Implementation of Inter-processor Synchronization/Communication and Design Issues of ITRON-MP

Hiroaki Takada          Ken Sakamura

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

Abstract

The ITRON-MP specification is the extension of ITRON for shared-memory multiprocessor systems. This paper outlines the design goals and main features of the ITRON-MP specification, and discusses the implementation issues and their relationship with the specification.

The implementation methods of synchronization and communication mechanism between processors are discussed in detail. We compare the use of remote procedure call and direct memory access, and conclude that direct memory access is appropriate for real-time kernels. As the result, the ITRON-MP specification is designed to be realizable using the direct memory access method.

1 Introduction

One of the objectives of the TRON Project is to construct HFDS (Highly Functional Distributed System) [1]. In HFDS environment, intelligent objects play an important role. We utilize ITRON based kernels to construct many kinds of intelligent objects, including home electronic appliances, automobiles, and large-scale control systems.

We are engaged in extending the ITRON specification to two directions for making it applicable to wider area of applications. One of them is ITRON-MP [2] which is the extension of the ITRON specification for shared-memory multiprocessor systems ("MP" stands for MultiProcessor), and the other is ITRON-N which is the extension for distributed systems ("N" stands for Network). We also plan to integrate these two extensions to the IMTRON specification, which is a milestone to MTRON (Figure 1).

High run-time performance and real-time property is essential for real-time kernels. Therefore, in designing a specification of a real-time kernel, we must carefully investigate its implementation methods.

In this paper, implementation methods of synchronization and communication mechanism between processors with shared memory, which is among the most important issues to implement ITRON-MP based kernels, are discussed. We compare the use of remote procedure call (RPC) and direct memory access in detail. The design guideline of ITRON-MP, which is the result of the discussion, is also described.

The application areas and the design goals of ITRON-MP are described in Section 2, and typical multiprocessor architectures for the application areas are illustrated in Section 3. In Section 4, the implementation methods of synchronization and communication mechanisms between processors are discussed in detail. We outline the ITRON-MP specification in Section 5.

2 Design Concepts of ITRON-MP

As the application area of embedded real-time systems becomes large, requirements for large-scale and high performance real-time systems increase. Especially, these requirements are rapidly increasing in the application areas of large-scale control systems, transaction processing systems, and communication processing machines.

In these application areas, a system requires not only big computational power but also responsive in-
Interrupt service and fast task switching at the order of microseconds to handle a large number of external devices. Because these requirements cannot be satisfied with one processor, necessities for multiprocessor systems are emerging. A multiprocessor system is particularly an effective approach to make a system responsive to many external events.

2.1 Application areas of ITRON-MP

ITRON-MP is a real-time kernel specification for embedded systems or special-purpose machines. Its major applications areas are described below.

Large-scale control systems

Large-scale control systems (e.g. plant control systems and aircraft control systems) require not only big computational power, but also hard real-time property, in that missing a timing constraint can cause catastrophic results. Some systems controlling quickly working objects require interrupt response and task switching time as fast as the order of microseconds.

Transaction processing systems

Transaction processing systems (e.g. on-line banking systems and seat reservation systems) usually have a number of I/O devices, such as disk units and terminals, and require soft real-time property. As these systems often communicate with end users, the required total response time is 100msec. or 1sec. Though very fast response time is not necessary, the number of transactions to be processed is very large and fast task switching is required.

Communication processing machines

Communication processing machines (e.g. packet switchers, network routers, and communication processors which serve as the front-ends of mainframes) require fast I/O processing, and sometimes need big computational power for packet routing and so on. Faster processing will be required, as multi-media communication becomes popular.

Dedicated computation servers

Machines dedicated to a specific numeric calculation or simulation usually require only big computational power. Massively parallel machines, in which the processor connection topology heavily depends on the computing domain, are effective for this kind of machines.

Micro kernel for higher-level OS

ITRON-MP is also used as a basis on which operating systems with higher-level programming interface are constructed. In other words, an ITRON-MP based kernel can be used as a microkernel for these operating systems. For example, the 2B operating system [3], which conforms to the BTRON2 specification [4], is constructed using an ITRON based kernel. 2B can work on a multiprocessor system by extending the ITRON based kernel to ITRON-MP.

2.2 Design Goals of ITRON-MP

We set the following goals in designing the ITRON-MP specification, in order to make it effective for above application areas.

Adaptability to an architecture

The adaptability to an architecture is an ability that a kernel based on the ITRON-MP specification can be used for various multiprocessor architectures in spite of the difference among them, such as the kind and the number of processors, processor connection topology, and the accessibility and the access cost of hardware resources from each processor. The architectures of embedded systems and special-purpose machines are designed to be optimal for each application, the difference among them is remarkable. Architecture diversity is discussed in Section 3.

ITRON-MP adopts the following approach to meet this goal. The specification defines a standard set of kernel interface which can be applicable to wide varieties of multiprocessor architectures, and a tuned kernel code, which includes only necessary functions for each system, is used for the construction of application systems.

