OPERATING SYSTEM SUPPORT FOR EASY DEVELOPMENT OF DISTRIBUTED FILE SYSTEMS

KENICHI KOURAI† SHIGERU CHIBA‡ TAKASHI MASUDA†

† Department of Information Science
University of Tokyo
7–3–1 Hongo, Bunkyo-Ku, Tokyo 113–0033, JAPAN

‡ Institute of Information Science and Electronics
University of Tsukuba
1–1–1 Tennodai, Tsukuba, Ibaraki 305–8573, JAPAN

ABSTRACT

A number of new distributed file systems have been developed, but the development of such file systems is not a simple task because it requires the operating system kernel to be modified. We have therefore developed the CAPELA operating system, which makes distributed file systems easy to develop. CAPELA allows the users to develop a file system as an extension module separated from the kernel, and protects the kernel from erroneous extension modules by a new fail-safe mechanism, called multi-level protection. The multi-level protection can avoid unnecessary performance degradation by enabling the protection level to be changed without modifying the source code of the module. We have implemented the CAPELA operating system on the basis of NetBSD 1.2 and confirmed that the file system can run more efficiently when the protection level is lowered.

keywords: multi-level protection, fail-safe mechanism, distributed file system, operating system

1 INTRODUCTION

A number of distributed file systems such as NFS [5], AFS [6], and Coda [7] have already been developed, and most of them are embedded in the monolithic kernel like UNIX systems. This makes it difficult to develop distributed file systems because it is hard to debug the file systems in the kernel. The developers cannot use tools like a symbolic debugger that helps identify the cause of errors. To make matters worse, they must reboot the computer every time an error occurs because the operating system kernel crashes.

Many distributed file systems are therefore first implemented as a user-level library that emulates the system calls for file access and are later re-implemented in the kernel. The emulation makes it easy for the developers to implement and debug the file system because they do not need to modify the kernel. After the prototype of the file system developed using the emulation is implemented, they evaluate the file system experimentally. Considering the results of this evaluation, they can easily change the policy of the distributed file system before re-implementing the file system in the kernel. This way of development, however, makes the burden on the developers heavy. Because of the differences in data structure, application programming interfaces (APIs), and timing, they must debug both the library for the emulation and the file system embedded in the kernel, and this debugging makes the development laborious.

To solve this problem, we have developed the CAPELA operating system, which makes distributed file systems easy to develop. CAPELA allows the users to develop a file system as an extension module separated from the kernel. Because the extension module is installed in the operating system on demand, the users can debug their file systems without continually rebooting the computer. CAPELA also uses a new fail-safe mechanism, called multi-level protection, that allows the users to install an extension module at various protection levels without modifying the source code. The users can therefore avoid performance penalties by running the released module at a lower protection level than that used for debugging the file system.

We have implemented the CAPELA operating system on the basis of NetBSD 1.2. The multi-level protection has been implemented by a set of protection managers, which provide a different level of protection. The users can easily change the protection level of the extension module simply by selecting one of these protection managers. Because the protection managers provide common APIs for the file system module, the users do not need to modify the source code of the module when changing the protection level.

We experimented to make sure of the usefulness of the multi-level protection and confirmed (1)that the performance of a file system module is improved when the protection level is lowered, (2)that the overheads at the maximal protection level are acceptable, and (3)that the overheads at the minimal protection level are almost negligible compared with those of the file system hand-crafted in the kernel.

2 CAPELA

We have developed the CAPELA operating system on the basis of NetBSD 1.2, which makes distributed file systems easy to develop. CAPELA allows the users to develop a file system as an extension module separated from the kernel so that it can be installed without rebooting the system. To protect the operating system kernel from erroneous modules, CAPELA provides a new fail-safe mechanism, called multi-level protection. The multi-level protection allows the users to change the protection level of the extension modules according to the stability of the file system under development.

In this section, we describe the multi-level protection and how it can make a new file system easy to develop. And we explain our implementation of the multi-level protection in CAPELA.

2.1 MULTI-LEVEL PROTECTION

The multi-level protection enables the users to change the protection level without modifying the source code of the extension modules. The users need only select the appropriate protection level, considering the trade-off between the cost of the protection and the performance of the extension module. If they select the highest protection level, all errors are detected and recovered; if they select lower one, some errors are neither detected nor recovered. To keep source-level compatibility of the extension modules between different protection levels, the multi-level protection provides common APIs for the extension modules.

This reduces the difficulty of debugging a new file system. The developers can, for example, implement a prototype of a distributed file system almost as easily as the library for the emulation. The file system is implemented as an extension module which is a user process and fully protected when the protection level is higher. The developers can therefore obtain accurate error information, easily identify the causes of the errors, and fix them. In addition, an extension module in which an error has been detected is safely detached by CAPELA so that it does not compromise the rest of the operating system.

