the first sort of priority, the concept is defined relatively. In Multi-programming systems, the priority is defined so as to allocate multiple tasks to one processor. If there are unlimited similar processors, the concept of priority for allocating processors is meaningless; it simply disappears. However if there are 5 tasks for 3 processors, then the concept of priority is needed again. Indeed when there are many processors compared with the number of tasks, the second type of priority will still be important.

From a historical point of view, the system has many constraints which come from scarcity of resources rather than a large number of processors and severe requests. The system must provide sufficient services for the requests, therefore the use of semaphores is common to secure control of the system. In real-time environments, *priority inversion* is a well known phenomenon in which lower priority tasks block higher priority tasks because of semaphore blocking. It has been realised more recently that priority inversion is a serious problem when a real-time system uses task synchronisations for sharing resources. In order to solve the priority inversion problem, the *priority inheritance* protocol was invented about 20 years ago[**Needham92**] [1]. This priority inheritance protocol has since become the primary solution for the priority inversion problem in uniprocessor environments. The essential idea of the priority inheritance protocol is that when a task blocks higher priority tasks, it executes its critical section at the highest priority level of all of the blocked tasks and after exiting its critical section, the task immediately returns to its original priority level. The priority inheritance protocol was finally exploited in the standardisation of IEEE POSIX after it was invented 20 years ago.

However in an environment with a large number of processors; the priority inheritance protocol is not effective, because there are enough processors for nonblocking tasks to run. We therefore need to find new techniques and mechanisms based on a new principle.

The blocking duration of higher priority tasks, caused by semaphore operations activated by lower priority tasks, represents one of biggest problems in the design of hard real-time systems. We will therefore introduce the concept of the *working unit* [2] in order to measure blocking duration caused by semaphore locking.

Our primary goal in the design of new real-time synchronisation algorithms will be as follows:

1. The blocking duration of the higher task which is blocked on semaphores held by the lower tasks should be greatly reduced, ideally to no more than one *working unit*.

2. The achievement of real-time synchronisations will avoid deadlock.

The first of these goals is the more important, and we might well be able to create a deadlock avoidance procedure to achieve the second goal which does not impact the first.

The abnormal phenomenon caused by semaphore blocking occurs in any real time scheduling. Thus we will first concentrate only on the blocking duration caused by semaphore blocking in a real-time context, which will be called real-time synchronisation. Then we consider real-time scheduling algorithms and find one which is underlying for real-time synchronisation protocols.

By inventing new techniques for real-time synchronisation protocols for both uniprocessors and multiprocessors, we will also propose a new process state and process state transition for real-time systems.

## 1.2 Background

Real-time mechanisms are important for supporting user interfaces, multi-media, and network processing on workstations. Current operating systems, such as SUN-OS5.0 [**Khanna92**] provide such real-time mechanisms. However workstations do not yet fully exploit such strong real-time properties and have only recently begun to partially use them for user interfaces, multimedia and network processing.

[1] Roger Needham said: " I encountered priority inheritance for the first time in the E4 executive for the Modular One Computer in about 1970. The computer, and the company that made it, are long gone. I am not sure whose idea it was, it may possibly have been mine. I was a Director of the company, called Computer Technology Ltd."

[2] The concept of the *working unit* is defined in Section 3.1.1

The development of VLSI technology supplies a great number of CPU chips very cheaply; so that we can assume that proposed systems would have many CPU chips available with which to construct multiprocessor systems. For example, in multimedia systems, each system must deal with many kinds of very high volume continuous streams, requiring many different types of computations, including compression and decompression. We assume that multimedia systems will require multiprocessor systems. As for compression and decompression, specific VLSI chips are already being developed, such as JPEG chips and MPEG chips. While these are now being developed for dedicated purposes, general processing by software on multiprocessor systems will be much more feasible [**Leslie93**].

Some multimedia workstations are implemented by adding multimedia peripheral devices to workstations, for example the *Pandora* system [**Hopper90**]. The problem is that such external devices are expensive. Generally it can be expected that the work of multimedia processing should be done by software in processors on workstations. In such an environment, two primary limitations are likely: (1) bus bandwidth; (2) the behaviour of operating systems which turn out to need better real-time behaviour.

In order to solve the first problem, a *Desk Area Network* architecture was proposed [**Hayter91**] and is being developed at the University of Cambridge Computer Laboratory[**Hayter93**]. This system uses *Asynchronous Transfer Mode (ATM)* in place of bus control, which is documented by the CCITT. For example, in multimedia processing it is typical to be required to present frames of video at regular intervals, or to provide a constant rate of bits to drive a loudspeaker. If it is desired to have a number of streams of data going at one time, it is important to know whether the required work can be done in a reasonable time, and to ensure that it is. The ear is far more critical than the eye, so the more careful programming for audio drivers is much harder than that for video because of time constraints involved. Many authors, [**Steinmetz90**, **Campbell92**, **Sreenan92**, **Anderson93**], for example, have pointed out that real-time processing mechanisms are essential to achieve smoothness in multimedia applications.

It can therefore be assumed that many types of workstation, for example those which provide multimedia systems as suggested above, are implemented as:

1. real-time systems;

2. multiprocessor systems.

## 1.3 Multiprocessor Systems

There are a number of types of multiprocessor system: those with shared memory, those with distinct areas of memory for each processor, and a variety of hybrid systems in between [**Hennessy94**]. We feel it would be too large an undertaking to consider all synchronisation problems in all such systems. We have therefore chosen to concentrate upon one particular case, where all memory is shared between all processors. However, the techniques that we will propose, that are intended to apply such systems, are sometimes applicable to other types of multiprocessor systems.

The development of semiconductor process technology is very rapid. Workstations now have a large amount of memory; the difference of memory and disc capacity is now only one or two orders of magnitude. Paging mechanisms can therefore be reconsidered [**Wilkes92**]. Almost all current computer architectures are based on *locality of reference* [**Patterson90**]. For a single processor with a large amount of memory, frequent task switching reduces the effect of the cache and also causes the repetition of the creation and deletion of working sets. With shared-memory multiprocessor systems, it is not yet clear how to use a large amount of memory effectively.

Multiprocessor systems have had their drawbacks. Maurice Wilkes described some of these in explaining why in 1970, he decided to start the project of the capability system called CAP instead of developing a multiprocessor system:

> Everyone was saying that at that time that it would not long before very low cost microprocessors would be available, and that a large number of these, suitable interconnected, might form the main frame of the future. We gave much thought to a possible project based on this idea, but in the end concluded that all of them were open to objection. If a small number of memory modules were used, then the available memory bandwidth would not enable more than two or three processors to be supported. If, on the other hand, a large number of memory modules were used, then the cost of the necessary crossbar switch and its slow speed of operation would prove serious disadvantages. Systems in which each processor has access only to part of the memory did not appeal to us at all. [**Wilkes85**]

Compared with the situation described above, the *snoopy cache* [**Goodman83**] has been successful in solving the problem of cache consistency in multiprocessor systems. Now some main-frame computers and super computers are implemented with multiprocessors. Fast microprocessors, high-capacity DRAMs, and increasing network bandwidth are available to multiprocessor designers and are all improving at comparable rates [**Hennessy94**].

However multiprocessor systems have problems with regards to synchronisation among their processors; they need more efficient algorithms or mechanisms for processor synchronisation [**Owicki89**, **Mellor-Crummey91**, **Lim93**]. We need more sophisticate processor synchronisation mechanisms [**Anderson90**] for implementing proposed real-time synchronisation algorithms rather than simple *Test and Set* or *Compare and Swap*.

The design and implementation of multiprocessor operating systems based on UNIX [**Bach86**] are now quite common. The development of real-time operating systems for multiprocessors, for example, Spring [**Stankovic87**] and Real-Time Mach [**Tokuda90**], is also becoming common. A new challenge in advanced real-time systems is not only creating a fast and responsive kernel, but also providing a predictable and analysable real-time computing environment. However, it is often difficult to analyse the runtime behaviour of time-critical activities, due to lack of suitable real-time scheduling and synchronisation algorithms. The design and implementation of high level and low level synchronisation primitives for multiprocessor systems has therefore become an important subject of research.

## 1.4 Real-Time Scheduling

Priority is the most important parameter in real-time systems. In ordinary real-time systems, priorities are determined by a system policy. For example, in scheduling periodic tasks, the *rate monotonic algorithm* in which a task with a shorter period is given a higher priority than a task with a longer period [**Liu73**]. This classic real-time scheduling discipline has been adopted by many systems and investigated in great detail [**Lehoczy89**]. It is also very common that one real-time operating system has several real-time scheduling algorithms to respond flexibly to many kinds of external event [**Tokuda89a**, **Furht91**].

In this thesis, we first propose new real-time synchronisation protocols for both uniprocessor and multiprocessor systems where the priorities have already been set up, and will investigate their features in great detail. These real-time synchronisation protocols are at the lowest level and are for *hard real-time*. In real-time systems, an apparatus is needed to become a characteristic requirement that cannot break time constraints under any circumstances. Such mechanisms are more demanding than those designed for general purpose systems simply to get the best use of processors. The real-time synchronisation protocols which will be discussed in this thesis should support the simplest approach which is appropriate for a system.

As the next step, we will discuss the real-time scheduling policy underlying proposed real-time synchronisation protocols. There are two types of priority assignment policies: static and dynamic. In a static priority assignment policy, a priority for each task is predefined. In a dynamic priority assignment policy, a priority is assigned to each task according to observation of a real-time scheduling. There are two types of processor allocator for multiprocessors: static and dynamic. In a static processor allocation policy, a task can run only one dedicated processor called the *host processor*. In a dynamic processor allocation policy, a task can run on any of the processors which the system has available. We will propose a real-time scheduling which exploits both a dynamic priority assignment policy and a dynamic processor allocation. It is that a ready task is assigned to the busiest processor in the system according to the earliest deadline first algorithm. Finally, it should be noted that as processors become faster, more sophisticated real-time scheduling algorithms become feasible.

## 1.5 Overview

Priority inversion is a serious and common problem in real-time environments. It is very similar to the deadlock problem in operating systems. When the system uses shared resources and controls securely by setting up mutual exclusion sections using semaphores, deadlock is common in operating systems as is priority inversion in real-time systems. It is important to avoid deadlock in operating systems for keeping system secure. It is important to avoid unbounded priority inversion in real-time systems for keeping system secure. There are two possible approaches for deadlock avoidance: (1) try to re-plan by calculating the circular semaphore dependencies of all tasks, and (2) make the task claiming a semaphore wait until the possibility of deadlock is cancelled. For the first approach, there are both static algorithms and dynamic algorithms.

We will propose static and dynamic solutions for avoiding priority inversion. The second approach is not applicable for real-time systems.

In Chapter 2, we will discuss real-time synchronisation for uniprocessors in which we assume that the system provides a preemptive mechanism. Chapter 2 will begin to describe the phenomenon called *priority inversion* which appears in real-time synchronisations for uniprocessor systems with preemption. A task of high priority cannot proceed because a semaphore it needs is held by a task of low priority; the latter cannot run because a task of intermediate priority is using the processor by preemption. Thus the high priority task is arbitrarily delayed. The high priority claiming tasks are blocked by a lower priority locking task. Priority inversion is a serious problem in real-time environments because an urgent higher task may be blocked for a very long time.

There are two types of real-time synchronisation: short lived and long lived. Short lived synchronisation refers to mutual exclusion to confine access to simple data structures. Long lived synchronisation refers to mutual exclusion to confine access to complex data structures, data streams and external devices. Priority inversion is particularly serious in the case of long lived synchronisation. However restartable preemption might be available for very long lived synchronisation

The *priority inheritance* protocol was first proposed in order to address this problem about 20 years ago, though its importance has recently has been realised. When a higher task claims the semaphore which a lower task has locked, the lower task inherits the higher priority. In place of the higher claiming task being blocked, the holder of the semaphore, the locking task, runs with the higher priority. In other words, the basic concept of the priority inheritance is that when a lower task blocks higher tasks, it executes its critical section at the highest priority level of the blocked tasks. After exiting its critical section, the task immediately returns to its original priority. The principle of priority inheritance is that *"a locking lower task inherits the higher priority and completes the critical section as soon as possible"*.

Inadequacies in priority inheritance were discussed by Sha, Rajkumar, and Lehoczky [**Sha90**, **Rajkumar91**]. The biggest problem of the priority inheritance protocol is that the blocking time of the higher priority task depends on semaphore chains and semaphore nests. The second problem of the priority inheritance protocol is that it cannot prevent deadlocks. The *ceiling protocol* was proposed in order to address the problems of the priority inheritance protocol. The ceiling protocol gives higher tasks the benefit, and bounds the blocking time of higher priority tasks. Moreover, the ceiling protocol provides deadlock avoidance. In the ceiling protocol each semaphore has its own ceiling value. When a task locks a semaphore, it sets up the ceiling value which the system manages. When a task claims a semaphore, it needs a higher priority than the current highest ceiling value in order to lock the semaphore. If not, the task must be blocked before claiming a semaphore and the locking task inherits the highest priority of the blocked tasks. *"To block or stop before the semaphore"* without locking will be a very important mechanism overall in this thesis. Implementation will be discussed in Chapter 6, where we will discuss a proposed new process state called the paused state.

The rule of the ceiling value makes the ceiling protocol prevent deadlocks, but this also causes the ceiling protocol to be too conservative. A great many tasks which claim each semaphore which has not been locked could be blocked by the ceiling value of a quite unrelated semaphore. Therefore we propose a new real-time synchronisation algorithm for uniprocessors called the *semaphore dependency protocol* which is less conservative, on the basis that static analysis of the use of semaphores is already required for the ceiling protocol to work.

Static analysis of the programs can also lead to a set of dependency lists in the following sense. A semaphore $S_k$ is dependent on another semaphore $S_i$ at priority $P$ if there is a task at that priority which will seek to lock $S_i$ while holding $S_k$. The essential step is that an attempt to lock $S_k$ at priority $P$ should be blocked if there is a semaphore with $S_k$ dependent on it held at a higher or equal priority.

The dependency protocol is deadlock-free and the proof is given.

We will compare this with the previous algorithms for advantages and disadvantages. To sum up, the priority inheritance protocol works in simple cases, the ceiling protocol works in short deadline cases, and the dependency protocol works in long live synchronisation cases.

In Chapter 3 and Chapter 4, for the sake of simplicity we will assume that there exist the same number of tasks as there are processors in a multiprocessor environment. This simplification is necessary in considering the nature of priority in a multiprocessor environment, and to measure blocking duration by semaphore blocking. In such an ideal situation, preemption and priority inheritance cannot apply. We will focus on solving the blocking duration of higher tasks which are caused by lower tasks locking a member of semaphore nests and semaphore chains. In considering real-time synchronisation for multiprocessors, it is extremely difficult to grasp the complexities of a real solution, therefore using this ideal situation is helpful.

The concept of the *working unit* which is the innermost code of a semaphore nest will also be defined. According to this concept, the blocking time of a higher priority task which is blocked by a lower task locking can be measured in a simple form. The aim of the research is to find new real-time synchronisation protocols to reduce the blocking time of higher tasks to ideally at most one *working unit*.

In the ideal situation for multiprocessors, the principle of priority inheritance that "*a locking lower task inherits the higher priority and completes the critical section as soon as possible*", does not apply. In Chapter 3, we will discuss a new principle that "*a lower task is forced to avoid locking a semaphore which would block higher tasks*". For realising this principle, we will propose the *semaphore plug-in* method, which will generally shorten the *plug-in* method. In the ideal situation for multiprocessors, only semaphore nests and semaphore chains are effective in the multiple blocking of tasks. From the aspect of semaphore nests, the definition of dependency given in chapter 2 is also exploited. The essential point of the *plug-in* method is that while a task is locking a semaphore $S_i$, a task which claims a semaphore $S_k$, which is a parent semaphore of the dependent semaphore $S_i$, must use the sequence $\mathrm{P}(S_i)\mathrm{P}(S_k)\mathrm{V}(S_i)$ rather than simply $\mathrm{P}(S_k)$. In relation to time order, $S_i$ is classified as *forward*, *inward* and *backward* to $S_k$.

The *downward plug-in* method involves the plugging semaphore $S_i$, which is already locked by a higher task being applied to the parent semaphore $S_k$ which lower tasks are claiming. The *downward plug-in* method can work in the *inward* case. The *upward plug-in* method involves the plugging semaphore $S_i$ which is already locked by a lower task being applied to the dependent semaphore $S_k$ which higher tasks are claiming. The *upward plug-in* method works in the *forward* case. Both methods realise the principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"*. We will discuss the behaviour of systems which use the *plug-in* method, using a number of examples, and with them investigate essential features. This investigation will provide the basis for considering real-time synchronisation protocols in more realistic situations.

Semaphore nests and semaphore chains cause semaphore blocking. By analysing these examples, we will attempt to identify methods of avoiding semaphore blocking by operating semaphore nests and semaphore chains using the *plug-in* method.

In Chapter 4, we will discuss simultaneous operations for multiple semaphores. The simultaneous multiple semaphore operation is a special case of a semaphore nest. We will propose a new semaphore mechanism called *colour semaphores*, by introducing (1) simultaneous operations for multiple semaphores, (2) the priority ordered semaphore queues and (3) two other types of queues. We will discuss the real-time properties of colour semaphores for real-time synchronisation. The ordinary semaphore queue is time ordered to maintain fairness among tasks, especially in Time Sharing Systems. But for real-time systems, this time antecedent queue mechanism is not effective. In discussing the colour semaphores, we will propose the priority descendent ordered queue mechanism for semaphores. It will be shown that this mechanism produces high quality real-time properties for real-time systems. We will then discuss the use of *colour semaphores* in combination with the *plug-in* method.

In Chapter 5, we will proceed to look at more realistic situations for multiprocessor systems, in contrast to the ideal situations considered in the previous two chapters. In multiprocessor systems the number of tasks is greater than the number of processors, for example 8 tasks to 5 processors. In this more realistic situation for multiprocessors, the preemptive mechanism can apply again, and so too can the priority inheritance protocol. When a higher task claims a semaphore which is already locked, it must be blocked and relinquish the assigned processor which should be assigned to one of the ready tasks. When it is woken up, it should preempt a lower running task. First as a convenient tool for the preemption mechanism relating to semaphore operations which include the work of processor assignment, we will propose a new real-time synchronisation called *preemptive semaphores.*

Priority inheritance for multiprocessors will also be defined. This will be based on the simple fact that when a higher task claims a semaphore, which is locked by a lower task, it must be blocked and relinquish the assigned processor; the lower task will inherit the higher priority and can preempt or at least gain the relinquished processor. The priority inheritance for multiprocessors is thus recovered in a simple and useful form.

In previous chapters (Chapter 2, Chapter 3 and Chapter 4), we do not assume any real-time scheduling policy which is underlying for real-time synchronisation protocols. In this chapter we will propose a real-time scheduling policy that a ready task is assigned to the busiest processor in the system according to the earliest deadline first algorithm.

In Chapter 6, the dependency protocol for multiprocessors which includes the following three protocols will be designed: (1) the principle that *"a lower task is forced to avoid* *locking a semaphore which would block higher tasks*" (2) a priority descendent ordered semaphore queue, (3) the integrated priority inheritance protocol for multiprocessors. The principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"* is realised by introducing a process state called the *paused* state. The paused state is expressed as *"be blocked before claiming a semaphore without locking"*, or *"stop before a semaphore"*. This paused state is realised by the priority descendent ordered semaphore queue and therefore has no special queue for the paused state. A new process state transition for a real-time system, including the paused state, will then be defined. The dependency semaphores for multiprocessors will also be defined. Design and implementation details are considered.

In Chapter 7, we will describe related work, called the *ceiling protocol for multiprocessors* [**Rajkumar90**] and discuss the comparison between it and our proposed protocols. We will give a simple conclusion, and discuss future work.

In Appendix A, we will discuss the specification of the blocking time requirement for real-time synchronisation protocols in this thesis. The method of specification is important in describing complex mechanisms such as real-time systems.

In this thesis, we will discuss the nature of priority by inventing real-time synchronisation protocols. We consider the following two concepts of priority: (1) as a measure for allocation of processor time, (2) as a measure for managing semaphore blocking. We will show the concept of priority still works from the aspect of (2), where the concept of priority from the point of view of (1) disappears. When the number of tasks increases to more than that of processors, the ordinary concept of priority (1) is recovered, then the mixed situation where the system employs priority as both measures occurs.

Moreover, in this thesis, we will propose the *plug-in* method, *colour semaphores*, *pre-emptive semaphores* and *dependency semaphores* as real-time synchronisation protocols for multiprocessor systems. All of these include the mechanism of the priority descendent ordered queue for semaphores. We will investigate how this mechanism produces real-time properties for real-time synchronisations compared with the ordinary time antecedent queues for semaphores.

We will propose several algorithms for real-time synchronisation protocols. In the manner of *Literate Programming* [**Knuth92**], we endeavour to design these complex algorithms which are easily readable. To describe these algorithms in both compact and readable form, we will employ a pseudo programming language, using mathematical logic formulae.

# Chapter 2

# The Dependency Protocol for Uniprocessors

## 2.1 Outline

This chapter discusses methods of avoiding abnormal behaviour caused by priority inversion, and semaphore blocking in real-time processing in uniprocessor environments. We will attempt to show how these can be cured by more complex algorithms, namely: (1) the *priority inheritance protocol* (or the *inheritance protocol*), (2) the *priority ceiling protocol* (or the *ceiling protocol*), and (3) the *semaphore dependency protocol* (or the *dependency protocol*) to deal with increasingly complex situations.

## 2.2 Objective

Over the years there has been a considerable amount of research in this area under the heading of real-time systems, and indeed a recent operating system, SUN-OS5.0 [**Khanna92**] provides real-time synchronisation. However complications arise from the interference between processes, because of shared locks, so that straightforward synchronisation may not work. The present research will consider improvements to current available algorithms for synchronisation, and will then discuss whether the implicit complication involved is worthwhile.

Problems arise from shared locks when particular locks are shared between processes of different priority. The classical case is when a high priority process cannot proceed because a lock it needs is held by a low priority process; the latter cannot run because an intermediate priority process is using the processor. The high priority process is thus arbitrarily delayed. The earliest attempts to solve this problem, now twenty years old, depend on what is now known as priority inheritance, in which the holder of a lock runs with the priority of the most senior process waiting. The inadequacies of this mechanism were discussed by Sha *et al*, [**Sha90**]. They proposed a new protocol called, the priority ceiling protocol, which avoids the problems. The name comes from the association of each lock with a ceiling priority - that of the highest process which ever holds it.

However the ceiling protocol is rather too conservative in that it may cause waiting when it may not be necessary. We propose a less conservative protocol, observing that static analysis of the use of locks is already required for the priority ceiling protocol. The static analysis of a program can also lead to a set of general dependency lists for locking semaphores, and based on these a new scheduling rule is proposed.

We first give an overview of existing real-time synchronisation protocols, then set out the details of the proposed real-time synchronisation protocols called the *dependency protocol*, and finally compare the ceiling protocol with this dependency protocol.

It is traditional in the literature of this field to refer to '*semaphores*' rather than '*locks*', and to make comparison with earlier work easier we do so here, although it is a longer word and strictly speaking it refers to something more particular. The term '*task*' is designated as '*process*' for the purpose of this thesis. Throughout the work, *preemptive everywhere* is assumed, however the implementation details of the preemptive mechanism are not described in detail.

We mention several notations which we use in this thesis. Priorities are denoted by integers; higher integers representing higher priorities. The fact that priority $P_i$ is higher than $P_j$ for example is denoted as $P_i > P_j$. $S_i$ represents a semaphore and $T_j$ a task with priority $P_j$. The fact that the critical section of the task $T_i$ is controlled by the semaphore $S_j$ is denoted by $z_{ij}$.

## 2.3 Priority Inversion

*Priority inversion* describes the phenomenon that a high priority task is blocked waiting for the execution of a lower priority task. Some *priority inversion* is unavoidable because of mutual exclusion. A typical example is shown as follows:

—*Example of priority inversion*—
$T_0$ = {.........,P($S_0$),.......,V($S_0$),.....}
$T_1$ = {..........................................}
$T_2$ = {...,P($S_0$),.........,V($S_0$),.........}
($P_0 > P_1 > P_2$)

1. $T_2$ is initiated and locks $S_0$.
2. $T_1$ is initiated and preempts $T_2$.
3. $T_0$ is initiated and preempts $T_1$. It then claims $S_0$ and becomes blocked.
4. $T_1$ restarts.

First we explain the time order map notation to show how the semaphore P and V primitives are issued in the program code of each task. In the example above, on the left side, $T_0$, $T_1$ and $T_2$ denote tasks and suggest what program they will execute. The right side gives the time axis along which each task executes its own code. For example, the P primitive of $S_0$ of $T_2$ is issued earlier than that of $T_0$ if $T_0$ and $T_2$ arrives at the same time. We use this notation throughout this thesis.

The above example shows priority inversion for real-time systems in a simple form. While the highest priority task $T_0$ locks is blocked on the semaphore $S_0$ which the lowest priority task $T_2$ locks, the middle priority task $T_1$ is running. The priorities are inverted and the duration of the blocking time of $T_0$ is unpredictable because $T_1$ should first finish its work for $T_0$ to restart. This is a serious problem for real-time scheduling. One approach to partially solve the problem is that the task in critical sections is not allowed to be preempted: However, since this solution is only suitable where the duration of the critical sections is short, if they are not short it would produce unnecessarily long blocking. The preemptive mechanism is essential for higher tasks not to miss time constraints in hard real-time systems.

We discuss two examples of priority inversion, which actually occurred at the University of Cambridge Computer Laboratory using the WANDA Micro-kernel Operating System, which has the preemption mechanism. The first example was reported by Richard Hayton [**Hayton93**] as follows:

> The problem was that when two user level threads both performed several *printf* for debugging, the routines called involved various semaphores for several *mutexs*. I think there are two in the *printf* code itself and a further one or two in the serial driver. As these semaphores were used at the priority of the user level code, we got priority inversion if the threads had different priorities. This was particularly unacceptable as inversion was happening within the kernel, so the user had no control, and modifying the user level code (other than running everything at the same priority) would not solve the problem.

The other example was reported by Sai Lai Lo [**Lo93**] as follows:

> I need two threads to play back a CM(Continuous Media) file. One thread reads ahead and fills up a buffer. Another thread (CM) reads the data out of the buffer. Since I do not want the read-ahead thread to block the CM thread, because that would introduce unwanted jitter to the processing work done by the CM thread, I put the CM thread at a higher priority than the read-ahead thread. Unfortunately, access to the buffer has to be synchronised. So each thread has to grab a semaphore before it can touch the buffer. And occasionally, the read-ahead thread grabs the semaphore before it is preempted by the CM thread. As the CM-thread tries to grab the semaphore, it blocks, which is a classic priority inversion.

## 2.4 The Basic Priority Inheritance Protocol

Because *priority inversion* can cause an unpredictable blocking time, *priority inheritance* was invented. To avoid the problem of priority inversion, the priority of a higher task which is blocked should be inherited by the lower task which is holding the semaphore. More precisely, to prevent a low priority task which is blocking a higher task from being preempted by the middle priority task, the priority inheritance protocol makes the lower priority task inherit the higher priority whenever the higher task is blocked by the lower task, for the duration of the blocking. The blocking time of the higher task is then not infinite, but simply the execution time of the lower task running in its critical section. The general rule of priority inheritance is that a task runs at the highest priority of any tasks waiting for semaphores it holds. The rule is described in detail in [**Sha90**].

| *Priority Inheritance Rules* |
|---|
| 1. Task $T$ which has the highest priority is assigned to the processor.<br>It claims a semaphore $S$.<br>It will block on the semaphore $S$ if a lower priority task has already locked $S$.<br>Otherwise it locks $S$. |
| 2. If task $T$ blocks a higher priority task $T_H$ on the semaphore $S$, $T$ inherits $P_H$.<br>When it unlocks $S$, it resumes the priority it had at the point when it locked $S$. |
| 3. Priority inheritance is transitive. |
| 4. A Task $T$ can preempt a running task $T_L$ if its assigned or inherited priority<br>is higher than $P_L$. |

According to the example in the previous section,

1. $T_2$ is initiated and locks $S_0$.
2. $T_1$ is initiated and preempts $T_2$.
3. $T_0$ is initiated and preempts $T_1$. Then claims $S_0$ and becomes blocked. $T_2$ inherits the priority of $T_0$. $T_2$ restarts.
4. $T_2$ releases $S_0$, then $T_0$ restarts, then locks $S_0$.

The delay of $T_0$ is the time of blocking $S_0$ by $T_2$. Another example is shown below. This example shows the difference of the priority inheritance protocol by the ceiling protocol from this original mode.

—*Example for the Priority Inheritance Protocol*—
$T_0$ = {.........,P($S_0$),.........,V($S_0$),..}
$T_1$ = {.......,P($S_1$),.........,V($S_1$),....}
$T_2$ = {....,P($S_1$),.........,V($S_1$),.......}
$T_3$ = {..,P($S_0$),..........,V($S_0$),........}
$(P_0 > P_1 > P_2 > P_3)$

1. $T_3$ is initiated and locks $S_0$.
2. $T_2$ is initiated and locks $S_1$.
3. $T_1$ is initiated and is blocked. $T_2$ inherits the priority of $T_1$ and $T_2$ restarts.
4. $T_0$ is initiated and is blocked. $T_3$ inherits the priority of $T_0$, $T_3$ restarts and releases $S_0$. $T_0$ is woken up and restarts.

The priority inheritance mechanism was invented about 20 years ago, but its importance has only been realized more recently. The priority inheritance mechanism has been adopted as a standard for real-time systems and many operating systems implement it [**Lampson80**, **Tokuda89b**, **Khanna92**, **Fisher92**]. The priority inheritance protocol is exploited in the standardisation of IEEE, POSIX [**POSIX.4a90**]. An experience of implementation in a POSIX real-time system support is reported in [**Gallmeister91**].

## 2.5 The Priority Ceiling Protocol

The basic priority inheritance protocol has the following two major problems. First, a chain of blocking, which will be described in section 3.1.1, causes the same number of blocking as the number of semaphores in the chain. The second problem is that the basic priority inheritance does not prevent deadlocks. In order to address these problems, a new protocol called the *priority ceiling protocol* was proposed by Sha, Rajkumar and Lehoczky [**Sha90**]. Burns [**Burns91**] and Chen [**Chen91**] described the protocol in simple terms as follows:

| *The Priority Ceiling Protocol Rules* | |
|---|---|
| 1. | All tasks have a static priority assigned. |
| 2. | All semaphores have a ceiling value defined, which is the maximum priority of the tasks that may use it. |
| 3. | A task has a dynamic priority that is the maximum of its own static priority and any that it inherits because it has blocked a higher priority task on any other semaphores. |
| 4. | A task $T$ can only lock a semaphore $S$, if its dynamic priority is strictly higher than the priority ceiling of the semaphore $S^*$ which has the highest priority ceiling among all locked semaphores by tasks other than $T$.<br>Otherwise, the task should be blocked before claiming a semaphore without locking. (excluding any that it has already locked itself).<br>The task holding $S^*$ inherits the highest priority of the blocked tasks. |

Particularly good formal specifications for both the priority inheritance and the priority ceiling protocol, using the Z specification language [**Potter91**], were given in [**Pilling90**].

The priority ceiling protocol uses (or we also call it simply the *ceiling protocol*) a static analysis of dependency between lower priority $P_L$ tasks and the ceiling priority $P_H$ task which uses the semaphore $S$. A simple diagram of this dependency is shown below:

$$\boxed{P_L \xrightarrow{S} P_H}$$

When a lower priority task locks $S$, only tasks dynamically higher than $P_H$ can lock another semaphore. This simple model is used for proving the *deadlock avoidance* property of the ceiling protocol. Sha *et al*, [**Sha90**] give a thorough and mathematical analysis of the ceiling protocol using a graphical treatment. This protocol reduces the blocking time of the highest task to, at most, one critical section.

The performance evaluation of the ceiling protocol compared with priority inheritance and other synchronisation protocols in a Real-Time Mach environment is discussed in [**Tokuda91a**]. It is concluded that priority inversion degrades the schedulability significantly and the ceiling protocol is a very effective policy in real-time synchronisation.

### 2.5.1 Examples of the Ceiling Protocol

We will show the features of system behaviour of the ceiling protocol using examples quoted from [**Sha90**].

—*Example 1: Ceiling Protocol*—
$T_0$ = {........,P($S_0$),........,V($S_0$),........................}
$T_1$ = {......,P($S_1$),..,P($S_2$),...,V($S_2$),..,V($S_1$),......}
$T_2$ = {...,P($S_2$),..,P($S_1$),..,V($S_1$),..,V($S_2$),..........}
$(P_0 > P_1 > P_2)$

1. $T_2$ is initiated and locks $S_2$, the ceiling value of which is $P_1$.
2. $T_1$ is initiated and claims $S_1$, but the task priority is not higher than the current ceiling of locked semaphore $P_1$. It is therefore blocked. $T_2$ inherits the priority and $T_2$ restarts.
3. $T_0$ is initiated and preempts $T_2$.
4. $T_0$ locks $S_0$ and releases $S_0$.
5. $T_2$ restarts, locks $S_1$ and releases it, and then releases $S_2$.
6. Then $T_1$ locks $S_1$ and proceeds.

The above example illustrates deadlock avoidance using the ceiling protocol. It shows that the control rule of the ceiling value allows the ceiling protocol to prevent deadlocks.

—*Example 2: Ceiling Protocol*—
$T_0$ = {.......,P($S_0$),..,P($S_1$),...,V($S_1$),..,V($S_0$),......}
$T_1$ = {.....,P($S_1$),........,V($S_1$),............................}
$T_2$ = {...,P($S_0$),........,V($S_0$),..............................}
$(P_0 > P_1 > P_2)$

1. $T_2$ is initiated and locks $S_0$, the ceiling value of which is $P_0$.
2. $T_1$ is initiated, preempts $T_2$ and claims $S_1$, but the task priority is not higher than the current highest ceiling value. It should be blocked before $S_1$, then $T_2$ restarts and inherits the priority of $T_1$.
3. $T_0$ is initiated and is blocked by $S_0$. $T_2$ inherits the priority.
4. $T_2$ restarts and releases $S_0$. $T_0$ claims $S_0$.
5. $T_0$ locks $S_0$ and $S_1$, and releases $S_1$ and $S_0$.
6. $T_1$ restarts, then locks and releases $S_1$.

The above example shows that the ceiling protocol provides the highest priority task with the most benefit. It is important that $T_1$ be blocked before $S_1$ without locking because $T_0$ should be able to lock $S_1$.

—*Example 3: Ceiling Blocking*—
$T_0$ = {..............,P($S_0$),...,V($S_0$),..,P($S_1$),...,V($S_1$),....}
$T_1$ = {.....,P($S_2$),...,V($S_2$),.........................................}
$T_2$ = {..,P($S_2$),.,P($S_1$),....,V($S_1$),....,V($S_2$),................}
$(P_0 > P_1 > P_2)$

1. $T_2$ is initiated and locks $S_2$, the ceiling value of which is $P_1$.

2. $T_1$ is initiated and blocked by $S_2$. $T_2$ restarts, inherits $P_1$ and locks $S_1$, the ceiling value of which is $P_0$.
3. $T_0$ is initiated and claims $S_0$, but the task priority is not higher than the highest ceiling of locked semaphore $S_1$. It should be blocked before $S_0$. This is called *ceiling blocking*, which is unnecessary blocking. $T_2$ inherits the priority and restarts.
4. $T_2$ releases $S_1$. $T_0$ claims $S_1$, locks it and releases it.
5. $T_2$ releases $S_2$, then $T_1$ is initiated.

The above example shows the main drawback of the ceiling protocol. Instead of giving the greatest benefit to the highest task and preventing deadlocks, the ceiling protocol introduces unnecessary blocking, called *ceiling blocking*.

—*Example 4: Ceiling Protocol*—
$T_0$ = {........P($S_0$),....,V($S_0$),.................................}
$T_1$ = {......,P($S_0$),..,P($S_1$),...,V($S_1$),...,P($S_0$),.........}
$T_2$ = {....,P($S_0$),........................,V($S_0$),.................}
$T_3$ = {..,P($S_1$),...........................,V($S_1$)................}
($P_0 > P_1 > P_2 > P_3$)

1. $T_3$ is initiated and locks $S_1$, the ceiling value of which is $P_1$.
2. $T_2$ is initiated and claims $S_0$, but the task priority is not higher than the highest ceiling of locked semaphore. $T_2$ is blocked before $S_0$.
3. $T_1$ is initiated and claims $S_0$, but the task priority is not higher than the highest ceiling of locked semaphore. $T_1$ is blocked before $S_0$
4. $T_0$ is initiated and locks $S_0$.

### 2.5.2 Problems with the Ceiling Protocol

The ceiling protocol introduces a great many examples of the unnecessary blocking called *ceiling blocking* illustrated above. This kind of blocking occurs even in very simple cases, such as follows:

—*Example of the Drawback of the Ceiling Protocol*—
$T_0$ = {............,P($S_1$),....,V($S_1$),.....,P($S_0$),..,V($S_0$),....}
$T_1$ = {......,P($S_1$),........,V($S_1$),...................................}
$T_2$ = {....,P($S_1$),............,V($S_1$),.................................}
$T_3$ = {..,P($S_0$),.................,V($S_0$),..............................}
($P_0 > P_1 > P_2 > P_3$)

1. $T_3$ claims and locks $S_0$, the ceiling value of which is $P_0$.
2. $T_2$ claims $S_1$, but the task priority is not higher than $P_0$, therefore, is blocked before $S_1$.

In this example, the ceiling blocking is unnecessary: $T_2$ should be able to lock $S_1$. This means that ceiling protocol may be too restrictive and conservative a protocol for real-time synchronisation.

## 2.6 The Semaphore Dependency Protocol

In order to keep the delay of the highest priority task to a minimum, the ceiling protocol has to be too restrictive, conservative and *pessimistic* [**Rajkumar88**].

We can approach this problem by considering the case in which semaphores are held for one working unit[1]. We would like to ensure that high priority tasks are not delayed more than one working unit per set of semaphores used by that task as a result of lock holding by lower priority tasks. We would also like to do this without holding up tasks unnecessarily, as the ceiling protocol does.

The required *semaphore dependency protocol* would have the following features below:

1. It uses static general dependency lists and a dynamic dependency mechanism.

2. It exploits the preemption mechanism.

3. It provides deadlock avoidance. The proof of deadlock-free is given in Section 2.6.5.

4. It can be applied to multiprocessor systems. This is described in Chapter 6.

### 2.6.1 Definition of Dependency

For each semaphore held, and for each priority level higher than the native priority of the holder (i.e. before any increase due to priority inheritance), the runtime system keeps a list of semaphores dependent on the held semaphore, in the following sense.

| *The Definition of Dependency* |
| --- |
| $S_k$ is dependent on $S_i$ at priority $P$ if there is a task with priority $P$ which will seek to lock $S_i$ while holding $S_k$. The dependency list of $S_i$ is a set of pairs of a dependent semaphore and its priority. The dependency list of $S_i$ is static and applies while $S_i$ is being held. |

A simple diagram of the dependency is as follows:

$$\boxed{S_i \xrightarrow{P} S_k}$$

We call $S_i$ the *parent semaphore* or *GUARDIAN* and $S_k$ the *dependent semaphore* or *WARD* and use the following notations:
A set of dependent semaphores of $S_i$: $S_j \in S_D(S_i) = \{S_j | S_j \text{ is dependent on } S_i\}$
A set of parent semaphores of $S_k$: $S_j \in S_P(S_k) = \{S_j | S_k \text{ is dependent on } S_j\}$
$\mathsf{D}S_i = [(S_j, P_k)]$ means that $S_i$ is locked, then the element of the dependency list, $(S_j, P_k)$, a pair of a semaphore and a priority, is active.

Now we shall investigate the definition of dependency in more detail:

[1] The working unit is defined in Section 3.1.1

| |
|---|
| **If** $S_i$ is dependent on $S_j$ at priority $P$ and $S_j$ is dependent on $S_k$ at priority $Q$, $(Q > P)$.<br>**then**<br>$S_i$ is dependent on $S_k$ at priority $P$. |

Therefore we can exploit more general dependency lists:

| *The Definition of the General Dependency List* |
|---|
| Dependencies are transitive.<br>We should define the general dependency list as the transitive closure of dependencies. |

A set of transitive dependent semaphores of $S_i$:
$S_j \in GS_D(S_i) = \{S_j | S_j \text{ is transitive dependent on } S_i\}$
A set of transitive parent semaphores of $S_k$:
$S_j \in GS_P(S_k) = \{S_j | S_k \text{ is transitive dependent on } S_j\}$
The general dependency list $\mathsf{GD}S_i = [(S_j, P_k)]$ means that $S_i$ is locked, the element of the general dependency list $(S_j, P_k)$, a pair of a semaphore and a priority, is active.

