Studies on Scalable Real-Time Kernels for Function-Distributed Multiprocessors

Hiroaki Takada

Doctor Dissertation

Department of Information Science
Graduate School of Science
University of Tokyo

October 1996

Abstract

Recent advances in microprocessor technologies have led to extensive use of computer systems in real world. Because many of these systems require some real-time properties, importance of real-time computing technologies is rapidly increasing. Demands for large-scale and high-performance real-time systems are also growing, and multiprocessor systems, especially *function-distributed multiprocessors*, are often adopted to meet the demands.

In order to reduce the maintenance cost of a multiprocessor real-time system, even when a part of the system is modified or when some processors are added to the system, changes in the worst-case timing behavior of the unmodified part of the system should be minimized. We call this property as *scalability*. Ideally, the worst-case execution time of each routine executed on a processor is determined independently of the number of processors in the system and of the activities of other processors. However, the worst-case execution time of a routine that exclusively accesses a shared resource is unavoidably prolonged, as the number of contending processors is increased.

When a real-time system is realized on a function-distributed multiprocessor, external devices and tasks handling them are allocated to processors so that the number of inter-processor synchronizations is minimized and that as many time-critical tasks as possible are closed within a processor. Therefore, it is advantageous that the worst-case timing behavior of the processings that can be done within a processor is determined independently of the number of processors in the system and of the other processors' activities.

In this dissertation, we discuss the specification and implementation issues of a real-time kernel that facilitate to realize scalable application systems on existing shared-memory multiprocessor system. In order to realize scalable systems, the real-time kernel itself must also be scalable. Though real-time kernels running on shared-memory multiprocessors have been actively studied, none of the studies has focused on the scalability of worst-case behavior.

At first, we clarify the desired properties of a *scalable real-time kernel* for function-distributed multiprocessors, and summarize them in four required properties. Implementation approaches of a real-time kernel on shared-memory multiprocessors are discussed, and two obstacles for satisfying the required properties are pointed out; lack of scalability in local operations, and incompatibility of predictable inter-processor synchronization and constant interrupt response. Then, we propose their solutions when task-independent synchronization and communication objects, such as semaphores and eventflags, are not supported. With the proposed method, the four required properties are satisfied, and the execution time and the response time of each kernel service have reasonable upper bounds. In these discussions, we assume that the underlying inter-processor synchronization mechanism and hardware architecture have some necessary properties.

We also propose the approach to classify kernel resources into classes with different characteristics to improve the performance of local operations. Among them, a task belonging to the private class satisfies the condition that its maximum execution time is independent of the number of contending processors, but the task cannot directly synchronize or communicate with other processors.

Effectiveness of our proposed methods are demonstrated through performance measurements using an existing multiprocessor system. Though the evaluation environment dose not satisfy the assumption on underlying inter-processor synchronization and hardware, it is confirmed through the measurements that the four required properties of a scalable real-time kernel are practically satisfied with our proposals, while they cannot be met at the same time with other methods.

In the second half of this dissertation, we investigate on spin lock algorithms for use in scalable real-time kernels for function-distributed multiprocessors. We propose two kind of spin lock algorithms, queueing *spin lock with preemption* and *spin lock with local precedence*, which are combined to use in our implementation of a scalable real-time kernel. We also discuss the scalability issues on *nested spin locks*, and propose the scheme to make nested spin locks scalable and the algorithms of *priority inheritance spin locks*. The effectiveness of these algorithms is also demonstrated through performance evaluations.

Acknowledgment

This work is owing to the support, guidance, and encouragement of many people. At first, I would like to express my highest appreciation to my supervisor, Professor Ken Sakamura of University of Tokyo. He has taught me a great deal about doing research on computer science. His research philosophies have always guided my research to the *proper* direction in these eight years. He has also given me a prominent research environment: the goal of the TRON Project which is the essential background of this study, good working environment, excellent colleagues, connections with many researchers in the industries, and so on.

Next, I would like to express my gratitude to the former and current members of Sakamura Laboratory for their comments to this work and encouragement. Especially, I would like to acknowledge Dr. Noboru Koshizuka and Mr. Hirohisa Mori for their useful suggestions and discussions. Also, I would like to thank my co-workers, the members of the real-time system group of our laboratory, including Mr. Cai-Dong Wang and Mr. Nobuo Sakiyama. I also thank Ms. Kaori Shiomi, the secretary of Prof. Sakamura, for her assistance and care to my daily working environment. Mr. Nobuhiko Nishio and Mr. Yasushi Saito have made much contribution to the initial stages of the development of ItIs (ITRON Implementation by Sakamura Lab). Thanks also go to Mr. Hideki Yoshida, Mr. Yasuharu Sasaki, and Mr. Tomonori Usaka who have done much effort to port the GNU software development environment for TRON-specification microprocessors.

I also would like to thank the members of the ITRON Technical Committee of TRON Association, especially Dr. Kiichiro Tamaru of Toshiba Corp., Mr. Kenji Kudou of Fujitsu Device Inc., and Mr. Tsuyoshi Shimizu of Hitachi, Ltd., for their fruitful discussions on the current issues of the real-time operating systems for embedded systems. Special thanks go to Mr. Hideo Tsubota of Mitsubishi Electric Corp., because the discussion with him has given me an initial motivation of this work. I also thank to other people engaged in the TRON Project including Mr. Akira Matsui of Personal Media Corp. and Mr. Toshikazu Okubo of NTT.

I am owing much to other researchers of the real-time system community in Japan and foreign countries, especially Dr. Kenji Toda of ETL, Dr. Morikazu Takegaki and

Mr. Hiro Kanamaru of Mitsubishi Electric Corp., Prof. Heonshik Shin of Seoul National Univ., Prof. Hideyuki Tokuda of Keio Univ. and CMU, and Prof. Tatsuo Nakajima of JAIST, for their valuable comments on my research. Also, I would like to appreciate Prof. Theodore Johnson of Univ. of Florida and Mr. Travis S. Craig of Univ. of Washington for their discussions and suggestions on spin lock algorithms for real-time systems. I also would like to thank the anonymous reviews of our papers presented in various journals and symposiums.

In these seven years, I have also engaged in the work to construct the Internet in Japan. I am owing much to the researchers of the Internet community in Japan for their support and encouragement, though they have little direct relation with this study. Among them, special thanks go to Prof. Masaki Hirabaru of Nara Inst. of Science and Technology, Prof. Tsuneyoshi Kamae of Univ. of Tokyo, Mr. Koki Higashida of Science Univ. of Tokyo, Prof. Shigeki Goto of Waseda Univ., Ms. Atsuko Oka of NTT, Mr. Ikuo Kojima of JPNIC, Prof. Naomasa Maruyama of The Inst. of Statistical Mathematics, and Prof. Jun Murai of Keio Univ.

Finally, my deepest appreciation is reserved for my beloved wife, Yoshino, for her support, moderate encouragement, and patience that made this work possible. Thanks also go to my son, Yasufumi, and my daughter, Haruka, for their innocent smiles.

Contents

List of Figures

List of Tables

Part I

Introduction

1 Real-Time Systems and Real-Time Kernel

A *real-time system* is a system in which the correctness of the system depends not only on the logical results of the computation, but also on the time at which the results are produced [64, 62].[1] In other words, a real-time system is required to satisfy a set of *timing constraints*. In a *hard* real-time system, severe consequences can result if a timing constraint is not satisfied.

Timing constraints on a real-time system always come from the external environment. Therefore, a real-time system has some relations to its external environment. From another point of view, a real-time system is considered to be embedded in a larger environment, and thus is also called an *embedded system* [15].

In [15], four fundamental requirements on real-time systems are listed: *timeliness*, *simultaneity*, *predictability*, and *dependability*. The first two of them are user requirements. Timeliness means that a system must satisfy the given timing constraints, which are typically described in the form that the result of the computation must be produced within the predefined and predictable time-bound, called the *deadline*. Consequently, not the average but the worst-case timing behavior, i.e. the worst-case execution times and the worst-case response times, are primary concern in real-time systems. The worst-case execution (or response) times usually correspond to the maximum execution (or response) times.[2] Simultaneity means that real-time systems must provide parallel processing capabilities to cope with the native simultaneity of the external environment.

Predictability and dependability are supplementary requirements to the former two requirements. Predictability means that the functional and timing behavior of a system should be as deterministic as necessary to satisfy system specification [62]. More precisely, "predictability means that it should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made, for example, concerning failures and workloads" [65].

A *real-time kernel*, also called as a real-time monitor or a real-time executive, is the basic software module around which a real-time system is realized. The essential role of a real-time kernel is to support multitasking facility for the requirement of simultaneity. It should also support inter-task synchronization and communication functions and basic memory management functions. On the other hand, it is not necessary for a real-time kernel to handle various external (or input/output) devices directly. One of the reasons

[1]This definition is one of the many definitions of a real-time system (or computing). We consider that this statement is appropriate for the definition of a (general) real-time system, though Stankovic and Ramamritham defined a *hard* real-time system with this statement in [64].

[2]If the result of a computation is obtained too early, it is usually possible to wait until the appropriate time.

is that the external devices should be handled by tasks running on a real-time kernel, because their response times are generally very long compared to the response times of the core components of a computer system (such as processors and memories). Another reason is that there is a great variety of external devices attached to deeply embedded systems, and that efficient and uniform handling of them is very difficult.

The role of a real-time kernel can be paraphrased in contrast to the role of an operating system as follows. Supporting the construction of an application system through the *virtualization* of hardware resources of a computer system is an essential role of an operating system. A real-time kernel is a core module of an operating system that virtualizes only processors and memories.

2 Function-Distributed Multiprocessors

As the application areas of real-time systems expand, requirements for large-scale and high-performance real-time systems are increasing. Areas of rapid growth include large-scale control systems (plant- and aircraft-control systems), transaction processing (on-line banking and seat reservation systems), and communication processing (network routers and switches).

In these application areas, a large number of external devices such as sensors, actuators, and network controllers are connected to a system, and the system is required to respond to the external events from the devices within predefined and usually short time-bounds. It is usually the case that such a system also requires large computational power. To meet these requirements, multiprocessor systems are often adopted to real-time systems.

Because the required processing time for each external device can be estimated beforehand in most real-time systems, it is preferable that each device is handled by a fixed processor (or a fixed set of processors) and that the interface with the device is connected to the local bus of the processor. A distributed shared-memory architecture is also adopted, in which memory modules are connected to the local buses of processors (Figure 1). In this kind of *function-distributed* (or asymmetric) *multiprocessors*, because the code and data areas of the program that handles an external device are placed in the local memory of its host processor, the number of shared-bus (or interconnection network) transactions can be reduced compared to symmetric multiprocessors. This is profitable not only because the high-performance shared bus and expensive cache mechanisms can be omitted, but also because the predictability of the system can be improved through the reduction of access conflicts on the shared bus.

As a general rule, when a real-time system is realized on a function-distributed

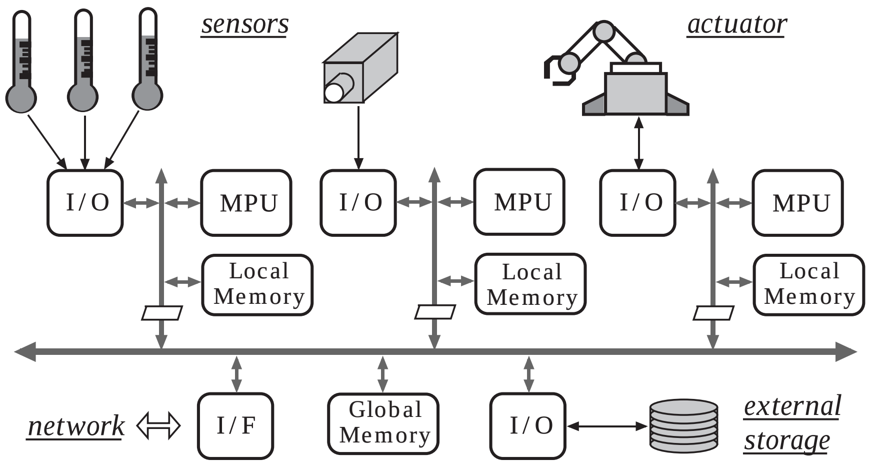


Figure 1: An Example Use of Function-Distributed Multiprocessors

multiprocessor, external devices and tasks handling them are allocated to processors so that the following goals are satisfied; (1) the number of inter-processor synchronizations and communications is minimized and (2) as many time-critical tasks as possible are closed within a processor. Consequently, in well-designed systems on function-distributed multiprocessors, many tasks, including most of the time-critical tasks in the system, can be processed without synchronizing or communicating with other processors. In other words, tasks on a processor are fairly independent with tasks on other processors.

Multiprocessor systems discussed in this dissertation are those consisting of several or around ten processors. Massively parallel systems are outside the scope of this study.

3 Real-Time Scalability

It is often the case that functional or performance requirements on a system are changed during its life-time. It is also a frequent situation that the system is required to support some additional devices. In order to reduce the maintenance cost of the system in such situations, it is advantageous that modifications of a part of the system do not affect the timing behavior of the unmodified parts of the system. When the computational resources of the system are insufficient for the new requirements, the measure is often adopted with a function-distributed multiprocessor system that one or more processors are added to the system. The maintenance cost of the system can be greatly reduced, if the changes in timing behavior of the unmodified parts of the system are very small in this situation. We call this property as *scalability*[3] or *modularity in time domain*.

Because the worst-case behavior is the primary concern in real-time systems, the timing

[3]We use the word "scalability" with stress on the case that processors are added to the system.

behavior mentioned above should be the worst-case timing behavior. Consequently, the above requirements can be summarized as follows. Even when a part of the system is modified (including the case that some processors are added to the system), the extension of the worst-case execution times and response times of the unmodified parts of the system should be minimized. This property is called *real-time scalability*, or simply *scalability*, in this dissertation.

Real-time scalability also facilitates the reuse of a module consisting of a processor, local memory, external devices, and the software handling them, i.e. the reuse in the unit of a board in Figure 1. With real-time scalability (or modularity in time domain), timing constraints imposed on the tasks executed within the module are kept satisfied no matter what kind of system the module is reused for.

It goes without saying that scalability is also an important issue when the number of processors is very large, though we do not investigate on massively parallel systems in this study.

4 Objectives of This Study

The objectives of this study is to clarify the desired properties of a real-time kernel for function-distributed *shared-memory* multiprocessors that facilitates to realize scalable real-time systems, and to propose its realization methods in both specification and implementation aspects. In order to realize scalable real-time systems, the real-time kernel itself must also have the property of real-time scalability.

Real-time kernels running on shared-memory multiprocessor systems have been actively studied and implemented. Famous examples include Spring Kernel [63, 43, 66], Chaos [59, 1], Chorus [49], Harmony [12], and Chimera [67]. However, none of the studies has focused on real-time scalability. In other research areas including real-time algorithms for multiprocessor systems, little attention has been paid to real-time scalability either.

As described in Section 1, predictability is a fundamental requirement in real-time systems. In case of a real-time kernel, predictability means that the maximum execution time and response time of each kernel service are bounded and known beforehand. This is because real-time scheduling algorithms and synchronization protocols are usually implemented within or upon the kernel layer, and because the service times of a real-time kernel itself are treated as *constant* scheduling overheads and cannot be scheduled as *variables* with most real-time scheduling algorithms and synchronization protocols [4, 47, 29].

system call		worst-case
name	function	execution times
cre_tsk	create a task	T_{cre_tsk}
...	...	...
sus_tsk	suspend executing a task	T_{sus_tsk}
	(with a task switch)	T'_{sus_tsk}
rsm_tsk	resume executing a task	T_{rsm_tsk}
	(with a task switch)	T'_{rsm_tsk}
...	...	...
vsnd_tmb	send a message to a task	T_{vsnd_tmb}
	(with a task switch)	T'_{vsnd_tmb}
vrcv_tmb	receive a message sent to me	T_{vrcv_tmb}
	(with a task switch)	T'_{vrcv_tmb}
...	...	...
maximum interrupt response time		T_{int}

Table 1: Timing Behavior of a Uniprocessor Real-Time Kernel

The worst-case behavior of a real-time kernel is usually represented using a table. For example, the maximum execution time of each system call and the maximum interrupt response time of a real-time kernel for single processor systems can be presented like Table 1, where T_{xxx} designates a constant value that is determined for each target hardware.

In case of a multiprocessor real-time kernel, it is ideal that the worst-case execution time and response time of each kernel service are determined independently of the number of processors in the system and of the activities of other processors. However, the worst-case execution time of a routine that exclusively accesses a shared resource[4] is prolonged, as the number of contending processors is increased, at least with its linear order. This is because concurrent executions of the routine must be serialized.[5]

In executing a system call of a real-time kernel, a task usually needs to access some of the kernel data structures exclusively,[6] such as the control blocks of kernel resources (tasks and task-independent synchronization and communication objects) and the ready queue(s).[7] Because these data structures are also accessed from other processors and should be accessed exclusively, the maximum execution time of such system call is

[4]In strict, a shared resource that is fairly accessible from each processor.

[5]This limitation cannot be removed with the techniques of wait-free or block-free synchronizations [17, 37].

[6]With message passings or remote invocations, processors can synchronize without using a shared resource exclusively. In function-distributed shared-memory multiprocessors, however, this synchronization method has some drawbacks. We will describe the drawbacks in Section II.2.3.

[7]A ready queue includes all the tasks that are ready to execute on the processor. The task scheduler utilize it to find the next task to be executed efficiently. In our basic kernel model described in Section II. 2, a ready queue is prepared for each processor.

prolonged as the number of contending processors is increased.

On the other hand, the worst-case timing behavior of the processings that can be done within a processor is desired to be determined independently of the number of contending processors and of the other processors' activities. Processings that can be done within a processor include synchronizations and communications with another task on the same processor and interrupt services requested by the external devices. This property is especially advantageous in function-distributed multiprocessors, because most of the time-critical tasks can be processed without synchronizing or communicating with other processors in well-designed systems. It is also desirable that modifications in some processings that can be done within a processor do not affect the timing behavior of the processings on other processors.

However, these properties cannot be obtained straightforwardly. In this dissertation, we propose a realization method of a scalable real-time kernel with these properties without task-independent synchronization and communication objects (such as semaphores and eventflags), and point out the difficulty of supporting task-independent synchronization and communication objects. In order to realize a scalable real-time kernel on an existing multiprocessor system, we investigate on spin lock algorithms for use in scalable real-time kernels for function-distributed multiprocessors.

5 Outline of This Dissertation

The organization of this dissertation is described in this section. We have presented the main contributions of this dissertation in various journals and symposiums. Each reference cited in this section shows the paper in which the contribution is presented.

In the rest of Part I, we introduce the evaluation environment with which the performance of our proposed realization methods of real-time kernels and underlying algorithms is measured. The evaluation metric used in the following parts is also described.

Part II discusses the realization methods of a scalable real-time kernel for function-distributed multiprocessors. At first, Section 1 presents the overview of the ITRON[8] specifications, a series of standard real-time kernel specifications for embedded systems. The ITRON-MP[9] project, which is to extend the ITRON specifications to support shared-memory multiprocessors, is also outlined [72]. In Section 2, the basic real-time kernel model for function-distributed multiprocessors is described and its implementation

[8]ITRON is an abbreviation of "Industrial TRON" and TRON is an abbreviation of "The Real-time Operating system Nucleus."

[9]"MP" stands for MultiProcessor.

approaches are discussed [71]. Two implementation approaches, direct access method and remote invocation method, are introduced and some drawbacks of the latter method are pointed out [82]. The section also discusses the issue on lock granularity.

In Section 3, two problems in implementing a scalable real-time kernel are described; the problem that the worst-case execution times of synchronizations within a processor depend on the number of contending processors [83], and the problem that predictable inter-processor synchronization and constant interrupt response are incompatible [76]. The section also summarizes the required properties of a scalable real-time kernel.

Then, our proposed solutions to these problems when task-independent synchronization and communication objects are not supported are presented in Section 4 [83]. With the proposed methods, each worst-case service time that is necessary for schedulability analyses can be bounded, on the assumption that underlying inter-processor synchronization mechanism and hardware architecture satisfy the necessary properties, which are also described in this section.

In Section 5, we propose a new kernel model in which tasks and task-independent synchronization and communication objects are classified into some classes with different characteristics [82]. For example, there exists a class of tasks whose maximum execution times are independent of the number of contending processors, but the tasks of this class cannot synchronize or communicate with the tasks executed on other processors. Another class of tasks can synchronize with the tasks on other processors, but their worst execution times depend on the number of contending processors. The kernel resources belonging to the class having the appropriate properties for a processing should be used for implementing the processing.

In Section 6, the effectiveness of our proposed methods is investigated through performance measurements. In the measurements, underlying inter-processor synchronization is realized with spin locks implemented with software, which do not have the necessary properties described in Section 4. The hardware platform used for the measurements does not have the necessary properties, either. In spite of the missing properties in our evaluation environments, the advantage of our proposals over other methods is confirmed through the measurements.

In Section 7, the difficulty of realizing a scalable real-time kernel that supports task-independent synchronization and communication objects is discussed [84]. In the system calls that operate on a task-independent synchronization object, both the lock guarding the control block of the synchronization object and the lock guarding the control block of the task must be acquired one by one. This kind of *nested locks* are the obstacle for satisfying the required properties of a scalable real-time kernel. Finally, the main contributions of Part II are summarized in Section 8.

Part III discusses spin lock algorithms for use in scalable real-time kernels. Spin lock is a fundamental synchronization primitive for exclusive access to shared resources on shared-memory multiprocessors. In realizing a scalable real-time kernel described in the previous part, the characteristics of underlying mutual exclusion mechanisms, i.e. spin locks, have great importance. In this study, we assume that processors support atomic read-modify-write operations on a single word (or aligned contiguous words) of shared memory and propose some extensions to existing spin lock algorithms. Typical examples of the read-modify-write operations are test_and_set, fetch_and_store (swap), fetch_and_add, and compare_and_swap. A brief survey on spin lock algorithms using these operations is presented in Section 1.

In Section 2, we propose two algorithms of queueing spin lock with preemption. We point out that conventional spin lock algorithms cannot satisfy two important requirements on scalable real-time systems, namely, predictable inter-processor synchronization and constant interrupt response, at the same time, and present two spin lock algorithms to solve this problem [76, 79]. These algorithms, which are extensions of queueing spin locks modified to be preemptable for servicing interrupts, can give upper bounds on the times to acquire and release an inter-processor lock, while achieving constant response to interrupt requests. We also demonstrate that the algorithms have required properties through performance measurements in this section.

Section 3 presents an algorithm of spin lock with local precedence, which is necessary to make the worst-case execution times of intra-processor synchronizations independent of the number of contending processors. Though spin lock with local precedence can be realized using a priority-ordered spin lock algorithm, the overhead of priority-ordered spin locks is generally quite large. We propose a more efficient algorithm in this section.

Section 4 and Section 5 discuss two issues on nested spin locks, which are necessary to implement task-independent synchronization and communication objects. In Section 4, the scalability issue of the maximum execution times of critical sections guarded by nested spin locks is discussed. With the simplest method, the maximum execution times become $O(n^m)$, where n is the number of contending processors and m is the maximum nesting level of locks. In this section, we propose an algorithm with which this order can be reduced to $O(n \cdot e^m)$ and demonstrate its effectiveness when $m = 2$ through performance measurements [80]. The proposed method requires *priority inheritance spin lock*, a spin lock algorithm that are enhanced with the priority inheritance scheme, when $m > 2$.

In Section 5, we present two algorithms of priority inheritance spin locks and demonstrate their effectiveness through performance measurements. This section also

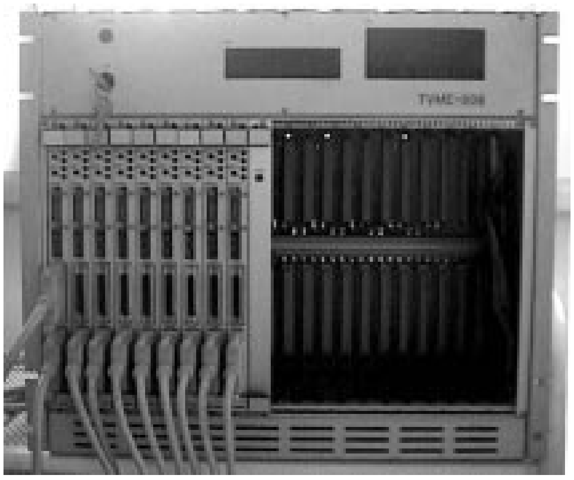


Figure 2: The Front Panel of TRONBOX

illustrates the problem of uncontrolled priority inversions in the context of spin locks. Finally, the contributions of Part III are summarized in Section 6.

Part IV summarizes the overall contributions of this dissertation and describes the future work. The most important future work to do is to solve the difficulty described in Section II.7 for supporting task-independent synchronization and communication objects. Others include the support of the global class of tasks that can be executed on multiple (or all) processors in the system and migrate between them [82].

After the bibliography, Appendix A presents the implementation details of our real-time kernel for multiprocessor systems. Especially, data structures managing the resource classes are discussed. In Appendix B, we present the correctness proofs on the queueing spin lock algorithm with the simple preemption scheme described in Section III.2. We show that the algorithm realizes mutual exclusion and deadlock freedom in this appendix [74].

6 Evaluation Environment and Performance Metric

6.1 Evaluation Environment

In this dissertation, we present the results of some performance measurements of real-time kernels and spin lock algorithms. For the measurements, we use a shared-bus multiprocessor system named the TRONBOX [87] (Figure 2 and 3).

The system consists of nine processor boards and a global memory board which are connected with a shared backplane bus conforming to the VMEbus specification [21]

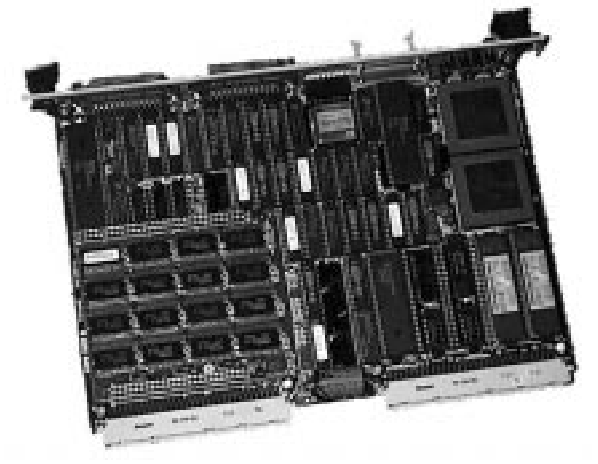


Figure 3: A Processor Board of TRONBOX

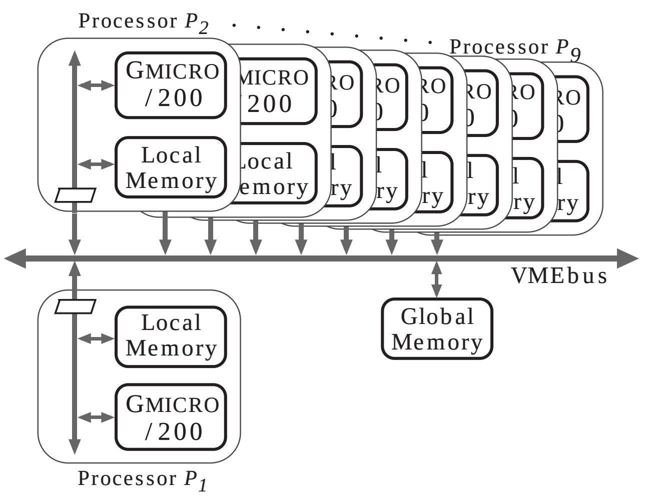


Figure 4: Evaluation Environment

(Figure 4). Each processor board consists of a GMICRO/200 microprocessor [86, 23], 1 MB of local memory, and some I/O interfaces. The GMICRO/200 is the first TRON-specification microprocessor and rated approximately at 10 MIPS with a 20 MHz clock. The local memory can be accessed from other processors through the shared bus. No coherent cache is equipped on the board. Accessing a local memory on another processor board takes nearly 1 μs and is a relatively slow operation compared with the performance of the processor. In our experiments, the data area necessary for each processor and all the program code area are placed in the local memory of the processor. Data requiring only one instance in the system is placed in the local memory P_1 of the master processor or in the global memory.

TRON-specification microprocessors support three read-modify-write instructions:

bit_test_and_set (BSETI), bit_test_and_clear (BCLRI), and compare_and_swap (CSI) [53]. Since the fetch_and_store operation which is used in many spin lock algorithms presented in Part III is not supported, it is emulated using the compare_and_swap instruction and a retry loop. The evaluation programs are written in C programming language, with some inline assembler code for special instructions including the read-modify-write instructions. There is some overhead in passing data between code written in C and code in assembler.

This hardware platform has some problems as our evaluation environment. The problems and our measures to them are as the followings.

1. Because the VMEbus has only four pairs of bus request/grant lines, the round-robin scheme can be applied to at most four bus masters [21]. Therefore, the access time of the local memory of another processor has no upper bound. The maximum execution time of a routine in which a remote memory is accessed cannot be bounded either.

   In our evaluation environment, processors are classified into four classes by the bus request line they use. The round-robin arbitration scheme is adopted among classes and the static priority scheme is applied among processors belonging to a same class.

2. The local memory of each processor board can be accessed from its host processor with the addresses 0x00000000 – 0x000fffff, and can be accessed from other processors with the addresses 0x00n00000 – 0x00$n$fffff where n is the ID number of the board ($1 \leq n \leq 9$). This configuration makes it possible to use the same program code on all processors. Because a processor on board n cannot access its own local memory with the addresses 0x00n00000 – 0x00$n$fffff, however, an address conversion is necessary to follow a pointer between the local memories of different processor boards. This address conversion causes some overhead in pointer operations.

   In evaluating spin lock algorithms, because the case in which this kind of address conversion is necessary is very rare, we convert the address with software when necessary. In implementing a real-time kernel, we convert the address using the MMU (Memory Management Unit) because too many conversions are necessary. In other words, the local memory of a processor is also mapped to the addresses 0x00n00000 – 0x00$n$fffff. The MMU is used only for this address conversion.

3. The processor board causes a bus error under the following condition. If a processor P_j tries to access the local memory of another processor P_i when P_i initiates a

read-modify-write operation on a remote memory, a kind of deadlock occurs in which P_i cannot acquire the shared bus because P_j is using the shared bus, and P_j cannot acquire the local bus of P_i because P_i is using the local bus. This problem occurs because the processor is directly attached to the local bus (no line buffer is used between them), and because the processor would not release the bus once it initiates a read-modify-write operation. To solve this deadlock, the processor board raises a bus error on P_i. When a bus error occurs, P_i should retry the operation with software. This retry overhead is quite large and degrade the preciseness of performance measurements.

In our performance measurements, we record the occurrences of bus errors and subtract the overhead from the measured time if possible. When the estimation of the overhead is very difficult, we discard the measurement time when a bus error occurs.

In spite of these problems, the advantage of our proposals over other methods can be confirmed through the performance measurements.

6.2 Performance Metric

In real-time systems, the effectiveness of implementation methods or algorithms should not be evaluated with their average performance but with their worst-case execution (or response) times. In our performance evaluations, however, adopting worst-case times as performance metric has following difficulties.

1. Worst-case times cannot be measured through experiments because of unavoidable non-determinism in asynchronous multiprocessor systems.

2. With our evaluation environment, the execution time of a routine in which a remote memory is accessed has no upper bound. Therefore, the worst-case execution times of such routines cannot be determined inherently.

3. It is often possible to give a *practical* upper bound on the execution time of a routine, even if the routine does not have the maximum execution time inherently. For example, if a fetch_and_add operation is emulated with a compare_and_swap and a retry loop, the maximum execution time of the fetch_and_add operation cannot be bounded theoretically.

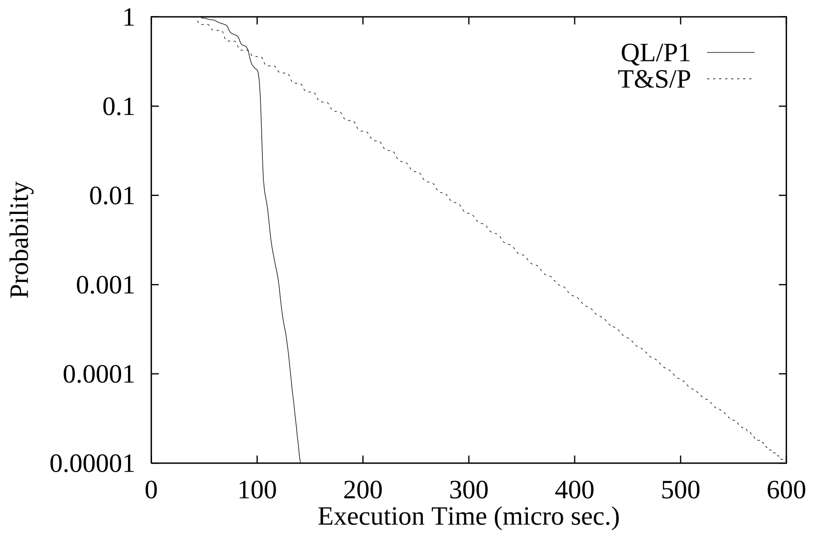


Figure 5: Distributions of Execution Times

4. Adopting the maximum execution (or response) time appeared during a measurement is not appropriate, because the maximum time widely varies for each measurement.

In order to illustrate this situation, we present in Figure 5 the distributions of the execution times of the critical region[10] with the first algorithm of queueing spin lock with preemption (represented as QL/P1) and the test&set lock with preemption (represented as T&S/P), which are described in Section III.2. In this figure, the vertical axis represents the probability that the execution is *not* finished within the specified time in logarithmic scale. We can say that if the probability is rapidly decreasing with the increase of the execution time, a practical upper bound on the execution time can be determined. Consequently, this figure indicates that the execution time with QL/P1 has a practical upper bound, while it is not the case with T&S/P. This demonstrates that T&S/P is not suitable for real-time systems.

From this observation, in place of a worst-case time, we have adopted a p-reliable time, the time within which a processor finishes to execute (or responds) with probability p, as the performance metric. In other words, when a p-reliable time is determined to be the deadline, the probability that the deadline is kept is p, or the deadline miss ratio is $1 - p$.

[10]In strict, this figure presents the distributions of the execution times of the critical region, when no interrupt request is serviced while waiting for a lock. Four processors are executing spin locks. Refer to Section III.2.5 for the details.

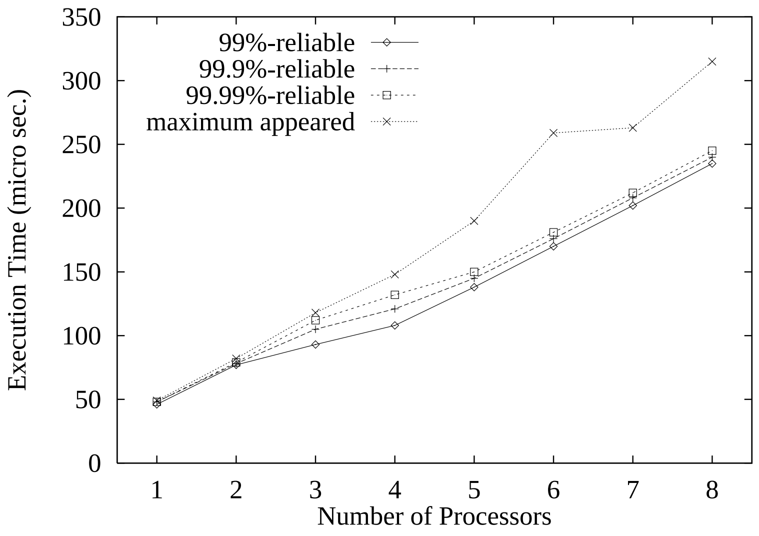


Figure 6: P-reliable Execution Times

In this dissertation, we use the 99.99%-reliable execution (or response) times as the performance metric. Figure 6 presents p-reliable execution times of the critical region with QL/P1 when p = 99%, 99.9%, and 99.99%, and its maximum execution times appeared during the measurements, when the number of contending processors is changed from one to eight. Although the absolute times are different, the same evaluation results can be derived from each performance metric. In order to check the adequacy of the other evaluation results using p-reliable times, we have also confirmed that the same evaluation results can be derived from the maximum times appeared during the measurements.

