An Experimental Implementation of Unified Real-Time Operating System

Koichi Sato, Hideo Tsubota, Osamu Yamamoto, and Kazunori Saitoh
Mitsubishi Electric Corp., LSI Laboratory
4-1 Mizuhara, Itami, Hyogo 664, Japan

Abstract
We are developing an unified OS which provides superior real-time response and versatile functionality simultaneously. To accomplish both features, this OS consists of two kernels. One is called v-kernel and has large number of system calls (about 130). The other is called p-kernel and is characterized by short interrupt masking time (maximum 15 µsec). The v-kernel is compressed into a task in the p-kernel in order to achieve short interrupt latency. Using facilities of inter-kernel communication and synchronization, this OS makes it possible to realize a decentralized real-time application system by distributing p-kernels to multiple MPUs.

1 Introduction
Recently faculties of microprocessors (MPUs) have improved and their applications have become to handle complicated jobs more speedily. As a result, tools become important to develop applications utilizing the features of MPUs in a short time. A real-time operating system (OS) is one of those tools. Applications are requiring further functionality from an OS. Moreover, facilities to support MMU for reliability and virtual memory mechanism for large scale systems are also needed. On the other hand, high speed and predictable response time are increasingly required to achieve superior real-time performance.

For example, Fig.1.1 shows construction of application program using a real-time OS. Application tasks are classified into an upper part and a lower part. Tasks in the upper part process heavy jobs such as data process, user interface and debugger. Tasks in the lower part control hardware devices and execute I/O processes required by the upper part. The former requires versatile functionality and computation power. The latter requires real-time performance. So an OS should provide a suitable execution environment which would meet different needs of individual tasks. However it's difficult to realize such an OS because versatile functionality and real-time performance conflict with each other.

One of the methods to solve this problem is that tasks and their suitable OSs are distributed across multiple MPUs. Special issues remain to be solved in heterogeneous multi-processor environments. The issues include increased cost of using multiple OSs and MPUs, complicated programming, no synchronization and no communication methods between OSs and so on. After all, multi-processor systems also need OSs which are compatible with corresponding single-processors. Moreover, architectures of multi-processor systems are well-discussed in the point of synchronization and communication to achieve high performance, but not much in the point of real-time efficiency. Therefore there exists the necessity for an OS which provides both versatile functionality and real-time performance and can be adapted easily to a multi-processor system with unified manner of single-processor. We call this OS "unified OS".

The TRON project[1][2] has designed the µTRON specification and the ITRON2 specification as real-time OSs. We have already completed developing these two OSs as shown in Table1.1. MR3200[3] based on the µTRON specification is characterized by short interrupt masking time (15µsec) and quick task switching (about
Table 1.1 Characteristics of the MR3200 and the MR3210

<table>
<thead>
<tr>
<th></th>
<th>MR3200</th>
<th>MR3210</th>
</tr>
</thead>
<tbody>
<tr>
<td>specification</td>
<td>µITRON2 spec.</td>
<td>ITRON2 spec.</td>
</tr>
<tr>
<td>number of system calls</td>
<td>53</td>
<td>129</td>
</tr>
<tr>
<td>interrupt masking time (worst case)</td>
<td>15μsec</td>
<td>30μsec</td>
</tr>
<tr>
<td>system call execution time</td>
<td>wup_task 17.9μsec (1) 140μsec (2) 69μsec</td>
<td>set_flg 21.6μsec (1) 31μsec (2) 76μsec</td>
</tr>
</tbody>
</table>

Note: (1) the time without task dispatch
(2) the time with task dispatch

20 μsec). MR3210[4] based on the ITRON2 specification is characterized by rich functionality and large number of system calls. In order to solve the above-mentioned problems, we implemented a unified OS by combining these OSs. The MR3200’s kernel is compressed into one task running on top of the MR3200’s kernel. This OS has facilities of inter-kernel’s synchronization and communication, and also provides real-time processing which is compatible with a corresponding single-processor and unified manner to a multiple processing system.

This paper describes concept of the unified OS, implementation techniques and the performance of this OS implemented on a single MPU. Adaptability of this OS to a multi-processor environment is also discussed.

2 Issues to be solved in a real-time OS

In order to adapt real-time OS to wider application fields, issues to be solved in a real-time OS are discussed.

