TOWARDS A SCALABLE REAL-TIME KERNEL FOR FUNCTION-DISTRIBUTED MULTIPROCESSORS

H. TAKADA and K. SAKAMURA

University of Tokyo, Department of Information Science,
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Abstract. Scalability is one of the most important requirements for real-time systems on function-distributed multiprocessors. In this paper, requirements on a scalable real-time kernel for function-distributed shared-memory multiprocessors are clarified, and its specification and implementation issues are discussed. We propose a new kernel model in which kernel resources are classified into some classes with different characteristics.

Keywords. real-time kernel, function-distributed multiprocessor, scalability, synchronization, predictability

1. INTRODUCTION

In many applications of high performance real-time systems, 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. Adopting function-distributed (or asymmetric) multiprocessors in which each device is handled by a fixed processor is a promising approach to satisfying this requirement.

Because it is often the case that external devices are added to the system during its life-time in such kind of systems, the system should have the scalability when the number of processors is increased. In the concrete, the maximum execution times and response times of as many jobs as possible should not be prolonged when some processors are added to the system. However, little work has been done to design or implement a *scalable real-time kernel* suitable for realizing application systems with the scalability of worst-case behavior.

In this study, requirements on a scalable real-time kernel for function-distributed shared-memory multiprocessors are clarified, and its specification and implementation issues are investigated.

In implementing a real-time kernel on shared-memory multiprocessors, some mutual exclusion mechanism among processors, such as spin locks, is necessary for the access control of shared data structures within the kernel (e.g. task control blocks and various kind of queues). It is unavoidable that the worst-case execution time of a task that must acquire an interprocessor lock during its execution becomes long as the number of contending processors is increased. Therefore, such a task cannot meet the scalability condition (i.e. the maximum execution time of the task becomes longer when the number of processors is increased).

To cope with this situation, we have adopted the approach that tasks are classified into some classes with different characteristics, and that the tasks belonging to the appropriate class having the required property for a job are used for implementing the job. For example, there exists a class of tasks that can meet the scalability condition, but that cannot synchronize or communicate with the tasks executed on other processors. Time-critical jobs should be implemented with this class of tasks if possible. Another class of tasks can synchronize with the tasks on other processors, but cannot meet the scalability condition.

In this paper, we propose a classification method of kernel resources with which a scalable real-time kernel satisfying the above requirements can be realized, and describe its implementation methods.

A real-time kernel investigated in this paper is a basic software for real-time systems supporting task management, priority-based preemptive scheduling, intertask synchronization and communication, and memory management functions, which is also called as a real-time monitor or a real-time executive. In the concrete, we have adopted the μITRON3.0 Specification (Sakamura, 1994), an open real-time kernel specification for embedded systems, as the base kernel specification to be extended to support multiprocessors. A multiprocessor discussed in this paper is that having several or around ten processors. We do not discuss on massively parallel systems.

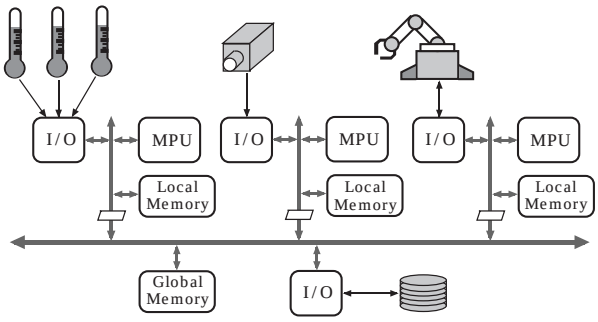

Fig. 1. Function-Distributed Multiprocessor

In Section 2, we describe the important features of function-distributed multiprocessors and clarify the requirements on a scalable real-time kernel for them. Section 3 describes some implementation methods and issues of multiprocessor real-time kernels. We analyze the access pattern on data structures within kernel and investigate on the lock granularity used in the kernel. Then, we discuss the classification methods of tasks and intertask synchronization/communication objects (such as semaphores and mailboxes), and their accessibility from each class of tasks in Section 4.

2. SCALABILITY FOR FUNCTION-DISTRIBUTED MULTIPROCESSORS

2.1 Function-Distributed Multiprocessors

In many applications of real-time systems, where the required computational resources for each external device can be estimated beforehand, it is preferable that each device be handled by a fixed processor and that the interface with the device be connected to the local bus of the processor (Fig. 1). A distributed shared-memory architecture is also adopted in which a memory module is connected to the local bus of each processor. In this kind of function-distributed multiprocessors, because the code and data areas of the program handling an external device are placed in the local memory of the 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.