### 2.6.2 The New Scheduling Rule

The new scheduling rule becomes:

| *The Scheduling Rule of the Dependency Protocol* |
|---|
| A task $T$ running at priority $P$ is not allowed to lock a semaphore $S_k$<br>if any semaphore already locked has $S_k$ at a priority greater than or equal to $P$<br>in one of its general dependency lists not activated by the task $T$ itself.<br>A task which is unable to lock a semaphore is blocked immediately before the lock call,<br>and released when the condition on which it was blocked disappears.<br>The task locking the parent semaphore inherits the priority $P$ from the blocked task.<br>The ordinary priority inheritance scheme [2]is also applied. |

This new scheduling rule means that the potentially lower tasks which are claiming a semaphore in the general dependency lists of locked semaphores should be blocked immediately before the lock call. This rule assumes that tasks higher than $P$ priority would claim the semaphore $S_k$ in the near future. If $S_k$ has already been locked, the task higher than $P$ should be blocked. Instead of the task locking $S_k$, it is blocked just before locking $S_k$. We call this *predictable blocking* or *pausing*. The semaphore dependency protocol provides a *predictable blocking* or *pausing* mechanism so that a task higher than $P$ can lock the dependent semaphore $S_k$ instead of being blocked because of locking by lower tasks. At this time, $P$ is the highest priority to claim $S_k$ and $P$ is inherited by the lower task locking the semaphore which has $S_k$ in its general dependency list.

[2]We might propose the $\epsilon$-plus priority inheritance scheme which causes the task to inherit the priority of the blocked task, $P$ plus a small increment, $\epsilon$, less than 1.

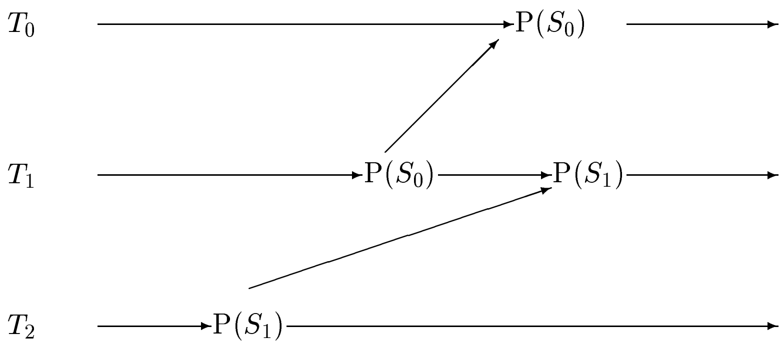


Figure 2.1: Predictable Blocking

For the ceiling protocol, all semaphores which a task uses from the arrival to the completion are targets of the rule. Therefore if a higher task uses many semaphores during this period, the current ceiling value remains hight: This causes unnecessary blocking. Comparing the ceiling protocol with the dependency protocol, only semaphore nests which a task use at some time are targets of the rule.

### 2.6.3 The Basic Concept of the Dependency Protocol

The major function of the dependency protocol is to ensure predictable blocking for higher priority tasks. As shown in Figure 2.1 on this page, when the lowest task already holds the parent semaphore of the general dependency lists in which a middle task is included, the middle task should be blocked just before the dependent semaphore of the general dependency lists. The dependency semaphore of the middle task has not been locked. By this predictable blocking, a higher task can benefit such that it can avoid being blocked by the middle task locking the dependent semaphore that the higher task will claim in the near future. This means that the parent semaphores have already been locked by a lower task; therefore, instead of trying to lock the dependent semaphore, the processor is given to a *predictable* higher task.

In Section 6.1 of Chapter 5, a new process state will be defined: That a task stops before a semaphore is expressed that the task is in the paused state. According to this new process state, the middle task is said to enter the paused state. As discussed in the section, tasks in the paused state need to be processed carefully.

### 2.6.4 Examples of the Semaphore Dependency Protocol

—*Example 1:The Dependency Protocol*—
$T_0$ = {........,P($S_0$),........,V($S_0$),.........................}
$T_1$ = {......,P($S_1$),..,P($S_2$),...., V($S_2$),..,V($S_1$),.....}
$T_2$ = {...,P($S_2$),..,P($S_1$),..,V($S_1$),..,V($S_2$),..........}
($P_0 > P_1 > P_2$)

The general dependency lists are:
$\mathsf{GD}S_1 = [(S_2, P_2)]$,$\mathsf{GD}S_2 = [(S_1, P_1), (S_2, P_1)]$

1. $T_2$ is initiated and locks $S_2$.
2. $T_1$ is initiated and claims $S_1$, but the general dependency list:
   $\mathsf{GD}S_2 = [(S_1, P_1), (S_2, P_1)]$
   shows it should be blocked and $T_2$ inherits the priority.
3. $T_0$ is initiated and preempts $T_2$.
4. $T_0$ locks $S_0$ and releases $S_0$.
5. $T_2$ restarts, locks $S_1$ and releases it, and releases $S_2$.

This execution process is the same as the ceiling protocol. The above example shows how both protocols avoid the deadlock.

—*Example 2:The Dependency Protocol*—
$T_0 = \{$.......,P($S_0$),..,P($S_1$),...,V($S_1$),..,V($S_0$),........$\}$
$T_1 = \{$.....,P($S_1$),.........,V($S_1$),...............................$\}$
$T_2 = \{$...,P($S_0$),........,V($S_0$),.................................$\}$
$(P_0 > P_1 > P_2)$

The general dependency list is: $\mathsf{GD}S_1 = [(S_0, P_0)]$

1. $T_2$ is initiated and locks $S_0$.
2. $T_1$ is initiated, preempts $T_2$ and locks $S_1$.
3. $T_0$ is initiated and by the general dependency list:
   $\mathsf{GD}S_1 = [(S_0, P_0)]$
   stops before $S_0$. $T_2$ inherits the priority.
4. $T_2$ restarts and releases $S_0$. $T_0$ claims $S_0$, but by the general dependency list:
   $\mathsf{GD}S_1 = [(S_0, P_0)]$
   $T_0$ stops before $S_0$ and $T_1$ inherits the priority.
5. $T_1$ restarts, then releases $S_1$.
6. $T_0$ locks $S_0$ and $S_1$, then releases them.

Using the ceiling protocol, $T_1$ is blocked instead of locking $S_1$. The delay of $T_0$ is only the blocking time of $T_2$. In the case of the dependency protocol, the delay of $T_0$ is the blocking time of both $T_2$ and $T_1$.

—*Example 3:The Dependency Protocol*—
$T_0 = \{$...............,P($S_0$),..,V($S_0$),.,P($S_1$),...,V($S_1$)........$\}$
$T_1 = \{$.....,P($S_2$),.., V($S_2$),..........................................$\}$
$T_2 = \{$...,P($S_2$),.,P($S_1$),....,V($S_1$),....,V($S_2$),................$\}$
$(P_0 > P_1 > P_2)$

The general dependency list is: $\mathsf{GD}S_1 = [(S_2, P_2)]$

1. $T_2$ is initiated and locks $S_2$.

2. $T_1$ is initiated and is blocked by $S_2$. $T_2$ restarts, inherits $P_1$ and locks $S_1$.
3. $T_0$ is initiated, locks $S_0$, releases it and claims $S_1$, then is blocked and $T_2$ inherits its priority.
4. $T_2$ releases $S_1$. $T_0$ claims $S_1$, locks it and releases it.
5. $T_2$ releases $S_2$, then $T_1$ is initiated.

In the ceiling protocol, $T_0$ should be delayed in locking $S_0$ because $T_2$ locks $S_1$ with priority $P_0$. This is a new form of blocking introduced by the ceiling protocol called *ceiling blocking* and is unnecessary. The semaphore dependency protocol works in a similar way to the basic priority inheritance protocol.

—*Example 4:The Dependency Protocol*—
$T_0 = \{$........,P($S_0$),....,V($S_0$),..............................$\}$
$T_1 = \{$......,P($S_0$),..,P($S_1$),...,V($S_1$),...,P($S_0$),.......$\}$
$T_2 = \{$....,P($S_0$),........................,V($S_0$),...............$\}$
$T_3 = \{$..,P($S_1$),...........................,V($S_1$)...............$\}$
$(P_0 > P_1 > P_2 > P_3)$

The general dependency list is: $\mathsf{GD}S_1 = [(S_0, P_1)]$

1. $T_3$ is initiated and locks $S_1$.
2. $T_2$ is initiated and claims $S_0$. But as shown in the general dependency lists, it becomes blocked.
3. $T_1$ is initiated and claims $S_0$. But as shown in the general dependency lists, it becomes blocked.
4. $T_0$ is initiated, locks $S_0$.

The above example shows that the priority requirement in the scheduling rule is necessary. The ceiling protocol also works, because the ceiling priority of $S_1$ is $P_1$.

—*Example 5: Deadlock Avoidance*—
$T_0 = \{$........,P($S_0$),..,P($S_2$),....,V($S_2$),....,V($S_0$),......$\}$
$T_1 = \{$......,P($S_1$),..,P($S_0$),.........,V($S_0$),..,V($S_1$),.....$\}$
$T_2 = \{$...,P($S_2$),..,P($S_1$),..,V($S_1$),..,V($S_2$),...............$\}$
$(P_0 > P_1 > P_2)$

The general dependency lists are:
$\mathsf{GD}S_0 = [(S_1, P_1), (S_2, P_1)]$, $\mathsf{GD}S_1 = [(S_2, P_2)]$,$\mathsf{GD}S_2 = [(S_0, P_0), (S_1, P_1), (S_2, P_2)]$

1. $T_2$ is initiated and locks $S_2$.
2. $T_1$ is initiated and claims $S_1$, but the general dependency list of $S_2$ shows it should be blocked and $T_2$ inherits the priority.
3. $T_0$ is initiated and preempts $T_2$. $T_0$ claims $S_0$ and is blocked by the general dependency list of $S_2$. Then $T_2$ inherits the priority.
4. $T_2$ restarts, locks $S_1$ and releases it, and releases $S_2$.
5. $T_0$ restarts, locks $S_0$ and $S_2$ and releases $S_2$ and $S_0$.

The original example causes typical deadlock, however the dependency protocol can avoid this type of deadlock.

### 2.6.5 The Properties of the Semaphore Dependency Protocol

The ceiling protocol uses the maximum possible dependency. It causes unnecessary blocking compared with the semaphore dependency protocol. This is because the semaphores of the higher tasks that are not related to the current locking semaphores block the middle tasks or the higher task itself, as in example 3 in Section 2.5.1, by *ceiling blocking*. In the semaphore dependency protocol, only the semaphores in the general dependency list (of the current locking semaphore) block the higher task.

The dependency protocol is deadlock-free. The deadlock problem is discussed in [**Holt71**, **Holt72**, **Shaw74**]. The proof is shown in the following.

The set of tasks is as follows:
$\mathbf{T} = \{ T_0, T_1, T_2, ........., T_{n-1} \}$.
The set of Semaphores is as follows:
$\mathbf{S} = \{ S_0, S_1, S_2, ........, S_{m-1} \}$.
$(S_i, T_n)$ means that $S_i$ is acquired by $T_n$. $(T_n, S_j)$ means that $S_j$ is required by $T_n$. For simplicity, semaphores express states, tasks express events. A transition, $S_i \xrightarrow{n} S_j$ means that $T_n$ acquires $S_i$ and requires $S_j$, a sequence $(S_i, T_n)$ and $(T_n, S_j)$.

If the dependency protocol were not deadlock-free, then the sequence transitions form a circle. Therefore, if the circle exists, it would be:
$S_{s_0} \xrightarrow{t_0} S_{s_1} \xrightarrow{t_1} S_{s_2} ......S_{s_{m'}} \xrightarrow{t_{m'}} S_{s_0}$.
In the case of $m' = 1$, the proof is given directly by the rule of the dependency protocol in Section 2.6.2. In the case of $m' = 2$, Example 5 shows deadlock-free.
Let's $T_{t_0}$ is the lowest priority task. It locks $S_{s_0}$. By the rule of the dependency protocol, the task $T_{t_{m'}}$ should stop before $S_{s_{m'}}$. By transitions from $S_{s_0}$, some tasks $T_{i-1}$ and $T_i$ becomes $P_{t_{m'}} > P_i \wedge P_{i-1} > P_i$ such that $T_i$ stops before $S_i$ because of transitive property. $T_{i-1}$ can lock both $S_{i-1}$ and $S_i$ because $S_i$ is at $P_i$ in the general dependency list and is not locked. This is contradictory.
Therefore by the above discussion, the dependency protocol is deadlock-free.

## 2.7 Discussion

Both the ceiling protocol and the semaphore dependency protocol use static dependency lists. What then is the essential function of the dependency-based protocol compared with the basic protocol inheritance?

### 2.7.1 Dependency Order

One assumption is that using dependency lists causes an ordering of the priorities of tasks. If this were true, to have more dependencies is better, which means that the ceiling protocol works. For example:

—*For Ceiling Protocol*—
$T_0 = \{$.........,P($S_1$),.........,V($S_1$),..$\}$
$T_1 = \{$.......,P($S_2$),.........,V($S_2$),....$\}$
$T_2 = \{$....,P($S_2$),.........,V($S_2$),.......$\}$

$T_3$ = {..,P($S_1$),..........,V($S_1$),........}
($P_0 > P_1 > P_2 > P_3$)

In the above example, the dependency protocol cannot find any dependencies. The ceiling protocol assumes the most dependencies, therefore it causes unnecessary blocking, i.e. *ceiling blocking*. The number of dependencies from the semaphore dependency protocol is lower, as shown in the following example:

—*Against the Dependency Protocol*—
$T_0$ = {.......,P($S_1$)P($S_2$)P($S_3$),....,V($S_3$)V($S_2$)V($S_1$)..}
$T_1$ = {.....,P($S_3$),....................,V($S_3$),......................}
$T_2$ = {....,P($S_2$),....................,V($S_2$),........................}
$T_3$ = {..,P($S_1$),....................,V($S_1$),..........................}
($P_0 > P_1 > P_2 > P_3$)

The highest task $T_0$ should wait for three working units, even though the general dependency lists are activated compared with one working unit by the ceiling protocol. Therefore the semaphore dependency protocol has no advantages in the above example.

### 2.7.2 The Degree of Solutions

It can be said that in the simplest cases the inheritance protocol works well enough. However it cannot avoid deadlock and provide any solutions for the semaphore chain problem. The ceiling protocol is deadlock-free, provides the solution for the semaphore chain problem and gives guarantee of the highest task waiting for only one working unit. However if a higher task use great many of semaphores, the current ceiling value is raised high, a great amounts of unnecessary blocking are introduced. If a higher task which runs rarely, say once in 5 years uses a semaphore, this introduces unnecessary blocking. Moreover if the size of critical section is long, the unnecessary blocking may cause system degradation. If a task is alive in short term which means short deadline, it does not use many semaphores, therefore the ceiling protocol works very well. The dependency protocol is deadlock-free and provides the solution for the semaphore chain problem. If the size of critical section is long, higher task should not wait unnecessarily long. However it must maintain and check the complex data structure rather ceiling values.

From the size of critical section, if a critical section is very short we can afford to use *kernelised monitor* which takes the non-preemptive mode [**Tokuda91a**]. We can therefore conclude the following:

- The *priority inheritance* works in simple cases.
- The *priority ceiling protocol* works well in short deadline cases.
- The *semaphore dependency protocol* works well in long lived synchronisation cases.

### 2.7.3 The Complexity of Dependency

Both the ceiling protocol and the dependency protocol are based on static analysis and require the following two procedures:

1. They calculate the static dependencies in source code.

2. They put information about the dependencies into the kernel data area where the semaphore primitives can operate.

The dependency protocol is more complex in calculating dependency and creating the general dependency lists than the ceiling protocol because the dependency protocol treats semaphore nests, the expression of which is not easy in the system code. In section 6.3 in Chapter 6, the explicit declaration of semaphore nests will be proposed.

# Chapter 3

# The Plug-in Method for Multiprocessors

## 3.1 Outline

In this chapter we discuss real-time synchronisation for multiprocessor systems, which is the primary subject of this thesis.

We would like to note that in Chapter 3 and Chapter 4 a shared memory multiprocessor system is assumed and that there were the same number of processors as of tasks. This idealistic assumption is very important in the study of real-time synchronisation for multiprocessor systems. Real-time synchronisation for multiprocessor systems is too complex to be intuitive. In an environment with the same number of tasks as of processors, the preemption mechanism and the priority inheritance mechanism lose their meaning on multiprocessor systems. Ultimately the aim is to solve the problem of semaphore blocking which causes *priority inversion* in a real-time environment of multiprocessors. However, in ideal conditions, we are better able to find new mechanisms which can later be applied to more realistic situations for multiprocessors. The principle of priority inheritance is that *"a locking lower task inherits the higher priority and completes the critical section as soon as possible"*. We propose a new principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"*. This principle leads to two new real-time synchronisation protocols: the *plug-in* method and the *colour semaphores* for multiprocessor systems.

We discuss the semaphore blocking problem using several examples similar to the following example. In the use of the notation, the horizontal dots represent time elapsing; it is a time serial system. Priorities are denoted by integers; higher integers represent higher priorities. $S_i$ denotes a semaphore and $T_j$ denotes a task with priority $P_j$. $P_i > P_j$ means that the task $T_i$ has a priority $P_i$ higher than the task $T_j$. $\wp_i$ denotes a processor. We assume that all processors have the same performance. The critical section of the task $T_i$ controlled by the semaphore $S_j$ is denoted by $z_{ij}$. The notation "DO-WORK" will be defined later.

We begin the discussion with the following simple example:

—*Typical Example for Multiprocessor*—
$\boxed{\wp_0}$ $T_0$ = {..........,P($S_1$),.......,V($S_1$),.....................................}
$\boxed{\wp_1}$ $T_1$ = {........,P($S_1$),.,P($S_0$),..DO-WORK1..,V($S_0$),V($S_1$),...}
$\boxed{\wp_2}$ $T_2$ = {............,P($S_0$),...DO-WORK2....,V($S_0$),................}
($P_0 > P_1 > P_2$)

1. $T_0$,$T_1$,$T_2$ are initiated. $T_1$ locks $S_1$.
2. $T_0$ claims $S_1$, but is blocked.
3. $T_2$ locks $S_0$ and executes DO-WORK2.
4. $T_1$ claims $S_0$ and is blocked.
5. $T_2$ releases $S_0$. $T_1$ locks $S_1$, executes DO-WORK1, then releases $S_0$.
6. $T_0$ can lock $S_1$.

In the above example, the highest task is blocked by $S_1$ issued by the middle task $T_1$, $T_1$ is then blocked by $S_0$ issued by the lowest task $T_2$.

### 3.1.1 The Concept of the Working Unit

In this thesis, we concentrate on investigating the blocking time caused by semaphore blocking. The main problem is how to measure the blocking duration in a simple manner. For this purpose, we introduce the concept of the *working unit*.

| *The Definition of the working unit* |
|---|
| Assuming there is a nest of semaphores. A task does real work in the innermost code of the nest. We call the part of code DO-WORK part and the execution time the *working unit*. DO-WORK represents computational capability, and is preemptive. It does not include any I/O delays. |

In the previous example, $T_1$ has a DO-WORK part, and its *working unit* is expressed as DO-WORK1. $T_2$ has a DO-WORK part; its *working unit* is expressed as DO-WORK2.

Even though there is a sufficient number of processors, for example three processors, the blocking time of the highest task $T_0$ is at least two working units, that is: DO-WORK1 + DO-WORK2.

As already stated, the primary aim of this work is to develop algorithms to reduce the blocking time of higher tasks blocked by lower tasks, ideally to no more than one working unit.

We return to the starting points of the blocking problem here. The primary sources of semaphore blocking are:

1. Intra-task and intertask semaphore nests,
2. Intertask semaphore chains.

We will now illustrate examples in which semaphore nests and semaphore chains cause semaphore blocking.

——*Intra-task Semaphore Nest*——

$\boxed{\wp_0}$ $T_0$ = {..........,P($S_0$),......,P($S_1$),......,P($S_2$),.DO-WORK0....,V($S_2$),V($S_1$),V($S_0$),..}
$\boxed{\wp_1}$ $T_1$ = {.......,P($S_0$),.DO-WORK1........,V($S_0$),......................................................}
$\boxed{\wp_2}$ $T_2$ = {..................,P($S_1$),..DO-WORK2....,V($S_1$),.............................................}
$\boxed{\wp_3}$ $T_3$ = {................................,P($S_2$),..DO-WORK3............,V($S_2$),.......................}
$(P_0 > P_1 > P_2 > P_3)$

1. $T_0$, $T_1$, $T_2$, $T_3$ are initiated.
2. $T_1$ locks $S_0$. $T_0$ claims $S_0$ and is blocked.
3. $T_2$ locks $S_1$. $T_1$ executes DO-WORK1 and releases $S_0$. $T_0$ restarts, claims $S_1$ and is blocked.
4. $T_3$ locks $S_2$. $T_2$ executes DO-WORK2 and releases $S_1$. $T_0$ restarts, claims $S_2$ and is blocked.
5. $T_3$ executes DO-WORK3 and releases $S_2$, then $T_0$ locks $S_2$ and executes DO-WORK0.

The highest task $T_0$ might be blocked three times before it can execute its own working unit DO-WORK0. In the worst case, the blocking time is DO-WORK3 + DO-WORK2 + DO-WORK1.

——*Intertask Semaphore Chain*——

$\boxed{\wp_0}$ $T_0$ = {..........,P($S_0$),.......,P($S_1$),.....DO-WORK0....,V($S_1$),V($S_0$),..........}
$\boxed{\wp_1}$ $T_1$ = {..........,P($S_1$),.......,P($S_2$),.....DO-WORK1....,V($S_2$),V($S_1$),..........}
$\boxed{\wp_2}$ $T_2$ = {..........,P($S_2$),.......,P($S_3$),.....DO-WORK2....,V($S_3$),V($S_2$),..........}
$\boxed{\wp_3}$ $T_3$ = {..........,P($S_3$),.......,P($S_4$),.....DO-WORK3....,V($S_4$),V($S_3$),..........}
$(P_0 > P_1 > P_2 > P_3)$

1. $T_0$, $T_1$, $T_2$, $T_3$ are initiated.
2. $T_0$ locks $S_0$, $T_1$ locks $S_1$, $T_2$ locks $S_2$ and $T_3$ locks $S_3$.
3. $T_0$ claims $S_1$ and is blocked. $T_1$ claims $S_2$ and is blocked. $T_2$ claims $S_3$ and is blocked.
4. $T_3$ locks $S_4$ and executes DO-WORK3, then releases $S_4$ and $S_3$.
5. $T_2$ locks $S_3$ and executes DO-WORK2, then releases $S_3$ and $S_2$.
6. $T_1$ locks $S_2$ and executes DO-WORK1, then releases $S_2$ and $S_1$.
7. $T_0$ locks $S_1$ and execute DO-WORK0.

The blocking time of the highest task $T_0$ is DO-WORK3 + DO-WORK2 + DO-WORK1, a total of three working units.

By introducing the concept of the *working unit*, we simplify the problem to clarify the model so that new algorithms for real-time synchronisation of multiprocessor systems can be developed and the blocking time in such complex systems can be measured. In almost all of the existing literature, all scheduling events caused by semaphores, and especially critical sections are taken into consideration, and as a result problems become extremely complex. To get a simple model is important to study a complicated system. The advantages of exploiting the *working unit* are as follows:

1. Essential works are executed in innermost critical sections.

2. The definition of *blocking time* becomes simple.

3. For measuring the blocking time of higher tasks, we concentrate on two types of semaphore combinations: (1) semaphore nests and (2) semaphore chains.

We now know that higher tasks share semaphores with lower tasks and the higher tasks might be blocked by a lower task locking the shared semaphore. Clearly the highest task should not be blocked for more than one *working unit* because the semaphore is held by lower priority task. We need a more careful statement about the higher tasks apart from the highest, which might be as follows:
The *natural start time* for the work period of a task $T$ is when it would start if there were semaphores shared with lower priority tasks, and if all tasks lower than $T$ were removed from the system. No task should ever be delayed by more than one working unit after its *natural start time*.

### 3.1.2 The Concept of Priority

**Priority** can be defined as:

> the condition or quality of being earlier or previous in time, or of preceding something else. Precedence in order, rank, or dignity. A precedence among claims, or a preference in order of payment. < *Oxford English Dictionary* >

The notion of priority is important when dealing with systems using the technique of processor sharing [**Wirth77**]. In computer systems all sorts of resource constraints can be used to implement the concept of priority. In multiprogramming systems, multiple tasks compete to get processor time given from a single processor which the system has. The processor used to be a serious constraint, and priority in the allocating of processor time is really common. The development of VLSI technology has led to a steady and rapid reduction in the price of processors, this allows us to include many processors in a system that might previously have had only one. With many processors, we might be tempted to ignore the requirement to allocate process priorities.

In a real-time system, external requests produce many sorts of constraints, which are reasons for setting up priorities. The most common way to securely control a large number of constraints is to use semaphores. As we have seen, using semaphores can cause tasks to be blocked and the most serious blocking for a real-time system is when lower priority tasks block higher priority tasks.

### 3.1.3 The Classification of Priority

For the purpose of this thesis, we consider two sorts of priority:

1. as a measure for the allocation of processor time.

2. as a measure for managing semaphore blocking.

In the case of the first sort of priority, the concept of priority is defined relatively because it is usually defined in the situation where there are many tasks for one processor. But if there are sufficient processors, the concept of this type of priority disappears. Moreover, if there are say 5 tasks for 3 processors, then priority appears again.

Even though there are many processors compared with the number of tasks, the second sort of priority has meaning.

A priority is assigned to a task in a waiting queue. In the first case, a task in the run queue has a priority, and in the second case a task in a semaphore queue has a priority.

### 3.1.4 Critical Sections

The code enclosed between P and V operations is called a *critical section* for resource sharing. But we exploit the concept of *working unit* instead.

Tasks can be *mutually exclusive* by forming a pair of P and V primitives of a semaphore $S$, the initial value of which is 1, of each task as follows:

—*Mutual Exclusion*—

...

$T_i = \{$.......,P($S_0$),..DO-WORK$_i$.....,V($S_0$),..............$\}$

...

$T_k = \{$.......,P($S_0$),..DO-WORK$_k$.....,V($S_1$),.............$\}$

...

When the higher task $T_i$ is able to lock $S_0$, the concern is to reduce its blocking time by other semaphores held by lower tasks if nested. When the lower task $T_k$ locks $S_0$, rather than $T_i$, the delay of the lower task $T_k$ propagates to the delay of the higher task $T_i$, therefore the concern is to minimise the blocking time of the lower task $T_k$ with the purpose that the delay of the higher task $T_i$ is also reduced, ideally to no more than one working unit.

## 3.2 Objective

We can approach the problem by considering the case in which semaphores are held for one unit of time. We would like to ensure that high priority tasks are not delayed more than one unit of time per set of semaphores used by that task as a result of lock holding by lower priority tasks.

We have investigated three real-time synchronisation algorithms for uniprocessors: the *priority inheritance*, the *ceiling protocol* and the *semaphore dependency protocol*. All of them use the preemptive mechanism and the priority inheritance mechanism. In this chapter we assume that there is the same number of processors as tasks. Unlike in uniprocessor cases, the preemptive mechanism and the priority inheritance mechanism lose their meaning in such ideal multiprocessor cases, consequently we need to look for another effective mechanism.

In this chapter, we propose a new real-time synchronisation protocol for multiprocessor systems, which we call the *semaphore plug-in method.* The sources of semaphore blocking are semaphore nests and semaphore chains, on which we can concentrate in an ideal situation. We intensively discuss how the *semaphore plug-in* method provides solutions to avoid opportunities for semaphore blocking caused by both semaphore nests and semaphore chains, based on the principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"*.

However the plug-in method is only given for purposes of arguments, as a step on the way to the later colour semaphores and future discussion. It is not intended for practical use because it can partially solve deadlock avoidance, but is still subject to deadlock and sometimes plugging in semaphores introduces the possibility of deadlock.

We assume that every semaphore V primitive wakes up the highest task in its semaphore queue. This assumption is not the ordinary definition of the V primitive that usually uses a time order based semaphore queue. However it is important for real-time properties because the exploited V primitive uses a priority descendent ordered semaphore queue. This subject will be discussed in more detail in Section 4.1.2 in relation to semaphore list operations.

## 3.3 The Semaphore Plug-in Method

The definition of dependency is the same that used by the dependency protocol. For each semaphore held, and for each priority level higher than the native priority of the holder (i.e. before any increase due to priority inheritance), the runtime system keeps a list of semaphores dependent on the held semaphore, in the following sense:

| *The Definition of Dependency* |
|---|
| $S_k$ is dependent on $S_i$ at priority $P$ if there is a task with priority $P$ which will seek to lock $S_i$ while holding $S_k$.<br>The dependency list of $S_i$ is static and applies while $S_i$ is being held.<br>We call $S_i$ the *parent semaphore* denoted $S_p$ or *GUARDIAN* and<br>$S_k$ as the *dependent semaphore* denoted $S_d$ or *WARD.* |

$\mathsf{D}S_i = [(S_j, P_k)]$ means that $S_i$ is locked, the element of the dependency list of the dependency list, $(S_j, P_k)$, a pair of a semaphore and priority, is active.

We use the concept of the *working unit* denoted as 'DO-WORK' to refer to the interval when a task is doing something useful rather than doing complicated work with nested semaphores, which means the innermost part of the code in nested semaphores.

Informally, the simplest form of the semaphore *plug-in* method is described as follows. The semaphore at the lower priority task into which the P and V primitives' pairing of a semaphore is plugged in is limited to the outermost semaphore in a semaphore nest. This limitation simplifies the problem, and make the description of the *plug-in* method easier.

| *Informal Description for the Plug-in Method* |
|---|
| A task with priority $P$ is about to claim $S_1$ which is the outermost semaphore of a semaphore nest, in the circumstances where $S_2$ is on the dependency list of $S_1$ at a priority greater than $P$.<br>Instead of going "P($S_1$) DO-WORK V($S_1$)",<br>"$\boxed{\mathrm{P}(S_2)}$ P($S_1$) $\boxed{\mathrm{V}(S_2)}$ DO-WORK V($S_1$)"<br>will be adopted. |

From the definition of *semaphore dependency*, a *merged list* can be derived. From that definition, we remark that if $k > i$, then $P_k < P_i$.

| *The Pure Plug-in Method* |
|---|
| We work out a list $L_{ij}$, in the acquisition order, elements of which are the semaphores that are dependent on $S_i$ at $P_j$, that is those which are acquired by some task running at priority $P_j$ after $S_i$ locking.<br>When a task attempts to lock $S_i$ which is the outermost semaphore of the nest at $P_k$, the system merges all the $L_{ij}$ for which $k > j$.<br>This represents all the semaphores that depend on $S_i$ at priorities greater than $P_k$.<br>We call the merged list $LL_{ik}$ and plug in the following semaphore pairing.<br>$\boxed{\wp_k}$ $T_k$: $\boxed{\mathrm{P}(LL_{ik})}$ P($S_i$) $\boxed{\mathrm{V}(LL_{ik})}$ DO-WORK V($S_i$)<br>We call $S_i$ which is the target semaphore to be plugged in the *point semaphore*.<br>This will have the effect that an attempt to claim a semaphore that is in use<br>by a lower priority task than that asking will cause a delay of at most one DO-WORK. |

We will show many examples of how the behaviour of systems in the ordinary cases and the *plug-in* method applied cases is different, and investigate the features of the *plug-in* method from the point of view of real-time synchronisation. From this investigation an ideal situation, it is hoped to obtain insight into more realistic real-time synchronisation. As suggested in [**Wirth77**], real-time programming is complicated. Therefore in order to find the clue to invent real-time synchronisations should be considered from the ideal situation rather than starting from more realistic, but such complicated situations.

Before giving examples, we describe the basic concept of the *plug-in* method.

## 3.4 The Basic Concept of the Plug-in Method

As will be shown in subsequent examples, the semaphore plug-in method can be informally summarised as follows:
*"If a high priority task is going to claim a nest of semaphores, we should ensure a low priority task avoid locking any inner semaphore of the nest while the high priority task may be locking an outer semaphore. We can do this by making the low priority task be forced to avoid locking the outer semaphore as well as the inner."*

To realise the above, we should arrange the following:

1. the P and V primitive pairing of a semaphore which may be held by a higher task is plugged into the semaphore at lower tasks, which is the inner nested semaphore.

2. the plug-in semaphore can prevent the lower task from locking the inner semaphore, otherwise the higher task will be blocked.

Events in the plug-in method are expressed by the following three-tuple as follows:
( $t^i_j \in$Time, $P_i \in$Priority, $S_k \in$ Semaphore)
In $t^i_j \in$ Time, the super script, $i$, of $t^i_j$ expresses a task number which is just the same as the subscript, $i$, of $P_i \in$Priority. The subscript, $j$, of $t^i_j$ is the event number, which indicates when task $T_i$ gets a semaphore. The nesting of semaphores is such that if $S_k, S_l \in$Semaphore and $S_l$ is inner than $S_k$, the nest is expressed as $S_k \prec S_l$. We can define the event order, thus:
$(t^{i_m}_{j_m}, P_{i_m}, S_{k_m}) \lhd (t^{i_n}_{j_n}, P_{i_n}, S_{k_n}) \Leftrightarrow t^{i_m}_{j_m} < t^{i_n}_{j_n}$

The basic case, in which the nest of semaphores is limited to two can be illustrated by diagrams with priority as a vertical axis and time as a horizontal axis. The basic situations in the plug-in method are classified as follows: *inward*, *backward*, *forward*.

In the *inward* case, the higher task has the outer semaphore (the dependent semaphore $S_d$) of the double nest, but before it gets the inner semaphore (the parent semaphore $S_p$), the lower task claims this inner semaphore. Using the three events that the higher task has got the outer semaphore (the dependent semaphore $S_d$), and will claim the inner semaphore (the parent semaphore $S_p$), at the same time, the lower task claims this inner semaphore. The *inward* situation is illustrated in Figure 3.1 on the next page. The shape of the triangle is similar to an *English plug*, therefore we call this method the *plug-in* method. In the *inward* case, the plug-in method works ideally so that the lower task is blocked by the plugged-in semaphore, the higher task can get the inner semaphore (the parent semaphore $S_p$) instead of being blocked by the lower task which might get this inner semaphore. In place of the higher task, the lower task is blocked by the outer semaphore, which prevents it from locking the inner semaphore which would cause blocking of the higher task. In the *inward* case, $t^0_1 < t^1_1 < t^0_2$. The *inward* case is expressed as:

$\boxed{(t^0_1, P_0, S_0) \lhd (t^1_1, P_1, S_1) \lhd (t^0_2, P_0, S_1)}$

In the *backward* case shown in Figure 3.2 on the next page, the higher task has both the outer semaphore (the dependent semaphore $S_d$) and the inner semaphore (the parent semaphore $S_p$) of the double nest, before the lower task claims the inner semaphore (the parent semaphore $S_p$). In this case, the lower task is blocked on the plugged-in semaphore (the outer semaphore) instead of the original semaphore (the inner semaphore) which avoids side effects for other synchronisations by grasping the original. If we only focus on the blocking of the plugged-in semaphore, we find it appears to cause unnecessary blocking, but anyway the lower task is bound to be blocked on the inner semaphore which the higher task has already locked. In this case, $t^0_1 < t^0_2 < t^1_1$. The *backward* case is expressed as:

$\boxed{(t^0_1, P_0, S_0) \lhd (t^0_2, P_0, S_1) \lhd (t^1_1, P_1, S_1)}$

$$\begin{tikzcd}
(t_1^0,\, P_0,\, S_0) \arrow[rr] \arrow[dr] & & (t_2^0,\, P_0,\, S_1) \\
& \boxed{\mathrm{P}(S_0)}\ (t_1^1,\, P_1,\, S_1)\ \boxed{\mathrm{V}(S_0)} \arrow[ur] &
\end{tikzcd}$$

Figure 3.1: Plug-in Triangle: Inward

$$\begin{tikzcd}
(t_1^0,\, P_0,\, S_0) \arrow[r] \arrow[drr] & (t_2^0,\, P_0,\, S_1) & \\
& & \boxed{\mathrm{P}(S_0)}\ (t_1^1,\, P_1,\, S_1)\ \boxed{\mathrm{V}(S_0)} \arrow[ul]
\end{tikzcd}$$

Figure 3.2: Plug-in Triangle: Backward

$$\begin{tikzcd}
& (t_1^0,\, P_0,\, S_0) \arrow[r] \arrow[dl] & (t_2^0,\, P_0,\, S_1) \\
\boxed{\mathrm{P}(S_0)}\ (t_1^1,\, P_1,\, S_1)\ \boxed{\mathrm{V}(S_0)} \arrow[urr] & &
\end{tikzcd}$$

Figure 3.3: Plug-in Triangle: Forward

In the *forward* case shown Figure 3.3 on the preceding page, the lower task claims the inner semaphore (the parent semaphore $S_p$) before the higher task claims the outer semaphore (the dependent semaphore $S_d$. The lower task goes through the plugged-in semaphore pair and locks the original, the inner semaphore (the parent semaphore $S_p$). The *forward* case does not work for our purpose, but when we consider a more realistic mutual exclusion case, it sometimes works quite well. As will be shown in example 4 in Section 3.5, the plugged-in semaphore produces a relatively useful side effect; the lower task is blocked on the plugged-in semaphore, which prevents the original semaphore (the parent semaphore) from blocking the higher task:

•*Plug-in for example 4: relatively beneficial case*
$T_0$ = {......,P($S_0$),..,P($S_1$),..,V($S_1$),..,V($S_0$),.........................}
$T_1$ = {....,P($S_0$),P($S_1$),V($S_0$), DO-WORK1 ,V($S_1$),............}
$T_2$ = {..,P($S_0$), DO-WORK2 ,V($S_0$),..................................}
$(P_0 > P_1 > P_2)$

This sort of side effect benefit is relative. For example, if the duration of DO-WORK1 is very short, the blocking of $T_1$ might be unnecessary. This example shows a typical mutual exclusion case using semaphores, so the *plug-in* method works relatively well in the *forward* case.

However, the plugged-in semaphore pair produces bad side effects for other synchronisations, as shown in example 8 in Section 3.5.

This is because the plugged-in semaphore pair is set *downward* from the higher task. In this case, $t_1^1 < t_1^0 < t_2^0$. The *forward* case is expressed as:

$\boxed{(t_1^1, P_1, S_1) \lhd (t_1^0, P_0, S_0) \lhd (t_2^0, P_0, S_1)}$

## 3.5 Examples of the Plug-in Method

In this section, we examine a variety of types of synchronisation combinations to investigate the special features that will appear by introducing the *plug-in* method. Several examples come from previous uniprocessor cases. We will show in order as follows:

1. an example program which presents a type of synchronisation
2. the behaviour of the program
3. its merged list by analysing dependencies of the program
4. the modified program to which the *plug-in* method has been applied
5. the behaviour of the *plug-in* version of the program

We assume that each task $T_i$ is assigned to one processor $\wp_i$, which is fixed.

—— *Example 1: Comparison with the ceiling protocol*——
$\boxed{\wp_0}$ $T_0$ = {.........,P($S_0$),..........,V($S_0$),..}

$\boxed{\wp_1}$ $T_1$ = {.......,P($S_1$),..........,V($S_1$),....}
$\boxed{\wp_2}$ $T_2$ = {....,P($S_1$),..........,V($S_1$),.......}
$\boxed{\wp_3}$ $T_3$ = {..,P($S_0$),..........,V($S_0$),.........}
($P_0 > P_1 > P_2 > P_3$)

Table 3.1: The merged list table of example 1

| *Priority* | $P_3$ | $P_2$ | $P_1$ | $P_0$ |
|---|---|---|---|---|
| $S_0$ | $\phi$ | $\phi$ | $\phi$ | $\phi$ |
| $S_1$ | $\phi$ | $\phi$ | $\phi$ | $\phi$ |

The merged list is shown in Table 3.1. The example 1 has no dependency, therefore the *plug-in* method does not work in this system. A lower task which locks a semaphore will block higher tasks. If the ceiling protocol were applied to multiprocessors, after $T_3$ locks $S_0$ with the ceiling value $P_0$, $T_2$ and $T_1$ are blocked even though there are enough processors. Processors would be starved for tasks. This example shows the main drawback of the ceiling protocol for multiprocessors.

——*Example 2: Typical beneficial case for Plug-in*——
$\boxed{\wp_0}$ $T_0$ = {..........,P($S_1$),.......,V($S_1$),....................................}
$\boxed{\wp_1}$ $T_1$ = {........,P($S_1$),.,P($S_0$),..DO-WORK1..,V($S_0$),V($S_1$),..}
$\boxed{\wp_2}$ $T_2$ = {............,P($S_0$),...DO-WORK2....,V($S_0$),................}
($P_0 > P_1 > P_2$)

Table 3.2: The merged list table of example 2

| *Priority* | $P_2$ | $P_1$ | $P_0$ |
|---|---|---|---|
| $S_0$ | $\phi$ | $S_1$ | $\phi$ |
| $S_1$ | $\phi$ | $\phi$ | $\phi$ |

Example 2 is a typical example of those illustrated in Section 3.1. $T_0$ should wait for two working units, DO-WORK1 and DO-WORK2. The merged list of example 2 is shown in Table 3.2. The $S_1$ pair is plugged in to $S_0$ of task $T_2$.