This performance metric is also justified from application requirements. It is obvious that the failure rate of any system cannot be zero. Even with the hardest real-time system, the system specification cannot require that the failure rate is zero, but that the (estimated) failure rate is below the permissible value determined in design time. The deadline miss ratio of each software component should be as low as necessary that the system as a whole can satisfy the specification. It should be noted that a deadline miss ratio always depends on system workloads. Therefore, a p-reliable time that is obtained through our performance measurements does not correspond to a p-reliable time in application systems. Generally speaking, because we evaluate the performance of implementation methods or algorithms under a very heavy workloads, a p-reliable time with our performance measurements has much higher reliability in application systems.

Part II

Scalable Real-Time Kernels for Function-Distributed Multiprocessors

1 ITRON Specifications and ITRON-MP

In this section, we present the overview of the ITRON specifications, a series of standard real-time kernel specifications for embedded systems in Section 1.2 and 1.3, after a short introduction of the TRON Project in Section 1.1. Then, the overview of ItIs (ITRON Implementation by Sakamura Laboratory), which we extend to support shared-memory multiprocessors, is presented in Section 1.4. We describe the design goals and approaches of ITRON-MP, which is an extension of the ITRON specifications to support shared-memory multiprocessors in Section 1.5.

1.1 TRON Project and ITRON

Recent advances in microprocessor technologies have made every kind of electric and electronic equipment around our daily life embedded with microcomputers and offer higher functions to the users. In the next decade, most kind of equipment, appliances, tools, and other objects making up our living environments will be augmented with embedded computers, be connected with networks, and cooperate each other to provide better living environments for human beings. In other words, these objects and networks constitute a large distributed computing system and support human activities on many aspects. We call this kind of system as a *highly functionally distributed system (HFDS)* and have been conducting a research and development project, called the TRON Project, for its realization [51, 55, 77].

In HFDS environments, a large number of embedded systems are developed and utilized. We have been investigating on standard real-time operating system specifications for embedded systems, called the ITRON specifications, and have published a series of kernel specifications [50, 45, 73, 78]. The reason for centering these studies on kernel specifications is that only the kernel functions are used in most deeply embedded systems. We will describe the overview of the ITRON kernel specifications in the following sections.

The TRON Project is going ahead on various subprojects including the ITRON subproject. The BTRON subproject aims to design an operating system specification for personal computers and workstations. CTRON is an OS interface specifications for communication and information processing. MTRON is an attached OS architecture for connecting various systems in HFDS. CHIP subproject aims to design a VLSI microprocessor architecture for use in these operating systems. HMI subproject designs standard human-machine interface guidelines. Application subprojects, including the TRON-concept Computer Augmented Building subproject, are proceeded to find problems in actual applications of HFDS.

1.2 Design Principles of the ITRON Specifications

Requirements on a standard real-time kernel for embedded systems can be summarized as follows [85, 78].

- Deriving maximum hardware performance.
- Software productivity improvement.
- Uniform application to various processor scales and types.

In order to satisfy these requirements, the following design principles are established in designing the ITRON specifications [78].

- Avoiding excessive hardware virtualization.

  To derive the maximum performance from hardware and achieve high real-time performance, we must limit the amount of hardware virtualization. Although intended for a variety of processors, the ITRON kernel specifications assume each implementation will possess processor-specific aspects.

  To this end, we divided the specification into aspects that are standardized across all processors and implementation-dependent aspects. Standardized items include task scheduling rules; system call names and functions; parameter names, sequence and meanings; and error code names and meanings.

  On the other hand, we did not strictly standardize those aspects that need to be decided separately for each implementation based on runtime performance considerations. Examples are parameter bit size, the method of invoking interrupt handlers, and exception handlings.

- Permitting adaptation to application.

  Modifying the kernel specification and internal implementation method, based on the kernel functions and performance required by a particular application, increases system performance. For embedded systems, the kernel object code is generated for each application, making this adaptation especially effective.

  Specifically, the specification was designed so as to make the kernel functions independent of each other to the extent possible, so that each application can use just the functions it needs. In fact, many ITRON-specification kernels are provided in the form of libraries, and are designed so that only the necessary modules are loaded when the kernel is linked to the application. Also, each system call provides a single function, making it easy to select out the necessary functions for an application.

- Permitting adaptation to hardware.

  Modifying the kernel specification and internal implementation method, based on the characteristics of the hardware and its performance, also increases system performance. For example, the method of invoking interrupt handlers is left unspecified in the ITRON specifications. In fact, it is a usual approach to invoke a user-defined interrupt handler when an external interrupt occurs without going through the operating system. The overhead required here is practically zero. The user must, instead, save the registers used in the interrupt handler.

- Easing training.

  A primary aim of standardization in the ITRON specifications is to facilitate learning by and training of software engineers, so that once they learn something they will be able to apply that knowledge broadly. To archive this, for example, the use of terminology in the specification, and things like the way system calls are named, are made as consistent as possible. Consistent concepts and terminology also leads to the improved communication among software engineers.

- Creating a specification series and/or level divisions.

  Specifications are issued in series and divided into levels to make them applicable to a wide variety of hardware. Of the specifications developed in the past, the μITRON specification (Ver. 2.0) was designed mainly for 8-bit MCUs (Micro-Controller Units) and other smaller-scale systems, while the ITRON2 specification was geared to large-scale systems including 32-bit processors. Moreover, each specification divides functions into different levels based on their degree of necessity. The latest specification, μITRON3.0, uses a level-division of system calls to enable this one specification to cover the range from small-scale to high-performance processors (Table 2).

- Making available a full range of functions.

  Rather than limiting the number of primitives provided by the kernel, the approach is taken of making available a wide variety of primitives with different functions. The idea is to enable implementors to raise the runtime performance and improve

Level R (Required)	Functions required in all μITRON3.0-specification kernels.
Level S (Standard)	Functions to be provided in a standard μITRON3.0-specification kernel.
Level E (Extended)	Advanced or additional functions.
Level C (CPU dependent)	Functions dependent on the processor, hardware configuration, or implementation.
Level X (option)	Extended functions that may be introduced as part of system call functions.

Table 2: Levels in μITRON3.0 Specification

ease of programming by using primitives suitable for the particular hardware and application.

A concept common to many of these design principles is that of *loose standardization*. This means setting uniform standards only to the extent that performance will not suffer, rather than trying to force all systems into one rigid mold, and leaving room to decide matters depending on the processor or application.

1.3 History and Current Status of the ITRON Specifications

The first ITRON kernel specification was released in 1987 as ITRON1. Thereafter studies were carried out on a reduced-function specification called μITRON (Ver. 2.0) for smaller-scale 8- and 16-bit MCUs [52], and on the ITRON2 specification for larger-scale systems with 32-bit processors [54]. Both of these were released in 1989.

Of these, the μITRON specification offered very realistic performance even on an MCU with only very limited processing and memory resources, and has therefore been implemented on many different MCUs. Its application has even widened to various 16-bit MCUs as well as 32-bit processors. Just counting the μITRON-specification products that have been registered officially, there are around thirty implementations for more than twenty processors. In addition to them, the μITRON-specification kernel, with its small size and relative ease of implementation, has been used in numerous developments for in-house systems. There are also several μITRON-specification kernels that have been

Consumer Applications	TVs, VCRs, audio components, air-conditioners, washing machines, microwave ovens, rice cookers, lighting
OA Applications	printers, copiers, image scanners, word processors, optical filing systems
Communications	answer phones, ISDN telephones, cellular phones, FAX, broadcasting equipment, wireless systems, antenna controllers, satellite controllers, ATM switches
FA and Other Applications	PDAs, game gear, automobiles, vending machines, electronic musical instruments, digital cameras, FA computers, industrial robots

Table 3: Typical ITRON-specification Kernel Applications

made available as free software.

It goes without saying that the reason for this large number of ITRON-specification kernel implementations is the wide range of application fields and numerous application examples. Table 3 lists some of the applications in which ITRON-specification kernels are used.

As the μITRON-specification kernel has come to be applied to a wide range of fields, a clearer picture has emerged as to the necessity of each function and the performance demands. Also, as noted above, the μITRON-specification kernel has in some instances been implemented for 32-bit processors, something we did not originally anticipate. It was therefore decided to reexamine the existing ITRON specifications, resulting in the release in 1993 of the third-generation ITRON specification, called μITRON3.0 [56]. The main functions in the μITRON3.0 kernel are listed in Table 4.

1.4 Overview of ItIs

ItIs (ITRON Implementation by Sakamura Laboratory) is a real-time kernel developed for research and educational purposes by the members of Sakamura Laboratory [69]. It conforms to the μITRON3.0 specification and runs on TRON-specification microprocessors. The current version implements all the functions in the μITRON3.0 specification up to level E (Extended level), as well as all level X (optional) functions. It also has some original extended functions. The target microprocessors presently supported

Task management

- Direct manipulation and referencing of task status

Task-dependent synchronization

- Task synchronization functions in the task itself

Synchronization and communication

- Three task-independent synchronization and communication functions: semaphores, eventflags, and mailboxes

Extended synchronization and communication

- Two advanced task-independent synchronization and communication functions: message buffers and rendezvous

Interrupt management

- Function for defining a handler for external interrupts
- Function for disabling and enabling external interrupts

Memory pool management

- Functions for software management of memory pools and memory block allocation

Time management

- Functions for system clock setting and reference
- Task delay function
- Timer handler functions, for time-triggered starting

System management

- Functions for setting and referencing the system environment as a whole

Network management

- Management and support functions for a loosely coupled network

Table 4: Main Functions Supported in the μITRON3.0-specification Kernel

are GMICRO/200 [86, 23] and GMICRO/300 [24]. It is designed to be easily ported to other target systems based on TRON-specification microprocessors. Porting to other microprocessors is also possible.

Main features of ItIs are as follows.

- Emphasizing ease of extension and maintenance.

  Development of ItIs is aimed mainly at research and educational use. For this reason, the implementation emphasizes such factors as ease of understanding, modification, and maintenance over run-time performance. For example, C language is used throughout, with assembly language use kept to a bare minimum.

  ItIs implements all the functions in the μITRON3.0 specification and can be reconfigured as needed by means of compile options, as the amount of kernel coding is approximately 8,000 lines, including the generation script and definition files (but not including blank lines or comments).

- Supporting two system call interfaces.

  The μITRON3.0 specification defines two different interfaces for invoking system calls, one using a software interrupt with a function number set in a register, and the other making use of an ordinary subroutine call. ItIs allows both of these methods to be used in the same system. Accordingly, in a large-scale system, subroutine calls can be used in system tasks providing basic services for the system, while other user tasks are able to make use of a software interrupt.

- Providing original extended functions.

  ItIs supports some original extended functions, including functions for automatic ID assignment, debugging support functions, and priority inheritance semaphores [75].

- Taking advantage of the TRON-specification microprocessor architecture.

  ItIs takes full advantage of the high-level instructions, delayed interrupt, and other features of the TRON-specification microprocessor architecture. Because of the policy of minimizing assembly language use, the functions using high-level instructions are written in an inline assembler, which is called by a C language routine. The same functions are also provided as C language routines to facilitate porting to other microprocessors.

- Designed for flexible reconfiguration.

Changes in the kernel configuration are generated from the source code, enabling flexible reconfiguration.

- Available as free software.

ItIs also supports a simulation environment running on BSD UNIX. Multiple tasks are switched and run in a UNIX process, an approach that makes it usable as a prototyping environment for system development on an ITRON-specification kernel. Use as a thread library on UNIX is also possible, and this environment has the potential for effective use in education and training regarding the ITRON specifications [70].

1.5 Design Goals and Approaches of ITRON-MP

ITRON-MP is an extension of the ITRON kernel specifications to support shared-memory multiprocessors. The design goals of the ITRON-MP specification are as follows.

- ITRON-MP should be implementable with satisfying the required properties of a scalable real-time kernel, which will be described in Section 3.
- ITRON-MP should be valid for various multiprocessor architectures. Namely, it has the adaptability to an architecture.
- The kernel code for an application system can be generated to be optimal for the nature of its application. Namely, it has the adaptability to an application.
- An ITRON-MP based kernel must not degrade the native performance of a machine or an architecture.
- Programmer can easily grasp the real-time natures of the system developed on an ITRON-MP based kernel.
- ITRON-MP should be applicable to applications requiring fault-tolerance.
- The ITRON-MP specification should be easy to learn.

The first goal is the main theme of this dissertation and is discussed in the rest of Part II.

The second goal means that a real-time kernel based on the ITRON-MP specification can be used for various multiprocessor architectures in spite of the differences among them, such as the kind and the number of processors, how to connect processors each other, and the accessibility of hardware resources from each processor. In order to achieve this property, a standard set of kernel interface which can be adapted to wide varieties of multiprocessor architectures is defined in the specification, and a tuned kernel code, which is generated from the description of the architecture and the kernel constitution, is used for the construction of application systems.

The sixth goal comes from the fact that fault-tolerance is another important feature for almost all real-time systems. The adoption of multiprocessor architecture to a fault-tolerant system is a promising approach and has been studied for a long period [19]. Because the actual mechanism to achieve fault-tolerance varies for each system, ITRON-MP should serve as a basis for the construction of fault-tolerant systems. Therefore, we include some kernel functions necessary for the realization of fault-tolerant feature in the ITRON-MP specification. For example, ITRON-MP has a set of system calls which enable user programs to take a snapshot of a task and to resume the task from the snapshot. In other words, ITRON-MP should have the adaptability to a fault-tolerant architecture.

The other goals of the ITRON-MP specification are inherited from the ITRON specifications. The same approaches with ITRON are also valid for ITRON-MP.

2 Basic Kernel Model

In this section, the basic real-time kernel model for function-distributed multiprocessors is presented (Section 2.1) and its implementation approaches are discussed. Two implementation approaches, direct access method and remote invocation method, are introduced in Section 2.2 and some drawbacks of the latter method are pointed out in Section 2.3. We also discuss the issue on lock granularity in Section 2.4.

2.1 Basic Kernel Model for Function-Distributed Multiprocessors

When a hard real-time system is realized on a function-distributed multiprocessor, the method is often adopted as a realistic approach that a real-time kernel for single processor is used on each processor, and that synchronizations and communications among processors are implemented with application-level programs. However, this method has a drawback that when the configuration of the system is modified due to the change of the requirements for example, and when the allocation of the tasks to the processors is changed, a large part of the application program is necessary to be modified. This is because the synchronization and communication interface with tasks on the same processor and that with tasks on other processors are different.

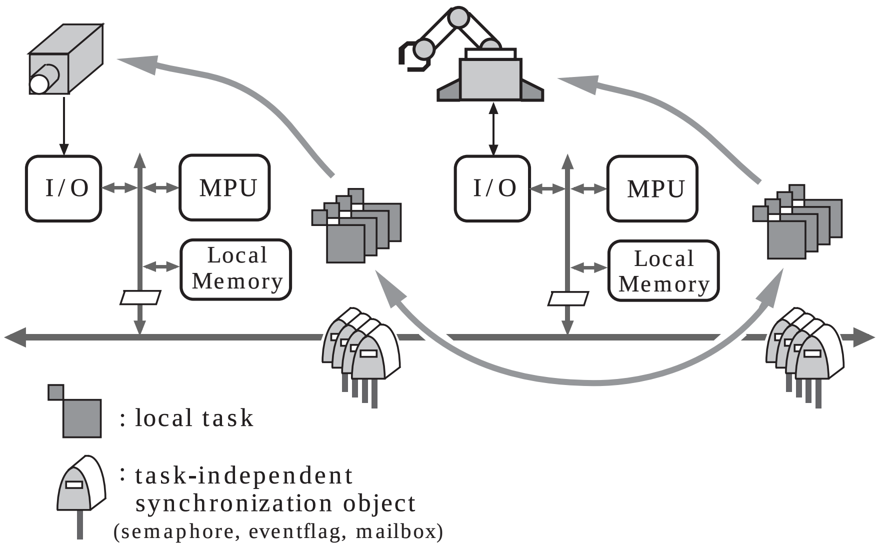

I/O
MPU
Local Memory
I/O
MPU
Local Memory
: local task
: task-independent synchronization object (semaphore, eventflag, mailbox)

Figure 7: Basic Kernel Model

To remedy this problem, a real-time kernel is required with which a task can synchronize and communicate with tasks on other processors with the same interface with tasks on the same processor. In other words, a task can operate on any task with the same set of system calls. In this dissertation, we call this kernel model as the basic model of real-time kernels for function-distributed multiprocessors, or the basic kernel model in short (Figure 7). In the basic kernel model, each task has its host processor on which it is executed, and is called a *local task* of the processor. A ready queue is prepared for each processor in which all the local tasks that are ready to execute are included in the descending order of their priorities. Each task-independent synchronization and communication object (called as synchronization objects or simply as objects in this part), such as a semaphore and an eventflag, also has its host processor and can be accessed from any task in the system. In other words, each kernel resource is classified into the local resource of its host processor.

2.2 Direct Access Method and Remote Invocation Method

There are two approaches to implementing an operating system kernel on function-distributed shared-memory multiprocessors: the direct access method and the remote invocation method [6, 7].

With the direct access method, when a task operates on a kernel resource on another processor, it directly accesses the control block of the resource located on the local memory of the processor. Therefore, some mutual exclusion mechanism among processors is necessary for the access control of the control blocks. In implementing a real-time kernel, because the execution time of each primitive operation is very short, spin locks are usually used for this exclusive control.

With the remote invocation method, which is also applicable to multiprocessors without shared memory, when a task operates on a kernel resource on another processor, it sends a message to the processor requesting the operation and receives the result. The requesting processor spins until the requested processor completes the operation.

Below, we will illustrate the behavior with these two approaches when a task τ_1 on processor P_1 invokes a system call to resume the execution of a task τ_2 on another processor P_2. We denote the resume task operations as `rsm_tsk` after the system call name in μITRON3.0 specification [56].

Direct Access Method

At first, τ_1 finds the address of τ_2's task control block (TCB) and then tries to lock the lock unit guarding the TCB. When τ_1 succeeds to acquire the lock, it accesses the TCB and changes the status of τ_2. Because τ_2 becomes ready to execute with the operation, τ_1 acquires the lock unit guarding the ready queue of P_2[1] and enqueues τ_2 to the ready queue. If (and only if) P_2 executes lower priority task than τ_2 (the priority of the currently executed task must be stored on a shared memory), τ_1 requests τ_2 to switch the executing task using an inter-processor interrupt.

Remote Invocation Method

At first, τ_1 checks some kind of parameter errors which can be detected statically. Then, it enqueues a *request information block* into the request queue of P_2. The request information block includes the kind of operation (`rsm_tsk`, in this case), the parameters passed to it (the identification of τ_2), and an empty field to which the requested processor writes the result. Then, τ_1 asks P_2 to process the request using an inter-processor interrupt and spins until the result of the operation is written in the request information block. When P_2 accepts the interrupt, it dequeues the request information block, executes the requested operation, and writes the result in the block.

Which method of them is appropriate is determined by the characteristics of the underlying hardware (e.g. remote memory access cost) and the performance requirements of the application.

[1]This is necessary, only when the ready queue of P_2 is included in a different lock unit with τ_2's TCB.

2.3 Drawbacks of the Remote Invocation Method

From the performance requirements of real-time applications, the direct access method is usually suitable because the serialization unit of processing is too large with the remote invocation method. More precisely, the remote invocation method has the following drawbacks in implementing real-time kernels for function-distributed multiprocessors.

1. With the remote invocation method, because requests come from other processors asynchronously, any task can be delayed by the processing of the requests. This makes the schedulability analysis of the system very difficult.

   In order to predict the timing behavior of time-critical tasks, it is possible to disable interrupt services during their executions. With this method, however, a request that makes a higher priority task executable is also pended. It is also difficult to predict the maximum time since the time-critical task are completed (or blocked) until another task starts, because all pended requests are processed at this moment. It also has a problem that the requesting task must wait for the completion of the time-critical task.

2. In functional-distributed multiprocessors, interrupt requests from external devices are raised on each processor. If an external interrupt has a higher priority than the inter-processor interrupt, the execution of a requested operation can be delayed due to the service of the external interrupt. This makes it difficult (or even impossible depending on the situation) to bound the time until the remote invocation is finished. Otherwise (i.e. if the inter-processor interrupt has a higher priority than an external interrupt), it is difficult to bound the response time to the external interrupt.

   When a requested operation is very simple and its result is not necessary, the requesting processor can proceed without waiting for the completion of the operation. In this case, this problem can be avoided by making the priority of the external interrupt always higher than that of the inter-processor interrupt. However, the limitation that the requesting task cannot receive the result of a remote operation at all is usually too restrictive to realize the access transparency of remote resources.

With these reasons, we adopt the direct access method as the base implementation method below. We will also refer to the remote invocation method when necessary. Differences of these methods will be clarified through performance measurements in Section 6.

2.4 Kernel Data Structures and Lock Granularity

In implementing a real-time kernel for shared-memory multiprocessors, the lock granularity of kernel data structures is one of the most important issues. Below, we first describe the data structures and access patterns on them in a real-time kernel for single processor systems, and then investigate on the granularity of lock units.

In general, using fine-grained lock units reduces lock contention rate and then improves concurrency. Conversely, using coarse-grained lock units reduces lock acquisition overhead and deadlock avoidance overhead. For real-time kernels, making lock units so small that many locks are necessary to be acquired in some operations is not a suitable approach. This is because the execution time of each critical section is very short in real-time kernels and therefore the lock acquisition overhead is relatively large. Another reason is that the necessity of acquiring multiple locks at the same time has a great impact on the worst-case behavior, because the maximum execution time of a critical section guarded by nested spin locks increases with the exponential order of the maximum nesting level of locks (Refer to Section III.4 for detailed discussions).

The simplest method to avoid nested locks is to enter all kernel data structures in one lock unit. Another method in which all kernel services are executed on one processor is essentially the same approach. With these methods, only one kernel service can be executed at the same time. Therefore, the execution throughput of kernel services cannot scale well and the methods are thought to be problematic from the viewpoint of scalability. It is also reported that the computational power of a processor is not sufficient to execute all the kernel services, when kernel services are heavily used [22, 25]. We consider that kernel data structures on different processors, at least, should be placed in different lock units.

In order to determine an appropriate granularity of lock units, we have examined a real-time kernel implementation for single processors based on the μITRON3.0 specification. Major data structures in the kernel are as the followings.

(1) The task control blocks (TCBs).

(2) The (task) ready queue.

(3) The control blocks of each kind of synchronization and communication objects (including a task queue in which waiting tasks on the object are included).

(4) The timer event queue (a queue which manages various events triggered by the system timer).

As described in Section 2, a ready queue is prepared for each processor in the basic kernel model for function-distributed multiprocessors. Also, a timer event queue should be prepared for each processor.

We analyze the access pattern on the data structures of each system call what should be supported in level S in the μITRON3.0 specification, which is listed in Table 5. For example, the `sig_sem` system call, which returns a resource to the designated semaphore, first accesses the control block of the semaphore. When a task that is waiting on the semaphore becomes ready to execute by the system call, it also needs to access the TCB of the awaked task and the ready queue. The `rel_wai` system call, which forcibly releases the waiting state of the designated task, accesses the TCB of the task and the ready queue. When the task is waiting for a synchronization object and is included in its waiting queue, it also accesses the control block of the object and the TCBs of the tasks that are waiting for the object.

From these observations, because the ready queue is usually accessed with a TCB, we have concluded that the TCBs of the local tasks of a processor and the ready queue for the tasks should be included in the same lock unit. We also conclude that the timer event queue for the tasks should be included in the same lock unit. Another observation is that one-writer/many-readers type synchronization primitives are not necessary. This is because a read access on a data structure is usually followed by a write access.

System calls in Table 5 are classified into the following six categories from their access patterns on the kernel data structures. We omit the accesses on a timer event queue, because whenever the ready queue for the task is accessed, the timer event queue for a task may also be accessed.

(a) Normal operations on a task.

A system call of this category accesses the TCB of the designated task (or issuing task) and/or the ready queue for the task.

(b) Special operations on a task.

A system call of this category accesses the TCB of the designated task, the ready queue for the task, and the control block of the synchronization or communication object on which the task is waiting. In some situations, it also accesses the TCBs of the other tasks that are waiting on the object and the ready queues for the tasks. At most one TCB and the ready queue for it must be locked at once.

(c) Simple operations on a synchronization or communication object.

Name	Function	Category
`sta_tsk`	start a task	(a)
`ext_tsk`	exit the issuing task	(a)
`ter_tsk`	terminate a task	(b)
`dis_dsp`	disable task dispatch	(f)
`ena_dsp`	enable task dispatch	(f)
`chg_pri`	change the priority of a task	(a),(b)
`rot_rdq`	rotate tasks on a ready queue	(a)
`rel_wai`	release a task from wait state	(b)
`get_tid`	get the issuing task identifier	(a)
`sus_tsk`	suspend executing a task	(a)
`rsm_tsk`	resume executing a task	(a)
`slp_tsk`	make the issuing task sleep	(a)
`wup_tsk`	wakeup a sleeping task	(a)
`can_wup`	cancel wakeup requests	(a)
`sig_sem`	signal a semaphore	(e)
`wai_sem`	wait on a semaphore	(d)
`preq_sem`	poll and request a semaphore	(c)
`set_flg`	set an eventflag	(e)
`clr_flg`	clear an eventflag	(c)
`wai_flg`	wait for an eventflag	(d)
`pol_flg`	poll an eventflag	(c)
`snd_msg`	send a message to a mailbox	(e)
`rcv_msg`	receive a message from a mailbox	(d)
`prcv_msg`	poll and receive a message from a mailbox	(c)
`loc_cpu`	disable interrupt and dispatch	(f)
`unl_cpu`	enable interrupt and dispatch	(f)
`ret_int`	return from interrupt handler	(f)
`set_tim`	set the system clock	(f)
`get_tim`	get the system clock	(f)
`dly_tsk`	delay execution of the issuing task	(a)
`get_ver`	get the version information	(f)

Table 5: Classification of System Calls

A system call of this category accesses only the control block of the designated synchronization or communication object.

(d) Wait operations on a synchronization or communication object.

A system call of this category first accesses the control block of the designated synchronization or communication object. When the issuing task is blocked, it also accesses the TCB of the issuing task and the ready queue for the task.

(e) Release operations on a synchronization or communication object.

A system call of this category first accesses the control block of the designated synchronization or communication object. When some tasks that are waiting on the object are released from the waiting states, it also accesses the TCBs of the tasks and the ready queues for the tasks. At most one TCB and the ready queue for it must be locked at once.

(f) Other operations.

A system call of this category does not access these kernel data structures.

Table 5 also presents the category to which each system call is classified. The `chg_pri` system call, which changes the priority of the designated task, is classified into both (a) and (b), because its access pattern varies depending on the state of the designated task.

Another kernel service routine that should be considered here is the timer interrupt handler, which is periodically executed with constant interval and processes various time-triggered events. In processing timeouts, a typical time-triggered event, the handler accesses kernel data structures in the same pattern with the system calls in category (b), i.e. the handler accesses the TCB of the designated task, the ready queue for the task, and the control block of the synchronization or communication object on which the task is waiting.

As the results of these investigations, we conclude that a separate lock unit should be prepared for the control blocks of synchronization and communication objects on each processor. As described before, the TCBs and the ready queue on the processor are included in another lock unit. In order to avoid deadlocks, when both kind of locks are necessary to be acquired, the lock unit of the synchronization and communication objects should be acquired first.

In implementing the system calls of category (b), which are special operations on a task, a deadlock detection and re-execution mechanism must be adopted. Therefore, it is very difficult to bound the maximum execution times of the system calls of this category. Because the system calls of this category is rarely used, we give up solving this problem. In processing timeouts, the synchronization or communication object whose control block is necessary to be accessed can be determined beforehand. Thus it is possible to acquire the lock unit of the object first, and the deadlock can be avoided.

On the other hand, when the TCBs and the control blocks of synchronization and communication objects were included in the same lock unit, two parallel invocations of system calls of category (e), which are used very frequently, could cause a deadlock.

3 Requirements and Problems

This section presents two major problems in implementing a scalable real-time kernel for function-distributed multiprocessors; the degraded scalability of intra-processor synchronization (Section 3.1), and the incompatibility of predictable inter-processor synchronization and constant interrupt response (Section 3.2). We also summarize the required properties of a scalable real-time kernel in Section 3.3.

3.1 Scalability of Intra-Processor Synchronization

The first problem is that the worst-case execution times of inter-task synchronizations within a processor depend on the number of contending processors in the system. This is because a task must acquire an inter-processor lock before it accesses the TCB of another task, even when both tasks are executed on a same processor.

As described in Section I.4, the worst-case timing behavior of the processings that can be done within a processor is desired to be independent of the number of contending processors and of the other processors' activities. Because tasks on each processor are fairly independent with tasks on other processors in function-distributed multiprocessors, this property is an essential requirement to reduce the maintenance cost of the system. It is also a prerequisite to facilitates the reuse of a module consisting of a processor, local memory, external devices, and the software handling them.

3.2 Predictable Inter-Processor Synchronization and Interrupt Response

The second problem is that constant interrupt response is not compatible with predictable inter-processor synchronization. This problem is similar to the problem with the remote invocation method on the precedence of external interrupts and inter-processor synchronizations, which is pointed out in Section 2.3.

In order to bound the time until a processor acquires an inter-processor lock, the duration that each processor holds the lock must be bounded as well as the number of contending processors that the processor must wait for. The latter condition can be met with a bounded spin lock algorithm, such as the ticket locks and the FIFO-ordered queueing locks [38], with which the turn that a processor acquires a lock is reserved when it begins waiting for the lock. To satisfy the former condition, the relationship with interrupt services must be considered.

In function-distributed multiprocessors, interrupt services for external devices are requested for each processor. When multiple devices are connected to a processor, interrupt requests from them are usually raised independently and the maximum time to service all of the requests becomes very long or even unbounded. Consequently, in order to give a practical bound on the duration that a processor holds a lock, interrupt services should be inhibited for that duration (1).

On the other hand, the worst-case interrupt latency should be given independently of the number of contending processors. If a processor disables interrupt services before enqueueing itself to the queue, the interrupt disabled period includes the time to acquire the lock and its upper bound depends on the number of contending processors. Therefore, interrupt requests must be serviced while the processor is waiting for a lock (2).

Though the test-and-set locks, which are not suitable for real-time systems, can be extended to satisfy both (1) and (2) easily, bounded spin lock algorithms, such as the ticket locks and the queueing locks, cannot be extended similarly. The reason is as follows. In all bounded spin lock algorithms, a processor modifies some shared variable and reserves its turn to acquire the lock when it begins waiting for the lock. When its turn comes, the lock is passed to the processor by another. If the processor simply branches to an interrupt handler while waiting for the lock, it cannot begin to execute the critical section immediately after the lock is passed to the processor, and makes the contending processors wait wastefully until the interrupt service is finished.

When a processor finishes the interrupt request that is serviced while waiting for a lock, it resumes waiting for the lock. It is usual that the maximum time that the processor is waiting for the lock is prolonged by the interrupt service. It is also the case with some spin lock algorithms that some processings are necessary after the interrupt service to resume waiting for the lock.

When the schedulability of the system is analyzed, all the overhead that is caused by an interrupt service should be added to the maximum service time of the interrupt request. We call this overhead as *interrupt service overhead*. Because the maximum frequency of interrupt requests is usually quite high compared with tasks, a little increase of the interrupt service overhead can severely degrade the schedulability of the system. Therefore, the interrupt service overhead should also be independent of the number of contending processors.

3.3 Required Properties

From the above discussions, the properties that a scalable real-time kernel should satisfy can be summarized as follows.

system call		intra-processor operations	inter-processor operations
name	function		
cre_tsk	create a task	T_{cre_tsk}	$n \cdot T_{wait} + T''_{cre\_tsk}$
. . .	. . .	. . .	. . .
sus_tsk	suspend executing a task	T_{sus_tsk}	$n \cdot T_{wait} + T''_{sus\_tsk}$
	(with a task switch)	T'_{sus_tsk}	$n \cdot T_{wait} + T'''_{sus\_tsk}$
rsm_tsk	resume executing a task	T_{rsm_tsk}	$n \cdot T_{wait} + T''_{rsm\_tsk}$
	(with a task switch)	T'_{rsm_tsk}	$n \cdot T_{wait} + T'''_{rsm\_tsk}$
. . .	. . .	. . .	. . .
vsnd_tmb	send a message to a task	T_{vsnd_tmb}	$n \cdot T_{wait} + T''_{vsnd\_tmb}$
	(with a task switch)	T'_{vsnd_tmb}	$n \cdot T_{wait} + T'''_{vsnd\_tmb}$
vrcv_tmb	receive a message sent to me	T_{vrcv_tmb}	—
	(with a task switch)	T'_{vrcv_tmb}	—
. . .	. . .	. . .	. . .
maximum interrupt response time		T_{int}	
interrupt service overhead		$T_{int\_overhead}$	

Table 6: Required Timing Behavior

(A) The maximum execution time of a system call that is to synchronize or communicate with tasks on the same processor can be determined independently of the other processors' activities and the number of contending processors.

(B) The maximum execution time of a system call that is to synchronize or communicate with tasks on other processors can be determined independently of the other processors' activities and be bounded with a linear order of the number of contending processors.

(C) The maximum interrupt response time on each processor can be determined independently of the other processors' activities and the number of contending processors.

(D) The interrupt service overhead can be determined independently of the other processors' activities and the number of contending processors.

The required timing behavior is illustrated in Table 6.

4 Proposed Solutions

In Section 4.1 and 4.2, we present our proposed solutions to the two problem described in the previous section when task-independent synchronization and communication objects are not supported. With the proposed methods, each worst-case service time that is necessary for schedulability analyses can be bounded, on the assumption that underlying inter-processor synchronization mechanism and hardware satisfy the required properties, which are described in Section 4.3.

Because task-independent synchronization and communication objects are not considered in this section, all shared data structures located on the local memory of a processor are thought to be included in a single lock unit.

4.1 Spin Lock with Local Precedence

In order to improve the worst-case execution times of an operation on a local task (called a local operation, in short), the local lock guarding the local data structures should be obtained with precedence over the other processors. With this approach, the maximum execution time of a local operation is determined independently of the number of contending processors. More precisely, a task must wait for at most one critical section executed by other processors until it acquires its local lock. On the other hand, the maximum number of critical sections that a processor must wait for until it acquires a non-local lock is increased. More precisely, when a task tries to acquire a non-local lock, it must wait for $n - 1$ critical sections executed by its host processor in addition to $n - 2$ critical sections executed by the other processors, where n is the number of contending processors.

The spin lock algorithms with which the local lock can be acquired with precedence over the other processors, called spin locks with local precedence, will be described in Section III.3.

4.2 Spin Lock with Preemption

To satisfy both of the conditions (1) and (2) described in Section 3.2 at the same time, we adopt FIFO-ordered queueing spin lock algorithms with preemption.

As described in Section 3.2, in a bounded spin lock algorithm, a processor modifies some shared variable and reserves its turn to acquire the lock. In order not to make the contending processors wait wastefully, a processor must inform others that it is servicing interrupts and should not be passed the lock, when it begins to service interrupts while waiting for the lock. The processor trying to release the lock checks if the succeeding processor is servicing interrupts. If the succeeding one is found to be servicing interrupts, the lock is passed to the next in line.

More precisely, when the processor trying to release the lock finds that the succeeding one is servicing interrupts, the processor is dequeued from the waiting queue for the lock. When the processor finishes the interrupt service, it checks whether it is dequeued from the waiting queue during the interrupt service or not. If it has been dequeued, it re-executes the lock acquisition routine from the beginning. Obviously, this simple preemption scheme has the problem that the interrupt service overhead depends on the number of contending processors.

In order to solve this problem, we propose an improved preemption scheme, in which the processor is not dequeued even when its turn to acquire the lock comes during an interrupt service. Instead, the processor trying to release the lock simply passes the lock to the next processor in the waiting queue. When the processor returning from the interrupt service, it resumes waiting for the lock in its original position. With this improved preemption scheme, the interrupt service overhead can be reduced to a constant time length, which is independent of the number of contending processors.