Adaptability to an application

The adaptability to an application is an ability that a kernel based on the ITRON-MP specification can be tuned according to the functions and the performance requested by the application for the improvement of the total system performance. For example, when one of the processors is dedicated to a heavy load task, the removal of the task dispatching mechanism from the kernel improves the run-time performance of the system. It is usually a case that the responsiveness of a system is not compatible with its total throughput. In this case, it is important to balance the trade-off between them for each application.
Predictable execution time

Predictable execution time is essential for real-time kernels. It is also necessary that programmer can easily grasp the real-time natures of the system developed on an ITRON-MP based kernel.

High run-time performance

High run-time performance is also important for real-time kernels. To achieve this goal, we avoid excessive abstraction of kernel resources.

Applicability to a fault-tolerant system

Fault-tolerance is an important feature for almost all real-time systems. As 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.

In addition to these goals, the ITRON-MP specification should inherit the features of ITRON, such as weak standardization and ease of training [5].

3 Architecture Diversity

Various multiprocessor architectures to which ITRON-MP is applied are illustrated in this section.

Symmetric architectures

In symmetric architectures (Figure 2), every processor can equally access each hardware resource. Though they are widely adopted in commercial multiprocessor systems because of their generality, the implementation cost of the architectures is quite high, because a high-speed bus and large cache memories with costly coherence mechanism are necessary to obtain good performance. Consequently, they are usually inappropriate for embedded systems and special-purpose machines, and are not the main targets of ITRON-MP.

Symmetric architectures are often called uniform memory access time (UMA) architectures in contrast to non-uniform memory access time (NUMA) architectures described below.

NUMA architectures

In NUMA architectures, every processor can access each hardware resource, but the access cost is not uniform. Hierarchical bus architectures are typical examples of NUMA architectures (Figure 3), in which the access cost of local hardware resources is much smaller than that of remote resources.

NUMA architectures make large-scale multiprocessor systems possible. They relieve the problem of bus traffic saturation. However, they are still too general for embedded systems or special-purpose machines. They also restricts processor connection topology. For example, it is very hard for NUMA architectures to connect processors in an array structure.

Asymmetric architectures

In asymmetric architectures, some hardware resources are accessible only from limited processors. In other words, they are non-uniform resource accessibility architectures.

When the device requires fast response time or large processing power, one of the processors is often dedicated to the control of an I/O device. In this case, the I/O device should be connected to the local bus of the processor. Because I/O
control tasks for the device are always executed on the processor for high run-time performance, the I/O device may be inaccessible from other processors. Optimal architecture for an application tends to be an asymmetric architecture (Figure 4).

Heterogeneous architectures

In heterogeneous architectures, some general processors and some special-purpose processors (e.g., digital signal processors) co-exist. When special-purpose processors do not have task switching facilities, ITRON-MP can offer task synchronization and communication mechanism between different kinds of processors.

4 Remote Resource Access Methods

Inter-task synchronization and communication functions are among the main facilities of ITRON. Tasks can synchronize or communicate each other, by directly accessing a task or by using synchronization/communication objects, such as semaphores and mailboxes. We call the objects managed by the kernel, such as tasks and synchronization/communication objects, as kernel resources. A kernel resource is implemented by a control block for the resource, and accessing a kernel resource is implemented by accessing its control block.

In ITRON-MP, because some tasks are executed simultaneously on different processors, some inter-processor synchronization mechanism is necessary to manage the control blocks of kernel resources. There are two major methods to realize inter-processor synchronization. One is to make each kernel resource accessible from only one processor and to use the synchronization technique for single processor systems. The other is to use locks to access control blocks (explicit synchronization).

With the first method, when a task executed on processor \( p \) tries to access a kernel resource managed by process \( q \), \( p \) must request \( q \) to access the resource and receive the result. In other words, processor \( p \) calls a procedure on processor \( q \). Then, we call this method as the resource access using remote procedure call (RPC), or the RPC method in short. We call the second method as the resource access via direct memory access, or the direct memory access method in short.

Igarashi et al. evaluated these implementation methods from the standpoint of the granularity of lock units in a symmetric architecture \([6]\). In symmetric architectures, because all kernel resources are located on the global memory which can be uniformly accessed from all processors, the contention overhead for the locks on control blocks is a serious problem. We will discuss this problem, later again.

In asymmetric architectures and NUMA architectures, which are important targets of ITRON-MP, kernel resources that are frequently accessed from a specific processor should be located on the local memory of the processor. To show an effectiveness of a local memory, we take an example of a typical multi-processor architecture in which some processor boards with memories and I/O interfaces on their local bus are connected to a backplane bus (Figure 5). Suppose that processor \( p \) sends some data to processor \( q \). If the global memory is used for the communication, two transactions occur on the backplane bus without write-back cache. Communication through the local memory reduces the transactions to only once. We call a kernel resource whose control block is located on the local memory of another processor as a remote resource.

