A FRAMEWORK FOR EASILY AND EFFICIENTLY EXTENDING OPERATING SYSTEMS

ＯＳの機能拡張を容易にかつ性能よく行なうための枠組

by

Kenichi Kourai

光来健一

Master's Thesis

修士論文

Submitted to

The Graduate School of the University of Tokyo

on February 9, 1999

in Partial Fulfillment of the Requirements

for the Degree of Master of Science in Information Science

Thesis Supervisor: Takashi Masuda　益田隆司

Title: Professor for Information Science

ABSTRACT

Extensible operating systems enable the users to extend their functions by adding extension modules on demand. Such operating systems need a fail-safe mechanism for protecting the systems from erroneous extension modules. However, the mechanism with the full capability of the protection has implied serious performance penalties. To address this problem, we propose a new fail-safe mechanism called *multi-level protection*. It allows the users to install an extension module in the operating system at various protection levels without changing the module, and thereby, the users can run the module at the minimum protection level to avoid performance penalties. For example, they can choose a higher level for an unstable module, but a lower one for a stable module. We have implemented the CAPELA system for the multi-level protection on the basis of NetBSD. CAPELA provides multiple protection managers of various protection levels so that the users can choose one of the protection managers and easily change the protection level. We constructed file system modules and network subsystem modules on top of CAPELA. Also, we confirmed that the performance of the extension modules is improved if the protection level is lowered. When the overheads of the maximum protection level are between 70% and 220%, compared with those of the minimum protection level. On the other hand, the overheads of the minimum protection level are between 1.3% and 12%, compared with those of the hand-crafted version.

論文要旨

拡張可能 OS は拡張モジュールを追加することで、その機能を動的に拡張することができる。このような OS では、拡張モジュールのエラーから OS を守るために fail-safe 機構が必要とされる。しかしながら、十分な fail-safe 機構を実現しようとすると、従来の実装技術ではシステムの性能低下が避けられなかった。そこで我々は、新しい fail-safe 機構である多段階保護機構を提案する。この機構は拡張モジュールを、変更なしに、様々な保護レベルで OS に組み込むことを可能にする。この機能を用いて、デバッグ時には保護レベルを高くし、デバッグが終われば保護レベルを低くするなど、必要最低限の保護レベルで拡張モジュールを動かすことにより、システムの性能低下を回避することができる。我々は多段階保護機構を実現するシステム CAPELA を NetBSD を基に実装した。CAPELA では保護マネージャが複数用意されており、それぞれが異なる保護レベルを提供している。ユーザはこの保護マネージャを交換することで、拡張モジュールの保護レベルを変更することができる。我々は CAPELA 上にファイルシステム・モジュールとネットワーク・モジュールを作成した。さらに、多段階保護機構の有効性を調べるための実験を行ない、fail-safe の機能を減らせば、その分実行性能を改善できることを確かめた。また保護レベルを最大にしても、オーバヘッドは 70%から 220%程度であった。さらに保護レベルを最小にすれば、オーバヘッドは 1.3%から 12%となり、カーネルに作り込んで手で最適化したものに近い性能が得られた。

Acknowledgements

I would like to thank my supervisor, Prof. Takashi Masuda, for his significant advice and generous encouragement. I am also thankful to Dr. Shigeru Chiba for his valuable suggestions and useful discussions. I appreciate Mr. Kenji Kono's good suggestion of my work. I would be grateful to all the members of the Masuda laboratory for many suggestions and their encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the purposes of applications, particularly middlewares, get advanced, many applications are not satisfied with traditional operating systems. The applications require rich functionality and good performance toward operating systems so that they can do what they need as efficiently as possible. However, it is unreasonable that operating systems possess all of the functions needed by all applications. This is because it is impossible to foresee all functions needed by all applications in advance, and because the operating systems with all functions become very enormous and make the maintenance hard even if foreseeing is possible. Also, since the performance of the functions is often improved by the developers as time goes on, operating systems should provide the facility with which users can replace the implementation of the functions easily, so that users can receive a benefit of the best performance.