When a real-time system is constructed using a function-distributed multiprocessor, external devices and jobs handling them are allocated to processors so that the following conditions are satisfied: (a) interprocessor synchronization are minimized and (b) time-critical jobs are processed without synchronizing with other processors as far as possible. Consequently, in well-designed real-time systems on function-distributed multiprocessors, many tasks, including most of the time-critical tasks in the system, can be processed without synchronizing with other processors (we call such a task as closed within a processor below).

Even in a system with severe timing constraints, there often exists some tasks that are not time-critical. The mechanism that such kind of background tasks is executed on idle processors is very useful. In shared-memory multiprocessors, task migrations can be realized with a simple mechanism.

2.2 Scalability for Function-Distributed Multiprocessors

In general, the maintenance cost of a real-time system is said to be very high because a small modification of a job can affect the timing behavior of the whole system. In order to reduce the maintenance cost of the system, the influence of a modification should be minimized. In function-distributed multiprocessors, it is desirable that the timing behavior of the jobs closed within a processor should not be influenced by a modification of the jobs on other processors, and vice versa. We will discuss the most serious problem to be solved for this requirement in Section 3.4.

In multiprocessor systems, it is often the case that the number of processors is increased during the life-time of the system, for example, when the performance requirements on the system is changed or when some external devices are added. In this situation, it is preferable that the worst-case execution times and response times of the time-critical jobs that are not modified are not prolonged. In other words, the worst-case behavior of the system should have the scalability to the number of processors.

However, it is unavoidable that the maximum execution time of a job that uses a shared resource[1] exclusively is prolonged as the number of contending processors is increased, with its linear order at least. By making the maximum execution times and response times of other jobs (i.e. jobs that are executed without synchronizing with other processors[2]) independent of the number of contending processors, the system maintenance cost can be considerably reduced. This is because most of the time-critical jobs are closed within a processor in well-designed real-time systems on function-distributed multiprocessors.

Consequently, the scalability requirements on real-time systems using function-distributed multiprocessors can be described as follows.

1. The maximum execution time and response time of a job that is closed within a processor should not depend on the jobs on other processors or on the number of contending processors.
2. The maximum execution time and response time

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

[2] 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 3.2.

of a job that synchronize with jobs on other processors should not depend on the jobs closed within other processors and should be bounded with the linear order of the number of contending processors.

In practice, because we cannot satisfy both requirements in strict at the same time, the first requirement is relaxed a little. In the concrete, we allow that the execution time of a tight loop of the number of processors is included in the maximum execution time or response time of a job closed within a processor. This is justified from the assumption that we do not handle massively parallel systems.

3. IMPLEMENTATION METHODS OF MULTIPROCESSOR REAL-TIME KERNELS

3.1 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 synchronizations among processors are implemented with application-level programs. However, this method has a drawback that 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 interface with tasks on the same processor and that with tasks on other processors are completely different.

To remedy this problem, a multiprocessor real-time kernel is necessary with which a task can synchronize with tasks on other processors with the same interface with tasks on the same processor. In other words, a task can synchronize with any task with the same set of system calls. In such a kernel, 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/communication object (called as a synchronization object or simply as a object below), such as a semaphore and a mailbox, also has its host processor and can be accessed from any task in the system. In this paper, we call this kernel model as the basic model of real-time kernels for function-distributed multiprocessors, or the basic kernel model in short (Fig. 2).

3.2 Direct Access Method and Remote Invocation Method

There are two approaches to implementing an OS kernel on function-distributed shared-memory multiprocessors: the direct access method and the remote invocation method (Chaves *et al.*, 1991).

With the direct access method, when a task operates on a resource on another processor, it directly accesses the control block of the resource located on the local memory of the processor. In implementing a real-time kernel, because the execution time of each primitive operation is very short, spin locks are used to access the control block exclusively.