Though the direct memory access method can be implemented with small costs, it has a restriction that a processor cannot access a kernel resource whose control block is located on the memory inaccessible from the processor. On the contrary, the RPC method has...
an advantage that a processor can access a kernel resource located on an inaccessible memory. The RPC method is also effective to access a remote resource, because the access cost of a remote memory is quite high.

In the following, we describe these two implementation methods in detail, and compare them when they are used to access a remote resource. We also discuss the case that a kernel resource is located on a global memory.

4.1 Resource Access using RPC

A task accesses kernel resources through system calls, except the contents of memory blocks. Then, tasks synchronize or communicate each other through system calls, but for the exception. As the synchronization control on shared memory blocks is left to the programmers in ITRON, it is natural to make the unit of RPCs correspond with the unit of system calls, except the case that a task blocks in a system call. We will discuss this exception, later.

The control flow of the resource access method using RPC is described below. We suppose the case that processor $p$ invokes a system call to access a kernel resource whose control block is located on the local memory of processor $q$ (we call this the $q$-local memory, below). The RPC method needs a request queue to which a request block is linked, and a return value block in which processor $q$ writes the return value. A request block includes the function code of the invoked procedure and the parameters passed to it (Figure 6).

1. $p$ puts a remote procedure call request to the request queue.
2. $p$ branches to the service routine of the system call.
3. $p$ raises an interrupt to $q$.
4. The interrupt handler is invoked on $q$.
5. $q$ gets a request from the request queue.
6. $q$ parses the request, and branches to each service routine.
7. $q$ executes the system call requested by $p$.
8. $q$ writes a return value to the return value block.
9. $q$ writes the return value, and returns from the system call.
10. $q$ switches the running task if necessary.

Here, there are some implementation issues: the construction of the request blocks/queue, the location of the request blocks/queue and the return value block, the synchronization mechanism on the request queue, and the method how $p$ waits for the return value. Below, we give a solution to these issues in order to evaluate the runtime efficiency of the method. It should be noted that the solution does not prescribe the implementation method of the ITRON-MP specification.

There are two construction methods of the request blocks/queue: one is to use an array of request blocks as a ring-buffer, and the other is to link request blocks by pointers. The memory location of the request

\[\text{Figure 5: Example Architecture}\]

\[\text{Figure 6: Resource Access using RPC}\]
blocks/queue does not matter, because both processors access them almost the same times. Suppose that an interrupt occurs when processor \( p \) waits for the completion of the RPC, and that the invoked interrupt handler calls a system call which also needs an RPC. In this case, another pair of the request block and the return value block is necessary for the interrupt handler. In general, as many as \( m + 1 \) pairs of these blocks are necessary for each requesting processor, where \( m \) is the maximum number of nesting interrupts on the processor. Then, the area for these blocks should be allocated for each requesting processor. Consequently, it is efficient to locate the request block on the \( p \)-local memory and to link it to the request queue whose head and tail pointers are located on \( q \)-local memory.

Some synchronization mechanisms on the request queue are possible: (1) mutual exclusion control using a lock, (2) preparing a request queue for each pair of \( p \) and \( q \), and (3) using non-blocking synchronization technique \([7]\). Here, we assume the first method, as it is the simplest one.

The simplest method that processor \( p \) waits for the return value is to spin on the return value block until the block is rewritten by processor \( q \). This is possible because some values are never used as a return value of any system call of ITRON. In order to reduce the traffic on the backplane bus, the return value block should be located on \( p \)-local memory. As a return value block is necessary for each request block, a return value block may be one of the fields of a request block.

When processor \( q \) has local I/O devices on its local bus, we must carefully determine the interrupt priority raised by processor \( p \). If we attach higher priority to the interrupt by \( p \) than the interrupt by a local I/O device, the worst interrupt latency for the I/O device is the sum of the maximum interrupt masking time and the processing time of system calls requested by \( p \). Because the maximum processing time of system calls is quite longer than the maximum interrupt masking time in general, the response to the I/O device becomes quite worse. As other processors may request services to \( q \) at the same time, the worst-case response time is unacceptably long. Consequently, the interrupt by \( p \) should have lower priority than interrupts by local I/O devices requiring short interrupt latency.

We present a time chart of the RPC method based on these assumptions in Figure 7. The meaning of time parameters are described in Table 1. The time to invoke the interrupt handler and the time to return from the handler are included in \( t\text{INT}_1 \) and \( t\text{INT}_2 \), respectively.

