Extensible Distributed Operating System for Reliable Control Systems

Katsumi Maruyama, Kazuya Kodama, Soichiro Hidaka, Hiromichi Hashizume
National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
Email:{maruyama,kazuya,hidaka,has}@nii.ac.jp

Abstract

Since most control systems software is hardware-related, real-time-oriented and complex, adaptable OSs which help program productivity and maintainability improvement are in strong demand.

We are developing an adaptable and extensible OS based on micro-kernel and multi-server scheme: each server runs in a protected mode interacting only via messages, and could be added/extended/deleted easily. Since this OS is highly modularized, inter-process messaging overhead is a concern. Our implementation proved good efficiency and maintainability.

1. Introduction

Most systems, from large scale public telephone switching systems to home electronics, are controlled by software, and improvement of control program development productivity is necessary .

To facilitate the program development , suitable OSs are required . The followings are required to OSs for control systems.

- Required features largely depend on target fields. Extensible and adaptable OS is required .
- Control systems, such as telephone switching systems, require very severe multi and realtime processing capabilities. Very efficient multi-threading must be supported.
- Hardware controls are essential.

  In most OSs, driver programs run in kernel mode, and their program development is very difficult.
- Control programs should be maintained for a very long time, adding new functions. Maintainability is of great importance .

General purpose OSs are not sufficient for these requirements. Therefore, in large and severe systems, specially designed OSs are used. In economical embedded systems, small monitor-like OSs are used. However, these monitor-like OSs lack program protection mechanisms, and program development is difficult.

Therefore, an extensible/adaptable OS for control systems is required . We are developing a new OS characterized by:

- Use of an efficient and flexible micro-kernel (L4-ka).
- Multi-server based modular OS. (Each OS service is implemented as individual user-level process.)
- Robustness. Only the micro-kernel runs in kernel mode and in kernel space. Other modules run in a protected user space and mode.
- Hardware driver programs in user-level process.
- Flexible distributed processing by global message passing.

This OS structure proved to enhance OS modularity and ease of programming. However, inter-process messaging overhead should be considered . We measured the overhead, and the overhead was proved to be small enough.

2. Structure of this OS

2.1. The outline of this OS

Fig. 1 shows the structure of this OS. Thick square boxes represent independent logical spaces.

Micro kernel manages logical spaces (processes), multi threads, message passing (IPC) and interrupts. Only the micro kernel is executed in the kernel mode.

OS services, such as process management, file service, network service, are implemented by user-level processes, not by the kernel. They have their own logical space and are executed in user mode. Even hardware drivers are located in user-level processes. Processes interact only via messages.

2.2. L4 Micro kernel

We adopted L4-ka micro kernel [2] implemented at University of Karlsruhe, because it is very efficient and flexible. L4-ka is characterized by:

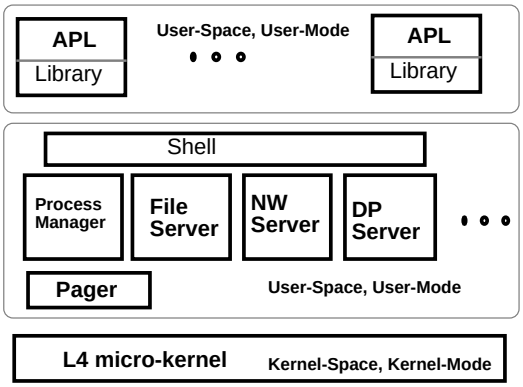

Figure 1. Multi server OS.

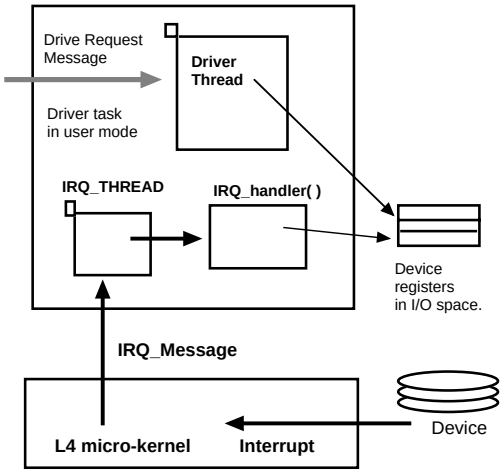

Figure 2. Driver program.

1. Efficient thread facilities.
2. Efficient and flexible message passing facilities: messages are sent synchronously to the destined threads by either value copy, buffer copy or page mapping.
3. Flexible memory management.

2.3. Logical space and pager

Pager is a user-level process to assign page frames when pagefault occurs . By rewriting the pager, a user can provide his own mapping algorithm. When a pagefault occurs , the kernel sends pagefault message to the pager. Pager assigns appropriate page frame, and reply message lets the kernel map new pages.

2.4. Service servers

Process manager manages allocation/freeing of logical space and processes. **File server** and **Network server** are implemented by rewriting those of Minix-OS [5] .