One more problematic situation is as follows. Assume the case that a processor P_1 services an interrupt request while the task executed on P_1 is waiting for a lock. The problem occurs when the interrupt handler executed by P_1 tries to acquire the same lock. If P_1 executes the lock acquisition routine from the beginning, another processor P_2 that has just begun waiting for the same lock must possibly wait for the two executions of critical sections by P_1. As the result, the maximum number of critical sections that P_2 must wait for is increased with an interrupt service executed on P_1. This violates the required property (B) presented in Section 3.3. More precisely, the maximum time until P_2 acquires the lock cannot be bounded with a linear order of the number of contending processors without some assumptions on the occurrence of interrupt requests.

Our solution to this problem is that the interrupt handler trying to acquire the lock inherits the turn that the preempted task have reserved to acquire the lock. In this case, the task must re-execute the lock acquisition routine from the beginning after the interrupt service, and thus the interrupt service overhead is prolonged. Instead, the interrupt service time is shortened, because the interrupt handler inherits the turn reserved by the preempted task. Because the sum of the interrupt service time and the interrupt service overhead remain unchanged, schedulability of the system is not affected with this method. The same method can be applied to the situation that another task that becomes ready to execute by the interrupt service tries to acquire the same lock.

With these methods, all of the required properties described in Section 3.3 are satisfied, on the assumption that underlying inter-processor synchronization mechanism and hardware satisfy the properties described in the next section. Timing behavior of our proposed method is illustrated in Table 7, in which T_{cs} denotes the maximum time duration that a processor holds a lock.

system call		intra-processor operations	inter-processor operations
name	function		
cre_tsk	create a task	T_{cre_tsk}	$2 \cdot n \cdot T_{cs} + T''_{cre\_tsk}$
...	...	...	...
sus_tsk	suspend executing a task	T_{sus_tsk}	$2 \cdot n \cdot T_{cs} + T''_{sus\_tsk}$
	(with a task switch)	T'_{sus_tsk}	$2 \cdot n \cdot T_{cs} + T'''_{sus\_tsk}$
rsm_tsk	resume executing a task	T_{rsm_tsk}	$2 \cdot n \cdot T_{cs} + T''_{rsm\_tsk}$
	(with a task switch)	T'_{rsm_tsk}	$2 \cdot n \cdot T_{cs} + T'''_{rsm\_tsk}$
...	...	...	...
vsnd_tmb	send a message to a task	T_{vsnd_tmb}	$2 \cdot n \cdot T_{cs} + T''_{vsnd\_tmb}$
	(with a task switch)	T'_{vsnd_tmb}	$2 \cdot n \cdot T_{cs} + T'''_{vsnd\_tmb}$
vrcv_tmb	receive a message sent to me	T_{vrcv_tmb}	—
	(with a task switch)	T'_{vrcv_tmb}	—
...	...	...	...
maximum interrupt response time		T_{int}	
interrupt service overhead		$T_{int\_overhead}$	

Table 7: Timing Behavior of the Proposed Method

The bounded spin lock algorithms with preemption will be discussed in Section III.2.

4.3 Assumptions on Underlying Synchronization Mechanism and Hardware

In order that our proposed method strictly satisfies the required properties described in Section 3.3, the following assumptions on underlying inter-processor synchronization mechanism (i.e. spin lock) and hardware are necessary to be satisfied.

1. The maximum execution time of underlying inter-processor synchronization mechanism can be determined independently of the number of contending processors.

2. The maximum access time of a local memory can be determined independently of the number of contending processors.

3. The maximum access time of a remote memory can be bounded with a linear order of the number of contending processors.

Because the maximum execution time of our queueing spin lock algorithm with preemption which will be described in Section III.2 depends on the number of contending processors, the first assumption is not satisfied. However, the dependency is very small and can be ignored in usual applications. For very hard real-time applications, the underlying synchronization mechanism should be implemented with hardware. Because only one lock is necessary for each processor, we think that the cost of the synchronization hardware can be justified.

In order to satisfy the second assumption, the maximum access time of the local bus of the processor should be able to be determined independently of the number of contending processors, or the local memory should be realized using two-port memories. The third assumption requires that the maximum access time of the shared bus (or interconnection network) and that of the local bus of another processor are bounded with a linear order of the number of contending processors.

A hardware architecture in which all these assumption are satisfied with reasonable cost is as follows. A complete round-robin scheme should be adopted as the arbitration scheme of the shared bus (or interconnection network). The local bus of a processor should also be scheduled in a round-robin fashion between its host processor and the other processors. More precisely, after the local bus is used by its host processor, another (remote) processor should be able to acquire the bus. After a remote processor uses the local bus, the host processor of the bus can acquire the bus with higher precedence over the other processors.

5 Classification of Kernel Resources

In the basic kernel model for function-distributed multiprocessors described in Section 2.1, each kernel resource is classified into the *local class* of its host processor. Kernel resources included in each *local class* have the same characteristics except that they are located on the local memory of its host processor and that (in case of local tasks) they are executed only by its host processor.

In this section, we propose a new kernel model in which kernel resources are classified into some classes with different characteristics. The kernel resources belonging to the class having the appropriate property for a processing should be used for implementing the processing.

At first, we introduce the class of private tasks, whose maximum execution times are independent of the number of contending processors, but that cannot synchronize or communicate with the tasks executed on other processors in Section 5.1. Task-independent synchronization and communication objects are also classified into the private class and the shared class in Section 5.2. We also introduce the class of isolated tasks in Section 5.3. Though isolated tasks themselves have little use, the same access restriction with it should be imposed on interrupt handlers. Finally, we describe the kernel interface with which resources of different classes are accessed in Section 5.4.

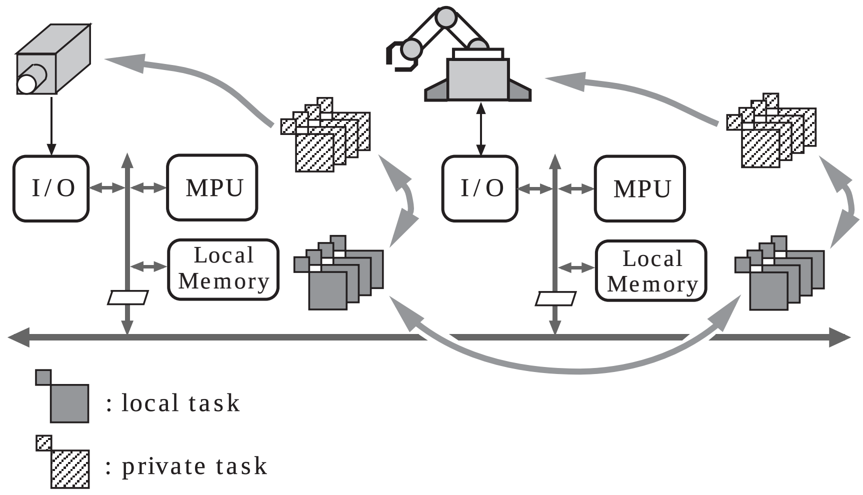


Figure 8: Kernel Model with Private Tasks

5.1 Private Tasks

Though the spin lock with local precedence described in Section 4.1 makes the worst-case performance of an intra-processor synchronization independent of the number of contending processors, its performance is quite low compared with a single processor system. As described in Section I. 2, many of the tasks can be processed within a processor and need not synchronize or communicate with other processors in well-designed application system on a function-distributed multiprocessor. The total performance of the system is expected to be improved, if such tasks can be executed with the same performance with a single processor system.

To meet this requirement, we propose an approach to classify tasks according as their characteristics. In the concrete, we classify the tasks that are not operated by tasks executed on other processors as *private tasks*, which are managed differently from the other tasks (i.e. local tasks). Because the TCB of a private task is not accessed by other processors than its host processor, no inter-processor lock is necessary to access its TCB. A separate ready queue and a timer event queue also accessible without an inter-processor lock are prepared for the private tasks on each processor. Both the ready queue for the private tasks and that for the local tasks are checked in determining which task to be executed. The kernel model with private tasks is illustrated in Figure 8.

A private task on a processor can synchronize or communicate with a local task on the same processor. When the private task accesses the TCB of the local task, the maximum time until it acquires the lock guarding the TCB is independent of the number of contending processors, because the private task, which is on the same processor with the local task, can acquire the lock with precedence over the other processors.

Another motivation to introduce the class of private tasks is as follows. Because the maximum execution time of an operation on a remote resource (called a remote operation, in short) is prolonged as the number of contending processors is increased, a task whose worse-case behavior should not depend on the number of contending processors must not invoke such operations. Moreover, the same restriction applies to any higher priority task than the former task in order to bound its response time independently of the number of contending processors. If this restriction is imposed on each private task, and if the private tasks are always scheduled with higher priorities than the local tasks on the same processor, the worst-case behavior of private tasks can be determined independently of the number of contending processors.

In order to schedule the private tasks with higher priorities than the local tasks, the task dispatcher (a kernel module which switches the contexts of tasks) first checks the ready queue for the private tasks, then checks the ready queue for the local tasks only when the former one is empty, and determines to which task to dispatch.

5.2 Classification of Task-Independent Synchronization and Communication Objects

In order that local tasks on different processors synchronize and communicate each other through task-independent objects (such as semaphores and eventflags), a class of synchronization and communication objects that can be accessed by local tasks on any processor is necessary. We call this class of objects as *shared objects*. When the control blocks of shared objects is located on the local memory of a processor, it is also called *local objects* of the processor. Non-local shared objects are called *global objects*.

When a task operates on a shared object, it is necessary for the task to access the TCBs of other tasks that are waiting on the object in addition to the control block of the object. Because a private task cannot access the TCBs of the tasks on other processors that can wait on a shared object, a private task cannot operate on the shared object. Consequently, in order that private tasks and local tasks on a processor synchronize and communicate each other through task-independent objects, a class of synchronization and communication objects that can be accessed only from the tasks on the processor is necessary. We call this class of objects as *private objects*. No inter-processor lock is necessary to access the control blocks of the private objects (Figure 9).

Table 8 presents the accessibility of each class of kernel resources from each class of tasks. P_1 and P_2 in this table represent different processors in the system, and P_1-private (or local) task denotes a private (or local) task on processor P_1. “*1” represents that a

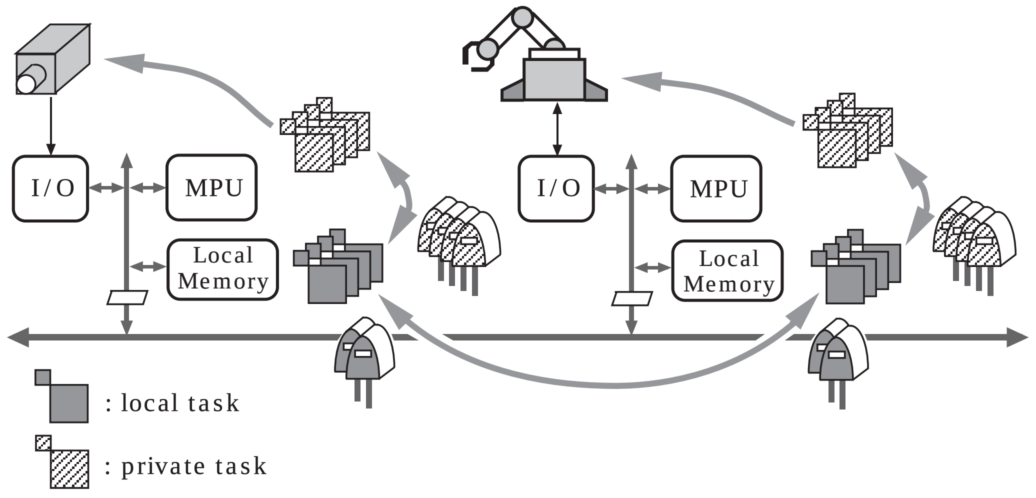


Figure 9: Kernel Model with Private Tasks and Objects

accessing task \ accessed resource	P_1-private task	P_1-private object	P_1-local task	shared object	P_2-local task	P_2-private object	P_2-private task
P_1-private task	OK	OK	*1	NA	NA	NA	NA
P_1-local task	OK	OK	OK	OK	*1	NA	NA

Table 8: Accessibility of Kernel Resources

task can access another task with normal operations (the system calls of category (a) in Section 2.4) but cannot access with special operations (the system calls of category (b)). When a task tries to operate on an unaccessible resource, the kernel reports an error.

In Table 8, a P_1-private task cannot access a P_1-local task with special operations, because the private task cannot access the control block of a shared object on which the P_1-local task may be waiting. A P_1-local task cannot access a P_2-local task with special operations, because the P_1-local task cannot access the control block of a P_2-private object on which the P_2-local task may be waiting.

5.3 Isolated Tasks and Interrupt Handlers

As described in Section 5.1, a private task is necessary to acquire an inter-processor lock when it synchronizes with a local task on the same processor. Therefore, its maximum execution time and response time are long compared with a single processor system. When some deadlines are very short and the same response time with a single processor system is required, another class of tasks that never use inter-processor locks becomes

accessing task \ **accessed resource**	P_1-isolated task	P_1-isolated object	P_1-private task	P_1-private object	P_1-local task	shared object	P_2-local task	P_2-private object	P_2-private task	P_2-isolated object	P_2-isolated task
P_1-isolated task	OK	OK	OK	NA	NA	NA	NA	NA	NA	NA	NA
P_1-private task	OK	OK	OK	OK	*1	NA	NA	NA	NA	NA	NA
P_1-local task	OK	*2	OK	OK	OK	OK	*1	NA	NA	NA	NA

Table 9: Accessibility of Kernel Resources with Isolated Classes

necessary. We call this class of tasks as *isolated tasks*. Isolated tasks are always scheduled with higher priorities than the private tasks and the local tasks on the same processor. Because an isolated task cannot operate on a private object on which a local task may be waiting, *isolated objects* that can be operated on only by the isolated tasks and the private tasks are necessary. An isolated task cannot access even the task control block of a local task because a local task can be accessed from other processors, while a private task can access it.

The accessibility of kernel resources with isolated classes are summarized in Table 9. In this table, "*2" represents that a task can access a synchronization object with the operations of category (c) and (e) but cannot access with the operations of category (d), that is, the task cannot wait on the object. A P_1-local task cannot wait on a P_1-isolated object, because a P_1 isolated task, which cannot access the TCB of the local task, must be able to operate on the object.

In the μITRON3.0 specification, application programmers can write interrupt handlers. System calls can be invoked from interrupt handlers, except for the operations that make the issuing task blocked.[2] Because the execution time of an interrupt handler is included in the maximum response time of isolated tasks, interrupt handlers should not use inter-processor lock and thus the same access restriction with the isolated tasks should be applied. When isolated tasks are not used, it is still reasonable that the same access restriction is applied to interrupt handlers.

A system call that disables interrupt services is prepared in the μITRON3.0 specification. While a task disables interrupt services, both the access restriction on the task and that on an isolated task on the same processor should be applied to the task.

[2]This is because an interrupt handler does not have a task context and cannot be blocked.

5.4 Kernel Interface

The classification of kernel resources is reflected to the kernel interface through ID numbers of the resources. In the μITRON3.0 specification, a kernel resource is accessed with its ID number. We divide a resource ID into the field indicating to which class the resource belongs and the field identifying the resource in the class. With this approach, the system call interface, especially the number of parameters, remains unchanged.

It is usually the case that the ID numbers of kernel resources are represented with symbols in source code and that the mapping of the symbols to actual numbers is given within a definition module. With this guideline, when the class of a kernel resource is changed, only the definition module is necessary to be modified.

6 Performance Measurements

In this section, the effectiveness of our proposals is investigated through performance measurements. The measurement method is described in Section 6.1, and the measurement results, which are to see whether the four required properties of a scalable real-time kernel described in Section 3.3 are satisfied or not, are presented in Section 6.2.

In the measurements, underlying inter-processor synchronization is realized with spin locks implemented with software, which do not satisfy the required property presented in Section 4.3. Our evaluation environment described in Section I.6.1, does not satisfy one of the required properties either. In spite of the missing properties, the advantage of our proposals over other methods is confirmed through the measurements.

6.1 Measurement Method

We have prepared five versions of real-time kernels for the evaluation: a real-time kernel using the proposed method (i.e. spin lock with the improved preemption scheme and local precedence rule; titled “proposed” in the graphs in this section), one using spin lock with the improved preemption scheme but without local precedence rule (“w/o local precedence”), one using spin lock with the simple preemption scheme and local precedence rule (“w/o improved preemption”), one using spin lock without preemption and with local precedence rule (“w/o preemption”), and one using the remote invocation method (“remote invocation”).

We have measured the execution times of system calls and the interrupt response times using two synthetic workloads. The workloads are determined so that worst-case situations can occur.

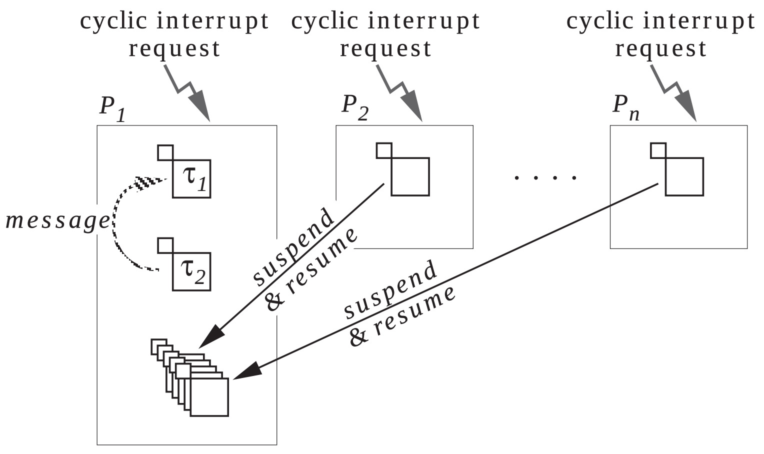


Figure 10: The First Workload

The first workload is to evaluate the performance of a local operation, or an intra-processor synchronization. A local task τ_2 on processor P_1 repeatedly invokes a system call that sends a message to a higher priority local task τ_1 on the same processor, and the execution times of the system call (the time since τ_2 invokes the system call until τ_1 starts execution) are measured. The execution times when an interrupt request is serviced during the execution are recorded separately. The execution times of a system call that sends a message to a private task τ_1 on the same processor are also measured.

In order to interfere the local operation, local tasks on the other processors alternately suspend and resume the execution of lower priority tasks on P_1 at random intervals. The average interval is about 500 μs. During the measurement, periodic interrupt requests are also raised on each processor, and the interrupt response times are measured within the interrupt handler. The interrupt period is around 5 ms and is varied in 0–5% for each processor in order that the timing of interrupt requests for each processor should not be synchronized. The execution time of the interrupt handler including the time for invoking and returning from the handler is about 33 μs. Other external interrupt requests are inhibited during the measurement.[3] The relation among tasks in this workload is illustrated in Figure 10.

The second workload is to evaluate the performance of a remote operation, or an inter-processor synchronization. A local task τ_2 on processor P_2 repeatedly invokes a system call that sends a message to a local task τ_1 on processor P_1, and the execution times of the system call (the time since τ_2 invokes the system call until the execution of the system call is finished) are measured. The execution times when an interrupt request

[3]The inter-processor interrupts should not be inhibited, of course. The word “external” excludes the inter-processor interrupts.

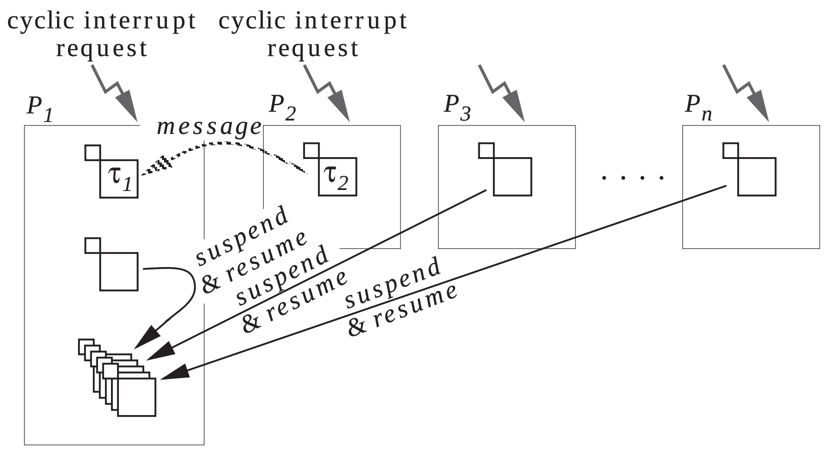


Figure 11: The Second Workload

is serviced during the execution are recorded separately.

In order to interfere the remote operation, a task on P_1 and tasks on the other processors alternately suspend and resume the execution of lower priority tasks on P_1 at random intervals. The average interval is same with the first workload. During the measurement, periodic interrupt requests are also raised on each processor, and the interrupt response times are measured. The interrupt period and the execution time of the interrupt handler are same with the first workload. The relation among tasks in this workload is illustrated in Figure 11.

6.2 Measurement Results

Figure 12 presents the 99.99%-reliable execution times of a system call that sends a message to a higher priority task under the first workload, when no interrupt request is serviced during the execution. The number of contending processors (including P_1) is changed from one (no interference) to nine (eight interfering tasks). With the proposed method, the execution time is nearly constant when the number of processors is larger than two. Its slight increase is due to the contentions for the local bus of P_1 and for the shared bus. Without local precedence scheme, the execution time is prolonged as the number of contending processors is increased. The execution time with the remote invocation method, which can not be bounded inherently, is prolonged more rapidly. This result demonstrates that our proposed method can practically satisfy the required property (A) in Section 3.3, but other methods cannot.

The execution time of a system call that sends a message to a private task is quite short because no inter-processor synchronization is necessary to execute the system call. Moreover, the number of contending processors has only a little influence on the execution

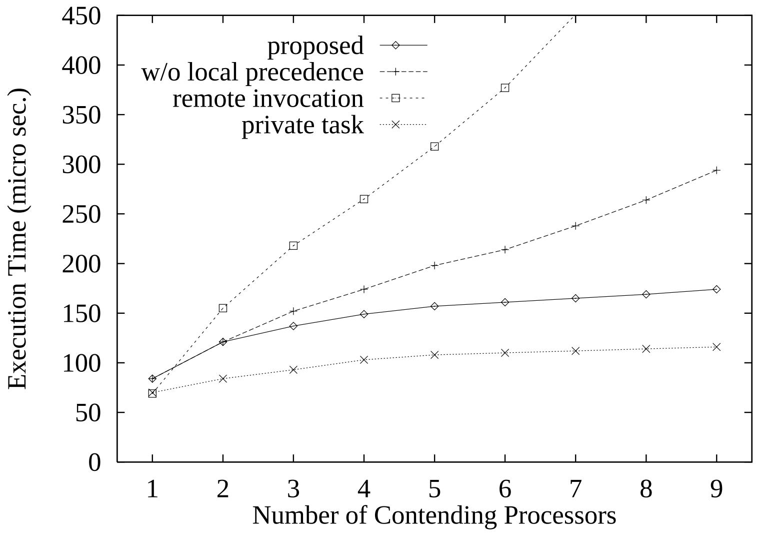


Figure 12: Execution Times of Local Operation

time. When the number of processors is one, the execution time is about 70 μs, which corresponds to the execution time of the system call in single processor systems. The execution time with the remote invocation method when the number of processors is one is almost same with this, because inter-processor synchronization is also unnecessary with the remote invocation method.

Figure 13 presents the 99.99%-reliable execution times of a system call that sends a message to a local task on another processor under the second workload, when no external interrupt request is serviced during the execution. The number of contending processors (including P_1) is changed from two (an interfering task on P_1) to nine (eight interfering tasks). The proposed method has worse performance than the other methods, because of the performance penalty imposed on non-local operations. This result demonstrates that each method satisfies the required property (B) in Section 3.3.

In order to show that our proposed method can satisfy the required property (C), we present the 99.99%-reliable interrupt response times on processor P_2 under the second workload in Figure 14. The number of contending processors is changed from two to nine. Under this workload, P_2 repeatedly acquires the lock guarding the TCB of P_1-local tasks. Unless a preemption scheme is adopted, the interrupt response time on P_2 includes the time that P_2 is waiting for the lock and is prolonged as the number of contending processors is increasing. With either preemption scheme, the interrupt latency becomes almost independent of the number of contending processors.

Finally, we demonstrate that our proposed method can satisfy the required property

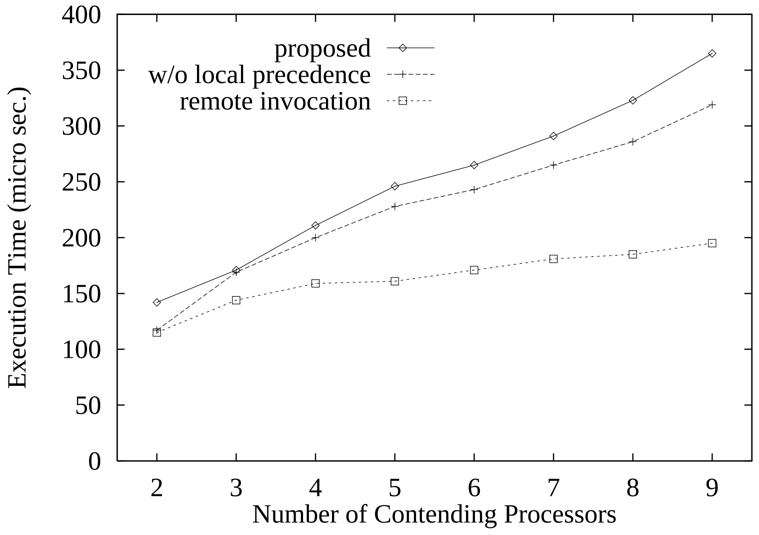


Figure 13: Execution Times of Remote Operation

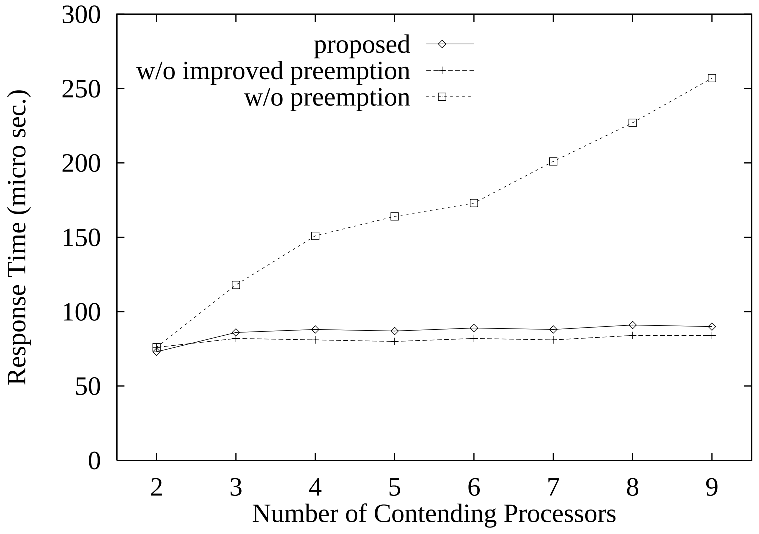


Figure 14: Interrupt Response Times

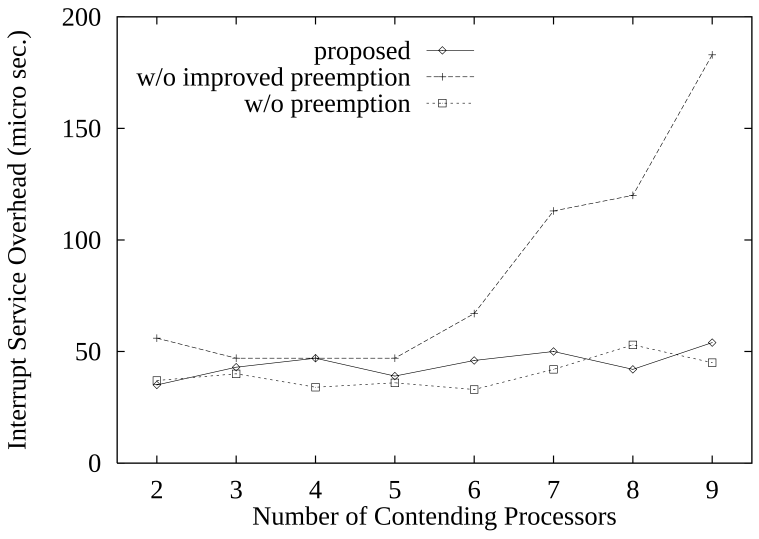


Figure 15: Interrupt Service Overheads

(D). Figure 15 presents the differences of the 99.99%-reliable execution times of a remote operation when an interrupt request is serviced during the execution and those when no interrupt request is serviced, which represent the measured interrupt service overheads, under the second workload. With the proposed method, the interrupt service overhead does not depend on the number of contending processors. With the simple preemption scheme, the interrupt service overhead becomes long as the number of contending processors is increased.

From these measurement results, we can say that the proposed method has advantage over other implementation methods. The four required properties of a scalable real-time kernel described in Section 3.3 are not satisfied in strict, because the assumptions on the underlying synchronization mechanism and hardware are not satisfied in our evaluation environments. However, we found that their effect is quite small and the four properties can be thought to be satisfied in practice, except for very hard read-time applications.

7 Difficulty To Be Solved

In this section, we discuss the implementation method of a scalable real-time kernel that satisfies the four required properties presented in Section 3.3 and that supports task-independent synchronization and communication objects, such as semaphores and eventflags. As the result, the difficulty for its realization is illustrated. Discussions in this section are also based on the assumptions on underlying inter-processor synchronization

```
acquire_lock(Lock_for_Objects);
deterimine which lock to acquire;
if (lock is necessary to be acquired) then
    acquire_lock(Lock_for_Tasks);
    execute the system call;
    release_lock(Lock_for_Tasks)
else
    execute the system call
end;
release_lock(Lock_for_Objects);
```

Figure 16: Acquiring Nested Spin Locks

and hardware presented in Section 4.3.

7.1 Necessity of Nested Spin Locks

When tasks whose control blocks are guarded by different lock units can wait on a synchronization object, the control block of the object should be included in a different lock unit with the TCBs as described in Section 2.4. A system call that operates on a task-independent synchronization object first acquires the lock guarding its control block. When a task that has been waiting on the object is released from the blocked state with the system call, or when the task that issues the system call begins to wait on the object, the system call also needs to acquire the lock guarding the TCB of the target task. Note here that which TCB is necessary to be accessed is determined after accessing the control block of the synchronization object. Consequently, the lock guarding the control block of the synchronization object must be acquired at first, and after accessing the control block, the TCB of the target task must be acquired. This kind of *nested locks* are the obstacle for satisfying the required properties of a scalable real-time kernel. Figure 16 illustrates a skeleton of a routine that executes a system call requiring nested spin locks.

On the other hand, when only the tasks included in a class can wait on a synchronization object, the control block of the object can be included in the same lock unit with the TCBs of the tasks. Therefore, this type of synchronization object can be realized with the method described in Section 4. Its typical example is the task-dependent mailbox[4] on which only its host task can wait. We have used this type of mailbox for the performance measurements in Section 6.

A private synchronization object can also be realized without nested spin locks. This is because a private object cannot be accessed from other processors, and because the

[4]The task-dependent mailbox function is defined in the version 2.0 of the μITRON specification, but not defined in the latest μITRON specification, μITRON3.0.

control block of the object need not be guarded with an inter-processor lock. Another important feature of a private synchronization object is that the maximum execution time of an operation on the object does not depend on the number of contending processors. The reason is as follows. When a task τ_1 on processor P_1 operates on a P_1-private synchronization object, the only inter-processor lock that τ_1 possibly needs to acquire is the lock guarding the TCBs of P_1-local tasks. This is because a P_1-private object can be waited on only by P_1-local tasks and P_1-private tasks. The maximum time to acquire the lock guarding the TCBs of P_1-local tasks can be bounded independently of the number of contending processors thanks to the local precedence scheme described in Section 4.1. As the result, the maximum execution time of an operation on a private synchronization object can be bounded independently of the number of contending processors. If tasks within a processor synchronize and communicate using private objects of the processor, the required property (A) in Section 3.3 can be satisfied.

7.2 Candidate Implementation Methods

Below, we try to satisfy the three other required properties (B), (C), and (D) presented in Section 3.3. The first method can satisfy the properties (B) and (C), but cannot satisfy (D). Though we propose the second method for satisfying the required property (D), it can not satisfy (B) instead.

The First Method

In order to make the required properties (B) and (C) compatible, when an interrupt is requested to a processor while it is waiting for a lock, the processor must suspend the spin-waiting and start servicing the interrupt request as discussed in Section 3.2. When the interrupt request occurs while a processor is waiting for the outer lock (the lock guarding the control blocks of synchronization objects), the same preemption scheme with that proposed in Section 4.2 can be applied straightforwardly.

The problem arises when the interrupt request occurs while a processor is waiting for the inner lock (the lock guarding the TCBs). In this case, in addition to suspend the spin-waiting for the inner lock, the processor must release the outer lock before servicing the interrupt request. Otherwise, the maximum duration that the processor holds the lock includes interrupt service times. The skeleton of the routine supporting preemption is presented in Figure 17. In this figure, the *acquire_lock* function is assumed to return false, when an interrupt in requested while waiting for the lock. After returning from the interrupt service, the processor must re-acquire the outer lock and re-execute the

```
retry:
    disable interrupts;
① if (¬acquire_lock(Lock_for_Objects)) then
        enable interrupts;
        interrupt requests are serviced here;
        goto retry
    end;
② deterimine which lock to acquire;
    if (lock is necessary to be acquired) then
      ③ if (¬acquire_lock(Lock_for_Tasks)) then
            release_lock(Lock_for_Objects);
            enable interrupts;
            interrupt requests are serviced here;
            goto retry
        end;
        execute the system call;
        release_lock(Lock_for_Tasks)
    else
        execute the system call
    end;
    release_lock(Lock_for_Objects);
    enable interrupts;
```

Figure 17: Acquiring Nested Spin Locks with Preemption

processings to determine which lock to be acquired (the routine between ② and ③ in Figure 17), because which lock to be acquired may be changed during the interrupt service. This re-execution overhead should be treated as included in the interrupt service overhead.

With this preemption scheme, the required properties (B) and (C) are satisfied. The methods to bound the maximum time to acquire nested spin locks with a linear order of the number of contending processors will be discussed in Section III.4.

However, the processor must re-execute the lock acquisition routine for the outer lock from the beginning after it finishes interrupt services. In other words, this method corresponds to the simple preemption scheme presented in Section 4.2. Therefore, the interrupt service overhead depends on the number of contending processors and the required property (D) cannot be satisfied with this method.

The Second Method

In order to satisfy the required property (D), Section 4.2 has proposed the improved preemption scheme, with which the processor returns to its original position in the waiting queue instead of enqueues itself at the tail of the queue. We try to apply this policy to this case.

After a processor returns from an interrupt service which is requested while waiting for the inner lock, the processor should be enqueued to the head of the waiting queue for the outer lock instead of the tail of it. With this preemption scheme, the interrupt service overhead can be bounded independently of the number of contending processors, and the required property (D) can be satisfied.

With this method, however, the property (B) cannot be met with the following reason. Suppose the case that a processor P_1 is holding the outer lock L on which two other processors P_2 and P_3 are waiting. When an interrupt is requested on P_1 while it is waiting for the inner lock, P_1 suspends waiting for the inner lock, passes the lock L to P_2, and starts the interrupt service. Assume that P_2 is waiting for the inner lock and is still holding L when P_1 returns from the interrupt service. In this case, P_1 enqueues itself at the head of the waiting queue, i.e. in front of P_3. If an interrupt request is raised on P_2 at this moment, it passes the lock L to P_1. Again, P_2 can return to the head of the waiting queue, i.e. in front of P_3. This process can continue permanently and violates the required property (B). More precisely, the maximum time until P_3 acquires L cannot be bounded without some assumptions on the occurrence of interrupt requests.

8 Summary

In this part, the required properties of a scalable real-time kernel for function-distributed multiprocessors have been summarized in four items, and its realization methods have been discussed. Before the discussions on a scalable real-time kernel, we have presented the overview of the TRON project, the ITRON specifications, and the ITRON-MP specification, which constitute the background of this study.

In Section 2, we have presented the basic real-time kernel model for function-distributed multiprocessors. We have also described the two implementation approaches of the model, the direct access method and the remote invocation method, and illustrated that the latter method is not suitable for real-time system. It is one of the reasons why we focus on shared-memory multiprocessors in this study. The granularity of inter-processor locks with the direct access method has also been discussed.