We investigate the system call latency on processor

---

**Figure 7: Time Chart of the RPC Method**

Processor \( p \):
- (1) put request
- (2) raise interrupt
- (3) interrupt handler invoked
- (4) get request
- (5) parse the request
- (6) system call execution
- (7) write return value
- (8) get return value

Processor \( q \):
- normal processing
- contention overhead
- \( t\text{CON} \)

**Diagram Notes:**
- \( t\text{RPC1} \) access
- \( t\text{RPC2} \)
- \( t\text{RPC3} \)
- \( t\text{RPC4} \)
- \( t\text{INT} \)
- \( t\text{INT}_1 \)
- \( t\text{INT}_2 \)
- \( t\text{SYS} \)
- \( t\text{RPC} \)
- \( t\text{RPC}_1 \)
- \( t\text{RPC}_2 \)

**Table 1:**
- \( t\text{INT}_1 \)
- \( t\text{INT}_2 \)
When accessed resource is located on p-local memory, the system call latency is \( t_{SYS} \). When a remote resource is accessed using RPC, the sum of \( t_{RPC1} \) to \( t_{RPC4} \), which are overheads of a remote service request, \( t_{INT} \), \( t_{LINT} \), and \( t_{INT} \) is added to \( t_{SYS} \).

\( t_{RPC1} \) is the time to prepare a request block and to link it to the request queue of processor \( q \). As the head and tail pointers to the request queue is located on \( q \)-local memory, some remote memory accesses are included in \( t_{RPC1} \). Let \( t_{RPC1} \) be the time to prepare a request block and to link it to a locally located request queue without any synchronization. Then, \( t_{RPC1} = t'_{RPC1} + t_{LOCK} + N_{ENQ} \cdot t_{RMA} + t_{REL} \), where \( N_{ENQ} \) is the number of remote memory accesses which are necessary to link a request block to the remote request queue. \( N_{ENQ} \) is 2-4 in typical implementations. \( t_{LINT} \) is the time until the interrupt is accepted. Its worst-case value is the sum of the maximum interrupt masking time and the processing time of interrupt handlers with higher or same priority.

Next, we investigate the processing time which processor \( q \) spends for the request from processor \( p \). The sum of \( t_{INT1} \), \( t_{RPC2} \), \( t_{SYS} \), \( t_{RPC3} \), and \( t_{INT2} \) is used for the request, in addition to the contention overhead time caused by the accesses of \( q \)-local memory from \( p \).

When processor \( p \) requests processor \( q \) to execute a system call in which a task blocking occurs (e.g. when a task executed on \( p \) tries to wait for a semaphore located on \( q \)-local memory), the unit of RPCs does not correspond with the unit of system calls. In this case, \( q \) informs \( p \) of the occurrence of a blocking using the return value. Then, \( p \) moves the running task to the sleep state, and begins to execute another task. In typical implementations of ITRON, the TCB of the waiting task is linked to the waiting queue of the waiting kernel resource. But, it is usually the case that \( q \) cannot access the TCB, because only one processor can access each control block. Moreover, when the task control block (TCB) is located on the memory inaccessible from \( q \), the TCB cannot be linked at all. One of the advantages of the RPC method is to enable a processor to access a kernel resource located on an inaccessible memory. To make the most of this advantage, a substitute for the TCB should be allocated on \( q \)-local memory and be linked to the waiting queue. When the blocking conditions is released on \( q \), \( q \) requests \( p \) to move the blocked task to the ready state using the same mechanism with RPC (with \( p \) and \( q \) exchanged).

There is a modified version of the RPC method, in which processor \( p \) executes another task while it is waiting for the return value. The idle time of \( p \) can be saved in this new method. However, because the processing time of each system call is very short in real-time kernels, a task switching overhead is relatively large compared with the system call execution time. Therefore, this method may degrade the runtime efficiency because of the overhead. It is also a problem that a system call invoked from an interrupt handler cannot use this method, because an interrupt handler cannot be blocked.

4.2 Resource Access via Direct Memory Access

The control flow of the resource access method via direct memory access is described below. We suppose the same case with the previous section (Figure 8).

(0) \( p \) branches to the service routine of the system call.
(1) \( p \) locks necessary control blocks.
(2) \( p \) executes the system call by accessing the local memory of \( q \).
(3) \( p \) raises an interrupt to \( q \) requesting task switching if necessary.
(4) \( q \) releases the lock, and returns from the system call.
(5) \( q \) switches the running task if necessary.

The step (3) and (5) is executed only when a task switching is necessary on processor \( q \) as the result of the system call.

We present a time chart of the direct memory access method in Figure 9. The lower figure illustrates the case that a task switching occurs on processor \( q \). The system call latency on \( p \) is prolonged by \( t_{LOCK} \), \( t_{INT} \), (when necessary), and \( t_{REL} \), as well as the remote memory access overhead. Let \( N_{SYS} \) be the number of remote memory accesses during the execution of the system call. Then, \( t'_{SYS} = t_{SYS} + N_{SYS} \cdot t_{RMA} \).
$N_{SYS}$ varies with the kind of the system call, the situation that the system call is invoked, and the implementation of the system call. We estimate $N_{SYS}$ by counting the number that a single processor ITRON kernel accesses control blocks during the execution of a system call. In a $\mu$TRON based kernel [8], the $wup\_tak$ system call, which is to wake up a specified task and is among the most important system calls for inter-task synchronization, accesses the TCBs about 4 times when the task to be waken up is already running or in the ready state (the wake-up request is enqueued in this case), and about 20 times when the task is waken up and begins to execute. The $sig\_sem$ system call, which is to release a task waiting for a semaphore, accesses the control blocks of the specified semaphore and the released task about 3 times when no task is waiting for the semaphore, and about 20 times when a task is released and starts to execute. The $wai\_sem$ system call, which is to wait for a semaphore, accesses the control block of the specified semaphore about 2 times when the task obtains the semaphore, and about 4 times when the task is blocked. It also accesses the TCBs in the latter case. Then, $N_{SYS}$ is 2–20 in typical inter-task synchronization calls.