•*Plug-in for example 2*
$\boxed{\wp_0}$ $T_0$ = {............,P($S_1$),.......,V($S_1$),.........................................}
$\boxed{\wp_1}$ $T_1$ = {........,P($S_1$),.,P($S_0$),..DO-WORK1..,V($S_0$),V($S_1$),........}
$\boxed{\wp_2}$ $T_2$ = {............,P($S_1$),P($S_0$),V($S_1$),...DO-WORK2....,V($S_0$),..}
( $P_0 > P_1 > P_2$)

1. $T_0$, $T_1$ and $T_2$ are initiated. $T_1$ is initiated and locks $S_1$.
2. $T_0$ is blocked by $S_1$.
3. $T_2$ is blocked by plugged in $S_1$.
4. $T_1$ locks $S_0$, executes DO-WORK1 and releases $S_0$ and $S_1$.
5. $T_0$ restarts and locks $S_1$. $T_2$ is still blocked by $S_1$.

$T_1$ is not blocked by $S_0$ held by lower task $T_2$, and $T_0$ need not be blocked for two working units, but waits for only one working unit, DO-WORK1. The plugged-in semaphore $S_1$ prevents $T_2$ from locking $S_1$ and realises the principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"*. The lowest task $T_2$ is a new victim, because it need not be blocked by $S_1$ for multiprocessors after $T_1$ releases $S_0$.

—— *Example 3*——

$\boxed{\wp_0}$ $T_0 = \{$.....,P$(S_0)$,........,V$(S_0)$,............................$\}$
$\boxed{\wp_1}$ $T_1 = \{$......,P$(S_1)$,..,P$(S_2)$,..,V$(S_2)$,..,V$(S_1)$,........$\}$
$\boxed{\wp_2}$ $T_2 = \{$........,P$(S_2)$,..,P$(S_1)$,..,V$(S_1)$,..,V$(S_2)$,......$\}$
$(P_0 > P_1 > P_2)$

The dependency lists are: $\mathsf{D}S_1 = [(S_2, P_2)]$,$\mathsf{D}S_2 = [(S_1, P_1)]$

1. $T_0$ is initiated, claims $S_0$ and locks it. $T_1$ is initiated, claims $S_1$ and locks it. $T_2$ is initiated, claims $S_2$ and locks it.
2. $T_1$ claims $S_2$ and is blocked. $T_2$ claims $S_1$ and is blocked. This causes deadlock between $T_1$ and $T_2$.

So the system in example 3 causes deadlock. The merged list of example 3 is shown in Table 3.3. $S_1$ pairing is plugged-in $S_2$ of task $T_2$.

Table 3.3: The merged list table of example 3

| *Priority* | $P_2$ | $P_1$ | $P_0$ |
|---|---|---|---|
| $S_0$ | $\phi$ | $\phi$ | $\phi$ |
| $S_1$ | $S_2$ | $\phi$ | $\phi$ |
| $S_2$ | $\phi$ | $S_1$ | $\phi$ |

•*Plug-in for example 3:Deadlock Avoidance in the Inward Case*

$\boxed{\wp_0}$ $T_0 = \{$.....,P$(S_0)$,........,V$(S_0)$,.........................................$\}$
$\boxed{\wp_1}$ $T_1 = \{$......,P$(S_1)$,..,P$(S_2)$,..,V$(S_2)$,..,V$(S_1)$,.......................$\}$
$\boxed{\wp_2}$ $T_2 = \{$........,P$(S_1)$,P$(S_2)$,V$(S_1)$,..,P$(S_1)$,..,V$(S_1)$,..,V$(S_2)$,.$\}$
$(P_0 > P_1 > P_2)$

1. $T_0$ is initiated, claims $S_0$ and locks it. $T_1$ is initiated, claims $S_1$ and locks it.

2. $T_2$ is initiated, claims $S_1$ and is blocked. It avoids the deadlock between $T_1$ and $T_2$ in this timing case.

The plugged-in semaphore $S_1$ of $T_2$ can prevent $T_1$ being blocked by $S_2$. This seems to work as *predictable blocking*. It depends when $S_2$ is claimed. In this case, the plugged in semaphore $S_1$ of $T_2$ in the inward case avoids the deadlock. However this is not applicable in the forward case.

The plug-in method is subject to deadlock. The ceiling protocol provides deadlock avoidance, which is its major advantage. We intend to pursue this deadlock avoidance as a minor goal in this thesis. However deadlock-free is very important for practical use, therefore the plug-in method is given for purpose of arguments.

——*Example 4*——

$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),..,P($S_1$),...,V($S_1$),..,V($S_0$),............}

$\boxed{\wp_1}$ $T_1$ = {.....,P($S_1$),.DO-WORK1.......,V($S_1$),................}

$\boxed{\wp_2}$ $T_2$ = {...,P($S_0$),..DO-WORK2....,V($S_0$),....................}

$(P_0 > P_1 > P_2)$

The dependency list is: $\mathsf{D}S_1 = [(S_0, P_0)]$

1. $T_2$ is initiated, claims $S_0$ and locks it. $T_1$ is initiated, claims $S_1$ and locks it. $T_0$ is initiated, claims $S_0$ and is blocked.
2. $T_2$ releases $S_0$. $T_0$ restarts and is blocked by $S_1$.

The highest task is blocked twice, so the maximum blocking time is two working units of $T_1$ and $T_2$, DO-WORK1 + DO-WORK2. The merged list of example 4 is shown in Table 3.4. $S_0$ pairing is plugged in to $S_1$ of task $T_1$.

Table 3.4: The merged list table of example 4

| *Priority* | $P_2$ | $P_1$ | $P_0$ |
|---|---|---|---|
| $S_0$ | $\phi$ | $\phi$ | $\phi$ |
| $S_1$ | $\phi$ | $\phi$ | $S_0$ |

•*Plug-in for example 4:The relative beneficial case*

$\boxed{\wp_0}$ $T_0$ = {......,P($S_0$),..,P($S_1$),..,V($S_1$),..,V($S_0$),.........................}

$\boxed{\wp_1}$ $T_1$ = {....,P($S_0$),P($S_1$),V($S_0$), DO-WORK1 ,V($S_1$),............}

$\boxed{\wp_2}$ $T_2$ = {..,P($S_0$), DO-WORK2 ,V($S_0$),..................................}

$(P_0 > P_1 > P_2)$

1. $T_2$ is initiated, claims $S_0$ and locks it. $T_1$ is initiated, claims $S_0$ and is blocked. $T_0$ is initiated, claims $S_0$ and is blocked.
2. $T_2$ releases $S_0$. $T_0$ restarts.

The plug-in semaphore $S_0$ blocks middle priority task $T_1$. It can avoid the second blocking of the highest task $T_0$. The highest task is blocked for only one working unit, DO-WORK1, as compared to two working units before applying the *plug-in* method. This gives the best benefit for the highest task in the environment of a typical mutual exclusion case. This case shows the plug-in method works well according to our target of reducing the blocking time of the highest task to at most one working unit. But this effect is relative because if DO-WORK2 is much longer than DO-WORK1, $T_1$ could run through it before it may block $T_0$. However even though the effect is relative, the highest task can get the best benefit and its blocking time is within one working unit.

•—*Plug-in for a variant of example 4:The relative case*—

$\boxed{\wp_0}$ $T_0$ = {........,P($S_0$),....DO-WORK0............,V($S_0$),.................}
$\boxed{\wp_1}$ $T_1$ = {......,P($S_0$),..,P($S_1$),..DO-WORK1..,V($S_1$),..,V($S_0$),.....}
$\boxed{\wp_2}$ $T_2$ = {....,P($S_0$),P($S_1$),V($S_0$),..DO-WORK2.. ,V($S_1$),............}
$\boxed{\wp_3}$ $T_3$ = {..,P($S_0$), DO-WORK3 ,V($S_0$),.....................................}
$(P_0 > P_1 > P_2 > P_3)$

The above example is slightly different from example 4. In more detail:

1. $T_3$ is initiated, claims $S_0$ and locks it. $T_2$, $T_1$ and $T_0$ are initiated, claim $S_0$ and are blocked.
2. After executing DO-WORK3, $T_3$ releases $S_0$. $T_0$ restarts and executes DO-WORK0. $T_0$ releases $S_0$ and $T_1$ restarts.
3. After executing DO-WORK1, $T_1$ releases $S_0$, then $T_2$ can start.

$T_2$ has to wait for three working units in this case, as compared with no working units if not plugged-in. This does not give any benefit to the highest task, but the second highest task $T_1$ can get some benefit in reducing the blocking duration, if not plugged-in, $T_2$ could lock $S_1$ and $T_1$ might be blocked on $S_1$. This effect is relative. If DO-WORK3 + DO-WORK0 is longer than DO-WORK2, then $T_2$ could go through DO-WORK2 and $T_2$ would be blocked unnecessarily. This is a relative disadvantage of the *plug-in* method.

•—*Plug-in for a variation of example 4*—

$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),..,P($S_1$),..,V($S_1$),..,V($S_0$),...........................}
$\boxed{\wp_1}$ $T_1$ = {....,P($S_0$),P($S_1$),V($S_0$), DO-WORK1 ,V($S_1$),................}
$\boxed{\wp_2}$ $T_2$ = {......,P($S_0$), DO-WORK2 ,V($S_0$),..................................}
$(P_0 > P_1 > P_2)$

The situation is slightly changed from the original example, in which case an interesting phenomenon can be observed:

1. $T_1$ is initiated and locks $S_0$ and just claim $S_1$.
2. $T_2$ is initiated and is blocked on $S_0$.
3. $T_0$ is initiated and is blocked on $S_0$.
4. $T_1$ releases $S_0$. The highest task $T_0$ restarts and locks $S_0$.

This case happens very rarely. The plug-in method avoids the highest task $T_0$ being blocked by the lowest priority task $T_2$.

——*Example 5a*——

$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),.....,P($S_1$),...,V($S_1$),..,V($S_0$),...............................}

$\boxed{\wp_1}$ $T_1$ = {.......,P($S_2$),......,P($S_1$),..DO-WORK1..,V($S_1$),V($S_2$),............}

$\boxed{\wp_2}$ $T_2$ = {...............,P($S_1$),..DO-WORK2...,V($S_1$),.............................}

($P_0 > P_1 > P_2$)

The dependency list is: $\mathsf{D}S_1 = [(S_0, P_0), (S_2, P_1)]$

Table 3.5: The merged list table of example 5

| *Priority* | $P_2$ | $P_1$ | $P_0$ |
|---|---|---|---|
| $S_0$ | $\phi$ | $\phi$ | $\phi$ |
| $S_1$ | $\phi$ | $S_2$ | $S_0$ |
| $S_2$ | $\phi$ | $\phi$ | $\phi$ |

1. $T_0$, $T_1$, $T_2$ are initiated.
2. $T_0$ locks $S_0$. $T_1$ locks $S_2$. $T_2$ locks $S_1$.
3. $T_0$ claims $S_1$ and is blocked. $T_1$ claims $S_1$ and is blocked.
4. $T_2$ releases $S_1$. $T_0$ can restart.

$T_0$ waits for one working unit, DO-WORK2. The merged list table is shown in Table 3.5. The merged list $LL_{12}$ = {{$S_2$}, {$S_0$}}. Example 5a shows why the merged list is required, because every multiple task of higher priority has the dependent semaphores which should be plugged-in to the point semaphore of a lower task. In the definition of the *plug-in* method, the structure of the merged list has not yet been defined clearly. We define now the detailed structure of the merged list. The semaphore nest of the dependent semaphores of priority antecedent ordered tasks should be plugged in.

| *Structure of Plugged-in Semaphore Pairing* |
|---|
| $\boxed{\mathrm{P}(LL_{ik})}$ $\mathrm{P}(S_i)$ $\boxed{\mathrm{V}(LL_{ik})}$ is the abbreviation of: |
| $\boxed{\mathrm{P}(L_{ik-1})\ldots\mathrm{P}(L_{i1})\mathrm{P}(L_{i0})}$ $\mathrm{P}(S_i)$ $\boxed{\mathrm{V}(L_{i0})\mathrm{V}(L_{i1})\ldots\mathrm{V}(L_{ik-1})}$ |

Using the detailed structure of the *pure plug-in* method, $S_1$ of $T_1$ is not the outermost semaphore; it is not plugged-in by $S_0$. The blocking duration of $T_0$ contains no working units, and that of $T_1$ becomes one or no working units.

•—*Plug-in for Example 5a*—

$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),......,P($S_1$),...,V($S_1$),..,V($S_0$),.......................................}

$\boxed{\wp_1}$ $T_1$ = {.......,P($S_2$),.......,P($S_1$),...,V($S_1$),V($S_1$),..........................................}
$\boxed{\wp_2}$ $T_2$ = {...........,P($S_2$),P($S_0$),P($S_1$),V($S_0$),V($S_2$),..DO-WORK2...,V($S_1$),...}
$(P_0 > P_1 > P_2)$

1. $T_0$, $T_1$, $T_2$ are initiated.
2. $T_0$ locks $S_0$. $T_1$ locks $S_2$.
3. $T_2$ is blocked by the semaphore $S_2$.
4. $T_0$ claims $S_1$ and locks. $T_1$ claims $S_1$ and is blocked.
5. $T_0$ releases $S_1$. $T_1$ can restart.

By the plugged-in semaphore nest of the dependent semaphores, $S_1$ and $S_0$, of priority antecedent ordered tasks, $T_1$ and $T_0$, both the higher tasks $T_1$ and $T_0$ need not be blocked by $S_1$. The merged list $LL_{ik}$ is a good mechanism to realise the goal that *"a lower task is forced to avoid locking a semaphore which would block multiple higher tasks"*.

—*Example 5b*—
$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),.....,P($S_1$),...,V($S_1$),..,V($S_0$),.....................................}
$\boxed{\wp_1}$ $T_1$ = {..............,P($S_2$),......,P($S_1$),..DO-WORK1..,V($S_1$),V($S_2$),...........}
$\boxed{\wp_2}$ $T_2$ = {............,P($S_1$),..DO-WORK2...,V($S_1$),......................................}
$(P_0 > P_1 > P_2)$

For this variation of example 5a, the result of plugging is just same as that of example 5a.

• —*Plug-in for Example 5b*—
$\boxed{\wp_0}$ $T_0$ = {.....,P($S_0$),.....,P($S_1$),...,V($S_1$),..,V($S_0$),..........................................}
$\boxed{\wp_1}$ $T_1$ = {............,P($S_2$),......,P($S_1$),..DO-WORK1..,V($S_1$),V($S_2$),...............}
$\boxed{\wp_2}$ $T_2$ = {..........,P($S_2$),P($S_0$),P($S_1$),V($S_0$),V($S_2$),..DO-WORK2...,V($S_1$),...}
$(P_0 > P_1 > P_2)$

1. $T_0$, $T_1$, $T_2$ are initiated.
2. $T_0$ locks $S_0$. $T_2$ locks $S_2$.
3. $T_2$ is blocked on the semaphore $S_0$.
4. $T_0$ claims $S_1$ and locks. $T_1$ claims $S_2$ and is blocked.
5. $T_0$ releases $S_1$,$S_0$. $T_2$ can restart, but $T_1$ is still blocked.

In this case, the middle priority task $T_1$ is disadvantaged, because the lower task $T_2$ holds $S_2$ while being blocked on $S_0$. This disadvantage will be solved in the next chapter by introducing *colour* semaphores which issue a semaphore list simultaneously instead of issuing the members separately.

——*Example 6: A Semaphore Chain* ——
$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),.,P($S_1$),....,V($S_1$),..,V($S_0$),......................................}
$\boxed{\wp_1}$ $T_1$ = {............,P($S_1$),..,P($S_2$),..DO-WORK1..,V($S_2$)V($S_1$),.............}

$\boxed{\wp_2}$ $T_2 = \{$...................,P($S_2$),...DO-WORK2.....,V($S_2$),........................$\}$
$(P_0 > P_1 > P_2)$

The dependency lists are:
$\mathsf{D}S_1 = [(S_0, P_0)]$, $\mathsf{D}S_2 = [(S_1, P_1)]$

1. $T_0$, $T_1$, $T_2$ are initiated.
2. $T_0$ locks $S_0$. $T_1$ locks $S_1$.
3. $T_0$ claims $S_1$, but is blocked. $T_2$ locks $S_2$.
4. $T_1$ claims $S_2$, but is blocked.
5. $T_2$ does DO-WORK2 and releases $S_2$. $T_1$ restarts, executes DO-WORK1 and releases $S_1$. Then $T_0$ can restart.

$T_0$ should be blocked for two working units. The merged list is shown in Table 3.6.

Table 3.6: The merged list table of example 6

| *Priority* | $P_2$ | $P_1$ | $P_0$ |
|---|---|---|---|
| $S_0$ | $\phi$ | $\phi$ | $\phi$ |
| $S_1$ | $\phi$ | $\phi$ | $S_0$ |
| $S_2$ | $\phi$ | $S_1$ | $\phi$ |

•*Plug-in for Example 6: A Semaphore Chain*
$\boxed{\wp_0}$ $T_0 = \{$.......,P($S_0$),.,P($S_1$),....DO-WORK0.,V($S_1$),.,V($S_0$),....................................$\}$
$\boxed{\wp_1}$ $T_1 = \{$.............,P($S_0$)P($S_1$),V($S_0$),..,P($S_2$),..DO-WORK1..,V($S_2$)V($S_1$),...............$\}$
$\boxed{\wp_2}$ $T_2 = \{$...................,P($S_1$)P($S_2$),V($S_1$),...DO-WORK2.....,V($S_2$),..........................$\}$
$(P_0 > P_1 > P_2)$

1. $T_0$, $T_1$, $T_2$ are initiated.
2. $T_0$ locks $S_0$. $T_1$ claims $S_0$, but is blocked.
3. $T_0$ locks $S_1$. $T_2$ claims $S_1$, but is blocked.
4. $T_0$ executes DO-WORK0 and releases $S_1$.
5. $T_2$ can restart and locks $S_2$.
6. $T_0$ releases $S_0$.
7. $T_1$ can restart, locks $S_1$, claims $S_2$ and is blocked by $S_2$.

The main sources of semaphore blocking are semaphore nests and semaphore chains. The essential function of the *plug-in* method in dealing with semaphore chains is that it cuts the chains. This example suggests possible drawbacks for the *plug-in* method to cut a semaphore chain, that is:
1. the plugging of $S_1$ into $T_2$ does not apply to the middle task $T_1$.
2. the first block of $T_2$ is thus unnecessary.

However these drawbacks do not influence the benefits of the higher task by the *plug-in* method. To avoid this problem, the rule that once a semaphore has been plugged in, the semaphore cannot plug into the lower task in a semaphore chain might be considered. The colour semaphore which is applied to the *plug-in* method will solve this drawback; please see Section 4.3.4.

——*Example 7* ——

$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),..,P($S_1$),...,V($S_1$),..,V($S_0$),....}
$\boxed{\wp_1}$ $T_1$ = {.....,P($S_1$),........,V($S_1$),..........................}
$\boxed{\wp_2}$ $T_2$ = {...,P($S_1$),........,V($S_1$),.............................}
$(P_0 > P_1 > P_2)$

The dependency list is: $\mathsf{D}S_1 = [(S_0, P_0)]$

This example is very similar to example 4, in which the *plug-in* method has a beneficial effect; but in this case, the *plug-in* method is harmful. The merged list is exactly the same as that of example 4.

•*Plug-in for example 7*

$\boxed{\wp_0}$ $T_0$ = {......,P($S_0$),..,P($S_1$),..,V($S_1$),..,V($S_0$),.........................}
$\boxed{\wp_1}$ $T_1$ = {....,P($S_0$),P($S_1$),V($S_0$), DO-WORK1 ,V($S_1$),............}
$\boxed{\wp_2}$ $T_2$ = {..,P($S_0$),P($S_1$),V($S_0$) DO-WORK2 ,V($S_1$),...............}
$(P_0 > P_1 > P_2)$

1. $T_0$, $T_1$, $T_2$ are initiated. $T_2$ locks $S_1$ and executes DO-WORK2. $T_1$ locks $S_0$ and is blocked on $S_1$. $T_0$ is blocked on $S_0$.
2. $T_2$ finishes DO-WORK2 and releases $S_1$.
3. $T_1$ restarts, locks $S_1$ and releases $S_0$
4. $T_0$ can restart and is blocked on $S_1$.

$T_0$ is blocked twice and waits for two working units, DO-WORK2 + DO-WORK1. This is because $T_1$ is blocked on $S_1$ holding $S_0$. This causes the worst case. These drawbacks which are described in example 7 can be solved by introducing the *colour semaphore* discussed in Chapter 4, Section 4.3.

## 3.6 Uniprocessor Applications

The semaphore plug-in method could also be applied to a uniprocessor case with priority inheritance. Because the dependency protocol uses expensive atomic actions, its application would be advantageous. The *plug-in* method could realise the dependency protocol function using the ordinary semaphore mechanism instead of implementing the expensive dependency protocol. But the problems of applying the *plug-in* method to uniprocessor cases as follows:

1. Multiprocessor solutions will not, in general, work on uniprocessors
2. Uniprocessor solutions may often be too conservative for multiprocessors

Using the *plug-in* method on example 1 in Section 2.5.1, it becomes as follows:

—*Example 1, Plug-in for Uniprocessors*—
$\boxed{\wp_0}$ $T_0$ = {........,P($S_0$),......,V($S_0$),.........................................}
$\boxed{\wp_0}$ $T_1$ = {....,P($S_2$),P($S_1$),V($S_2$),..,P($S_2$),..,V($S_2$),..,V($S_1$),..}
$\boxed{\wp_0}$ $T_2$ = {.,P($S_2$),..,P($S_1$),..,V($S_1$),..,V($S_2$)..........................}
$(P_0 > P_1 > P_2)$

We should find some algorithm similar to the semaphore plug-in method for uniprocessors. One approach which will be discussed in Section 3.8 is the *upward plug-in* method.

## 3.7 The Triple Nested Case

In previous sections, we considered double semaphore nest cases. Now we discuss a more complex case: a task using triply nested semaphores. The investigation of the triple nest can be extended to a more deeply nested semaphore case. For simplicity, we begin with an example which is shortened from a real case.

— *The Simplest Triple Nest*—
$\boxed{\wp_0}$ $T_0$ = {..,P($S_0$),..,P($S_1$),...,P($S_2$), DO-WORK0 ,V($S_2$)V($S_1$)V($S_0$),.}
$\boxed{\wp_1}$ $T_1$ = {.........,P($S_2$),.. DO-WORK1 ...............,V($S_2$),.......................}
( $P_0 > P_1$)

1. $T_0$ is initiated and locks $S_0$.
2. $T_1$ is initiated and locks $S_2$.
3. $T_0$ locks $S_1$, claims $S_2$ and is blocked.

The merged list of the simple triple nest example is shown in Table 3.7. It should be noted that the semaphore $S_2$ at priority $P_0$ should keep the order of nesting, as $S_0 \prec S_1$. The search program to look for dependencies should record each nest situation to create the merged list.

We should now discuss how to apply the *plug-in* method to this triple nest case. The event description is explained using Figure 3.4 on the next page. The situations, which are from the point of view of the lower task, are classified as follows:

1. In the *backward* case expressed by $\boxed{\mathrm{B}}$, no plugged-in semaphore pair is needed.
2. There are two *inward* cases expressed by $\boxed{\mathrm{I}_1}$, $\boxed{\mathrm{I}_2}$.
   (a) In the case of $\boxed{\mathrm{I}_2}$, plugging $S_1$ pair would work.

Table 3.7: The merged list table of triple nest (simple case)

| *Priority* | $P_1$ | $P_0$ |
|---|---|---|
| $S_0$ | $\phi$ | $\phi$ |
| $S_1$ | $\phi$ | $S_0$ |
| $S_2$ | $\phi$ | $S_0 \prec S_1$ |

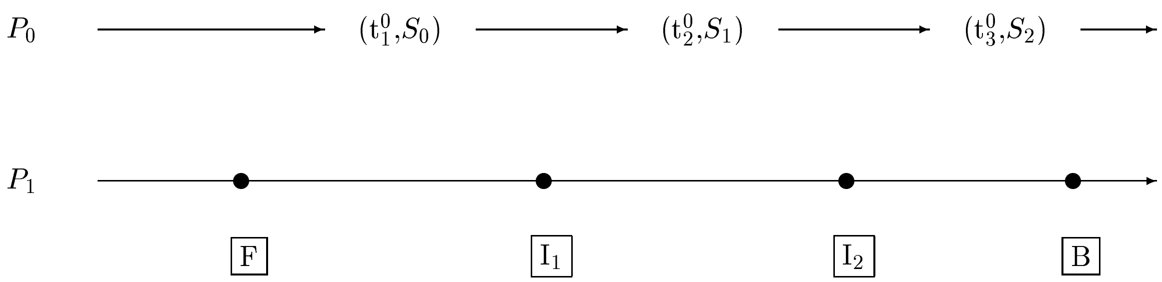


Figure 3.4: A Triple Nest Case

(b) In the case of $\boxed{\mathrm{I}_1}$, there may exist two approaches, (1) plugging the outermost semaphore $S_0$, (2) plugging double pairing of $S_0$ and $S_1$.

3. In the *forward* case expressed by $\boxed{\mathrm{F}}$, (1) plugging the outermost semaphore $S_0$, (2) plugging double pairing of $S_0$ and $S_1$.

The results of plugging by both approaches (1) and (2) are shown below. The behaviour of the system in each case is exactly the same.

•*Plug-in for Triple Nest by the approach(1)*
$\boxed{\wp_0}$ $T_0 = \{$...,P$(S_0)$,....,P$(S_1)$,...,P$(S_2)$,.......,V$(S_2)$,V$(S_1)$,V$(S_0)$..$\}$
$\boxed{\wp_1}$ $T_1 = \{$.....,$\boxed{\mathrm{P}(S_0)}$,P$(S_2)$,$\boxed{\mathrm{V}(S_0)}$,........,V$(S_2)$,......................$\}$
$(P_0 > P_1)$

•*Plug-in for Triple Nest by the approach(2)*
$\boxed{\wp_0}$ $T_0 = \{$...,P$(S_0)$,....,P$(S_1)$,...,P$(S_2)$,.......,V$(S_2)$,V$(S_1)$,V$(S_0)$..$\}$
$\boxed{\wp_1}$ $T_1 = \{$.....,$\boxed{\mathrm{P}(S_0),\mathrm{P}(S_1)}$P$(S_2)$,$\boxed{\mathrm{V}(S_0),\mathrm{V}(S_1)}$,........,V$(S_2)$,....$\}$
$(P_0 > P_1)$

1. $T_0$ is initiated and locks $S_0$.
2. $T_1$ is initiated claims $S_0$ and is blocked.

Table 3.8: The merged list table of triple nest (complex case)

| *Priority* | $P_3$ | $P_1$ | $P_1$ | $P_0$ |
|---|---|---|---|---|
| $S_0$ | $\phi$ | $\phi$ | $\phi$ | $\phi$ |
| $S_1$ | $\phi$ | $\phi$ | $\phi$ | $S_0$ |
| $S_2$ | $\phi$ | $\phi$ | $\phi$ | $S_0 \prec S_1$ |

### 3.7.1 Triple Nest Cases

A more typical example of triple nest cases would be as follows.

— *The More Complex Triple Nest*—
$\boxed{\wp_0}$ $T_0$ = {....,P($S_0$),..,P($S_1$),...,P($S_2$), DO-WORK ,V($S_2$)V($S_1$)V($S_0$),..}
$\boxed{\wp_1}$ $T_1$ = {................,P($S_2$),.. DO-WORK .............,V(S$_2$),..................}
$\boxed{\wp_2}$ $T_2$ = {..........,P($S_1$),.. DO-WORK ......,V($S_1$),................................}
$\boxed{\wp_3}$ $T_3$ = {..,P($S_0$),.. DO-WORK .....,V($S_0$),..........................................}
$(P_0 > P_1 > P_2 > P_3)$

1. $T_0$, $T_1$, $T_2$, $T_3$ are initiated and $T_3$ locks $S_0$. $T_0$ claims $S_0$ and is blocked. $T_2$ locks $S_1$. $T_1$ locks $S_2$.
2. $T_3$ releases $S_0$. $T_0$ restarts, claims $S_1$ and is blocked.
3. $T_2$ releases $S_1$. $T_0$ restarts, claims $S_2$ and is blocked
4. $T_1$ releases $S_2$. $T_0$ restarts and locks $S_2$.

The highest task $T_0$ is blocked for the three working units of every lower task, $T_1$, $T_2$, $T_3$. The merged list, shown in Table 3.8, is similar to that of Table 3.7. Adopting the approach(1):

•*Plug-in for The More Complex Triple Nest: Applying the approach (1)*
$\boxed{\wp_0}$ $T_0$ = {.....,P($S_0$),..,P($S_1$),...,P($S_2$),.. DO-WORK0 ..,V($S_2$)V($S_1$)V($S_0$),.}
$\boxed{\wp_1}$ $T_1$ = {..........,P($S_0$),P($S_2$),V($S_0$),.. DO-WORK1 ............,V($S_2$),...........}
$\boxed{\wp_2}$ $T_2$ = {........P($S_0$),P($S_1$),V($S_0$).. DO-WORK2 ......,V($S_1$),.....................}
$\boxed{\wp_3}$ $T_3$ = {...,P($S_0$),.. DO-WORK3 .....,V($S_0$),.............................................}
$(P_0 > P_1 > P_2 > P_3)$

1. $T_0$, $T_1$, $T_2$, $T_3$ are initiated and $T_3$ locks $S_0$. $T_0$ claims $S_0$ and is blocked. $T_2$ claims $S_0$ and is blocked. $T_1$ claims $S_0$ and is blocked.
2. $T_3$ releases $S_0$, $T_0$ restarts, then locks $S_1$, then $S_2$ and releases $S_2$,$S_1$, $S_0$.
3. $T_1$ restarts and releases $S_1$,$S_0$. $T_2$ restarts.

### 3.7.2 The Structure of the Merged List

In Section 3.3, the structure of the merged list $LL_{ij}$ and structure of semaphore pairings which will be plugged-in was not described in detail. Through the discussion of the triple nest cases, the structure of the merged list and plugging-in semaphores pairing should now become clearer.

In the merged list, the order of semaphores in the nest should be kept in the elements of the merged list. $S_k, S_l \in$Semaphore, $S_k$ is further out than $S_l$ in the nest, which is expressed as $S_k \prec S_l$.

| *Semaphore Pairings for Plug-in in the approach(1)* |
|---|
| If the nest consists of semaphores $S_0 \prec S_1 \prec, \ldots, \prec S_n$, then plug in the most outer semaphore $S_0$ pairing as follows: $\boxed{\mathrm{P}(S_0)}$ $\mathrm{P}(S_i)$ $\boxed{\mathrm{V}(S_0)}$ |

| *Semaphore Pairings for Plug-in in the approach(2)* |
|---|
| If the nest consists of semaphores $S_0 \prec S_1 \prec, \ldots, \prec S_n$, then plug in the semaphore pairing as follows: $\boxed{\mathrm{P}(S_0),\mathrm{P}(S_1),\ldots,\mathrm{P}(S_{i-1})}$ $\mathrm{P}(S_i)$ $\boxed{\mathrm{V}(S_0),\mathrm{V}(S_1),\ldots,\mathrm{V}(S_{i-})}$ |

Approach (2) is complex and disadvantageous because the excessive use of semaphores sometimes causes troublesome side effects. Approach (1) is simple and sufficient. This means that the outermost semaphore works essentially for plugging-in in the semaphore nest cases. Therefore we need not bother with quite so complicated a structure as in approach (2). We need only consider the outermost semaphore of multiple semaphore nest cases. But introducing the *colour semaphore* in the next chapter might give some benefit to approach (2).

## 3.8 The Upward Plug-in Method

The *plug-in* method works effectively in the *inward* case in Section 3.4, but does not work in the *forward* case. We need to find some mechanism by which the *plug-in* method works even in the *forward* case. Remembering Figure 2.1 on page 19, we can infer that the semaphore dependency protocol works for the *forward* case. In this example of the *forward* case, the parent semaphore $S_1$ is already locked by $T_1$, therefore $T_0$ should avoid locking the dependent semaphore $S_1$ so that higher tasks than $T_0$ can lock $S_1$. The principle of the *plug-in* method is that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"*. Therefore the plugging-in of the locking parent semaphore to the dependent semaphore of higher tasks would avoid locking for far higher tasks.

The principle of the *upward plug-in* method is as follows:

1. In the *forward* case, when a lower task has already locked the parent semaphore, the *downward plug-in* does not work.

2. A middle task should avoid locking the dependent semaphore which higher tasks are going to claim.

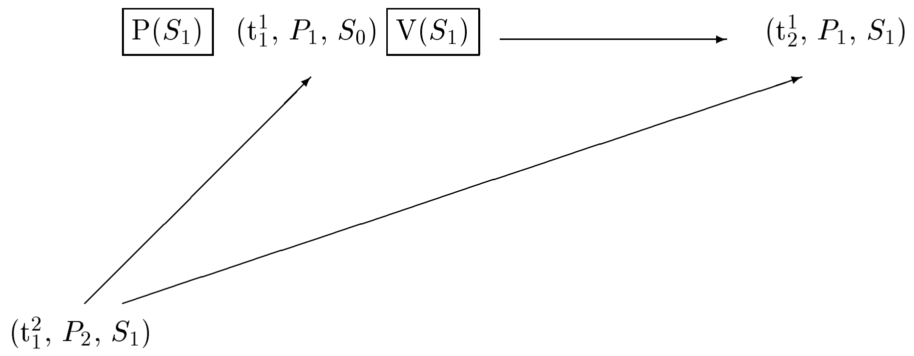


Figure 3.5: Plug-in Triangle: Upward

3. In the uniprocessor systems case, we can exploit the policy of *predictable blocking*. The *upward plug-in* method simulates the dependency protocol for multiprocessors in the ideal condition.

We can now look at the example of how the system behaves when the upward plug-in method is applied.

—*Forward Application of Dependency Protocol*—
$\boxed{\wp_0}$ $T_0 = \{..........,\mathrm{P}(S_0),..............,\mathrm{V}(S_0),....................\}$
$\boxed{\wp_1}$ $T_1 = \{.......,\mathrm{P}(S_0),..,\mathrm{P}(S_1),.....,\mathrm{V}(S_1),\mathrm{V}(S_0)..........\}$
$\boxed{\wp_2}$ $T_2 = \{.....,\mathrm{P}(S_1),....................,\mathrm{V}(S_1),...................\}$
$(P_0 > P_1 > P_2)$

The dependency protocol for multiprocessors will be similar to the *forward* case of the *plug-in* method, therefore the example can be rewritten as follows:

•—*Upward Plug-in*—
$\boxed{\wp_0}$ $T_0 = \{..........,\mathrm{P}(S_0),..............,\mathrm{V}(S_0),............................................\}$
$\boxed{\wp_1}$ $T_1 = \{.......,\boxed{\mathrm{P}(S_1)},\mathrm{P}(S_0),\boxed{\mathrm{V}(S_1)},..,\mathrm{P}(S_1),...,\mathrm{V}(S_1),\mathrm{V}(S_0)...........\}$
$\boxed{\wp_2}$ $T_2 = \{.....,\mathrm{P}(S_1),...................,\mathrm{V}(S_1),..........................................\}$
$(P_0 > P_1 > P_2)$

1. $T_0$, $T_1$ and $T_2$ are initiated.
2. $T_2$ locks $S_1$. $T_1$ is blocked on $S_1$.
3. $T_0$ can lock $S_0$.

The situation of semaphore pair plugging-in is shown in Figure 3.5 on this page.

We call this plugging-in of a semaphore pair the *upward plug-in* method, and the ordinary plugging-in the *downward plug-in* method. The system behaviour in the example above shows that the *upward plug-in* method seems to fulfil the same function as the dependency protocol for multiprocessors. But there is a limitation to the application of the *upward plug-in* method for simulating the dependency protocol, as shown in the following:

•—*The Limitation of Upward Plug-in*—
$\boxed{\wp_0}$ $T_0$ = {................,P($S_0$),..............,V($S_0$),................................................}
$\boxed{\wp_1}$ $T_1$ = {.......,$\boxed{\text{P}(S_1)}$,P($S_0$),$\boxed{\text{V}(S_1)}$,...,P($S_1$),.DO-WORK1..,V($S_1$),V($S_0$)...}
$\boxed{\wp_2}$ $T_2$ = {............,P($S_0$),...DO-WORK2....,V($S_0$),..........................................}
$\boxed{\wp_3}$ $T_3$ = {.....,P($S_1$),...DO-WORK3..........,V($S_1$),...........................................}
$\boxed{\wp_4}$ $T_4$ = {.........,P($S_0$),...DO-WORK4......,V($S_0$),...........................................}
$(P_0 > P_1 > P_2 > P_3 > P_4)$

1. $T_0$, $T_1$, $T_2$, $T_3$, $T_4$ are initiated.
2. $T_3$ locks $S_1$.
3. $T_1$ is blocked on $S_1$.
4. $T_4$ locks $S_0$. $T_2$ is blocked. $T_0$ is blocked.

The purpose of $T_3$ plugging $S_1$ into $T_1$ is for the highest task $T_0$ to avoid blocking. But instead of $T_0$, the lowest task $T_4$, which is lower than $T_1$, can lock. If $T_2$ were to lock, rather than $T_4$, it would seem reasonable because $T_2$ is higher than the locking task. If $T_2$ uses a semaphore nest, the blocking time of the highest task $T_0$ becomes longer. But under the application of the *plug-in* method, the possibility that instead of the higher task the middle task can lock is acceptable. So far we have assumed the ideal situation where preemption does not apply. When in more realistic situations preemption does apply, so the effectiveness of the *upward plug-in* method will be greater because higher tasks can preempt lower tasks.

Despite this, the effectiveness of the *upward plug-in* method may still be limited. Here we consider a conservative approach to the application of the *upward plug-in* method in this ideal situation. We propose the concept of the *floor value* of a semaphore to limit the *upward plug-in* method for simulating the dependency protocol for multiprocessors.

Each semaphore has a *floor value* denoted by $P_F(S)$ which is the lowest priority of the tasks which may use the semaphore $S$. It is denoted by $P_F(S)$ similarly to the ceiling value $P_C(S)$. The priorities of all the tasks which use $S$ range from $P_F(S)$ to $P_C(S)$, inclusive. One possible approach would be that the *upward plug-in* method is applicable only when the priority of the task which is locking the parent semaphore is lower than $P_F(S)$. This limitation is specific to an ideal environment of multiprocessor systems. In the uniprocessor case, the dependency protocol has no such limitation. Another approach to this problem would be to set up no limitation for the application of the *upward plug-in* method, because the rule that the priority of the task which is locking the parent semaphore is lower than $P_F(S)$ is conservative.

| *Informal Description for Upward Plug-in Method (Conservative)* |
| --- |
| A task with priority $P_i$ is holding $S_1$ , in circumstances where $S_0$ is on the dependency list of $S_1$ at a priority greater than $P_i$, for example, $P_j$ which is not the highest priority, and $S_0$ is the outermost semaphore of the nest of the dependency.<br>Instead of going "P($S_0$)... P($S_1$)",<br>$T_j$: "$\boxed{\mathrm{P}(S_1)}$P($S_0$)$\boxed{\mathrm{V}(S_1)}$... P($S_1$)"<br>will be adopted only when the priority of $P_i$ is lower than $P_F(S_0)$. |

The essential solution will be proposed in Chapter 6 through defining the dependency protocol for multiprocessors, because by the definition of the dependency protocol a task can lock the dependency semaphore $S_i$ only if it is higher than $max\{\mathsf{D}[S_p](S_i)\}$.

## 3.9 Integration of Downward and Upward Plug-in

The *downward plug-in* method works in the *inward* case, and the *upward plug-in* method works in the *forward* case. The integration of both the *downward plug-in* and *upward plug-in* method therefore covers both the *forward* case and the *inward* case; we can therefore proceed to investigate the integration of both methods. A major source of semaphore blocking comes from semaphore chains. The principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"* realises the essential work of the *plug-in* method; that is it cuts semaphore chains using their dependency feature. Cutting the chain can reduce the blocking duration of working units. The integration of *downward plug-in* and *upward plug-in* cuts the semaphore chain in both *inward* and *forward* cases.

In order to integrate the *downward* and the *upward plug-in* methods, the example in the previous section is rewritten as follows:

•—*Downward and Upward Plug-in*—

$\boxed{\wp_0}$ $T_0 = \{$..........,P($S_0$),..............,V($S_0$),.....................................$\}$

$\boxed{\wp_1}$ $T_1 = \{$.......,$\boxed{\mathrm{P}(S_1)}$,P($S_0$),$\boxed{\mathrm{V}(S_1)}$,..,P($S_1$),...,V($S_1$),V($S_0$).....$\}$

$\boxed{\wp_2}$ $T_2 = \{$.....,$\boxed{\mathrm{P}(S_0)}$,P($S_1$),$\boxed{\mathrm{V}(S_0)}$,..................,V($S_1$),..............$\}$

$(P_0 > P_1 > P_2)$

1. $T_0$, $T_1$, $T_2$ are initiated.
2. In the *forward* case, $T_2$ locks $S_1$ and $T_1$ is blocked by plugged-in $S_1$. In the *inward* case, $T_1$ locks $S_0$ and $T_2$ is blocked by plugged-in $S_0$.

Let us consider the more typical case of semaphore chains which was described in Section 3.1.1, and investigate the system behaviour with regard to the cutting of the semaphore chain when *downward* and *upward plug-in* are applied.