In a well-designed application system on a function-distributed multiprocessor architecture, many of the tasks can be processed without direct synchronizations or communications with other processors. Therefore, it is advantageous that the worst-case timing behavior of such tasks is determined independently of the other processors' activities and the number of contending processors. The timing behavior of interrupt handling should be also independent of the number of contending processors. In Section 3, we have summarized these requirements on a scalable real-time kernel in four properties and pointed out two problems to realize the properties.

In Section 4, we have proposed the solutions to the problems presented in the previous section. With the proposed implementation method, a multiprocessor real-time kernel that does not support task-independent synchronization and communication objects can be realized with satisfying the four required properties, on the assumption that underlying inter-processor synchronization mechanism and hardware architecture satisfy the required properties described in Section 4.3.

In Section 5, we have proposed a new kernel model in which tasks and task-independent synchronization and communication objects are classified into some classes with different characteristics. Tasks are classified into the local tasks, the private tasks, and the isolated tasks of each processor. Task-independent synchronization objects are also classified into the shared objects, the private objects, and the isolated objects. The accessibility of each class of kernel resources from each task class has been presented in a table.

In Section 6, the effectiveness of our proposals in Section 4 and 5 are demonstrated through performance evaluations. Though underlying inter-processor synchronization mechanism and hardware architecture do not satisfy the assumptions described in Section 4.3, the four required properties of a scalable real-time kernel are practically satisfied with our proposals. They cannot be satisfied at the same time with other methods.

Section 4 has focused on direct operations on tasks and has not considered task-independent synchronization and communication objects, such as semaphores and event-flags. Because tasks belonging to different classes can wait on a task-independent object, the control block of the object should be included in a different lock unit from the TCBs, and two lock units are necessary to be acquired one by one in some system calls. Section 7 has shown the difficulty to implement task-independent synchronization and communication objects while satisfying all of the required properties presented in Section 3.3.

With the kernel model proposed in this part, the asymmetry of the underlying architecture is directly reflected to the kernel interface. Here, a criticism is expected that this approach put a burden on the system designer. We consider that this criticism is inadequate with the follow reasons.

1. Under the current technologies of real-time computing, it is necessary for a system designer to be conscious of the underlying execution mechanism of the system in designing a hard real-time system with severe timing constraints. Therefore, it is not a good approach that the characteristics of underlying hardware architecture is

hidden with an operating system kernel.

2. When a technology is developed with which a system designer need not be conscious of the underlying hardware architecture in designing a hard real-time system, the technology should be incorporated to the tools supporting real-time system design such as schedulability analyzers and the CASE tools, and not to the real-time kernel.

Part III

Spin Lock Algorithms for Scalable Real-Time Kernels

1 A Brief Survey on Spin Lock Algorithms

An inter-processor *lock* is used for exclusive access to shared resources on shared-memory multiprocessors. When a processor accesses a shared resource, it must acquire the lock guarding the resource. When the lock is held by another processor, the processor must wait until the lock is released. In waiting for the lock, two approaches exist; busy-waiting and blocking.

A *spin lock* is the mechanism for realizing an inter-processor lock with busy-waiting approach. When the lock is held by another processor, the processor *spins* until the lock is released. Though spin-waiting wastes processor cycles, it is useful in two situations: when the execution time of the critical section is so short that the expected waiting time is shorter than the time to block and resume the task, and when there is no other work to do. In implementing a multiprocessor real-time kernel, spin locks are usually adopted because the execution time of each critical section is very short.

1.1 Hardware Primitives and Spin Locks

Spin lock algorithms for shared-memory multiprocessors have been intensively studied under various hardware environments.

The first spin lock algorithm was proposed by Dijkstra in 1965 [10], which assumes that the hardware supports only (atomic) read and (atomic) write operations. After some proposals of its improvements [30, 11, 32], an efficient algorithm in the absence of contention was proposed under the same hardware assumption quite recently [33]. More recently, the algorithm is improved with the timing-based approach, in which the relative execution speed of each processor is assumed to be bounded at any moment [35, 2, 41].

Because these algorithms that use only (atomic) read and (atomic) write operations have quite large overhead, however, most modern shared-memory multiprocessor architectures provide hardware support for exclusive accesses to shared resources. The most popular approach is to support atomic read-modify-write operations on a single word of shared memory. Another approach is to support spin locks with hardware [13, 34].

In this study, we assume that atomic read-modify-write operations on a single word (or aligned contiguous words) of shared memory are supported with hardware. Typical examples of the operations are test_and_set, fetch_and_store (swap), fetch_and_add, and compare_and_swap.

Among the operations, the compare_and_swap operation is most powerful and is supported by many microprocessors. With a compare_and_swap operation and a retry loop, the other read-modify-write operations can be emulated. Compare_and_swap is also universal in the sense that a wait-free implementation of any concurrent data object is possible with the operation, while the other operations listed above are not [16, 17].

Several recent high-performance microprocessors support load_linked (or load_and_reserve) and store_conditional operations [28, 61, 20]. The load_linked operation reads the value of a shared variable to a register. A subsequent store_conditional operation to the shared variable changes its value only if no other processor has modified the variable since the last load_linked operation. The store_conditional operation returns true if it succeeds to store a new value to the shared variable.

With a pair of load_linked and store_conditional operations and a retry loop, the other read-modify-write operations including compare_and_swap can be emulated [68]. These operations are also universal in principle [18]. In practice, there are some differences between the pair of load_linked and store_conditional operations and the compare_and_swap operation [42]. Because all compare_and_swap operations used in this dissertation can be replaced with these operations, the results of this study are also applicable to the processors supporting only load_linked and store_conditional.

1.2 Notations Used in Pseudo-Codes

In the following sections, several pseudo-codes of spin lock algorithms are presented. In these pseudo-codes, the following notations are used.

In presenting the pseudo-codes, we use our original syntax which is a mixture of Modula-2 programming language [88] and C programming language. We also use some non-ASCII characters, such as "$\rightarrow$", "$\neg$", and "$\neq$", for readability. Lines beginning with "//" are comments, which we borrow from C++ programming language.

The keyword *shared* indicates that only one instance of the variable is allocated and shared in the system. Other variables are allocated for each processor. The binary operator *and* is assumed to be the conditional-and operation, i.e. the right hand side of the *and* operator is evaluated only if its left hand side is true. When priorities are represented with numbers, we assume that a larger value represents a higher priority. Therefore, if *prio1* $>$ *prio2* is satisfied, *prio1* represents a higher priority than *prio2*.

Fetch_and_store reads the shared variable addressed by the first parameter (which must be a pointer), returns the contents of the variable as its value, and atomically writes the second parameter to the variable. *Compare_and_swap* is a Boolean function with three parameters. It first reads the shared variable addressed by the first parameter and compares its contents with the second parameter. If they are equal, the function writes the third parameter to the variable atomically and returns true. Otherwise, it returns false.[1]

[1]The compare_and_swap instructions of many microprocessors store the contents of the memory to the

1.3 Basic Spin Lock Algorithms

On the assumption that atomic read-modify-write operations on a single word (or aligned contiguous words) of shared memory are supported with hardware, we can classify major basic spin lock algorithms into following four categories.[2] In the following, a *bounded spin lock* is defined to be a spin lock algorithm with which the maximum time to acquire a lock can be bounded. Obviously, a *FIFO-ordered spin lock* is a class of bounded spin locks.

Test&Set Locks

Each processor trying to acquire a lock repeatedly executes a test_and_set operation on a shared Boolean variable indicating the lock status. When it sets the variable, it succeeds to acquire the lock. It releases the lock by clearing the variable. There are many variations of this algorithm in how each processor retries to execute a test_and_set operation [3].

Because the time until a processor can acquire a lock cannot be bounded with test&set locks, they are not appropriate for real-time systems.

Ticket Locks

Two shared counters are used in ticket locks: a request counter and a release counter. A processor trying to acquire a lock increments its request counter using a fetch_and_add operation and obtains the old value of the counter, which indicates its turn to acquire the lock. Then, it waits until the release counter is equal to the value. To release the lock, the processor increments the release counter. There are some variations in how each processor retries to read the release counter [3]. Obviously, processors can acquire a lock in a FIFO order with ticket locks.

FIFO-Ordered Queueing Locks

There are two subclasses of this category of algorithms: array-based FIFO-ordered queueing locks and list-based FIFO-ordered queueing locks.

In array-based FIFO-ordered queueing locks, a processor trying to acquire a lock is linked at the tail of an array-based waiting queue for the lock. If the waiting queue is empty, the processor can acquire the lock at once. Otherwise, the processor spins on a memory location in the array-based queue on which only the processor spins.

third parameter in this case. This facility is not used in this study.

[2]On the same assumption, Mellor-Crummey and Scott have classified spin lock algorithms into a bit different four categories [38].

The processor trying to release the lock passes the lock to the next processor in the waiting queue. If there are no other processors in the queue, the processor makes the waiting queue empty.

An algorithm using the fetch_and_add operation [3] and another using the fetch_and_store operation [14] have been proposed. On cache-coherent multi-processors, the number of shared-bus transactions is bounded independently of the number of processors with these algorithms, and the contention problem on the shared bus (or interconnection network) can be resolved.

In list-based FIFO-ordered queueing locks, a processor trying to acquire a lock is linked at the tail of a list-based waiting queue. Two famous algorithms in this class is the MCS lock algorithm [40, 38], which uses the fetch_and_store operation and the compare_and_swap operation, and the Craig's FIFO-ordered queueing lock algorithm [9, 8], which uses only the fetch_and_store operation. Another advantage of the Craig's algorithm is that the required memory space is as small as $O(L + P)$, where L is the number of locks and P is the number of processors, even when spin locks are nested. With the MCS lock, this becomes $O(L + P * D)$, where D is the maximum number of locks that a processor must acquire at the same time.

Other Bounded Locks

With the spin lock algorithms proposed by Burns [5], the maximum time to acquire a lock can be bounded, but the lock is not passed in a FIFO order. There is also a trial to improve the efficiency of the algorithm [44].

Because the MCS lock algorithm, the representative FIFO-ordered queueing lock algorithm, has some good features and is very simple, many extensions of the algorithm are proposed [39]. We also propose some extensions in the following sections.

Pseudo-code for the MCS lock appears in Figure 18, and its behavior is illustrated in Figure 19. The queue node of the lock holder (the processor that holds the lock) is at the head of the waiting queue for the lock, and the queue nodes of the processors waiting for the lock are linked to the queue in a FIFO order. *Lock* points to the last node of the queue. When a processor begins waiting for the lock, it enqueues its queue node at the tail of the queue. Precisely, it initialize its queue node at first (Figure 19 (a)), and swings the *Lock* to its queue node with a fetch_and_store operation (Figure 19 (b)). After the processor rewrites the *next* field of its predecessor's queue node (Figure 19 (c)), it begins spinning on the *locked* filed of its queue node. When the lock holder releases the lock, it passes the lock to the next processor in the queue by assigning *Released* to the *locked* field of its queue node (Figure 19 (d)).

```
  type Node = record
      next: pointer to Node;
      locked: (Released, Locked)
  end;
  type Lock = pointer to Node;

  shared var L: Lock;
  // L is initialized to NULL.

  var I: Node;
  var pred: pointer to Node;

  // try to acquire the lock L.
  I.next := NULL;
  // enqueue myself.
  pred := fetch_and_store(&L, &I);
  if pred ≠ NULL then
      // when the queue is not empty.
      I.locked := Locked;
      pred→next := &I;
      // spin until the lock is passed.
      repeat until I.locked = Released
  end;
  //
  // critical section.
  //
  // try to release the lock L.
  if I.next = NULL then
      if compare_and_swap(&L, &I, NULL) then
          // the queue becomes empty.
          goto exit
      end;
      repeat until I.next ≠ NULL
  end;
  I.next→locked := Released;
exit:
```

Figure 18: The MCS Lock

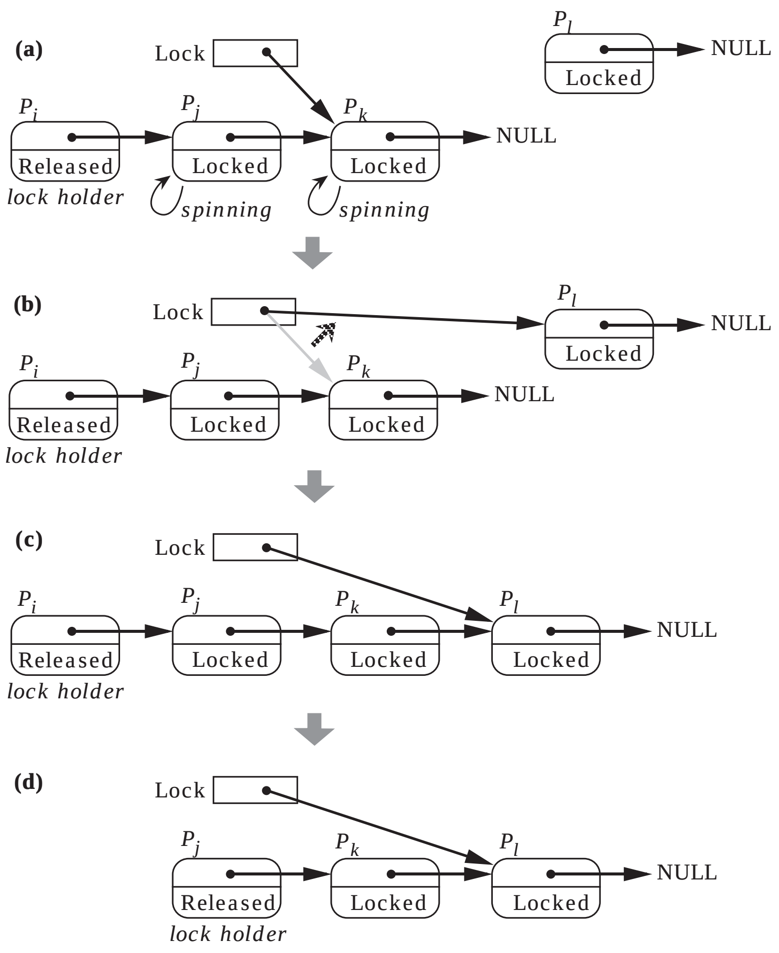


Figure 19: Behavior of the MCS Lock

With the MCS lock, if the queue node of each processor (variable *I*) is located on its locally-accessible shared memory, the number of shared-bus (or interconnection) transactions is bounded even on multiprocessors without a coherent cache. A simple proof of its correctness is presented in [27] (The original proof in [38] is quite complicated).

1.4 Priority-Ordered Spin Locks

It is often the case with a multiprocessor real-time system that a spin lock is desirable to pass the lock in a priority order. To meet this requirement, some *priority-ordered spin lock* algorithms, in which processors acquire a lock in the order of their priorities, have been proposed.[3]

Markatos has extended the MCS lock to realize a priority-ordered spin lock [36]. The extended algorithm also uses both fetch_and_store and compare_and_swap operations.

[3] A strict definition of a priority-ordered spin lock is appeared in [36].

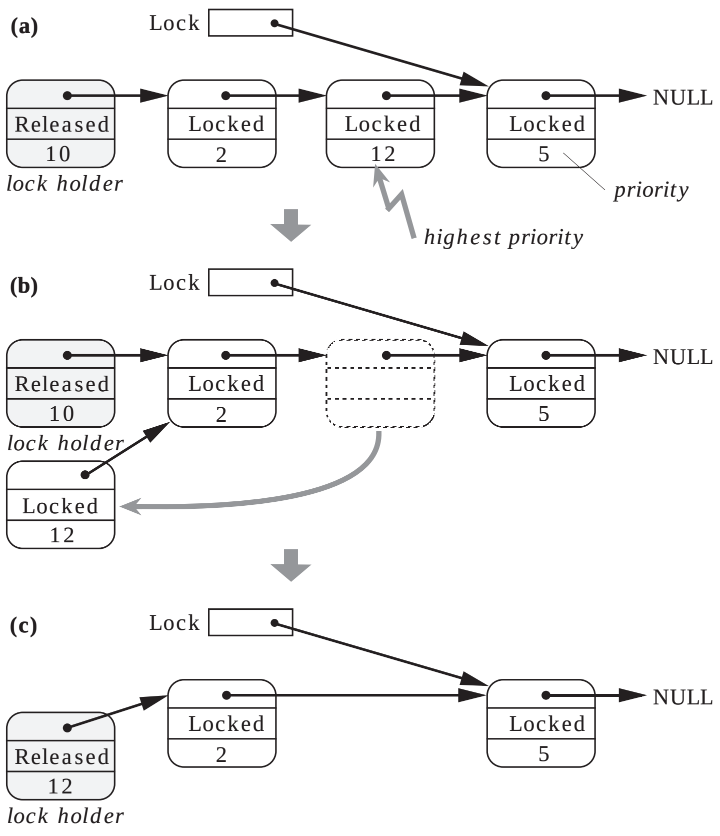


Figure 20: Behavior of the Markatos' Lock

With the Markatos' algorithm, processors trying to acquire a lock are linked to the waiting queue in a FIFO order. The processor trying to release the lock searches for the highest priority processor in the waiting queue (Figure 20 (a)), moves it to the head of the queue (Figure 20 (b)), and passes the lock to it (Figure 20 (c)). Therefore, the maximum execution time of the lock release routine depends on the number of processors in the system.

The original algorithm proposed by Markatos adopts a double-linked queue structure for the waiting queue. We found that a single-linked queue structure is sufficient to implement the algorithm. Pseudo-code for the single-linked queue version of the Markatos' algorithm is presented in Figure 21 and 22.

Though there is a non-local spinning (marked with #) in this algorithm, it is limited to a very short period after another processor writes the pointer to its queue node to **L** (a successful execution of the fetch_and_store operation marked with ①) and until it writes non-*NULL* value to the **next** field of its predecessor (marked with ②), and its effect is very limited.

Craig has also proposed priority-ordered versions of queueing spin locks that require

```
    type Node = record
        next: pointer to Node;
        locked: (Released, Locked);
        prio: integer
    end;
    type Lock = pointer to Node;

    shared var L: Lock;
    // L is initialized to NULL.

    procedure move_to_top(lock: pointer to Lock,
                          entry, pred, oldtop: pointer to Node);
    // move entry to the top of the waiting queue of lock.
    // pred is the predecessor of entry.
    // oldtop is the top of the queue before the move.
        var succ: pointer to Node;
    begin
        succ := entry→next;
        if succ = NULL then
            pred→next := NULL;
            if compare_and_swap(lock, entry, pred) then
                entry→next := oldtop;
                return
            end;
#           repeat succ := entry→next until succ ≠ NULL
        end;
        pred→next := succ;
        entry→next := oldtop
    end;
```

Figure 21: The Markatos' Algorithm (Definition Part)

only the fetch_and_store operation [9, 8]. Similarly to the Markatos' algorithm, processors trying to acquire a lock are linked to the waiting queue in a FIFO order. The processor trying to release the lock finds the highest priority processor and passes the lock to it.

With the PR-lock algorithm on the other hand, processors trying to acquire a lock are enqueued to the waiting queue in a priority order, and the processor trying to release the lock passes the lock to the first processor in the waiting queue [26]. Therefore, the maximum execution time of the lock acquisition routine depends on the number of processors in the system. This algorithm has an advantage over the previous algorithms that enqueueing operations, which are the most time-consuming part of the algorithm, can be done in parallel.

We are also proposing a priority-ordered spin lock named the bubble lock [57], which adopts another scheme for realizing priority-ordered spin locks.

```
   var I: Node;
   var top, entry, pred: pointer to Node;
   var hentry, hpred: pointer to Node;

   // try to acquire the lock $L$.
   I.next := NULL;
   // enqueue myself.
① pred := fetch_and_store(&L, &I);
   if pred $\neq$ NULL then
       // when the queue is not empty.
       I.locked := Locked;
       I.prio := my_priority;
     ② pred$\rightarrow$next := &I;
       repeat until I.locked = Released
   end;
   //
   // critical section.
   //
   // try to release the lock $L$.
   top := I.next;
   if top = NULL then
       if compare_and_swap(&L, &I, NULL) then
           // the queue becomes empty.
           goto exit
       end;
       repeat top := I.next until top $\neq$ NULL
   end;
   // search for the higest priority processor.
   hentry := top;
   pred := top;
   entry := pred$\rightarrow$next;
   while entry $\neq$ NULL do
       if (entry$\rightarrow$prio $>$ hentry$\rightarrow$prio) then
           // when entry has a higher priority that hentry.
           hentry := entry;
           hpred := pred
       end;
       pred := entry;
       entry := pred$\rightarrow$next
   end;
   // now, hentry is the higest priority processor.
   if hentry $\neq$ top then
       move_to_top(&L, hentry, hpred, top)
   end;
   hentry$\rightarrow$locked := Released;
 exit:
```

Figure 22: The Markatos' Algorithm (Main Part)

2 Bounded Spin Lock with Preemption

In this section, we propose two algorithms of queueing *spin lock with preemption* and demonstrate their effectiveness through performance measurements. The necessity of spin lock with preemption is pointed out in Section II.4.2 and is described in more detail in this section.

In Section 2.1, the difficulty to satisfy two important requirements on scalable real-time systems, predictable inter-processor synchronization and constant interrupt response, at the same time. Section 2.2 describes that the adoption of a preemption scheme to spin locks can solve the difficulty. Two queueing spin lock algorithms supporting different preemption schemes are presented in Section 2.3 and 2.4, and their effectiveness is evaluated through performance measurement in Section 2.5. Finally, in Section 2.6, we point out the necessity to support two preemption scheme at the same time, and describe a combined algorithm.

2.1 Spin Locks and Interrupt Latency

When a spin lock is used for a real-time system, the maximum times to acquire and release a lock should be bounded. In order to bound the time until a processor acquires a lock, the maximum duration that each processor holds the lock must be bounded, in addition to bound the number of contending processors that the processor waits for, which can be satisfied with bounded spin lock algorithms.

In order to bound the maximum duration that a processor holds the lock, the service time of interrupt requests should be considered. In function-distributed multiprocessor systems, interrupt services for external devices are requested for each processor. When multiple devices are connected to a processor, interrupt requests from them are usually asynchronous and the maximum time to service all of them becomes very long or even unbounded. Consequently, in order to give a practical upper bound on the duration that a processor holds a lock, interrupt services should be inhibited for that duration.

On the other hand, fast response to external events is also an important feature for real-time systems. Because external events are notified to each processor as interrupt requests as mentioned above, interrupt mask times on each processor should be minimized to realize a system with fast response. Particularly, when the scalability of the system is an important issue, the maximum interrupt mask time should be given independently of the number of processors in the system.

Here a problem arises in deciding whether interrupts should be disabled first or an inter-processor lock should be acquired first. Figure 23 presents a method in which

```
acquire_lock();
disable_interrupts;
//
// critical section.
//
enable_interrupts;
release_lock();
```

Figure 23: Acquiring a Lock Precedes Disabling Interrupts

```
disable_interrupts;
acquire_lock();
//
// critical section.
//
release_lock();
enable_interrupts;
```

Figure 24: Disabling Interrupts Precedes Acquiring a Lock

acquiring a lock precedes disabling interrupts. With this method, interrupts are serviced while the processor holds the lock, and the condition that interrupt services should be inhibited while a processor holds a lock is not satisfied. Figure 24 presents another method where acquiring a lock follows disabling interrupts. With this method, the interrupt mask time includes the time to acquire a lock and its upper bound heavily depends on the number of processors.

2.2 Spin Locks with Preemption

In order to solve the problem described above, interrupt services must not be inhibited while a processor waits for an inter-processor lock and must be kept inhibited once the processor acquires the lock. One of the methods to realize this principle is the following. While a processor is waiting for a lock, it repeatedly probes interrupt requests. When interrupt requests are detected, it suspends waiting for the lock and services the requests.

The test&set locks can be extended easily with this method as presented in Figure 25 [58]. The algorithm is not suitable for real-time systems, however, because the time until a processor acquires a lock cannot be bounded with it. The ticket locks and the FIFO-ordered queueing locks, on the other hand, cannot be extended similarly.

In the following sections, we present two spin lock algorithms with which a processor can service interrupts with short latency while satisfying the principle described above. The algorithms are based on the MCS lock described in Section 1.3.

```
type Lock = (Released, Locked);

shared var L: Lock;
// L is initialized to Released.

disable_interrupts;
while test_and_set(L) = Locked do
    if interrupt_requested then
        enable_interrupts;
        // interrupt service.
        disable_interrupts
    else
        delay
    end
end;
//
// critical section.
//
L := Released;
enable_interrupts;
```

Figure 25: The Test&Set Lock with Preemption

2.3 Queueing Lock with Simple Preemption Scheme

In all the spin lock algorithms that can give an upper bound on the time until a processor acquires a lock, a processor modifies some shared variable and reserves its turn to acquire the lock when it begins waiting for the lock. If the processor simply branches to an interrupt service routine while waiting for the lock, it cannot begin the execution of the critical section immediately when the lock is passed to the processor, and makes the contending processors wait wastefully until the interrupt service is finished. Therefore, when a processor begins to service interrupts while waiting for a lock, it must inform others that it is servicing interrupt requests by modifying some shared variable. The processor trying to release the lock checks if the succeeding processor is servicing interrupts. If the succeeding one is found to be servicing interrupts, its turn to acquire the lock is canceled or deferred, and the lock is passed to the next processor in line.

Pseudo-code of our first algorithm, which is an extension of the MCS lock to enable interrupt services while waiting for a lock, appears in Figure 26 and 27. In this algorithm, a processor informs others that it is servicing interrupts by assigning the value *Preempted* to the *locked* field of its queue node (i.e. *I.locked*).

If the processor P_0 that is trying to release a lock finds that the succeeding processor P_1 is servicing interrupts, P_0 dequeues P_1 from the waiting queue and tries to pass the lock to the successor of P_1. During this process, a transient status occurs in which P_1's

```
  type Node = record
      next: pointer to Node;
      locked: (Released, Locked, Preempted, Canceled)
  end;
  type Lock = pointer to Node;

  shared var L: Lock;
  // L is initialized to NULL.

  var I: Node;
  var pred, succ, sn: pointer to Node;

  // try to acquire the lock L.
retry:
  I.next := NULL;
  disable_interrupts;
  // enqueue myself.
  pred := fetch_and_store(&L, &I);
  if pred ≠ NULL then
      // when the queue is not empty.
      I.locked := Locked;
      pred→next := &I;
      while (I.locked ≠ Released) do
          if interrupt_requested and
                  compare_and_swap(&(I.locked), Locked, Preempted) then
              enable_interrupts;
              // interrupt service.
              disable_interrupts;
              if ¬compare_and_swap(&(I.locked), Preempted, Locked) then
                  enable_interrupts;
                  repeat while I.locked ≠ Released;
                  goto retry
              end
          end
      end
  end;
  //
  // critical section.
  //
```

Figure 26: The Queueing Lock with Simple Preemption Scheme (Part 1)

```
  //
  // critical section.
  //
  // try to release the lock $L$.
  succ := I.next;
  **if** succ = NULL **then**
      **if** compare_and_swap(&L, &I, NULL) **then**
          // the queue becomes empty.
          **goto** exit
      **end**;
      **repeat** succ := I.next **until** succ $\neq$ NULL
  **end**;
  // try to pass the lock to the successor.
  **while** $\neg$compare_and_swap(&(succ$\rightarrow$locked), Locked, Released) **do**
      // when the successor is servicing interrupts.
      **if** compare_and_swap(&(succ$\rightarrow$locked), Preempted, Canceled) **then**
          // dequeue the successor from the waiting queue.
          sn := succ$\rightarrow$next;
          **if** sn = NULL **then**
              **if** compare_and_swap(&L, succ, NULL) **then**
                  // the queue becomes empty.
                  succ$\rightarrow$locked := Released;
                  **goto** exit
              **end**;
              **repeat** sn := succ$\rightarrow$next **until** sn $\neq$ NULL
          **end**;
          succ$\rightarrow$locked := Released;
          succ := sn
      **end**
  **end**;
exit:
  *enable_interrupts*;
```

Figure 27: The Queueing Lock with Simple Preemption Scheme (Part 2)

queue node has been dequeued but the node area must not be reused because the value of its *next* field is necessary. P_0 informs P_1 of this transient status by assigning the value *Canceled* to the *locked* field of P_1's queue node. When the node becomes reusable, P_0 informs P_1 of it by changing the *locked* field to *Released*. When P_0 finds that all the waiting processors are servicing interrupts, P_0 makes the waiting queue empty.

When the processor that has branched to an interrupt service routine while waiting for a lock finishes the interrupt service, it reads the *locked* field of its queue node and checks whether it has been dequeued (during the interrupt service) or not. If it has been dequeued, it re-executes the lock acquisition routine from the beginning after waiting until its queue node area becomes reusable. Otherwise, it recovers its *locked* field to the value *Locked* and resumes waiting for the lock.

With this algorithm, a processor waiting for a lock can acquire the lock in the order of the waiting queue if no interrupt request is raised on the processor. In releasing a lock, the algorithm also gives an upper bound on the number of search loops for identifying to which processor the releasing processor should pass the lock, unless interrupt services start and finish repeatedly on the waiting processors.[4] As interrupt services are inhibited while a processor holds a lock, no interrupt service time is included in the lock holding time. Consequently, both the time until a processor acquires a lock and the time until it releases the lock can be bounded with this algorithm under the above conditions.

Because a processor repeatedly probes interrupt requests while waiting for a lock, the upper bound of the interrupt mask time in the lock acquisition routine can be determined independently of the number of processors. On the other hand, the interrupt mask time in the lock release routine depends on the number of processors in strict, because the number of search loops for identifying the processor to which to pass the lock depends on the number of processors. However, the problem is not severe in practice, because the processing time of one loop is very short.

The proofs of the important features of this algorithm, mutual exclusion and deadlock freedom when a certain condition is laid on interrupt occurrence, are presented in Appendix B.

Wisniewski et al. have proposed a similar algorithm with ours from a different motivation [89].[5] The algorithm by Craig can also support the same preemption scheme.

With this algorithm, when a processor services interrupts while waiting for a lock and is dequeued from the waiting queue, the processor must re-execute the lock acquisition routine from the *beginning*. Because the processor enqueues itself at the end of the waiting queue, the maximum number of critical sections executed by other processors that the processor must wait for is increased. When the schedulability of the system is analyzed, this re-execution overhead should be added to the interrupt service time and should be included in the interrupt service overhead described in Section II.3.2.

[4]A processor can be visited twice in the search loops in the following case. Immediately after the processor is dequeued from the waiting queue, it finishes the interrupt service and links itself to the end of the queue. If this case repeatedly occurs until the processor to which to pass the lock is identified, the number of the loops cannot be not bounded. This case rarely occurs. But, when this problem cannot be ignored (when the number of processors is large and when interrupts are requested frequently, in general), the algorithm should be modified so that the assignment of *Released* to the *locked* field of dequeued processors is delayed until the processor to which to pass the lock is determined.

[5]Their algorithm has a problem that the transient status in which a queue node is not reusable is not considered, thus the algorithm possibly falls into a deadlock. We have pointed out the problem to them, and they have acknowledged it [31].

2.4 Queueing Lock with Improved Preemption Scheme

With our first algorithm, the interrupt service overhead depends on the number of contending processors, because a processor possibly has to re-execute the lock acquisition routine from the beginning after it services an interrupt request. This is problematic when the algorithm is used for the implementation of a scalable real-time kernel as described in Section II.4.2.

In order to solve this problem, we propose an improved preemption scheme which avoids dequeueing a processor from the waiting queue while servicing interrupts. Specifically, the processor P_0 trying to release a lock searches for the first processor P_2 that is not servicing interrupts in the waiting queue, moves P_2 to the top of the queue (with the same method used in the Markatos' priority-ordered queueing spin lock), and passes the lock to P_2. With this algorithm, when a processor finishes interrupt services, it resumes waiting for the lock in its original position. Therefore, the interrupt service overhead, which must be added to the interrupt service time in schedulability analysis, is minimized.

When all processors in the waiting queue are servicing interrupts, the difficulty occurs that there is no processor to which to pass the lock and that the waiting queue should not be made empty. To handle this situation, a new flag variable called the global lock flag is introduced. The global lock flag indicates that the lock is released but that the waiting queue is not empty. If the processor trying to release the lock finds that all processors in the queue are servicing interrupts, it sets the global lock flag. A processor returning from interrupt services tries to acquire the global lock with the same method with test&set locks. If it succeeds in acquiring the global lock, it moves itself to the top of the waiting queue. Because the processor needs to know the top processor in the queue to move itself to the top, the processor releasing the global lock must pass the information in some shared variable. It is also necessary for a processor to check the global lock flag once, after it links itself at the end of the queue, because it is possible that all the processors in the queue be servicing interrupts and the global lock be set.

Pseudo-code for the improved algorithm appears in Figure 28, 29, and 30. In this pseudo-code, a double-linked queue structure is adopted because a processor needs to know is predecessor when it succeeds to acquire the global lock. The *glock* field of L serves both as the global lock flag and as the variable to pass the top processor of the waiting queue. Exponential backoff scheme is adopted to get the global lock in this code to reduce the number of shared-bus transactions. Two constant parameters α and β should be tuned for each target hardware and application.

With this preemption scheme, a transient status also occurs during the lock release

```
type Node = record
    next: pointer to Node;
    prev: pointer to Node;
    locked: (Released, Locked, Preempted, Dequeueing)
end;
type Lock = record
    last: pointer to Node;
    glock: pointer to Node
end;

shared var L: Lock;
// L.last and L.glock are initialized to NULL.

procedure move_to_top(lock: pointer to Lock,
                      entry, pred, oldtop: pointer to Node);
// move entry to the top of the waiting queue of lock.
// pred is the predecessor of entry.
// oldtop is the top of the queue before the move.
    var succ: pointer to Node;
begin
    succ := entry→next;
    if succ = NULL then
        // when succ is at the tail of the waiting queue.
        pred→next := NULL;
        if compare_and_swap(&(lock→last), entry, pred) then
            entry→next := oldtop;
            return
        end;
        repeat succ := entry→next until succ ≠ NULL
    end;
    pred→next := succ;
    succ→prev := pred;
    entry→next := oldtop
end;
```

Figure 28: The Queueing Lock with Improved Preemption Scheme (Part 1)

```
  var I: Node;
  var pred, succ, top: pointer to Node;
  var interval, i: integer;

  // try to acquire the lock $L$.
  I.next := NULL;
  disable_interrupts;
  // enqueue myself.
  pred := fetch_and_store(&(L.last), &I);
  if pred = NULL then
      goto acquired
  end;
  // when the queue is not empty.
  I.prev := pred;
  I.locked := Locked;
  pred→next := &I;
  i := 1;                          // check the global lock once.
  interval := $\infty$;                  // never expires.
  while (I.locked $\neq$ Released) do
      if interrupt_requested and
              compare_and_swap(&(I.locked), Locked, Preempted) then
          enable_interrupts;
          // interrupt service.
          disable_interrupts;
          I.locked := Locked;
          i := 1;
          interval := $\alpha$
      end;
      i := i – 1;
      if i = 0 then
          // check the global lock and try to acquire it if it is set.
          top := L.glock;
          if top $\neq$ NULL and compare_and_swap(&(L.glock), top, NULL) then
              // when succeed to acquire the global lock.
              if top $\neq$ &I then
                  move_to_top(&L, &I, I.prev, top);
              end;
              I.locked := Released
              goto acquired
          end;
          i := interval;
          interval := interval $\times$ $\beta$
      end
  end;
acquired:
  //
  // critical section.
  //
```

Figure 29: The Queueing Lock with Improved Preemption Scheme (Part 2)

```
  //
  // critical section.
  //
  // try to release the lock L.
  succ := I.next;
  **if** succ = NULL **then**
      **if** compare_and_swap(&(L.last), &I, NULL) **then**
          // the queue becomes empty.
          **goto** exit
      **end**;
      **repeat** succ := I.next **until** succ $\neq$ NULL
  **end**;
  // try to pass the lock to the successor.
  **if** compare_and_swap(&(succ$\rightarrow$locked), Locked, Released) **then**
      **goto** exit
  **end**;
  top := succ;
  **repeat**
      pred := succ;
      succ := pred$\rightarrow$next;
      **if** succ = NULL **then**
          // set the global lock.
          L.glock := top;
          // check if pred is really the last processor.
          **if** L.last = pred **then**
              **goto** exit
          **end**;
          // try to withdraw the global lock.
          **if** $\neg$compare_and_swap(&(L.glock), top, NULL) **then**
              **goto** exit
          **end**;
          **repeat** succ := pred$\rightarrow$next **until** succ $\neq$ NULL
      **end**;
  **until** compare_and_swap(&(succ$\rightarrow$locked), Locked, Dequeueing);
  // now, the lock is passed to succ.
  move_to_top(&L, succ, pred, top);
  succ$\rightarrow$locked := Released;
exit:
  enable_interrupts;
```

Figure 30: The Queueing Lock with Improved Preemption Scheme (Part 3)

```
for i := 1 to NoLoop do
 ① acquire_lock_and_disable_interrupts;
    //
    // critical section.
    //
    release_lock;
 ② enable_interrupts;
    random_delay
end;
```

Figure 31: Measurement Program Skeleton

process. The time window is after the processor P_0 trying to release determines to which processor to pass the lock (we denote the processor as P_2), and before P_0 passes the lock by assigning *Released* to the *locked* field of P_2's queue node. When an interrupt is requested on P_2 during this time window, the interrupt request should not be serviced. Otherwise, the lock may be passed to P_2 while it is servicing the interrupt. In this time window, P_2's queue node should not be reused either. In our algorithm, P_0 informs P_2 of this time window by assigning *Dequeueing* to the *locked* field of P_2.