Here, the control blocks are locked at once at the beginning, and are released at once at the end of the system call. In order to reduce lock contention and to enable processors to run in parallel, the control blocks should be locked with finer granularity. Of course, locking overhead becomes larger, because some locks must be obtained to execute a system call. Let $N_{LOCK}$ be the number of lock units which must be obtained during the execution of the system call. The overall locking overhead is $N_{LOCK} \cdot (t_{LOCK} + t_{REL})$. It is also possible to adopt non-blocking synchronization technique to some data structures. Though this method reduces lock contention overhead, the average throughput tends to be degraded.

In the direct memory access method, processor $p$ spares its processing time only for the contention overhead caused by the accesses of $q$-local memory from $p$.

4.3 Comparison of the Methods

We compare the RPC method and the direct memory access method in this section.

First, we compare the system call latency on processor $p$ of the methods. From the respective discussion, we should compare the following parameters (common terms are removed).

The RPC method:

$$t'_{RPC} + t_{LINT} + t_{INT} + t_{RPC} + t_{RPC} + t_{RPC}.$$
The direct memory access method:

\[(N_{SYS} - N_{ENQ}) \cdot t_{RMA}\]

We assume the case of a typical \textmu ITRON based kernel on a TRON specification chip for further discussion. Each of \(t_{RP1}, t_{INT},\) and \(t_{RPC}\) is 1-2\(\mu\)sec. \(t_{RPC}\) is almost same with \(t_{RMA}\), and \(t_{RPC}\) is less than 1\(\mu\)sec when the size of the return value is small). \(t_{RMA}\) is the maximum system call latency, which is important in real-time systems. In a typical \textmu ITRON based kernel on a TRON specification chip, maximum interrupt masking time is about 100-500\(\mu\)sec. Since \(N_{SYS}\) is 2-20 in typical system calls as mentioned above, \((N_{SYS} - N_{ENQ}) \cdot t_{RMA}\) is 0-10\(\mu\)sec. If processor \(q\) does not mask interrupt when \(p\) raises an interrupt, \(t_{INT}\) is 0. In this case, both methods have almost the same efficiency. Precisely, the direct memory access method is more efficient for simple system calls, and the RPC method is more efficient for complex system calls.

However, \(t_{INT}\) has a great influence on the maximum system call latency, which is important in real-time systems. In a typical \textmu ITRON based kernel on a TRON specification chip, maximum interrupt masking time is about 10\(\mu\)sec. If processor \(q\) also serves interrupts from I/O devices, the maximum length of \(t_{INT}\) is prolonged by the processing time of the interrupt handlers. Consequently, the maximum system call latency is much longer in the RPC method, and we conclude that the RPC method is not suitable for real-time kernels.

In the above comparison, we do not consider the case that some processors try to access a kernel resource located on a local memory at the same time. In the direct memory access method, the access collision is appeared as lock contention, and \(t_{LOCK}\) becomes long. In the RPC methods, because processor \(q\) can serve only one request at once, the interrupt request from a later processor is delayed and \(t_{INT}\) becomes long. Lock contention for the request queue also occurs. In both methods, the system call latency becomes long because of the access collision.

In the direct memory access method, the probability of lock contention and the waiting time for a lock can be reduced by using finer grained lock units and by locking only necessary control blocks [6]. As finer grained lock units have some additional locking overhead as mentioned before, the best lock granularity varies with the nature of the application program. In this aspect, the direct memory access method has an adaptability to an application, which is one of the design goals of ITRON-MP.

Next, we compare the penalty on the processing time of processor \(q\) caused by the kernel resource access by processor \(p\). In the direct memory access method, the penalty is only the local memory access contention overhead, while much more processing time of \(q\) is used in the RPC method. Because one memory access contention overhead is almost the same with the access time of the memory, total overhead is still quite small. Then, the direct memory access method is also preferable from this criterion.

Moreover, the RPC method has some difficulties in the management of some kernel data area. One of them is the return value block area. In ITRON2 specification [9], some system calls receive a pointer to a memory area and write the return data to the area. However, passing the pointer to processor \(q\) is not a solution, because \(q\) possibly cannot access the memory area pointed by the pointer. In this case, \(q\) must write the return data to the return value block. Because the size of the memory area is not bounded statically in some system calls (e.g. \texttt{striction} and \texttt{source}), efficient allocation of the return value block is difficult.  

