Message-based Microkernel for Real-time System*

Seong Rak Rim, Yoo Kun Cho

Department of Computer Engineering
Seoul National University
Seoul, 151-742 Korea

Abstract

This paper describes a design and implementation of the basic primitives and major components of the message-based microkernel for real-time systems to find out it's shortcomings and ways to improve them. Through our experience, the real-time OS with message-based microkernel enables a user to add or change the system services easily for special purposes. But it has rather large overhead of interrupt latency and system call due to the message copy and synchronization. In order to support true real-time performance, kernel preemption and efficient message exchange mechanism is required.

1. Introduction

A system that has to respond to external, asynchronous events within a predictable (or deterministic) amount of time is called a real-time system [1,2]. Many real-time systems have been developed and are being used for various purposes. Traditionally, development of these systems depends on the unique characteristics of the proprietary *Operating System (OS)*, hardware architecture and instruction set. With the escalating cost of software development and massive conversion efforts required for porting to state-of-the-art hardware, it became important to provide the real-time systems with flexibility and extensibility [3,4,5].

A recent trend in *OS* development is the restructuring of the traditional monolithic *OS* kernel into independent servers running on top of a nucleus or microkernel[6]. In this *OS* architecture, the microkernel provides system servers with generic services independent of a particular *OS*, such as processor dispatching and events (interrupt, trap, exception) handling. It also provides a simple *Inter-Process Communication (IPC)* facility that allows system servers to call each other and exchange data. In this paper, we give an overview of experimental message-based microkernel for the real-time systems. We then describe a design and implementation of the basic primitives and major components of the message-based microkernel. The objective of this paper is to find out the shortcomings of message-based microkernel for real-time systems and ways to improve them. The experimental message-based microkernel is evaluated in terms of interrupt latency and system call by comparing with the monolithic kernel scheme.

The idea of *OS* with microkernel is not new. Such as Chorus[6,7], Mach[8] have realized it completely for the general-purpose *OS*. Basically they are a communication message passage-style *OS* kernel. What makes the difference in this microkernel is the level of simplicity, since it is designed only for real-time systems. It means the development, debugging and extension become easier.

*The research was supported by the Korea Science & Engineering Foundation under the object-oriented basic research project 89-01-01-03.

0-8186-2755-7/92 $03.00 © 1992 IEEE

2. Model of Real-time Operating System

One of the primary tasks of an *OS* is the efficient management of the shared resources including *CPU* among a set of competing user programs. It can be viewed as an allocator of resources or an arbitrator between the demands of application programs which can be subdivided into tasks and/or processes. It also manages the interface between different tasks or processes, and between the processes and *I/O* subsystems. Conventional *OS* consists of a number of software entities performing process management, memory management, file management, *I/O* device management and many others, depending on the particular properties of the computing systems on which the *OS* is implemented. We assume that above tasks could be organized as a set of independent system servers such as Chorus's rationale [7].

Our model of real-time *OS* is composed of a microkernel and a number of system servers. System servers may be executed in kernel or user mode as an application. They need to communicate each other via microkernel to provide a coherent set of system services. Microkernel provides a communication mechanism and arbitrates the service requests from application or hardware. This idea of organizing the tasks of *OS* as a set of system servers is to increase modularity, portability and scalability of the overall system. *Figure 1* shows an overall organization of *OS*. It can be applied to a centralized as well as a distributed computing environment.

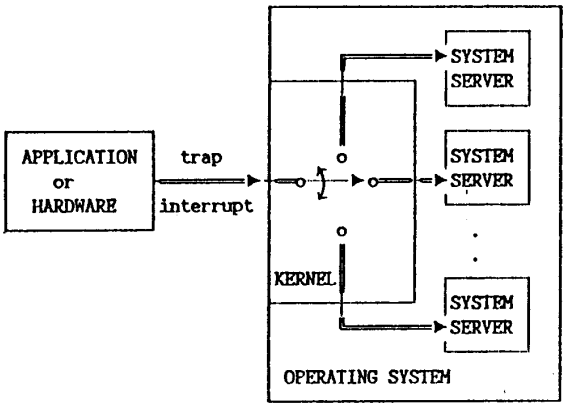

Figure 1. Overall Organization

3. The Microkernel

The microkernel is a single object running in kernel mode and provides the system server with the essential primitives. Its main function is to recognize the system service requests from application or hardware and invocation of appropriate system server according to the request types. It is composed of a set of procedures. *Figure 2* shows the major components and their relationship. A more detailed description of components can be found in the reference [9].