2.1 OS’s overhead and facilities required by task

Individual data for a task are stored in its TCB (Task Control Block). In order to support exception management and dynamic object creation, for example, the size of a TCB is large. It also takes long time to check exceptions and the existence of objects. If tasks use FPU or tasks are in the debugging state, additional saving area is required. Kinds of context to be saved is checked and switched whenever task dispatch occurs. These are mere OS’s overhead for tasks not requiring such facilities or for tasks only requiring superior real-time response. So it is important that an OS provides suitable functionality and performance to individual task.

2.2 Isolation of task execution environment

In the ITRON specification, objects are designated with ID numbers which are assigned statically during system development. If application programs are developed by plural programmers, duplicate ID numbers are often assigned by each other. In distributed systems, this issue must be solved with dynamic ID number allocation or other methods. Even if dynamic ID number allocation is introduced, facility to share objects using ID number or other identifiers are needed because ID numbers of shared objects are determined on run time.

Another issue is that priorities determining execution order of tasks are managed together in a whole system. One application program will be divided into some tasks whose priorities are assigned by considering execution order and timing constraints. But execution order is affected by tasks which have higher priority and have no relation to this application.

These issues become serious in large scale and complicated application programs. They also make it difficult to integrate multiple applications on a system. So facilities to couple tasks, isolate and connect execution environment of coupled tasks are needed.

2.3 Real-time response affected by mutual exclusion

Objects provided by OS include tasks, semaphores, mailboxes and so on. Tasks which invoke system calls are only active objects. In an OS, system calls can be also invoked in interrupt handlers in order to realize device drivers for hardware control. Object access between tasks or between tasks and interrupt handlers may conflict. In order to avoid this conflict and to assure consistency of object state and system data, mutual exclusion needs to be established in the OS. Fig.2.1 shows mutual exclusion methods in system call execution.

(1) Inter-task: Mutual exclusion for inter-task is realized using delayed dispatch while tasks execute system calls. So there is no interrupt latency because of no mutual exclusion between tasks and interrupt handlers.

(2) Between task and interrupt: Hardware controls and interrupt handling are not executed by each task but by device drivers composed of dedicated tasks and interrupt handlers as shown in Fig.1.1. Objects used in a device driver are shared by tasks and interrupt handlers, so it is necessary to make mutual exclusion for them by inhibiting interrupts.
(3) Inter-MPU: Adding to aforementioned methods, mutual exclusion for inter-MPU as shown in Fig.2.2 are needed in a multi-processor system. This mutual exclusion forces requesters to wait for specified object to be unlocked if it is locked by a task or a interrupt handler on another MPU. Waiting time is also dependent on scheduling order to be given the right to lock. Interrupt latency can be fixed by improving algorithm of mutual exclusion shown in Fig.2.2(b). But the execution time is not predictable. So system calls adopting this mutual exclusion can't be used by real-time processes aimed at interrupt handling and time bound tasks.

As aforementioned reasons, homogeneous object management increases interrupt latency and system call execution time and this affects real-time response. Interrupt latency is also affected by increase of OS's facilities because part of process shown in Fig.2.1 becomes longer. Therefore real-time response should be accomplished by classification of object's usage and adopting of mutual exclusion according to it.

2.4 Scheduling principle in multi-processor systems

Task scheduling occurs due to system call's invoked from both tasks and interrupt handlers. In an OS for a single MPU, mutual exclusion between tasks and interrupt handlers assures consistency of ready queue state by delaying task dispatch and inhibiting interrupt when these two operations conflict. The MPU runs the task with READY state that has the highest priority after interrupt handlers are completed.

In multi-processor systems, real-time response is lost due to mutual exclusion as shown in Fig.2.1(d), if both tasks to be accessed and not to be accessed by the interrupt handler are managed on the same ready queue. In order to reduce overhead derived from mutual exclusion, the ready queue must be separated into two ready queues. One is for MPU bound tasks which execute real-time process with hardware. The other is for MPU unbound tasks which require computation power achieved by multi-processor. In this method, real-time kernel have to reside in each MPU. So the kernel should be small in order to reduce the size of its code and data.

3 Concept of the unified OS

Taking aforementioned issues into consideration, we investigated a unified OS to meet various needs required by applications.