Extensible operating systems enable users to add new functions to the operating systems or to replace old functions with new ones on demand. The new functions are implemented as a program called extension module, and they are installed into the operating system without rebooting the system. In the extensible operating systems, a fail-safe mechanism is mandatory because the system must be protected from erroneous extension modules. If the extensible operating systems have no fail-safe mechanisms, no users would install new extension modules except sufficiently stable modules, and, in addition, developers would be worried about debugging the extension modules. To prevent such erroneous extension modules from destroying the operating systems, a fail-safe mechanism should provide the ability to detect errors of the extension modules and the ability to recover from the errors. However, there is the trade-off between fail-safety and performance, and it is difficult to achieve both sufficient fail-safety and good performance. Many

microkernel operating systems like Mach [1] and many extensible operating systems like VINO [43] sacrifice the performance for the fail-safety more or less. On the other hand, an approach using loadable kernel modules gives up the fail-safety for the performance.

To avoid this dilemma, we propose a new fail-safe mechanism called *multi-level protection* [17, 18, 20, 21, 19]. It enables users to change the protection level of the extension modules without modifying the binary code. Using the multi-level protection, users can install the extension modules in the operating system at an appropriate protection level, depending on the stability of the modules. For example, if an extension module is unstable, users should select a higher protection level and then protect the operating system from errors of the extension module. But if an extension module is stable, users should select a lower protection level and then improve the performance.

We have implemented the CAPELA operating system on the basis of NetBSD 1.3.2. CAPELA is an extensible operating system with the multi-level protection and provides multiple protection managers, each of which provides a different protection level. Users can select one of the protection managers in order to change the protection level of the extension module to an appropriate one. In the current implementation, users need to exchange the protection managers to change the protection level. To provide different protection levels, the protection manager changes the ability to detect errors and the ability to recover from errors. Also, to modify no binary code of the extension modules when changing the protection level, all the protection managers provide the same application programming interface (API).

CAPELA supports the development of file systems and network subsystems in the current implementation. We experimented to make sure of the usefulness of the multi-level protection on file system modules and network subsystem modules. From our experiments, we confirmed that the performance of the extension modules is improved if the protection level is lowered. Also the overheads of the maximum protection level are between 70% and 220%, compared with the minimum protection level. On the other hand, the overheads of the minimum protection level are between 1.3% and 12%, compared with the hand-crafted kernel modules.

The rest of this thesis is organized as follows. Chapter 2 explains extensible operating systems and the needs of a fail-safe mechanism, which is the most significant facility of the extensible operating systems. Also we reviews previous approaches in terms of the fail-safety. Chapter 3 proposes a new fail-safe mechanism called multi-level protection and mentions the practical applicability. Chapter 4 describes the implementation of the CAPELA operating system. In this chapter, it is

described how the protection managers and the kernel achieve the multi-level protection. Chapter 5 measures the overheads of the multi-level protection on two file system modules and two network subsystem modules that we have developed on top of CAPELA. Finally, Chapter 6 concludes this thesis and suggests the future directions.

Chapter 2

Background

2.1 Needs of a Fail-Safe Mechanism

Extending operating systems is motivated by the needs of many applications. For example, developers for database applications want to change disk I/O buffering because a traditional general algorithm for buffering does not suit for disk access patterns of database [46]. In the field of multimedia, the issues of CPU scheduling, memory management, and network implementation in the current operating systems are pointed out [42]. A part of these issues would be solved by extending the operating systems on demand.

The extension of operating systems has two purposes: rich functionality and good performance. For rich functionality, it seems that it is sufficient for operating systems to provide several policies necessary for various applications [5]. However, it is unreasonable that operating systems possess all of the functions needed by all applications. This is because it is impossible to foresee all functions needed by all applications in advance, and because the operating systems with all functions become very enormous even if foreseeing is possible. Therefore, it is desirable that operating systems can be extended depending on the requirements of applications. For good performance, the functions of operating systems are often improved by the developers as time goes on. Since users can always receive a benefit of as good performance as possible, the operating systems should provide the facility with which users can replace the implementation of the functions easily.

In traditional operating systems like UNIX [36], it is difficult for users to extend the functions of the operating systems. To extend them, users must apply patches to the source code of the operating systems and recompile them, or apply patches to their binary code directly. Although

commercial operating systems also trend open source recently, the source code is generally expensive. There are some operating systems whose source code users can get without fee like Linux, but it is difficult to apply patches since the order applying patches and the combination of them are often troublesome. In any cases, users must shut down the whole system when they extend the operating systems.

To solve this problem, extensible operating systems have been developed. The extensible operating systems enable users to add new functions to the operating systems or to replace old functions with new ones on demand. The new functions are implemented as a program called extension module, and the operating systems do not need to be rebooted when they are extended. In the extensible operating systems, high extensibility, which indicates to what degree users can extend the operating systems, is important because it is more useful for applications to extend the operating systems more fine-grainly. However, the higher extensibility tends to be more difficult to extend the operating systems for programmers. Extremely speaking, if programmers modify the source code of the operating systems, they can do everything, but this operating system is not easy to extend since it is very hard for most programmers. The extensible operating systems should consider the trade-off between extensibility and ability to easily extend.