2.5 Performance Evaluation

The effectiveness of the two spin lock algorithms presented in the previous sections (called QL/P1 and QL/P2, respectively, below) are examined through performance measurement. The performance of the algorithms is compared with the MCS lock without inhibiting interrupts (QL/ei), the method presented in Figure 24 with the MCS lock (QL/di), and the test&set lock with preemption presented in Figure 25 (T&S/P). In T&S/P, the intervals between successive test_and_set operations (*delay* in Figure 25) are made constant (the constant backoff scheme), because it is usually better than the exponential backoff scheme in real-time systems.

Measurement Method

Each processor executes the code presented in Figure 31 while periodic interrupt requests are raised on the processor. The execution time of a critical region (the region between ① and ② in Figure 31) is measured for each execution, and its distributions when the processor services no interrupt request during the region and when it services an interrupt are collected. The interrupt latency is also measured for each interrupt service and its distribution is obtained.

Inside the critical section, a processor accesses the shared bus some number of times for making the effect of shared-bus contention explicit and waits for a while using an empty loop. Without spin locks, the execution time of the critical region is about 40 μs including some overhead for measuring the execution time of the region. In order to change timing conditions, each processor waits for a random time before it re-enters the critical region (*random_delay* in Figure 31). The average time of the random delay is about 40 μs including some overhead for recording the execution time of the critical region.

Empty loops are also included in the interrupt handler in addition to the routine for the measurement of the interrupt latency time. The total execution time of the interrupt handler is about 80 μs. The period of interrupt requests is about 5 ms. The exact length of the period is varied in 0–2% for each processor in order that the timing of interrupt requests for each processor should not be synchronized. Other interrupt requests are masked during the measurement.

Evaluation Results

Figure 32 presents the 99.99%-reliable execution times of the critical region (when no interrupt is serviced on the processor during the region) as the number of processors is increased from one to eight. With QL/P1 and QL/P2, the execution time of the critical region increases linearly with the number of processors, and the algorithms are found to be scalable. QL/ei exhibits poorer performance because processors service interrupt requests during the critical region. In Figure 33, the relation between the execution time of the interrupt handler and that of the critical region is presented, when four processors are executing spin locks. As the execution time of the interrupt handler becomes longer, the performance of QL/ei becomes even worse (Figure 33). With T&S/P, the execution time rapidly increases when the number of processor becomes large, and the algorithm does not scale well.

In Figure 34, the interrupt latency time is almost independent of the number of processors with QL/P1 and QL/P2. With QL/di on the contrary, the interrupt latency becomes long as the number of processors increases. With T&S/P, the interrupt latency slowly increases because the execution time of the code inside the critical section becomes longer due to the effect of shared-bus contention.

From these observations, it is demonstrated that QL/P1 and QL/P2 can give a practical upper bound on the time to acquire and release an interprocessor lock while achieving fast response to interrupt requests. The other algorithms cannot satisfy these two requirements at the same time.

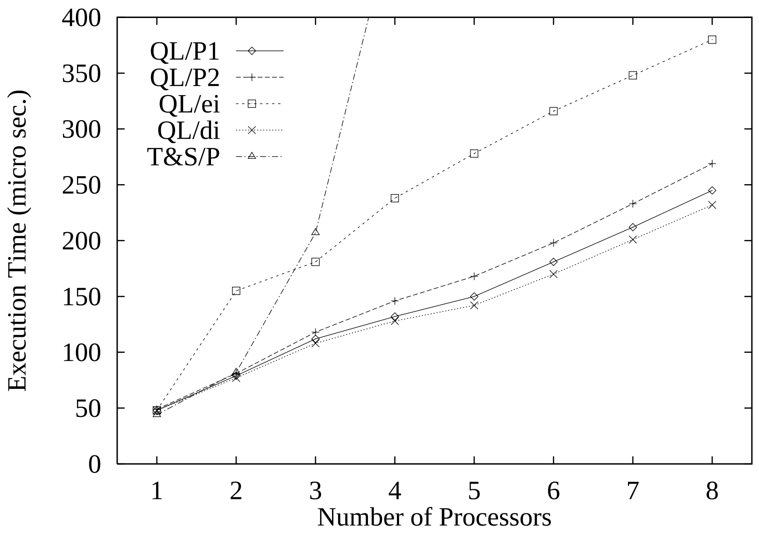


Figure 32: 99.99%-Reliable Execution Times of Critical Region
(when no interrupt is serviced)

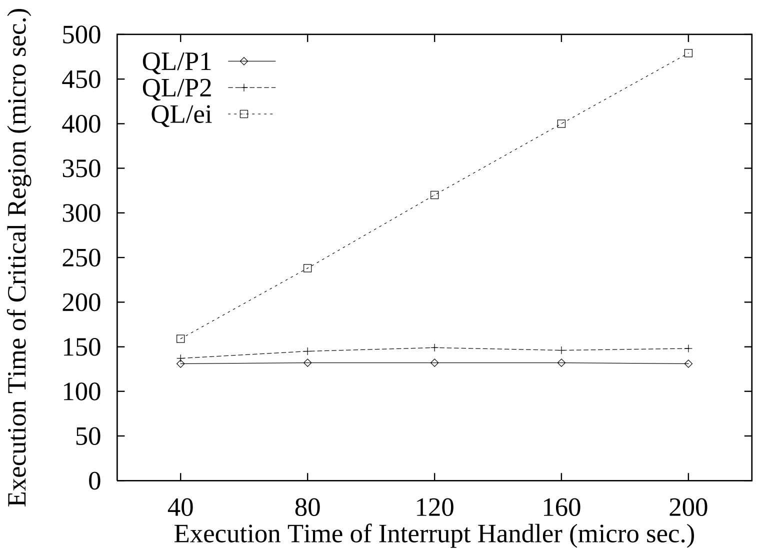


Figure 33: 99.99%-Reliable Execution Times of Critical Region
(the execution time of the interrupt handler is changed)

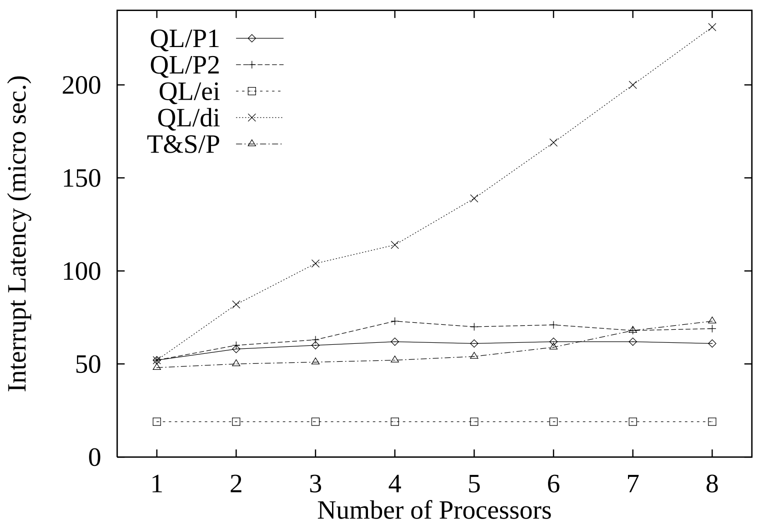


Figure 34: 99.99%-Reliable Interrupt Latency Times

In order to examine the difference of QL/P2 and QL/P1, we present the the 99.99%-reliable execution times of the critical region when an interrupt request is serviced while waiting for the lock in Figure 35. This figure demonstrates that the re-execution overhead after servicing an interrupt request is smaller with QL/P2 than QL/P1, especially when the number of processors is large.

Next, in order to evaluate the overhead of the two algorithms, we compare the average execution times of the critical region (when no interrupt is serviced) with QL/P1, QL/P2, and T&S/P. In Figure 36, its average execution time with QL/P1 or QL/P2 is about 10% longer than that with T&S/P, when the number of processors is small. When the number of processors becomes large, however, T&S/P exhibits poorer performance. This is due to the effect of shared-bus contention.

Finally, in order to check the adequacy of our evaluation metric, the 99.99%-reliable execution times of the critical region are compared with 99.9%- and 99.999%-reliable execution times and the worst execution times appeared during our measurement. As the result, though the absolute length of the execution times are different, the same evaluation result with above can be derived from each measurement data.

2.6 Combination of the Two Preemption Schemes

In Section 2.4, a problem of the first algorithm that the interrupt service overhead depends on the number of processors is pointed out, assuming that the processor continues the trial

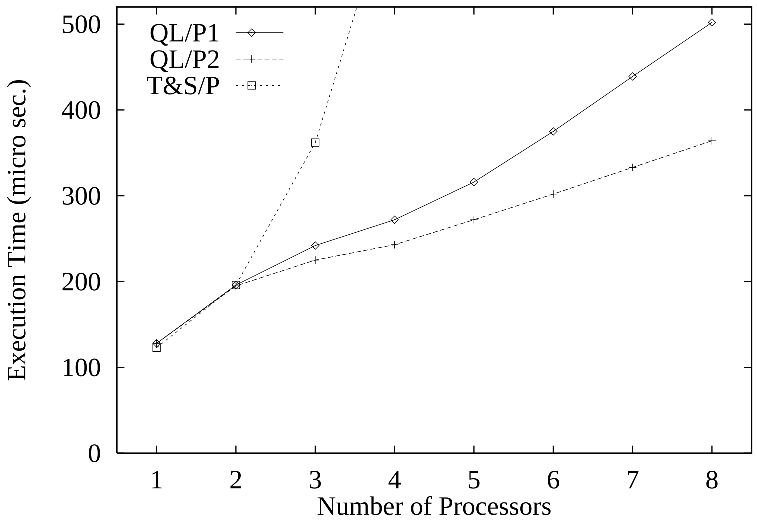


Figure 35: 99.99%-Reliable Execution Times of Critical Region
(when an interrupt is serviced)

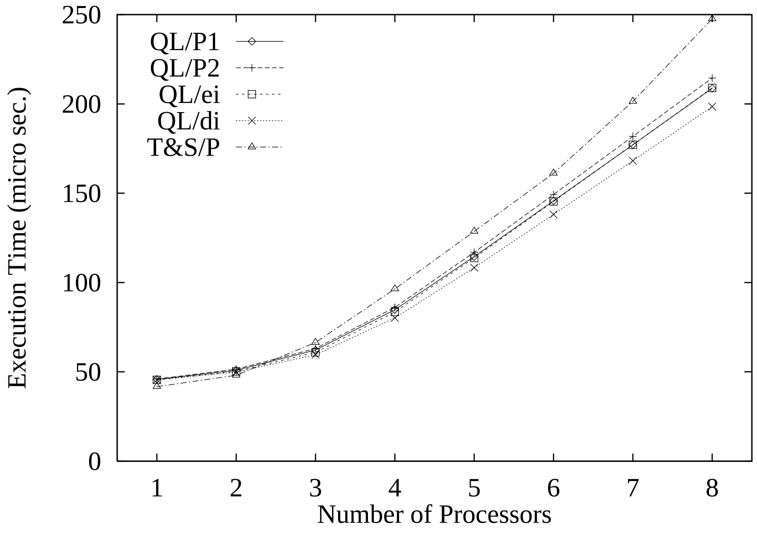
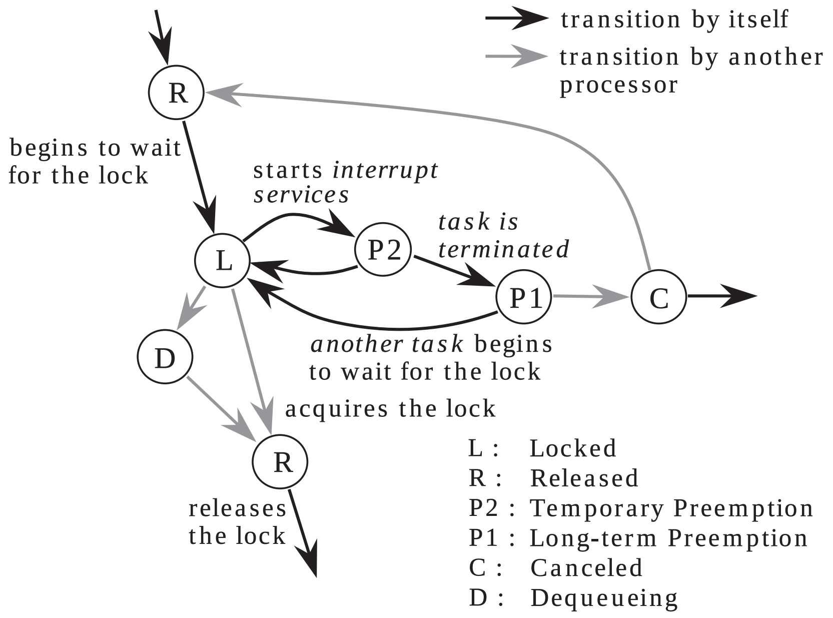


Figure 36: Average Execution Times of Critical Region



transition by itself
transition by another processor
R
begins to wait for the lock
starts *interrupt services*
L
P2
task is terminated
P1
C
D
another task begins to wait for the lock
acquires the lock
R
releases the lock
L : Locked
R : Released
P2 : Temporary Preemption
P1 : Long-term Preemption
C : Canceled
D : Dequeueing

Figure 37: State Transition of the Combined Algorithm

to acquire the lock after interrupt services are finished. When the algorithms are applied to real-time kernels, however, the interrupt service routine can request the preempted task to terminate. If the preempted task is terminated, the trial to acquire the lock is not continued.

In our improved preemption scheme, even when a processor ceases waiting for the lock, its queue node remains in the waiting queue as a garbage. The improved scheme suffers a larger overhead than the simple scheme, because the processor trying to release the lock must search the garbage queue node every time. Consequently, when the preempted task is terminated, its queue node should be removed from the waiting queue. In other words, the first preemption scheme should be adopted in this case.

For its realization, a combination of the two preemption scheme is necessary. The state of a queue node is necessary to be extended to distinguish temporary preemption and long-term preemption, and the processor trying to release the lock should handle them differently. The state transition of the combined algorithm is presented in Figure 37. In this figure, "P2" represents temporary preemption (i.e. preemption in the improved scheme) and "P1" represents long-term preemption (i.e. preemption in the simple scheme). "C" designates the transient status introduced in the algorithm with the simple preemption scheme in which a queue node is not reusable, and "D" designates another transient status that is necessary in the algorithm with the improved scheme.

3 Spin Lock with Local Precedence

In this section, we present an efficient algorithm of spin lock with local precedence based on the MCS lock algorithm described in Section 1.3. The necessity of spin lock with local precedence is pointed out in Section II.4.1.

It is obvious that a spin lock with local precedence can be realized with a priority-ordered spin lock algorithm. Specifically, a processor acquires its local lock with a higher priority and other locks with a lower one. Only two priority levels are necessary to be used. As described in Section 1.4, the maximum execution time of the lock acquisition routine or release routine depends on the number of contending processors with every priority-ordered spin lock algorithm. As the result, the overhead of a priority-ordered spin lock is generally quite large.

By making use of the fact that a spin lock with local precedence is much simpler than a priority-ordered spin lock, we can devise a more efficient algorithm of spin lock with local precedence. A spin lock with local precedence is much simpler in the following two points: (1) there are only two priority levels required (as described above), and (2) only one processor (i.e. its host processor) has the higher priority for each lock. Therefore, we can extend the MCS lock algorithm to support local precedence by preparing a variable indicating the queue node of the prioritized processor. When the prioritized processor enqueues itself to the waiting queue, it updates the variable to point to itself. The processor P_0 trying to release the lock can find the prioritized processor P_2 using the variable without searching in the waiting queue. Then, P_0 moves P_2 to the top of the queue (with the same method adopted in the Markatos' priority-ordered queueing spin lock), and passes the lock to P_2. A double-linked queue structure is necessary because P_0 needs to know the predecessor of P_2 without searching.

Pseudo-code for our algorithm based on the MCS lock is presented in Fig. 38 and 39. In this pseudo-code, the *prec* field of the *Lock* record is the variable indicating the prioritized processor.

4 Scalability of Nested Spin Locks

For real-time systems, two kind of spin locks are used depending on the timing requirements on them: (1) bounded spin locks, in which the maximum times that processors acquire and release a lock are bounded, and (2) priority-ordered spin locks, in which processors acquire a lock in the order of their priorities [9].

In this section, the scalability issue of bounded spin locks is discussed. Because worst-

```
type Node = record
    next: pointer to Node;
    prev: pointer to Node;
    locked: (Released, Locked)
end;
type Lock = record
    last: pointer to Node;
    prec: pointer to Node
end;

shared var L: Lock;
// L.last and L.prec are initialized to NULL.

procedure move_to_top(lock: pointer to Lock,
                      entry, pred, oldtop: pointer to Node);
// move entry to the top of the waiting queue of lock.
// pred is the predecessor of entry.
// oldtop is the top of the queue before the move.
    var succ: pointer to Node;
begin
    succ := entry→next;
    if succ = NULL then
        pred→next := NULL;
        if compare_and_swap(&(lock→last), entry, pred) then
            entry→next := oldtop;
            return
        end;
        repeat succ := entry→next until succ ≠ NULL
    end;
    pred→next := succ;
    succ→prev := pred;
    entry→next := oldtop
end;
```

Figure 38: The Spin Lock with Local Precedence (Part 1)

case behavior has the primary importance in real-time systems, we focus on scalability of the maximum execution times of critical sections guarded by spin locks, under the assumption that the maximum processing time within a critical section is bounded.

In general, shared resources that must be accessed exclusively by a processor are divided into some lock units in order to improve concurrency. When a processor accesses some shared resources included in different lock units, it must acquire multiple locks one by one. If FIFO spin locks are used for this kind of *nested spin locks*, the maximum execution times of a whole critical section become $O(n^m)$, where n is the number of contending processors and m is the *maximum nesting level* of locks. The strict definition of the maximum nesting level is presented in Section 4.1.

```
 var I: Node;
 var top, pred: pointer to Node;

 // try to acquire the lock L.
 I.next := NULL;
 // enqueue myself.
 pred := fetch_and_store(&(L.last), &I);
 if pred ≠ NULL then
     // when the queue is not empty.
     I.prev := pred;
     I.locked := Locked;
     pred→next := &I;
     if L_is_local_to_me then
         // direct the precedence indicator to me.
         L.prec := &I
     end;
     repeat until I.locked = Released;
     if L_is_local_to_me then
         // clear the precedence indicator.
         L.prec := NULL
     end
 end;
 //
 // critical section.
 //
 // try to release the lock L.
 top := I.next;
 if top = NULL then
     if compare_and_swap(&L, &I, NULL) then
         // the queue becomes empty.
         goto exit
     end;
     repeat top := I.next until top ≠ NULL
 end;
 // check the precedence indicator.
 if L_is_not_local_to_me and L.prec ≠ NULL then
     // the lock is passed to L.prec.
     if L.prec ≠ top then
         move_to_top(&L, L.prec, L.prec→prev, top)
     end;
     L.prec→locked := Released
 else
     // the lock is passed to top.
     top→locked := Released
 end;
exit:
```

Figure 39: The Spin Lock with Local Precedence (Part 2)

```
acquire_lock($L_3$);
acquire_lock($L_2$);
// critical section.
release_lock($L_2$);
release_lock($L_3$);
```
routine (a)

```
acquire_lock($L_2$);
acquire_lock($L_1$);
// critical section.
release_lock($L_3$);
release_lock($L_1$);
```
routine (b)

Figure 40: Example of Nested Locks

It is obvious that this simple method is not acceptable from the viewpoint of real-time scalability. In this section, we propose a method in which this order can be reduced to $O(n \cdot e^m)$, which is acceptable when m can be kept small.

In Section 4.1, assumptions and notations adopted in this section are described. An $O(n)$ algorithm when the maximum nesting level is two is proposed in Section 4.2 and its effectiveness is evaluated through performance measurements in Section 4.3. In Section 4.4, an $O(n \cdot e^m)$ algorithm for general case is discussed.

4.1 Assumptions and Notations

A system consists of n processors supporting atomic read-modify-write operations. Each processor repeatedly executes critical sections guarded by one or more locks. The maximum execution time of a critical section except for the waiting time for the locks is assumed to be bounded.

In order to avoid deadlocks, a partial order $\succ$ is defined on the set of locks in the system. A processor must acquire locks following the order. We assume that if and only if processors possibly acquire a lock L_j while holding a lock L_i, an order $L_i \succ L_j$ exists.

The nesting level λ_i is defined for each lock L_i as follows. If L_i is a minimal element (i.e. there is no L_j such that $L_i \succ L_j$), λ_i is defined to be one. Otherwise, λ_i is defined to be $max\{\lambda_j \mid L_i \succ L_j\} + 1$. We call $max\{\lambda_i\}$ as the maximum nesting level of locks in the system. Consider the example that processors in the system execute one of the two routines presented in Figure 40. In this example, $\lambda_1 = 1$, $\lambda_2 = 2$, $\lambda_3 = 3$, and the maximum nesting level in the system is three.

A lock whose nesting level is i is denoted as L_i below. When there are some locks with the same nesting level, they are represented as L_i, L_i', L_i'', $\cdots$.

We also assume that the two-phase protocol is adopted on each processor. In other words, once a processor releases a lock, it cannot acquire any lock until it releases all the locks it is holding. This assumption is adopted in order to simplify the evaluation of the maximum number of the critical sections that a processor must wait for. The estimation of its order is also valid without the assumption.

```
acquire_lock($L_1$);
// critical section.
release_lock($L_1$);
```
routine (a)

```
acquire_lock($L_2$);
// critical section.
release_lock($L_2$);
```
routine (b)

```
acquire_lock($L_2$);
acquire_lock($L_1$);
// critical section.
release_lock($L_1$);
release_lock($L_2$);
```
routine (c)

Figure 41: Nesting in Two Levels

```
acquire_lock($L'_2$);
// critical section.
release_lock($L'_2$);
```
routine (d)

```
acquire_lock($L'_2$);
acquire_lock($L_1$);
// critical section.
release_lock($L_1$);
release_lock($L'_2$);
```
routine (e)

Figure 42: Nesting in Two Levels (cont.)

4.2 Nesting in Two Levels

In this section, we focus on nested spin lock algorithms when the maximum nesting level is two. We regard them as important because the implementation method of a real-time kernel described in Section II.2.4 can be realized with the maximum nesting level being two.

Problems of Simple Methods

As mentioned before, if FIFO spin locks are simply applied to the system in which the maximum nesting level of locks is two, the maximum execution times of a whole critical section become $O(n^2)$, where n is the number of contending processors.

As an example, consider the case that each processor in the system repeatedly executes one of the three routines presented in Figure 41 in random order. Below, we illustrate the case in which the number of the critical sections that a processor P_1 must wait for until it finishes an execution of routine (c) is maximized. Assume that when P_1 tries to acquire L_2 in (c), another processor P_2 has just acquired the lock and all the other processors $P_3, \cdots, P_n$ are waiting for the lock in routine (c) in this order (Figure 43 (a)). When P_2 releases the lock, P_3 succeeds to acquire the lock. Just before P_3 tries to acquire L_1, P_2 can acquire the lock in routine (a). In this case, P_3 must wait until P_2 finishes the critical section and releases L_1, and P_1 must wait for two critical sections executed by P_2 and P_3 (Figure 43 (b)). Similarly, when P_{i-1} releases L_2, P_i succeeds to acquire the lock. Before P_i tries to acquire L_1, $P_2, \cdots, P_{i-1}$ can wait for the lock in (a). P_i must wait for the executions of $i - 2$ critical sections until it succeeds to acquire L_1, and P_1 must

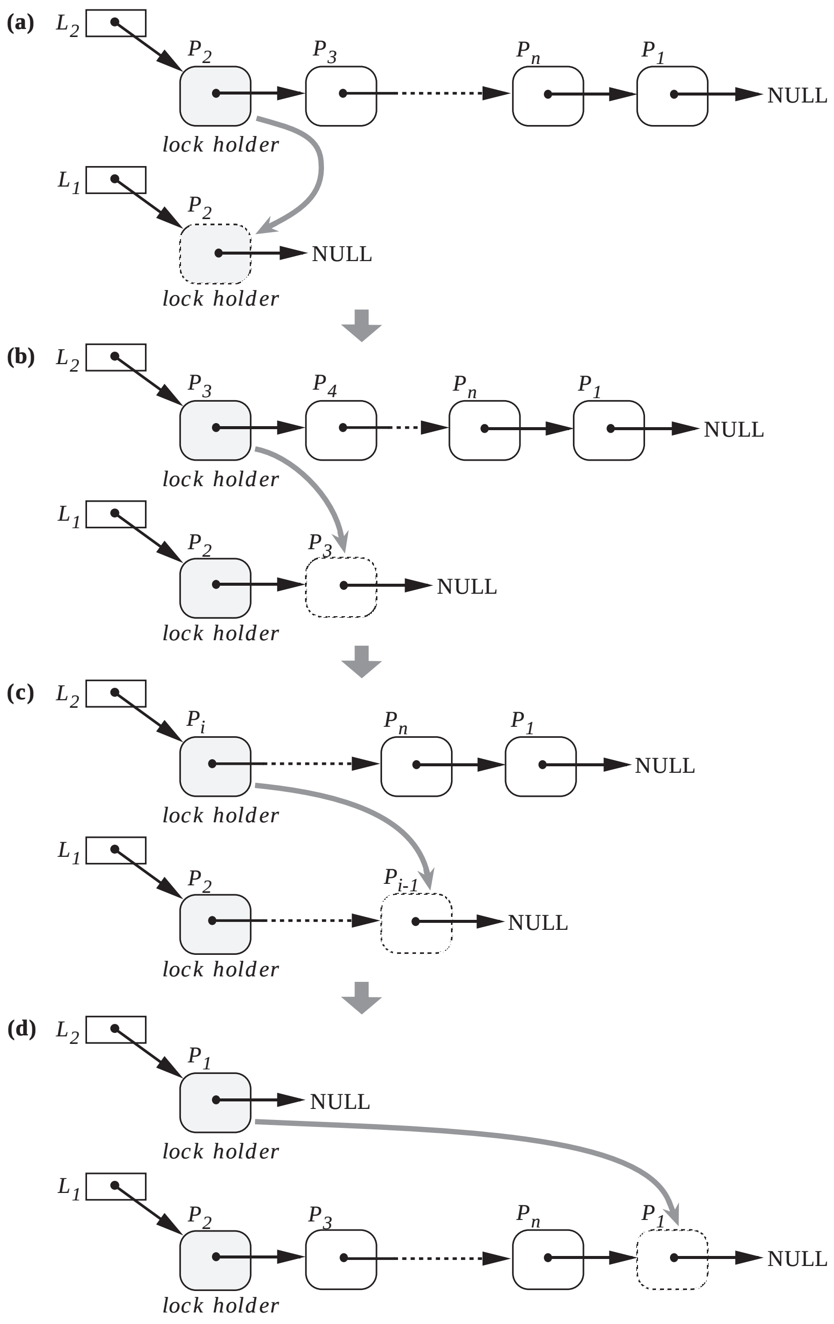


Figure 43: Worst-Case Scenario of the Simple Method

wait for $i-1$ critical sections until P_i finishes routine (c) (Figure 43 (c)). Finally, after P_1 succeeds to acquire L_2, P_1 must wait for $n-1$ critical sections before it acquires L_1 (Figure 43 (d)). As a result, the maximum number of the critical sections that a processor P_1 must wait for is $1+2+\cdots+(n-1)+(n-1)=n(n+1)/2-1$, thus $O(n^2)$. Because the maximum processing time within a critical section has an upper bound, the order of the maximum execution times of routine (c) is $O(n^2)$. That of routine (b) is also $O(n^2)$, while that of routine (a) is $O(n)$.

A simple method to improve this order is that precedence is given to the processor holding an outer lock. In case of Figure 41, the processor that is waiting for L_1 in routine (c) can acquire the lock with higher priority than other processors. Because the maximum number of the critical sections that a processor must wait for while trying to acquire L_1 in (c) is reduced to one with this method, the maximum execution times of both (b) and (c) are improved to $O(n)$. The maximum execution times of routine (a) remain to be $O(n)$, because a processor never waits for L_1 in (c) while another processor holds L_1 in (c), and because the lock is passed to a processor executing (a) when the processor in (c) releases the lock.

However, this method has a problem when each processor can also execute the two routines presented in Figure 42. In this case, a processor executing routine (a) can starve while waiting for L_1. Specifically, a processor trying to acquire L_1 in (a) can be passed by a processor executing (c) and a processor executing (e) by turns, and the maximum time until it succeeds to acquire L_1 cannot be determined.

Another method is that a processor trying to acquire nested locks reserves its turn to acquire the inner lock by enqueueing itself to the wait queue of the lock, when it begins waiting for the outermost lock. This method, however, cannot be applied when which inner lock to be acquired is determined after accessing the shared resource guarded by the outer lock, which is the case with the implementation of a real-time kernel described in Section 7.2.

Proposed Method

To solve the problem described above, we propose the following algorithm, which can make the maximum execution times of each routine $O(n)$.

When a processor begins waiting for the outermost lock, it obtains a time stamp by reading a real-time clock. Instead of using FIFO spin locks, priority-ordered spin locks are used with the time stamps as the priorities (an earlier time stamp has a higher priority).[6] With this method, the processor that begins waiting for the outermost lock

[6]The fact that a FIFO-ordered lock can be realized with a priority-ordered lock using time stamps as

earlier can acquire each lock with higher precedence. In other words, the FIFO policy is applied to the whole critical section.

This method can reduce the order of the maximum execution times of each routine to $O(n)$ with the following reason. At first, the maximum number of the higher priority critical sections (the critical sections executed by the processors with higher priorities than P_1) that a processor P_1 must wait for is $n - 1$. This is because only the processors obtaining time stamps before P_1 can acquire locks with precedence over P_1, and because each processor can execute only one critical section with a time stamp. P_1 must also wait for some lower priority critical sections. When a processor tries to acquire an inner lock, another processor with a lower priority possibly holds the lock. This is a kind of priority inversion and occurs at most once whenever a processor begins waiting for an inner lock. Note that this priority inversion does not occur in acquiring an outer lock.

When a processor P_2 with a higher priority than P_1 acquires the outer lock on which P_1 is waiting, and when P_2 tries to acquire an inner lock, P_2 must possibly wait for a critical section executed by a lower priority processor P_3 due to priority inversion. In this case, the critical section executed by P_3 should be counted in the number of the critical sections that P_1 must wait for. As a result, an upper bound on the number of the critical sections that P_1 must wait for is $2(n - 1) + 1 = 2n - 1$, thus the order of the maximum execution times of routine (c) is $O(n)$. Those of the other routines are also $O(n)$.

More precisely, the number of the critical sections that P_1 must wait for in routine (c) in Figure 41 becomes maximum in the following case. Assume that when P_1 tries to acquire L_2 in (c), another processor P_2 holds the lock and all the other processors $P_3, \cdots, P_n$ are waiting for the lock in routine (c) in this order. When P_2 releases the lock, P_3 succeeds to acquire the lock. Just before P_3 tries to acquire L_1, P_2 can acquire the lock in routine (a). In this case, P_3 must wait until P_2 releases L_1, and P_1 must wait for two critical sections. Similarly, when P_i succeeds to acquire L_2 and tries to acquire L_1, one of $P_2, \cdots, P_{i-1}$ possibly holds L_1, and P_1 must wait for two critical sections. Finally, after P_1 succeeds to acquire L_2, it possibly needs to wait for a critical section before it acquires L_1. As a result, the maximum number of the critical sections that P_1 must wait for is $1 + 2 + \cdots + 2 + 1 = 2n - 2$. The result of this exact estimation is smaller than the previous estimation, because the fact that P_2 does not suffer any priority inversions is counted in.

In implementing this method, following optimizations are possible.

1. In acquiring an outer lock (a lock whose nesting level is two), a FIFO spin lock algorithm can be used instead of a priority-ordered one.

priorities is pointed out by Craig [9].

2. A sequence number that a processor begins waiting for the outermost lock, which can be implemented with fetch_and_increment operation, can be used as the time stamp instead of an absolute time read from a real-time clock.

4.3 Performance Evaluation

In this section, the effectiveness of the algorithm proposed in the previous section (called TF, in this section) is examined through performance evaluation. Its performance is compared with the method that FIFO spin locks are simply used for all locks (called SF) and the method that precedence is given to the processor holding an outer lock (called PI).

Evaluation Method

We have adopted the MCS lock algorithm [38] for the FIFO spin locks and the single-linked queue version of the Markatos' algorithm presented in Figure 21 and 22 for priority-ordered spin locks. The FIFO spin lock with precedence, which is necessary to implement PI, is realized using the single-linked queue version of the Markatos' algorithm. In implementing TF, we have used a FIFO spin lock algorithm for the outer locks and a priority-ordered one for the inner locks. We have also used a sequence number that a processor begins waiting for the outermost lock instead of a real-time clock.

Evaluation Results

At first, processors in the system repeatedly execute one of the three routines presented in Figure 41 in random order. The probability that a processor executes routine (a) is made four times larger that each of other routines. A processor accesses the shared bus several number of times and waits for a while using empty loops inside the critical section. In case of routine (c), shared bus accesses and an empty loop are also inserted between two acquire_lock operations. Without spin locks (and the routine for obtaining the sequence number in case of TF), the execution time of each critical section is about 30 μs, including the overhead for measuring execution times. As an example, pseudo-code of the measurement routines with TF are presented in Figure 44.

Figure 45 presents the 99.99%-reliable execution times of routine (c). When the number of processors is large, the execution times of routine (c) is quite slower with the simplest method (SF) than our proposed method (TF). The execution times with TF increase a little more than $O(n)$. This is because the lock release times in the Markatos' lock become long as the number of processors is increased. This problem is expected

```
t0 := read_current_time();
prio := get_sequence_number();
acquire_lock_markatos($L_1$, prio);
// some shared bus accesses
//    and two empty loops (about 22$\mu$sec).
release_lock_markatos($L_1$);
t1 := read_current_time();
// measurement result is ($t1 - t0$).
```

routine (a)

```
t0 := read_current_time();
acquire_lock_mcs($L_2$);
// some shared bus accesses
//    and two empty loops (about 22$\mu$sec).
release_lock_mcs($L_2$);
t1 := read_current_time();
// measurement result is ($t1 - t0$).
```

routine (b)

```
t0 := read_current_time();
prio := get_sequence_number();
acquire_lock_mcs($L_2$);
// some shared bus accesses
//    and an empty loop (about 11$\mu$sec).
acquire_lock_markatos($L_1$, prio);
// some shared bus accesses
//    and an empty loop (about 11$\mu$sec).
release_lock_markatos($L_1$);
release_lock_mcs($L_2$);
t1 := read_current_time();
// measurement result is ($t1 - t0$).
```

routine (c)

```
for i := 1 to number_of_loop do
  case random_number() of
    1,2,3,4:
      execute routine (a);
    5:
      execute routine (b);
    6:
      execute routine (c);
  end
end
```

main routine

Figure 44: Measurement Routines with TF

to be relieved with the PR-lock algorithm [26]. The 99.99%-reliable execution times of routine (b) are almost same with routine (c) except that the absolute times are little shorter.