3.1 Construction and effect of unified OS

Fig.3.1 shows construction of the unified OS. In order to solve issues described in section 2, the unified OS is composed of two kernels, primary/physical kernel (called p-kernel) and versatile/virtual kernel (called v-kernel). V-kernel is compressed into one task in unified p-kernel. These kernels have following features.

![Fig.2.1 Mutual exclusion](image)

![Fig.2.2 Mutual exclusion for multi-processor](image)
*p-kernel
• Predictable and quick response
• Facilities needed for interrupt handling
• Resident kernel to execute tasks bound with MPU and hardware
• Small size to be allocated to each MPU in multi-processor system (~10KB)

*v-kernel
• Versatile functionality and reliability
• Facilities to distribute task's load to MPUs
• Scheduling principle to take advantage of computation power derived from multi-processor

In the unified OS, following effects are accomplished by constructing OS hierarchically.

(1) Quick response to an interrupt: Interrupt handling is executed in only p-kernel having light functions. Interrupts are enabled all the time in v-kernel because v-kernel is compressed into one p-kernel's task and executes without interrupts disabled. So quick response to interrupts and versatile functionality are achieved simultaneously on a single MPU.

(2) Easy enhancement of functionality: It is easy to enhance functionality without affecting p-kernel's real-time response because v-kernel is one of p-kernel's applications. As p-kernel resides in physical memory and executes processes bound with hardware, kernel supporting virtual memory mechanism can be implemented in v-kernel.

(3) Multiple execution environment: To allocate identifiers of object and to determine execution order of tasks are important for multiple applications in a system. An application consists of tasks which are related to each other. In an unified OS, light kernel is compressed into p-kernel's task to manage objects and task scheduling. Application works on this light kernel as shown in Fig.3.2. So ID numbers of objects are isolated from other applications. Objects or facilities shared by some applications are realized using p-kernel's tasks. Their ID numbers are reserved and indicate location with manner described in 3.3.

Once an application starts, its execution order isn't changed by other applications. In the ITRON specification, chg_pri is provided to change a task's priority dynamically. Chg_pri is effective to determine execution order in an application but doesn't affect other application execution environment. In a whole system, execution order of each application is determined with priorities assigned to the p-kernel task. Unit of execution is an application program located in each kernel.

(4) Classified usage of objects: Usage and access right of objects provided by p-kernel and v-kernel are defined in table 3.1. As a result of consideration for real-time response affected by mutual exclusion, direct access method to v-kernel's objects from interrupt handlers is eliminated. This decision brings significant effect to multi-processor system. That is, if this access is
allowed and v-kernel is located in shared memory to distribute task's loads to MPUs, real-time response is lost due to inter-MPU mutual exclusion as shown in Fig.2.1(d). In the unified OS, this direct access isn't allowed.

Tasks in p-kernel and v-kernel have access right to objects in both kernels in order to communicate and synchronize with each other. For example, tasks in v-kernel can issue I/O request to device drivers in p-kernel and receive notice of completion from them. In case of access from v-kernel to p-kernel, the same real-time response as p-kernel is assured because p-kernel is called by one of p-kernel's task. In case of access from p-kernel to v-kernel, extra mutual exclusion isn't needed because no v-kernel's tasks are executed at this time and no interrupt handlers call the v-kernel. So these accesses to objects don't affect real-time response.

(5) Scheduling order for applications: P-kernel and v-kernel have separated ready queue, and only the p-kernel deals with interrupts. So tasks requiring real-time response are managed by the p-kernel, and tasks requiring high functionality are managed by the v-kernel. In a multi-processor system, v-kernel may reside in shared memory to distribute tasks' load to MPUs and is executed as p-kernel's task running on each MPU as shown in Fig.3.3. That is, response time to interrupts assured by p-kernel isn't affected. Compressing v-kernel into a p-kernel's task with lowest priority gives priority to drivers and system tasks in p-kernel. In this case, execution of v-kernel's tasks are pre-empted when interrupts occur and system calls are issued in them. The wake up of v-kernel's task by p-kernel is used to notice completion of process requested by v-kernel, and is postponed until no READY task exist in p-kernel. So there is no way from p-kernel to v-kernel with execution of v-kernel in a p-kernel's task. Fig.3.4 shows scheduling order and program's state transition in the unified OS.