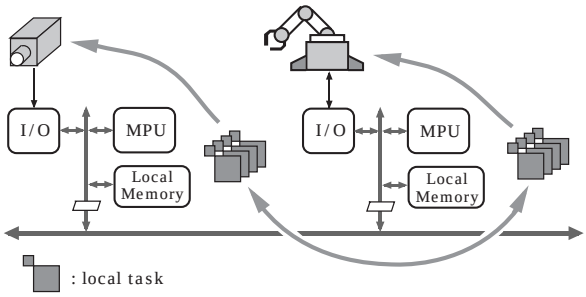

Fig. 2. Basic Kernel Model

With the remote invocation method, which is used for multiprocessors without shared memories, when a task operates on a resource on another processor, it sends a message requesting the operation to the processor 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 invokes a system call that operates on a blocked task τ_2 on another processor P_2 and that makes τ_2 ready to execute.

Direct Access Method

At first, τ_1 finds the address of τ_2's task control block (TCB) and then tries to lock the TCB. When τ_1 succeeds to acquire the lock, it accesses the TCB and changes the state of τ_2. Because τ_2 becomes ready to execute, τ_1 enqueues τ_2 to the ready queue of P_2. 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 P_2 to switch the executing task using an interprocessor interrupt.

Remote Invocation Method

τ_1 enqueues a request information block into the request queue of P_2. The request information block includes the kind of operation, the parameters passed to it, and an empty field to which the requested processor writes the result. Then, τ_1 asks P_2 to process the request using an interprocessor interrupt and spins until the result of the operation is written in the request information block. When P_2 accepts the interrupt, it 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 and the performance requirements of the application. From the performance requirements of real-time applications, the direct access method is usually suitable because the serialization unit of processing is smaller than the remote invocation method. More precisely,

the remote invocation method has the following drawbacks in implementing real-time kernels for function-distributed multiprocessors.

1. 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.
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 interprocessor 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 execution time of a remote invocation. If the interprocessor interrupt has a higher priority than an external interrupt, it is difficult to bound the response time of the external interrupt (Takada & Sakamura, 1991).

With these reasons, we assume the direct access method in the following discussion.

3.3 Kernel Data Structures and Lock Units

In implementing a real-time kernel for shared-memory multiprocessors, the lock granularity of kernel data structures is one of the most important issues. In this section, 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 for an operation is not a suitable approach, because the execution time of each system call is very short and because lock acquisition overhead is relatively large. In function-distributed multiprocessors, kernel data structures on different processors should be placed in different lock units from the viewpoint of scalability.

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 (Sakamura, 1994), which we have adopted as the base kernel specification. Major data structures in the kernel are as the followings.

(1) task control blocks (TCB)

(2) a task ready queue (a task queue which includes all the tasks that are ready to execute)

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

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

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

Table 1. System Calls of the μITRON Specification

In this paper, we omit the discussion on the timer event queue.

The list of the level S[3] system calls of the μITRON Specification is presented in Table 1. We have analyzed the access pattern on the data structures of each system call. 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, it also accesses the TCB of the 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 task should be included in the same lock unit. Another observation is that one-write/many-readers type of synchronizations are not

[3] In the μITRON3.0 Specification, system calls are classified into level R (required), level S (standard), level E (extended), and level C (CPU-dependent) according as their necessity.

necessary. This is because a read access on a data structure is usually followed by a write access.

System calls in Table 1 are classified into the following six categories from their access patterns on the kernel data structures.

(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 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 object

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

(d) wait operations on a synchronization object

A system call of this category first accesses the control block of the designated synchronization 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 object

A system call of this category first accesses the control block of the designated synchronization 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 1 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.

As the results of these investigations, a lock unit should be prepared for the control blocks of synchronization 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 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. If the TCBs and the control blocks of synchronization objects were included in the same lock unit, two parallel invocations of system calls of category (e), which are usual operations, could cause a deadlock.

3.4 Scalable Synchronization and Interrupt Latency

In Section 3.2, we have pointed out that the remote invocation method has the problem on the precedence of external interrupts and interprocessor synchronizations. A similar problem can occur with the direct access method.

In bounded spin lock algorithms, a processor reserves its turn to acquire a lock when it begins to wait for the lock. For example, in queueing spin locks (Mellor-Crummey & Scott, 1991), a processor trying to acquire a lock enqueues itself to a FIFO queue of waiting processors. If the processor services external interrupts and its turn to acquire the lock comes during the interrupt service, the remaining interrupt service time is included in the time that the processor holds the lock. As the consequence, it is difficult (or impossible) to give a practical upper bound on the time that a processor holds a lock and also on the time until another processor acquires a lock. If a processor disables interrupt services before enqueueing itself to the queue, on the other hand, the interrupt disabled period includes the time to acquire the lock and its upper bound depends on the number of contending processors.