After the prototype is implemented, the developers can decrease the protection level without modifying the source code. Because the performance is thereby improved, it becomes easier to run the module longer in order to find more errors. It is generally sufficient to detect only errors depending on timing, like deadlocks, because in this phase of testing the distributed file systems seem to be rather stable.

Finally, the released version of the distributed file system developed with the multi-level protection is embedded into the kernel without modifying the source code. It can run almost as efficiently as one handcrafted in the kernel without any protection.

2.2 SYSTEM OVERVIEW

In CAPELA, the multi-level protection has been implemented by a set of protection managers, which provide a different level of protection. A protection manager is a library linked to an extension module and it provides common APIs for the module to communicate with the kernel. These common APIs enable to execute an extension module without modifying the source code either in a user address space or in the kernel address space. Also, modifying the source code is not needed to customize the ability of other kinds of error detection and recovery. All the differences among protection levels are absorbed by the protection managers.

CAPELA allows the users to change the protection level of an extension module by exchanging protection managers. In the current implementation, the users relink the module with the protection manager which provides a desirable protection level. After relinking, they restart the module without rebooting the computer in order to make the new protection level available.

2.3 PROTECTION MANAGER

The roles of the protection manager are module management, upcall processing, and safe manipulation of the kernel data. For module management, the protection manager registers and unregisters an extension module with the kernel. For upcall processing, the protection manager invokes a callback function registered by an extension module when it receives an upcall from the kernel. For safe manipulation of the kernel data, the protection manager protects the kernel data from an erroneous module by using various protection techniques.

The protection manager provides an API for manipulating the kernel data like `vnode`. The extension modules can access the kernel data only through this API because the kernel data is protected by the protection manager. For example, `Vref` function increments the reference count of `vnode`. In addition, this API enables the modules to deal with a kernel data structure of low abstraction as if it were one of high abstraction. Consequently, complex operations of the kernel data and dangerous pointer manipulations are hidden from the modules.

The protection manager also provides an API for registering callback functions. The extension modules need to register callback functions with the protection manager in order to receive upcalls from the kernel, e.g. `VOP_READ` and `VOP_WRITE`. The protection manager first receives these upcalls from the kernel and thereafter invokes the corresponding callback function registered. At this time, the protection manager transforms the data structure passed as the arguments to more abstract structure.

2.4 PROTECTION TECHNIQUE

The protection managers combine various techniques shown below and implement different protection levels. For more details, see a different article [3].

2.4.1 ILLEGAL MEMORY ACCESS

To detect a hardware-trapped illegal memory access, the protection manager exploits switching address spaces. When an extension module is located in a user space, any illegal memory accesses to the kernel memory are easily detected by hardware. The module uses shared memory to exchange data with the kernel, but the use of shared memory enables the module to illegally access the kernel data. To prevent such illegal memory accesses, CAPELA protects the shared memory by using virtual memory system, e.g. `mmap` system call.

CAPELA allows the users to adjust the overheads of detecting illegal memory access by selecting the use of switching address spaces and the way of memory protection. The users can protect the shared memory for communication by simply changing its protection mode to read-only with `mprotect` system call. They can even protect no shared memory. To decrease the overheads more, they can also locate an extension module in the kernel space.

To detect semantically illegal data modification, the protection manager replicates the kernel data, makes the extension modules manipulate the replicas, and writes it back to the kernel periodically. When writing it back, the protection manager checks the data by examining various properties of the data structure; for example, that the reference count of data is positive or zero. CAPELA allows the users to adjust the overheads by selecting what types of data are checked and how the data is checked.

To recover from these illegal memory accesses, CAPELA rolls back modified kernel data to stable states. CAPELA uses a log in which manipulations modifying the kernel data are recorded. On recovering, CAPELA checks the log and executes the manipulations reverse to those recorded in the log. CAPELA allows the users to adjust the overheads of recording the log and restoring the modified data by selecting what manipulations are recorded and what data is restored.

2.4.2 DEADLOCK

To detect a deadlock, CAPELA periodically checks whether the system falls into a deadlock state. When locking, unlocking, and waiting for resources like `vnode`, the extension modules notify the system of that and CAPELA records this information. To check for a deadlock, CAPELA creates a wait-for-graph based on the information and checks for loops. CAPELA allows the users to adjust the overheads by selecting the interval between the checks.

protection technique	1	2	3	4	5
memory protection	√*	√**			
data replication	√	√	√		
address space switch	√	√	√	√	
logging	√	√	√	√	
deadlock check	√	√	√	√	

Table 1: FIVE PROTECTION LEVELS. *memory unmap **read-only mode

To recover from a deadlock, CAPELA destroys the loop of a wait-for-graph. CAPELA first finds out where the loop is and then temporarily releases one of the locks in the loop. The extension module whose lock is forcedly released is stopped until it can obtain the lock again.