—*Intertask Semaphore Chain*—

$\boxed{\wp_0}$ $T_0 = \{$...........,P($S_0$),.......,P($S_1$),......................,V($S_1$),V($S_0$),..........$\}$

$\boxed{\wp_1}$ $T_1 = \{$...........,P($S_1$),.......,P($S_2$),......................,V($S_2$),V($S_1$),..........$\}$

$\boxed{\wp_2}$ $T_2$ = {..........,P($S_2$),.......,P($S_3$),.....................,V($S_3$),V($S_2$),..........}
($P_0 > P_1 > P_2$)

The semaphore $S_1$ of $T_1$ is the encounter point of the *downward plug-in* from $T_2$ and the *upward plug-in* from $T_0$. We can ignore the *upward plug-in* of the highest task $T_0$, which is useless according to the definition of the *upward plug-in*. The following rule is applied to the semaphore which is the encounter point for cutting the semaphore chain.

| *The Rule for the order of plug-in* |
|---|
| 1. Apply the *downward plug-in* method.<br>The lower priority tasks are blocked in the *inward* case.<br>2. Apply the *upward plug-in* method.<br>The middle priority tasks are blocked in the *forward* case.<br>The semaphore is free for a higher task. |

The plugging-in result is shown as follows:

—*Downward and Upward Plug-in for Intertask Semaphore Chain*—
$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),.......,P($S_1$),......................,V($S_1$),V($S_0$),.............................}
$\boxed{\wp_1}$ $T_1$ = {.......,P($S_2$),P($S_0$),P($S_1$),V($S_0$),V($S_2$),.......,P($S_2$),........,V($S_2$),V($S_1$),..}
$\boxed{\wp_2}$ $T_2$ = {.......,P($S_1$)P($S_2$),V($S_1$).......,P($S_3$),.............,V($S_3$),V($S_2$),...................}
($P_0 > P_1 > P_2$)

We have not yet described in detail how the *plug-in* method influences semaphore chains. We now describe in some detail both the *downward plug-in* and the *upward plug-in*. We can simplify the problem to reduce the number of event combinations.

1. $(t_1^0, P_0, S_0) \lhd (t_1^1, P_1, S_1) \lhd (t_2^0, P_0, S_1)$

   (a) $(t_1^2, P_2, S_2) \lhd (t_1^1, P_1, S_1)$
   $T_1$ is blocked on $S_2$. $T_2$ goes through P($S_1$)P($S_2$),V($S_1$) and locks $S_2$. If $T_1$ releases $S_2$, $T_1$ is blocked on $S_0$. The blocking duration of $T_0$ is zero working units.

   (b) $(t_1^1, P_1, S_1) \lhd (t_1^2, P_2, S_2) \lhd (t_2^0, P_0, S_1)$
   $T_1$ is blocked on $S_0$ holding $S_2$. $T_2$ is blocked on $S_2$ holding $S_1$. This blocks $T_0$ on $S_1$. The system is deadlocked.

   (c) $(t_2^0, P_0, S_1) \lhd (t_1^2, P_2, S_2)$
   $T_1$ is blocked on $T_0$ holding $S_2$, $T_2$ is blocked on $S_1$ and then on $S_2$ which will block $T_1$. The blocking duration of $T_0$ is zero working units.

2. $(t_2^0, P_0, S_1) \lhd (t_1^1, P_1, S_1)$

(a) $(t_1^2, P_2, S_2) \lhd (t_2^0, P_0, S_1)$
$T_2$ only goes through the plugged-in semaphore pair and locks $S_2$, which does not influence $T_0$. $T_1$ is blocked on $S_1$ or $S_0$. The blocking duration of $T_0$ is zero working units.

(b) $(t_2^0, P_0, S_1) \lhd (t_1^2, P_2, S_2)$
$T_2$ is blocked by the plugged-in semaphore $S_1$. This looks like unnecessary blocking. The blocking duration of $T_0$ is zero working units.

3. $(t_1^1, P_1, S_1) \lhd (t_1^0, P_0, S_0)$

(a) $(t_1^2, P_2, S_2) \lhd (t_1^1, P_1, S_1)$
$T_1$ locks $S_2$. The *upward plug-in* semaphore $S_2$ blocks $T_1$ which prevents $T_1$ from locking $S_1$ which will block $T_0$. The blocking duration of $T_0$ is zero working units.

(b) $(t_1^1, P_1, S_1) \lhd (t_1^2, P_2, S_2)$
$T_1$ locks $S_1$. $T_2$ will be blocked by plugged-in semaphore $S_1$. The blocking duration of $T_0$ is one working unit.

Using both the *downward plug-in* and the *upward plug-in* method, in most cases the blocking duration of $T_0$ is generally zero working units, and at most one working unit, but a deadlock occurs in case (1-b), and also the middle task $T_1$ has on some occasions to wait for two working units. Also there is some unnecessary blocking to the lower task in the *backward* case.

Using the *downward plug-in* for the *inward* case, and the *upward plug-in* for the *forward* case, the blocking duration of the highest task $T_0$ is at most one working unit. It is generally successful in cutting the blocking chain except for one occasion of deadlock.

However this deadlock shows the drawback of the plug-in method because the original example will not deadlock and the example with plug-in introduces deadlock. In addition there are many timings that might cause deadlock between semaphores. However an improved solution can be found by introducing *colour semaphores*, see Section 4.3.4 in Chapter 4.

## 3.10 Plug-in for Semaphore Nests

Each task that is claiming a member of a semaphore nest at lower priority could block higher tasks. In the *pure plug-in* method, we considered the outermost semaphores of semaphore nests. We now look at all possible members of a semaphore nest which may block the higher tasks.

Let us consider the example below:

—*A variation of Example5 in section 3.5*—
$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),........,P($S_1$),...,V($S_1$),..,V($S_0$),.............................}
$\boxed{\wp_1}$ $T_1$ = {.......,P($S_2$),.....,P($S_1$),..DO-WORK1..,V($S_1$),V($S_1$),...............}

$\boxed{\wp 2}$ $T_2$ = {..............,P($S_1$),..DO-WORK2...,V($S_1$),...............................}
($P_0 > P_1 > P_2$)

The dependency list is:
$\mathsf{D}S_1 = [(S_0, P_0), (S_2, P_1)]$
The merged list and the system which the *plug-in* method is applied to are shown in Table 3.6 and the diagram in Section 3.5. According to the *pure plug-in* method, the system behaves as follows:

1. $T_0$, $T_1$, $T_2$ are initiated.
2. $T_0$ locks $S_0$.
3. $T_1$ locks $S_2$.
4. $T_2$ is blocked on $S_0$.
5. $T_1$ locks $S_1$.
6. $T_0$ is blocked on $S_1$. $T_0$ is blocked for DO-WORK1.

The highest priority task $T_0$ is blocked by the lower task $T_1$ because $S_1$, which is not the outermost of the semaphore nest, is excluded from the target of plugging-in by the *pure plug-in* method. We would like to apply the *plug-in* method to the inner member of semaphore $S_1$ of the semaphore nest. There are two approaches:
(1) We would like to relax the limited condition that the *point semaphore* of the *pure plug-in* method in Section 3.3 should be the outermost semaphore of the semaphore nest.
(2) We would like to extend the definition of the *point semaphore* to all member of a semaphore nest.
The advantage of the *pure plug-in* method is that it is simple. However the above example shows a disadvantage of the *pure plug-in* method. So let us try an alternative version of the *plug in* rule.

| *Informal Description for Modified Plug-in Method* |
|---|
| A task with priority $P_i$ which has already been holding $S_{i0}, \ldots, S_{in-1}$ is about to claim $S_{in}$ of a semaphore nest, in circumstances where $S_{jk}$ is on the dependency list of $S_{in}$ at a priority $P_j$ greater than $P_i$.<br>Instead of going "P($S_{i0}$)...P($S_{in}$) DO-WORK V($S_{in}$)...V($S_{i0}$)",<br>$\boxed{\mathrm{P}(S_{jk})}$ P($S_{i0}$)...P($S_{in}$) $\boxed{\mathrm{V}(S_{jk})}$ DO-WORK V($S_{in}$)...V($S_{i0}$)<br>will be adopted. |

The merged list $LL_{ink}$ is defined in just the same manner as in the *pure plug-in* method. Applying the above rule to the semaphore nest of $T_1$ and the *pure plug-in* method to $S_0$ of $T_2$ in the example, the result is as follows:

•*Modified Plug-in for Example 5 in section 3.5*
$\boxed{\wp 0}$ $T_0$ = {.......,P($S_0$),..,P($S_1$),.DO-WORK0..,V($S_1$),..,V($S_0$),...........................}
$\boxed{\wp 1}$ $T_1$ = {.......,$\boxed{\mathrm{P}(S_0)}$ P($S_2$),.....,P($S_1$),$\boxed{\mathrm{V}(S_0)}$...,V($S_1$),V($S_1$),.........................}
$\boxed{\wp 2}$ $T_2$ = {..............,P($S_2$),P($S_0$),P($S_1$),V($S_0$),V($S_2$),..DO-WORK2...,V($S_1$),....}
($P_0 > P_1 > P_2$)

1. $T_0$, $T_1$, $T_2$ are initiated.
2. $T_0$ locks $S_0$.
3. $T_1$ is blocked on $S_0$.
4. $T_2$ is blocked on $S_0$.
5. $T_0$ locks $S_1$.

$T_0$ is not blocked on $S_1$. But the middle task $T_1$ is blocked for the duration of $T_0$'s working unit DO-WORK0, then it may be blocked by the lower task $T_2$. If $T_1$ locks $S_0$, the highest task $T_0$ is blocked for the duration of the P primitives sequence of the semaphore nest. However this is not so different from the *forward* case in the *pure plug-in* method.

The biggest problem is shown in the following example:

—*Example*—
$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),.,P($S_1$),..,P($S_2$),..,V($S_2$),V($S_1$),V($S_0$),...............}
$\boxed{\wp_1}$ $T_1$ = {......,P($S_1$),...,P($S_2$),..DO-WORK1..,V($S_2$)V($S_1$),................}
$(P_0 > P_1)$

The dependency lists of the example are set out below;
$\mathsf{D}S_1 = [(S_0, P_0)]$, $\mathsf{D}S_2 = [(S_0, P_0), (S_1, P_0)]$

If $S_1$ is plugged-in to the semaphore nest of $T_1$, then the nest would include two P primitives of $S_1$, which itself causes *deadlock*. We can avoid the problem by insisting that the semaphores which the lower task has been holding should be excluded from plugging-in semaphores. We must therefore define a new version of merged list $LLLM_{ik}$, which does not include the semaphores which the lower task has already been holding.

We have defined the *point semaphore* whose P primitive is a target of plugging in. The problem is which semaphore, $S_1$ or $S_2$, should be a *point semaphore* or should both? We can consider two cases.

1. The innermost member of a semaphore nest of the lower task should be a *point semaphore*. This approach exploits plugging only once for the overall P primitive sequence of the semaphore nest. Compared with the *pure plug-in* method where the outermost semaphore is a point semaphore, the advantages and disadvantages seem comparable.

2. Each member of semaphore nest which has a dependency list of the higher task should be a *point semaphore*. This approach exploits plugging for all semaphore members of the semaphore nest. This is a complete procedure that ensures *"a lower task is forced to avoid locking a semaphore which would block higher tasks"*.

The first modified version is that the overall P primitive sequence of a semaphore nest is the target of plugging in. The modified version of the *plug-in* method in the first case would be defined as follows.

| *The First Modified Version of the Plug-in Method* |
|---|
| Let us consider the semaphore list $S_{i_0} \prec \ldots \prec S_{i_m} \prec \ldots \prec S_{i_n}$.<br>We work out a list $L_{ij}$, in the acquisition order, elements of which are the semaphores<br>that are dependent on $S_i$ at $P_j$, that is those which are acquired after $S_i$ locking<br>by some task running at priority $P_j$. When a task attempts to lock $S_{i_n}$ at $P_k$,<br>which has already been holding $S_{i_0}$,$S_{i_1}$, $\ldots$, $S_{i_{n-1}}$, and $S_{i_n}$<br>is the innermost semaphore of the nest, then delete all of them from $L_{ij}$ in such a way that;<br>$LM_{i_nj} = L_{i_nj} \setminus \{S_{i_0}, S_{i_1}, \ldots, S_{i_{n-1}}\}$<br>the system merges all the $LM_{i_nj}$for which $k > j$.<br>This represents all the semaphores that depend on $S_{i_n}$ at priorities greater than $P_k$.<br>We call the merged list $LLM_{i_nk}$. Plug in the semaphore pairing as follows.<br>$T_k$: $\boxed{\mathrm{P}(LLM_{i_nk})}$ P($S_{i0}$)… P($S_{i_n}$) $\boxed{\mathrm{V}(LLM_{i_nk})}$ DO-WORK V($S_{i_n}$)… V($S_{i_0}$) |

Applying this first modified *plug-in* method to the example:

—*Example*—

$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),.P($S_1$),..,P($S_2$),..,V($S_2$),V($S_1$),V($S_0$),.............................}

$\boxed{\wp_1}$ $T_1$ = {......,$\boxed{\mathrm{P}(S_0)}$,P($S_1$),..,P($S_2$),$\boxed{\mathrm{V}(S_0)}$,..DO-WORK1..,V($S_2$)V($S_1$),......}

$(P_0 > P_1)$

As shown in this example, all the semaphores of the nest of $T_1$ are wrapped with $S_0 = LLM_{1_1}$.

The second version of modified *plug-in* method is that, instead of the overall semaphore nest, all semaphores of a semaphore nest are targets for plugging in. The second version is defined as follows.

| *The Second Modified Version of the Plug-in Method* |
|---|
| Let us consider the semaphore list $S_{i_0} \prec \ldots \prec S_{i_m} \prec \ldots \prec S_{i_n}$.<br>We work out a list $L_{ij}$, in the acquisition order, elements of which are the semaphores<br>that are dependent on $S_i$ at $P_j$, that is those which are acquired after $S_i$ locking<br>by some task running at priority $P_j$. When a task attempts to lock $S_{i_m}$ at $P_k$,<br>which has already been holding $S_{i_0}$,$S_{i_1}$, $\ldots$, $S_{m-1}$, then delete all of them from $L_{i_mj}$<br>so that: $LM_{i_mj} = L_{i_mj} \setminus \{S_{i_0}, S_{i_1}, \ldots, S_{i_{n-1}}\}$<br>the system merges all the $LM_{i_mj}$ for which $k > j$.<br>This represents all the semaphores that depend on $S_{i_m}$ at priorities greater than $P_k$.<br>We call the merged list $LLM_{i_nk}$.<br>Plug in for all $S_{i_m}$ which has dependency list,the semaphore pairing as follows.<br>$T_k$: $\boxed{\mathrm{P}(LLM_{i_mk})}$ P($S_{i_m}$) $\boxed{\mathrm{V}(LLM_{i_mk})}$ DO-WORK V($S_{i_n}$)… V($S_{i_0}$) |

Applying this second modified *plug-in* method to the example:

—*Example*—

$\boxed{\wp_0}$ $T_0$ = {..,P($S_0$),.P($S_1$),..,P($S_2$),..,V($S_2$),V($S_1$),V($S_0$),...............................................}

$\boxed{\wp_1}$ $T_1$ = {..,$\boxed{\text{P}(S_0)}$P($S_1$),$\boxed{\text{V}(S_0)}$,..,$\boxed{\text{P}(S_0)}$P($S_2$),$\boxed{\text{V}(S_0)}$,..DO-WORK1..,V($S_2$)V($S_1$),..}

($P_0 > P_1$)

Each semaphore of the nest of $T_1$ is plugged in by the modified rule. $T_1$ will stop before $S_1$ and $S_2$ without locking them.

In the discussion in this chapter, we have introduced three real-time synchronisation rules which the *plug-in* method has been applied to:

1. The *pure plug-in* method. The *point semaphores* for plugging are the outermost semaphores of semaphore nests. This method misses the chance of plugging-in if a member of a semaphore nest of a lower task could be the parent semaphore of a semaphore nest of a higher task.

2. The *first modified plug-in* method. The *point semaphores* for plugging are the innermost semaphores of semaphore nests. This method can effectively cover the gaps of the *pure plug-in* method.

3. The *second modified plug-in* method. The *point semaphores* for plugging are all members of semaphore nests. This method completely ensures that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"*.

## 3.11 Summary

In the *plug-in* method, the semaphore V primitive automatically assigns a processor to a higher priority task. This is not the ordinary definition of the semaphore V primitive.

The *plug-in* method is an intuitive and simple idea, but we believe that it is a novel idea which has not been used in the study of real-time synchronisation for multiprocessors. We have shown examples of how the *plug-in* method can be applied. We have pointed out that the *plug-in* method has several drawbacks. However, the *plug-in* method is valuable for investigating the nature of priority for multiprocessor systems, which is extremely complex. We believe that the *plug-in* method may provide the first step in investigating this area. It is then a tool for getting an idea of more realistic situations. However it is not proposed for practical use because it introduces side effects , and is subject to deadlock.

The *plug-in* method itself is only a technique, not a fixed synchronisation protocol. By using a variety of types of *plug-in* method, we could produce suitable synchronisation protocols to fit with each real-time system. It would work effectively in a complex environment as well as the three nested cases. The next step is to implement the plug-in method efficiently.

We return to the starting points of the semaphore blocking problem here. The sources of semaphore blocking are: (1) intra-task semaphore nests, (2) intertask semaphore chains.

The essential work of the *plug-in* method is that it cuts semaphore chains using the dependency feature of semaphore nests and semaphore chains. Cutting the chain can reduce the blocking duration of working units.

In an environment where priority inheritance does not work, the *downward plug-in* and the *upward plug-in* methods provide new mechanisms to reduce the blocking time of higher tasks blocked by a locking lower task. These mechanisms realise the principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"*.

# Chapter 4

# Colour Semaphores for Multiprocessors

Semaphore nests are primary causes of semaphore blocking. We continue to discuss the problem in this chapter. We consider two cases of the distance of the program code between the members of a semaphore nest: short and long. For example, the first semaphore of a nest is issued in the user mode and the second semaphore of the nest is issued in the kernel mode, where between them there may be a trap. This means that the distance between each semaphore is long. If the first and the second semaphores are both issued in the kernel mode, the distance is likely to be short. Therefore the distance is not expressed by physical code length.

If the code distance between each semaphore in a nest owned by a higher task is long and time consuming; the *semaphore plug-in* method works well. In the case that the distance between semaphores is short, multiple semaphores can be linked together to make a semaphore list, as shown in Figure 4.1 on this page.

In case $\boxed{\mathbf{1}}$ the *plug-in* method discussed in the previous chapter works well. However, in

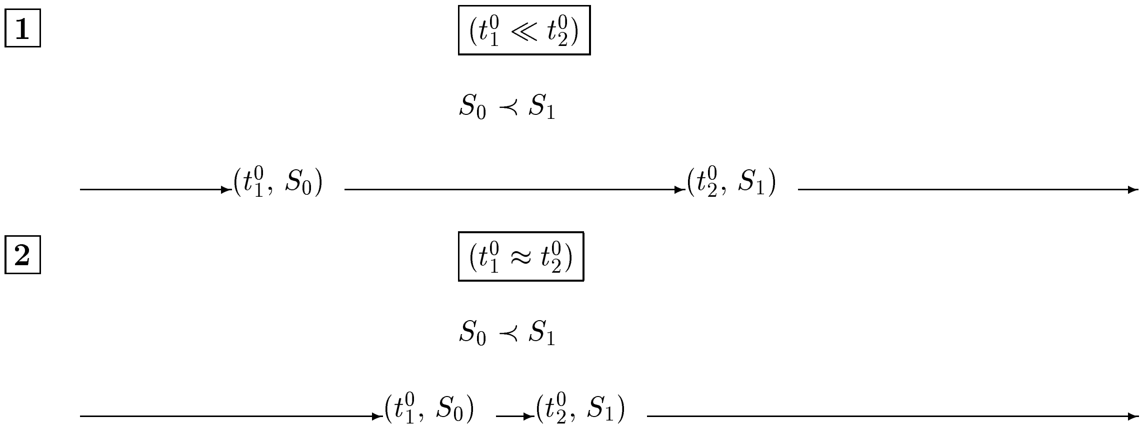


Figure 4.1: Code Distance

case $\boxed{\mathbf{2}}$ in Figure 4.1 on the preceding page, the *plug-in* method does not work effectively; instead, the sequence of semaphores can be called simultaneously, therefore new techniques will be considered.

In this chapter, we propose classifying semaphore queues as either a priority (descendent) ordered queue or a time (antecedent) ordered queue. We investigate the functions of a priority descendent ordered queue for semaphores and reach the conclusion that it is an essential mechanism for real-time synchronisation.

In this chapter, we propose a new semaphore primitive called the *colour semaphore*, which issues simultaneous multiple semaphore operations with two types of priority based queues for semaphores.

First, let us return to the ordinary semaphore algorithm in which the P primitive puts tasks in the semaphore queue in first-in order and the V primitive wakes up the first-in task in the semaphore queue rather than the highest priority task. We call this ordinary semaphore queue mechanism a time antecedent semaphore queue mechanism. This maintains the fairness among tasks for Time Sharing Systems. By means of colour semaphores, we propose priority descendent ordered queue mechanisms for semaphores. It will be shown that the priority descendent ordered semaphore queue mechanism produces essential real-time properties for real-time systems. We will then discuss the use of the *colour semaphore* combined with the *plug-in* method, which appears to solve several of the problems of the *plug-in* method.

It is the original and ordinal definition that a semaphore is an integer variable [**Dijkstra68**]. From this chapter, we will describe several real-time synchronisation algorithms in the manner of *Literate Programming* [**Knuth92**]. For this purpose, we will extend semaphores as a data structure. Basic form is described as follows:

**type** $semaphore =$
**record**
  $semaphore\_count : Integer;$
  $semaphore\_queue : Queue;$
**end** $semaphore;$

The original semaphore is expressed as $semaphore\_count : Integer;$. As a matter of convenience and convention, for a semaphore $S$,
$S \leftarrow S \pm 1$ means $S.semaphore\_count \leftarrow S.semaphore\_count \pm 1$
This allows us to describe algorithms in a compact notation.

## 4.1 The Simultaneous Priority-ordered Semaphores

We begin with a discussion of simple example. What we want to do is to make sure that if a high priority task is triggered, it should not be delayed for more than one *working unit* blocking time caused by lower priority tasks. We therefore need to find an approach to avoid the phenomenon described as below:

—*Example for Sequential Semaphore Operation*—

$\boxed{\wp_0}$ $T_0$ = {.............,P($S_0$),P($S_1$),.....,V($S_1$),V($S_0$),...........}

$\boxed{\wp_1}$ $T_1$ = {....,P($S_1$),...DO-WORK1......,V($S_1$),..................}

$\boxed{\wp_2}$ $T_1a$ = {......,P($S_1$),..DO-WORK2........,V($S_1$),............}

$\boxed{\wp_3}$ $T_2$ = {........,P($S_0$),.DO-WORK3..,V($S_0$),....................}

$(P_0 > P_1 = P_1a > P_2)$

1. $T_1$ is initiated and locks $S_1$.
2. $T_1a$ is initiated, claims $S_1$ and is blocked.
3. $T_2$ is initiated and locks $S_0$.
4. $T_0$ is initiated and claims $S_0$, then is blocked.
5. $T_2$ executes DO-WORK3 and releases $S_0$. $T_0$ locks $S_0$ and claims $S_1$, then is blocked.
6. $T_1$ executes DO-WORK1, releases $S_1$, then $T_a$ locks $S_1$ and executes DO-WORK2.
7. $T_1a$ releases $S_1$, then $T_0$ locks $S_1$.

It should be noted that the V primitive operates the semaphore queue in time order as ordinary semaphore V primitives. In the above example, the highest priority task $T_0$ waits for three working units. According to the queue operation exploited by the V primitive, it is possible that $T_0$ can lock $S_1$ first, prior to $T_1a$. In such a case $T_0$ waits for two working units. However in the worst case, as in the above example, $T_0$ could wait for three working units.

Our goal in this chapter is to develop a new semaphore algorithm which deals with priority order beneficially within semaphores themselves without the influence of the CPU scheduler.

### 4.1.1 SP and SV Primitives

It is clear that in the example above, blocking for two working units is caused by $T_0$ claiming the $S_0$ and $S_1$ semaphores separately. One way of avoiding this would be that instead of claiming each semaphore in $T_0$, semaphores $S_0$ and $S_1$ are claimed simultaneously as a sequence list.

This kind of semaphore operation is called the *simultaneous semaphore list operation*. First we discuss a previous related work, and investigate specific problems for the simultaneous semaphore list operation. This type of semaphore was proposed in [**Maekawa87**] who used the notation **SP** and **SV** as abbreviations of *simultaneous* P and V primitives. Maekawa's definitions of **SP** and **SV** are described in Figure 4.2 on the next page.

The task that has requested the **SV** primitive continues or is placed in the ready queue and control is passed to the CPU scheduler. **SP** and **SV** primitives were invented to increase the power of process coordination to get a solution more easily, otherwise, it would be very difficult to construct a deadlock-free solution for complex problems such as the dining philosophers problem. Using **SP** and **SV** primitives, the solution of the dining philosophers problem is simple and direct, and is shown in Figure 4.3 on page 61, where forks and knives are changed to chopsticks for simplicity of description.

---

**SP**$(S_0; S_1; ....; S_{n-1}\ ) \equiv$
**begin**
$semaphore\_list \leftarrow (S_0, S_1, ....., S_{n-1});$
**if** $S_0 \geq 1 \wedge .... \wedge S_{n-1} \geq 1$
**then**
    **for** $i \leftarrow 0$ **to** n -1 **do**
        $S_i \leftarrow S_i - 1;$
    **endfor**
**else**
    $first\_semaphore \leftarrow lookfor(S_i < 1; S_i \in semaphore\_list);$
    $enqueue(first\_semaphore.semaphore\_queue, current\_task);$
    $program\_counter \leftarrow the\_begining\_of(\mathbf{SP});$
    {all conditions are re-examined when the process is reactivated}
**endif**
**end**

**SV**$(S_0; S_1; .....; S_{n-1}) \equiv$
**begin**
**for** $i \leftarrow 0$ **to** n -1 **do**
    $S_i \leftarrow S_i + 1;$
    **while** $(t \leftarrow dequeue(S_i.semaphore\_queue)) \neq \{\,\}$ **do**
        $enqueue(ready\_queue, t);$
    **endwhile**
    {remove all the processes waiting associated with $S_i$ into the ready queue }
**endfor**
$continue(current\_task) \vee enqueue(ready\_queue, current\_queue); cpu\_scheduler();$
{The task that has requested **SV** operation continues
or is placed in the ready queue and the control is passed to the CPU scheduler.}
**end**

---

Figure 4.2: **SP** and **SV** algorithm

**Shared Variables**
**var** chopstick: **array**[0..4] **of semaphore** := (1, 1, 1, 1, 1)

<**Task i**>
**loop**
Think;
**SP**(chopstick[(i + 1 ) **mod** 5], chopstick[i]);
Eat;
**SV**(chopstick[(i + 1 ) **mod** 5], chopstick[i]);
**endloop**

Figure 4.3: The Solution of the Dining Philosophers Problem with **SP** and **SV**

The solution of this simple simultaneous semaphore operation cannot prevent a philosopher from being permanently locked out by two conspiring neighbour tasks. This is known as *starvation*, or *indefinite delay*, which will be discussed later.

These **SP** and **SV** primitives only consider simultaneous semaphore operations, thus do not include any consideration of real-time scheduling from the aspect of priority. They can still not resolve the blocking time problem, as shown below:

—*Example for Busy Waiting*—
$\boxed{\wp_0}$ $T_0 = \{$..............,**SP**$(S_0; S_1)$,.....,**SV**$(S_0; S_1)$,.............$\}$
$\boxed{\wp_1}$ $T_1 = \{$....,P$(S_1)$,..........,V$(S_1)$,..................................$\}$
$\boxed{\wp_2}$ $T_1a = \{$.......,P$(S_1)$,.........,V$(S_1)$,...............................$\}$
$\boxed{\wp_3}$ $T_2 = \{$..........,P$(S_0)$,...,V$(S_0)$,...................................$\}$
$(P_0 > P_1 = P_1a > P_2)$

1. $T_1$ is initiated and locks $S_1$.
2. $T_1a$ is initiated, claims $S_1$ and is blocked.
3. $T_2$ is initiated and locks $S_0$.
4. $T_0$ is initiated, claims $S_0$ and $S_1$ simultaneously, then is blocked and enqueued on the semaphore queue of $S_0$.
5. $T_2$ releases $S_0$, $T_0$ reactivates **SP** primitive, is still blocked on $S_1$ and enqueued on the semaphore queue of $S_1$.
6. $T_1$ releases $S_1$. $T_0$ or $T_1a$ are woken up. By the ordinary time order based semaphore queue operation, the V primitive wakes up $T_1a$, which will locks $S_1$, then $T_0$ should wait on $S_1$ again. By the priority order based semaphore queue operation which will be discussed later, the V primitive would wakes up $T_0$, which re-executes **SP** and locks $S_0$ and $S_1$.

The blocking time of $T_0$ is three or two working units, depending on the semaphore V primitive, which should be separated from the implementation of **SP** and **SV** primitives. The other problem of **SP** is that the primitive is essentially *busy waiting*. Moreover, **SP** and **SV** primitives suffer from *starvation*, as illustrated below:

——*Example of Starvation by SP and SV*——
$\boxed{\wp_0}$ $T_0$ = {..........,**SP**($S_0$; $S_1$),.....,**SV**($S_0$; $S_1$),............}
$\boxed{\wp_1}$ $T_1$ = {.......,P($S_1$),.....,V($S_1$),................................}
$\boxed{\wp_2}$ $T_2$ = {......,P($S_1$),....,V($S_1$),.................................}
$\boxed{\wp_3}$ $T_3$ = {.....,P($S_0$),....,V($S_0$),..................................}
$\boxed{\wp_4}$ $T_4$ = {....,P($S_0$),....,V($S_0$),...................................}
$(P_0 > P_1 > P_2 > P_3 > P_4)$

1. $T_4$ locks $S_0$. $T_3$ waits for $S_0$. $T_2$ locks $S_1$. $T_1$ waits for $S_1$.
2. $T_0$ waits for $S_0$.
3. $T_4$ releases $S_0$. $T_0$ and $T_3$ are activated, $T_0$ waits for $S_1$ and $T_3$ locks $S_0$.
4. $T_2$ releases $S_1$. $T_0$ and $T_1$ are activated, $T_0$ waits for $S_0$. $T_1$ locks $S_1$.

The circulation can occur so that $T_0$ is in the *starvation* state, and **SP** and **SV** are not concerned with priority order.

As described in previous examples, these simple simultaneous semaphore primitives, **SP** and **SV**, have the following drawbacks:

1. Primitives themselves do not include a scheduling algorithm that considers priority.
2. They work as busy waiting.
3. They cause *starvation*.

### 4.1.2 SL-P and SL-V Primitives

Considering the above phenomenon, both ordinary semaphores and **SP** and **SV** primitives, the V primitive should provide a scheduling algorithm related to priority.

The original definition of P and V primitives did not mention about any queueing mechanism [**Dijkstra68**]. However it is usual that the semaphore P primitive puts blocked tasks into its semaphore queue in time order and the V primitive wakes up the first-in task. The reason why the ordinary semaphores exploit the time ordered FIFO semaphore queue policy is that they seek to maintain the principle of *fairness* which is essential for Time Sharing Systems.

Here, on the contrary, we consider the properties of real-time systems and should solve the time constraint problem rather than realising fairness. Therefore we introduce the priority ordered queue for semaphores. Tasks waiting on a semaphore queue are ordered, with the highest priority first, so that the task on the queue which has the highest priority is scheduled first. This means that the synchronisation primitives themselves include a scheduling with regard to priority.

We can summarise two types of semaphore queue operation by the V primitives as follows:

1. Time order based on *fairness*. The semaphore queue is arranged in time antecedent order.

2. Priority order based on *urgency*. The semaphore queue is arranged in priority descendent order.

Priority order manipulation is not great, and therefore, for real-time synchronisation, priority order based semaphore queue is the better policy.

**SP** operates on a single semaphore queue only, as shown in the algorithm in Figure 4.2 on page 60. Compared with this, the implementation of new operations should be realised so that they operate on multiple semaphore queues simultaneously.

We therefore proposed a new semaphore primitive which has the following features:

1. simultaneous operations on multiple semaphore queues.

2. priority descendent ordered queues.

The data structure of the semaphore and operations of semaphore queues are as follows:

**type** $sl\_semaphore =$
**record**
  $semaphore\_count : Integer;$
  $semaphore\_queue : Priority\_Descendent\_Order\_Queue;$
**end** $sl\_semaphore;$
$enqueue\_priority\_descendent\_order(semaphore.semaphore\_queue, task);$
$dequeue(semaphore.semaphore\_queue, priority);$

In multiprocessors, the concept of priority mainly applies to semaphore queues, so the structure of a semaphore queue needs to be fully investigated.

We call new semaphore primitives the *simultaneous list semaphore*, denoted **SL-V** and **SL-P**. **SL-P** scans the list of claimed semaphores given by the parameter. The task can lock the semaphore list if the semaphore counts of all members of the semaphore nest are positive. If not, the task is blocked such that each claimed semaphore state is in the ready state at the semaphore in the list whose semaphore count is positive, and the wait state at the semaphore in the list whose semaphore count is not positive. **SL-V** removes the top task of each semaphore in the list given by the parameter that is the highest task in the semaphore queue and sets the claimed semaphores' states of the task the ready state. Tasks whose claimed semaphores' states become all ready are invoked to run in the **SL-V**. In the algorithms, $claimed\_semaphores(S_i)\hat{}.state$ expresses that the field of *claimed_semaphores* matches the $SemaphoreID$ of $S_i$. The data type of *claimed_semaphores* is *semaphore_nest* which will be defined in Section 6.5. Their definition is given in Figure 4.4 on the next page. Each task has its own semaphore state array, which represents the task state for its semaphore queue.

The P($S$) and V($S$) primitives are defined as the one parameter case of **SL-P** and **SL-V**. In the case of **SP** and **SV**, P and V primitives should be defined in a different way. We will show an example of how **SL-P** and **SL-V** work as follows:

| **SL-P** and **SL-V** Algorithms |
|---|
| **SL-P**$(S_0; S_1; ....; S_{n-1}) \equiv$<br>**begin**<br>**if** $S_0 \geq 1 \wedge .... \wedge S_{n-1} \geq 1$<br>**then**<br>  **for** $i \leftarrow 0$ **to** n -1 **do**<br>    $S_i \leftarrow S_i - 1;$<br>  **endfor**<br>**else**<br>  $current\_task.claimed\_semaphores \leftarrow (S_0, S_1, ....., S_{n-1});$<br>  **for** i:= 0 **to** n-1 **do**<br>    **if** $S_i \geq 1$ **then**<br>      $current\_task.claimed\_semaphores(S_i)\hat{}.state \leftarrow ready;$<br>      $S_i \leftarrow S_i - 1;$<br>    **else**<br>      $enqueue\_priority\_descendent\_order(S_i.semaphore\_queue, current\_task);$<br>      $current\_task.claimed\_semaphores(S_i)\hat{}.state \leftarrow wait;$<br>    **endif**<br>  **endfor**<br>  $relinquish(current\_task);$<br>**endif**<br>**end** |
| **SL-V**$(S_0; S_1; .....; S_{n-1}) \equiv$<br>**var** $ht$ : $task$; { the highest task in queue}<br>**begin**<br>**for** $i \leftarrow 0$ **to** n-1 **do**<br>  **if** $S_i.semaphore\_queue \neq \{\,\}$ **then**<br>    $ht \leftarrow dequeue(S_i.semaphore\_queue, thehightest);$<br>    $ht.claimed\_semaphores(S_i)\hat{}.state \leftarrow ready;$<br>    **if** $\forall S_i \in ht.claimed\_semaphores\{ht.claimed\_semaphores(S_i)\hat{}.state = ready\}$<br>    **then** $run(ht);$<br>    {Tasks, all elements of their semaphore list of which is *ready*, are invoked to run.}<br>    **endif**<br>  **else** $S_i \leftarrow S_i + 1;$<br>  **endif**<br>**endfor**<br>**end** |

Figure 4.4: **SL-P** and **SL-V** Algorithms

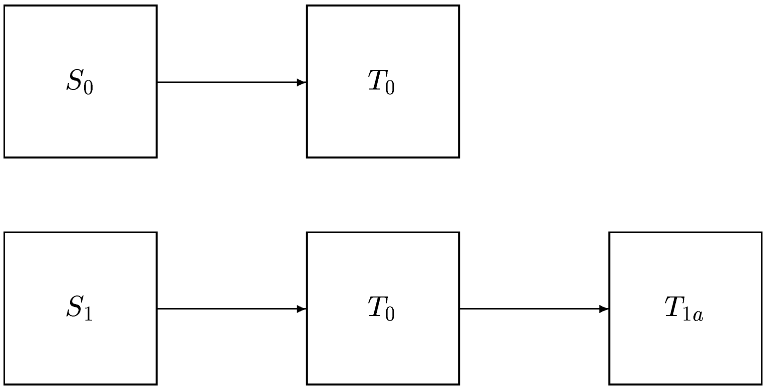


Figure 4.5: Queue Situations using **SL** operators

—*Implementation Using Simultaneous List Semaphore*—

$\boxed{\wp_0}$ $T_0 = \{$..............,**SL-P**$(S_0; S_1)$,.......,**SL-V**$(S_0; S_1)$,....$\}$

$\boxed{\wp_1}$ $T_1 = \{$....,**SL-P**$(S_1)$,............,**SL-V**$(S_1)$.....................$\}$

$\boxed{\wp_2}$ $T_1a = \{$......,**SL-P**$(S_1)$,........,**SL-V**$(S_1)$,....................$\}$

$\boxed{\wp_3}$ $T_2 = \{$.........,**SL-P**$(S_0)$,....,**SL-V**$(S_0)$,.......................$\}$

$(P_0 > P_1 = P_1a > P_2)$

1. $T_1$ is initiated and locks $S_1$.
2. $T_1a$ is initiated, claims $S_1$ and is blocked.
3. $T_2$ is initiated and locks $S_0$.
4. $T_0$ is initiated and claims $S_0$ and $S_1$ simultaneously. It is queued into both $S_0$ queue and $S_1$ queue. It is put on the top of both queues because it is the highest priority task in those queues. The claimed semaphores' states become (*wait*, *wait*)
5. $T_2$ releases $S_0$ and then the claimed semaphores' states of $T_0$ become (*ready*, *wait*).
6. $T_1$ releases $S_1$, then the claimed semaphores' states of $T_0$ becomes (*ready*, *ready*), then $T_0$ locks $S_0$ and $S_1$ simultaneously.

In the above example, the situation of semaphore queues at the stage 4 is illustrated in Figure 4.5 on this page, which shows that the highest priority task $T_0$ is placed on the top of both $S_0$ and $S_1$ semaphore queues.

It is required that the blocking time of the highest task $T_0$ is no more than the total number of working units of the semaphores in the list[1].

The blocking time is determined by the new simultaneous semaphore list operations themselves, while in the **SP** and **SV** primitives the blocking time is determined by the CPU scheduler. Moreover, the **SL-P** and **SL-V** primitives are not *busy waiting*.

[1]The highest task might wait for one working unit for each semaphore in the list

### 4.1.3 Summary

The first simultaneous semaphore primitives which operate multiple semaphores at the same time were proposed to increase the descriptive power of semaphore primitives. [**Maekawa87**], for example, shows how easily we can write a valid synchronisation program for the dining philosophers problem using simultaneous semaphores. However this has no concerns with real-time scheduling. The *simultaneous list semaphores* provide priority-based queue processing; scheduling based on priority is done within semaphore processing instead of CPU scheduling.

## 4.2 Colour Queues

The operations on semaphore lists should be carefully designed in order to avoid deadlocks. As a next step, the design of **SL** primitives can be improved to be deadlock free.

### 4.2.1 A Drawback of SL-P and SL-V Primitives

We consider the transition of the claimed semaphores' states according to the following example:

—*Deadlock by Simultaneous List Semaphores*—

$\boxed{\wp_0}$ $T_0 = \{$..........,$\mathbf{SL\text{-}P}(S_0; S_1; S_2)$,.......,$\mathbf{SL\text{-}V}(S_0; S_1; S_2)$,....$\}$

$\boxed{\wp_1}$ $T_1 = \{$........,$\mathbf{SL\text{-}P}(S_0; S_1; S_2)$,.......,$\mathbf{SL\text{-}V}(S_0; S_1; S_2)$,......$\}$

$\boxed{\wp_2}$ $T_2 = \{$....,$\mathbf{SL\text{-}P}(S_1)$,.....,$\mathbf{SL\text{-}V}(S_1)$,.................................$\}$

$\boxed{\wp_3}$ $T_3 = \{$..,$\mathbf{SL\text{-}P}(S_0)$,.....,$\mathbf{SL\text{-}V}(S_0)$,...................................$\}$

$(P_0 > P_1 > P_2 > P_3)$

1. $T_3$ is initiated and locks $S_0$.
2. $T_2$ is initiated and locks $S_1$.
3. $T_1$ is initiated, waits for $S_0$ and $S_1$, but locks $S_2$. The claimed semaphores' states of $T_1$ becomes (*wait*, *wait*, *ready*).
4. $T_0$ is initiated and waits for $S_0$, $S_1$ and $S_2$. The claimed semaphores' states of $T_0$ becomes (*wait*, *wait*, *wait*).
5. $T_3$ releases $S_0$. The claimed semaphores' states of $T_0$ become (*ready*, *wait*, *wait*).
6. $T_2$ releases $S_1$. The claimed semaphores' states of $T_0$ become (*ready*, *ready*, *wait*), this means $T_0$ and $T_1$ compete for the semaphores and become deadlocked.