2.5. Driver program

Fig. 2 shows the driver program structure. Each device driver has a **driver thread** which waits for request messages, and an **IRQ thread** which waits for interrupt messages. Drivers are executed in user mode. This helps facilitating their development. When the L4 kernel notices hardware interrupt, it sends an interrupt message to the driver, and driver action is activated.

In our implementation, HDD driver and ETH driver are included in File server and INET server, respectively. They can be separate processes, because drivers interact only by messages.

3. IPC overhead and file server

In this OS structure, IPC efficiency determines the system efficiency. IPC overhead lies in system calls and block data transfers.

Prof. Liedtke et al. reported [1] a simple L4 message (inter address space, round trip) costs about 500 machine cycles, whereas the shortest Linux system call `getpid()` costs about 220 machine cycles, and that overall overhead will be several percent in comparison with monolithic OSs.

To evaluate the block data transfer overhead, we measured it in the case of file service. In monolithic OSs, file subsystem is included in the OS kernel; it accesses APL space directly and transfers data from/to APL by an efficient string copy operation.

In our OS, the **file server** is a user-level process, and it cannot access APL space directly. Data must be transferred using IPC, which may result in performance degradation. We measured the overhead and proved it small enough.

3.1. Buffer cache in file server

File server adopts **buffer caches** to improve file access speed (Fig. 3). Data blocks in HDD are identified by device number and block number. Recently accessed data blocks are cached in buffer caches, size of which is 1KB each. Buffer caches are located using hash table whose keys are device and block numbers. Data is transferred between an APL buffer and one or more buffer cache entries in FS.

3.2. IPC in L4 micro kernel

To transfer large volume data between processes, L4 provides an efficient buffer copy mechanism using temporal page mapping, and the following schemes can be used.

1. **Straight transfer**

   Fig. 4 shows the program to transfer data block “a1” of process-A to buffer “b1” of process-B. The sender

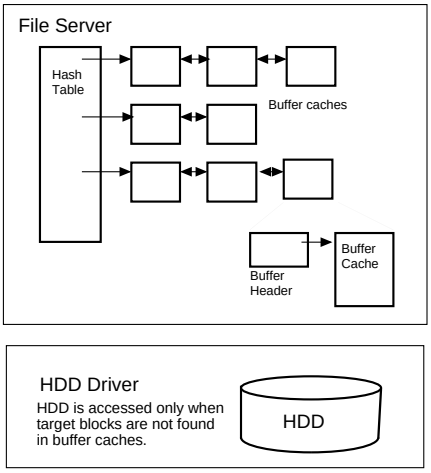

Figure 3. UNIX file subprogram.

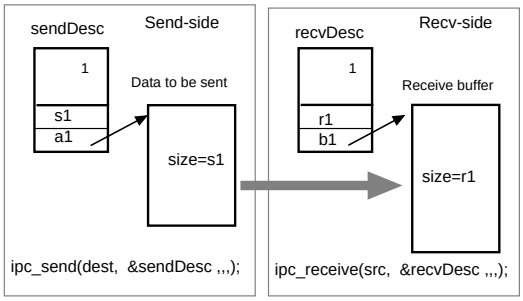

Figure 4. Straight buffer transfer.

prepares the send descriptor "sendDesc" and invokes `ipc_send()`. The receiver prepares the receive descriptor "recvDesc" and invokes `ipc_receive()`.

2. **Scatter/gather transfer**

   Sender and receiver can specify multiple buffers to send/receive data. Data is transferred in one IPC, automatically scattering and gathering data according to the send/recv descriptors. In Fig. 5, data blocks "a1", "a2" and "a3" of process-A are transfered to the buffer "b1" of process-B.

3.3. File server and data transfer

Let's assume that APL is requesting to read block data of 2.5KB. As each buffer cache is 1KB in size, requested data is cached in 3 buffer caches. Therefore 3 IPCs are required in a straight transfer.

In the scatter/gather transfer, data in multiple buffer caches can be transferred in one IPC. Fig. 6 shows this scheme.

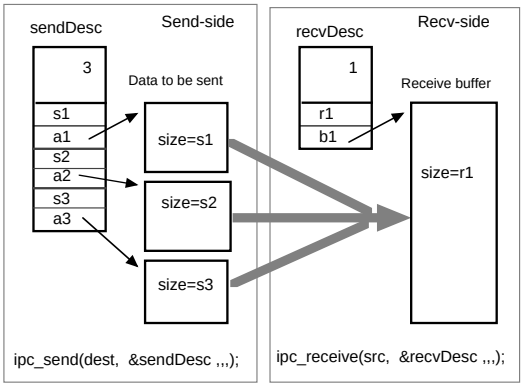

Figure 5. Scatter/gather buffer transfer.

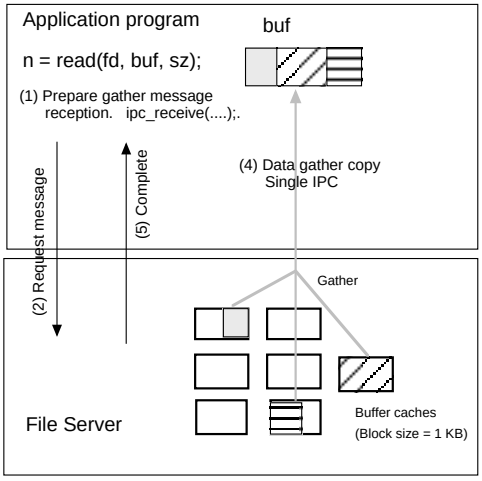

Figure 6. File read using scatter/gather transfer.