The other is the area of the substitutes for TCBs. Because the maximum number of the substitutes for processor \(q\) is the maximum number of tasks that may wait for kernel resources located on \(q\)-local memory, it wastes the memory area to allocate sufficient substitutes for the maximum case.

The above is the comparison of the methods to access kernel resources located on a remote memory. The discussion is almost parallel for kernel resources located on a global memory. \(t_{RMA}\) is replaced by \(t_{GMA}\) which is the global memory access overhead compared with a local memory. As the global memory access overhead also affects the RPC method, \(t_{SYS}\) should be replaced with \(t_{GMA}\). But, with an effective cache mechanism, \(t_{SYS}\) is not much longer than \(t_{SYS}\), because each control block is accessed from one processor. The memory access contention on a local memory does not occur in this case (i.e. \(t_{CONT} = 0\)). Consequently, the direct memory access method can take the advantage of the global memory through relieving \(q\) from the local memory access contention overhead, while the RPC method cannot.

As the result, the direct memory access method is appropriate for real-time kernels in typical environment. However, in some special situations, the RPC method is valid. For example, when the return value of a system call is not necessary, which is often the case for system calls returning only error codes, processor \(p\) need not wait for the return value from processor \(q\). This resolves some serious problems of the RPC method. Actually, it is an ITRON implementation using

\[4\text{This problem does not occur in } \mu \text{ITRON.}\]
Time to put a request to the request queue.  
Time to get a request and to parse it.  
Time to write a return value to the return value block.  
Time to get a return value.  
Time to raise an interrupt.  
Interrupt latency.  
Interrupt handler startup time (saving registers, etc.).  
Interrupt handler finish time (restoring registers, etc.).  
System call execution time accessing a local resource.  
System call execution time accessing a remote resource.  
Time to lock a remote resource.  
Time to release a remote resource lock.  
Remote memory access overhead compared with a local memory.  
Memory access contention overhead.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{RPC1}$</td>
<td>Time to put a request to the request queue.</td>
</tr>
<tr>
<td>$t_{RPC2}$</td>
<td>Time to get a request and to parse it.</td>
</tr>
<tr>
<td>$t_{RPC3}$</td>
<td>Time to write a return value to the return value block.</td>
</tr>
<tr>
<td>$t_{RPC4}$</td>
<td>Time to get a return value.</td>
</tr>
<tr>
<td>$t_{INT}$</td>
<td>Time to raise an interrupt.</td>
</tr>
<tr>
<td>$t_{INT1}$</td>
<td>Interrupt latency.</td>
</tr>
<tr>
<td>$t_{INT2}$</td>
<td>Interrupt handler startup time (saving registers, etc.).</td>
</tr>
<tr>
<td>$t_{SYS}$</td>
<td>Interrupt handler finish time (restoring registers, etc.).</td>
</tr>
<tr>
<td>$t_{SYS}$</td>
<td>System call execution time accessing a local resource.</td>
</tr>
<tr>
<td>$t_{SYS}$</td>
<td>System call execution time accessing a remote resource.</td>
</tr>
<tr>
<td>$t_{LOCK}$</td>
<td>Time to lock a remote resource.</td>
</tr>
<tr>
<td>$t_{REL}$</td>
<td>Time to release a remote resource lock.</td>
</tr>
<tr>
<td>$t_{RMA}$</td>
<td>Remote memory access overhead compared with a local memory.</td>
</tr>
<tr>
<td>$t_{CONT}$</td>
<td>Memory access contention overhead.</td>
</tr>
</tbody>
</table>

Table 1: Time Parameters

4.4 Combination Methods

As mentioned above, the RPC method is effective in some situations. We describe combination methods in this section.

One of effective combination methods is to use the appropriate one for each kernel resource. For example, it is effective that the direct memory access method is used for the kernel resources on a global memory and that the RPC method is used for the resources on a local memory with high remote access cost. It is also possible to use both methods for one kernel resource. Complex system calls should be realized using the RPC method. In this combination method, explicit locking of control blocks is also necessary in the RPC method.

Another combination is to use both methods in a system call. We pointed out that control blocks are accessed about 20 times in the $imp_tas$ system call, when a task is actually waken up and begins to execute. Most of the accesses are used to move the running task to the ready state and the waken-up task to the run state. Then, the following method is considered to be effective. Processor $p$ changes the status of the waken-up task, and requests processor $q$ to do the remaining jobs, such as the handling of the ready queue. In this method, $p$ can determine the return value of the system call by its own processing, $p$ need not wait for the return value from $q$. A difficulty of this method is the temporal inconsistency of the control blocks which occurs between the processing of $p$ and the processing of $q$. Even if some other processors read this inconsistent status, such an inconsistency must be hidden from programmers.

4.5 Related Work

Chaves et al. adopt these two methods to a UNIX-like operating system for a NUMA architecture machine, and compare them in detail [11]. They conclude that the RPC method has better performance at least in their evaluation environment.