In order to realize fast interrupt response and predictable interprocessor synchronizations at the same time, we have proposed a queueing spin lock algorithm supporting temporary preemptions for interrupt services (Takada & Sakamura, 1994). In order to apply the algorithm to a real-time kernel, a long-term preemption scheme (Wisniewski *et al.*, 1993; Takada & Sakamura, 1993) is also necessary to handle the case that a higher priority task becomes ready to execute and that the task waiting for a lock is preempted.

4. CLASSIFICATION OF KERNEL RESOURCES

4.1 Private Tasks

In order to meet the scalability requirements presented in Section 2.2, we adopt the approach to classify tasks into some classes with different characteristics. The tasks belonging to the class having the required property for a job should be used for implementing the job.

At first, we classify the tasks that are executed without synchronizing with tasks on other processors as *private tasks*, which are managed differently from local tasks. Because the TCB of a private task is not accessed by other processors than its host processor, no interprocessor lock is necessary to access the TCB. A separate ready queue that can be accessed without an interprocessor lock is also prepared for the private tasks on each processor. The task dispatcher (a kernel module that switches task contexts) 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.

In general, when a task synchronizes with another task, the former task must access the TCB of the latter task, and vice versa. Here, the problem arises that a private task, whose maximum execution time should be independent of the number of contending processors, cannot access the TCB of a local task with a simple method, because the TCB of a local task is shared among processors and because an interprocessor lock is necessary to access it. To solve this problem, a precedence to acquire the lock of the local task's TCB is given to its host processor. With this technique, the maximum time duration until a processor can lock the TCB of its local task can be determined independently of the number of contending processors for the lock, and a private task on the processor can access the TCB while satisfying the scalability condition.

Because the maximum execution time of an operation on a remote resource is prolonged as the number of contending processors is increased, a task whose worse-case behavior should not depend on the activities of other 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. Because this restriction is imposed on each private task and because 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 and of the tasks on other processors.

4.2 Task-Independent Synchronization Objects

In this section, the classification of synchronization objects (semaphores, eventflags, and mailboxes in the μITRON Specification) are discussed.

In order that local tasks on different processors synchronize each other, a class of 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 block of a shared object is located on the local memory of a processor, it is also called as a local object 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 the 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 each other, a class of synchronization objects that can be accessed only from the tasks on the processor is necessary. We call this class of objects as *private objects*. No interprocessor lock is necessary to access the control blocks of private 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 2. Accessibility of Kernel Resources

Table 2 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 task can access another task with normal operations (operations of category (a)) but cannot access with special operations (operations of category (b)). When a task tries to operate on an unaccessible resource, the kernel reports an error.

In Table 2, 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.

4.3 Isolated Tasks and Interrupt Handlers

As described in Section 4.1, a private task is necessary to acquire an interprocessor 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 interprocessor locks becomes 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.

In the μITRON Specification, application programmers can write interrupt handlers. System calls can be invoked from interrupt handlers, with the exception of the operations that make the issuing task blocked[4]. Because the execution time of an interrupt handler is included in the maximum response time of isolated tasks, interrupt handlers should not use interprocessor lock and thus the same access restriction with the isolated tasks is applied. When isolated tasks are not used, it is still reasonable that the same access restriction is applied to interrupt handlers.

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

A system call that disables interrupt services is prepared in the μITRON Specification. While a task disables interrupt services, both the access restriction on the task and that on an isolated task on the same processor are applied to the task.

4.4 Global Tasks

One of the advantages of shared-memory multiprocessors is that task migrations can be easily implemented. As described in Section 2.1, time-critical tasks should be bound to a processor. 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 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 interprocessor 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 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.

4.5 Accessibility of Kernel Resources

With the similar discussions with the previous sections, the classes of isolated tasks, isolated object, and global tasks are added to the accessibility table. The result is presented in Table 3 and Fig. 3. 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.

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. 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.

Note that it is not necessary to implement all the resource classes in a kernel. It is also possible that some of the classes are removed from a full-set kernel when they are not used.