This example shows a drawback of the **SL-P** and **SL-V** primitives. The delay of a high task is unbounded. The problem is caused by $T_0$ and $T_1$ locking the partial member of semaphores in the list. To avoid the problem, $T_0$ and $T_1$ should only acquire a right to lock the semaphores instead of really locking them. This situation represents a state waiting for locking a semaphore, which is different from the blocked state where a task cannot lock a semaphore until it is released. To realise the new wait state in the semaphore mechanism, a new queue will be introduced.

### 4.2.2 The Red and Blue Wait

As a temporary assumption, two states, *ready* and *wait*, were defined for the semaphore state list. However the *ready* state caused by the **SL-P** primitive and the *ready* state which is transited from the *wait* state by **SL-V** cannot be a true ready state which represents that the task did really lock. We therefor now introduce a new wait state to apply to this situation, called the *blue-wait*, where the task acquires a right to lock the semaphore in the list. On the other hand, we call the ordinary wait state the *red-wait* where the task does not acquire any right to lock the semaphore until it is released.

It is assumed that a semaphore has two types of queue; a *red queue* which is an ordinary semaphore waiting queue, and a *blue queue* which is invented for the semaphore list operation. The task in the *red queue* waits for the semaphore to be released. The task in the *blue queue* acquires a right to lock the semaphore. The task which is located in the *blue queues* of all member of the semaphores in the list can lock that list.

**SL-P** and **SL-V** primitives can be redefined so that instead of setting *ready*, **SL-P'** and **SL-V'** put a task into the *blue queue*, and instead of setting up *wait* and enqueueing its semaphore waiting queue, they put a task into the *red queue*.

——*Example of Two Wait States*——

$\boxed{\wp_0}$ $T_0 = \{$........,**SL-P'**$(S_0; S_1; S_2)$,.......,**SL-V'**$(S_0; S_1; S_2)$,.....$\}$
$\boxed{\wp_1}$ $T_1 = \{$.......,**SL-P'**$(S_0; S_1; S_2)$,.......,**SL-V'**$(S_0; S_1; S_2)$,......$\}$
$\boxed{\wp_2}$ $T_2 = \{$....,**SL-P'**$(S_1)$,.....,**SL-V'**$(S_1)$,...............................$\}$
$\boxed{\wp_3}$ $T_3 = \{$..,**SL-P'**$(S_0)$,.....,**SL-V**$(S_0)$,..................................$\}$
$(P_0 > P_1 > P_2 > P_3)$

1. $T_3$ is initiated and locks $S_0$.
2. $T_2$ is initiated and locks $S_1$.
3. $T_1$ is initiated, waits for $S_0$ and $S_1$ and locks $S_2$. The claimed semaphores' states of $T_1$ become (*red-wait*, *red-wait*, *blue-wait*).
4. $T_0$ is initiated and waits for $S_0$, $S_1$ and $S_2$. The claimed semaphores' states of $T_0$ becomes (*red-wait*, *red-wait*, *red-wait*)
5. $T_3$ releases $S_0$. The claimed semaphores' states of $T_0$ become (*blue-wait*, *red-wait*, *red-wait*).
6. $T_2$ releases $S_1$. The claimed semaphores' states of $T_0$ becomes (*blue-wait*, *blue-wait*, *red-wait*). $T_0$ and $T_1$ still seems to become deadlocked.

$T_0$ has only acquired rights to lock $S_0$ and $S_1$, and $T_1$ has only acquired a right to lock $S_2$. They do not really lock, so the situation is not real deadlock, but both $T_0$ and $T_1$ cannot lock the list and make progress. To solve the problem, assuming that $T_0$ has higher priority than $T_1$, when $T_0$ issues **SL-P'**, its claimed semaphores' states should become (*red-wait*, *red-wait*, *blue-wait*) from (*red-wait*, *red-wait*, *red-wait*) and that of $T_1$ should change (*red-wait*, *red-wait*, *red-wait*) from (*red-wait*, *red-wait*, *blue-wait*). This means that the higher task $T_0$, which is on the top of the *red queue* of $S_2$, steals the right from the lower task $T_1$ for locking $S_2$.

We are approaching a solution.

## 4.3 Colour Semaphores

We accordingly propose *colour semaphores* as a powerful descriptive real-time synchronisation protocol that solves the deadlock problem. Colour semaphores P and V primitives denoted by **CSL-P** and **CSL-V** respectively are defined in Figure 4.6 on the next page and in Figure 4.7 on page 70.

A colour semaphore has two priority ordered queues, whereas an ordinary semaphore has only one time-ordered queue. Each queue, called the *red queue* and the *blue queue*, are priority-descendent ordered queues which lead to the real-time properties.

A new mechanism is introduced in the P primitives denoted **CSL-P**. When a task which enters into the wait state has priority which is higher than the lowest task on the *blue queue*, it enters into the *blue queue* and the lowest task on the *blue queue* will be moved to the *red queue*. This exchange mechanism effectively solves the deadlock problem highlighted in the previous section. The higher priority tasks have privilege to hold semaphores in advance.

The V primitive denoted by **CSL-V** moves the tasks in the *red queue* of the list to the *blue queue* instead of the ready queue. Then only the tasks, all claimed semaphores of which are in the *blue queue*, can run. Tasks which are in at least one *red queue* of the list must not run.

The data structure of a colour semaphore, and operations on semaphore queues, are as follows:

**type** $colour\_semaphore =$
**record**
  $semaphore\_count : Integer;$
  $red\_queue : Priority\_Descendent\_Order\_Queue;$
  $blue\_queue : Priority\_Descendent\_Order\_Queue;$
**end** $colour\_semaphore;$
$enqueue\_priority\_descendent\_order(colour\_semaphore.red\_queue, task);$
$enqueue\_priority\_descendent\_order(colour\_semaphore.blue\_queue, task);$
$dequeue(colour\_semaphore.red\_queue, priority);$
$dequeue(colour\_semaphore.blue\_queue, priority);$

We will show that colour semaphores produce the powerful descriptive properties of real-time synchronisation by examples the following section.

*The P Primitive for Colour Semaphores*

```
CSL-P(S_0; S_1; ....; S_{n-1}) ≡
var ctcs : set_of_semaphores; {claimed semaphores of the current task}
begin
if S_0 ≥ 1 ∧ .... ∧ S_{n-1} ≥ 1
then
  for i ← 0 to n -1 do
    S_i ← S_i − 1;
  endfor
else
  ctcs ← current_task.claimed_semaphores ← (S_0, S_1, ......, S_{n-1});
  for i ← 0 to n-1 do
    if S_i ≥ 1 then
      S_i ← S_i − 1;
      current_task.claimed_semaphores(S_i)^.state ← blue-wait;
      enqueue(S_i.blue_queue, current_task);
    else
      { Exchange the current task with the lowest task in the blue queue}
      if current_task.priority > lowest_priority(S_i.blue_queue) then
        exchange_task ← dequeue(S_i.blue_queue, thelowest);
        enqueue_priority_descendent_order(S_i.blue_queue, current_task);
        current_task.claimed_semaphores(S_i)^.state ← blue-wait;
      else exchange_task ← current_task;
      endif
      enqueue_priority_descendent_order(S_i.red_queue, exchange_task);
      exchange_task.claimed_semaphores(S_i)^.state ← red-wait;
    endif
  endfor
  if ∃S_j ∈ ctcs{current_task.claimed_semaphores(S_j)^.state = red-wait}
  then relinquish(current_task);
  else ∀S_j ∈ ctcs{dequeue(S_j.blue_queue, current_task)}; {continue to run}
  endif
endif
end
```

Figure 4.6: The P Primitive for Colour Semaphores

*The P Primitive for Colour Semaphores*

**CSL-V**$(S_0; S_1; .....; S_{n-1}) \equiv$
**var** $run\_tasks : Set\_of\_task;$
$rht : task;$
**begin**
$run\_tasks \leftarrow \{\};$
**for** $i \leftarrow 0$ **to** n-1 **do**
{ For each semaphore of the list }
**if** $S_i.red\_queue \neq \{\}$
**then**
$rht \leftarrow dequeue(S_i.red\_queue, thehighest);$
$rht.claimed\_semaphores(S_i)\hat{}.state \leftarrow$ *blue-wait*;
$enqueue\_priority\_descendent\_order(S_i.blue\_queue, rht);$
**if** $\forall S_j \in rht.claimed\_semaphores\{rht.claimed\_semaphores(S_j)\hat{}.state =$ *blue-wait*$\}$
**then**
{ Tasks whose all claimed semaphores are in the *blue wait* can run }
$\forall S_j \in rht.claimed\_semaphores\{dequeue(S_j.blue\_queue, rht);\}$
$run\_tasks \leftarrow run\_tasks \cup rht;$
**endif**
**else** $S_i \leftarrow S_i + 1;$
**endif**
**endfor**
**if** $run\_tasks \neq \{\}$
**then** $\forall T_j \in run\_tasks\{run(T_j);\}$
{ Run tasks whose all all claimed semaphores are in the *blue wait*.}
**endif**
$current\_task.claimed\_semaphores \leftarrow nil;$
**end**

Figure 4.7: The V Primitive for Colour Semaphores

### 4.3.1 Examples of Colour Semaphores

Using a colour semaphore of the semaphore list of $S_0$, $S_1$ and $S_2$, the example with the above problem can be rewritten as follows. The claimed semaphores' states of the task $T_i$ is denoted by *Colour-*$T_i$(*claimed_semaphores*).

—*Implementation Using Colour Semaphores*—
$\boxed{\wp_0}$ $T_0$ = {.........,**CSL-P**($S_0; S_1; S_2$),.......,**CSL-V**($S_0; S_1; S_2$),.......}
$\boxed{\wp_1}$ $T_1$ = {.......,**CSL-P**($S_0; S_1; S_2$),.......,**CSL-V**($S_0; S_1; S_2$).........}
$\boxed{\wp_2}$ $T_2$ = {....,**CSL-P**($S_1$),.....,**CSL-V**($S_1$),..................................}
$\boxed{\wp_3}$ $T_3$ = {..,**CSL-P**($S_0$),.....,**CSL-V**($S_0$),....................................}
$(P_0 > P_1 > P_2 > P_3)$

1. $T_3$ is initiated and locks $S_0$.
2. $T_2$ is initiated and locks $S_1$.
3. $T_1$ is initiated, waits for $S_0$ and $S_1$ and locks $S_2$. *Colour-*$T_1$($S_0$,$S_1$, $S_2$) = (*red-wait*, *red-wait*, *blue-wait*).
4. $T_0$ is initiated and waits for $S_0$, $S_1$ and $S_2$. $T_0$ is higher than $T_1$, the exchange occurs. *Colour-*$T_0$($S_0$,$S_1$,$S_2$) = (*red-wait*, *red-wait*, *blue-wait*) and *Colour-*$T_1$($S_0$,$S_1$,$S_2$) = (*red-wait*, *red-wait*, *red-wait*).
5. $T_3$ releases $S_0$. *Colour-*$T_0$($S_0$,$S_1$,$S_2$) = (*blue-wait*, *red-wait*, *blue-wait*).
6. $T_2$ releases $S_1$. *Colour-*$T_0$($S_0$,$S_1$,$S_2$) = (*blue-wait*, *blue-wait*, *blue-wait*). then $T_0$ starts.

The situation of semaphore queues of $S_0$, $S_1$ and $S_2$ at the stage 4 is illustrated in Figure 4.8 on the next page.

Another example shows the cyclic usage, which would cause deadlock using an ordinary semaphore. However with colour semaphores, there is no problem to the benefit of the highest priority task.

—*Example of a Semaphore Chain*—
$T_0$ = {........,**CSL-P**($S_1$),.......,**CSL-V**($S_1$),.............}
$T_1$ = {......,**CSL-P**($S_1; S_0$),........,**CSL-V**($S_1; S_0$),....}
$T_2$ = {....,**CSL-P**($S_0$),........,**CSL-V**($S_0$),.................}
$(P_0 > P_1 > P_2)$

1. $T_2$ is initiated and locks $S_0$.
2. $T_1$ is initiated, claims $S_1$ and $S_0$, then *Colour-*$T_1$($S_1$,$S_0$) = (*blue-wait*, *red-wait*).
3. $T_0$ is initiated and claims $S_1$, then *Colour-*$T_1$($S_1$,$S_0$) = (*red-wait*, *red-wait*), then $T_0$ can lock $S_1$.

$T_0$ should not wait for two working units. Thus the colour semaphore **CSL-P**($S_1; S_0$) provides to work as *predictable blocking* for $T_0$.

The dining philosophers problem of Section 4.1.1 is now slightly changed for simplicity. Image each task occurs periodically in the following example. If colour semaphores did not provide the exchange mechanism, the highest task $T_0$ might be starved. Using colour semaphores with priority effectively avoids *starvation*.

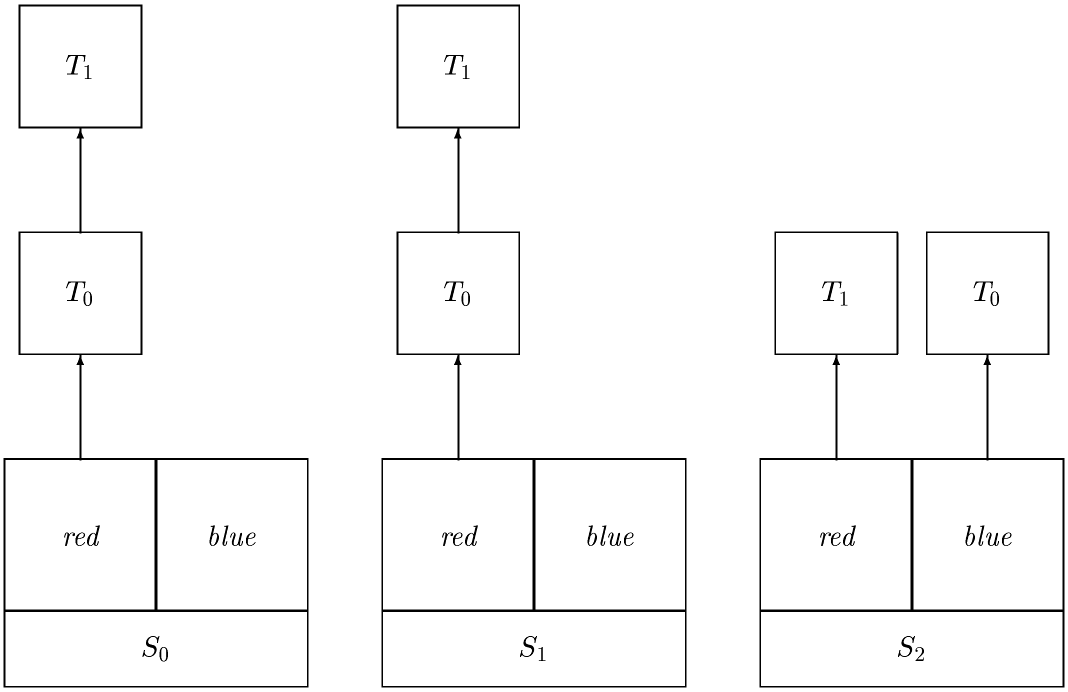


Figure 4.8: Queue Situations in Colour Semaphores

—*Example of Starvation Avoidance*—
$\boxed{\wp_0}$ $T_0 = \{$........,**CSL-P**$(S_2; S_0)$,.......,**CSL-V**$(S_2; S_0)$,......$\}$
$\boxed{\wp_1}$ $T_1 = \{$......,**CSL-P**$(S_1; S_2)$,.......,**CSL-V**$(S_1; S_2)$,........$\}$
$\boxed{\wp_2}$ $T_2 = \{$....,**CSL-P**$(S_0; S_1)$,.......,**CSL-V**$(S_0; S_1)$,..........$\}$
$(P_0 > P_1 > P_2)$

1. $T_2$ locks $S_0$ and $S_1$ simultaneously.
2. $T_1$ claims $S_1$ and $S_2$, then is blocked, *Colour-*$T_1(S_1,S_2)$ = (*red-wait*, *blue-wait*).
3. $T_0$ claims $S_2$ and $S_0$, then is blocked. *Colour-*$T_0(S_2,S_0)$ = (*blue-wait*, *red-wait*) and *Colour-*$T_1(S_1,S_2)$ = (*red-wait*, *red-wait*).
4. $T_2$ releases $S_0$ and $S_1$. $T_0$ can start, then *Colour-*$T_1(S_1,S_2)$ = (*blue-wait*, *red-wait*).
5. $T_0$ releases $S_2$ and $S_0$, then $T_1$ can start.

Consider a nested chain of semaphores. Exchanging the *blue-wait* is a means of acquiring the right hole which is bubbled up by a colour semaphore issued by a higher task. Using the above example, such a movement is described in Figure 4.9 on page 74, where $\times$ denotes locking, $\bullet$ denotes blocked (in the *red queue*) and $\circ$ denotes acquirable (in the *blue queue*). The starvation avoidance can be shown by moving up the acquisition of the right hole toward to the higher task.

The solution of the dining philosophers problem using colour semaphores, which means to allocate priority to each philosopher, is illustrated in Figure 4.10 on page 74. In the figure, there are two moves for acquiring the right holes. $T_2$ gets $S_2$ from $T_3$, and $T_0$ gets $S_4$ from $T_1$.

—*Dining Philosophers Problem*—
$\boxed{\wp_0}$ $T_0$ = {........,**CSL-P**$(S_4; S_0)$,.......,**CSL-V**$(S_4; S_0)$,......}
$\boxed{\wp_1}$ $T_1$ = {......,**CSL-P**$(S_3; S_4)$,...........,**CSL-V**$(S_3; S_4)$,....}
$\boxed{\wp_2}$ $T_2$ = {....,**CSL-P**$(S_2; S_3)$,.......,**CSL-V**$(S_2; S_3)$,..........}
$\boxed{\wp_3}$ $T_3$ = {...,**CSL-P**$(S_1; S_2)$,...........,**CSL-V**$(S_1; S_2)$,.......}
$\boxed{\wp_4}$ $T_4$ = {..,**CSL-P**$(S_0; S_1)$,.......,**CSL-V**$(S_0; S_1)$,...........}
$(P_0 > P_1 > P_2 > P_3 > P_4)$

1. $T_0$, $T_1$,$T_2$,$T_3$,$T_4$ are initiated.
2. $T_4$ gets chopsticks $S_0$ and $S_1$.
3. $T_3$ tries to get chopsticks $S_1$ and $S_2$, then is blocked.
4. $T_2$ tries to get chopsticks $S_2$ and $S_3$, then get rid of $S_2$ from $T_3$. It holds $S_2$ and $S_3$.
5. $T_1$ tries to get chopsticks $S_3$ and $S_4$, then is blocked.
6. $T_0$ tries to get chopsticks $S_4$ and $S_0$, then gets rid of $S_4$ from $T_1$.
7. $T_4$ releases $S_0$ and $S_1$. $T_0$ gets $S_0$ holding $S_4$ and $T_3$ holds $S_1$.
8. $T_3$ releases $S_2$ and $S_3$. $T_3$ gets $S_2$ holding $S_1$ and $T_1$ holds $S_3$.
9. $T_0$ releases $S_0$ and $S_4$. $T_1$ gets $S_4$ holding $S_3$.

The reason why *starvation* can be avoided using priority is that the holder of a chopstick is clearly determined and the task can be preempted. The highest task $T_0$ gets $S_4$ from the lower task $T_1$ and higher task $T_2$ gets $S_2$ from the lower task $T_3$. It reflects the movement of acquiring right holes, which explains the essential function of colour semaphores.

### 4.3.2 The Deadlock Problem with Colour Semaphores

Colour semaphores assume that a higher priority task will acquire a right of semaphores in a semaphore list in advance. Thus, if a lower task which issues the P primitives of semaphores of a semaphore nest separately and a higher task also holds a semaphore list, deadlock would occur. In an environment using colour semaphores, it is usually assumed that all P primitive sequences of semaphore nests should be issued simultaneously in the colour semaphore rather than separately. However here, we consider the case of separate issuing and discuss the problem caused by it. Events in the colour semaphore method are expressed by the three tuple as follows:
( $t^i_j$ $\in$Time, $P_i$ $\in$Priority, $SS_{kl}$ $\in$ SemaphoreList)
The simplest case of deadlock using colour semaphores is illustrated in Figure 4.11 on page 75. In this example, $T_1$ holds $S_0$, then $T_0$ claims $S_0$ and $S_1$ simultaneously, but $S_0$ is already locked by $T_1$, the semaphore state is *Colour*-$T_0(S_0,S_1)$ = (*red-wait*, *blue-wait*). The higher task $T_0$ acquires $S_1$ in advance. Then the lower task claims $S_1$, but cannot lock it because $T_0$ has already acquired $S_1$, therefore $T_0$ and $T_1$ become deadlocked.

There are two approaches to avoid this drawback of the colour semaphore. One approach is based on the assumption of the use of the colour semaphore. When using the

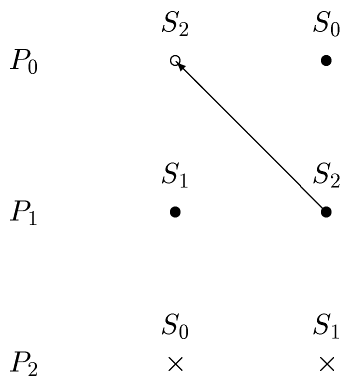


Figure 4.9: Move up the Acquisition in Colour Semaphores

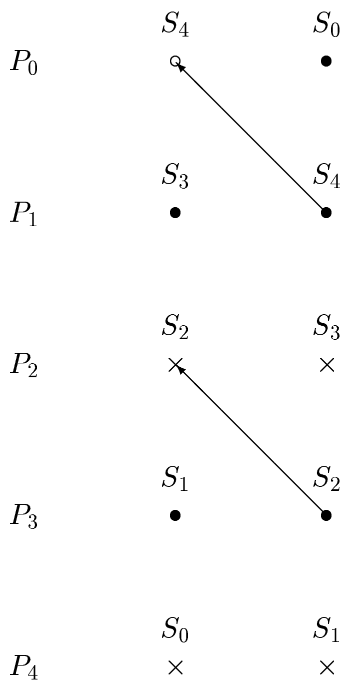


Figure 4.10: Move up the Acquisition in Dining Philosophers

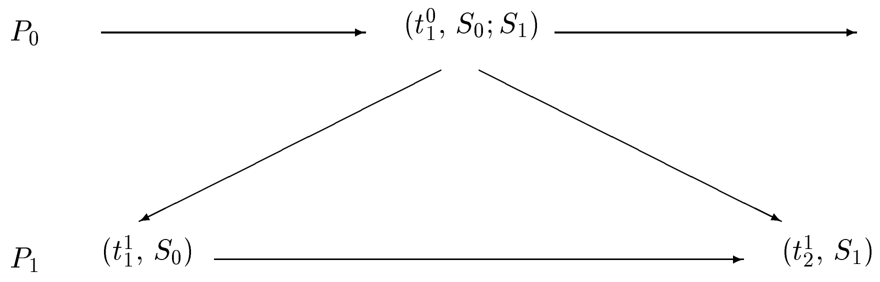


Figure 4.11: Deadlock of Colour Semaphores

colour semaphore, by ignoring the short distance of members of the semaphore nest, they are combined into a semaphore list, all P primitives of semaphores of semaphore nests should be issued simultaneously as the semaphore list in the colour semaphore, and not separately.

Another approach for avoiding this deadlock is that when a lower task issues P primitives of semaphores of a semaphore nest, semaphore acquisition in advance should be prohibited. To achieve this, the following rule needs to be added to the colour semaphore algorithm.

| *Additional Rule for Colour Semaphores (Conservative)* |
|---|
| Assuming the dependency list defined in Section 3.3,<br>when a lower priority task is holding a semaphore $S_k$ which is dependent on $S_i$,<br>in the colour semaphore which has both $S_k$ and $S_i$ in the semaphore list,<br>for $S_i$, the current task should be put only in the *red queue*.<br>The *blue queue* mechanism and the mechanism of exchanging queues should be ignored. |

The deadlock which the lower priority task causes with colour semaphores can be avoided by exploiting the above rule, which adopts dependency. However in the ordinary cases, the first approach based on the assumption of using the colour semaphore is acceptable. The second approach is too conservative.

### 4.3.3 A Comparison of SP and SV Primitives and Colour Semaphores

**SP** and **SV** primitives caused *starvation*, but they work in the example below.

$\boxed{\wp_0}$ $T_0 = \{$.........,$\mathbf{SP}(S_0; S_1; S_2)$,.......,$\mathbf{SV}(S_0; S_1; S_2)$,.....$\}$
$\boxed{\wp_1}$ $T_1 = \{$.......,$\mathbf{SP}(S_0; S_1; S_2)$,.......,$\mathbf{SV}(S_0; S_1; S_2)$,.......$\}$
$\boxed{\wp_2}$ $T_2 = \{$....,P($S_1$),..,..,V($S_1$),......................................$\}$
$\boxed{\wp_3}$ $T_3 = \{$..,P($S_0$),..,..,V($S_0$),........................................$\}$
$(P_0 > P_1 > P_2 > P_3)$

1. $T_3$ is initiated and locks $S_0$. $T_2$ is initiated and locks $S_1$. $T_1$ is initiated and waits for $S_0$. $T_0$ is initiated and waits for $S_0$.

2. $T_3$ releases $S_0$. $T_0$ and $T_1$ are reactivated. $T_0$ and $T_1$ wait for $S_1$.
3. $T_2$ releases $S_1$. $T_0$ and $T_1$ are reactivated. $T_0$ restarts and $T_1$ waits for $S_0$.

The above example works well. However after one task is added, **SP** and **SV** primitives do not work because the lower priority tasks lock the semaphore, instead of the higher priority tasks consuming the $S_0$ semaphore as follows:

$\boxed{\wp_0}$ $T_0 = \{$.........,$\mathbf{SP}(S_0;S_1;S_2)$,.......,$\mathbf{SV}(S_0;S_1;S_2)$,.....$\}$
$\boxed{\wp_1}$ $T_1 = \{$.......,$\mathbf{SP}(S_0;S_1;S_2)$,.......,$\mathbf{SV}(S_0;S_1;S_2)$,.......$\}$
$\boxed{\wp_2}$ $T_2 = \{$....,P$(S_1)$,..,..,V$(S_1)$,......................................$\}$
$\boxed{\wp_3}$ $T_3 = \{$..,P$(S_0)$,..,..,V$(S_0)$,........................................$\}$
$\boxed{\wp_4}$ $T_4 = \{$.,P$(S_0)$,..,..,V$(S_0)$,.........................................$\}$
$(P_0 > P_1 > P_2 > P_3 > P_4)$

Colour semaphores have several advantages compared with **SP** and **SV** primitives in terms of both simultaneous operations on a semaphore list and real-time properties.

### 4.3.4 Colour Semaphores with the Pure Plug-in Method

The behaviour of colour semaphores when adopting the *pure plug-in* method will now be considered using a slightly altered from the example in Section 3.1:

—*Example*—
$\boxed{\wp_0}$ $T_0 = \{$...........,P$(S_1)$,P$(S_0)$,......,V$(S_0)$,V$(S_1)$,................$\}$
$\boxed{\wp_1}$ $T_1 = \{$....,P$(S_1)$,.,P$(S_0)$,..DO-WORK1..,V$(S_0)$,V$(S_1)$,...$\}$
$\boxed{\wp_2}$ $T_2 = \{$........,P$(S_0)$,...DO-WORK2....,V$(S_0)$..................$\}$

The behaviour of the tasks is the same as in the original example in Section 3.1. $T_1$ should wait for two working units. The plug-in method is now applied to the semaphore $S_0$ of task $T_2$.

•*Plug-in method for the Example*
$\boxed{\wp_0}$ $T_0 = \{$...........,P$(S_1)$,P$(S_0)$,......,V$(S_0)$,V$(S_1)$,...........................$\}$
$\boxed{\wp_1}$ $T_1 = \{$.....,P$(S_1)$,....,P$(S_0)$,..DO-WORK1..,V$(S_0)$,V$(S_1)$,...........$\}$
$\boxed{\wp_2}$ $T_2 = \{$........,P$(S_1)$,P$(S_0)$,V$(S_1)$,...DO-WORK2....,V$(S_0)$,..........$\}$

1. $T_0$,$T_1$,$T_2$ are initiated and $T_1$ locks $S_1$.
2. $T_2$ claims $S_1$ and is blocked.
3. $T_0$ claims $S_1$ and is blocked.
4. $T_1$ claims $S_0$, executes DO-WORK1 and releases $S_0$ and $S_1$.
5. According to the semaphore queue operation policy, either of $T_0$ (priority order) or $T_2$ (time order) can restart.
6. if $T_2$ restarts, then it locks $S_0$ and $S_1$ and releases $S_1$. $T_0$ can restart, but is blocked by $S_0$.

According to the semaphore queue operation policy, in the worst case $T_0$ should wait for two working units.

A colour semaphore is applied to all semaphore lists. Therefore, the colour semaphore method is applied to the two sequences of semaphores, $S_1$,$S_0$ of $T_0$ and $S_1$, $S_0$ of $T_2$.

••*Colour Semaphore for the Example*

$\boxed{\wp_0}$ $T_0$ = {..........,**CSL-P**($S_1; S_0$),......,**CSL-V**($S_1; S_0$),....................................}
$\boxed{\wp_1}$ $T_1$ = {.....,**CSL-P**($S_1$),.,**CSL-P**($S_0$),..DO-WORK1..,**CSL-V**($S_0; S_1$),..........}
$\boxed{\wp_2}$ $T_2$ = {.........,**CSL-P**($S_1; S_0$),**CSL-V**($S_1$),...DO-WORK2....,**CSL-V**($S_0$),.....}

1. $T_0$,$T_1$,$T_2$ are initiated. $T_1$ locks $S_1$.
2. $T_2$ claims $S_1$ and $S_0$ and blocked.
3. $T_0$ claims $S_1$ and $S_0$, then is blocked.
4. $T_1$ claims $S_0$. It should hold $S_0$, otherwise the system would deadlock.
5. $T_1$ releases $S_0$ and $S_1$. $T_0$ can restart.

In the previous section 4.3.2, we proposed an algorithm using deadlock avoidance for the colour semaphore method, we can conclude that colour semaphores work effectively with the *plug-in* method.

The more beneficial example of the colour semaphore method used with the *plug-in* method is shown in example 4 in section 3.5. It is slightly changed to use a semaphore list as follows:

—*Example 4 in Section 3.5*—

$\boxed{\wp_0}$ $T_0$ = {.......,P($S_0$),P($S_1$),...,V($S_1$),V($S_0$),..........}
$\boxed{\wp_1}$ $T_1$ = {.....,P($S_1$),........,V($S_1$),..........................}
$\boxed{\wp_2}$ $T_2$ = {...,P($S_0$),........,V($S_0$),............................}

$T_0$ waits for two working units.

•*Upward Plug-in for Example 4 in Section 3.5*

$\boxed{\wp_0}$ $T_0$ = {......,P($S_0$),P($S_1$),..,V($S_1$),V($S_0$),..............................}
$\boxed{\wp_1}$ $T_1$ = {....,P($S_0$),P($S_1$),V($S_0$), DO-WORK1 ,V($S_1$),............}
$\boxed{\wp_2}$ $T_2$ = {..,P($S_0$), DO-WORK2 ,V($S_0$),.................................}

$T_1$ is blocked by the plugged-in semaphore. The semaphore queue operation which exploits time order prefers $T_1$ to $T_0$.

••*Colour Semaphores for example 4 in section 3.5*

$\boxed{\wp_0}$ $T_0$ = {......,**CSL-P**($S_0$;$S_1$),..,**CSL-V**($S_1; S_0$),......................................}
$\boxed{\wp_1}$ $T_1$ = {....,**CSL-P**($S_0; S_1$),**CSL-V**($S_0$), DO-WORK1 ,**CSL-V**($S_1$),.....}
$\boxed{\wp_2}$ $T_2$ = {..,**CSL-P**($S_0$), DO-WORK2 ,**CSL-V**($S_0$),................................}

When $T_0$ issues **CSL-P**$(S_0;S_1)$, *Colour*-$T_0(S_1,S_0)$ = (*red-wait*, *blue-wait*), and *Colour*-$T_1(S_1,S_0)$ = (*red-wait*, *red-wait*). $T_2$ releases $S_0$, $T_0$ can definitely restart.

In example 6 in Section 3.5, in the case of semaphore chains, several drawbacks of the *plug-in* method were pointed out. They can now be solved by applying colour semaphores to the *plug-in* method as follows:

—*Colour Semaphore to the Plug in of Example 6, Semaphore Chains, in Section 3.5* —

$\boxed{\wp_0}$ $T_0$ = {..,**CSL-P**$(S_0)$,.,**CSL-P**$(S_1)$,....DO-WORK0.,**CSL-V**$(S_1;S_0)$,...............................}

$\boxed{\wp_1}$ $T_1$ = {.......,**CSL-P**$(S_0;S_1)$,**CSL-V**$(S_0)$,..,**CSL-P**$(S_2)$,..DO-WORK1..,**CSL-V**$(S_2;S_1)$,..}

$\boxed{\wp_2}$ $T_2$ = {...............,**CSL-P**$(S_1;S_2)$,**CSL-V**$(S_1)$,...DO-WORK2.....,**CSL-V**$(S_2)$,..................}

1. $T_0$, $T_1$, $T_2$ are initiated.
2. $T_0$ locks $S_0$.
3. $T_1$ claims $S_0$ and $S_1$, but is blocked. *Colour*-$T_1(S_0,S_1)$ = (*red-wait*, *blue-wait*).
4. $T_0$ can lock $S_1$. *Colour*-$T_1(S_0,S_1)$ = (*red-wait*, *red-wait*).
5. $T_2$ claims $S_1$ and $S_2$, but is blocked. *Colour*-$T_2(S_1,S_2)$ = (*red-wait*, *blue-wait*).
6. $T_0$ executes DO-WORK0 and releases $S_1$. *Colour*-$T_1(S_0,S_1)$ = (*red-wait*, *blue-wait*). *Colour*-$T_2(S_1,S_2)$ = (*red-wait*, *blue-wait*).
7. $T_0$ releases $S_0$. *Colour*-$T_1(S_0,S_1)$ = (*blue-wait*, *blue-wait*). $T_1$ can restart and locks $S_1$. *Colour*-$T_2(S_1,S_2)$ = (*red-wait*, *blue-wait*).
8. $T_1$ locks $S_2$. *Colour*-$T_2(S_1,S_2)$ = (*red-wait*, *red-wait*). $T_1$ executes DO-WORK1 and releases $S_2$, $S_1$.
9. $T_2$ can restart.

—*Colour Semaphore to the Plug-in of Example 7 in Section 3.5*

$\boxed{\wp_0}$ $T_0$ = {......,**CSL-P**$(S_0)$,..,**CSL-P**$(S_1)$,..,**CSL-V**$(S_1)$,..,**CSL-V**$(S_0)$,..........}

$\boxed{\wp_1}$ $T_1$ = {....,**CSL-P**$(S_0;S_1)$,**CSL-V**$(S_0)$, DO-WORK1 ,**CSL-V**$(S_1)$,............}

$\boxed{\wp_2}$ $T_2$ = {..,**CSL-P**$(S_0;S_1)$,**CSL-V**$(S_0)$, DO-WORK2 ,**CSL-V**$(S_1)$,..............}

1. $T_2$ locks $S_1$.
2. $T_1$ claims $S_0$ and $S_1$, then is blocked. *Colour*-$T_1(S_0,S_1)$ = (*blue-wait*, *red-wait*).
3. $T_0$ claims $S_0$ and holds. *Colour*-$T_1(S_0,S_1)$ = (*red-wait*, *red-wait*).
4. $T_0$ claims $S_1$ and is blocked.
5. $T_2$ releases $S_1$. $T_0$ can locks it.

The blocking duration of $T_0$ is only one working unit, compared with two working units in the example in Section 3.5.

In Section 3.9 on page 49, the integration of the *downward plug-in* and the *upward plug-in* was discussed. In the example shown below, the P primitive part of the semaphore pair which is plugged in by either the *downward* or the *upward*, and the original P primitive can be associated to produce the semaphore list, which can be rewritten using colour semaphores.

—*Downward and Upward Plug-in*—

$\boxed{\wp_0}$ $T_0 = \{$..........,P($S_0$),..............,V($S_0$),.....................................$\}$

$\boxed{\wp_1}$ $T_1 = \{$.......,$\boxed{\text{P}(S_1)}$,P($S_0$),$\boxed{\text{V}(S_1)}$,..,P($S_1$),...,V($S_1$),V($S_0$).....$\}$

$\boxed{\wp_2}$ $T_2 = \{$.....,$\boxed{\text{P}(S_0)}$,P($S_1$),$\boxed{\text{V}(S_0)}$,..................,V($S_1$),..............$\}$

This has the drawback that deadlock occurs. Using colour semaphores, this problem would be resolved.

—*Colour Semaphore for Downward and Upward Plug-in*—

$\boxed{\wp_0}$ $T_0 = \{$..........,**CSL-P**($S_0$),.............,**CSL-V**($S_0$),........................................$\}$

$\boxed{\wp_1}$ $T_1 = \{$.......,**CSL-P**($S_1; S_0$),**CSL-V**($S_1$)..,**CSL-P**($S_1$),...,**CSL-V**($S_1; S_0$),.$\}$

$\boxed{\wp_2}$ $T_2 = \{$.....,**CSL-P**($S_0; S_1$),**CSL-V**($S_0$),..................,**CSL-V**($S_1$).,..............$\}$

Another example of integrating the *downward* and the *upward plug-in* which can be applied to colour semaphores is that of the intertask semaphore chains in Section 3.9 as follows:

—*Colour Semaphore for Intertask Semaphore Chain*—

$\boxed{\wp_0}$ $T_0 = \{$....,**CSL-P**($S_0$),.......,**CSL-P**($S_1$),......................,**CSL-V**($S_1; S_0$),......................$\}$

$\boxed{\wp_1}$ $T_1 = \{$....,**CSL-P**($S_0; S_2; S_1$),**CSL-V**($S_0; S_2$),.....,**CSL-P**($S_2$),.......,**CSL-V**($S_2; S_1$),....$\}$

$\boxed{\wp_2}$ $T_2 = \{$....,**CSL-P**($S_1; S_2$),**CSL-V**($S_1$).....,**CSL-P**($S_3$),...........,**CSL-V**($S_3; S_2$),...........$\}$

Only one deadlock condition is caused by introducing the *upward plug-in* method. This deadlock occurs in the following condition:
$(t_1^0, P_0, S_0) \lhd (t_1^1, P_1, S_1) \lhd (t_2^0, P_0, S_1) \wedge (t_1^1, P_1, S_1) \lhd (t_1^2, P_2, S_2) \lhd (t_2^0, P_0, S_1)$

However in most cases, the use of colour semaphores can avoid deadlock with the *plug-in* method, while the original version of the latter was threatened with deadlock. This example suggests a procedure for keeping the blocking duration of the higher task to only one working unit. The procedure is described as follows:

| *The Rule for Minimising Blocking Duration* |
|---|
| 1. Apply the *downward plug-in* method.<br>The lower priority task is blocked in the *inward* case.<br>2. Apply the *upward plug-in* method.<br>The middle priority task is blocked in the *forward* case.<br>The semaphore is free for a higher task.<br>3. Apply the *colour semaphore* method to avoid deadlock. |

The plug-in method refined to colour semaphores is also subject to deadlock as presented here. The protocols should be implemented like conservative two phase lock. That is, claim all together; once any semaphore has been released no other can be claimed [**Bacon93**].

### 4.3.5 Summary

When the distances between each member of a semaphore nest is short, for example, each semaphore is issued in the kernel mode, we should consider a new real-time synchronisation algorithm to solve the priority inversion problem instead of the *plug-in* method. The *plug-in* method is applicable when the distances between each member of a semaphore nest is long, for example, each member is issued in the user mode, or one is issued in the user mode and others are issued in the kernel mode.

In this chapter, we began with the simplest *simultaneous list semaphore operations*, which were introduced to make descriptive powers of synchronisation greater. Based on the investigation of the limitations of the simplest *simultaneous list semaphore operations*, more powerful *simultaneous list semaphore operations* with real-time properties were proposed. Furthermore, two priority descendent ordered queues; the *blue queue* and the *red queue* were introduced and the colour semaphores were invented. The colour semaphores have strong descriptive powers, for example, to write solutions for complex synchronisation problems like the dining philosophers problem. In a real-time environment where priority is the most important measurement, compared with the simple simultaneous semaphore operation the colour semaphores have the following beneficial properties:

1. They includes an effective scheduler in themselves, using the priority descendent ordered semaphore queues. Ordinary semaphores exploit the time antecedent ordered semaphore queue policy, so, they are less capable of including an effective scheduling in themselves. Moreover, the CPU scheduler has its own policy and it may be not necessary to obey the priority scheme to allocate processors. It is advantageous that semaphore primitives have a scheduling capability.