The difference of their result with ours comes from the difference between real-time kernels like ITRON and kernels of UNIX-like operating systems. The followings are the key differences.

- In real-time kernels, the functionality of a system call is small compared to UNIX-like operating systems in general. Then, the number of accesses to control blocks in a system call (i.e. $N_{SYS}$) is very small, and the total overhead of the direct memory access method is not large. On the contrary, the overhead of the RPC method is almost constant (depends only on the number and the size of the parameters and the return value).

---

6In strict, they classify the RPC method into two variations. But, one of them cannot be applied to real-time kernels like ITRON.
- When the granularity of lock units is fine, the number of lock units obtained in a system call (i.e., $N_{\text{LOCK}}$) is smaller in real-time kernels by the same reason. Then, the resource lock overhead is small.

- In real-time kernels, predictability of the execution time is very important, while workstations require high total throughput. As the system call latency of the RPC method tends to change with the status of the service processor, the RPC method is not good for real-time kernels.

5 ITRON-MP Specification

As described in the previous section, the direct memory access method or a combination method is preferable for real-time kernels. Therefore, we design the ITRON-MP specification to be realizable using the direct memory access method.

Due to the restriction of the direct memory access method that a processor cannot access a kernel resource located on an inaccessible memory, some access restriction appears in the kernel interface. Because the RPC method is more flexible in this sense, ITRON-MP can also be implemented using the RPC method. As the result, the ITRON-MP specification reflects the asymmetry of an architecture to its kernel interface.

On the other hand, most parallel or distributed operating systems adopt the approach to conceal the asymmetry from their kernel interface and to provide programmers with the location transparency of resources. This approach has an advantage that programmers need not keep resource accessibility in mind. However, when programmers construct an application with hard real-time constraints, they must be conscious of the run-time efficiency of the program, which heavily depends on the physical location of the resources. The asymmetric kernel interface of ITRON-MP keeps programmers conscious of the run-time efficiency of the program they are writing. This approach also conforms to the design policy of the ITRON specification that excessive abstraction of kernel resources should be avoided. In the TRON project, the BTRON specification is an operating system providing resource transparency [4], while ITRON is used for applications requiring tight real-time property.

In order to show the asymmetry of kernel resources to programmers, ITRON-MP classifies kernel resources according as their characteristics, such as the accessibility and the access efficiency from each processor. Tasks, which are a kind of kernel resources, are further classified by their executability by each processor. These characteristics of kernel resources come from the characteristics of hardware resources on which they are implemented. We describe the classification of hardware resources first, and then the classification of kernel resources.

5.1 Classification of Hardware Resources

We classify hardware resources from two viewpoints; physical location and accessibility. Access efficiency of hardware resources are mostly determined by their physical location. Suppose an example architecture of Figure 5. From the viewpoint of physical location, hardware resources are classified with the global resources (resources directly connected to the backplane bus) and the local resources of a processor (resources connected to the local bus of the processor). From the viewpoint of accessibility, they are classified with the shared resources (resources accessible from all processors) and the private resources of a processor (resources accessible only from the processor). As global private resources do not exist, hardware resources are classified with global shared resources, local shared resources, and private local resources. We simply call these classes global resources, local resources, and private resources, respectively.

An application system which should never stop working utilizes on-line maintenance technology which enables the insertion and deletion of a board while the system is working. Hardware resources are added to or removed from the system as the unit of a board. In this case, the physical location of a hardware resource includes two meanings; the access efficiency and the replaceable units.

5.2 Classification of Kernel Resources

A kernel resource of ITRON-MP is implemented using the same class of hardware resources in principle. Then, kernel resources are classified by their accessibility and their physical location, according to the classification of hardware resources implementing them.

As the classification of kernel resources represents the kernel model from the programmers' view, there are some exceptions to this principle. For example, a kernel resource implemented on private hardware resources of a processor can be accessed from other processors using the RPC method. In this case, the kernel resource is classified to be accessible from other processors. Another example is that a private kernel resource may be implemented using local hardware re-
sources, if their access cost is almost the same with the access cost of private hardware resources.

5.3 Classification of Tasks

A task, which is a kind of kernel resources, requires a program code region, a data region, a stack region, a TCB region, and so on. In this case, the principle that a kernel resource should be realized on a class of hardware resources is too restrictive. In ITRON-MP, only the stack region and the TCB region are managed by the kernel, and the management of other regions are left to the linker or the loader. Moreover, in typical implementations, the accessibility of a task agrees with the accessibility of its TCB. Accordingly, a task is classified by the class of the hardware resource on which its TCB resides.

Tasks are further classified by their executability. A processor could execute a task, if all hardware resources for the task are accessible from the processor. However, if the access cost of the hardware resources from the processor is very high, it should not execute the task in order to achieve high run-time performance. Because the high run-time efficiency is among the most important features of ITRON-MP, the classification of tasks by their executability should agree with the classification by the access efficiency of the hardware resources on which the tasks are located in principle. By this simplification, the classification of tasks agrees with the classification of other kernel resources.