3.2 Unified manner based on ITRON specification

It is important to provide unified manner between p-kernel and v-kernel in order to avoid user's cost derived from understanding and development. And manner to develop applications on multi-processor system should be compatible with that of single-processor system. In ITRON project, which is a sub-project of TRON project, the μITRON specification aiming at quick response and the ITRON2 specification aiming at high functionality were designed. These are unified as ITRON specification regarding system call names, their operations and some manners. So we decided to conform unified OS to the

Table 3.1 Access right to object from each kernel

<table>
<thead>
<tr>
<th>capability</th>
<th>p-kernel</th>
<th>v-kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>of access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interrupt task</td>
<td>allowed</td>
<td>allowed</td>
</tr>
<tr>
<td>interrupt task</td>
<td>not allowed</td>
<td>allowed</td>
</tr>
</tbody>
</table>

Fig.3.3 Unified OS applied to multi-processor system

Fig.3.4 Program state transition in Unified OS
ITRON specification, the p-kernel is based on the μITRON specification and v-kernel is based on the ITRON2 specification.

3.3 Objects designated with ID number

In the ITRON specification, objects are designated by ID number. In order to provide the same manner as ITRON specification, p-kernel or v-kernel in the unified OS must be designated with ID number. So unified OS's ID number has two meanings as shown in Fig.3.5. One is object ID and the other is registration information to indicate kernel or node in which objects reside. Facilities to add registration information to object ID are provided in the unified OS.

From standpoint of application programs, system call interface is same as ITRON specification. So it is necessary in programs only to the alter assignment of object's registration without modifying systems calls when construction or connection of the system are changed. It is easy to adapt a single-MPU system to multi-processor system using bus or network.

3.4 Performance required to unified OS

An aim of the unified OS is to achieve high functionality and real-time response simultaneously without affecting single kernel's performance. So overheads caused by the unified OS are estimated and design goals are set as below.

1) Overheads determining kernels: It takes a few micro seconds to decide which object of two kernels is designated. These overheads are allowed. Because to unify the manner conforming to the ITRON specification is more important than the cost of these overheads.

2) Execution time with intra-kernel: It is self defeating to increase the cost of task switching or system call execution intra-kernel by unifying the OS. That is, performance of the unified OS must be close compared with one of single kernel. In the unified OS, there is very little affection except for examination of ID number. Scheduler and dispatcher are separated into p-kernel and v-kernel.

(3) Execution time with inter-kernel: In the unified OS, tasks in both kernels may issue system calls to each other. One of features expected by unified OS is to achieve high speed execution of tasks having light context and hardware processing with using p-kernel. So it is important to execute operation from the v-kernel task to p-kernel task quickly. In order to obtain effect by the unified OS, execution time of system call issued from v-kernel to p-kernel must be over two times comparing with v-kernel's.

3.5 Adaptability to multi-processor system

Asymmetric architecture as shown in Fig.3.3 is effective for real-time application fields. In this architecture, p-kernel dealing with light tasks and interrupt handing resides in local memory for each MPU, because interrupts are local events for MPUs connected to I/O devices. Employing symmetric architecture and monolithic kernel, mutual exclusion between MPUs as shown in Fig.2.1(d) should be used and then quick and predictable response can't be achieved.

In the unified OS adapting to multi-processor system, access right to objects and program state transitions are same as shown in table3.1 and Fig.3.4 respectively. The difference from single-processor system is that v-kernel's tasks are executable on multiple MPUs in order to obtain high computation power. So v-kernel, v-kernel's objects and task's programs reside in global memory. If v-kernel's tasks work on a MPU, they can't
make direct access to local resources of other MPUs. In such case, scheduler in v-kernel forces them to work on the MPU having local resources to be accessed as shown in Fig.3.6. Dispatcher is invoked with an interrupt to that MPU. When no tasks with higher priority are working on that MPU, the task is allowed to access local resources in local memory. As another method, RPC (stands for remote procedure call) may be used. But it causes two overheads as compared with direct access method. One is that tasks to process the RPC service are needed on each MPU. The other is that task switch is occurred four times because v-kernel's tasks request to/wait for completion from RPC processing task and RPC servicing tasks wake up requester/wait for next request. Therefore direct access method is adopted in the unified OS.