Figure 46 presents the 99.99%-reliable execution times of routine (a) under the same condition. Though the execution times of routine (c) are fastest with PI, those of routine (a) are slowest with the method.

The problem of PI becomes more obvious, when processors repeatedly execute one of the fives routines in Figure 41 and 42 in random order. Figure 47 presents the 99.99%-reliable execution times of routine (a) under this condition. The probability that a processor executes routine (a) is made twice larger than each of other routines. In this figure, the execution times with PI are much slower than the other methods.

From these results, we can see that our proposed method is the most appropriate algorithm of the three methods from the viewpoint of real-time scalability.

Finally, in order to examine the average performance of the algorithms, we present the average execution times of routine (c) and (a) in case of three routines in Figure 48 and 49 respectively. Because the difference between SF and TF is very small in routine (c) (Figure 48), we can say that SF is more appropriate in case that improving average performance is the primary concern.

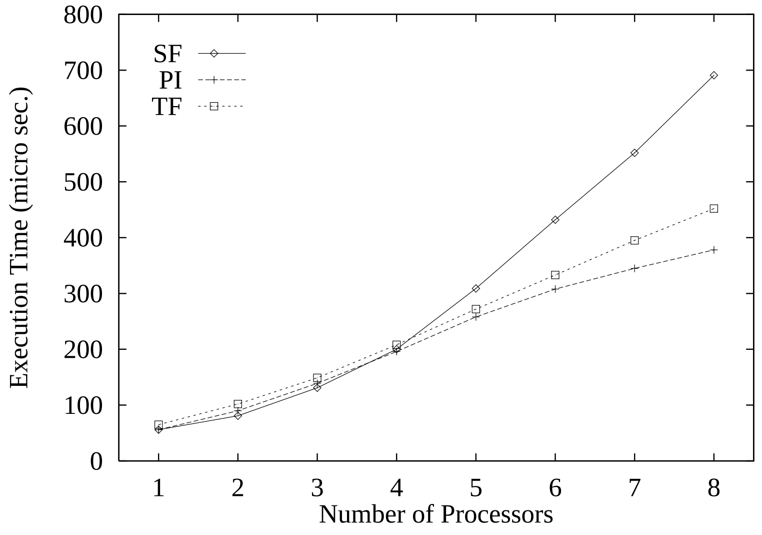


Figure 45: 99.99%-Reliable Execution Times of Routine (c)

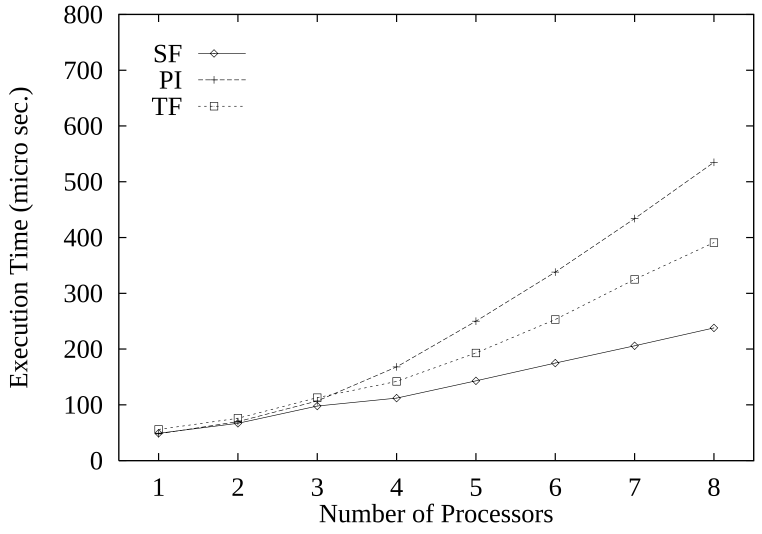


Figure 46: 99.99%-Reliable Execution Times of Routine (a)

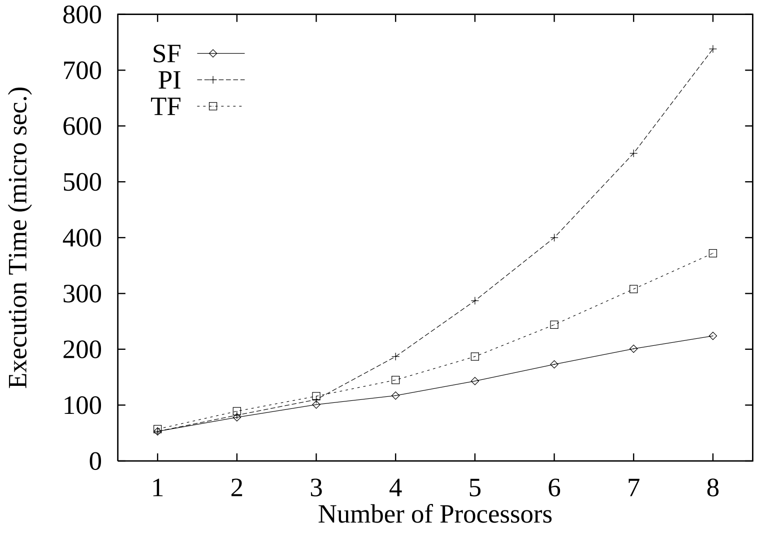


Figure 47: 99.99%-Reliable Execution Times of Routine (a)

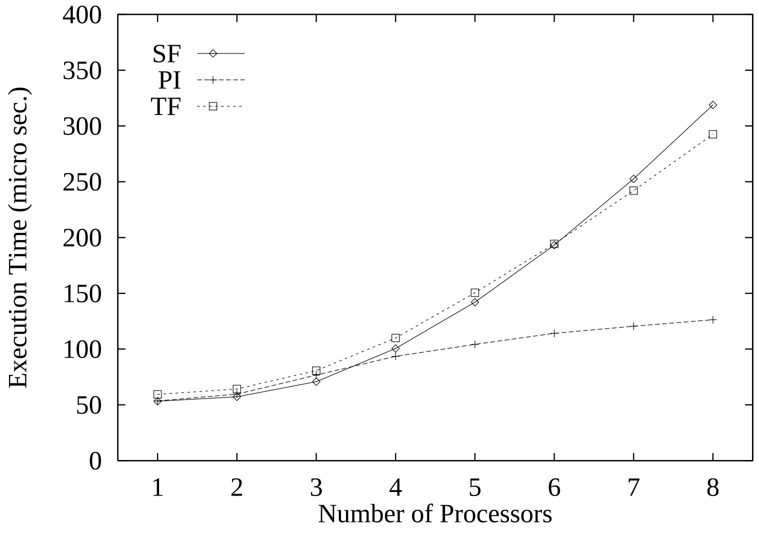


Figure 48: Average Execution Times of Routine (c)

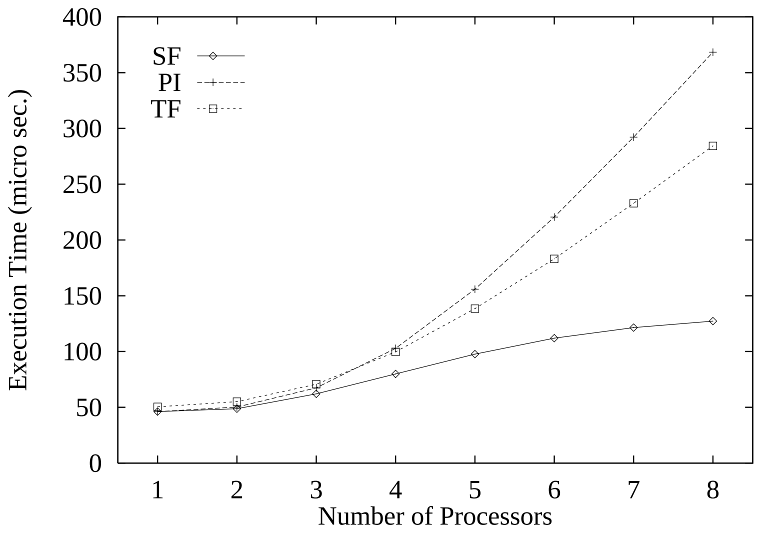


Figure 49: Average Execution Times of Routine (a)

4.4 Nesting in Three or More Levels

If FIFO spin locks are simply used when the maximum nesting level of locks is m, the maximum execution times of a whole critical section become $O(n^m)$. An effective method to improve this order is proposed in this section.

Priority Inversion Problem

When the maximum nesting level of locks is more than or equal to three, the method proposed in Section 4.2 does not work effectively due to uncontrolled priority inversions.

Consider the example that processors execute one of the three routines in Figure 50 in random order. Assume the case that a processor P_1 holds L_3 and waits for L_2 in routine (c), and that another processor P_2 with a lower priority than P_1 holds L_2 and tries to acquire L_1 in (a). Processors with priorities lower than P_1 and higher than P_2 can acquire L_1 with precedence over P_2. While P_2 is waiting for those processors, P_1 must wait also and the duration of the priority inversion becomes long. As a result, the maximum execution times of (c) cannot be improved to $O(n)$. Note that this uncontrolled priority inversions do not occur when the maximum nesting level is two.

Priority inversion problems in the context of spin locks are discussed in Section 5 in more detail.

```
acquire_lock($L_2$);
acquire_lock($L_1$);
// critical section.
release_lock($L_1$);
release_lock($L_2$);
```
routine (a)

```
acquire_lock($L'_2$);
acquire_lock($L_1$);
// critical section.
release_lock($L_1$);
release_lock($L'_2$);
```
routine (b)

```
acquire_lock($L_3$);
acquire_lock($L_2$);
acquire_lock($L_1$);
// critical section.
release_lock($L_1$);
release_lock($L_2$);
release_lock($L_3$);
```
routine (c)

Figure 50: Nesting in Three Levels

Incorporating Priority Inheritance Scheme

We adopt a priority inheritance scheme to solve this problem. With the basic priority inheritance scheme in which a processor holding some locks inherits the highest priority of the processors that are waiting for one of the locks, the duration of priority inversions can be reduced. Since chained priority inversions cannot be avoided with this method, however, the maximum execution times of a critical section become $O(n \cdot e^m)$ with the following reason.

At first, we estimate the maximum number of priority inversions that a processor P encounters while it executes a critical section guarded by a lock L_i whose nesting level is i, under the assumption that there are no higher priority processor than P. We denote the maximum number of these critical sections as $inv(i)$ and estimate it with an induction on i. When P tries to acquire L_i, another processor P_1 which has a lower priority than P possibly holds the lock and P must wait for the critical section executed by P_1. If the nesting level of the lock is one (i.e. $i = 1$), no other priority inversions can occur, thus $inv(1) = 1$. When $i > 1$, at most $inv(i-1)$ priority inversions also occur during P_1 is executing the critical section because P_1 may try to acquire another lock whose nesting level is smaller than i within the critical section. After P succeeds to acquire L_i, it may also try to acquire another lock whose nesting level is smaller than i within the critical section. During its execution, at most $inv(i-1)$ priority inversions can occur. As the result, $inv(i) = 2 \cdot inv(i-1) + 1$ then $inv(i) = 2^i - 1$. When L_m has the maximum nesting level in the system, the maximum number of critical sections that P must wait for until it finishes the execution of a critical section guarded by L_m is $inv(m-1) = 2^{m-1} - 1$ under the assumption that there is no higher priority processor than P.

Next, we consider the case that a processor P_1 which has a higher priority than P is added, and estimate its effect on the maximum number of critical sections that P must wait for until it finishes the execution of a critical section guarded by L_m, which has the maximum nesting level in the system. We estimate the effect with the following two cases.

(a) Suppose the case that P_1 is holding or waiting for L_m when P begins waiting for L_m. In this case, P_1 can encounter at most $inv(m-1)$ priority inversions during its execution of the critical section guarded by L_m. Because P must also wait for the critical section of P_1, the maximum number of critical sections that P must wait for is increased with $inv(m-1)+1 = 2^{m-1}$.

(b) Suppose the case that P_1 is holding or waiting for another lock when P begins waiting for L_m. In this case, some lower priority processors than P can inherit the priority of P_1 and cause additional priority inversions on P. The maximum number of the lower priority processors that can inherit the priority of P_1 corresponds to the maximum number of priority inversions that P_1 encounters, i.e. $inv(m-1)$. In addition to them, P is necessary to wait for the critical section of P_1, when P and P_1 try to acquire a same inner lock. As the result, the maximum number of critical sections that P must wait for is increased with $inv(m-1)+1 = 2^{m-1}$.

In each case, the maximum number of critical sections that P must wait for is increased with 2^{m-1}. Because the outermost lock is acquired in a FIFO order with our method, at most $n-1$ processors have higher priorities than P. Consequently, the maximum number of critical sections that P must wait for until it finishes the execution of a critical section guarded by L_m is $(2^{m-1}-1)+(n-1)\cdot 2^{m-1} = n\cdot 2^{m-1}-1$. Note that this also includes some overestimations.

As the result, the order of the maximum execution times of critical sections is shown to be $O(n\cdot e^m)$ with the basic priority inheritance scheme. We can say that this method has real-time scalability on the number of contending processors but not on the maximum nesting level. Algorithms of spin locks with the basic priority inheritance scheme will be presented in Section 5.

The priority ceiling policy can also be adopted, when there is prior knowledge on which locks are acquired in each critical section. In the concrete, when a processor acquires the outermost lock, the priority ceiling of the other locks that are required (or possibly required) by the processor within the critical section is set to the priority of the processor. When the priority ceiling of the lock that a processor tries to acquire is higher than its priority, the processor must wait with spinning even if the lock is not held by any processor.[7]

[7]Though induced from the same policy, the behavior of "priority ceiling spin lock" is quite different from those of the priority ceiling protocol [60] or its extension for shared memory multiprocessors [46].

In Section 4.2, we have mentioned the method that a processor trying to acquire nested locks reserves its turn to acquire the inner locks by enqueueing itself to their wait queue when it begins waiting for the outermost lock. When complete knowledge on all required locks in each critical section is available, the priority ceiling method is same with this method. To the contrary, if there is no knowledge on required locks at all, the priority ceiling method reduced to the situation that all shared resources in the system are guarded by a single lock, which severely degrades concurrency of the system.

5 Priority Inheritance Spin Locks

As described in Section 4.4, in order to realize bounded and scalable nested spin locks for real-time systems, a priority inheritance scheme is necessary to be incorporated in priority-ordered spin locks. A *priority inheritance spin lock* is also necessary for priority-ordered nested spin locks. This section proposes two algorithms of priority inheritance spin lock based on the Markatos' algorithm [36].

Shared resources that must be accessed exclusively by a processor are usually divided into some lock units in order to improve concurrency. When a processor accesses some shared resources included in different lock units, it must acquire multiple locks one by one. If priority-ordered spin locks are simply used for this kind of nested spin locks, uncontrolled priority inversions can occur. The uncontrolled priority inversion problem in nested spin locks is described in Section 5.1.

After describing the necessity of priority inheritance spin locks in Section 5.1, we present two algorithms of priority inheritance spin lock in Section 5.2. In Section 5.3, their effectiveness is evaluated through performance measurements.

5.1 Priority Inversion and Priority Inheritance

Priority inversion and priority inheritance schemes, which are promising approaches to solve the uncontrolled priority inversion problem, are actively studied in the context of task scheduling algorithms [48, 60, 47]. In this section, we illustrate that the uncontrolled priority inversion problem also occurs in the context of spin locks and demonstrate that the basic priority inheritance scheme is also effective in this case.

This is because the processor which cannot acquire a lock is blocked with those protocols, while it spins with our situation.

```
acquire_lock($L_2$);
// critical section.
release_lock($L_2$);
```

routine (a)

```
acquire_lock($L_1$);
acquire_lock($L_2$);
// critical section.
release_lock($L_2$);
release_lock($L_1$);
```

routine (b)

Figure 51: Example of Nested Spin Locks

Priority Inversion Problem in Nested Spin Locks

Priority inversion in the context of spin locks is the phenomenon that a higher priority processor is forced to wait for the execution of a lower priority processor. Because priority inversion cannot be avoided unless a higher priority processor can steal the lock held by a lower priority one, how to minimize its duration is a concern. When the maximum duration of a priority inversion cannot be determined, it is called uncontrolled.

When priority-ordered spin locks are used for nested spin locks, uncontrolled priority inversions can occur. A typical case is described as follows.

Example 1 (uncontrolled priority inversion)

We assume that P_1, P_2, P_3, and P_4 are processors arranged in descending order of priority with P_1 having the highest priority, and that L_1 and L_2 are locks. These processors repeatedly execute one of the two routines presented in Figure 51. Suppose the case that when P_1 begins executing routine (b) and tries to acquire the lock L_1, P_4 is holding L_1 and is waiting for the other lock L_2 in routine (b). If P_2 and P_3 repeatedly execute routine (a) in this situation, P_2 and P_3 can acquire L_2 alternately and P_4 must wait for L_2 all the while. Because P_1 must also wait for the executions of P_2 and P_3, this duration is a priority inversion. Obviously, the maximum duration of this priority inversion cannot be determined.

Spin Lock with Priority Inheritance

In order to solve this problem of uncontrolled priority inversions, we introduce the priority inheritance scheme to spin locks. The fundamental concept of priority inheritance scheme is that when a processor makes some higher priority processors wait, its priority should be raised to the level of the highest priority processor among the waiting ones. In other words, the processor inherits the priority of the highest priority processor blocked by it. Also, priority inheritance must be transitive. For example, suppose that P_1, P_2, and P_3 are three processors in descending order of priority. When P_2 makes P_1 wait and P_3 makes P_2 wait, P_3 should inherit the priority of P_1.

With the basic priority inheritance scheme, which is the naive realization of the concept, the uncontrolled priority inversion problem illustrated in Example 1 is solved as follows. When P_1 tries to acquire L_1 and begins waiting for it, P_4, which is holding L_1, inherits the priority of P_1 because P_1 is forced to wait by P_4. Because the inherited priority is higher than the priorities of P_2 and P_3, P_4 can acquire L_2 with precedence over P_2 and P_3. As the result, P_1 need not wait for the alternate executions of routine (a) by P_2 and P_3, and the maximum duration of the priority inversion can be bounded.

When a processor releases one of the locks, its priority is necessary to be re-calculated in general. Specifically, its priority is changed to the highest one of its original priority and the priorities of the processors that is waiting for the locks held by the former one. When the processor releases the last lock it is holding, its priority is recovered to its original level.

This re-calculation can be omitted under the following two assumptions. The first assumption is that the inherited priority is used only for spin locks, and not used for task scheduling. In more specific, the inherited priority is used only when the processor tries to acquire another lock. The second assumption is that the two-phase protocol is adopted. In other words, once a processor releases a lock, it cannot acquire another lock until it releases all the locks it is holding. In the following sections, we assume that these two conditions are satisfied and propose priority inheritance spin lock algorithms. Under these two assumptions, once the priority of a processor is raised, it need not be lowered until it releases all the locks. These assumptions can be removed by adding re-calculation routines to the algorithms proposed in Section 5.2 at the cost of some runtime overhead.

With the two assumptions described above, the required behavior of priority inheritance spin locks can be summarized as follows.

1. Processors acquire a lock in the order of their priorities.

2. When a processor P_1 begins waiting for a lock, and when its priority is higher than the priority of the processor P_2 that is holding the lock, the priority of P_2 is raised to that of P_1.

3. When the priority of a processor P_1 is raised while waiting for a lock, and when its new priority is higher than the priority of the processor P_2 that is holding the lock, the priority of P_2 is raised to the new priority of P_1.

```
// global shared variables.
**shared var** L1, L2: Lock;

// local variables (allocated for each processor).
**var** I1, I2: Node;
**var** my_prio: integer;
**var** my_notify: boolean;
// *my_notify* is necessary only in the second algorithm.

// initialize *my_prio*.
acquire_first_lock(&L1, &I1);
acquire_second_lock(&L2, &I2, &L1);
// critical section.
release_lock(&L2, &I2);
release_lock(&L1, &I1);
```

Figure 52: Usage of Priority Inheritance Spin Locks

5.2 Priority Inheritance Spin Lock Algorithms

In this section, we present two algorithms of priority inheritance spin locks, which are based on the single-linked queue version of the Markatos' lock algorithm presented in Figure 21 and 22. With the Markatos' algorithm, processors trying to acquire a lock are linked to the waiting queue in a FIFO order. In releasing the lock, a processor searches the highest priority processor in the waiting queue and passes the lock to it.

The first algorithm is a straightforward extension of the Markatos' lock algorithm. A new variable that indicates the highest priority of the processors that is waiting for the lock is prepared for each lock. The processor holding the lock *polls* the variable while it is waiting for another lock. When the processor detects that the highest priority is raised, it inherits the priority. Because any processor can poll this highest priority variable for each lock, pollings on the variable are remote memory accesses and severely increase the interconnection network traffic with a multiprocessor system without a coherent cache. The second algorithm is to avoid this non-local spinning and is expected to have higher performance without a coherent cache.

In order to avoid unnecessary complexity, this section presents the pseudo-codes of the algorithms when a processor acquires at most two locks at the same time. With this simplification, we prepares two lock acquisition routines: *acquire_first_lock* for acquiring the outer lock and *acquire_second_lock* for acquiring the inner lock. A typical usage of the routines is illustrated in Figure 52. The third argument of *acquire_second_lock* is the pointer to the lock that the processor is holding.

In Figure 52, the *my_prio* variable is to store the current priority of the processor,

```
type Node = record
    next: pointer to Node;
    locked: (Released, Locked);
    prio: integer
end;

type Lock = record
    last: pointer to Node;
    maxprio: integer;
    notifyp: pointer to boolean
end;
// notifyp is necessary only in the second algorithm.
// last and notifyp fields should be initialized to NULL.
// maxprio field should be initialized to MIN_PRIO.

type NodePtr = pointer to Node;
type LockPtr = pointer to Lock;
```

Figure 53: Data Structures for Priority Inheritance Spin Locks

and must be initialized before the processor tries to acquire the outermost lock. With a multiprocessor without a coherent cache, the local variables should be placed on the processor's locally accessible shared memory.

The First Algorithm

Figure 53 and 54 present the common data structures and subroutines for both algorithms (some of them are necessary only in the second algorithm). The *Lock* record should be prepared for each lock in the system. Its *maxprio* field is the highest priority variable for the lock. When the lock is empty (in other words, no processor holds the lock), the *last* field of its *Lock* record is *NULL* and its *maxprio* field is *MIN_PRIO*, which designates the minimum priority value. A *Node* record is necessary for each nested lock for each processor.

Figure 55 and 56 present the pseudo-code of the first algorithm. Compared to the Markatos' algorithm, two invocations of the *raise_priority* procedure, which is to update the *maxprio* field of *lock* when it is lower than the *newprio* parameter, are added to the *acquire_first_lock* procedure and the *acquire_second_lock* procedure in Figure 55. The first invocation (marked with "$*1$") is to raise *maxprio* for priority inheritance, when the processor begins waiting for the lock. The second one (marked with "$*2$") is to set *maxprio*, when the processor succeeds to acquire the lock without waiting. In the *acquire_second_lock* procedure, the processor must check the *maxprio* field of *lock1*, which is the lock being held by the processor, while waiting for *lock*. When *maxprio*

```
procedure raise_priority(lock: LockPtr, newprio: integer): boolean;
    var prio: integer;
begin
  retry:
    prio := lock→maxprio;
    if newprio > prio then
        if compare_and_swap(&(lock→maxprio), prio, newprio) then
            return TRUE
        end;
        goto retry
    end;
    return FALSE
end;

procedure raise_priority_notify(lock: LockPtr, newprio: integer);
// necessary only in the second algorithm.
    var notifyp: pointer to boolean;
begin
    if raise_priority(lock, newprio) then
        notifyp := lock→notifyp;
        if notifyp ≠ NULL then
            // set the notification flag.
            ∗notifyp := TRUE
        end
    end
end;

procedure move_to_top(lock: LockPtr, entry, pred, oldtop: NodePtr);
// move entry to the top of the waiting queue of lock.
// pred is the predecessor of entry.
// oldtop is the top of the queue before the move.
    var succ: NodePtr;
begin
    succ := entry→next;
    if succ = NULL then
        pred→next := NULL;
        if compare_and_swap(&(lock→last), entry, pred) then
            entry→next := oldtop;
            return
        end;
        repeat succ := entry→next until succ ≠ NULL
    end;
    pred→next := succ;
    entry→next := oldtop
end;
```

Figure 54: Subroutines for Priority Inheritance Spin Locks

```
    procedure acquire_first_lock(lock, LockPtr, me: NodePtr);
        // try to acquire lock.
        var pred: NodePtr;
    begin
        me→next := NULL;
        // enqueue myself.
        pred := fetch_and_store(&(lock→last), me);
        if pred ≠ NULL then
            // when the queue is not empty.
            me→locked := Locked;
            me→prio := my_prio;
            pred→next := me;
*1          raise_priority(lock, my_prio);
            repeat until me→locked = Released
        else
            // succeed to acquire the lock without waiting.
*2          raise_priority(lock, my_prio)
        end
    end;

    procedure acquire_second_lock(lock: LockPtr, me: NodePtr, lock1: LockPtr);
        // try to acquire lock.
        var pred: NodePtr;
    begin
        me→next := NULL;
        // enqueue myself.
        pred := fetch_and_store(&(lock→last), me);
        if pred ≠ NULL then
            // when the queue is not empty
            me→locked := Locked;
            me→prio := my_prio;
            pred→next := me;
*1          raise_priority(lock, my_prio);
            repeat
*3              if lock1→maxprio > my_prio then
                        // lock1→maxprio is non-local access.
                    my_prio := lock1→maxprio;
                    me→prio := my_prio;
                    raise_priority(lock, my_prio)
                end
            until me→locked = Released
        else
            // succeed to acquire the lock without waiting.
*2          raise_priority(lock, my_prio)
        end
    end;
```

Figure 55: The First Algorithm (Part 1)

```
    procedure release_lock(lock: LockPtr, me: NodePtr);
    // try to release lock.
        var top, entry, pred: NodePtr;
        var hentry, hpred: NodePtr;
    begin
*4      lock→maxprio = MIN_PRIO;
        top := me→next;
        if top = NULL then
            if compare_and_swap(&(lock→last), me, NULL) then
                // the queue becomes empty.
                return
            end;
            repeat top := me→next until top ≠ NULL
        end;
        // search for the higest priority processor.
        hentry := top;
        pred := top;
        entry := pred→next;
        while entry ≠ NULL do
            if (entry→prio > hentry→prio) then
                hentry := entry;
                hpred := pred;
            end;
            pred := entry;
            entry := pred→next
        end;
        // now, hentry is the higest priority processor.
        if hentry ≠ top then
            move_to_top(lock, hentry, hpred, top)
        end;
*5      raise_priority(lock, hentry→prio);
        hentry→locked = Released
    end;
```

Figure 56: The First Algorithm (Part 2)

becomes higher than the priority of the processor (the *if* statement marked with “$*3$”), it inherits *maxprio* of *lock1* and updates *maxprio* of *lock* for transitive priority inheritance.

The only difference of the *release_lock* procedure in Figure 56 with that of the Markatos’ algorithm is the necessity of updating the *maxprio* field (two lines marked with “$*4$” and “$*5$”). Assigning *MIN_PRIO* to the *maxprio* field at first is necessary to avoid some racing conditions.

This algorithm can be easily generalized to the case that a processor acquires more than two locks at the same time with the following method. The list of locks held by a processor should be maintained using an array or a linked list. In the generalized version of the *acquire_lock* procedure, the *maxprio* fields of all the locks in the list should be checked while waiting for another lock. If some of them are higher than the priority of the processor, it inherits the highest priority among them.

Avoiding Non-Local Spinning

While a processor is waiting for a lock in the *acquire_second_lock* procedure of the first algorithm, the *maxprio* field of the holding lock is accessed repeatedly (marked with "$*3$"). This accesses cause a heavy traffic on the interconnection network without a coherent cache.

With the second algorithm presented in Figure 57 and Figure 58, this problem is solved by introducing a flag to notify that the *maxprio* field is modified. This notification flag (the *my_notify* variable in Figure 52) is prepared for each processor on its locally accessible shared memory. A processor waiting for a lock in the *acquire_second_lock* procedure in Figure 57 reads the *maxprio* field only when the notification flag of the processor is set (the *if* statement marked with "$*6$"). Thus the non-local spinning can be avoided. It also checks the *maxprio* field when it begins waiting for a lock (by assigning *TRUE* to *my_notify*). Introducing the notification flag is also advantageous when a processor acquires more than two locks at the same time, because only one memory location (i.e. the notification flag) is necessary to be checked in the waiting loop. Maintaining the list of locks held by a processor is still necessary in this case.

Also, the *raise_priority_notify* procedure is used instead of *raise_priority* (three lines marked with "$*7$") in Figure 57. After updating the *maxprio* field of *lock*, the *raise_priority_notify* procedure sets the notification flag of the processor holding the lock. In order to locate the notification flag of the lock holder, a new field *notifyp* which points to the notification flag is introduced in the *Lock* record. The *notifyp* field of a lock is set when a processor succeeds to acquire the lock (two lines marked with "$*8$"). The field is also necessary to be cleared to *NULL* at the top of the *release_lock* procedure in Figure 58 (marked with "$*9$").

There is a slight chance that the notification flag of a wrong processor is set. Specifically, suppose the case that the processor holding a lock passes the lock to another one and its *notifyp* field is changed, after yet another processor reads the *notifyp* field of the lock in the *raise_priority_notify* procedure and before it writes *TRUE* on $*$*notifyp*. In this case, the notification flag of the processor that has already passed the lock to another is set. Although this difficulty can increase the interconnection network traffic a little, it does not cause wrong behavior.

```
   procedure acquire_first_lock(lock: LockPtr, me: NodePtr);
       var pred: NodePtr;
   begin
       me→next := NULL;
       // enqueue myself.
       pred := fetch_and_store(&(lock→last), me);
       if pred ≠ NULL then
           // when the queue is not empty.
           me→locked := Locked;
           me→prio := my_prio;
           pred→next := me;
*7         raise_priority_notify(lock, my_prio);
           repeat until me→locked = Released
       else
           // succeed to acquire the lock without waiting.
           raise_priority(lock, my_prio)
       end;
*8     lock→notifyp := &my_notify
   end;

   procedure acquire_second_lock(lock: LockPtr, me: NodePtr, lock1: LockPtr);
       var pred: NodePtr;
   begin
       me→next := NULL;
       pred := fetch_and_store(&(lock→last), me);
       if pred ≠ NULL then
           me→locked := Locked;
           me→prio := my_prio;
           pred→next := me;
*7         raise_priority_notify(lock, my_prio);
           my_notify := TRUE;
           repeat
               // check if a priority inheritance is notified.
*6             if my_notify then
                   my_notify := FALSE;
                   if lock1→maxprio > my_prio then
                       my_prio := lock1→maxprio;
                       me→prio := my_prio;
*7                     raise_priority_notify(lock, my_prio)
                   end
               end
           until me→locked = Released
       else
           raise_priority(lock, my_prio)
       end;
*8     lock→notifyp := &my_notify
   end;
```

Figure 57: The Second Algorithm (Part 1)

```
    procedure release_lock(lock: LockPtr, me: NodePtr);
        var top, entry, pred: NodePtr;
        var hentry, hpred: NodePtr;
    begin
        lock→maxprio := MIN_PRIO;
*9      lock→notifyp := NULL;
        top := me→next;
        if top = NULL then
            if compare_and_swap(&(lock→last), me, NULL) then
                // the queue becomes empty.
                return
            end;
            repeat top := me→next until top $\neq$ NULL
        end;
        // search for the higest priority processor.
        hentry := top;
        pred := top;
        entry := pred→next;
        while entry $\neq$ NULL do
            if (entry→prio $>$ hentry→prio) then
                hentry := entry;
                hpred := pred
            end;
            pred := entry;
            entry := pred→next
        end;
        // now, hentry is the higest priority processor.
        if hentry $\neq$ top then
            move_to_top(lock, hentry, hpred, top)
        end;
        raise_priority(lock, hentry→prio);
        hentry→locked := Released
    end;
```

Figure 58: The Second Algorithm (Part 2)

5.3 Performance Evaluation

In this section, the effectiveness of the priority inheritance spin lock algorithms proposed in the previous section is examined through performance evaluation. Their performance is compared with the simple priority-ordered spin locks without supporting priority inheritance scheme. We have used the single-linked queue version of the Markatos' lock algorithm for this purpose.

Evaluation Method

We have used one to eight processors for the evaluation. The original (or assigned)

```
acquire_lock($L_2$);
// critical section.
release_lock($L_2$);
```

routine (a)

```
acquire_lock($L_1$);
acquire_lock($L_2$);
// critical section.
release_lock($L_2$);
release_lock($L_1$);
```

routine (b)

```
acquire_lock($L'_1$);
acquire_lock($L_2$);
// critical section.
release_lock($L_2$);
release_lock($L'_1$);
```

routine (c)

```
acquire_lock($L''_1$);
acquire_lock($L_2$);
// critical section.
release_lock($L_2$);
release_lock($L''_1$);
```

routine (d)

Figure 59: Evaluation Routines

priority of processor is fixed to its ID number. Each processor repeatedly executes one of the four routines presented in Figure 59 in random order. Routines (c) and (d) are introduced in order to expose the problem of non-local spinning with the first algorithm.[8] The execution time of each routine is measured for each execution, and their distributions are obtained. Inside the critical section, a processor accesses the shared bus several number of times and waits for a while using empty loops. In case of routines (b), (c), and (d), shared bus accesses and an empty loop are also inserted between *acquire_first_lock* and *acquire_second_lock*. Without spin locks, the execution time of each routine is about 30 μs, including the overhead for measuring execution times. Each processor also waits for a random time after each execution of the routines.

Because our evaluation system has no coherent cache, the simple implementation of the first algorithm causes heavy shared-bus traffic. In order to avoid shared-bus saturation, the frequency to read the *maxprio* field in the *acquire_second_lock* routine is reduced. In more concrete, *maxprio* is checked only once for every four checkings of *me→locked*.

Evaluation Results

Figure 60 presents the 99.99%-reliable execution times that the *highest* priority processor executes routine (b). When the number of processors is large, the execution time with Markatos' locks, which can not be bounded inherently, is much slower than those with our algorithms due to uncontrolled priority inversions. When the number of processors is small, our algorithms are slower because of the overhead for maintaining the *maxprio* field of each lock. Our second algorithm is a bit faster than the first one when the number

[8]With routines (a) and (b) only, the effect of shared-bus traffic is not revealed, because at most one processor spins on non-local memory at the same time.