3 EXPERIMENT

We experimented to make sure of the usefulness of the multi-level protection. The purposes of these experiments were (1)to make sure that the execution performance is improved when the users change the protection level and degrade the level of fail-safety, (2)to measure the overheads of the maximal protection level, and (3)to measure the overheads to enable an extension module to run in the kernel address space without modifying the source code.

We used a SPARCstation5 (MicroSPARC2/85MHz) for the SNFS client and a PC (Cyrix 6x86/133MHz) for the SNFS server. The client and server were connected with a 10Mbps network. Each operating system was CAPELA we developed.

We developed a file system module on CAPELA for these experiments: Simple Network File System (SNFS). We selected five kinds of protection managers for our experiments. Each protection manager uses the combinations of the protection techniques listed in Table 1. The first level is the highest protection level and the fifth level is the lowest protection level. For comparison, we also implemented SNFS hand-crafted in the kernel.

First, we measured the time needed to copy a file on SNFS using `read` and `write` system calls. The size of the copied file was 64KB. Next, we measured the time needed to compile a little program using `gcc`. The compiled program is a `ps` program of NetBSD 1.2, which consisted of five source files and two header files, and which had about 2,000 lines. These results are shown in Table 2.

The results of two experiments mean that the performance of SNFS is improved when the protection level is made lower. The first level takes 2.2 times

	1	2	3	4	5	hand-crafted
cp(ms)	1,272	1,154	795	705	516	515
gcc(s)	26.66	25.64	22.56	20.21	16.83	16.82

Table 2: THE TIME NEEDED TO EXECUTE A PROGRAM IN SNFS OF EACH PROTECTION LEVEL.

as much time as the fifth level, and the performance is rather degraded. It is acceptable, however, for debugging the SNFS module. In more practical circumstances like compilation of a program, the first level takes 1.6 times as much time as the fifth level, and the performance is good enough even for normal use.

Comparing SNFS of the fifth level with SNFS hand-crafted in the kernel, SNFS of the fifth level incurs the overhead of about 0.1%. This overhead is for enabling the SNFS module to run in the kernel space without modifying the source code. We think that the causes of this overhead are excess function calls, copies for transforming data structure, and so on. This result means that the performance of the SNFS module implemented using multi-level protection is almost as good as that of SNFS hand-crafted in the kernel if the protection level of the module is the lowest and the module is embedded in the kernel.

4 RELATED WORK AND CONCLUSION

The microkernel operating systems such as Mach [1] provide complete fail-safety. Since an extension module is implemented as a user process, the errors due to the module are not propagated to the rest of the operating system. However, the performance is sacrificed because the overheads of interprocess communication and context switches are large.

The extensible operating systems such as SPIN [2] and VINO [8] alleviate the difficulty of debugging distributed file systems by allowing the users to install an extension module into the kernel on demand. These systems provide a certain fixed level of fail-safety using type-safe language Modula-3 [4] and software fault isolation [9], respectively, but it is not appropriate in all phases of the development of file systems.

This paper has described the CAPELA operating system, which makes distributed file systems easy to develop. In CAPELA, a new file system is implemented as an extension module and the kernel is protected from errors of the module. Unnecessary performance degradation can be avoided by using a new fail-safe mechanism, called multi-level protection, to change the protection level. We have implemented the CAPELA operating system and confirmed the usefulness of the multi-level protection by some experiments.

The multi-level protection is so far implemented only for file systems, but distributed file systems often require the customization of network protocols. We will therefore apply the multi-level protection to network subsystems.

REFERENCES

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX 1986 Summer Conference*, pages 93–112, 1986.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, S. Chambers, and C. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings 15th ACM Symposium on Operating Systems Principles*, pages 267–284, 1995.

[3] K. Kourai, S. Chiba, and T. Masuda. Operating System Support for Easy Development of Distributed File Systems. Technical Report TR-98-01, Department of Information Science, University of Tokyo, 1998.

[4] G. Nelson. *System Programming with Modula-3*. Prentice Hall, 1991.

[5] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX 1985 Summer Conference*, pages 119–130, 1985.

[6] M. Satyanarayanan, John H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 35–50, 1985.

[7] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[8] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. An Introduction to the Architecture of the VINO Kernel. Technical Report TR–34–94, Harvard University Computer Science, 1994.

[9] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, 1993.