In case of reverse access direction, from p-kernel to v-kernel, p-kernel's tasks can make direct access to v-kernel's objects which reside in global memory. When task dispatch is required after the operation by p-kernel, it is postponed until completion of the p-kernel application because of compressing v-kernel into p-kernel's idle task. So p-kernel have only to put v-kernel's task on global ready queue resided in global memory without task dispatch.

4 Experimental implementation of an unified OS on a single MPU

We implemented an unified OS on a single MPU, M32/100 which is based on the TRON specification, in order to evaluate its efficiency. We paid attention to keep high speed performance and short interrupt masking time of the MR3200. We also provide synchronization and communication mechanism between tasks executing on different kernels. In this section, we mention implementation techniques and result of evaluation about this OS.

4.1 Structure of the unified OS

Fig.4.1 shows the structure of the unified OS implemented on a single MPU. The v-kernel is compressed into an idle task of the p-kernel; the idle task is one of the tasks that has the lowest priority in that kernel. Interrupt handlers are executed under the p-kernel. Ready queue of the unified OS has dual structure as shown in Fig.4.2. Ready queue of the v-kernel is connected to the idle task of the p-kernel. This structure fixes relation of task priorities between two kernels automatically, i.e. if there exist RUN or READY state tasks in the p-kernel, tasks in the v-kernel will never be executed.
when it causes an event flag to satisfy a condition for which a task is waiting, the task is released from its waiting state and becomes ready to run. The other way is used when the issuing task enters the WAIT state, for example wai_flg, which waits for one or some combination of bits to be set in the event flag.

If the WAIT state cannot be entered, the requesting kernel informs the other kernel with TRAPA interface directly. Fig.4.3(a) shows the mechanism of inter-kernel operation from the v-kernel to the p-kernel. In this case, the p-kernel handles the request the same as a request from an interrupt handler. Ret_int, one of the system calls in the µTRON specification which is used at the end of interrupt handlers, is executed at the end of the p-kernel service routine. The reason is that it is necessary to dispatch from the v-kernel to the p-kernel if some tasks in the p-kernel have entered the READY state after the system call is issued by v-kernel's task. On the other hand, Fig.4.3(b) shows the mechanism of inter-kernel operation from the p-kernel to the v-kernel. TRAPA interface is also used as the same as Fig.4.3(a). But, in this case, there is no need to dispatch from the p-kernel to the v-kernel because priority of the v-kernel is the lowest, so only REIT instruction is used at the end of the v-kernel service routine.

If the WAIT state can be entered, communication server task in the other kernel is used to work on behalf of the task which has issued system call from one kernel, as shown in Fig.4.4. Server task is needed because the issuing task doesn't have its context in the other kernel and can't enter the WAIT state. When a system call is issued to wait for the other kernel's object, it is sent to server task using snd_msg, which sends a message to another task using a mailbox, the issuing task waits in the kernel on which it runs. The server task receives the message and executes the system call on behalf of the issuing task. After the system call is completed, the server task wakes up the issuing task using wup_tsk, which changes a task from WAIT state to READY state or RUN state. The number of server tasks is determined in the system configuration.

4.4 Delayed interrupt and delayed context trap functions of the TRON spec MPU

The unified OS realizes mutual exclusion using Delayed Interrupt (DI) and Delayed Context Trap (DCT) functions of the TRON specification MPU. DI and DCT are hardware mechanism to activate some specified routines, called DI handler and DCT handler.

DI handler is activated when the IMASK value of the PSW register is greater than the DI value of the DI register. DI handlers are defined for each DI value, 0-14, respectively. For example, when the IMASK value is 13, setting 13 the DI register to 13 won't activated the DI handler immediately. Later, when the IMASK value is changed to 14 or 15, the DI(13) handler is activated.

DCT handler is activated when the SMRNG value of the PSW register is less than the DCT value of the CSW register. If the DCT value is equal to 7, DCT handler is never activated. If the DCT value is equal to 0, this handler is always activated except when the stack mode is SPI.

4.5 Mechanism of mutual exclusion and task dispatch

Table 4.1 shows the execution modes of tasks and system calls in the unified OS. With using delayed handlers as shown in table 4.2, we realized delayed task dispatch and mutual exclusion while the OS executes in a critical section. We will describe about the mechanism of mutual exclusion and task dispatch in case by case.