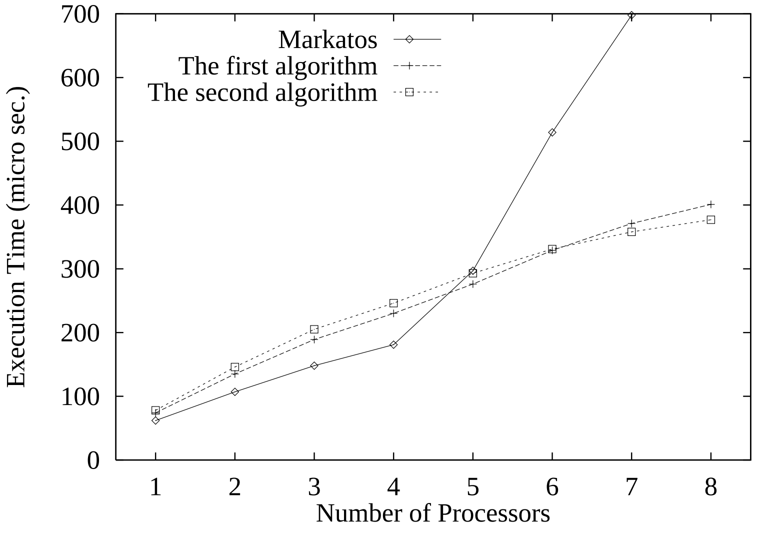


Figure 60: 99.99%-Reliable Execution Times of Routine (b)

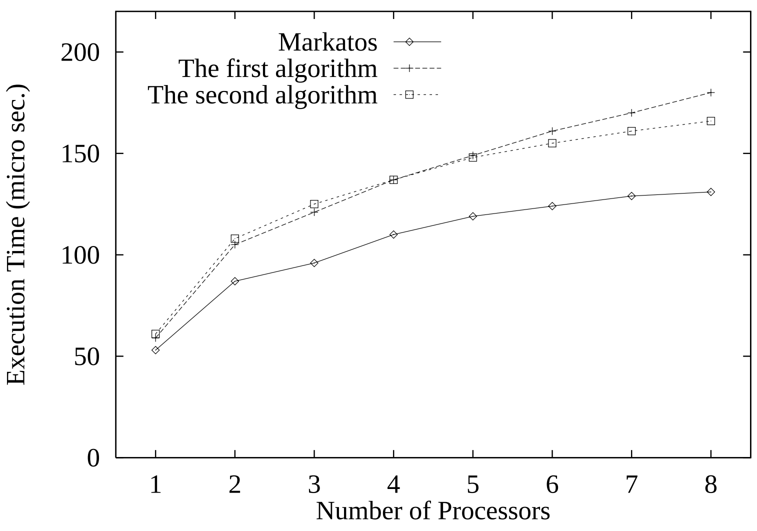


Figure 61: 99.99%-Reliable Execution Times of Routine (a)

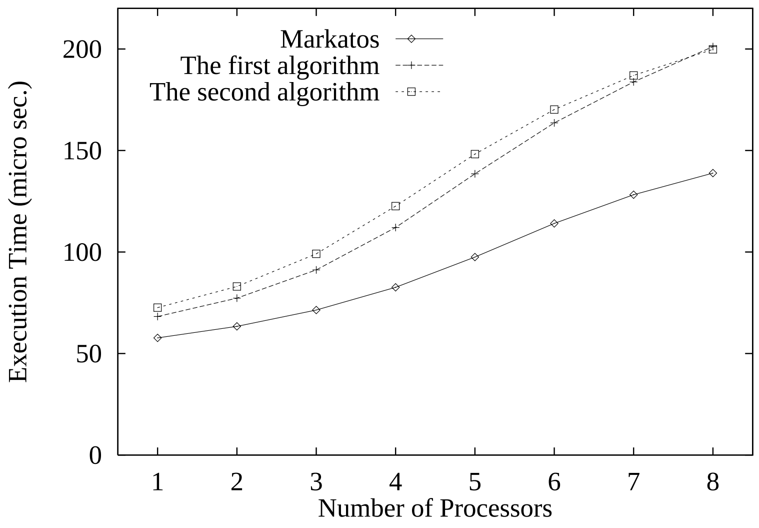


Figure 62: Average Execution Times of Routine (b)

of processors is more than six, but the difference is very small. Though it is not measured in our experiments, the shared-bus traffic is expected to be much larger with the first algorithm.

Figure 61 presents the 99.99%-reliable execution times that the highest priority processor executes routine (a). As easily imagined, there are little difference in the behavior of routine (a) with three algorithms. This graph confirms the conjecture.

Finally, in order to examine the average performance of the algorithms, we present the average execution times of routine (b) in Figure 62. From this graph, our algorithms are slower than Markatos' lock in average performance. We can say that priority inheritance spin locks are not appropriate when improving average performance is the primary concern.

6 Summary

In this part, we have proposed various spin lock algorithms with the properties required to implement scalable real-time kernels, and have evaluated their effectiveness through performance measurements. Before describing the algorithms, Section 1 has presented a brief survey on spin lock algorithms and has shown the pseudo-codes of some important algorithms on which our proposed algorithms are based.

In Section 2, we have proposed two algorithms of queueing spin lock with preemption that can give practical upper bounds on the times to acquire and release an inter-processor lock while realizing constant response to interrupt requests, in order to make the two important requirements for scalable real-time systems on function-distributed multiprocessors compatible. The first algorithm, which supports the simple preemption scheme, has a drawback that the interrupt service overhead depends on the number of contending processors. In order to solve the problem, we have proposed the second algorithm which adopts the improved preemption scheme. Their performance evaluation through experiments has confirmed that the algorithms have the required properties. We have also described a combined algorithm which supports both preemption schemes.

In Section 3, we have presented an efficient algorithm of spin lock with local precedence, which is required to make the worst-case execution times of intra-processor synchronizations independent of the number of contending processors.

In Section 4, real-time scalability of nested spin locks has been discussed. An algorithm with which the maximum execution times of critical sections are $O(n)$ when the maximum nesting level of locks in the system is two has been proposed, and its effectiveness is demonstrated with performance evaluation. By introducing the priority inheritance scheme to the algorithm, it can be applied to the system in which the maximum nesting level is more than two.

Though the section has focused on bounded spin locks (in other words, on the cases when each processor equally contends for nested spin locks ignoring the priority of the task it is executing), the results are also applicable to priority-ordered spin locks (the cases when each processor has its priority determined from the job it is executing). In concrete, when processors with the same priority should execute critical sections in a FIFO order, our proposed method should be utilized. In this case, a pair of the native (or assigned) priority of the processor and the time stamp obtained before trying to acquire the outermost lock should be used as the priority for acquiring inner locks.

In Section 5, we have discussed on priority inheritance spin locks. At first, we have pointed out the problem that the simple application of a priority-ordered spin lock algorithm to nested spin locks causes uncontrolled priority inversions, which are very harmful for satisfying the timing constraints imposed on real-time tasks. In order to solve the problem, we have incorporated the basic priority inheritance scheme to spin locks. Two algorithms of priority inheritance spin locks have been proposed based on the Markatos' spin lock algorithm: one for coherent cache multiprocessors and the other for multiprocessor systems without coherent cache. Performance evaluation to demonstrate their effectiveness has been conducted, and some affirmative results have been obtained.

In Section 4 and 5, we have adopted the Markatos' priority-ordered spin lock algorithm for the evaluations and for the base algorithm to which the basic priority inheritance scheme is incorporated. Doing the same thing with the PR-lock [26] remains as future work. Another important work remaining to be done is to combine the result of Section 2 and 5, in other words, to incorporate a preemption scheme to priority inheritance spin locks.

Part IV

Conclusion and Future Work

1 Conclusion

In this dissertation, we have discussed the specification and implementation issues of a scalable real-time kernel on function-distributed shared-memory multiprocessor systems. A scalable real-time kernel is the basic software module that facilitates the realization of scalable application systems. If a real-time system has the property of real-time scalability, even when a part of the system is modified or when some processors are added to the system, changes in the worst-case timing behavior of the unmodified part of the system are minimized, leading to the reduction of the maintenance cost of the system. Though many researchers have investigated on real-time kernels for shared-memory multiprocessors, none of them has focused on the issues of real-time scalability.

When a real-time system is realized on a function-distributed multiprocessor architecture, external devices and tasks handling them are allocated to processors so that the number of inter-processor synchronizations and communications is minimized and that as many time-critical tasks as possible are closed within a processor. Therefore, it is advantageous to reduce the maintenance cost of the system that changes in the worst-case timing behavior of the processings that can be done within a processor are minimized.

In this dissertation, we have clarified the required properties of a scalable real-time kernel for function-distributed multiprocessors and investigated on their realization methods. After describing the implementation approaches of a real-time kernel on shared-memory multiprocessors, two problems which are the obstacles for a straightforward implementation method to satisfy the required properties have been pointed out, and the solutions of the problems have been proposed when task-independent synchronization and communication objects are not supported. In order to solve the first problem that the worst-case execution times of synchronizations within a processor depend on the number of contending processors, spin lock with local precedence is adopted. For the second problem that predictable inter-processor synchronization and constant interrupt response are incompatible, bounded spin lock with preemption is devised. We have presented the algorithms of these two kind of spin locks assuming that processors support atomic read-modify-write operations on a single word of shared memory.

We have also proposed the approach to classify kernel resources into classes with different characteristics to improve the performance of intra-processor synchronization. In the concrete, tasks are classified into the isolated tasks, the private tasks, and the local tasks of each processor. Task-independent synchronization and communication objects are also classified into three classes: the isolated objects, the private objects, and the shared objects.

In order to demonstrate the effectiveness of our proposals, we have conducted performance measurements using an existing shared-bus multiprocessor system without coherent cache. The underlying inter-processor synchronization is realized with software-implemented spin locks. Although the hardware and the synchronization mechanism do not have the properties that is necessary to strictly satisfy the required properties of a scalable real-time kernel, the performance measurements have confirmed that the required properties are practically satisfied with our proposals, while they cannot be met with other methods.

In order to support task-independent synchronization and communication objects, nested spin locks are necessary. We have discussed on the scalability issues on nested spin locks and proposed the scheme for reducing the maximum execution times of nested spin locks to $O(n \cdot e^m)$, where n is the number of contending processors and m is the maximum nesting level of locks. Even though the scheme is adopted, however, the interrupt service overhead depends on the number of contending processors, and it is not possible to satisfy the required properties of a scalable real-time kernel.

2 Future Work

There are pretty much work to be tackled. The most pressing one is to realize a scalable real-time kernel that supports task-independent synchronization and communication objects by solving (or avoiding) the difficulty described in Section II.7. One of the possible approaches is to incorporate the notion of block-free or wait-free synchronizations to our real-time kernel implementation and to avoid nested spin locks. More precisely, the processings which needs the outer lock should be realized in a block-free or wait-free fashion [17, 37]. Because the manipulations of the TCBs and the ready queues are too complicated to realize in block-free or wait-free with reasonable performance, we think that the inner lock should be used even with this approach. This kind of mixed block-free and lock-based synchronization is a new research topic that has not been studied. Another promising approach is to adopt the realization concept of wait-free synchronization in acquring an inner lock. More precisely, the operation within the inner lock is posted to the waiting queue for the lock and is executed by another processor during an interrupt service.

Another important work to do is to extend our kernel model further to support global tasks which can migrate among processors. We will describe it in the next section (Section 2.1). There are some problems to be solved in implementing global tasks without degrading the performance of the other tasks [81].

Other future work includes the hardware implementation of the spin locks with which the maximum execution time can be determined independently of the number of contending processors, and further extensions of the spin lock algorithms which have been described in Section III.6. It is also necessary to extend our study to upper layers. For example, the design guidelines of scalable application systems on a scalable real-time kernel should be investigated on. The software development environment for our real-time kernel model, especially a tool that supports the fitting of the kernel resources to appropriate classes integrated with a schedulability analyzer, is also an important research topic.

Finally, applying our scalable real-time kernel to real applications and evaluating it in real-world environments are the most challenging work remaining to be done. To this end, we plan to port our scalable real-time kernel for shared-memory multiprocessors to other off-the-shelf hardware environments and distribute it in free.

2.1 Global Tasks

One of the advantages of shared-memory multiprocessors is that task migrations can be easily implemented. As described in Section II.2, time-critical tasks should be bound to a processor in function-distributed multiprocessor systems. On the other hand, task migrations are useful for background jobs without severe timing constraints. We call the class of tasks that can execute on any processors in the system and that can migrate to other processors during their execution as *global tasks*.

One of the most import issues on global tasks is their scheduling method. Because global tasks are introduced to support background jobs without severe timing constraints, we handle the priorities of global tasks always lower than those of local tasks.

Two shared task queues are prepared for global tasks: the ready queue that includes all global tasks that are ready to execute but are not being executed, and the run queue that includes all global tasks that are being executed. When no task of the other classes is ready to execute on a processor, the task dispatcher on the processor removes the highest priority task from the ready queue for the global tasks, and moves it to the run queue. When a processor makes a global task τ_1 ready to execute, it first finds the lowest priority task τ_2 in the run queue. If τ_1 has a higher priority than τ_2, the processor moves τ_2 to the ready queue and inserts τ_1 to the run queue instead. Then, it requests the processor that is executing τ_2 to switch the executing task using an inter-processor interrupt.

Here, a difficulty occurs when a private (or isolated) task becomes ready to execute with an external event on a processor P_1 that is executing a global task. In this case, the global task is preempted and should migrate to another processor that is executing

accessing task \ **accessed resource**	P_1-isolated task	P_1-isolated object	P_1-private task	P_1-private object	P_1-local task	shared object	global task	P_2-local task	P_2-private object	P_2-private task	P_2-isolated object	P_2-isolated task
P_1-isolated task	OK	OK	OK	NA	NA	NA	NA	NA	NA	NA	NA	NA
P_1-private task	OK	OK	OK	OK	*1	NA	NA	NA	NA	NA	NA	NA
P_1-local task	OK	*2	OK	OK	OK	OK	OK	*1	NA	NA	NA	NA
global task	NA	NA	NA	NA	*1	OK	OK	*1	NA	NA	NA	NA

Table 10: Accessibility of Kernel Resources (Full Set)

a lower priority task or is idle. Because the maximum processing time on P_1 for the migration unavoidably depends on the number of contending processors, the maximum response time of the private (or isolated) task becomes long as the number of contending processors is increased. In order to avoid this problem, we allow the situation that a global task is bound to a processor while it is executing a private (or isolated) task, just like when it is executing an interrupt handler. When the execution times of private tasks are relatively short compared to the deadlines of global tasks, this restriction is considered to be reasonable.

The accessibility of kernel resources with global tasks are summarized in Table 10. Because the control blocks of isolated and private resources on a processor cannot be accessed from other processors, a global task, which can be executed on any processor, cannot operate on them. A global task cannot access a P_1-local task with special operations, because the global task cannot access the control block of a P_1-private object on which the local task may be waiting.

Another possible extension is to support the class of tasks that can be executed on a predefined set of processors. For example, suppose a heterogeneous multiprocessor architecture, in which some general-purpose microprocessors and some special-purpose processors (e.g. DSPs) are adopted. It is very natural to support the class of tasks that can be executed only on the general-purpose microprocessors. Note here that it is not necessary to implement all the resource classes in a kernel. It is also a possible approach that some of the classes are removed from a full-set kernel when they are not used.

Bibliography

[1] G. Ahmed and K. Schwan, “CHAOS^{arc}: kernel support for multiweight objects, invocations, and atomicity in real-time multiprocessor applications,” *ACM Trans. Computer Systems*, vol. 11, pp. 33–72, Feb. 1993.

[2] R. Alur and G. Taubenfeld, “Results about fast mutual exclusion,” in *Proc. Real-Time Systems Symposium*, pp. 12–21, Dec. 1992.

[3] T. E. Anderson, “The performance of spin lock alternatives for shared-memory multiprocessors,” *IEEE Trans. Parallel and Distributed Systems*, vol. 1, pp. 6–16, Jan. 1990.

[4] T. P. Baker, “Stack-based scheduling of realtime processes,” *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[5] J. E. Burns, “Mutual exclusion with linear waiting using binary shared variables,” *SIGACT News*, vol. 10, no. 2, pp. 42–47, 1978.

[6] E. M. Chaves, Jr., P. C. Das, T. J. LeBlanc, B. D. Marsh, and M. L. Scott, “Kernel-kernel communication in a shared-memory multiprocessor,” Tech. Rep. TR 368, Computer Science Department, University of Rochester, 1991.

[7] E. M. Chaves, Jr., P. C. Das, T. J. LeBlanc, B. D. Marsh, and M. L. Scott, “Kernel-kernel communication in a shared-memory multiprocessor,” *Concurrency: Practice and Experience*, vol. 5, pp. 171–191, May 1993.

[8] T. S. Craig, “Building fifo and priority-queuing spin locks from atomic swap,” Tech. Rep. 93-02-02, Department of Computer Science and Engineering, University of Washington, Feb. 1993.

[9] T. S. Craig, “Queuing spin lock algorithms to support timing predictability,” in *Proc. Real-Time Systems Symposium*, pp. 148–157, Dec. 1993.

[10] E. W. Dijkstra, “Solution of a problem in concurrent programming control,” *Communications of the ACM*, vol. 8, p. 569, Sept. 1965.

[11] M. A. Eisenberg and M. R. McGuire, “Futher comments on dijkstra’s concurrent programming control problem,” *Communications of the ACM*, vol. 15, p. 999, Nov. 1972.

[12] W. M. Gentleman, S. A. MacKay, D. A. Stewart, and M. Wein, “An introduction to the Harmony realtime operating system,” *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, 1988.

[13] J. R. Goodman, M. K. Vernon, and P. J. Woest, “Efficient synchronization primitives for large-scale cache-coherent multiprocessors,” in *Proc. 3rd Int’l Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 64–75, Apr. 1989.

[14] G. Graunke and S. Thakkar, “Synchronization algorithms for shared-memory multiprocessors,” *IEEE Computer*, vol. 23, pp. 60–69, June 1990.

[15] W. A. Halang, “Real-time systems: Another perspective,” *Journal of System and Software*, pp. 101–108, Apr. 1992.

[16] M. Herlihy, “Impossibility and universality results for wait-free synchronization,” in *Proc. Seventh ACM Symposium on Principles of Distributed Computing*, pp. 276–290, Aug. 1988.

[17] M. Herlihy, “Wait-free synchronization,” *ACM Trans. Programming Languages and Systems*, vol. 13, pp. 124–149, Jan. 1991.

[18] M. Herlihy, “A methodology for implementing highly concurrent data objects,” *ACM Trans. Programming Languages and Systems*, vol. 15, pp. 745–770, Nov. 1993.

[19] A. L. Hopkins, Jr., T. B. Smith III, and J. H. Lala, “FTMP – a highly reliable fault-tolerant multiprocessor for aircraft,” *Proc. IEEE*, vol. 66, pp. 1221–1239, Oct. 1978.

[20] IBM, *The IBM PowerPC Architecture – A New Family of RISC Processors*. San Mateo, California: Morgan Kaufmann, 1994.

[21] IEEE, *IEEE Standard for a Versatile Backplane Bus: VMEbus*, 1987. ANSI/IEEE Std 1014-1987.

[22] Y. Igarashi, M. Joh, T. Hirai, K. Kawai, and K. Kawanishi, “Realization of CTRON based kernel for tightly coupled multi-processor system,” in *Proc. TRON Technical Workshop*, vol. 3, no. 3, pp. 15–27, TRON Association, Feb. 1991. (in Japanese).

[23] H. Inayoshi, I. Kawasaki, T. Nishimukai, and K. Sakamura, “Realization of GMICRO/200,” *IEEE Micro*, vol. 8, pp. 12–21, Apr. 1988.

[24] M. Itoh, “Architecture characteristics of GMICRO/300,” in *TRON Project 1987*, pp. 273–280, Springer-Verlag, 1987.

[25] M. Joh, Y. Igarashi, and T. Ozeki, “CTRON-specification kernel implementation for a tightly coupled multiprocessor system,” in *Proc. 8th TRON Project Symposium*, pp. 118–129, IEEE CS Press, 1991.

[26] T. Johnson and K. Harathi, “A prioritized multiprocessor spin lock,” Tech. Rep. TR-93-005, Department of Computer Science, University of Florida, 1993.

[27] T. Johnson and K. Harathi, “A simple correctness proof of the mcs contention-free lock,” *Information Processing Letters*, vol. 48, pp. 215–220, 1993.

[28] G. Kane and J. Heinrich, *MISP RISC Architecture*. Prentice Hall, 1992.

[29] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[30] D. E. Knuth, “Additional comments on a problem in concurrent programming control,” *Communications of the ACM*, vol. 9, pp. 321–322, May 1966.

[31] L. Kontothanassis, R. W. Wisniewski, and M. L. Scott, “Schedular-conscious synchronization,” Tech. Rep. TR550, Computer Science Department, University of Rochester, Dec. 1994.

[32] L. Lamport, “A new solution of dijkstra's concurrent programming problem,” *Communications of the ACM*, vol. 17, pp. 453–455, Aug. 1974.

[33] L. Lamport, “A fast mutual exclusion algorithm,” *ACM Trans. Computer Systems*, vol. 5, no. 1, pp. 1–11, 1987.

[34] J. Lee and U. Ramachandran, “Synchronization with multiprocessor cache,” in *Proc. 17th Int'l Symposium on Computer Architecture*, pp. 27–37, 1990.

[35] N. Lynch and N. Shavit, "Timing-based mutual exclusion," in *Proc. Real-Time Systems Symposium*, pp. 2–11, Dec. 1992.

[36] E. P. Markatos, "Multiprocessor synchronization primitives with priorities," in *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[37] H. Massalin and C. Pu, "A lock-free multiprocessor os kernel," Tech. Rep. CUCS-005-91, Department of Computer Science, Columbia University, 1991.

[38] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Computer Systems*, vol. 9, pp. 21–65, Feb. 1991.

[39] J. M. Mellor-Crummey and M. L. Scott, "Scalable reader-writer synchronization for shared-memory multiprocessors," in *Proc. Third ACM Symposium on Principle and Practice of Parallel Programming*, pp. 106–113, Apr. 1991.

[40] J. M. Mellor-Crummey and M. L. Scott, "Synchronization without contention," in *Proc. Fourth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 269–278, Apr. 1991.

[41] M. M. Michael and M. L. Scott, "Fast mutual exclusion, even with contention," Tech. Rep. TR460, Computer Science Department, University of Rochester, June 1993.

[42] M. M. Michael and M. L. Scott, "Scalability of atomic primitives on distributed shared memory multiprocessors," Tech. Rep. TR528, Computer Science Department, University of Rochester, July 1994.

[43] L. D. Molesky, K. Ramamritham, C. Shen, J. A. Stankovic, and G. Zlokapa, "Implementing a predictable real-time multiprocessor kernel – the Spring kernel," in *Proc. Real-Time Operating Systems and Software*, May 1990.

[44] L. D. Molesky, C. Shen, and G. Zlokapa, "Predictable synchronization mechanisms for multiprocessor real-time systems," *Real-Time Systems*, vol. 2, no. 3, pp. 163–180, 1990.

[45] H. Monden, "Introduction to ITRON, the industry-oriented operating system," *IEEE Micro*, vol. 7, pp. 45–52, Apr. 1987.

[46] R. Rajkumar, “Real-time synchronization protocols for shared memory multiprocessors,” in *Proc. Int’l Conf. Distributed Computing Systems*, pp. 116–123, May 1990.

[47] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[48] R. Rajkumar, L. Sha, and J. P. Lehoczky, “Real-time synchronization protocols for multiprocessors,” in *Proc. Real-Time Systems Symposium*, pp. 259–269, Dec. 1988.

[49] M. Rozier, V. Abrosimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Le’onard, and W. Neuhauser, “Overview of the Chorus distributed operating system,” in *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 39–70, Apr. 1992.

[50] K. Sakamura, “ITRON: An overview,” in *TRON Project 1987*, pp. 29–34, Springer-Verlag, 1987.

[51] K. Sakamura, “The objectives of the TRON project,” in *TRON Project 1987*, pp. 3–16, Springer-Verlag, 1987.

[52] K. Sakamura, ed., *μITRON Specification*. Tokyo: TRON Association, 1989.

[53] K. Sakamura, ed., *Specification of the Chip Based on the TRON Architecture*. Tokyo: TRON Association, 1989.

[54] K. Sakamura, ed., *ITRON Specification ITRON2*. Tokyo: TRON Association, 1990.

[55] K. Sakamura, “After a decade of TRON, what comes next?,” in *Proc. 11th TRON Project Int’l Symposium*, pp. 2–16, IEEE CS Press, Dec. 1994.

[56] K. Sakamura, ed., *μITRON 3.0 Specification*. Tokyo: TRON Association, 1994. (can be obtained from “ftp://tron.um.u-tokyo.ac.jp/pub/TRON/ITRON/SPEC/mitron3.txt.Z”).

[57] N. Sakiyama, H. Takada, and K. Sakamura, “Bubble lock: Another priority-orderd spin lock algorithm,” in *Collection of Position Papers for the 2nd Youth Forum in Computer Science and Engineering (YUFORIC)*, Oct. 1995.

[58] K. Sato, H. Tsubota, O. Yamamoto, and K. Saitoh, “An experimental implementation of unified real-time operating system,” in *Proc. 8th TRON Project Symposium*, pp. 57–68, IEEE CS Press, 1991.

[59] K. Schwan, P. Gopinath, and W. Bo, “CHAOS – kernel support for objects in the real-time domain,” *IEEE Trans. Computer*, vol. 36, pp. 904–916, Aug. 1987.

[60] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *IEEE Trans. Computers*, vol. 39, pp. 1175–1185, Sept. 1990.

[61] R. L. Sites, ed., *Alpha Architecture Reference Manual*. Burlington, Massachusetts: Digital Press, 1992.

[62] J. A. Stankovic, “Misconceptions about real-time computing,” *IEEE Computer*, vol. 21, pp. 10–19, Oct. 1988.

[63] J. A. Stankovic and K. Ramamritham, “The design of the Spring kernel,” in *Proc. Real-Time Systems Symposium*, pp. 146–157, Dec. 1987.

[64] J. A. Stankovic and K. Ramamritham, eds., *Hard Real-Time Systems*. IEEE CS Press, 1988.

[65] J. A. Stankovic and K. Ramamritham, “Editorial: What is predictability for real-time systems?,” *Real-Time Systems*, vol. 2, no. 4, pp. 247–254, 1990.

[66] J. A. Stankovic and K. Ramamritham, “The Spring kernel: A new paradigm for real-time systems,” *IEEE Software*, pp. 62–72, May 1991.

[67] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, “The Chimera II real-time operating system for advanced sensor-based robotic applications,” *IEEE Trans. Systems, Man, and Cybernetics*, vol. 22, pp. 1282–1295, Nov. 1992.

[68] J. M. Stone and R. P. Fitzgerald, “Storage in the PowerPC,” *IEEE Micro*, vol. 15, pp. 50–58, Apr. 1995.

[69] H. Takada, “ItIs: A μITRON3.0-specification kernel,” *TRON PROJECT BIMONTHLY*, vol. 32, pp. J9–J11,E11–E14, Apr. 1994. (both in Japanese and English).

[70] H. Takada, “An update on development of ItIs, a μITRON3.0-specification OS,” *TRON PROJECT BIMONTHLY*, vol. 38, pp. J8–J11,E9–E12, Apr. 1995. (both in Japanese and English).

[71] H. Takada and K. Sakamura, “Implementation of inter-processor synchronization/communication and design issues of ITRON-MP,” in *Proc. 8th TRON Project Symposium*, pp. 44–56, IEEE CS Press, Nov. 1991.

[72] H. Takada and K. Sakamura, “ITRON-MP: An adaptive real-time kernel specification for shared-memory multiprocessor systems,” *IEEE Micro*, vol. 11, pp. 24–27,78–85, Aug. 1991.

[73] H. Takada and K. Sakamura, “Advances in the ITRON specifications – supporting multiprocessor and distributed systems,” in *Proc. 9th TRON Project Symposium*, pp. 89–95, IEEE CS Press, 1992.

[74] H. Takada and K. Sakamura, “A bounded spin lock algorithm with preemption,” Tech. Rep. 93-2, Department of Information Science, University of Tokyo, July 1993.

[75] H. Takada and K. Sakamura, “Experimental implementations of priority inheritance semaphore on ITRON-specification kernel,” in *Proc. 11th TRON Project Int’l Symposium*, pp. 106–113, IEEE CS Press, Dec. 1994.

[76] H. Takada and K. Sakamura, “Predictable spin lock algorithms with preemption,” in *Proc. Real-Time Operating Systems and Software*, pp. 2–6, May 1994.

[77] H. Takada and K. Sakamura, “Compact, low-cost, but real-time distributed computing for computer augmented environments,” in *Proc. 5th IEEE CS Workshop on Future Trends of Distributed Computing Systems*, pp. 56–63, IEEE CS Press, Aug. 1995.

[78] H. Takada and K. Sakamura, “μITRON for small-scale embedded systems,” *IEEE Micro*, vol. 15, pp. 46–54, Dec. 1995.

[79] H. Takada and K. Sakamura, “Queueing spin lock algorithms with preemption,” *Trans. IEICE (D-I)*, vol. J78-D-I, pp. 661–669, Aug. 1995. (in Japanese).

[80] H. Takada and K. Sakamura, “Real-time scalability of nested spin locks,” in *Proc. 2nd Real-Time Computing Systems and Applications*, pp. 160–167, Oct. 1995.

[81] H. Takada and K. Sakamura, “Towards a scalable real-time kernel for asymmetric multiprocessor systems,” in *IEICE Technical Report (RTP’95)*, vol. 94, no. 573, pp. 1–8, IEICE, Mar. 1995. (in Japanese).

[82] H. Takada and K. Sakamura, “Towards a scalable real-time kernel for function-distributed multiprocessors,” in *Proc. 20th IFAC/IFIP Workshop on Real Time Programming*, Nov. 1995.

[83] H. Takada and K. Sakamura, “Inter- and intra-processor synchronizations in multiprocessor real-time kernel,” in *Proc. 4th Int’l Workshop on Parallel and Distributed Real-Time Systems*, pp. 69–74, Apr. 1996.

[84] H. Takada and K. Sakamura, “Scalable implementations of multiprocessor real-time kernels,” in *IEICE Technical Report (RTP’96)*, vol. 95, no. 603, pp. 1–6, IEICE, Mar. 1996. (in Japanese).

[85] H. Takada, K. Tamaru, K. Kudou, T. Shimizu, and H. Tsubota, “The present and future of the ITRON subproject – kernel specifications and their implementation –,” *Journal of IPSJ*, vol. 35, pp. 903–909, Oct. 1994. (in Japanese).

[86] K. Takagi, T. Nishimukai, K. Iwasaki, I. Kawasaki, and H. Inayoshi, “Outline of GMICRO/200 and memory management mechanism,” in *TRON Project 1987*, pp. 259–272, Springer-Verlag, 1987.

[87] T. Takahashi, N. Ito, R. Hayano, Y. Watanabe, K. Sakamura, H. Takada, H. Nakamura, and N. Nishio, “Development of the parallel computer system using the TRON chips and its application to physics,” in *Proc. TRON Technical Workshop*, vol. 2, no. 1, pp. 21–30, TRON Association, Apr. 1989. (in Japanese).

[88] N. Wirth, *Programming in Modula-2*. Springer-Verlag, 3rd corrected edition ed., 1985.

[89] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott, “Scalable spin locks for multiprogrammed systems,” in *Proc. 8th Int’l Parallel Processing Symposium*, Apr. 1994.

Appendix A

Implementation Details of our Real-Time Kernel

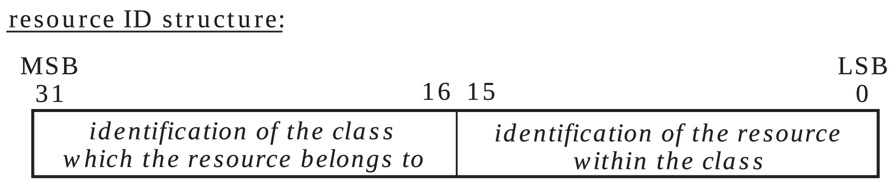
In this appendix, we present the implementation details of our real-time kernel, which is used for the evaluation in Section II.6. We have extended ItIs, a μITRON3.0-specification real-time kernel described in Section II.1.4, to support shared-memory multiprocessors. We call the extended version of ItIs as ItIs/MP in this appendix.

1 Management of Classes

The largest difference between ItIs and ItIs/MP is that ItIs/MP supports the classification of kernel resources. In order to manage the classes, a *class control block* is prepared for each class of resources. Though the classification of tasks and that of task-independent synchronization and communication objects have a bit difference, we have prepared four type of classes in which both tasks and task-independent objects are included: the isolated classes, the private classes, the local classes, and the global class. In the current implementation, only the private classes and the local classes are realized.

As described in Section II.5.4, the ID of a kernel resource is divided into the field indicating the class ID to which the resource belongs and the field identifying the resource within the class (Figure 63). The actual assignment of class IDs is also presented in Figure 63. The class ID 0 designates the same class with the task that uses the ID. For example, if a P_1-local task operates on the object with the ID number 0x00000052, it designates a P_1-local object whose identification number within the class is 0x52.

When a task operates on a kernel resource with its resource ID, the task first extracts the class ID field within the resource ID and finds the address of its class control block. The class control block includes the range of valid identification numbers of each resource type within the class, and the address of the control block table of each resource type


resource ID structure:
MSB
LSB
31
16
15
0
identification of the class which the resource belongs to
identification of the resource within the class

class ID assignment:

-2 (0xfffe)	the isolated class of the processor executing the issuing task
-1 (0xffff)	the private class of the processor executing the issuing task
0	the same class with the issuing task
$1 \cdots n$	the local classes of each processor
n+1	the global class

n : the maximum number of processors

Figure 63: The Structure of Resource ID

(Figure 64). The address of the ready queue of the class and that of the timer event queue are also included in the class control block. A class control block also includes two *lock objects*, one of which guards the TCBs of the class and the other guards the control blocks of task-independent synchronization and communication objects. Using the information, the task can find the address of the control block of the kernel resource and operate on it.

Though it is possible to prepare one set of class control blocks and share it by all the processors, we adopt another approach with which each processor has its own set of control blocks in order to reduce the shared-bus traffic (remember that our evaluation environment has no coherent cache). The class control blocks for each processor are initialized from the *shared class control blocks* (Figure 64). When the class control blocks are initialized, each processor customizes their contents.

2 Initialization Procedure

Booting up a multiprocessor system is a bit complicated procedure. We follow the following three initialization steps to boot up the system.

1. At first, the kernel program code is downloaded to the master processor,[1] and is started execution on the master processor. The master processor clears the shared class control blocks and other globally shared variables. Then, it distribute the kernel code to each processor and makes it start with an inter-processor interrupt. If the local memory of a processor cannot be accessed, the master processor judges that the processor is not available in the system.

[1]In the current implementation, we assume that each processor executes the same kernel code.

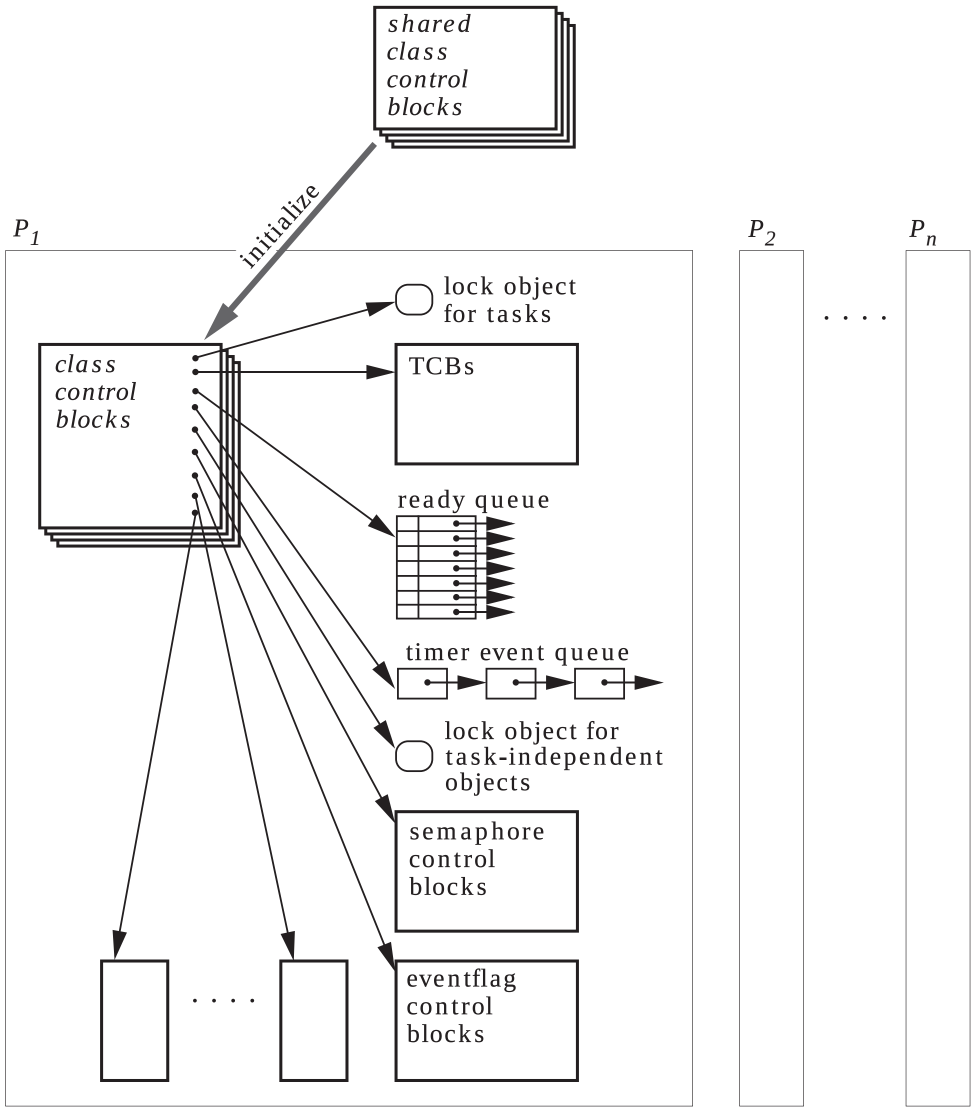


Figure 64: Class Control Blocks and Shared Class Control Blocks

2. Each processor (including the master processor) initializes its local and private variables, such as the control blocks of its local and private resources and the ready queues for its local and private tasks. It also initializes the shared class control block of its local class.