In the architecture of the previous example, it greatly degrades the run-time efficiency that processor $p$ executes a task whose program code region and data region reside on $q$-local memory. Consequently, the task is usually made to be executable only on $q$ in this case. It is also inefficient to execute a task located on the global memory only on one processor, because it wastes the bandwidth of the backplane bus. Then, the tasks are classified with the global tasks (tasks accessible and executable by all processors), local tasks (tasks accessible from all processors and executable only by a processor), and private tasks (tasks accessible and executable by a processor) in this example.

5.4 Resource Accessibility from Tasks

In the previous sections, kernel resources are classified by their accessibility from processors. However, the accessibility of kernel resources from tasks is more important for programmers.

The accessibility of each class of kernel resources from each class of tasks in the previous example is shown in Table 2. In this table, $p$-private tasks can access shared resources, but cannot wait for them. The reason is as follows. If a $p$-private task $T_1$ waits for a shared kernel resource such as a semaphore, task $T_2$ executed on processor $q$ which tries to access the resource cannot wake up the task $T_1$ because $q$ cannot access the TCB of $T_1$.

We can naturally extend this accessibility rule to a general case. For any sets of processors $S$, $T$, and $U$, a $U$-shared $T$-bound task (a task executable by the processors in $T$ and accessible from the processors in $U$) can access and wait for an $S$-shared kernel resource (a resource accessible from the processors in $S$), if and only if $T \subseteq S \land S \subseteq U$. The task can access but cannot wait for the resource, if and only if $U \subseteq S$.

There is one exception to this accessibility rule. The $rel\text{wait}$ system call of ITRON forcibly releases a specified task from the wait state. Suppose a case that a $p$-local task $T_1$ is waiting for a $p$-private semaphore $S$. Because a $q$-local task $T_2$ can access $T_1$ by the accessibility rule, $T_2$ could release $T_1$ from the wait state. However, because the TCB of $T_1$ is linked to waiting queue of $S$ in typical implementation, the semaphore control block of $S$ must be accessed from $T_2$ to release $T_1$, but it is not the case. Consequently, implementors of ITRON-MP is permitted to restrict the usage of the $rel\text{wait}$ system call.

5.5 Kernel Resource ID Scheme

A kernel resource is accessed through its ID number in ITRON-MP. The ID number of a kernel resource consists of the field representing the class to which the resource belongs and the field identifying the resource in the class. When a new kernel resource is created, the ID number of the resource to be created must be given in ITRON-MP. Then, which hardware resource to be used for the kernel resource can be determined from the class field of the ID number. By this resource ID scheme, almost all system calls become compatible with those of ITRON, and porting of a program developed for ITRON to ITRON-MP becomes easy.

The class field of a kernel resource ID can be expressed in two ways. One of them is the absolute method in which a class ID uniquely identifies a resource class in the system, and the other is the relative method in which a class is specified in relative to the class to which the accessing task belongs. For example, the relative method is effective, when a local task of a processor accesses its private resource.

In ITRON-MP, the absolute method is standard, because it agrees with typical shared-memory multiprocessor systems. The ID number of each class
Table 2: Accessibility of Kernel Resources from Tasks

<table>
<thead>
<tr>
<th></th>
<th>global task</th>
<th>p-local task</th>
<th>p-private task</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared resource</td>
<td>accessible</td>
<td>accessible</td>
<td>only accesses without waiting are possible</td>
</tr>
<tr>
<td>p-private resource</td>
<td>inaccessible</td>
<td>accessible</td>
<td>accessible</td>
</tr>
<tr>
<td>q-private resource</td>
<td>inaccessible</td>
<td>inaccessible</td>
<td>inaccessible</td>
</tr>
</tbody>
</table>

is fixed when the kernel is generated. The relative method is also permissible in ITRON-MP as an option, as it is suitable for some architectures or applications. Non-positive values are used to specify a class in relative. Especially, zero is reserved for the code designating the class to which the accessing task belongs.

6 Conclusions

This paper describes the purpose, the application areas, and the design goals of ITRON-MP, and discusses the implementation methods of synchronization and communication mechanism between processors in detail. As the result, we conclude that the direct memory access method is more appropriate for real-time kernels than the RPC method. Consequently, the ITRON-MP specification is designed to be realizable with the direct memory access method.

Because the existence of shared memory quite improves the run-time performance, we separate the ITRON-k1P specification, which is a kernel for multiprocessor systems, and the ITRON-N specification, which is a kernel for systems without shared memory.

We plan to integrate ITRON-MP and ITRON-N to IMTRON, and investigate various issues on IMTRON which is an important part of HFDS.

Acknowledgments

We wish to thank Mr. Nobuhiro Nishio, Mr. Noboru Koshibu, Mr. Hirohisa Mori and other colleagues of Sakamura Laboratory for their suggestion and encouragement.

References