As the most significant facility of the extensible operating systems, a fail-safe mechanism is mandatory because the extensible operating systems must be protected from erroneous extension modules. Unlike the operating system kernel provided by operating system vendors, extension modules are also developed by third-party vendors. Therefore it is not realistic that the operating systems require for all extension modules to be error free. It is said that operating systems are easy to extend when they have robustness that the whole systems do not get unstable even if unstable extension modules are installed. Such robustness makes programmers easy to develop new extension modules and also helps users identify which extension module is erroneous when the operating systems get unstable after they install several modules.

To achieve the robustness, the fail-safe mechanism must provide the ability to detect errors of extension modules and the ability to recover from the errors. The errors of extension modules should be detected as early as possible before it has a bad influence to the rest of the operating systems. Also, it is important that the errors are detected as accurately as possible so that users can identify the cause of them easily. After an error is detected, the fail-safe mechanism should recover from the error in order to prevent the operating system from its influence. The fail-safe

mechanism needs to always prepare for recovery so that it can recover from errors at any time.

The fail-safe mechanism should detect illegal memory accesses and deadlocks and then recover from them. There are two kinds of illegal memory accesses: hardware-trapped memory accesses and semantically illegal data modifications. The hardware-trapped memory accesses are detected as segmentation faults or alignment faults. The segmentation faults occur when an extension module accesses non-allowable memory; the alignment faults occur when an extension module accesses memory with a bad alignment. The semantically illegal data modifications are detected by checking for the contents of the data. For example, it should be detected as errors that a reference count becomes negative or that a pointer points to an out-of-range address. On the other hand, the deadlocks have various solutions: prevention, avoidance, detection. The fail-safe mechanism had better use deadlock detection so as not to decrease the utilization of system resources.

However, there is also the trade-off between fail-safety and performance. Sufficient fail-safety tends to degrade the performance of the extension modules because the fail-safe mechanism often involves large overheads. Considering the trade-off, many extensible operating systems have been proposed according to the purpose, but often must give up the fail-safety or something for the performance.

2.2 Related Work

In this section, we describe previous approaches in terms of the fail-safety.

2.2.1 User-Level Extension Modules

The microkernel operating systems like Mach [1] enables the extension modules to implement as user-level servers. At the early stage, most subsystems like file systems and network subsystems are included in the single UNIX server [14]. Therefore, the whole UNIX server crashes if one subsystem crashes due to the errors although the microkernel itself are not affected. To improve this insufficient fail-safety, Mach supporting multi-server is proposed [15]. For example, user-level protocol servers [35] allow a new network protocol to be implemented, and errors of each protocol server are not propagated to the rest of the operating system. An erroneous server is simply terminated by the kernel. The multi-server system achieves sufficient fail-safety, but the performance is sacrificed because the overheads of inter-process communication (IPC) and context switches are large [3, 7]. The performance of this cross-domain communication has been improved

in recent years [22].

The Chorus operating system [38, 37] allows users to download the extension modules created as the user-level servers into the kernel without recompiling them. This approach achieves both sufficient fail-safety at the user level and good performance at the kernel level. However, since the communication between the extension modules and the kernel is done by IPC even at the kernel level, the downloaded module is not enough efficient. For instance, in the read operation, the downloaded file system is 80% slower than the file system hand-crafted in the kernel [2].

2.2.2 Kernel-Level Extension Modules with Protection

Since the user-level servers tend to degrade the performance, some operating systems take approaches to download the extension modules into the kernel. Several UNIX systems like NetBSD allow users to link a loadable kernel module (LKM) with the kernel dynamically. A LKM is implemented as a part of the kernel and runs very efficiently after linked with the kernel. However, the errors due to the module can make the whole operating system crash because fail-safety is not considered at all.

To solve the problem of LKM, some extensible operating systems like VINO [43, 44] and DECADE [29] protect the extension modules downloaded into the kernel from the rest of the operating system. VINO use software fault isolation [49, 45] to protect the kernel from illegal memory accesses due to the downloaded extension modules. It also limits the maximum amount of resources that the extension module can use at any given time and automatically releases the resources if a certain timeout expires. To recover from errors, VINO provides a kernel transaction system. VINO thus provides a relatively light-weight and sufficient fail-safety, but VINO entails certain fixed overheads even if the extension modules are enough stable. For instance, the overhead of SFI used in VINO is always from 5% to 200%, depending on applications.