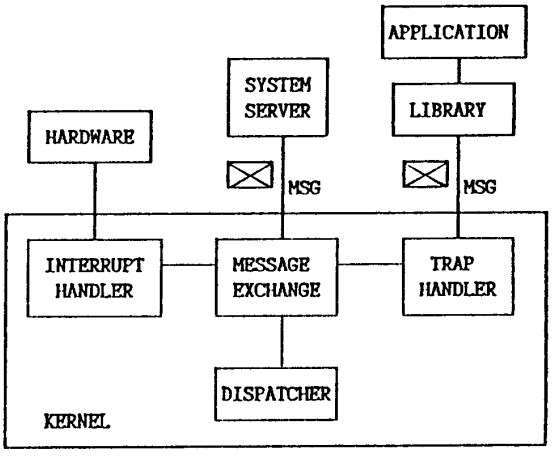

Figure 2. Major kernel components

1. *Message Exchange* : Message exchange is responsible for sending and receiving the message. It is called by interrupt or trap handler to send or receive a message.

2. *Dispatcher* : The dispatcher is called by message exchange to manipulate a task states transition for synchronization and context switching. The next executable task is determined upon the real-time scheduling policy.

3. *Interrupt Handler* : It makes up a message according to the hardware interrupt types and calls message exchange to send the message to the appropriate system sever.

4. *Trap Handler* : Application calls a system service through the library. The library makes up a message according to the system call types and causes a trap. Trap handler copies a message from user space to kernel and calls the message exchange.

3.1 Inter-Task Communication

Applications (including system server) communicate by exchanging fixed-size messages using the rendezvous principle. Basic primitives for massage exchange are *SEND* and *RECEIVE* which are called by interrupt or trap handler. When a hardware interrupt occurs, interrupt handler makes up a message only for *SEND* and calls message exchange. When an application does a system call, library makes up a message and causes a trap. In this case, two types of messages are needed for *SEND* and *RECEIVE* respectively. Trap handler changes the *CPU* mode from user to kernel, then calls a message exchange to send or receive the message to/from the system server. *SEND* checks to see if the destination is waiting for the message from sender. If so, the message is copied from sender's space to receiver's space immediately and dispatcher is called to make a context switching if needed. Otherwise, the message is copied from sender's space to kernel space temporary and linked to the receiver's *Task Control Block (TCB)*. *RECEIVE* checks the receiver's *TCB* to see if a message to be received is linked. If so, message is copied from kernel space to receiver's space. Otherwise, the receiver blocks until a message arrives.

3.2 Real-time Scheduling

The basic idea of task scheduling is straightforward: each task is assigned a priority, and a ready task with highest priority is allowed to run. Priorities can be assigned to tasks statically when the task is created or inherited

```
/* SEND PROCEDURE */
msg_snd(caller, dest, msg_ptr)
int     caller,          /* sender task */
        dest;           /* destination task */
struct msg *msg_ptr;  /* message pointer */
{
 if(dest is waiting for this message) {
  cp_msg(); /* message is copied */
  ready(dest); /* dest will be enrolled */
 }else{
  get a message frame from kernel space;
  cp_msg(); /* message is copied */
  link the messge frame to dest's TCB;
 }
}

/* RECEIVE PROCEDURE */
msg_rcv(caller, src, msg_ptr)
int     caller,          /* receiver task */
        src;            /* sourcer task */
struct msg *msg_ptr;  /* message pointer */
{
 for(message frames linked to caller's TCB)
  if(message from source task) {
   cp_pkt(); /* message is copied */
   remove message frame from caller's TCB;
   return;
   }
 block(caller); /* caller will be removed */
 }
```

Figure 3. *SEND* and *RECEIVE* Algorithm

temporary from the caller when the caller's priority is higher. This priority inheritance scheme[5] enables the higher priority task to complete service quickly. Because, the lower priority tasks called by the higher priority task can be selected directly without scheduling overhead.