4.6 Kernel Interface

The classification of kernel resources is reflected to the kernel interface through ID numbers of the resources. In the μITRON 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 remains unchanged.

It is usually the case that the ID of a kernel resource is represented with a symbol in source code and that the mapping of the symbol to an actual number is given within a definition module. When the class of a kernel resource is changed, only the definition module is necessary to be modified.

5. CONCLUSION

In this paper, we clarify the requirements on a scalable real-time kernel for function-distributed shared-memory multiprocessors. Then, we investigate its specification and implementation issues. As the result, a kernel model in which kernel resources are classified into some classes with different characteristics is proposed.

In the concrete, tasks are classified into the following four classes.

(1) the class of isolated tasks whose worst-case execution time is same with a single processor system, but that cannot do any synchronization with other processors.

(2) the class of private tasks that can meet the scalability condition, but that cannot synchronize with the tasks on other processors.

(3) the class of local tasks that can synchronize with the tasks on other processors, but that cannot

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 3. Accessibility of Kernel Resources (Full Set)

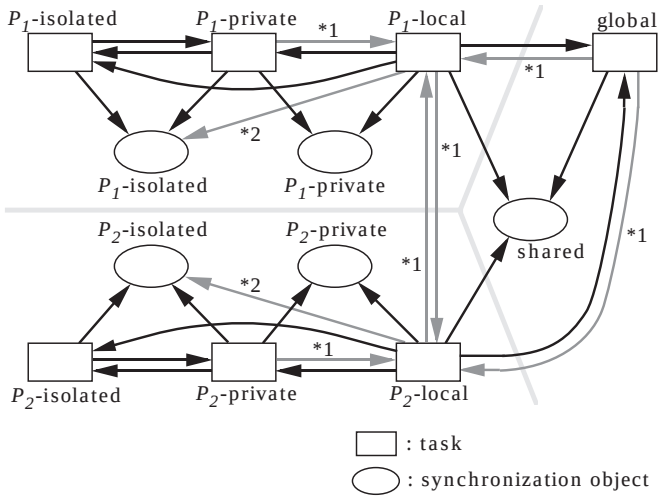

Fig. 3. Accessibility of Kernel Resources (Full Set)

meet the scalability condition.

(4) the class of global tasks that can be executed on any processor in the system, and that can migrate to other processors during their execution.

In correspondence with the classification of tasks, the synchronization objects are classified into three classes: isolated objects, private objects, and shared objects. Shared objects can further classified into local objects and global objects. Whether a task can operate on an object or not is determined by the class of the task and that of the object from Table 3.

We are now implementing the kernel model proposed in this paper. The evaluation of the model remains as future work. In order to complete the implementation, there exist some other problems remaining to be investigated.

6. REFERENCES

Chaves, Jr., E. M., Das, P. C., LeBlanc, T. J., Marsh, B. D., and Scott, M. L. (1991). Kernel-Kernel Communication in a Shared-Memory Multiprocessor. *Tech. Rep. TR368, Computer Science Department, University of Rochester*, Apr. 1991.

Mellor-Crummey, J. M., and Scott, M. L. (1991). Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Computer Systems* **9(1)**, 21 – 65.

Sakamura, K., Ed. (1994). *μITRON3.0 Specification.* TRON Association, Tokyo. (can be obtained from http://tron.is.s.u-tokyo.ac.jp/TRON/ITRON/SPEC/mitron3.txt.Z).

Takada, H., and Sakamura, K. (1991). Implementation of Inter-Processor Synchronization/Communication and Design Issues of ITRON-MP. *Proc. of 8th TRON Project Symposium*, Tokyo, Nov. 1991, 44 – 56, IEEE CS Press.

Takada, H., and Sakamura, K. (1993). A Bounded Spin Lock Algorithm with Preemption. *Tech. Rep. 93-2, Department of Information Science, University of Tokyo*, Jul. 1993.

Takada, H., and Sakamura, K. (1994). Predictable Spin Lock Algorithms with Preemption. *Proc. 11th Workshop on Real-Time Operating Systems and Software*, Seattle, May 1994, 2 – 6, IEEE CS Press.

Wisniewski, R. W., Kontothanassis, L., and Scott, M. L. (1993). Scalable Spin Locks for Multiprogrammed Systems. *Tech. Rep. TR454, Computer Science Department, University of Rochester*, Apr. 1993.