DECADE protects extension modules in the kernel from each other using a 64-bit address space, and the rest of operating system is not involved with the crash of a module. To avoid extra overheads, the extension modules are randomly located in the huge 64-bit kernel address space and thereby almost all illegal memory accesses are detected by a page fault trap [50]. Also DECADE provides a mechanism for safe inter-module calls and switches the stack frames at the call to prevent erroneous modules from inspecting the stack frame. DECADE enables the system administrator to consider the trade-off between safety and runtime overheads and to select what

it does for each inter-module call. The overheads of the inter-module call are from 65% to 90% of those between modules running in a separate address space, but the overheads are large, compared with a procedure call in the kernel.

2.2.3 Kernel-Level Extension Modules using Safe Languages

As another approach to download the extension modules into the kernel safely, several systems use language supports. The packet filter [26] is downloaded into the kernel and multiplexes network packets to appropriate applications. Since the packet filter uses a language specific to multiplexing packets, it does not suit for a general use. Moreover, the safe execution of packet filter gets rather overheads. To reduce the overheads, more efficient packet filters have been proposed. The BSD packet filter [24] redesigns the original stack-based packet filter and is up to 20 times faster. The dynamic packet filter (DPF) [10] uses dynamic code generation and is 10% to 50% faster than the other fastest packet filters.

The SPIN operating system [4] allows the users to download the extension modules written in Modula-3 [27] into the kernel. Modula-3 is a type-safe language and does not cause memory access violation at runtime. Although Modula-3 is a general language and enables programmers to write most functions of operating systems, the facility of type-safeness may restrict the programming or suffer extra overheads. For example, Plexus [13] can extend network subsystems on SPIN. When an Ethernet header is read from byte-stream data received by a network device driver, the system must copy the header from byte-stream data so that the header can be accessed safely. To reduce this overhead, Plexus uses safe casts which restrict the accesses to a converted type. However, since programmers can convert the byte-stream data to any types, it is not enough safe. Additionally, because all memory accesses cannot be checked statically, SPIN also needs runtime checks like an array range check.

2.2.4 User-Level Library per Application

The Exokernel operating system [11, 16] or some systems using protocol library [23, 48] link the functions of operating systems as a library with application programs. The fail-safety is sufficient since an only application with which the library is linked is affected if the extension module created as a library crashes. In Exokernel, almost all functions of operating systems are exported as a library operating system to reduce the number of cross-domain. The kernel provides only the

facilities to bind resources securely and multiplex physical devices. Programmers can modify the library to extend functions of the operating system, but that is as difficult as directly modifying the monolithic kernel since the abstraction is very low.

Chapter 3

Multi-Level Protection

3.1 Observation

We believe that a sufficient fail-safe mechanism is not always necessary because we can assume that the extension modules are not malicious like those provided by trusted vendors. In many extensible operating systems like SPIN and VINO, every user can install new extension modules into the operating system and therefore a part of such extension modules may be malicious or untrusted. To prevent the operating system from being crashed even in this case, the sufficient fail-safe mechanism is always indispensable. On the other hand, if only system administrators can install new extension modules into the operating system, the possibility in which malicious extension modules are installed gets very lower. In this case, it is significant that the fail-safe mechanism allows the extension modules to be executed as efficiently as possible and, at the same time, detects the errors of the extension modules to protect the operating system.

In addition, we believe that the protection level provided by the fail-safe mechanism should be changed depending on the stability of the extension modules since the errors of the extension modules go on decreasing little by little. For example, the sufficient fail-safe mechanism is needed when programmers develop a new extension module, whereas any fail-safe mechanism is not necessary when they release it as a product.