Microkernel maintains a single run queue to manipulate a task state transition for synchronization and context switching. We have simplified the task states as follows:

- o *RUNNING* : currently using the *CPU*
- o *READY* : ready to run but *CPU* is not available
- o *BLOCKED* : waiting for a message

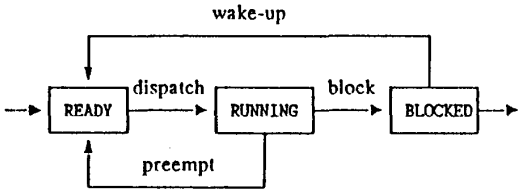

Figure 4. Task state transition

As shown in *Figure 4*, four transitions are possible among the three states. When a task is created, its priority is assigned. Based on the priority, it is enrolled in the run queue with *READY* state. It will transit to *RUNNING* state when selected to run. After a task completes a service, or discovers that it cannot continue (such as *I/O* request), it will be blocked waiting for a message and removed from the run queue. When the message for which a task was waiting arrives, the task is woken up and enrolled in the run queue with *READY* state. If the message comes from the higher priority task, the woken-up task starts running immediately. Otherwise it may have to wait in *READY* state for a while until the *CPU* is available. In order to support real-time scheduling, microkernel provides the primitives as shown in *Figure 5*.

4. Performance Evaluation

Even though the message-based microkernel can provide real-time systems with flexibility, modularity, and ease of extention, the performance degradation was expected due to the overhead of message exchanging and synchronization. To evaluate this overhead, we have measured the time of interrupt latency and the system call.

4.1. Interrupt Latency Time

One of the most critical requirements for real-time performance is the interrupt latency time[10]. In this *OS* architecture, most of the tasks are waiting for a message in order to be activated. When the interrupt occurs, the current executing lower priority task must

```
/* Enrollment in run queue */
ready(tid)
int     tid;    /* task id to be enrolled */
{
 set tid's state as 'READY';
 if(caller's priority > tid's)
  priority is inherited;
 put_ready(tid); /* tid is enrolled */
 if(tid's priority > current task's)
  dispatch(); /* context switching */
}

/* Remove from run queue */
block(tid)
int     tid;    /* task id to be removed */
{
 set tid's state as 'BLOCKED';
 put_block(tid); /* tid is removed */
 select the highest priority task;
 dispatch();
}
```

Figure 5. *READY* and *BLOCK* Algoritm

quickly be switched out and the appropriate task switched back in. Interrupt latency time can be defined as the time interval between the time that the *CPU* recognizes the interrupt and the beginning of execution of the appropriate task. It is composed of following components.

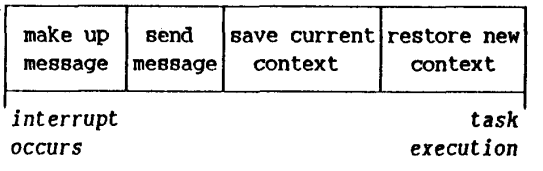

Figure 6. Components of interrupt latency

To measure the interrupt latency time, we have calculated the execution time of each component by counting the number of clocks per assembly instruction based on the 33Mhz i80386 *CPU* [11]. It is summarized in *Table 1* and compared with the scheme of traditional monolithic kernel which calls the interrupt service routine directly via interrupt vector table.

Table 1 Comparison of interrupt latency
(unit: μsec)

	component	# of clock	time
traditional monolithic kernel	save	150	4.54
	call	7	0.21
	interrupt latency (=x)		4.75
message-based microkernel	make msg	73	2.21
	send msg	698	21.15
	save	140	4.24
	restore	117	3.55
	interrupt latency (=y)		31.15
ratio (y/x)	6.56		

Table 1 shows the message-based microkernel takes about 7 times of interrupt latency overhead than the traditional monolithic kernel. This overhead is not acceptable for the real-time environment. Hence, we have decided on implementing the hasty interrupt service[12] routines within the microkernel to be called directly via interrupt vector table. And the rest of services are executed by the system server. But this easy implementation may make responses of the microkernel slow because the microkernel will not allow any other interrupts while executing an interrupt. So, the problem is how we can extract the hasty interrupt services and shorten the length of them.