![Inter-kernel operation mechanism(1)](image1)

![Inter-kernel operation mechanism(2)](image2)
(1) Mutual exclusion in v-kernel: Mutual exclusion among tasks in the v-kernel is realized by using DI(14) handler; i.e. system calls of the v-kernel are executed in interrupt mask level 14 (IMASK=14) and task dispatch is done by DI(14) handler. By this way, task dispatch among v-kernel's tasks never happens while v-kernel's system call executes. This ensures that the v-kernel accesses its control data block exclusively.

(2) Mutual exclusion in other kernels: While the v-kernel is executing in a critical section, the integrity and consistency of the v-kernel's data structure won't be guaranteed if a dispatch from v-kernel to p-kernel is allowed. Therefore task dispatch to p-kernel's task must be delayed while the v-kernel is executing in a critical section. To delay dispatch to p-kernel's task, we use DI(13) handler for the dispatcher from the v-kernel to the p-kernel and interrupt mask level is set to 13 while the v-kernel is executing in a critical section.

An example of this case is shown in Fig.4.5. We assume that an interrupt occurs when the v-kernel is executed at interrupt level 13 (IMASK=13) in its critical section. An iwup_tsk, one of the system calls in the μTRON specification which is used in task-independent portion to wake up a task, is issued in the interrupt handler so that DI register is set to 13, it means a request for dispatch is set. When the v-kernel exits from its critical section, IMASK value goes back to 14 and which triggers DI(13) handler. Then task dispatch from the v-kernel to the p-kernel is done by DI(13) handler. In this way, task dispatch is delayed.

(3) Mutual exclusion in p-kernel: When a critical section in the p-kernel is executed, mutual exclusion is realized by inhibiting interrupt with IMASK=0.

On the other hand, when a task in the p-kernel issues a v-kernel's system call, a dispatch from the issuing task to another p-kernel's task must be delayed. In other words, if a dispatch to another p-kernel's task is requested inside an interrupt handler which is invoked in executing the v-kernel's system call, the dispatch must be delayed until the v-kernel's system call is completed. The reason why this delayed dispatch is needed in the inter-kernel operation is that a v-kernel's system call invoked by a p-kernel's task is executed in task-independent portion without v-kernel's context. In order to achieve this delayed dispatch, IMASK is set to 13 before the p-kernel enters the v-kernel (see Fig.4.6). DI mechanism makes it possible to realize the delayed dispatch automatically.

(4) Dispatch from v-kernel to p-kernel: Dispatch from the v-kernel to the p-kernel happens when
a p-kernel's task is waked up by a v-kernel's task or by an interrupt handler. When a dispatch becomes needed, DI register is set to 13 to activate DI(13) handler and a dispatch is executed.

(5) Dispatch from p-kernel to v-kernel: Dispatch from the p-kernel to the v-kernel happens only when there is no READY or RUN tasks in the p-kernel. Since the v-kernel is applied to the idle task of the p-kernel, this dispatch is done by the p-kernel's dispatcher automatically. But there is possibility that higher priority task in the v-kernel has entered the READY state by a v-kernel operation issued by a p-kernel's task. In that case, it is necessary to execute the dispatcher of the v-kernel. DCT handler is then activated in dispatch to the idle task of the p-kernel, using CSW, and it activates dispatcher of the v-kernel if necessary (see Fig.4.7).

4.6 Evaluation

We developed prototype of the unified OS and evaluated the performance of it. We evaluated the following system calls:
(a) wup_tsk
(b) set_flg
(c) req_io

Req_io is one of the system calls which requests I/O processing to a device driver. We measured their execution time and compared them with those of the MR3200's and MR3210's.

In case of (c)req_io, we evaluated its execution time assuming a model of device driver as shown in Fig.4.8. We defined the time:Δ in Fig.4.8 as the execution time of req_io; Δ is the elapsed time from when the task issues req_io system call to when the issuing task is waked up by setting a flag. The model is a form of device driver we recommend. So we think that this model is adequate to evaluate I/O processing relatively.

Fig.4.9 and Fig.4.10 show the execution time of (a)wup_tsk and (b)set_flg respectively. To evaluated overheads of this OS, we used the data of the MR3200's and the MR3210's are shown in the same figure. In case of inter-kernel operations, the execution time from the v-kernel to the p-kernel includes task switching time. And the execution time from the p-kernel to the v-kernel doesn't include it.