3.2 What is Multi-Level Protection?

We propose a new fail-safe mechanism called multi-level protection. The multi-level protection enables users to change the protection level of the extension modules without modifying the binary

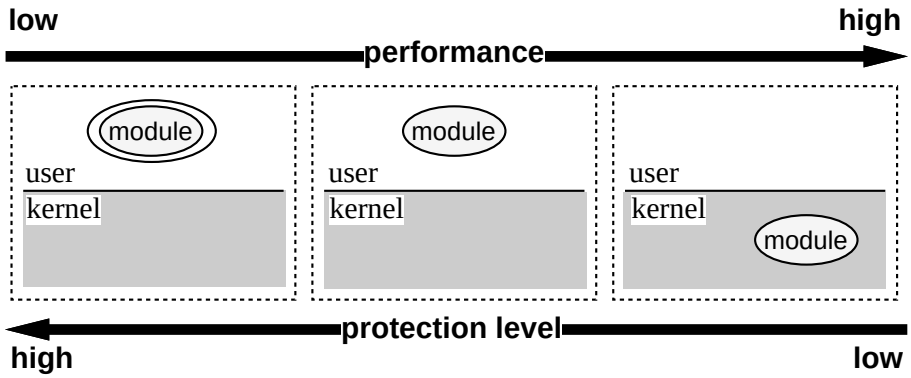

Figure 3.1: Consideration of the trade-off between fail-safety and performance of extension modules by the multi-level protection.

code. Using the multi-level protection, users can consider the trade-off between fail-safety and performance. The maximum protection level achieves sufficient fail-safety, but the minimal protection level does good performance. Figure 3.1 shows this concept roughly. For example, if an extension module is unstable, users can use a complete fail-safe mechanism, sacrificing the performance. On the other hand, if an extension module is stable, users can use a simplified fail-safe mechanism and then improve the performance.

The multi-level protection changes the ability to detect errors and the ability to recover from errors in order to change the protection levels. To decrease the ability of detection, some errors may not be detected. Also, to decrease the ability of recovery, some errors may neither be prepared for recovery nor be recovered from. For instance, the multi-level protection can allow illegal memory reads in order to reduce the overheads of the detection. It can also record no logs for recovery in order to reduce the overheads of the records.

To change the protection level without modifying the binary code of the extension modules, the multi-level protection provides an API to which the extension modules should conform. This API hides the differences between the implementation of fail-safe mechanisms. If programmers do not conform to this API, for example, using privileged instructions or using system calls, this facility is lost, but that is responsible to programmers. Also, the API exports the kernel data structure of high abstraction to the extension modules, and therefore programmers are easy to extend the operating system although the extensibility is not very high.

3.3 Applicability

We describe how the multi-level protection is used from the point of view of developers and users of extension modules.

3.3.1 Easy Development

So far, in many operating systems, the extension modules like file systems have been implemented directly in the kernel, or have been re-implemented in the kernel after they were developed as user-level libraries that emulates system calls, and so on for debugging. The former makes developers hard to debug the extension modules although the finished modules are very efficient. On the other hand, the latter makes developers write the extension modules twice for the emulation libraries and the kernel modules although debugging at the user level is easy.

This means that only a single protection level is not enough to make the extension modules easier to develop. The protection level of the extension modules should be changed during the development since the kinds and frequencies of errors depend on the stability of the extension modules. For instance, the extension modules include many errors at the beginning of the debug phase, but they are getting stable.

The multi-level protection provides appropriate fail-safety in each development phase below. In the debug phase, the fail-safe mechanism keeps the full capability of the protection even though this involves the maximum performance penalties. The errors are detected immediately and the accurate information of the errors is reported to programmers. These features considerably help programmers identify the reason of the errors and fix them. Moreover, the fail-safe mechanism can safely terminate the erroneous modules after they crash. Due to sufficient fail-safety, programmers can make rapid prototyping of the extension modules.

In the beta-test phase, the fail-safe mechanism does not have to keep the full capability of the protection. Rather it should run the extension modules as fast as possible so that the test users are satisfied with the performance to some degree. If the extension modules achieve better performance, more test users would use them and find more errors. Since the extension modules are expected to be fairly stable in this phase, only relaxed protection is needed. For example, it has only to detect and recover from a few kinds of errors depending on timing such as deadlocks. Illegal memory accesses like an access of null pointer are expected not to frequently occur.

Finally, the extension modules released as a product need the fail-safe mechanism no longer.

Unnecessary protection is removed and they can run as almost efficiently as one directly implemented in the kernel by hand.

3.3.2 Extension Modules by Third-Party Vendors

There are a number of extension modules by third-party vendors. Such extension modules may be unstable and crash every few days since the third-party vendors may misunderstand the specification of the extension modules and may not test the extension modules sufficiently. They include device drivers of minor devices like a CD-ROM changer and some third-party file systems like NT file system (NTFS) for PC UNIX, which the operating system vendors officially do not support. Although these unstable extension modules may frequently crash, the crash is acceptable if users seriously want to use them at any cost. However, the rest of the operating system should be kept stable even at that time.