4.2 System Call

To measure execution time of the system calls directly, we have designed a prototype of message-based microkernel for the real-time system complying with the *RTEID (Real-Time Executive Interface Definition)* [13]. Currently, it is running on the i80386 personal computer and basic system servers *(such as TIME, TASK, and MEMORY MANAGER)* are implemented only for the performance evaluation. Also, we have implemented a monolithic kernel which provides the same services as a set of procedure, and compared

Table 2. Comparison of system call
(unit: μsec)

	null	t_create	t_create ,t_start
microkernel(=x)	231.23	308.59	645.30
monolithic(=y)	50.27	63.82	159.45
ratio (x/y)	4.60	4.83	4.05

it with the massage-based microkernel *(Table 2)*.
In terms of strict comparison, *table 2* shows that message-based microkernel is slower 4-5 times than the monolithic. The main reason for this is considered the overhead of intercontext copy.

In order to reduce this overhead, we have modified the primitives of message exchange to use the shared memory. It makes up a message in the shared memory and just exchanges its address without copying the contents of message. With these primitives, we can get a good deal of performance enhancement *(Table 3)*. This has proven that the weakness in message-based microkernel architecture is the overhead of message copy between tasks.

Table 3. Comparison of system call
(unit: μsec)

	null	t_create	t_create ,t_start
microkernel(=x)	70.95	112.83	268.42
monolithic(=y)	50.27	63.82	159.45
ratio (x/y)	1.51	1.78	1.68

As a result of the performance evaluation, kernel preemption and efficient intercontext copy mechanism is required to support true real-time performance.

5. Conclusion

Next generation real-time sustems will require greater flexibility and predictability than the commonly found features such as a fast context switch in real-time systems. To meet this reqirement, this paper describes a design and implementation of a message-based microkernel for the real-time systems. In our experience, message-based microkernel architecture is simple yet powerful to provide the real-time systems with flexibility and extensibility by implementing a system server individually in the user space. But, kernel preemption and efficient message exchange mechanism is required to support true real-time performance.

By virtue of its simplicity, message-based microkernel can be used as a foundation to develop various real-time *OS*. They may be completely different from each other depending on the charateristic of system servers. Especially, it can be applied to the distributed computing environment with the distributed shared memory.

References

[1] B. Furht, D. Grostick, D. Gluch, G. Rabbat, J. Parker, M. McRoberts, REAL-TIME UNIX SYSTEMS Design and Application Guide. Modular Computer Systems, Inc., 1991

[2] A. Damm, et. al., "The Real-Time Operating System of MARS," Operating Systems Review, Vol.23, No.3, July 1989, pp.141-157

[3] J. Stankovic, "Real-Time Computing System: The Next Generation," Tutorial: Hard Real-Time Systems, IEEE Computer Society Press, 1988, pp.14-37

[4] J. Stankovic, K. Ramamrithm, "The Design of the Spring Kernel," IEEE Computer Society Press, 1988, pp.371-382

[5] H. Tokuda, C.W. Mercer, "ARTS: A Distributed Real-Time Kernel," Operating Systems Review, Vol.23 No.3, July 1989, pp.29-53

[6] M. Guillemont, J. Lipkis, D. Orr, M. Rozier, "A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility," Proc. of USENIX, Winter '91, pp.13-22

[7] M. Rozier, et. al., "CHORUS Distributed Operating Systems," Computing Systems, Vol.1, No.4, Fall 1988, pp.305-370

[8] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, "MACH: A New Kernel Foundation for UNIX Development," Proc. of USENIX, Summer '86, pp.93-112

[9] S. Rim, Y. Cho, "Basic Elements for a Microkernel-based Operating System," Journal of the Korea Information Science Society, Vol.18, No.6, November 1991.

[10] B. Furht, et. al., "Performance of Real/IX -Fully Preemptive Real-Time UNIX," Operating Systems Review, Vol.23 No.4, October 1989, pp.45-52

[11] i80386 Microprocessor Programmer's Reference.

[12] K. Fukuoka, A. Yokozawa, K. Tamaru, "Hierarchical Design of a μITRON Specification Kernel: TR2," Proc. of The Eighth TRON Project Symposium, November 1991, pp.69-76

[13] Real-Time Interface Definition Specification, Motorola Inc., 1988