Fig.4.11 shows the result of (c)req_io. The execution time of the unified OS means that a task in the v-kernel requests an I/O processing to a device driver in the p-kernel; we assure that the task which needs versatile functions in the v-kernel depends on the p-kernel for only I/O processing. The execution time of the MR3200's or the MR3210's in Fig.4.11 are the I/O processing time.
4.7 Discussion about the performance

We will discuss the performance of the unified OS in points of system call execution time (wup_tsk, set_flg), I/O processing time (req_io) and response to an interrupt.

(1) System call execution time: In case of intra-kernel operations, the overhead caused by unification is a little (only 1 μsec) in comparison with the MR3200 and the MR3210. This overhead is derived from examination of the kernel ID. On the other hand, the overheads of inter-kernel operations become bigger. In case from the p-kernel to the v-kernel, the overhead is about 7μsec and this is mainly caused by execution of TRAPA instruction and mutual exclusion. In case of the reverse direction, the overhead is about 20μsec as compared with the MR3200 and this is mainly caused by execution of the TRAPA instruction and the DI handler to dispatch. It seems that inter-kernel operation becomes longer. In fact, it shortens 30μsec as compared with task switching time of the MR3210. This means task synchronization from the v-kernel to the p-kernel is faster than that of intra v-kernel. So the expected performance described in section 3.4 could be achieved.

(2) I/O processing time: The execution time of req_io in the unified OS is faster over 200μsec than that of the MR3210. This data demonstrates the result caused by accelerating system calls in device driver. The time saved by execute I/O handling in the p-kernel is more significant than affect by combing the p-kernel and the v-kernel. Besides, more effective result can be obtained if memory access time is increased because this data is measured with no-wait memory. Therefore total performance can be improved using the unified OS.

(3) Response to an interrupt: As a response to an interrupt, we discuss the interrupt latency and the task switching latency[5] in the unified OS. The interrupt latency is the time during which an OS disables executing interrupt handler, and the task switching latency is the time during which an OS can not dispatch a new task even if an interrupt occurs and a higher priority task is readied by a system call issued in the interrupt handler. The interrupt latency and the task switching latency matter when an external interrupt occurs while system call is being executed. Of course, in a real-time OS, the latency should be made as short as possible.

Table 4.3 shows the latency in the unified OS. The interrupt latency is about 15μsec and it matters only when a pkernel system call is executed. This short interrupt latency is obtained by executing interrupt handlers under the p-kernel. Next, the task switching latency for executing a p-kernel system call is equal to the MR3200's system call execution time, maximum 26μsec. When a v-kernel's task issues a v-kernel's system call, the task switching latency is about 30 μsec; this latency is caused by delayed dispatch explained in 4.5(2) (see Fig.4.5). Only when a p-kernel's task issues a v-kernel's system call, the task switching latency is equal to the MR3210's system call execution time in the worst case; this latency is caused by delayed dispatch explained in 4.5(3) (see Fig.4.6). In this case, we can limit this latency by restricting system calls. As a matter of fact, only synchronization and communication functions are
system call in executing

<table>
<thead>
<tr>
<th>Table 4.3</th>
<th>Response to an interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>system call in executing</td>
<td>Interrupt latency</td>
</tr>
<tr>
<td>Intra p-kernel</td>
<td>15μsec</td>
</tr>
<tr>
<td>Inter-kernel (from p- to v-kernel)</td>
<td>0</td>
</tr>
<tr>
<td>Inter-kernel (from v- to p-kernel)</td>
<td>15μsec</td>
</tr>
<tr>
<td>Intra v-kernel</td>
<td>0</td>
</tr>
</tbody>
</table>

necessary for system calls from p-kernel's tasks to the v-kernel. Then we estimate the task switching latency is maximum 50μsec.

So we realized high performance for response to an interrupt, too.

5 Conclusion

We discussed problems of existing real-time OS, such as influence of mutual exclusion, real-time performance in multi-processor system and overhead by homogeneous handling of objects. As a result, we concluded that it is necessary to classify objects and prepare more than one ready queue. Then we proposed a concept of the unified OS which provides both versatile functionality and quick response. This OS also is easy to adapt to multi-processor system and is compatible with a corresponding multi-processor system. Then we implemented it on a single processor to examine its efficiency. We obtained the expected high performance and confirmed that this OS improves the performance and functionality of a whole system.

References