   When a processor finishes this step, it notifies the master processor of it via a shared variable and begins waiting. The master processor repeatedly checks if other processors finish this step. When all the processors finish this step, the master processor signals the other processors to proceed to the next step via a shared variable.

3. Each processor reads the shared class control blocks and initializes its own class control blocks. It also initializes the class control block for its private class.[2] Then, it starts executing tasks if some of the tasks are ready to execute.

3 Spin Locks Used in the Implementation

In the current implementation of our real-time kernel, a combined algorithm of the queueing spin lock with improved preemption scheme presented in Figure 28–30 and the spin lock with local precedence presented in Figure 38–39 is used with some improvements.

One of the improvements is that the processor trying to acquire the lock begins executing the critical section when its state becomes *Dequeueing*. If it remains to be *Dequeueing* when the processor tries to release the lock, it waits until the state becomes *Released*. Another improvement is that the global lock has now three states: the state in which the global lock is released, the state in which the global lock is not released and a processor must repeatedly check the global lock, and the state in which the global lock is not released and a processor need not check the global lock.

Pseudo-code for the combined algorithm is presented in Figure 65, 66, 67, 68, and 69. In the pseudo-code, *NADR* designates a special pointer value that has a different value with the other pointers, just like *NULL*. *NADR* is used with *NULL* to distinguish the new state introduced in the global lock. Actually, 0 is assigned to *NULL* and -1 to *NADR* in our implementation.

A processor should use *acquire_my_local_lock* and *release_my_local_lock* to acquire/release its local lock, and should use *acquire_lock* and *release_lock* to acquire/release the local locks of other processors. The *acquire_lock* and *acquire_my_local_lock* functions must be called with the interrupt request disabled. They return *TRUE* when they succeed to acquire the lock and return *FALSE* when an interrupt is requested while waiting for the lock. When *FALSE* is returned from these functions, the processor must enable interrupt request, service the interrupt request, and re-execute the function. In the *acquire_my_local_lock* function, the exponential backoff scheme is not adopted because the *glock* field of the lock is located on the local memory of the processor that issues the function.

The lock object, which is included in the class control block, includes the pointer to the *Lock* record, the memory area for its queue node (the *Node* record), and the pointers to the functions with which the lock should be acquired/released.

[2] Actually, we include this initialization in the second step.

```
type Node = record
    next: pointer to Node;
    prev: pointer to Node;
    locked: (Released, Locked, Preempted, Dequeueing)
end;
// The locked field must be intialized to NULL.

type Lock = record
    last: pointer to Node;
    glock: pointer to Node;
    prec: pointer to Node
end;

shared var L: Lock;
// L.last, L.glock, and L.prec are initialized to NULL.

procedure move_to_top(lock: pointer to Lock,
                      entry, pred, oldtop: pointer to Node);
// move entry to the top of the waiting queue of lock.
// pred is the predecessor of entry.
// oldtop is the top of the queue before the move.
    var succ: pointer to Node;
begin
    succ := entry→next;
    if succ = NULL then
        // when succ is at the tail of the waiting queue.
        pred→next := NULL;
        if compare_and_swap(&(lock→last), entry, pred) then
            entry→next := oldtop;
            return
        end;
        repeat succ := entry→next until succ ≠ NULL
    end;
    pred→next := succ;
    succ→prev := pred;
    entry→next := oldtop
end;
```

Figure 65: The Spin Lock Used in the Implementation (Part 1)

```
procedure acquire_lock(lock: LockPtr, me: NodePtr): boolean;
    var pred, succ: NodePtr;
    var interval, i: integer;
begin
    if me→locked = Preempted then
        me→locked := Locked;
        goto spin
    end;
    me→next := NULL;
    pred := fetch_and_store(&(lock→last), me);
    if pred = NULL then
        return TRUE
    end;
    me→prev := pred;
    me→locked := Locked;
    pred→next := me;
  spin:
    i := 1;
    interval := $\alpha$;
    while (me→locked = Locked) do
        if interrupt_requested and
                compare_and_swap(&(me→locked), Locked, Preempted) then
            return FALSE
        end;
        i := i – 1;
        if i = 0 then
            top := lock→glock;
            if top = NULL then
                i := $\infty$              // never expires.
            else if top $\neq$ NADR
                    and compare_and_swap(&(lock→glock), top, NULL) then
                if top $\neq$ me then
                    move_to_top(lock, me, me→prev, top)
                end;
                me→locked := Released;
                return TRUE
            else
                i := interval;
                interval := interval $\times$ $\beta$
            end
        end
    end
end;
```

Figure 66: The Spin Lock Used in the Implementation (Part 2)

```
procedure acquire_my_local_lock(lock: LockPtr, me: NodePtr): boolean;
    var pred, succ: NodePtr;
    var check_glock: boolean;
begin
    if me→locked = Preempted then
        me→locked := Locked;
        goto spin
    end;
    me→next := NULL;
    pred := fetch_and_store(&(lock→last), me);
    if pred = NULL then
        return TRUE
    end;
    me→prev := pred;
    me→locked := Locked;
    pred→next := me;
  spin:
    lock→prec = me;
    check_glock = TRUE;
    while (me→locked = Locked) do
        if interrupt_requested and
                compare_and_swap(&(me→locked), Locked, Preempted) then
            lock→prec = NULL;
            return FALSE
        end;
        if check_glock then
            top := lock→glock;
            if top = NULL then
                check_glock = FALSE
            else if top ≠ NADR
                    and compare_and_swap(&(lock→glock), top, NULL) then
                if top ≠ me then
                    move_to_top(lock, me, me→prev, top)
                end;
                me→locked := Released;
                lock→prec = NULL;
                return TRUE
            end
        end
    end
end;
```

Figure 67: The Spin Lock Used in the Implementation (Part 3)

```
procedure release_lock(lock: LockPtr, me: NodePtr);
    var top, entry, pred: NodePtr;
begin
    repeat until me→locked = Released;
    top := me→next;
    if top = NULL then
        if compare_and_swap(&(lock→last), me, NULL) then
            return
        end;
        repeat top := me→next until top ≠ NULL
    end;
    entry := lock→prec;
    if entry ≠ NULL
            and compare_and_swap(&(entry→locked), Locked, Dequeueing) then
        if entry ≠ top then
            move_to_top(lock, entry, entry→prev, top)
        end;
        entry→locked := Released;
        return
    end;
    repeat until lock→glock = NULL;
    lock→glock := NADR;
    if compare_and_swap(&(top→locked), Locked, Released) then
        lock→glock := NULL;
        return
    end;
    pred := top;
    entry := pred→next;
    while entry ≠ NULL then
        if compare_and_swap(&(entry→locked), Locked, Dequeueing) then
            lock→glock := NULL;
            move_to_top(lock, entry, pred, top);
            entry→locked := Released;
            return
        end;
        pred := entry;
        entry := pred→next
    end;
    lock→glock := top
end;
```

Figure 68: The Spin Lock Used in the Implementation (Part 4)

```
procedure release_my_local_lock(lock: LockPtr, me: NodePtr);
    var top, entry, pred: NodePtr;
begin
    repeat until me→locked = Released;
    top := me→next;
    if top = NULL then
        if compare_and_swap(&(lock→last), me, NULL) then
            return
        end;
        repeat top := me→next until top $\neq$ NULL
    end;
    repeat until lock→glock = NULL;
    lock→glock := NADR;
    if compare_and_swap(&(top→locked), Locked, Released) then
        lock→glock := NULL;
        return
    end;
    pred := top;
    entry := pred→next;
    while entry $\neq$ NULL then
        if compare_and_swap(&(entry→locked), Locked, Dequeueing) then
            lock→glock := NULL;
            move_to_top(lock, entry, pred, top);
            entry→locked := Released;
            return
        end;
        pred := entry;
        entry := pred→next
    end;
    lock→glock := top
end;
```

Figure 69: The Spin Lock Used in the Implementation (Part 5)

Appendix B

Proofs on the Queueing Spin Lock Algorithm with Simple Preemption Scheme

In this appendix, we show that the queueing spin lock algorithm with the simple preemption scheme described in Section III.2 realizes mutual exclusion and deadlock freedom.

We first show that the algorithm in Figure 70 and 71 realizes mutual exclusion. The difference between the algorithm and the one in Figure 26 and 27 is (1) the initial value of the *locked* field is determined to be *Released* and (2) compare_and_swap operations are used in assigning *Released* to the *locked* field of queue nodes (in the lines marked with ⑯ and ⑱). Next, we show that the algorithm is deadlock free. Once mutual exclusion and deadlock freedom are proved, the equivalence of these two algorithms is straightforward.

At first, the state of a processor is classified into nineteen states by the execution point of the processor, which is presented in Figure 70 and 71 as ①–⑲. A state transition occurs when the processor accesses a shared data, with which the processor interacts with others. For example, the transition from ① to ② occurs when the processor reads *I.next*. Similarly, the transition from ② to ③ or ⑨ occurs when the processor executes the fetch_and_store operation. Whether the processor moves to ③ or ⑨ is fixed at this moment. The only exception is the transition from ⑲ to ⑫ which occurs when the processor modifies its private variable *succ*.

The state of a processor is also classified by the value of the *locked* field of its queue node into the released state (R state, in short), the locked state (L state), the preempted state (P state), and the canceled state. The canceled state is further classified into two states: the state that the variable *L* is kept non-*NULL* after *Canceled* is assigned to the *locked* field (C state), and the state after *L* becomes *NULL* (C’ state).

```
    type Node = record
        next: pointer to Node;
        locked: (Released, Locked, Preempted, Canceled)
    end;
    type Lock = pointer to Node;

    shared var L: Lock;
    // L is initialized to NULL.

    var I: Node;
    // I.locked is initialized to Released
    var pred, succ, sn: pointer to Node;

    // try to acquire the lock L.
  retry:
①   I.next := NULL;
    disable_interrupts;
    // enqueue myself.
②   pred := fetch_and_store(&L, &I);
    if pred ≠ NULL then
        // when the queue is not empty.
    ③   I.locked := Locked;
    ④   pred→next := &I;
    ⑤   while (I.locked ≠ Released) do
            if interrupt_requested and
                ⑥ compare_and_swap(&(I.locked), Locked, Preempted) then
                enable_interrupts;
                // interrupt service.
                disable_interrupts;
            ⑦   if ¬compare_and_swap(&(I.locked), Preempted, Locked) then
                    enable_interrupts;
                ⑧   repeat while I.locked ≠ Released;
                    goto retry
                end
            end
        end
    end;
    //
⑨   // critical section.
    //
```

Figure 70: The Queueing Lock with Simple Preemption Scheme (Part 1)

```
    //
⑨  // critical section.
    //
    // try to release the lock $L$.
    succ := I.next;
    **if** succ = NULL **then**
     ⑩ **if** compare_and_swap(&L, &I, NULL) **then**
            // the queue becomes empty.
            **goto** exit
        **end**;
     ⑪ **repeat** succ := I.next **until** succ $\neq$ NULL
    **end**;
    // try to pass the lock to the successor.
⑫  **while** $\neg$compare_and_swap(&(succ$\rightarrow$locked), Locked, Released) **do**
        // when the successor is servicing interrupts.
     ⑬ **if** compare_and_swap(&(succ$\rightarrow$locked), Preempted, Canceled) **then**
            // dequeue the successor from the waiting queue.
         ⑭ sn := succ$\rightarrow$next;
            **if** sn = NULL **then**
             ⑮ **if** compare_and_swap(&L, succ, NULL) **then**
                    // the queue becomes empty.
                 ⑯ compare_and_swap(&(succ$\rightarrow$locked), Canceled, Released);
                    **goto** exit
                **end**;
             ⑰ **repeat** sn := succ$\rightarrow$next **until** sn $\neq$ NULL
            **end**;
         ⑱ compare_and_swap(&(succ$\rightarrow$locked), Canceled, Released);
         ⑲ succ := sn
        **end**
    **end**;
  exit:
    *enable_interrupts*;
```

Figure 71: The Queueing Lock with Simple Preemption Scheme (Part 2)

The state transition diagram of a processor presented in Figure 72 can be obtained from these two classifications and some observations of the code in Figure 70 and 71 such as the fact that a processor assigns *Locked* to the *locked* field of its queue node with the transition from ③ to ④, the fact that a processor changes the *locked* field of another processor only from *Locked* to *Released*, from *Preempted* to *Canceled*, and from *Canceled* to *Released*, and the fact that the transition from C' state to C state does not exist by definition.[1] The transitions marked with "$*$" in the diagram are caused by other processors, and the transition with "#" occurs only when an interrupt request is raised on the processor.

[1]Following discussions reveal two other facts that a processor never becomes 4R state and that the transition from 7P to 7C' does not occur.

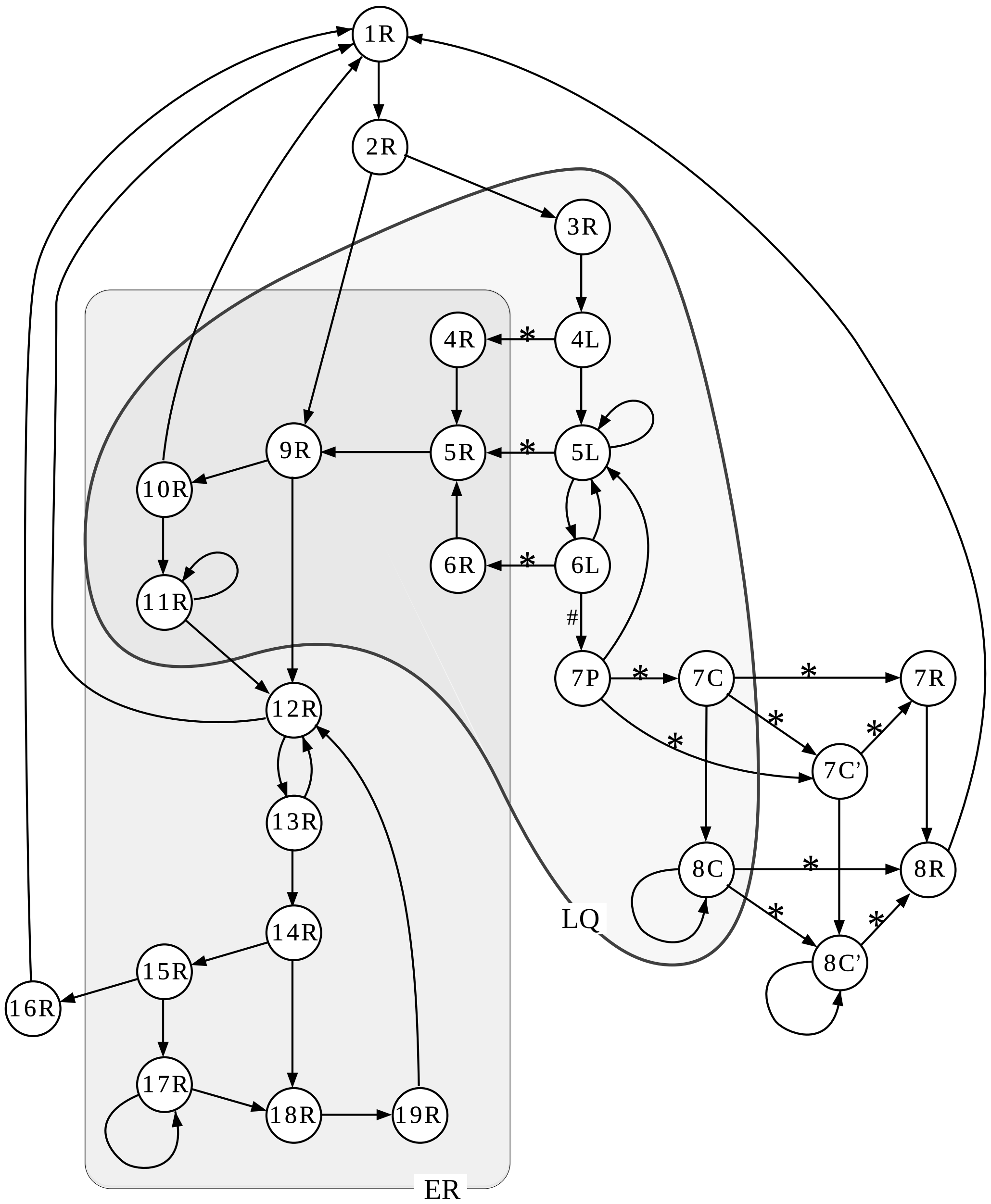


Figure 72: The State Transition Diagram of a Processor

A processor is called to be in the *exclusive region* (ER, in short), when its state is included in ER in Figure 72. In the following, we call the *locked* and *next* fields of the queue node of a processor simply as the *locked* and *next* fields of the processor, respectively.

Lemma 1 When L is *NULL*, no processor is in ER. When L is not *NULL*, there is one (and only one) processor that is in ER.
Proof: In the initial state, the condition is satisfied because L is initialized to *NULL* and the state of each processor is 1R. Then, the lemma can be proved by showing that for each transition, if the condition is satisfied before the transition, it is preserved with the transition. We may safely check only the transitions with which a processor enters/leaves ER or L is modified.

- 2R→9R (The processor enters ER and L is modified.)

  This transition occurs only when L is *NULL*, and changes it to non-*NULL*. There are no processor in ER before the transition since L is *NULL*. Therefore, the condition is preserved.

- 4L→4R, 5L→5R, 6L→6R (The processor enters ER.)

  These transitions occur only when another processor changes the *locked* field to *Locked*; in other words, it makes the transition from 12R to 1R. In this case, a processor enters ER while another leaves ER. As L is not modified with these transitions, the condition is preserved.

- 12R→1R (The processor leaves ER.)

  A processor making this transition changes the *locked* field of another processor from *Locked* to *Released*; in other words, it causes a transition from 4L/5L/6L to 4R/5R/6R on another processor. This is the same situation with the above.

- 10R→1R, 15R→16R (The processor leaves ER and L is modified.)

  These transitions occur only when L is not *NULL* and change it to *NULL*. Therefore, the condition is preserved.

- 2R→3R (L is modified.)

  L is kept non-*NULL* with this transition. Therefore, the condition is preserved. □

Theorem 2 (Mutual Exclusion) There is at most one processor which is in 9R state.
Proof: This directly follows from Lemma 1. □

In the following, the processor in ER is called the *lock holder* (LH, in short), if any. A processor is called to be designated by a pointer variable when its queue node is pointed to by the pointer.

Next, we define the *lock queue*We do not use the word “waiting queue” because the lock holder can be included in the queue. which is an ordered list of processors. The last processor of the lock queue is defined to be the one designated by *L*. When *L* is *NULL*, the lock queue is defined to be empty. The predecessor of a processor in the lock queue is the one designated by its *pred* variable. When *L* is not *NULL*, the first processor of the queue is defined according as the state of LH (which exists from Lemma 1) as follows.

(1) When LH is in 4R, 5R, 6R, 9R, 10R, or 11R, LH is the first processor of the lock queue.

(2) When LH is in 12R, 13R, 14R, 15R, 17R, or 18R, the processor designated by the *succ* variable of LH is the first one of the lock queue.

(3) When LH is in 19R, the processor designated by the *sn* variable of LH is the first one of the lock queue.

In the next lemma, we show that the lock queue is well-structured and handled focusing only on the lock queue operations. We need the following assumption for further discussion.

Assumption 3 Any processor has not been included in the lock queue when it is in 1R state. □

In the initial state, this assumption is satisfied because all processors are in 1R and because the lock queue is empty. To show that the assumption always holds, it is necessary to prove that a processor is not included in the lock queue when it returns to 1R state. The algorithm in Figure 70 and 71 realizes this property by introducing the transient status in which the *locked* field is *Canceled*.

In the following, we suppose that this assumption alway holds. It is proved that a processor is not included in the lock queue when it returns to 1R state in Lemma 7 after the discussions which take the value of *locked* fields into consideration. This result shows that the assumption is preserved if it is satisfied in the initial state. Therefore, the assumption is proved inductively using Lemma 7.

Lemma 4 Following two conditions hold under Assumption 3.

(1) A processor modifies the lock queue with only two kind of operations: (a) inserting itself at the end of the lock queue when it is not included in the queue and (b) removing the first processor of the lock queue from the queue.

(2) When the *next* field of a processor included in the lock queue is not *NULL*, it designates the successor of the processor in the lock queue.

Proof: In the initial state, the conditions are satisfied because no operation has been done on the lock queue and because the lock queue is empty. Then, the lemma can be proved by showing that for each transition, if the conditions are satisfied before the transition, they are preserved with the transition. We may safely check only the transitions with which the lock queue is changed or with which the *next* field of a processor included in the lock queue is modified. The lock queue is modified in the following four cases: (a) L is changed, (b) the *pred* variable of a processor in the lock queue is changed, (c) LH is changed, and (d) LH makes a transition beyond the boundaries with which the first processor of the lock queue is defined.

- 2R→3R, 2R→9R (L is changed and the *pred* variable is changed.)

  A processor making one of these transitions becomes the last processor of the lock queue after the transition. In case of 2R→3R, the last processor before the transition is designated by the *pred* variable. The first processor of the lock queue remains unchanged. In case of 2R→9R, the lock queue is empty before the transition and includes only the processor making the transition after the transition. In both cases, the processor making the transition is inserted at the end of the lock queue.

  Because a processor in 1R is not included in the lock queue from Assumption 3 and because a processor is not inserted to the lock queue by another processor from Condition (1), a processor in 2R is not included in the lock queue.

  Since the *next* field of a processor is modified only when it is designated by the *pred* variable of another processor, the *next* field of the processor which is not included in the lock queue or is at the end of the lock queue is not modified by another processor. Because the processor making the transition 2R→3R/9R is not included in the lock queue before the transition and is at the end of the lock queue after the transition, the *next* field of the processor is not modified for the while. Therefore, the *next* field of the processor is *NULL* immediately after the transition.

  From the above discussions, if the conditions are satisfied before one of the transitions, they are preserved after the transition.

- 10R→1R, 15R→16R (L is changed.)

  Before these transitions, the lock queue includes only one processor (LH in case of 10R→1R, and the processor designated by the *succ* variable of LH in case of 15R→16R) because the first processor of the lock queue is designated by L. After the transitions, the lock queue becomes empty. Therefore, the transitions remove the unique processor (witch is the first processor obviously) in the lock queue from the queue, and the conditions are preserved with the transitions.

- 4L→4R, 5L→5R, 6L→6R (LH is changed.)

  These transitions occur only when another processor makes the transition from 12R to 1R. Before the transitions, the first processor of the lock queue is the one designated by the *succ* variable of the latter processor, which is the former processor obviously. After the transitions, the former processor is the first one. Consequently, the lock queue is not modified with these transitions and the conditions are preserved.

- 12R→1R (LH is changed.)

  A processor making this transition causes a transition from 4L/5L/6L to 4R/5R/6R on another processor. This is the same situation with the above.

- 9R→12R, 11R→12R (LH makes a transition beyond the boundaries.)

  The first processor of the lock queue is changed from LH to the one designated by the *succ* variable of LH with these transitions. The *succ* variable of LH equals to *I.next* and designates the successor of LH in the lock queue. Therefore, the transitions remove LH, which is the first processor of the lock queue, from the queue, and the conditions are preserved.

- 18R→19R (LH makes a transition beyond the boundaries.)

  The first processor of the lock queue is changed from the one designated by the *succ* variable of LH (P_0) to the one designated by the *sn* variable (P_1) with this transition. The *sn* variable of LH equals to *succ*→*next* and designates the successor of P_0 in the lock queue. Therefore, the transitions remove P_0, which is the first processor of the lock queue, from the queue, and the conditions are preserved.

- 19R→12R (LH makes a transition beyond the boundaries.)

  The first processor of the lock queue is changed from the one designated by the *sn* variable of LH to the one designated by the *succ* variable with this transition from the definition. Because the *succ* variable after the transition equals to the *sn* variable before the transition, the first processor is not changed in actual and the conditions are preserved.

- 4L→5L, 4R→5R (The *next* field is modified.)

  The processor making one of these transitions makes the *next* field of the processor designated by its *pred* variable designate itself. Therefore, the *next* field designates the successor in the lock queue, and Condition (2) is shown to be preserved with the transitions. Since the lock queue is not modified with the transitions, Condition (1) is preserved obviously. □

Lemma 5 Following conditions hold under Assumption 3.

(1) When LH is in 14R, 15R, 17R, or 18R, the processor designated by the *succ* variable of LH is in C state. Conversely, a processor in C state is designated by the *succ* variable of another processor in 14R, 15R, 17R, or 18R.

(2) When a processor is in 16R, the processor designated by its *succ* variable is in C' state. Conversely, a processor in C' state is designated by the *succ* variable of another processor in 16R.

Proof: First, we prove that the following condition is satisfied under Assumption 3.

(0) When a processor is in 14R, 15R, 16R, 17R, or 18R (we call the processor is in *SC* in the following), the *locked* field of the processor designated by its *succ* variable is *Canceled*. Conversely, a processor whose *locked* field is *Canceled* is designated by the *succ* variable of another processor in SC.

Since this condition obviously holds in the initial state, it is proved to be satisfied by showing that every transition preserves the condition. We may safely check only the transitions with which a processor enters/leaves SC and the ones with which the *locked* field of a processor is changed from/to *Canceled* to/from another.

- 13R→14R

  With this transition, LH enters SC and *Canceled* is assigned to the *locked* field of the processor designated by the *succ* variable of LH. Therefore, if Condition (0) is satisfied before the transition, it is also satisfied after the transition.

- 18R→19R

  With this transition, LH leaves SC and *Released* is assigned to the *locked* field of the processor designated by the *succ* variable of LH. Therefore, Condition (0) is preserved.

- 16R→1R

  From the proof of Lemma 4, the processor designated by the *succ* variable (P_0) is not included in the lock queue immediately after the transition from 15R to 16R. Since the Condition (0) is assumed to be satisfied before the transition 16R→1R, the *locked* field of P_0 is kept to be *Canceled*. Because a new processor is added to the lock queue only with the transition from 2R to 3R/9R (from the proof of Lemma 4), the processor P_0, whose *locked* field is kept to be *Canceled*, is not inserted to the lock queue. Consequently, the processor designated by the *succ* variable of another processor in 16R is proved to be not included in the lock queue. Since the processor designated by the *succ* variable of another processor in 14R, 15R, 17R, or 18R is the first one in the lock queue by definition, it is never designated by the *succ* variable of any processor in 16R.

  Suppose the case that more than two processors are in 16R state. Because these processors have made the transition from 15R and because their *succ* variables are not modified for the while, the *succ* variables of each two of them never designate the same processor.

  From the above discussions, the transition 16R→1R does not change the states of the processors designated by the *succ* variables of other processors in SC and preserves Condition (0).

Since L does not become *NULL* while LH exists from Lemma 1, L is kept non-*NULL* while a processor is in 14R, 15R, 17R, or 18R. Therefore, the processor designated by the *succ* variable of LH is in C state for the while. As a processor assigns *NULL* to L with the transition from 15R to 16R, the processor designated by its *succ* variable becomes C' state after the transition. Condition (1) and (2) follow from the above discussion. □

Lemma 6 Following conditions hold under Assumption 3.

(1) The transition 13R→14R (and only the transition) causes the transition 7P→7C (not 7P→7C') on the processor designated by the *succ* variable.

(2) The transition 15R→16R (and only the transition) causes the transition 7C→7C' or 8C→8C' on the processor designated by the *succ* variable.

(3) The transition 16R→1R (and only the transition) causes the transition 7C'→7R or 8C'→8R (not 7C→7R or 8C→8R) on the processor designated by the *succ* variable.

(4) The transition 18R→19R causes (and only the transition) the transition 7C→7R or 8C→8R (not 7C'→7R or 8C'→8R) on the processor designated by the *succ* variable.

Proof: Because the processor designated by the *succ* variable of another processor in 14R is in C state from Lemma 6, the transition 13R→14R causes the transition 7P→7C (not 7P→7C') on the former processor. Since there are no other transitions which change the *locked* field from *Preempted* to *Canceled*, Condition (1) is shown to be satisfied.

They are also shown from Lemma 6 that the transition 16R→1R causes a transition from C' state to R state on another processor and that 18R→19R causes a transition from C state to R state. Since there are no other transitions which change the *locked* field from *Canceled* to *Released*, Condition (3) and (4) are shown to be satisfied.

Similarly, the transition 15R→16R causes a transition from C state to C' state on the processor designated by the *succ* variable from Lemma 6.

There are two transitions 15R→16R and 10R→1R which make *L* to *NULL*. As a processor making the transition from 10R to 1R is LH before the transition, there are no other processor in 14R, 15R, 17R, or 18R. Therefore, if there are some processors whose *locked* fields are *Canceled*, they are proved to be in C' state from Lemma 6. Consequently, the transition 10R→1R does not cause a transition from C state to C' state on another processor, and Condition (2) is proved to be satisfied. □

Lemma 7 The state of the processor linked to the lock queue is included in LQ in Figure 72. The processor whose state is included in LQ is linked to the lock queue.
Proof: In the initial state, the condition is satisfied because *L* is initialized to *NULL* and the state of each processor is 1R. Then, the lemma can be proved by showing that for each transition, if the condition is satisfied before the transition, it is preserved with the transition. We may safely check only the transitions with which a processor enters/leaves LQ or the lock queue is modified.

- 2R→3R, 2R→9R (The processor enters LQ and the lock queue is modified.)

  The processor making one of these transitions is added at the end of the lock queue (from the proof of Lemma 4). Therefore, the condition is preserved.

- 10R→1R (The processor leaves LQ and the lock queue is modified.)

  This transition occurs when only the processor making the transition is included in the lock queue, and the lock queue becomes empty after the transition. Therefore, the condition is preserved.

- 9R→12R, 11R→12R (The processor leaves LQ and the lock queue is modified.)

  The processor making one of these transitions is removed from the lock queue (from the proof of Lemma 4). Therefore, the condition is preserved.

- 7C→7C', 8C→8C' (The processor leaves LQ.)

  These transitions occur only when LH makes the transition from 15R to 16R from Lemma 6 (2). Since the processor making one of these transitions, which is designated by the *succ* variable of LH, is removed from the lock queue, the condition is satisfied after the transition.

- 15R→16R (The lock queue is modified.)

  This transition causes the transition from 7C/8C to 7C'/8C' on the processor designated by the *succ* variable from Lemma 6 (2). This is the same situation with the above.

- 7C→7R, 8C→8R (The processor leaves LQ.)

  These transitions occur only when LH makes the transition from 18R to 19R from Lemma 6 (4). Since the processor making one of these transitions, which is designated by the *succ* variable of LH, is removed from the lock queue, the condition is satisfied after the transitions.

- 18R→19R (The lock queue is modified.)

  This transition causes the transition from 7C/8C to 7R/8R on the processor designated by the *succ* variable from Lemma 6 (4). This is the same situation with the above.

- 7P→7C' (The processor leaves LQ.)

  The only transition which changes the state of another processor from P state to C/C' state is 13R→14R. Because it is shown that the transition 13R→14R changes the state of another processor from P state to C state from Lemma 6 (1), the transition from 7P to 7C' never occurs.

None of the transitions 4L→4R, 5L→5R, 6L→6R, 12R→1R, and 19R→12R actually changes the lock queue from the proof of Lemma 4. □

From this lemma, it is proved that a processor is not included in the lock queue when it returns to 1R, and Assumption 3 can be proved by induction.

To prove deadlock freedom of the algorithm, we assume that each processor makes the next transition in finite duration of time. First, we show that the *next* field is written non-*NULL* value in finite duration of time.

Lemma 8 If a processor included in the lock queue is not the last one in the queue, its *next* field becomes non-*NULL* in finite duration of time under the assumption that each processor makes the next transition in finite duration of time.

Proof: Suppose the case that a processor makes the transition from 2R to 3R and inserts itself at the end of the lock queue. From the assumption, the processor makes the *next* field of its predecessor designate itself, makes the field non-*NULL* in other words, within finite duration of time after the transition. From the other point of view, the *next* field of the processor which is included in the lock queue but not the last one in the queue becomes non-*NULL* in finite duration of time. □

The deadlock freedom of the algorithm can be derived as the following theorems.

Theorem 9 (Deadlock Freedom (1)) When no processor holds a lock and some processors try to acquire the lock, one of them can acquire the lock within finite duration of time.

Proof: When no processor holds the lock (or is in ER), L is *NULL* from Lemma 1. Therefore, the lock queue is empty by definition and there is no processor whose state is in LQ from Lemma 7. Then, all of the processors trying to acquire the lock are in 7C', 8C', 7R, 8R, 1R, or 2R.

A processor in 8C' moves to 8R in finite duration of time because the state 8C' is a result of the transition 15R→16R on another processor and because the transition 16R→1R occurs in finite duration of time on the processor. Similarly, a processor in 7C' moves to 7R or 8C' in finite duration of time.

Therefore, every processor trying to acquire the lock reaches 2R in finite duration of time. The first processor trying the transition from 2R moves to 9R since L remains to be *NULL* and succeeds in acquiring the lock. □

Theorem 10 (Deadlock Freedom (2)) A processor trying to release a lock finishes to release the lock within finite duration of time, if the number of interrupt requests raised on other processors during the release operation is bounded.

Proof: There are four loops in the lock releasing routine: 11R→11R, 17R→17R, 12R→13R→12R, and 12R→ $\cdots$ →19R→12R. This theorem can be proved by showing that a processor trying to release a lock finishes these loops in finite duration of time under the condition that the number that other processors make the transition from 6L to 7P is bounded.

1. 11R→11R, 17R→17R

   A processor finishes these loops in finite duration of time from Lemma 8.

2. 12R→13R→12R

   When LH is in 12R or 13R, *succ→locked* never becomes *Released* or *Canceled*. It never becomes *Released* because the processor designated by *succ* is included in the lock queue and is not LH. It never becomes *Canceled* from Lemma 5.

   Consequently, the transition 13R→12R occurs only when *succ→locked* is modified from *Preempted* to *Locked* while LH is in 13R. From the assumption that the number of interrupt requests raised on other processors during the release operation is bounded, the number of the transition from 6L to 7P, which is the only transition changing the *locked* field to *Preempted*, is bounded, and the execution of this loop is finished in finite duration of time.

3. 12R→ $\cdots$ →19R→12R

   When LH makes the transition from 18R to 19R, the first processor of the lock queue is removed from the queue. Therefore, the length of the lock queue becomes shorter as the processor executes this loop. From the assumption that the number of interrupt requests raised on other processors during the release operation is bounded, the maximum number of processors which are included in the lock queue when release operation is started and the processors which are inserted to the queue afterwards is bounded. Therefore, the maximum execution number of this loop is bounded. □

Finally, we show the equivalence of the algorithm in Figure 26 and 27 and the one in Figure 70 and 71. When a processor is in 16R or 18R, *succ→locked* is fixed to be *Canceled* from Lemma 5. Therefore, the compare_and_swap operations in the lines marked with ⑯ and ⑱ in Figure 71 are equivalent to simple assignments.

A processor refers to the *locked* field of another processor only when the latter processor is designated by the *next* field of LH or other processors in the lock queue. In other words, the *locked* field of a processor is referred to only when the processor is included in the lock queue and is not LH, and after it makes the *next* field of its predecessor designate itself. In short, it is referred only when the processor is in ⑤, ⑥, ⑦, or ⑧. Consequently, its initial value is never referred to.