3.4. Evaluation

To evaluate the file-read IPC overhead, the following test program is used:

```
for (i = 0; i < 100000; i++) {
    lseek(fd, 0, 0);
    read(fd, buf, size);
}
```

This measures the time to transfer data from file server buffer caches to APL space (CPU=800MHz Pentium-3, L2 cache size= 512KB). At the first access, all data are copied on buffer caches from HDD, so that HDD access time are amortized.

Fig. 7 shows the results. The same program is tested on Minix-OS (Version 2.0) and Linux (Kernel version 2.2.12), and results are shown as Minix(3) and Linux(5), respectively.

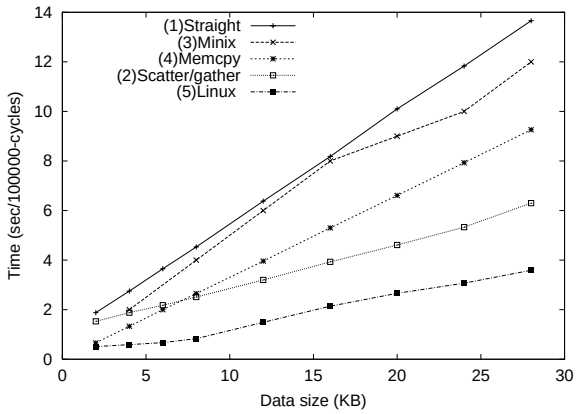

Figure 7. Inter-process communication overhead.

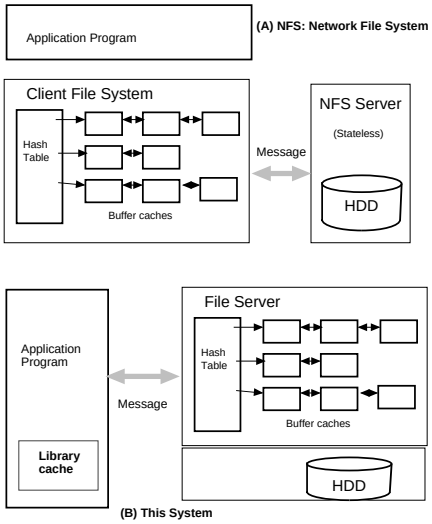

Figure 8. Distributed File Server.

In Minix, file server delegates copy to system task who has direct access to resident physical address space of APLs and servers, so no IPC is used for data transfer per se. On the other hand, Linux is a highly optimized monolithic OS.

Memcpy(4) shows the string copy using ANSI `memcpy` function (in the same logical space) for reference.

This test shows that:

- Straight transfer and Minix are comparable in transfer speed.
- Scatter/gather transfer improves speed significantly.
- Inter-process scatter/gather transfer is faster than intra-process `memcpy()`. This is because the former copies data word by word using temporary page mapping, and the latter copies data byte by byte.
- Linux is highly optimized and very fast.

Therefore, block data transfer overhead in concern was small enough in the scatter/gather transfer scheme.

4. Distributed processing

In this OS, all APLs and OS servers interact only via messages. We are now designing a distributed processing server, which delivers messages globally according to global thread IDs. This server would enable file server to reside on remote hosts without redesigning it. Fig. 8 compares this OS and NFS architecture.

In control systems, remote resource control is important. In this way, remote resources are easily controlled.

5. Conclusion

We are implementing this OS on IBM-PC. Through this OS implementation, the followings are confirmed:

Modularity and Extensibility New functions, which require kernel modification in monolithic OSs, can be easily extended only by adding user-level process in independent address space.

Robustness Most functions are implemented by protected user-level processes.

Ease of program development OS services, even a hardware driver, can be implemented by user-level process. Compared to kernel programs, user-level programs are easy to develop.

Hardware control Hardware control drivers are also user-level, and their development is facilitated.

Distributed processing We are now trying to extend the message passing to global scope. With global messaging, resources located in remote hosts can be accessed as local resources. This distributed processing is useful in control systems.

Acknowledgments We would like to thank Dr. Akira Nakamura at the International Christian University, Dr. Yusheng Ji, Dr. Ichiro Ide for their valuable discussions and suggestions.

References

[1] Jochen Liedtke. Improving IPC by kernel design. In *Proc. of SOSP'93*

[2] Jochen Liedtke. On μ-Kernel Construction. *Operating Systems Review*, Vol. 29, No. 5, pp. 237–250, December 1995.

[3] IBM Watson Research Center. SawMill home page http://www.research.ibm.com/sawmill/

[4] Hermann Härtig, et al. The performance of μ-Kernel-based systems. In *SOSP97*.

[5] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice-Hall