2. They avoid *busy waiting*, which **SP** and **SV** adopt. This is because the colour semaphore exploits two semaphore queues, the *red queue* and the *blue queue*, and the exchange mechanism.

3. They avoid *starvation* which occurs in ordinary solutions of the dining philosophers problem. This is because the higher priority task can usually keep semaphores in a semaphore list in advance.

4. They achieve the goal of providing the following real-time properties:

   (a) The blocking duration of a higher task which is blocked by semaphores held by lower tasks should be reduced, ideally to at most one *working unit*.

   (b) The colour semaphore has trouble with deadlock avoidance. However using only colour semaphores for semaphore nests or adding a rule which uses the definition of dependency to the original algorithm of colour semaphores, deadlock avoidance can be achieved.

# Chapter 5

# Light Loaded Multiprocessors

In the previous two chapters, it was assumed that multiprocessor systems had at least as many processors as the number of tasks. As a next step, we consider the situation that the number of tasks is slightly more than that of processors, which we call the *light load* situation. This is more realistic than the ideal situation in which there are at least as many processors as tasks.

The notion of light load is relative and the given semantics of task behaviour greatly influence the competitive situation between tasks and processors. A task with a high priority which uses semaphores, would relinquish its CPU by semaphore blocking. If the number of such tasks is increased, more tasks with lower priorities could run in the light load environment, even though the total number of tasks compared with the number of processors might be allowed to be increased. We call tasks which use semaphores, *coordinate tasks.* The numbers of tasks and processors are expressed as $\#(tasks), \#(processors)$, and the number of tasks which use semaphores is expressed as $\#(coordinatetasks)$. Consider the ratios as follows:

1.
$$\frac{\#(tasks)}{\#(processors)}$$

2.
$$\frac{\#(tasks) - L * \#(coordinatedtasks)}{\#(processors)}$$

In the second ratio, $0 < L < 1$ is a weight value. Instead of the first ratio, the second ratio is adopted to define the *light load.*

| *Definition of Light Load* |
|---|
| $\frac{\#(tasks) - L * \#(coordinatedtasks)}{\#(processors)} \leq n$ <br> $n = 2, 3, \ldots, small\ natural\ number$ |

The weight value $L$ shows the density of synchronisations in the system and includes I/O delays which decease processor times. This may reflect CPU demand. If the density of synchronisations is higher, then the weight value $L$ is nearer to 1. A small value of $n$, for example 2, 3, is considered the *light load*.

In the lightly loaded situation for multiprocessors, we need to consider a hybrid of the mechanisms for both multiprocessors and uniprocessors.

In this more realistic environment, compared to the ideal environment, the preemption mechanism can apply and the priority inheritance protocol can also apply again. We redefine the priority inheritance protocol so that it works for multiprocessors. This is based on the simple fact that while a higher task claims a semaphore which is already locked by a lower task, it must be blocked and relinquish the assigned processor; the lower task inherits the higher priority and it can preempt, or at least gain the relinquished processor. The priority inheritance protocol for multiprocessors can be applied again in a simple and useful form. This priority inheritance protocol for multiprocessors provides a potential solution for the abnormal phenomenon, called *priority inversion for multiprocessors*.

We also redefine preemption in multiprocessor environments.

In order to discuss preemption and priority inheritance for multiprocessors in an application of real-time synchronisation, we propose preemptive semaphores. This is based on the fact that when a higher task claims a semaphore which is already locked, it must be blocked and relinquish the assigned processor, which should in turn be allocated to one of the ready tasks. But when it is woken up, it preempts a lower running task. As a convenient tool of the preemptive mechanism related to semaphore operations, which include the work of processor allocator, we propose preemptive semaphores.

In this chapter, we discuss real-time scheduling algorithms, from which we choose a suitable real-time scheduling to test feasibility of our real-time synchronisation algorithms. We propose the earliest deadline first with assignment to the busiest processor algorithm, and assume it as the underlying real-time scheduling in this thesis.

## 5.1 Processor Allocator

We assume that there are $m$ processors, $\wp_0$, $\wp_1$,..., $\wp_{m-1}$, and $n$ tasks, $T_0$, $T_1$, ..., $T_{n-1}$ (where $m < n$). There are two types of processor allocator. First, the $m$ highest runnable tasks can be allocated to run on the $m$ processors. A task can be allocated to and run on any of processors which the system has, and therefore, does not have a particular host processor. This is referred to as *dynamic binding*. Secondly, a task can run on only one dedicated processor called the *host processor*. A task runs on its host processor when it becomes the highest priority runnable task for that processor. This is referred to as *static binding*. If after a task is allocated to a certain processor dynamically, the task can migrate from one processor to another during execution, which is referred to as *dynamic assignment*. If after a task is allocated to a certain processor, the task cannot migrate to another processor during execution, which is referred to as *static assignment*.

Let us consider the example of multiple tasks allocation under static binding. Assuming there are two processors $\wp_0$, $\wp_1$, and five tasks, $T_0$, $T_1$, $T_2$, $T_3$, $T_4$. $T_0$ and $T_4$

are allocated to $\wp_0$ and $T_1$, $T_2$ and $T_3$ are allocated to $\wp_1$. Consider the following example [**Rajkumar88**]. This example assumes the dedicated multiple task allocation policy. The following abnormal phenomenon occurs for multiprocessors and to solve this in multiprocessors is one of our goals.

—*Example of Priority Inversion for Multiprocessors*—
$\boxed{\wp_0}$ $T_0$ = {.....P($S_0$),....,V($S_0$),.............}
$\boxed{\wp_1}$ $T_1$ = {.......,P($S_1$),........,V($S_1$),.......}
$\boxed{\wp_1}$ $T_2$ = {......,P($S_1$),............,V($S_1$),....}
$\boxed{\wp_1}$ $T_3$ = {..,P($S_0$),.................,V($S_0$)....}
$\boxed{\wp_0}$ $T_4$ = {............................................}
$(P_0 > P_1 > P_2 > P_3 > P_4)$

1. $T_0$ and $T_3$ are initiated.
2. $T_3$ locks $S_0$.
3. $T_0$ claims $S_0$ and is blocked. $T_4$ gains $\wp_0$.
4. $T_2$ is initiated and preempts $T_3$.
5. $T_1$ is initiated and preempts $T_2$.

The behaviour of the above system shows that the highest priority task $T_0$ on $\wp_0$ should be blocked until $T_3$ and all of the tasks preempting it on $\wp_1$ have finished executing. This causes an infinite blocking duration of the highest task $T_0$. It is called *priority inversion for multiprocessors*. This is the drawback of the multiple task allocation system. One solution proposed is that the lower priority task in a critical section is prohibited from being preempted. However this is a very conservative solution. The other solution is to use a similar strategy to the uniprocessor case − the priority inheritance protocol − but for multiprocessors. This will be described later in this chapter.

—*Priority Inheritance*—

1. $T_0$, $T_3$ are initiated.
2. $T_3$ locks $S_0$.
3. $T_0$ claims $S_0$ and is blocked. $T_3$ inherits the priority of $T_0$. $T_4$ gains $\wp_0$.
4. $T_2$ cannot preempt $T_3$ because $T_3$ has priority of $T_0$. $T_3$ continues at priority $P_0$.
5. $T_3$ releases $S_0$. $T_0$ preempts $T_4$.

This priority inheritance for multiprocessors could avoid priority inversion. However if we exploit a static assignment policy, the example below shows that the priority inheritance for multiprocessors does not always work.

—*Example of Static Assignment and Priority Inheritance for Multiprocessors*—
$\boxed{\wp_0}$ $T_0$ = {..........................................}
$\boxed{\wp_0}$ $T_1$ = {....,P($S_0$),........,V($S_0$),..........}
$\boxed{\wp_1}$ $T_2$ = {......,P($S_0$),............,V($S_0$),....}
$(P_0 > P_1 > P_2)$

1. $T_2$ and $T_1$ are initiated.
2. $T_1$ locks $S_0$. $T_2$ claims $S_0$ and is blocked.
3. $T_0$ is initiated and preempts $T_1$. $T_2$ on $\wp_1$ should wait for the completion of $T_0$ on $\wp_0$. This is called *remote blocking*. In this case, priority inheritance does not work, the blocking duration of $T_2$ may very long.

The above abnormal phenomenon is caused by the policy of static assignment. If we exploit dynamic assignment policy, $T_1$ can migrate to $\wp_1$ and can execute the critical section. $T_2$ is not blocked by $T_1$.

### 5.1.1 Dynamic Real-Time Scheduling

Real-time scheduling algorithms are surveyed in [**Audsley90**]. A survey of the difference between static real-time scheduling and dynamic real-time scheduling algorithms has been described from the point of view of predictability, testability, resource utilisation, and extensibility in [**Kopetz93**].

A real-time scheduling algorithm is referred to as *static* if priorities are assigned to tasks once and for all. A static scheduling algorithm is also called a *fixed* priority scheduling algorithm. A real-time scheduling algorithm is referred to as *dynamic* if priorities of tasks might change from request to request. The rate monotonic algorithm is *static*, and the earliest deadline first algorithm is *dynamic* [**Liu73**]. The *static* ceiling protocol [**Sha90**] is based on the rate monotonic algorithms and the *dynamic* ceiling protocol is based on the earliest deadline first algorithm [**Chen91**].

We use the notation below in this section:
$A_i$ the arrival time of $T_i$, $S_i$ the start time of $T_i$, $C_i$ the worst-case computation time of $T_i$, $I_i$ the interval period time of $T_i$, $r_i$ the worst-case response time of $T_i$, $B_i$ the blocking time of $T_i$, $D_i$ the deadline time which is measured relative to the arrival time of $T_i$, $hp(i)$ the set of all tasks of higher priority than task $T_i$.
Each task is characterised by ($A_i$, $S_i$, $C_i$, $D_i$). A *sporadic* task is fully described by ($A_i$, $S_i$, $C_i$, $D_i$). A *periodic* task is described by ($A_i$, $C_i$, $D_i$, $I_i$).

The sufficient condition for schedulability of a set of n periodic tasks are shown as follows: The static ceiling protocol based on the rate monotonic algorithm:

$$\sum_{i=1}^{m} \frac{C_i}{I_i} + B_i \leq m(2^{\frac{1}{m}} - 1) \quad (1 < m < n)$$

The dynamic ceiling protocol based on the earliest deadline first algorithm:

$$\sum_{i=1}^{n} \frac{C_i + B_i}{I_i} \leq 1$$

Rajkumar suggested in [**Rajkumar93**] that Dhall and Liu [**Dhall78**] had shown that the rate monotonic algorithm performs badly for multiprocessors with dynamic binding.

For example, consider $n$ processors and $n+1$ periodic tasks. $C$ represents the computational time required and $I$ represents a period. $n$ tasks are identical with $C_i = 2$, and $I_i = 1000$. The $(n+1)^{th}$ task has $C = 1000$, and $I = 1001$. All tasks come in together. The identical $n$ tasks will be assigned higher priority than the $(n+1)^{th}$ task under both the rate-monotonic and the earliest deadline first scheduling and will execute on the $n$ processors. When the $(n+1)^{th}$ task gets into one of the processors to execute, it is already too late (its slack is 1, but it should wait for 2 units for one of the tasks to complete). The utilisation of the task set is $1000/1001 + 2n/1000$. Under this limit, $C_i$ and $I_i$ of the identical tasks are $2e$ and $T$ respectively, and the $C$ and $I$ of the last task are $T$ and $T+e$ respectively. The total utilisation as e goes to 0 is just 1 when deadlines begin to be missed. The processor allocation problem is NP-hard but static binding can be done offline to provide higher schedulable utilisation.

However recently, dynamic binding policies and dynamic assignment policies for multiprocessors were supported in [**Ramamritham89**] and [**McCann93**]. Current processors are sufficiently powerful that more elaborate dynamic scheduling is possible. When we investigate static or dynamic binding and assignment policies which have been proposed, we should consider the items below:

1. Static binding sometimes does not include the preemptive mechanism.
2. Semaphore blocking is similar to waiting for a processor.
3. The use of semaphores sometimes cancels the effect of preemption.

The simple form of dynamic binding is that if a task becomes runnable and if a processor is free, the processor is allocated to the task. A more thoughtful form of dynamic binding is that if there is a task $T_i$ which is not running and has slack $s_i$, then there should be a processor which will come free in less time than $s_i$. A task $T_j$ can be allocated to a processor $\wp_k$, if computational time is C$_j$ according to the following:

$$C_j \le \min_{i \in idletasks} s_i$$

Here, we exploit the *dynamic assignment* policy based discussion below:
For example, at a given time processor $\wp_0$ has 8 seconds of work scheduled for it. The time for the tasks assigned to $\wp_0$ adds up to 8 seconds. Now the execution time of a task $T$ is 2 seconds with deadline 11 seconds. The scheduler can give $T$ to $\wp_0$ with 1 second in hand. If there is one more processor $\wp_1$ which now has 5 second of work, in this case the scheduler can give $T$ to $\wp_1$ with 4 seconds in hand. By the observation above, the scheduler should assign a task to a processor which will have at least time in hand, other processors should be kept for future requests. It should maintain the greatest possible processing capacity left on the other processors which can be used for unassigned tasks.

The earliest deadline first algorithm works on the basis that at any time, among those tasks that have been requested and not yet fully serviced, the processor executes the one whose deadline is earliest. If no such task exists, the processor idles. The earliest deadline first algorithm was proposed by Liu and Layland [**Liu73**] in the case of periodic requests and fixed computation time; and was extended by Dertouzos [**Dertouzos74**] in the case of arbitrary requests and arbitrary computation time. A task will be assigned the highest

priority if the deadline of its current request is nearest, and will be assigned the lowest priority if the deadline of its current request is the furthest. The theorem 7 in [**Liu73**] suggested that
*For a given set of m periodic tasks for one processor, the earliest deadline first scheduling algorithm is schedulable if and only if*

$$\sum_{i=1}^{m} \frac{C_i}{I_i} \leq 1$$

As was pointed out above, the earliest deadline first scheduling algorithm is optimum, in the sense that if a set of tasks can be scheduled by any algorithm, it can be scheduled by the earliest deadline first scheduling algorithm. The theorem may be extended for multiprocessors such that:
*For a given set of m periodic tasks for n processors, the earliest deadline first scheduling algorithm is schedulable if*

$$\sum_{i=1}^{m} \frac{C_i}{I_i} \leq n$$

However the bin packing problem [**Johnson74**] suggested that the above is necessary but not sufficient.

From the above intuitive discussion, the earliest deadline first with assignment to the busiest processor which can meet deadlines is our underlying real-time scheduling in this thesis.
According to this scheduling policy, each processor asks the question when a new task is added to it, for which of those tasks already assigned to it will the earliest deadline first scheme work? That is, will the capacity inequality still hold? The answer may be "yes" for several processors, in which case choose the one with least slack and the capacity equation of which is nearest to one and which meets the deadline.

A rate is attached to each task. Each task $T_i$ may determine *instances*, denoted by $t_{im}$, { m = 0, 1, ......}. Such instances have deadlines. As the dynamic assignment policy, scheduler may be invoked at $t_{im}$. For simplicity, every task has the same instance $t_m$, { m = 0, 1, ......}. We use notations as follows: $at(\wp_j, t_m)$, the set of tasks allocated on the processor $\wp_j$; $C_{ij}[t_m]$, the computation time of task $T_i$ which is allocated to the processor $\wp_j$ during $t_m$; $C_i[t_m]$, the computation time of task $T_i$ during $t_m$.

$$C_{ij}[t_m] = \begin{cases} C_i[t_m] & T_i \text{ } on \text{ } \wp_j \\ 0 & otherwise \end{cases}$$

The computation time of $T_i$ is expressed as below:

$$C_i = \sum_{\forall \wp_j} \sum_{t_m = A_i}^{t_m < A_i + D_i} C_{ij}[t_m]$$

Let us choose a processor and allocate computation of $T_i$ in workload order at $t_m$:

$$workload(\wp_j, t_m) = \sum_{\forall i \in at(\wp_j, t_m)} C_{ij}[t_m]$$

Processors must satisfy the necessary condition at $t_n$ below:

$$processor\_laxity(\wp_j, t_n) = \sum_{\forall i \in at(\wp_j, t_n)} (D_i - \sum_{m=0}^{n} C_{ij}[t_m]) > 0$$

It is an open problem for us whether we will be able to analyse the schedulability of our real-time synchronisation protocols under the earliest deadline first with assignment to the busiest processor scheduling.

### 5.1.2 Advantages of Multiprocessors for Real-Time Systems

We should consider advantages and disadvantages of multiprocessor systems for real-time systems. To do so we investigate carefully the difference in functionality between:

1. a multiprocessor system which consists of, for example, five processors,

2. a uniprocessor system the performance of which is, for example, five times faster.

In order to compare these differences, we consider the following situations:
[**a locking task**] a lower priority task is holding a locked semaphore,
[**a blocked task**] a higher priority task claims the locked semaphore.
[**free tasks**] several tasks are woken up by external events.
At this stage, the higher priority task relinquishes its processor, then among a locking task and free tasks, the system keeps watching that each task is arbitrated whether it can get a processor or it is preempted. The outcome is as follows:

- [**uniprocessor systems**] Only the highest priority task among a locking task and free tasks can get the processor. The number of opportunities for preemption is much larger.

- [**multiprocessor systems**] The locking task may still be running. The relinquished processor may be assigned to one of the free tasks. There is a range of priorities for processor allocation. Therefore, the number of opportunities for preemption is smaller.

### 5.1.3 Preemption for Multiprocessors

In a multiprocessor environment a new definition of *preemption* should be considered. The new definition of preemption for the multiprocessor case is similar to the uniprocessor case. Two types of preemption are classified with regard to processor assignment as follows:

1. **Intraprocessor preemption** the preempted task will be reassigned to the same processor as before.

2. **Interprocessor preemption** the preempted task will be reassigned to a different processor than before.

With a static assignment policy, preemption occurs only among the tasks assigned to the same processor, this means that only intraprocessor preemption occurs. With a dynamic assignment policy, both intraprocessor and interprocessor preemption occur.

In this thesis, we adopted the earliest deadline first with assignment to the busiest processor policy. For future work, we had better consider on general form of preemption later in this thesis. We should define the concept of preemption fitted to both intraprocessor and interprocessor preemption for the dynamic real-time scheduling policy as below:

| *Definition of Preemption for Multiprocessors* |
|---|
| Relinquish a processor on which a lower task has been running and assign it to a higher task, then run the higher task. |

## 5.2 Preemptive Semaphores

As shown in the previous section, the V primitive has a connection with preemption. We consider a convenient semaphore algorithm for a higher task preempting a processor when it has been blocked by a lower task locking a semaphore. The goal of a new semaphore algorithm is that after a higher task has been blocked by the P primitive of the semaphore which a lower task is locking, the V primitive which will be issued by the lower task can obtain a processor for restarting the higher task. Moreover, a V primitive is in charge of the CPU scheduler on the current processor acting as a decentralised autonomous scheduling instead of depending on a centralised scheduling. Assuming that a task using synchronisations is assigned to a higher priority, a task without synchronisations is assigned to a lower priority.

Preemptive semaphores enable rapid preemption. Adding the priority inheritance protocol to preemptive semaphores, we can investigate the benefit of priority inheritance for multiprocessors for real-time synchronisation.

For the earliest deadline first with assignment to the busiest processor, both intraprocessor and interprocessor preemption occur in a processor, thus the algorithms of preemptive semaphores will be written in a general form.

Let us start with a typical example, where it is assumed that there are three processors and five tasks. This case is $\#(processors) \geq \#(coordinatetasks)$.

—*Five Tasks for Three Processors*—
$T_0$ = {...........,P($S_1$),.......,V($S_1$),......................................}
$T_1$ = {........,P($S_1$),.,P($S_0$),..DO-WORK1..,V($S_0$),V($S_1$),..}
$T_2$ = {.............,P($S_0$),...DO-WORK2....,V($S_0$)................}
$T_3$ = {...................................................................................}
$T_4$ = {...................................................................................}
$(P_0 > P_1 > P_2 > P_3 > P_4)$

1. $T_4$, $T_3$, $T_2$ are assigned to processors, $\wp_0$, $\wp_1$, $\wp_2$ respectively.
2. $T_1$ is initiated, preempts $T_4$ and gains $\wp_0$.
3. $T_0$ is initiated, preempts $T_3$ and gains $\wp_1$.
4. $T_1$ locks $S_1$.
5. $T_0$ claims $S_1$, but is blocked. The gained processor $\wp_1$ is assigned to task $T_3$.
6. $T_2$ locks $S_0$.
7. $T_1$ claims $S_0$, but is blocked. The gained processor $\wp_0$ is assigned to task $T_4$.
8. $T_2$ releases $S_0$. $T_1$ is invoked by preempting $T_4$ and runs on $\wp_0$.
9. $T_1$ locks $S_0$ and releases $S_0$.
10. $T_1$ releases $S_1$. $T_0$ is invoked by preempting $T_3$ and runs on $\wp_1$.

We propose a simple semaphore mechanism; *preemptive semaphores.* Preemptive semaphores use a similar semaphore queueing mechanism to that of colour semaphores. The basic concept is that a lower priority task which can get a processor by blocking a higher priority task should be linked to the *preemptive queue* of the semaphore which blocks the higher task. The lower task on the *preemptive queue* can be preempted by the V primitive and the assigned processor should be transferred to the higher task.

The simple preemptive semaphore has two queues:

1. A semaphore queue in the descendant priority order.
2. A *preemptive queue* in the ascendent priority order.

The algorithm for the simple preemptive semaphore is described in Figure 5.1 on the next page. The P primitive is expressed as **PP** and the V primitive is expressed as **PV**.

Let us examine the behaviour of the system using the simple case of five tasks and three processors.

—*Example for Simple Case of Five Tasks for Three Processors*—
$T_0$ = {............,$\mathbf{PP}(S_0)$,.................,$\mathbf{PV}(S_0)$,.............}
$T_1$ = {........,$\mathbf{PP}(S_0)$,.................,$\mathbf{PV}(S_0)$,.................}
$T_2$ = {.....,$\mathbf{PP}(S_0)$,.................,$\mathbf{PV}(S_0)$,....................}
$T_3$ = {.........................................................................}
$T_4$ = {.........................................................................}
$(P_0 > P_1 > P_2 > P_3 > P_4)$

1. $T_0$, $T_1$, $T_2$ are assigned to processors $\wp_0$, $\wp_1$, $\wp_2$ respectively.
2. $T_2$ locks $S_0$.
3. $T_1$ claims $S_0$ and is blocked. $\wp_1$ is assigned to $T_3$. $T_1$ is enqueued on the semaphore queue. $T_3$ is enqueued on the preemptive queue of $S_0$.
4. $T_0$ claims $S_0$ and is blocked. $\wp_0$ is assigned to $T_4$. $T_0$ is enqueued on the semaphore queue. $T_4$ is enqueued on the preemptive queue of $S_0$.
5. $T_2$ releases $S_0$. $T_4$ is preempted by $T_0$ which is assigned to $\wp_0$.
6. $T_0$ releases $S_0$. $T_3$ is preempted by $T_1$ which is assigned to $\wp_1$.

*The Algorithm of Preemptive Semaphores*

**PP**$(S_i) \equiv$
**begin**
**if** $S_i \geq 1$
**then**
  $S_i \leftarrow S_i - 1;$
**else**
  $enqueue\_priority\_descendent\_order(S_i.semaphore\_queue, current\_task);$
  $current\_task.process\_state \leftarrow$ **Blocked**;
  $new\_task \leftarrow cpu\_scheduler(run\_queue, thehighest);$
  $enqueue\_priority\_descendent\_order(S_i.preemptive\_queue, new\_task);$
  $free\_processor \leftarrow relinquish\_processor(current\_task);$
  $processor\_assign(free\_processor, new\_task); run(new\_task);$
**endif**
**end**

**PV**$(S_i) \equiv$
**begin**
**if** $S_i.semaphore\_queue \neq \{\,\}$ **then**
  $shighest\_task \leftarrow dequeue(S_i.semaphore\_queue, thehighest);$
  **if** $preemptive\_queue \neq \{\,\}$ **then**
    $plowest\_task \leftarrow dequeue(S_i.preemptive\_queue, thelowest);$
    **if** $plowest\_task.process\_state =$ **Running**
    **then** $preempt(plowest\_task, shighest\_task);$
    **else goto** $L1;$
    **endif**
  **else**
$L1:$
    $shighest\_task.process\_state \leftarrow$ **Ready**;
    $enqueue(run\_queue, shighest\_task);$
    $cpu\_scheduler();$
  **endif**
**else** $S_i \leftarrow S_i + 1;$
**end**

Figure 5.1: The Algorithm for Simple Preemptive Semaphores

### 5.2.1 Overflow Coordinate Cases

Consider the case in which $\#(processors) < \#(coordinatetasks)$. In this case, there exist two kinds of unsuitable situation for the simple preemptive semaphore, which is described in the examples below.

—*Predictable Processor Moving: 3 Processors and 5 tasks*—
$T_0 = \{............,\mathbf{PP}(S_0),.................,\mathbf{PV}(S_0),............\}$
$T_1 = \{........,\mathbf{PP}(S_0),.................,\mathbf{PV}(S_0),.................\}$
$T_2 = \{.....,\mathbf{PP}(S_1),.......................,\mathbf{PV}(S_1),..............\}$
$T_3 = \{..............,\mathbf{PP}(S_1),..................,\mathbf{PV}(S_1),.........\}$
$T_4 = \{......................................................................\}$
$(P_0 > P_1 > P_2 > P_3 > P_4)$

1. $T_0$, $T_1$, $T_2$ are assigned to processors $\wp_0$, $\wp_1$, $\wp_2$ respectively.
2. $T_2$ locks $S_1$. $T_1$ locks $S_0$.
3. $T_0$ claims $S_0$, then is blocked. $\wp_0$ is assigned to $T_3$. $T_0$ is enqueued on the semaphore queue and $T_3$ is enqueued on the preemptive queue of $S_0$.
4. $T_3$ is blocked by $S_1$. $\wp_0$ is assigned to $T_4$. $T_3$ is enqueued on the semaphore queue and $T_4$ is enqueued on the preemptive queue of $S_1$.
5. $T_1$ releases $S_0$. $T_0$ is woken up. But $T_3$ in the preemptive queue has no processor.

In the above case, a processor $\wp_0$ which should be predictably assigned to the higher task $T_0$ has been assigned to the lower task $T_4$ by the blocking of the other semaphore $S_1$. This situation can be solved by the method that the task in the preemptive queue should keep updated to register higher tasks.

—*Example of Processor Shortage for Assignment: 3 Processors and 5 tasks*—
$T_0 = \{............,\mathbf{PP}(S_0),.................,\mathbf{PV}(S_0),............\}$
$T_1 = \{........,\mathbf{PP}(S_0),.................,\mathbf{PV}(S_0),.................\}$
$T_2 = \{.....,\mathbf{PP}(S_0),.................,\mathbf{PV}(S_0),....................\}$
$T_3 = \{..............,\mathbf{PP}(S_0),..................,\mathbf{PV}(S_0),.........\}$
$T_4 = \{......................................................................\}$
$(P_0 > P_1 > P_2 > P_3 > P_4)$

1. $T_0$, $T_1$, $T_2$ are assigned to processors $\wp_0$, $\wp_1$, $\wp_2$ respectively.
2. $T_2$ locks $S_0$.
3. $T_1$ claims $S_0$ and is blocked. $\wp_1$ is assigned to $T_3$. $T_1$ is enqueued on the semaphore queue. $T_3$ is enqueued on the preemptive queue of $S_0$.
4. $T_0$ claims $S_0$ and is blocked. $\wp_0$ is assigned to $T_4$. $T_0$ is enqueued on the semaphore queue and $T_4$ is enqueued on the preemptive queue of $S_0$.
5. $T_3$ claims $S_0$ and is blocked. There is no task to be assigned to $\wp_1$, which is idle.
6. $T_2$ releases $S_0$. $T_4$ is preempted by $T_0$ which is assigned to $\wp_0$.
7. $T_0$ releases $S_0$. $T_1$ gains the idle processor $\wp_1$ which $T_3$ parted with.

8. $T_1$ releases $S_0$. $T_3$ is woken up, but there remains no processor.

In the above case, there remains no processor to be assigned to a coordinated task. The simple preemptive semaphore should be modified to address the two situations described above.

The modified preemptive semaphore is described in Figure 5.2 on the next page. We add the preemption mechanism for multiprocessors, which provides a decentralised autonomous scheduling, to the V primitive of the modified preemptive semaphores.

In the situation in which the number of tasks and that of processors are the same, we need not consider that the semaphore protocol should include its own processor assignment policy, but in the situation in which the number of tasks is more than that of processors, as shown in the previous example, the P and V primitives could include a connection with some sort of processor assignment policy. In a real-time system, especially, the V primitive has a connection with the following functions:

1. Preempt a processor which has been assigned to a lower priority task.
2. Assign it to the invoking task with high priority.

### 5.2.2 An Invariant for Preemptive Semaphores

We define the invariant for *preemptive semaphores* which will clear the state of system and processor assignment problem. The processor assignment problem is how precisely we can grasp the state of the whole system.

First, we directly define the invariant as follows:

| *An Invariant of Preemptive Semaphores* |
|---|
| The invariant of *a preemptive semaphore* $S$ is<br>either no task hold or awaits $S$,<br>or the task which owns $S$ has a processor. |

This is not the case when a task which claims $\mathrm{P}(S_0)\mathrm{P}(S_1)$ and locks $S_0$, waits on $S_1$. Adding to the waiting state;

| *An Invariant of Preemptive Semaphores* |
|---|
| The invariant of a *preemptive semaphore* $S$ is:<br>either free,<br>or a task which owns $S$ has got a processor or a task is awaiting another semaphore. |

The task states are classified into four states as follows:

1. Idle.
2. Actually running.
3. Ready to run without an assigned processor.
4. Awaiting a semaphore.

The task holding the *preemptive semaphore* is not in the idle, nor the ready.

*The Algorithm of the Preemptive Semaphore for Overflow Coordinate*

```
PP(S_i) ≡
begin
if S_i ≥ 1    then    S_i ← S_i − 1;
else
  enqueue_priority_descendent_order(S_i.semaphore_queue, current_task);
  current_task.process_state ← Blocked;
  new_task ← cpu_scheduler(run_queue, thehighest);
  if run_queue = { }
  then enqueue(idle_processor_queue, current_task.processor);
  else
    if ∃S_j{current_task ∈ S_j.preemptive_queue}
    then
      dequeue(S_j.preemptive_queue, current_task);
      enqueue_priority_descendent_order(S_j.preemptive_queue, new_task);
    else
      enqueue_priority_descendent_order(S_i.preemptive_queue, new_task);
    endif
    free_processor ← relinquish_processor(current_task);
    processor_assign(free_processor, new_task); run(new_task);
  endif
endif
end
```

```
PV(S_i) ≡
begin
if S_i.semaphore_queue ≠ { } then
  shighest_task ← dequeue(S_i.semaphore_queue, thehighest);
  if preemptive_queue ≠ { } then
    plowest_task ← dequeue(S_i.preemptive_queue, thelowest);
    if plowest_task.process_state = Running
    then preempt(plowest_task, shighest_task);
    else goto L1;
    endif
  else
L1 :
    if idle_processor_queue ≠ { }
    then
      slighest_task.process_state ← Running;
      processor_assign(free_processor, shighest_task); run(shighest_task);
    else
      if shighest_task.priority > thelowest_of_running.priority
      then preempt(thelowest_of_running, shighest_task);
      else shighest_task.process_state ← Ready; enqueue(run_queue, shighest_task);
      endif
    endif
  endif
else S_i ← S_i + 1;
end
```

Figure 5.2: The Algorithm for Modified Preemptive Semaphores

## 5.3 Preemptive Semaphores with the Priority Inheritance Protocol

In this environment, we modify the definitions of the *priority inheritance* protocol. In the environment of the multiprocessor case, if priority inheritance is applicable, the situation when a higher task is blocked by locking caused by a lower task is different from that in the uniprocessor case. The processor on which the higher task has been running is called the *current processor* which can be transferred to run another task because the higher task is blocked. There are two cases to do with the *locking task* when priority inheritance happens.

1. [**case 1**] The locking task is running on a processor.
2. [**case 2**] The locking task is ready in the run queue.

In [**case 1**], the *current processor* may be transferred to some ready task. In [**case 2**], according to the policy of processor assignment, *dedicated* or *dynamic* policy, processor assignment algorithms for the locking task are different:

1. [**dynamic policy**] Assign the *current processor* to the locking task.
2. [**dedicated policy**] 1. Assign an idle processor to the locking task, 2. If there are no idle processors, if the inherited priority was the highest, the locking task could get the dedicated processor, if not, wait for higher tasks to finish execution.

Later in this thesis, we exploit *dynamic policy* with a processor assignment algorithm.

| *Definition of priority inheritance for multiprocessors* |
|---|
| 1. Raise the priority of the locking task of the semaphore on which a higher task has been blocked to that of the higher task. |
| 2. The processor on which the higher task has been running is called the *current processor*.<br>(2.1) If the locking task is running, the *current processor* is assigned to another ready task.<br>(2.2) If the locking task is preempted, the *current processor* is assigned to the locking task. |

The priority inheritance facility can be united with the preemptive semaphore mechanism described in Fig 5.2 on page 93. The resulting *preemptive semaphore with priority inheritance* is described in Fig 5.3 on the next page and in Fig 5.4 on page 96.

—*Example of Processor Shortage for Allocation: 2 Processors and 3 tasks*—
$T_0 = \{$............,$\mathbf{PP}(S_0)$,..................,$\mathbf{PV}(S_0)$,.............$\}$
$T_1 = \{$.........................................................................$\}$
$T_2 = \{$.....,$\mathbf{PP}(S_0)$,.................,$\mathbf{PV}(S_0)$,....................$\}$
$(P_0 > P_1 > P_2)$

| *The P Primitive of Preemptive Semaphores with Priority Inheritance* |
|---|

**PP**($S_i$) $\equiv$
**begin**
**if** $S_i \geq 1$ **then** $S_i \leftarrow S_i - 1$;
**else**
  *enqueue_priority_descendent_order*($S_i$.*semaphore_queue*, *current_task*);
  *current_task.process_state* $\leftarrow$ **Blocked**;
  **if** $S_i$.*locking_task.priority* $<$ *current_task.priority* **then**
    $S_i$.*locking_task.original_priority* $\leftarrow$ $S_i$.*locking_task.priority*;
    $S_i$.*locking_task.priority* $\leftarrow$ *current_task.priority*; { priority inheritance }
    $S_i$.*locking_task.process_flag* $\leftarrow$ *inheriting*;
  **endif**
  *new_task* $\leftarrow$ *cpu_scheduler*(*run_queue*, *thehighest*);
  { *run_queue* for each processor or the system depending on the scheduling policy}
  **if** $S_i$.*locking_task.process_state* $=$ **Running**
  **then**
    **if** *run_queue* $= \{\}$
    **then** *enqueue*(*idel_processor_queue*, *current_task.processor*);
    **else**
      **if** $\exists S_j$\{*current_task* $\in$ $S_j$.*preemptive_queue*\}
      **then** *dequeue*($S_j$.*preemptive_queue*, *current_task*);
          *enqueue_priority_descendent_order*($S_j$.*preemptive_queue*, *new_task*);
      **else** *enqueue_priority_descendent_order*($S_i$.*preemptive_queue*, *new_task*);
      **endif**
      *free_processor* $\leftarrow$ *relinquish_processor*(*current_task*);
      *processor_assign*(*free_processor*, *new_task*); *run*(*new_task*);
    **endif**
  **else**
    *free_processor* $\leftarrow$ *relinquish_processor*(*current_task*);
    $S_i$.*locking_task.process_state* $\leftarrow$ **Running**;
    *processor_assign*(*free_processor*, $S_i$.*locking_task*);
    *run*($S_i$.*locking_task*);
  **endif**
**endif**
**end**

Figure 5.3: The P Primitive for Preemptive Semaphores with Priority Inheritance

*The V Primitive of Preemptive Semaphores with Priority Inheritance*

```
PV(S_i) ≡
begin
{Priority Inheritance }
if current_task.process_flag = inheriting
then
  current_task.prioriry ← current_task.original_priority;
  current_task.process_flag = nil
endif
if S_i.semaphore_queue ≠ { } then
  shighest_task ← dequeue(S_i.semaphore_queue, thehighest);
  if preemptive_queue ≠ { } then
    plowest_task ← dequeue(S_i.preemptive_queue, thelowest);
    if plowest_task.process_state = Running
    then preempt(plowest_task, shighest_task);
    else goto L1;
    endif
  else
L1 :
    if idle_processor_queue ≠ { }
    then
      shighest_task.process_state ← Running;
      processor_assign(free_processor, shighest_task);
      run(shighest_task);
    else
      if shighest_task.priority > thelowest_of_running.priority
      then preempt(thelowest_of_running, shighest_task);
      else shighest_task.process_state ← Ready;
          enqueue(run_queue, shighest_task);
      endif
    endif
  endif
else S_i ← S_i + 1; cpu_scheduler();
endif
end
```

Figure 5.4: The V Primitive for Preemptive Semaphores with Priority Inheritance

1. $T_1$, $T_2$ are assigned to processors $\wp_0$, $\wp_1$ respectively.
2. $T_2$ locks $S_0$.
3. $T_0$ preempts $T_2$ and runs on $\wp_1$. It claims $S_0$, then is blocked.
4. $T_2$ inherits the priority $P_0$, gets $\wp_1$ and runs on it. $T_2$ is put in the preemptive queue of $S_0$.
5. $T_2$ releases $S_0$, then the V primitive wakes up $T_0$, preempts $T_2$, assigns $\wp_1$ to $T_0$ and runs $T_0$.

The above example is typical in showing that the priority inheritance protocol for multiprocessors provides the processor assignment policy which assigns the processor relinquished by the higher blocked task to the lower locking task. Moreover, this shows the other effective function of preemptive semaphores in finding a processor for the woken-up higher task.

—*Example of Processor Shortage for Assignment: 2 Processors and 4 tasks*—
$T_0 = \{$............,$\mathbf{PP}(S_0)$,..................,$\mathbf{PV}(S_0)$,.............$\}$
$T_1 = \{$......................................................................$\}$
$T_2 = \{$......................................................................$\}$
$T_3 = \{$..........,$\mathbf{PP}(S_0)$,...................,$\mathbf{PV}(S_0)$,..............$\}$
$(P_0 > P_1 > P_2 > P_3)$

1. $T_2$, $T_3$ are assigned to processors $\wp_0$, $\wp_1$ respectively.
2. $T_3$ locks $S_0$.
3. $T_1$ preempts $T_3$ and runs on $\wp_1$.
4. $T_0$ preempts $T_2$ and runs on $\wp_0$.
5. $T_0$ claims $S_0$ and is blocked. $T_3$ which is locking $S_0$ inherits the priority $P_0$. $\wp_0$ is assigned to $T_3$ and runs on it. $T_3$ is put in the preemptive queue of $S_0$.
6. $T_3$ releases $S_0$, then the V primitive wakes up $T_0$, preempts $T_3$, assigns $\wp_0$ to $T_0$ and runs $T_0$.

The above example shows that if the preemptive semaphores did not provide priority inheritance, the processor $\wp_0$, which $T_0$ got, would have been assigned to $T_2$ rather than $T_3$. $T_3$ would wait for an unpredictable time, in consequence, the highest task $T_0$ should be blocked for an unpredictable time. This shows that priority inheritance for multiprocessors is able to avoid unbounded priority inversion in multiprocessors cases.

## 5.4 The Plug-in Method and Colour Semaphores Under Light Load

When the number of tasks is greater than that of processors, real-time scheduling algorithms which provide a priority allocation algorithm and processor allocation algorithms will recover. The *plug-in* method and *colour semaphores* have been invented by assuming the ideal condition that there are the same number of processors as tasks. We now consider whether or not the *plug-in* method and *colour semaphores* will be applicable in lightly loaded environments.

### 5.4.1 The Plug-in Method under Light Load

When the *plug-in* method is applied to two tasks, the task which plugs in its own semaphores is called the *plugging task* and the task which has semaphores to be plugged-in is called the *plugged task*. The plugging task and the plugged task should be assigned to distinct processors in order that the *plug-in* method can work effectively.

—*Processor Allocation for Plug-in*—
$\boxed{\wp_0}$ $T_0 = \{$....,P($S_0$),...,P($S_1$)....,V($S_1$),V($S_0$),............$\}$
$\boxed{\wp_1}$ $T_1 = \{$......,P($S_1$),............,V($S_1$),.........................$\}$
$\boxed{\wp_1}$ $T_2 = \{$.................................................................$\}$
$(P_0 > P_1 > P_2)$

•—*Plug-in for Processor Allocation*—
$\boxed{\wp_0}$ $T_0 = \{$....,P($S_0$),...,P($S_1$)....,V($S_1$),V($S_0$),............$\}$
$\boxed{\wp_1}$ $T_1 = \{$......,$\boxed{\mathrm{P}(S_0)}$P($S_1$),$\boxed{\mathrm{V}(S_0)}$,..........,V($S_1$),....$\}$
$\boxed{\wp_1}$ $T_2 = \{$..................................................................$\}$
$(P_0 > P_1 > P_2)$

1. $T_0$ and $T_1$ start.
2. $T_0$ locks $S_0$.
3. $T_1$ claims $S_0$ and is blocked. The processor $\wp_1$ is assigned to task $T_2$.

The above example suggests that more tasks could be allocated to the processor to which the *plugged task* has been allocated than to the processor to which the *plugging task* has been allocated.

—*Processor Allocation for Plug-in*—
$\boxed{\wp_0}$ $T_0 = \{$....,P($S_0$),...,P($S_1$)....,V($S_1$),V($S_0$),............$\}$
$\boxed{\wp_1}$ $T_1 = \{$.......,P($S_2$),...,P($S_3$).....,V($S_3$),V($S_2$),.........$\}$
$\boxed{\wp_1}$ $T_2 = \{$......,P($S_1$),............,V($S_1$),.........................$\}$
$\boxed{\wp_0}$ $T_3 = \{$............,P($S_3$),..........,V($S_3$),......................$\}$
$(P_0 > P_1 > P_2 > P_3)$

The above example shows that the allocation of plugging tasks and plugged tasks among which plugging matches are different to a single processor does not work well. The *plug-in* method was invented for the ideal condition, even though we have got several fruitful results for the complex problems of real-time synchronisation for multiprocessors, the direct application of the *plug-in* method for more realistic conditions is difficult. We find another new algorithm for lightly loaded environments, using any analogy with the essential mechanism of the *plug-in* method.

### 5.4.2 Colour Semaphores Under Light Load

We consider a very simple case. There are two processors and three tasks. $T_0$ and $T_1$ are allocated to $\wp_0$ and $T_2$ is allocated to $\wp_1$.

—*Example of Starvation Avoidance*—
$\boxed{\wp_0}$ $T_0 = \{$........,**CSL-P**$(S_2)$,.......,**CSL-V**$(S_2)$,..................$\}$
$\boxed{\wp_0}$ $T_1 = \{$......,**CSL-P**$(S_1; S_2)$,.......,**CSL-V**$(S_1; S_2)$,.........$\}$
$\boxed{\wp_1}$ $T_2 = \{$....,**CSL-P**$(S_0; S_1)$,.......,**CSL-V**$(S_0; S_1)$,...........$\}$
$(P_0 > P_1 > P_2)$

1. $T_2$ on $\wp_1$ and $T_1$ on $\wp_2$ are initiated.
2. $T_2$ locks $S_0$ and $S_1$.
3. $T_1$ claims $S_1$ and $S_2$ simultaneously, then is blocked. *Colour-*$T_1(S_1,\ S_2)$ = (*red-wait*, *blue-wait*).
4. $T_0$ on $\wp_1$ is initiated. $T_0$ claims $S_2$. *Colour-*$T_0(S_2)$ = (*blue-wait*). *Colour-*$T_1(S_1,\ S_2)$ = (*red-wait*, *red-wait*).

To compare with the *plug-in* method, the *colour semaphores* may work for uniprocessors, which means it may work under light loaded environments.

—*Colour semaphore: one processors and 4 tasks*—
$T_0 = \{$.........,**CSL-P**$(S_0; S_2)$,.......,**CSL-V**$(S_0; S_2)$,................$\}$
$T_1 = \{$.......,**CSL-P**$(S_1; S_2)$,.......,**CSL-V**$(S_1; S_2)$....................$\}$
$T_2 = \{$....,**CSL-P**$(S_1)$,.....,**CSL-V**$(S_1)$,...................................$\}$
$T_3 = \{$..,**CSL-P**$(S_0)$,.....,**CSL-V**$(S_0)$,.....................................$\}$
$(P_0 > P_1 > P_2 > P_3)$

1. $T_3$ is initiated and locks $S_0$.
2. $T_2$ is initiated, preempts $T_3$ and locks $S_1$.
3. $T_1$ is initiated, preempts $T_2$. It waits for $S_1$ and locks $S_2$. *Colour-*$T_1(S_1,\ S_2)$ = (*red-wait*, *blue-wait*). $T_2$ restarts.
4. $T_0$ is initiated, preempts $T_2$ and waits for $S_0$ and gets $S_2$ from $T_1$. *Colour-*$T_0(S_0,\ S_2)$ = (*red-wait*, *blue-wait*) and *Colour-*$T_1(S_1,\ S_2)$ = (*red-wait*, *red-wait*).
5. If $T_2$ restarts, it releases $S_1$, but $T_0$ gains $S_2$ because it is higher than $T_1$. *Colour-*$T_1(S_1,\ S_2)$ = (*blue-wait*, *red-wait*). $T_0$ and $T_1$ face a kind of deadlock. To solve the problem, $T_3$ should inherit the priority of $T_0$. Then $T_3$ restarts instead of $T_2$.
6. $T_3$ releases $S_0$. *Colour-*$T_0(S_0,S_2)$ = (*blue-wait*,*blue-wait*). then $T_0$ starts.

This example suggests that *colour semaphores* should provide *priority inheritance* in order to work in both uniprocessors and multiprocessors. While a higher task is claiming the **CSL-P**$(S_0; S_1; ....; Sn - 1)$ operation, all the tasks which are locking the semaphores which are also in the *read-wait* state have to inherit the highest priority.

## 5.5 Summary

In this Chapter, we started by investigating priority inversion for multiprocessors. We redefined priority inheritance for multiprocessors and extended the definition of preemption for multiprocessors. We considered a dynamic real-time scheduling called the earliest deadline first with assignment to the busiest processor.

We assumed the restricted multiprocessor environment called "light load", so preemptive semaphores were developed, integrating preemption and priority inheritance for multiprocessors.

Our final thought is how to develop a general form of real-time synchronisation protocols for multiprocessors rather than ideal forms and restricted forms of them.

# Chapter 6

# The Dependency Protocol for Multiprocessors

In Chapter 3, by introducing the *plug-in* method for an "ideal" situation for multiprocessors, the new principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"* was proposed. In Chapter 4, by introducing the colour semaphore, the priority ordered semaphore queue mechanism was shown to produces useful real-time properties. In Chapter 5, by introducing the preemptive semaphore for lightly loaded multiprocessors, the priority inheritance protocol for multiprocessors was defined.

In this chapter, these results from previous chapters will be integrated and a new protocol for real-time synchronisation is proposed. A new process state transition for real-time systems is proposed and the principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"* will be realised as the paused process state. Moreover, the *paused* process state will be realised by the priority descendent ordered semaphore queue mechanism.

## 6.1 The Dependency Protocol for Multiprocessors

There are two types of scheduling events which invoke processor assignment:

1. [**Preemption**] A higher task preempts a processor while a lower task is running. This event is *involuntary.*

2. [**Blocking**] A task claiming a semaphore which has already been locked is blocked, and relinquishes a processor. This event is *voluntary.*

The classification of scheduling events as *voluntary* and *involuntary* was described in [**Nakamura86**]. This idea is helpful in making precise process state transitions.

In the ideal situation, such that there are the same number of processors as tasks, we have investigated several kinds of situation using the *plug-in* method. We again use the definition of dependency from Section 2.6.1. However the dynamic dependency list becomes dynamic.

| *The Definition of Dependency* |
|---|
| $S_k$ is dependent on $S_i$ at priority $P$ if there is a task with priority $P$ which will seek to lock $S_i$ while holding $S_k$. We call $S_i$ the *parent semaphore*, denoted by $S_p$ or *GUARDIAN*, and $S_k$ is the *dependent semaphore*, denoted by $S_d$ or *WARD*. The general dependency list is dynamic and updated whenever tasks arrive and complete at the system. |

We use notations below:
$S_j \in S_D(S_i) = \{S_j | S_j \text{ is dependent on } S_i\}$
$S_j \in S_P(S_k) = \{S_j | S_k \text{ is dependent on } S_j\}$
$S_j \in GS_D(S_i) = \{S_j | S_j \text{ is transitive dependent on } S_i\}$
$S_j \in GS_P(S_k) = \{S_j | S_k \text{ is transitive dependent on } S_j\}$
GD represents the general dependency list, GD$[S_p]$ represents a set of transitive dependency semaphores of $S_p$, GD$[S_p](S_d)$ represents a set of priorities of tasks which claim the transitive dependent semaphore $S_d$ of $S_p$.

There are two cases for us to consider in the examples in Section 3.5 and 3.8. Observing the behaviour of programs using the *plug in* method, we should invent analogous algorithms by reconsidering carefully the mechanism of the *plug-in* method.

The dependency protocol for multiprocessor has the following features:

1. the *downward* and *upward plug-in* method for multiprocessors.
2. the priority descendent ordered semaphore queue.
3. the priority inheritance for multiprocessors.

### 6.1.1 A New Scheduling Rule

We discuss the algorithm in detail by defining the two cases as follows:

1. **Downward** *Inward* case:

   $$\boxed{(t_1^H, P_H, S_d) \lhd (t_1^l, P_l, S_p) \lhd (t_2^H, P_H, S_p)}$$

   A typical example of the *plug-in* method is as follows:

   •*Plug-in for example 3*
   $T_0$ = {.....,P($S_0$),........,V($S_0$),......................................................}
   $T_1$ = {......,P($S_1$),......,P($S_2$),..,V($S_2$),..,V($S_1$),............................}
   $T_2$ = {........,$\boxed{\text{P}(S_1)}$,P($S_2$),$\boxed{\text{V}(S_1)}$,.. P($S_1$),..,V($S_1$),..,V($S_2$),.....}

   Using this example, the rule for this case will be described below;
   While the dependent semaphore $S_d$ is locked by a task $T_H$ with priority $P_H$ and

before the parent semaphore $S_p \in S_P(S_d)$ is being locked,
every task $T_l \in \{T_i | P_i < P_H \wedge claim(T_i, S_P(S_d))\}$ which is running on a separate processor and about to claim the parent semaphore $S_p$ should stop just before the parent semaphore $S_p$.

If the higher task $T_H$ was preempted, the lower task $T_l$ could not run on some processor because it should also have been preempted, therefore, $T_l$ would not be unnecessarily blocked. In this case, the priority inheritance mechanism need not be exploited.

2. **Upward** *Forward* case:

$$\boxed{(t_1^L, P_L, S_p) \lhd (t_1^h, P_h, S_d) \lhd (t_2^h, P_h, S_p)}$$

•—*Upward Plug-in:Section3.8*—
$T_0 = \{$..........,P($S_0$),..............,V($S_0$),............................................$\}$
$T_1 = \{$.......,$\boxed{\mathrm{P}(S_1)}$,P($S_0$),$\boxed{\mathrm{V}(S_1)}$,..,P($S_1$),...,V($S_1$),V($S_0$),..........$\}$
$T_2 = \{$.....,P($S_1$),....................,V($S_1$),...........................................$\}$

While the parent semaphore $S_p$ is locked by a task with priority $P_L$
every task $T_h \in \{T_i | \exists S_d \in S_D(S_p) \wedge \neg locked(S_d) \wedge claim(T_i, S_d) \wedge P_i < max\{\mathsf{D}[S_p](S_d)\}\}$ which is about to claim the dependent semaphore $S_d \in S_D(S_p)$ should stop just before holding it. This situation is expressed as *stopbefore*$(T_h, S_d)$. The task locking $S_p$ which has lower priority $P_L$ than $max\{P_h | stopbefore(T_h, S_d)\}$, inherits the higher priority. The task which is about to claim the dependency semaphore $S_d$, with higher priority than $max\{\mathsf{D}[S_p](S_d)\}$, can lock $S_d$. The task holding $S_d$ with lower priority than $max\{\mathsf{D}[S_p](S_d)\}$, then inherits the priority. Only tasks claiming locked $S_d$ become blocked.

We should consider a limitation of the *upward plug-in* method, which was described in Section 3.8, that $P_L$ should be lower than $P_F(S_D(S_p))$ is a severe condition and should be relaxed. However if we return to the definition of the dependency protocol for uniprocessors, we can ignore this severe limitation and exploit the algorithms based on the general dependency list. We note below:
$max\{P_j | stopbefore(T_j, S_d)\} \leq max\{\mathsf{GD}[S_p](S_d)\}$

There are two possible approaches to relax the condition as follows:

(a) **[Approach 1]** *Allow lower task locking*
If the priority of the task which is claiming $S_d \in GS_D(S_p)$ is higher than $max\{P_j | stopbefore(T_j, S_d)\}$, then it locks $S_d$.
If not, the priority of the task, which is claiming $S_d$ and all transitive parent semaphores of which are not locked, is not higher than $max\{P_j | stopbefore(T_j, GS_D(S_p))\}$.
This is expressed as:

$T_l \in \{T_i | P_i < max\{P_j | stopbefore(T_j, S_d)\} \wedge claim(T_i, S_d) \wedge \neg locked(GS_P(S_d))\}$, then it can get $S_d$ and in addition inherits priority $max\{P_j | stopbefore(T_j, S_d)\}$

(b) **[Approach 2]** *Stop lower task locking*
If the priority of the task which is claiming $S_d \in GS_D(S_p)$ is higher than $max\{\mathsf{GD}[S_p](S_d)\}$ then it locks $S_d$.
If not, the priority of the task claiming $S_d$ is not higher than $max\{\mathsf{GD}[S_p](S_d)\}$ and it stops just before $S_d$. In addition, a task holding $S_P$ inherits the priority $max\{P_j | stopbefore(T_j, S_d)\}$. This rule is exploited by the dependency protocol for uniprocessors.

The first approach is *optimistic* in expecting the effect of the priority inheritance for multiprocessors. The second approach is a direct application of the dependency protocol for uniprocessors, but is not so *pessimistic* as the *ceiling protocol.* We exploit the second approach because the dependency protocol with priority inheritance protocol for multiprocessors is reasonably defined.

## 6.2 The Paused State

We need to define what "*A task T stops before a semaphore S* " means. This is one of the process states. We call this process state the **Paused** state. Using the concept of the paused state, it is said that "*a task T is paused for S*".

We call $max\{P_j | stopbefore(T_j, S_d)\}$ in the previous section the maximum paused value of $S_d$. The second rule in the previous section is that only a task higher than the maximum paused value can lock $S_d$.

Locking only one semaphore $S_p$ might cause multiple tasks to be paused for multiple corresponding semaphores $\{S_i | S_i \text{ is dependent on } S_p\}$. The problem is how the task in the pause state will behave.

Two possible scheduling approaches are considered as follows:

1. When a task locks a dependency semaphore $S_d$, all the tasks which are paused for $S_d$ are moved into the semaphore queue of $S_d$.
   In this approach, when a V primitive of $S_p$ is issued, there are two approaches.

   **1a** The highest task which is paused by each $S_d \in GS_D(S_p)$ may lock its $S_d \in GS_D(S_p)$, and the other tasks are moved to the blocked state in the semaphore queue of each $S_d \in GS_D(S_p)$. In this approach, the V primitive of the parent semaphore operates the semaphore queues of the dependency semaphores in a multiprocessor environment.

   **1b** All paused tasks caused by locking $S_p$ still remain in the paused state. This approach has the drawback that several tasks with middle level priority remain in the pause state even after they become be higher that those in the ready and preemptive states. This causes the effectiveness of priority inheritance to decrease .

2. If a task has locked a dependency semaphore $S_d$, all the tasks which are paused for $S_d$ remain in the paused state. This approach has a drawback that there is only one scheduling event which causes multiple tasks in the paused state to leave the paused state. The situation in which one task locks the semaphore and issues its V primitive is the only scheduling event which causes the transition from the paused state. The same situation occurs as in **1b** above.

We should therefor attempt to exploit approach **1a**.

## 6.3 Dependency Semaphores for Multiprocessors

There are several problems in implementing the paused state:

1. How do the paused tasks know which semaphores they are paused for?
2. Should a semaphore prepare the *paused queue* to solve the problem of identifying its own semaphore?
3. What scheduling events should cause the transition from the paused state to either the blocked or the ready state?
4. The scheduling events which cause the transition from the paused state have indirect control of the semaphore. How can we deal with such indirect control?

Our aim is that the paused state can be implemented by the priority descendent ordered semaphore queue. Consider the following situation for example:

The paused state is defined such that if a task is about to claim a semaphore which is busy for indirect reasons it should not enqueue on the semaphore but be paused. The paused tasks are all released from the paused state when the semaphore is freed, and presumably the highest priority task starts first and gets the semaphore. This was why we created the paused state. Notice that it is not certain that this will work correctly in a multiprocessor system. If several tasks go from paused to active at the same time, and all start to run, as can happen in a multiprocessor system, there is no obvious guarantee that the highest priority task will be the one that gets the semaphore first. We defined priority based semaphores where queues are held in descending order of task priority. In this case we have the paused state as a variant of enqueueing a semaphore queue; an attempt to claim a semaphore that is in use leads to enqueueing, but the queue is always maintained in priority order. So when the semaphore is freed it will be definitely given to the higher waiting task. This achieves the same result as the pausing system without the multiprocessor problem described above.

According to the above discussion, we try to implement the paused state using the priority descendent ordered semaphore queue. The algorithm of dependency semaphores in general form is described in Fig 6.1 on the next page and in Fig 6.2 on page 107. Here the initial semaphore count is assumed to 1.

Tasks lock semaphore nests dynamically, therefore, it is difficult to determine which semaphore nest a task is locking. This means that to implement the function *nest-in*$(task, semaphore)$ requires a special mechanism. We should therefore try to find an

*The P Primitive of Dependency Semaphores for Multiprocessors*

**DP**$(S_i, Snest : semaphore\_nest) \equiv$
**var** $ct : task$ {the current task}
**begin**
$ct \leftarrow current\_task;$
**if** $Snest \neq nil$ **then** $enqueue(S_i.nest\_queue, (ct, Snest));$ **endif**
**if** $S_i \geq 1$ **then**
  { *Downward plug-in* case }
  $S_{d_m} \leftarrow \{S_d |\ max_{S_d \in S_D(S_i) \wedge locked(S_d)} \mathsf{D}[S_i](S_d)\}$
  **if** $S_{d_m} \neq nil \wedge\ T_H \leftarrow holder(S_{d_m}) \wedge\ nest\_in(T_H, S_i)$
    $\wedge T_H.priority > ct.priority\ \wedge ct.processor \neq T_H.processor$
  **then**
    $ct.process\_state \leftarrow$ **Paused**; $ct.claimed\_semaphore \leftarrow S_i;$
    $enqueue\_priority\_descendent\_order(S_i.semaphore\_queue, ct);$
    $relinquish\_processor(ct); cpu\_scheduler();$
  **endif**
  $S_{p_m} \leftarrow \{S_p |\ max_{S_p \in GS_P(S_i) \wedge locked(S_p)} \mathsf{GD}[S_p](S_i)\}$
  **if** $S_{p_m} \neq nil \wedge T_L \leftarrow holder(S_{p_m}) \wedge T_L \neq ct \wedge ct.priority \leq max\{\mathsf{GD}[S_{p_m}](S_i)\}$
  **then**
    $ct.process\_state \leftarrow$ **Paused**; $ct.claimeded\_semaphore \leftarrow S_i;$
    $enqueue\_priority\_descendent\_order(S_i.semaphore\_queue, ct);$
    **if** $T_L.priority < ct.priority$ **then**
      $T_L.original\_priority \leftarrow T_L.priority; T_L.priority \leftarrow ct.priority;$
      $T_L.process\_flag \leftarrow$ *inheriting*;
    **endif**
    $relinquish\_processor(ct); cpu\_scheduler();$
  **endif**
  { Lock $S_i$ }
  $S_i \leftarrow S_i - 1;$
**else**
  $enqueue\_priority\_descendent\_order(S_i.semaphore\_queue, ct);$
  $ct.process\_state \leftarrow$ **Blocked**
  **if** $S_i.locking\_task.priority < ct.priority$ **then**
    $S_i.locking\_task.original\_priority \leftarrow S_i.locking\_task.priority;$
    $S_i.locking\_task.priority \leftarrow ct\_task.priority;$ { priority inheritance }
    $S_i.locking\_task.process\_flag \leftarrow$ *inheriting*;
  **endif**
  $relinquish\_processor(ct); cpu\_scheduler();$
**endif**
**end**

Figure 6.1: The P Primitive for Dependency Semaphores for Multiprocessors

*The V Primitive of Dependency Semaphores for Multiprocessors*

```
DV(S_i, Snest : semaphore_nest) ≡
var ct : task; {the current task} begin
ct ← current_task;
{ Priority Inheritance }
if ct.process_flag = inheriting
then ct.prioriry ← current_task.original_priority;
      current_task.process_flag = nil
endif
if Snest ≠ nil then dequeue(S_i.nest_queue, (ct, Snest)); endif
if S_i.semaphore_queue ≠ { } then
  shighest_task ← dequeue(S_i.semaphore_queue, thehighest);
  runnable(slighest_task);
else S_i ← S_i + 1;
  { Restore the tasks in the paused state to the blocked state }
  forall S_d ∈ GS_D(S_i) do
    if S_d ≥ 1 ∧ S_d.semaphore_queue ≠ { } then
      runnable(dequeue(S_d.semaphore_queue, thehighest));
    endif
    forall T_i ∈ S_d.semaphore_queue do
      if  T_i.process_state = Paused
      then T_i.process_state ← Blocked;
      endif
    endforall
  endforall
endif
end
```

```
Procedure runnable(t : task);
begin
  if idle_processor_queue ≠ { }
  then
    t.process_state ← Running;
    processor_assign(free_processor, t); run(t);
  else
    if t.priority > thelowest_of_running.priority
    then preempt(thelowest_of_running, t);
    else t.process_state ← Ready;  enqueue(run_queue, t);
  endif
end
```

Figure 6.2: The V Primitive for Dependency Semaphores for Multiprocessors

algorithm in which the outermost P and V primitives of a semaphore nest declare their semaphore nests explicitly as their parameters. An example of the use of dependency semaphores with explicit declaration is shown as below:

—*Dependency Semaphore*—
$T_0$ = {......,**DP**($S_1$, *nil*),.......,**DV**($S_1$, *nil*),.........................................................}
$T_1$ = {...,**DP**($S_1$, $S_1$; $S_0$),.,**DP**($S_0$, *nil*),..DO-WORK1..,**DV**($S_0$, *nil*),**DV**($S_1$, $S_0$; $S_1$),..}
$T_2$ = {........,**DP**($S_0$, *nil*),...DO-WORK2....,**DV**($S_0$, *nil*)............................................}

An essential feature of the dependency semaphore is to note the nested semaphore explicitly in the nested P primitives. We think this notation is acceptable. This avoids the complexities of treating semaphore dependency. However in many practical cases, a user task may issue a system call which may contain a nested critical section which the user may not know. Therefore this convention is limited.

There are a variety of uses of real-time synchronisation. In the case of short lived synchronisation, for example to realise locking for simple shared data structures, the dependency semaphore is too heavy weight and not applicable. There are long lived synchronisations which are caused: (1) by devices, for example discs and network devices and (2) by complex data structures. For example, the file directory search on the disc-less UNIX workstation needs long lived synchronisations of both discs and network devices. In the case of long lived synchronisations, the dependency semaphore is applicable. In the case that synchronisations use semaphore nests, the dependency semaphore is also applicable.

## 6.4 Process State Transition

We have considered problems to do with precisely describing the algorithm, for example, *"stop just before the semaphore"*. The process state transition diagram is the most important and effective mechanism to provide precise control over the behaviour of tasks. We propose a new process state transition to be applied to both real-time scheduling and real-time synchronisation. We define six process states, and two new states are introduced for real-time scheduling.

For Time Sharing Systems, the NIL state indicates that a task was not created, for example by the *fork* system call in UNIX, A running task exits into the NIL state. In real-time systems, a task has not arrived at the system, and it has been not allocated to a processor. When a task leaves the system, it enters into the NIL state.

A task is claiming a semaphore; if satisfied, it can lock the semaphore. If not, the P primitive of real-time synchronisation protocols caused semaphore blocking. In the real-time environment, as a scheduling event, semaphore blocking causes two process state transitions: from the running state to the blocked and from the running state to the paused state. The introduction of the paused state is quite important because in the paused state a task has not executed a semaphore, so the semaphore is free. Therefore a higher task is able to lock it, otherwise it would be blocked. Precisely, this blocking is not issued directly, but indirectly by parent semaphores.

In the blocked state, the task has executed a locked semaphore and will be woken into the ready state. Usually as in Time Sharing Systems, for example UNIX, the priority of the blocked tasks is raised and it will become higher in the ready state and the woken task can get a processor as soon as possible in order to process a transaction for the external event [**Bach86**, **Leffer89**]. In this thesis, we did not consider the dynamic process priority policy which the system should manage. Moreover, from the point of view of the earliest deadline first with assignment to busiest processor algorithm, priority should be redefined according to the deadline at instances.

When a task, the priority of which is higher than the lowest task in the running state, appears in the ready state, preemption occurs and this preempts the lowest task in the running state. The lowest task in the running state enters the preempted state and the assigned processor is transferred to the new task.
$preempt(current\_task, high\_task)$ function in Fig 5.1, Fig 5.2, Fig 5.3,Fig 5.4 and Fig 6.2 is described below:

**procedure** $preempt(thelowest\_of\_running : task, high\_task : task)$
**var** $free\_processor : processorID;$
**begin**
  $free\_processor \leftarrow relinquish\_processor(thelowest\_of\_running);$
  $proocessor\_assign(free\_processor, high\_task);$
  $thelowest\_of\_running.process\_state \leftarrow$ **Preempted**;
  $high\_task.process\_state \leftarrow$ **Running**;
  $enqueue(ready\_queue, thelowest\_of\_running); run(high\_task);$
**end**

This algorithm is based on the type of interprocessor preemption, and is described in general form for future work.

For the type of intraprocessor preemption, the procedure of $preemption()$ can be written in a more restricted form as follows:
The condition part becomes:
**if** $shighest\_task.priority > running\_task(shighest\_task.processor).priority$
The procedure of $preemption(high\_task)$ for intraprocessor preemption is as follows:

**procedure** $preempt(high\_task : task)$
**begin**
  $relinquish\_processor(running\_task(high\_task.processor));$
  $proocessor\_assign(high\_task.processor, high\_task);$
  $running\_task(high\_task.processor).process\_state \leftarrow$ **Preempted**;
  $high\_task.process\_state \leftarrow$ **Running**;
  $enqueue(ready\_queue, running\_task(high\_task.processor)); run(high\_task);$
**end**

When a task in the preempted state returns to the running state, there are three events which could cause the transitions.

1. [**Blocking**] A task in the running state claims a semaphore, but is blocked. It enters the blocked state and relinquishes a processor. If the priority of a preempted task is higher

| *The Definition of Process States* |
|---|
| 1. [**Running**]<br>A task gets a processor and runs on it.<br>A task may be locking semaphores. |
| 2. [**Ready**]<br>A task is waiting for a processor to be assigned. |
| 3. [**Preempted**]<br>A low priority task is running, then a higher task becomes ready.<br>The processor which has been assigned is transferred to a higher ready task.<br>The priority of *preempted* tasks remains the same. |
| 4. [**Blocked**]<br>A task has claimed a semaphore, but it has already been locked,<br>then it is put in the semaphore queue.<br>The processor which the task holds is relinquished. |
| 5. [**Paused**]<br>A task has claimed a semaphore whose parent semaphore is locked.<br>The task stops just before the semaphore, and does not hold the semaphore.<br>The processor which the task held is relinquished.<br>The priority of the *paused* task remains the same. |
| 6. [**NIL**]<br>A task in the NIL state is not created, in other words it is not initiated. |

Figure 6.3: Process States for Real-Time Systems

than the highest priority of tasks in the ready state, then this task returns to the running state.

2. [**Priority Inheritance**] A task claims a semaphore, but is blocked or paused. If a preempted task inherits the higher priority and this inherited priority is higher than the highest priority task in the ready state, then the task returns to the running state.

3. [**Exit**] A running task exits if the priority of a preempted task is higher than the highest ready task. Then it returns to the running state.

The six process states are summarised in in Fig 6.3 on this page.

These process states have new features which relate to processor assignment for multiprocessor environments.

The new process state transition is illustrated in Fig 6.4 on the next page.

## 6.5 A Design for Dependency Semaphores

In order to design dependency semaphores, we need to list the data structures and operations. The number of semaphores, tasks, processors, priorities and nests of semaphores are very limited, we should therefore be able to make efficient algorithms for a complex

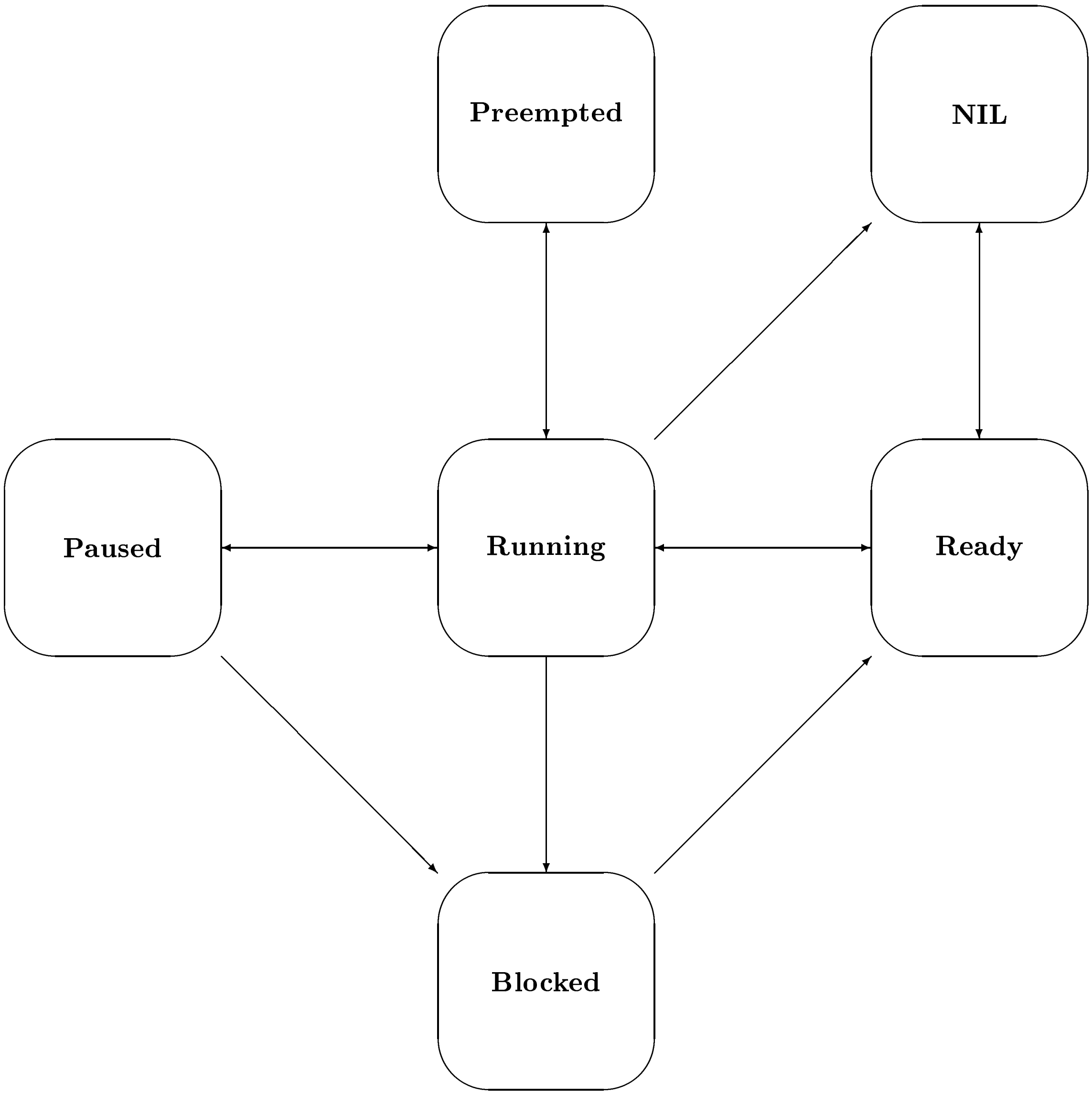


Figure 6.4: Process State Transition

dependency protocol for multiprocessors by adopting tricky data structures.

$SemaphoreID, TaskID, ProcessorID, PriorityID$ may be expressed at least by 8 bits.

Each task has its own information about semaphore dependency. For dependency protocols for uniprocessors, the general dependency lists and the dependeny lists are static and predefined, however adopting dynamic scheduling the general and ordinal dependency lists of the system are determined dynamically at runtime whenever tasks arrive and complete at the system. The structure and functions of the general and ordinal dependency lists were explained in detail in Chapter 2 and Chapter 3 and this chapter. $\mathsf{GD}[S_p](S_d)$ can be searched in the list. $S_p \in S_P(S_i)$ and $S_d \in S_D(S_i)$ are for searching for particular elements of the dependency list. $S_d \in S_D(S_i)$ is an easy operation, which is the same as $\mathsf{GD}[S_i]$. When a task locks $S_p$, then its dependency list works, which means that the *active_bits* of row element of $S_p$ are on. $\mathsf{GDL}[S_p]$ denotes $\mathsf{GD}[S_p] \wedge locked(S_p)$. $S_p \in S_P(S_i)$ is not a simple operation, but a tricky algorithm may be possible.

**type** *General_Dependency_Lists* = **array** [MaxSemaphoreID] [MaxSemaphoreID] **of**
  **record**
    *priority* : $PriorityID$;
    *active_bit* : *boolean*;
  **end** *General_Dependency_Lists*;
*activate_dependency*(*semaphore*);

The structure of tasks has functions which support:
(1) the new process state transitions for real-time systems, which include preemption and pausing.
*claimed_semaphore* : $SemaphoreID$ is for determining which semaphore the task is paused on.
(2) priority inheritance,
(3) dynamic priority assignment,
(4) dynamic processor allocation and assignment,
(5) real-time task structure; ($A_i$, $S_i$, $C_i$, $D_i$) for a *sporadic task* and ($A_i$, $C_i$, $D_i$, $I_i$) for a *periodic* task.

**type** *task* = **record**
  *taskid* : $TaskID$;
  *process_state* : **enum** *pstate* {NIL, Ready, Running, Blocked, Paused, Preempted };
  *process_flag* : **enum** *pflag* {*Inheriting, locking, nil*};
  *priority* : *Integer*; {Priority Inheritance}
  *original_priority* : *Integer*; {Priority Inheritance}
  *processor* : $ProcessorID$;
  *claimed_semaphore* : $SemaphoreID$;
  *arrival_time* : *Time*;
  *start_time* : *Time*;
  *computing_time* : *Time*;
  *deadline_time* : *Time*;

$interval\_time : Time;$
**end** $task;$
$relinquish\_processor(task);$
$processor\_assign(processor, task);$
$preempt\_task(task);$
$run(task);$

The structure of dependency semaphores has functions which support: (1) locking information, (2) a priority ordered queue which can implement both the blocked state and the paused state.

**type** $dependency\_semaphore =$
**record**
$semaphoreid : SemaphoreID;$
$semaphore\_count : Integer;$
$paused\_task\_count : Integer;$
$ceiling\_value : Integer;$ { For the *upward plug-in*}
$floor\_value : Integer;$ { For the *upward plug-in*}
$flag :$ **enum** $sflag \quad \{locked, nil\};$
$semaphore\_queue : Priority\_Descendent\_Order\_Queue;$
$locking\_task : TaskID;$
$locking\_queue : Priority\_Descendent\_Order\_Queue;$ {For future}
**end** $dependency\_semaphore;$
$enqueue\_priority\_descendent\_order(semaphore.semaphore\_queue, task);$
$dequeue(semaphore.semaphore\_queue, priority);$
$boolean\ locked(semaphore);$
$task \quad holder(semaphore);$

The structure of semaphore nest should be tricky designed for simply dealing with $S_p \in S_P(S_i)$ and $S_d \in S_D(S_i)$.

**type** $semaphore\_nest =$
**array** $[deepest\_nest]$ **of**
**record**
$semaphoreid : SemaphoreID;$
$state : Semaphore\_States;$
**end** semaphore_nest
$boolean \quad nest\_in(task, semaphore);$
$boolean \quad check\_parent\_semaphrore(parent\_semaphore, dependent\_semaphore);$
$boolean \quad check\_dependent\_semaphore(dependent\_semaphore, parent\_semaphore);$

The $Locked\_Semaphore$ list is designed for rapid calculation in dependency semaphores to search for a semaphore which the highest task holds according to the dependency. The part of the P primitive of dependency semaphores is improved.
$S_{d_m} \leftarrow \{S_d|\ max_{S_d \in S_D(S_i) \wedge locked(S_d)} \mathsf{D}[S_i](S_d)\}$
$S_{p_m} \leftarrow \{S_p|\ max_{S_p \in S_P(S_i) \wedge locked(S_p)} \mathsf{D}[S_p](S_i)\}$

**type** *Locked_Semaphores_List* =
**array** [MaxSemaphoreID] **of**
  **record**
    *semaphore* : *SemaphoreID*;
    *task* : *TaskID*;
    *priority* : *PriorityID*;
  **end** Locked_Semaphores_List
*lock_semaphore*(*semaphore*, *task*);
*unlock_semaphore*(*semaphore*, *task*);

## 6.6 Implementation Considerations

It is important to consider how to implement the proposed protocols in multiprocessor environments. Machine instructions such as *Test and Set* or *Compare and Swap* may not support priority-based queueing.

The DEC Alpha[**Sites92**] processor has two instructions, load-locked (LDL) and store-conditional (STC). A brief description from the architecture reference manual is as follows: (1) Each processor in a multiprocessor system contains a locked-physical-address register (LPAR) and a lock flag; (2) When a processor executes LDL, the physical address of the data being loaded is stored into the LPAR and the lock flag is set; (3) If the lock flag is still set when a STC is executed, the store occurs, otherwise it does not occur; (4) If processor A's lock flag is set and processor B successfully stores into the locked range of physical addresses, then A's lock flag is cleared.

Rajkumar suggested that a task should spin-read its own cache entry [**Dubois88**] while it is enqueueing to avoid excessive traffic on the memory bus [**Rajkumar90**]. Spin-locks with dynamic backoff [**Anderson90**] are more useful.

Guangxing Li and Jean Bacon [**Li93**] implemented the ceiling protocol in the WANDA Micro-kernel on the Firefly shared memory multiprocessor system [**Thacker87**]. Because a task must be non-preemptive while it executes the primitives, they proposed a *preemption variable* per task similar to the rescheduling variables. They proposed the **ceiling blocked**, similar to our paused state, and call the free mutex a **ceiling waiting** mutex. Each locking mutex maintains *a list of* **cw** mutexes. A mutex can be in one of three states: **Free**, **Locked** or **Ceiling Waiting**.

Interprocessor interrupts are reserved for implementing interprocessor preemption. While a task is executing a proposed semaphore primitive, it is non-preemptive. There are two approaches: (1) Disable interprocessor interrupt in semaphore primitives; (2) Like preemption variables, the preemptions caused by interprocessor interrupt are recorded and at the end of processing primitives interprocessor preemption is caused; This approach is employed by UNIX SVR4 ES/MP[**Kou93**].

# Chapter 7

# Conclusion

## 7.1 Related Work

The mechanism of the ceiling protocol was extended for multiprocessors [**Rajkumar88**, **Rajkumar90**, **Rajkumar91**, **Rajkumar93**]. The ceiling protocol for multiprocessors is defined as follows.

1. It exploits multiple task allocation with a static binding policy. Each task is assigned to a dedicated processor called the *host processor* and never migrates to other processors. Multiple tasks are allocated to each processor.

2. Semaphores are classified as *local semaphores* and *global semaphores.* Local semaphores are accessed by tasks allocated to a single processor and the critical section guarded by a local semaphore is referred to as the *local critical section* (lcs). Global semaphores are accessed by tasks allocated to different processors and the critical section guarded by a global semaphore is referred to as the *global critical section*(gcs).

3. The highest priority assigned to any task in the entire system is denoted as $\mathrm{P}_H$. Each semaphore has its own ceiling value as the ceiling protocol for uniprocessor systems. The priority ceiling of a local semaphore $S$ is defined to be the priority of the highest priority task that may lock the semaphore $S$ within the processor. Let the priority of the highest task that accesses a semaphore $S_i$ be $PS_i$. A global semaphore $S_G$ has its ceiling priority higher than $\mathrm{P}_H$. The ceiling priority of the global semaphore $S_{G_i}$ is
$\mathrm{P}_H + 1 + PS_{G_i}$.

4. A task uses its assigned priority unless it is within its critical section. The uniprocessor priority ceiling protocol is used for all requests to local semaphores. A task $T$ within a global critical section guarded by the global semaphore $S_G$ has the priority assigned to its gcs. When a task $T$ requests a global semaphore $S_G$, $S_G$ can be granted to $T$ by means of an atomic transaction on shared memory, if $S_G$ is not currently held by another task. If not granted, the task $T$ is added to a prioritised queue on $S_G$ without preemption. When a task $T$ attempts to release a global semaphore

$S_G$, the highest priority task $T_H$ waiting for $S_G$ is signalled and becomes eligible for execution at the task $T_H$'s host processor at its gcs priority.

We will point out several drawbacks of the ceiling protocol for multiprocessors which is described above.

First, it exploits multiple task allocation with static binding, therefore, it cannot exploit an interprocessor preemption mechanism. Moreover, as described in Section 5.1 on page 82, in an environment of multiple task allocation with a static binding policy the priority inheritance multiple processor cannot work well; this is serious a drawback compared with the single allocation with dynamic binding policy.

Secondly the mechanism of local semaphores and global semaphores is too conservative. Let us consider a example where local semaphores and global semaphores are accessed.

—*Example of Local Semaphores and Global Semaphores*—
$\boxed{\wp_0}$ $T_0$ = {.....P($S_0$),....,V($S_0$),...,P($S_G$),...,V($S_G$),...}
$\boxed{\wp_1}$ $T_1$ = {.......,P($S_1$),........,V($S_1$),.........................}
$\boxed{\wp_1}$ $T_2$ = {......,P($S_1$),............,V($S_1$),.......................}
$\boxed{\wp_1}$ $T_3$ = {..,P($S_1$),.......V($S_1$),.,P($S_G$),....,V($S_G$),....}
$\boxed{\wp_0}$ $T_4$ = {...,P($S_0$),......,V($S_0$),...............................}
$(P_0 > P_1 > P_2 > P_3 > P_4)$

Then if we allocate task $T_5$ to $\wp_0$ as below:
$\boxed{\wp_0}$ $T_5$ = {...,P($S_0$),..,P($S_1$),.......V($S_1$),V($S_0$),.............}
The semaphore $S_1$ must be the global semaphore. Whenever a new task arrives and is allocated to a processor, the classification of local and global semaphores and their ceiling values must be changed dynamically.

Thirdly, the ceiling protocol for multiprocessors excludes semaphore nests. The semaphore nests are major sources of semaphore blocking even in case of multiprocessors, therefore, instead of excluding these from our investigation, we need to consider how semaphore blocking from semaphore nests can be avoided.

The rate monotonic algorithm is a static binding real-time scheduling, which provides facilities for limited range of requirements for old fashioned real-time systems. Now workstations need to provide diverse services for increasing variety of requirements, which include audio and video, network and user interface processing. They expect more flexible and dynamic real-time scheduling algorithms. Therefore, the development of dynamic real-time synchronisation protocols became inevitable.

Guangxing Li and Jean Bacon implemented the ceiling protocol in WANDA microkernel on the Firefly shared memory multiprocessor system [**Li93**]. Their implementation details suggest the implementation of our protocols. They advocated (1) preemption variables like rescheduling variables, (2) busy-waiting mutual exclusion, (3) direct blocking and ceiling blocking, (4) a list of ceiling waiting mutexes, and (5) three states of a mutex, **Free**, **Locked** and **Waiting**.

Large amounts of real-time research were carried out by the ADA community, however we do not know any multiprocessor real-time synchronisation protocol done in the context of Ada [**Sha93**].

## 7.2 Conclusion

We developed real-time synchronisation protocols for both uniprocessors and multiprocessors based on the principle that *"a lower task is forced to avoid locking a semaphore which would block higher tasks"*. For uniprocessors, it was pointed out that the ceiling protocol is too conservative. The dependency protocol was therefore proposed.

For multiprocessors, our goal was to develop real-time synchronisation protocols which assume dynamic real-time scheduling. As pointed out in the previous section, the ceiling protocols for multiprocessors, which assume static binding allocation algorithms, is also conservative and *pessimistic*. In multiprocessor environments, we need to consider a processor allocation policy for real-time scheduling algorithms, and moreover priority inheritance and preemption had to be redefined to include a processor assignment policy for real-time synchronisation protocols. We assumed the ideal situation, then the plug-in method was developed. The plug-in method does not require a special mechanism compared to the ordinary semaphore, therefore, we can apply the technique with ordinary operating systems. By investigating semaphore blocking caused by semaphore nests, colour semaphores were developed, which have descriptive power to solve complex real-time synchronisation problems. Priority ordered queues were provided to ensure essential real-time properties.

We redefined preemption and priority inheritance in multiprocessor environments. They applied to preemptive semaphores, which provide rapid preemption connected with issuing the V primitive. This application showed that the redefinition of priority inheritance and preemption for multiprocessors is effective and useful. Finally, the dependency protocol for multiprocessors was proposed, integrating the dependency protocol for uniprocessors, the *downward plug-in* and *upward plug-in* methods, and priority inheritance with preemption for multiprocessors. The dependency protocol for multiprocessors did not assume static binding real-time algorithms.

## 7.3 Future Work

A task locks a semaphore and waits: (1) an internal events occurs; (2) an external event occurs. They are short lived or long lived. The long lived case is much more important than the first for real-time synchronisation. Processors are becoming very fast, external devices are becoming relatively slow. Fast processors can calculate more complex scheduling algorithms than old fashioned algorithms, for example, the rate monotonic algorithm and the earliest deadline first algorithm. We need not be worried about employing runtime dynamic scheduling algorithms because of fast processors.

However the analysis of timing behaviour of the system using our synchronisation protocols and their schedulability are an open problem.

Let us consider the integration of real-time scheduling algorithms and real-time synchronisation protocols for multiprocessors.

Real-time scheduling is invoked by the arrival and completion of tasks to the system. Real-time scheduling algorithms for multiprocessor provide facilities to assign a priority and allocate a processor to a task. The arrival of a higher task causes preemption in the allocated processor. The completion of a task causes task reassignment to the allocated processor.

Real-time synchronisation manages the process state transitions to and from the ready state, the blocked state and the paused state. P and V primitives causes rearrangement of the processor assignment and the priority order in the allocated processor.

Preemption is invoked by the arrival of tasks and the issue of a V primitive in a processor. However preemption caused by the V primitive is sometimes ignored. Preemptive semaphores provide facilities to allow preemption in the V primitive. The completion of tasks and the issue of P primitives manage reassignment of the processor to a task allocated to the processor. The degrees of preemption are classified according to restricted level as follows:
(1) no preemption anywhere,
(2) restricted preemptive like preemption points,
(3) no preemptive in critical sections,
(4) restricted preemptive in critical sections, for example, the stack resource policy [**Baker90**],
(5) preemptive everywhere.
We adopted *preemptive everywhere* where priority inversion may occur the most times among the above cases. Therefore we developed powerful real-time synchronisation protocols. However according to the integration of the static binding real-time scheduling algorithms and real-time synchronisation protocols, each task allocated to a processor at a time has both a partial ordered priority with each processor for preemption caused by task arrival and total ordered priority for preemption caused by real-time synchronisation. This make difficult how to achieve the schedulability for all tasks in the real-system, not for each local processor.

We might assume that real-time scheduling is invoked by both (1) the arrival and completion of tasks and (2) P and V primitives of real-time synchronisation (one example is the algorithm of preemptive semaphores). In this case we might adopt total ordered priority, but assignment of processor will become complex and need heavy computation. However we expect that such an approach will be feasible to achieve the schedulability of all tasks in the real-time system. For the future work, we named our scheduling algorithm as the earliest deadline first with assignment to the busiest processor, and discussed preemptive semaphores and dependency semaphores in general form.

Operating systems for multiprocessors such as Mach, Topaz and V kernel provide the thread mechanism as the computation unit rather than the concept of a process. Compared with a process that has its own address space, multiple threads can share an address space. A set of threads constitute an application where threads interact with one another. Thus the context switching of threads is light and threads are highly preemptive for frequent processor reassignment. Therefore threads require dynamic assignment processor allocation [**Anderson92**]. Threads for real-time systems are designed [**Schwan91**, **Tokuda91b**]. Therefore real-time synchronisation protocols for multiprocessors which assume dynamic assignment processor allocation are desirable.

The concept of the *working unit* did not include I/O delays. I/O specific tasks which are designed by monitors [**Hoare74**] should have an I/O delay in a critical section. We show typical example of priority inversion again:

—*Example of priority inversion*—
$T_0$ = {....,P($S_0$),.......,V($S_0$),.....}
$T_1$ = {.....................................}
$T_2$ = {...,P($S_0$),.........,V($S_0$),....}
($P_0 > P_1 > P_2$)

1. $T_2$ is initiated and locks $S_0$.
2. $T_1$ is initiated and preempts $T_2$.
3. $T_0$ is initiated and preempts $T_1$. It then claims $S_0$ and becomes blocked.
4. $T_1$ restarts.

There would be the following question if DO-WORK includes an I/O delay:

Processors become faster, $T_1$ completes earlier than $T_2$ with an I/O delay. $T_0$ delays for $T_2$'s I/O delay, which is not real priority inversion.

However for database systems, processor power is more important than discs and networking performance. (1) processing complex data structure and streams and (2) processing device drivers requires processor power. Therefore $T_1$ with longer execution time than the duration of I/O delay will still be very common. In some cases, $T_1$ executes for less time than the I/O delay. Even though processors become faster, some cases of priority inversion will disappear, but most cases of priority inversion remain. The principle of our work can apply when the concept of the *working unit* includes I/O delays.

Now operating systems provide increasingly diverse services which require real-time properties, and along with workstations, server systems of various kinds – file servers, database servers, print servers, network interface servers and computing servers will also be developed using multiprocessor systems which require new real-time properties. Therefore, a quantitative analysis of operating systems is becoming important [**Wilkes93**], as a quantitative analysis of computer architecture is becoming available [**Patterson90**].

Our challenge is to provide a predictable real-time computer environment that is easily analysed.

# Appendix A

# Specification for Real-Time Synchronisation

## A.1 Objective

Real-time scheduling algorithms for multiprocessors are extremely complex. Therefore to understand precisely their problems and solutions is very important. Specification techniques are useful to understand research targets precisely for complex algorithms. For example, mutual exclusion can be specified [**Lamport87**].

The benefits of formal methods are that they address the manner of:

1. generality,
2. redundancy.

A program itself is a specification, but it is algorithmic. The correctness of a program is different from that of a specification.

The synchronisation primitives in the *Topaz* operating system for *Firefly* multiprocessors were formally specified by an algebraic specification language *Larch* [**Guttag90**] in the impressive paper [**Birrell87**] which suggested that the formal specifications of the synchronisation primitives are relatively stable compared with frequent changes of their implementation.

The synchronisation primitives in the *Plan9* operating system [**Pike91**] for Silicon Graphics multiprocessor systems has also been specified with a computer protocol specification language *Spin* [**Holzmann91**]. An error in the original implementation was found and a new algorithm validated [**Holzmann93**].

The synchronisation primitives for real-time scheduling for multiprocessors are more complex than these, which must consider priority and semaphore blocking time of higher tasks held by lower tasks as described in the previous chapter.

### A.1.1 Examples of Specification for Real-Time Scheduling

Good specifications of both *priority inheritance* and the *ceiling protocol* using the specification language Z [**Potter91**] were written in [**Pilling90**].

## A.2 Requirement

It would be interesting to learn of any formalism which would be useful for description and reasoning in the following situation.

We have a number of tasks which are characterised by priorities. They are to be scheduled and executed on a computer which may have a number of processors. Each task when executed uses only one processor and runs for one working unit. In general, before a task may start one or more semaphores must be successfully acquired. The fact that a high priority task needs to be done does not mean that it can be done instantly, since the semaphores it needs may already be held by other tasks, specifically lower priority tasks.

As described in previous chapters, several algorithms have been proposed, the objective of which is to ensure that high priority tasks are not delayed for more than one working unit by lower priority tasks. (One can only do better than that by total resource dedication.) The algorithms themselves may presumably be specified using Larch or any other good specification system. Our question is whether Larch or other formal methods can specify the objective with a view to demonstrating that it can be achieved, or does one have to engage in informal reasoning outside the system to do that?

How hard a formal proof would be depends on how complex our optimum requirement is and how fair the algorithm is. If our goal is to make sure of the correctness of the algorithm, rather than wanting a mechanical proof for its own sake, it is recommended that we write a structured hand proof. A formal language is good for expressing something we already understand or something we think we understand and want to write down precisely to check our understanding.

### A.2.1 Simple Survey of TLA$^+$

The paper [**Abadi92**] specifies real-time properties using the specification language called TLA$^+$, temporal logic of actions, which Leslie Lamport invented [**Lamport93**]. TLA$^+$ concentrates on the temporal aspect of specifications, which is not really necessary for the requirements. There is nothing about TLA$^+$ that makes it much better than any other formalism for expressing our requirements. The specification uses only constants, so there is no TLA in it; it is just ordinary mathematics. The TLA$^+$ specification is probably a little nicer than one in a formalism that is based on types instead of sets.

A simple survey of TLA$^+$ is described below by [**Lamport93**] and [**Kurki-Suonio93**].

TLA$^+$ models represent histories as behaviours $\sigma$, defined as infinite sequences of states, $\sigma =< s_0, s_1, ..... >$, where each state $s_i$ is a collection of values of *program variable.*

A pair of states $(s_i, s_{i+1})$ is a *step.*

A predicate $P$ is satisfied by a behaviour $\sigma$, if $P$ gets the value true when its program variables are replaced by their values in the initial state $s_0$ of $\sigma$.

Two operators, $'$ (prime) and $\Box$ (*always*), are introduced. An *action* is a Boolean valued formula containing unprimed variable, primed variables, and constants. An action is interpreted as an assertion about a step, where the unprimed variable refer to their values in the first state of the step, and the primed variables refer to their value of the second state.

If $\mathcal{A}$ is an action, a step (pair of states) is called an $\mathcal{A}$ *step* iff $\mathcal{A}$ is true for the step. An action $\mathcal{A}$ is considered to be true or false of a behaviour iff the first step of the behaviour is an $\mathcal{A}$ step. An action thus makes an assertion about the first step (the first pair of states) of a behaviour. An action $\mathcal{A}$ is true of a behaviour $s_1, s_2, s_3, \ldots$ iff $s_1, s_2$ is an $\mathcal{A}$ step. The formula $\Box\mathcal{A}$ is true of this behaviour iff $\mathcal{A}$ is true of the behaviour $s_{n+1}, s_{n+2}, s_{n+3}, \ldots,$ for all $n$, which is the case iff $s_{n+1}, s_{n+2}$ is an $\mathcal{A}$ step, for all $n$. Hence, $\Box\mathcal{A}$ asserts that every step in the behaviour is an $\mathcal{A}$ step.

In TLA$^+$, `UNCHANGED` $f$ denotes $f' = f$. Let us denote $[\mathcal{A}]_v$ as $\mathcal{A} \vee (\texttt{UNCHANGED}\ v)$. The safety properties of a program can

$$\texttt{Safe} \;\triangleq\; \texttt{Init} \;\wedge\; \Box\,[\texttt{Step}]_{\texttt{v}}$$

Intuitively, a fairness condition for action $\mathcal{A}$ asserts that an $\mathcal{A}$ step eventually occurs if it is often enough possible. Let `Enabled` $\mathcal{A}$ be the predicate asserting that an $\mathcal{A}$ step is possible. More precisely, `Enabled` $\mathcal{A}$ is true for a state $s$ iff there exists some state $t$ such that $s, t$ is an $\mathcal{A}$ step. Fairness for $\mathcal{A}$ means that an $\mathcal{A}$ step eventually occurs if `Enabled` $\mathcal{A}$ is true often enough. One interpretation of *often enough* is *always*, so the condition is $\Box(\texttt{Enabled}\,\mathcal{A}) \;\Rightarrow\; \Diamond\mathcal{A}$, which is logically equivalent to $\Diamond(\neg\texttt{Enabled}\,\mathcal{A}) \;\vee\; \Diamond\mathcal{A}$. Fairness of $\mathcal{A}$ in a behavior $s_1, s_2, s_3, \ldots$ should mean that this formula is true for all suffixes $s_{n+1}, s_{n+2}, s_{n+3}, \ldots$ of the behavior. In other words, fairness means $\Box(\Diamond(\neg\texttt{Enabled}\,\mathcal{A}) \;\vee\; \Diamond\mathcal{A})$. It follows from the definitions of $\Box$ and $\Diamond$ that $\Box(\Diamond F \vee \Diamond G)$ is equivalent to $(\Box\Diamond F) \vee (\Box\Diamond G)$. Hence, a fairness condition can be defined, called *weak fairness of* $\mathcal{A}$, by the formula

$$\texttt{WF}(\mathcal{A}) = \Box\Diamond(\neg\texttt{Enabled}\,\mathcal{A}) \;\vee\; \Box\Diamond\mathcal{A}$$

The formula $\Box\Diamond F$ is true for the behaviour $s_1, s_2, s_3, \ldots$ iff there are infinitely many values of $n$ such that $F$ is true for $s_{n+1}, s_{n+2}, s_{n+3}, \ldots$. Weak fairness of $\mathcal{A}$ therefore asserts that $\mathcal{A}$ is infinitely often disabled or infinitely many $\mathcal{A}$ actions occur.

let $\langle\mathcal{A}\rangle_v$ denote $\mathcal{A} \wedge (v' \neq v)$ and

$$\texttt{WF}_v(\mathcal{A}) \triangleq \Box\Diamond\neg\texttt{Enabled}\,\langle\mathcal{A}\rangle_v \;\vee\; \Box\Diamond\langle\mathcal{A}\rangle_v$$

### A.2.2 A TLA$^+$ Specification for the Requirement

Figure A.1 on the next page describes preparation for the algorithm for the blocking time requirement in the previous section.

In the declaration part, `Time` is defined as variable. Functions can be described explicitly in TLA$^+$ with the construct

$[\ x\ \in\ S \mapsto e\ ]$

which denotes the function $f$ with domain $S$ such that $f[x] = e$. Function `f[T]` represents that task `T` is assigned to some processor or is not assigned. Function `Owns[s,t]` represents that at time $t$ the semaphore is locked by a task.

Figure A.2 on page 124 describes the blocking time requirement.

```
─────────────── MODULE RealSynchronisationParams ───────────────
IMPORT  Naturals, FiniteSets
I am assuming a module FiniteSets that defines the operators IsFiniteSet and Cardinality.
────────────────────────────────────────────────────────────────
PARAMETERS
Time      : VARIABLE    The "type" (set) of all possible time
Priority  : CONSTANT    The "type" (set) of all possible priorities
Semaphore : CONSTANT    The "type" (set) of all possible semaphores
ProcessId : CONSTANT    The type (set) of all processors
TaskId    : CONSTANT    The "type" (set) of all task identifiers
NumberOfProcessors : Nat CONSTANT    The number of processors
f ∈ [TaskId ↦ ProcessorId ∪ {null}]
Owns ∈ [ Semaphore × time ↦ TaskId ∪ Free]
────────────────────────────────────────────────────────────────
ASSUMPTION
Finiteness  ≜  IsFiniteSet(TaskId) ∧ IsFiniteSet(Semaphore)
We assume that TaskId and Semaphore are finite sets.

CONSTANTS

The following defines Task to be a set of records with the indicated fields.

 Task ≜  [[ id         : TaskID              The task's id
            priority   : Priority,           The task's priority
            semaphores : LIST of Semaphore   Its required semaphores
            starttime  : Subset Time         Start time
         ]]

 Free ≜ {free}
────────────────────────────────────────────────────────────────
```

Figure A.1: A Specification of Block-Time Requirement, the Declaration Part

`Init` represents the initial state at the system start. All task states are **`null`** which means that they are not allocated to processors. `Time` is 0.

`idle(t)` indicates that task has not been initiated.

`Waiting(t)` indicates that at time $t$ some tasks have claimed semaphores. For simplicity, each task can claim semaphore list once.

`Active(t)` indicates that at time $t$ some tasks are locking semaphores.

`Highest(T, t)` indicates that at time $t$ task $T$ is blocked and has the highest priority in the blocked state.

`Step` indicates that the task which is allocated to a processor and is highest in the blocked state will lock semaphores after WU time.

This algorithm explains that the highest priority task which is blocked waits for no more than one *working unit*.

```
──────────────── MODULE RealSynchronisationAlgorithm ────────────────
IMPORT  RealSynchronisationParams
─────────────────────────────────────────────────────────────────────
PREDICATES
 Init ≜   ∧ ∀ S ∈ Task: f[S] = null
          ∧ Time = 0

 idle(t) ≜ { u ∈ Task: t <  u.starttime }

 Waiting(t) ≜ { u ∈ Task: u.starttime ≤ t
                    ∧ ¬ (∀ s ∈ u.semaphores: Owns(s,t)= u.id}

 Active(t) ≜ { u ∈ Task: u.starttime ≤ t
                    ∧  ∀ s ∈ u.semaphores: Owns(s,t)= u.id}

 Highest(T,t) ≜ ( ∀ u ∈ Waiting(t): u.priority ≤ T.priority)
                    ∧ T ∈ Waiting(t)

ACTIONS
 Step ≜   ∃ S ∈ Task:
                  ∧ f[S] ∈ ProcessId
                  ∧ #(f[S]) ≤ NumberOfProcessors
                  ∧ {∀ T ∈ S:
                      ¬  ∃  t′ : Highest(T, t′) ∧ T ∉ Active(t′)}
                  ∧ Time′ = Time + WU

 Algorithm ≜  ∧ Init
              ∧ □[Step]_⟨Task, Time⟩
              ∧ WF_⟨Task, Time⟩(Step)
─────────────────────────────────────────────────────────────────────
```

Figure A.2: A Specification of Block-Time Requirement, the Algorithm Part

# Appendix B

# The General Dependency Protocol

As described in Chapter 2, Section 2.7.2, the example below shows a limitation of the dependency protocol.

—*Example 1*—
$T_0$ = {.......,P($S_1$)P($S_2$)P($S_3$),....,V($S_3$)V($S_2$)V($S_1$)..}
$T_1$ = {.....,P($S_3$),....................,V($S_3$),.......................}
$T_2$ = {....,P($S_2$),....................,V($S_2$),........................}
$T_3$ = {..,P($S_1$),....................,V($S_1$),..........................}
($P_0 > P_1 > P_2 > P_3$)

The highest task $T_0$ should wait for three working units, even though the dependency lists are activated. The semaphore dependency protocol has no advantages in the above example.

We will try to provide a solution for the above example. The purpose of *reliance* is to use the dependency the other way round.

| *The Definition of Reliance* |
|---|
| $S_k$ relies on $S_i$ at priority $P$ if there is a task with priority $P$ which will seek to lock $S_i$ while holding $S_k$. $S_i$ is being relied on in ( $S_k$, $P$ ).<br>The list of reliance of $S_i$ is a set of pairs of a parent semaphore and its priority.<br>The element of the list of reliance is active while the semaphore is locking, and disappears after released. |

In the above example, $S_3$ is being relied on in [($S_1$, $P_0$), ($S_2$, $P_0$) ].
$S_2$ is being relied on in [ ($S_1$, $P_0$) ].

A variant scheduling rule based on the definition of reliance is:

| *The Scheduling Rule of a Variant of the Dependency Protocol* |
|---|
| A task $T$ running at priority $P$ is not allowed to lock a semaphore $S_i$ if $S_i$ is relied on at a priority greater than or equal to $P$ by a semaphore $S_j$ that is already locked , unless the reliance comes from the task $T$ itself. A task which is unable to lock a semaphore is blocked immediately before the lock call, and released when the condition on which it was blocked disappears. The task locking the dependent semaphore inherits the priority $P$ from the blocked task. The ordinary priority inheritance scheme is also applied. |

If this rule is applied to the example-1:

1. $T_3$ is initiated, and locks $S_1$.
2. $T_2$ is initiated, and claims $S_2$, but by the list of reliance, $T_2$ must stop before $S_2$. $T_3$ inherits the priority.
3. $T_1$ is initiated, and claims $S_3$, but by the list of reliance, $T_1$ must stop before $S_3$. $T_3$ inherits the priority.
4. $T_0$ is initiated, and is blocked on $S_1$. $T_3$ inherits the priority of $T_0$.
5. $T_3$ releases $S_1$, then $T_0$ restarts.

The highest priority task $T_0$ waits for only one working unit compared with three working units in the ordinary dependency protocol. However, if the order of the semaphores in the nest is reversed in the following:

—*Example 2*—
$T_0$ = {.......,P($S_3$)P($S_2$)P($S_1$),....,V($S_3$)V($S_2$)V($S_1$)..}
$T_1$ = {.....,P($S_3$),....................,V($S_3$),.......................}
$T_2$ = {....,P($S_2$),....................,V($S_2$),........................}
$T_3$ = {..,P($S_1$),....................,V($S_1$),..........................}
$(P_0 > P_1 > P_2 > P_3)$

In Example 2, by the ordinary dependency protocol, $T_0$ waits for only one working unit, but by the variant dependency protocol, $T_0$ waits for three working units.

The previous discussion shows that the rule should ignore the order of semaphores in the nests. We considered one way dependency for the dependency protocol and a variant dependency protocol. We will try to make a rule for the general dependency protocol using two ways dependency as follows:

| *The Definition of Primary Dependency* |
|---|
| $S_i$ is *primary dependent* on $S_j$ at priority $P$ if there is a task at priority $P$ which needs both $S_i$ and $S_j$ in a nest to do its job. |

*Primary dependency* does not concern the order of elements in the relation. Transitivity of primary dependency is defined as follows:

| *The Transitivity of Primary Dependency* |
|---|
| **If** $S_i$ is primary dependent on $S_j$ at $P$ and<br>$S_j$ is primary dependent on $S_k$ at $Q$, ( $Q > P$),<br>**then** $S_i$ is primary dependent on $S_k$ at $Q$. |

| *The General Primary Dependency Set* |
|---|
| The general primary dependency set is the transitive closure under the rule of the primary dependency. |

| *The Scheduling Rule of the General Dependency Protocol* |
|---|
| A task $T_i$ running at priority $P_i$ needs a set $\sigma_i^j$, nested semaphores, before it can work.<br>A task $T$ is not allowed to lock a semaphore $S_k \in \sigma_i^j$ if any semaphore has already locked has $S_k$ at a priority greater than or equal to $P_i$ in its general dependency set not activated by $T$ itself.<br>A task which is unable to lock a semaphore is blocked immediately before the lock call, and released when the condition on which it was blocked disappears.<br>The task locking the related semaphore inherits the priority $P$ from the blocked task.<br>The ordinary priority inheritance scheme is also applied. |

—*Example, The General Dependency Protocol*—
$T_0$ = {..........,P($S_3$),.....,P($S_0$),.............,V($S_0$),..,V($S_3$),............}
$T_1$ = {.......,P($S_1$),....,P($S_2$),................,V($S_2$),.,V($S_1$),...............}
$T_2$ = {....,P($S_2$),....,P($S_3$),................,V($S_3$),.,V($S_2$),...................}
$T_3$ = {..,P($S_0$),...,P($S_1$),.................,V($S_1$),.,V($S_0$),....................}
$(P_0 > P_1 > P_2 > P_3)$

The primary dependency set DS is:
$[(S_0, S_1)@P_3, (S_2, S_3)@P_2, (S_1, S_2)@P_1, (S_3, S_0)@P_0]$.
By transitions, the general primary dependency set GDS is:
DS$\cup$ $[(S_1, S_3)@P_0, (S_2, S_0)@P_0, (S_1, S_0)@P_0]$.

1. $T_3$ is initiated, and locks $S_0$.
2. $T_2$ is initiated, and claims $S_2$, but by GDS, $T_2$ must stop before $S_2$. $T_3$ inherits the priority.
3. $T_1$ is initiated, and claims $S_1$, but by GDS, $T_1$ must stop before $S_1$. $T_3$ inherits the priority.
4. $T_0$ is initiated, and claims $S_3$. but by GDS, $T_0$ must stop before $S_3$. $T_3$ inherits the priority.
5. $T_3$ restarts, locks $S_1$, releases $S_1$ and $S_0$, then $T_0$ restarts.

This nested type example will work as the ceiling protocol. The general dependency protocol concentrates on the nested semaphores, not all semaphores which the ceiling protocol should concern with. Therefore the general dependency protocol does not cause unnecessary blocking in the circumstance which the ceiling protocol does. This is that the general dependency protocol is not conservative.

# Bibliography


**[Abadi92]** Martín Abadi and Leslie Lamport. *An Old-Fashioned Recipe for Real Time.* Technical Report 91, DEC System Research Center, 1992. (p 121)

**[Anderson90]** Thomas E. Anderson. *The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors.* IEEE Transactions on Parallel and Distributed Systems, 1(1):6–16, 1990. (pp 4, 114)

**[Anderson92]** Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, 10(1), 1992. (p 118)

**[Anderson93]** David P. Anderson. *Metascheduling for Continuous Media.* ACM Transactions on Computer Systems, 11(3):226–252, 1993. (p 3)

**[Audsley90]** N. Audsley and A. Burns. *Real-Time System Scheduling.* Technical Report, Department of Computer Science, University of York, 1990. (p 84)

**[Bach86]** Maurice Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986. (pp 4, 109)

**[Bacon93]** Jean Bacon. *Concurrent Systems, An Integrated Approach to Operating Systems, Database and Distributed systems.* Addison-Wesley, 1993. (p 79)

**[Baker90]** T. P. Baker. *A Stack-Based Resource Allocation Policy For Real-Time Processes.* In Proc. the IEEE Real-Time Systems Symposium, pages 191–200, 1990. (p 118)

**[Birrell87]** A. D. Birrell, J. V. Guttag, J.J. Horning, and R. Levin. *Synchronization Primitives for a Multiprocessor: A Formal Specification.* Technical Report, DEC System Research Center, August 1987. (p 120)

**[Burns91]** Alan Burns. *Scheduling Hard Real-Time Systems: A Review.* Software Engineering Journal, pages 116–128, May 1991. (p 14)

**[Campbell92]** A. Campbell, H. Leopold D. Huntchinson, and N. Singer. *Distributed Multimedia Communication Requirements*, 1992. OSI 95 Project Report. (p 3)

**[Chen91]** Min-Ih Chen. *Schedulability Analysis of Resource Access Control Protocols in Real-Time Systems.* PhD dissertation, University of Illinois at Urbana-Champaign, 1991. (pp 14, 84)

**[Dertouzos74]** Michael L. Dertouzos. *Control Robotics; The Procedural Control of Physical Process.* In Proc. IFIF 74, pages 807–813. North Holland Publishing Co., 1974. (p 85)

**[Dhall78]** S. K. Dhall and C. L. Liu. *On a Real-Time Scheduling Problem.* Operations Research, 26(1):127–140, 1978. (p 84)

**[Dijkstra68]** E. W. Dijkstra. *Co-operating Sequential Processes.* In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968. (pp 58, 62)

**[Dubois88]** M. Dubois, C. Scheurich, and F. A. Briggs. *Synchronization, Coherence and Event Ordering in Multiprocessors.* IEEE Computer, 21(2), 1988. (p 114)

**[Fisher92]** Tom Fisher. *Real-Time Scheduling Support in Ultrix-4.2 for Multimedia Communication.* In Proc. Third International Workshop on Network and Operating System Support for Digital Audio and Video, pages 282–289, November 1992. (p 13)

**[Furht91]** B. Furht, D. Grostick, D. Gluch, J. Parker G. Rabbat, and M. McRoberts. *Real-Time UNIX Systems: Design and Application Guide.* Kluwer Academic Publisher, 1991. (p 5)

**[Gallmeister91]** Bill O. Gallmeister and Chris Lanier. *Early Experience With POSIX 1003.4 and POSIX 1003.4A.* In Proc. IEEE Real-Time Systems Symposium, pages 190–198, 1991. (p 13)

**[Goodman83]** J. R. Goodman. *Using cache memory to reduce processor memory traffic.* In Proc. Tenth Annual Symposium on Computer Architecture, pages 124–131, 1983. (p 4)

**[Guttag90]** John V. Guttag, James J. Horning, and Andres Modest. *Report for the Larch Shared Language: Version 2.3.* Technical Report, DEC System Research Center, 1990. Report 58. (p 120)

**[Hayter91]** Mark Hayter and Derek McAuley. *The Desk Area Network.* ACM Operating Systems Review, 25(4):14–22, 1991. (p 3)

**[Hayter93]** Mark Hayter. *A Workstation Architecture to Support Multimedia.* PhD dissertation, Computer Laboratory, University of Cambridge, 1993. (p 3)

**[Hayton93]** Richard Hayton, September 1993. Private Communication. (p 12)

**[Hennessy94]** John L. Hennessy and David A. Patterson. *Computer Organization and Design; The Hardware/Software Interface.* Morgan Kaufmann Publishers, 1994. (pp 3, 4)

**[Hoare74]** C. A. R. Hoare. *Monitors: An Operating System Structuring Concept.* Comm. of ACM, 17(10):549–557, 1974. (p 119)

**[Holt71]** R. C. Holt. *Comments on Prevention of System Deadlocks.* Comm. of ACM, 14(1):36–38, 1971. (p 22)

**[Holt72]** R. C. Holt. *Some Deadlock properties of Computer Systems.* ACM Computing Surverys, 4(3):179–196, 1972. (p 22)

**[Holzmann91]** Gerard Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991. (p 120)

**[Holzmann93]** Gerard Holzmann, February 1993. Private Communication. (p 120)

**[Hopper90]** Andrew Hopper. *Pandora - An Experimental System for Multimedia Applications.* ACM Operating Systems Review, 24(2):19–34, April 1990. (p 3)

**[Johnson74]** D. S. Johnson, A. Dermers, J. D. Ullman, M. R. Garey, and R. L. Graham. *Worst-Case Performance Bounds For Simple One-Dimensional Packing Algorithms.* SIAM Journal Computation, 3(4):299–325, 1974. (p 86)

**[Khanna92]** Sandeep Khanna, Michael Sebree, and John Zolnowsky. *Real-time Scheduling in SunOS 5.0.* In Proc. USENIX 92 Winter, pages 375–390, 1992. (pp 2, 10, 13)

**[Knuth92]** Donald E. Knuth. *Literate Programming.* Centre for the Study Language and Information, 1992. (pp 9, 58)

**[Kopetz93]** Harmann Kopetz. *Scheduling.* In Sape Mullender, editor, *Distributed Systems, Second Edition*, chapter 18, pages 491–509. ACM Press, 1993. (p 84)

**[Kou93]** Showkow Kou, 1993. Private Communication. (p 114)

**[Kurki-Suonio93]** Reino Kurki-Suonio. *Stepwise Design of Real-Time Systems.* IEEE Transactions on Software Engineering, 19(1):55–66, 1993. (p 121)

**[Lamport87]** Leslie Lamport. *A Fast Mutual Exclusion Algorithm.* ACM Transactions on Computer Systems, 5(1):1–11, February 1987. (p 120)

**[Lamport93]** Leslie Lamport. *Musical Chairs in TLA+.* Technical Report, DEC System Research Centre, 1993. Preliminary Draft. (p 121)

**[Lampson80]** B. W. Lampson and D. D. Redell. *Experiences with Processes and Monitors in Mesa.* Comm. of ACM, 23(2):105–117, 1980. (p 13)

**[Leffer89]** Samuel J. Leffer, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison Wesley, 1989. (p 109)

**[Lehoczy89]** J.P. Lehoczy, L. Sha, and Y. Ding. *The Rate Monotonic Scheduling Algorithm - Exact Characterisation and Averse Can Behaviour.* In Proc. IEEE Real-Time Systems Symposium, pages 166–171, 1989. (p 5)

**[Leslie93]** Ian M. Leslie, Derek McAuley, and Sape J. Mullender. *Pegasus - Operating System Support for Distributed Multimedia Systems.* ACM Operating Systems Review, 27(1):69–79, 1993. (p 3)

**[Li93]** Gaungxing Li and Jean Bacon. *Realtime Thread Synchronization In A Micorkernel.* In Proc. IEE International Workshop on Systems Engineering for Real-Time Applications, pages 84–89, 1993. (pp 114, 116)

**[Lim93]** Beng-Hong Lim and Ananat Agrarwal. *Waiting Algorithms for Synchronization in Large Scale Multiprocessors.* ACM Transactions on Computer Systems, 11(3):253–294, 1993. (p 4)

**[Liu73]** C. L. Liu and J.W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment.* JACM, 20(1):46–61, 1973. (pp 5, 84, 85, 86)

**[Lo93]** Sai Lai Lo, September 1993. Private Communication. (p 12)

**[Maekawa87]** Mamoru Maekawa, Arthur E. Oldehoeft, and Rodney R. Oldehoeft. *Operating Systems, Advanced Concepts.* The Benjamin/Cummings Publishing Company, Inc., 1987. (pp 59, 66)

**[McCann93]** C. McCann, R. Vaswani, and J. Zahorjan. *A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors.* ACM Transactions on Computer Systems, 11(2):146–178, May 1993. (p 85)

[**Mellor-Crummey91**] John M. Mellor-Crummey and Michael L. Scott. *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.* ACM Transactions on Computer Systems, 9(1), February 1991. (p 4)

[**Nakamura86**] Akira Nakamura. *The Structure of the UNIX Operating System.* Maruzen, 1986. In Japanese. (p 101)

[**Needham92**] Roger Needham, February 1992. Private Communication. (p 2)

[**Owicki89**] Susan Owicki and Anant Agarwal. *Evaluating the Performance of Software Cache Coherence.* Technical Report, DEC System Research Centre, 1989. Report Number 41. (p 4)

[**Patterson90**] David Patterson and John Hennessy. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, 1990. (pp 4, 119)

[**Pike91**] Rob Pike, Dave Presotto, Ken Thompson, and Gerard Holzmann. *Process Sleep and Wakeup on a Shared-Memory Multiprocessor.* In Proc. Spring EuroOpen, Troms, May 1991. (p 120)

[**Pilling90**] Michael Pilling, Alan Burns, and Kerry Raymond. *Formal Specification and Proof of Inheritance Protocols for Real-Time Scheduling.* Software Engineering Journal, pages 263–279, September 1990. (pp 14, 120)

[**POSIX.4a90**] *Thread Extension for Portable Operating Systems*, 1990. IEEE, POSIX P1004a. (p 13)

[**Potter91**] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 1991. (pp 14, 120)

[**Rajkumar88**] Ragunathan Rajkumar, Lui Sha, and John Lehoczky. *Real-Time Synchronization Protocols for Multiprocessors.* In Proc. IEEE Real-Time Systems Symposium, pages 259–269, 1988. (pp 17, 83, 115)

[**Rajkumar90**] Ragunathan Rajkumar. *Real-Time Synchronization Protocols for Shared Memory Multiprocessors.* In 10 th International Conference on Distributed Computing Systems, 1990. (pp 9, 114, 115)

[**Rajkumar91**] Ragunathan Rajkumar. *Synchronization in Real-Time Systems.* Kluwer Academic Publisher, 1991. (pp 6, 115)

[**Rajkumar93**] Ragunathan Rajkumar, July 1993. Private Communication. (pp 84, 115)

[**Ramamritham89**] K. Ramamritham, J. A. Stonkovic, and W. Zhao. *Distributed Scheduling of Tasks with Deadline and Resource Requirements.* IEEE transactions on Computer, 38(8):1110–1123, 1989. (p 85)

[**Schwan91**] Karsten Schwan and Hongyi Zhou. *Real-Time Threads.* ACM Operating Systems Review, 25:35–46, October 1991. (p 118)

[**Sha90**] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization.* IEEE Transactions on Computer, 39(9):1175–1185, September 1990. (pp 6, 10, 13, 14, 84)

[**Sha93**] Lui Sha, June 1993. Private Communication. (p 117)

[**Shaw74**] Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, 1974. (p 22)

[**Sites92**] R. Sites. *Alpha Architecture Reference Manual.* Digital Equipment Corporation, 1992. (p 114)

[**Sreenan92**] Cormac J. Sreenan. *Synchronisation Services for Digital Continuous Media.* PhD dissertation, Computer Laboratory, University of Cambridge, 1992. (p 3)

[**Stankovic87**] John A. Stankovic and Krithi Ramamritham. *The Design of the Spring Kernel.* In Proc. IEEE Real-Time Systems Symposium, pages 146–156, 1987. (p 4)

[**Steinmetz90**] Ralf Steinmetz. *Synchronisation Properties in Multimedia Systems.* IEEE Journal on Selected Areas in Communication, 8(3), 1990. (p 3)

[**Thacker87**] Charles P. Thacker, Lawrence C. Stwart, and Edwin H. Satterhwaite Jr. *Firefly: A Multiprocessor Workstation.* Technical Report, DEC System Research Centre, 1987. (p 114)

[**Tokuda89a**] H. Tokuda and C. W. Mercer. *ARTS: A Distributed Real-Time Kernel.* ACM Operating Systems Review, 23(3):29–51, July 1989. (p 5)

[**Tokuda89b**] Hideyuki Tokuda, Clifford Mercey, Yutaka Ishikawa, and Tomas Marchok. *Priority Inversions in Real-Time Communication.* In Proc. Real-Time Systems Symposium, pages 348–359, 1989. (p 13)

[**Tokuda90**] H. Tokuda, T. Nakajima, and P. Rao. *Real-Time Mach: Towards a Predicable Real-Time System.* In Proc. USENIX Mach Workshop, 1990. (p 4)

**[Tokuda91a]** H. Tokuda and T. Nakajima. *Evaluation of Real-Time Synchronisation in Real-Time Mach.* In Proc. USENIX Mach Symposium, 1991. (pp 14, 23)

**[Tokuda91b]** Hide Tokuda. *RT-Thread Model for Real-Time Mach.* Technical Report, School of Computer Science, Carnegie Mellon University, 1991. (p 118)

**[Wilkes85]** Maurice V. Wilkes. *Memoirs of a Computer Pioneer.* MIT press, 1985. (p 4)

**[Wilkes92]** Maurice V. Wilkes. *The Long Term Future of Operating Systems.* Comm. of the ACM, 33(11):23–24, 1992. (p 4)

**[Wilkes93]** Maurice V. Wilkes. *Operating Systems in Today's World*, 1993. Special Lecture of Computer Laboratory, University Cambridge. (p 119)

**[Wirth77]** Niklaus Wirth. *Toward a Discipline of Real-Time Programming.* Comm. of the ACM, 20(8):577–583, 1977. (pp 28, 31)