The multi-level protection makes it possible to safely run these unstable extension modules. The fail-safe mechanism protects the extension modules sufficiently, and safely detaches the extension modules from the operating system if the extension modules crash. Therefore the rest of the operating system is not affected by erroneous extension modules.

On the other hand, many extension modules supplied by the third-party vendors are stable and do not need the fail-safe mechanism. The multi-level protection can run stable extension modules without any protection and eliminate the performance penalties. If there is still a possibility that the extension modules involve errors, the multi-level protection can run the extension modules at a lower protection level and more efficiently.

Chapter 4

Implementation

We have implemented the CAPELA operating system on the basis of NetBSD 1.3.2 [28]. CAPELA is an extensible operating system with the multi-level protection proposed in Chapter 3. CAPELA is running at Intel and SPARC platforms. But since the platform dependent part is a few, CAPELA can be easily ported to the other platforms that NetBSD 1.3.2 supports.

4.1 Overview

In the CAPELA operating system, programmers can create the extension modules as programs independent from the kernel in order to extend the functions of the operating system. The extension modules are driven by events hooked in the operating system kernel and achieve the functions of new subsystems, communicating with the kernel. The communication is done through a protection manager provided by CAPELA. A protection manager is provided per extension module and is a gateway between the extension module and the kernel. The protection manager provides a fail-safe mechanism, cooperating the kernel, so that the extension module can manipulate the kernel data safely. CAPELA provides multiple protection managers, each of which provides a different protection level to the extension modules. CAPELA enables users to exchange the protection managers in order to change the protection level of the extension modules, and thereby implements the multi-level protection.

The implementation of the protection managers and the kernel for the extension modules at the user level is largely different from that for the extension modules at the kernel level. In case that an extension module is located at the user level, it needs shared memory for communication with the kernel. The kernel uses an upcall mechanism to invoke the extension module when hooked

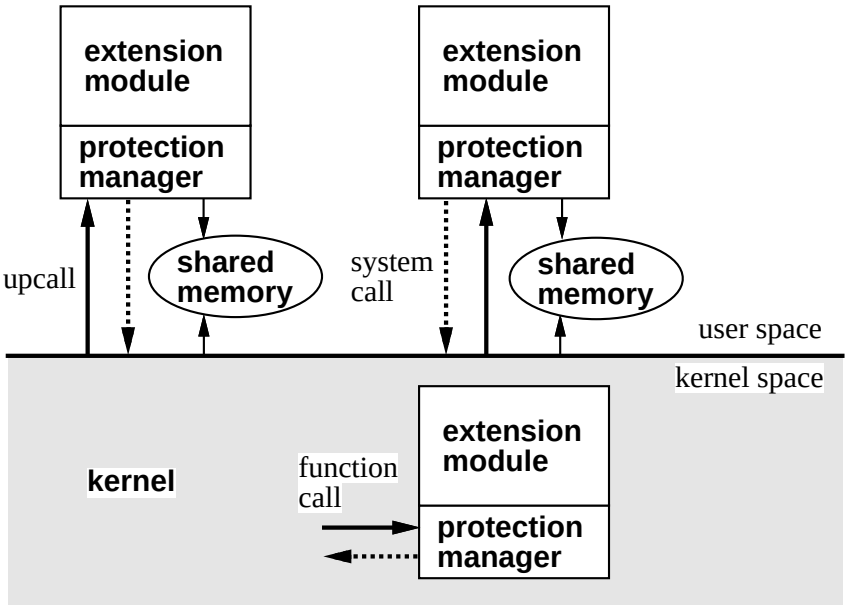

Figure 4.1: The relationship among extension modules, protection managers, and the kernel.

events occur. In case that an extension module is located at the kernel level, the communication between the extension module and the kernel is done through the kernel memory. The invocation from the kernel to the extension module is a direct function call. Figure 4.1 depicts the overview of the CAPELA operating system.

4.2 Changing Protection Levels

CAPELA allows users to change the protection level of an extension module by exchanging protection managers. CAPELA provides multiple protection managers, each of which provides a different protection level and can detect and recover from a different kind of error. Depending on the stability of the extension module, users can select the protection manager that provides an appropriate protection level at that time. Figure 4.2 illustrates that protection manager 3 is selected from four protection managers that provide various protection levels.

In the current implementation, the protection managers are implemented as a user-level library or a kernel library. The user-level library is used when the extension module is running at the user level, whereas the kernel library is used when the extension module is running at the kernel level. Because of this difference, users need to use two different ways to change the protection levels.

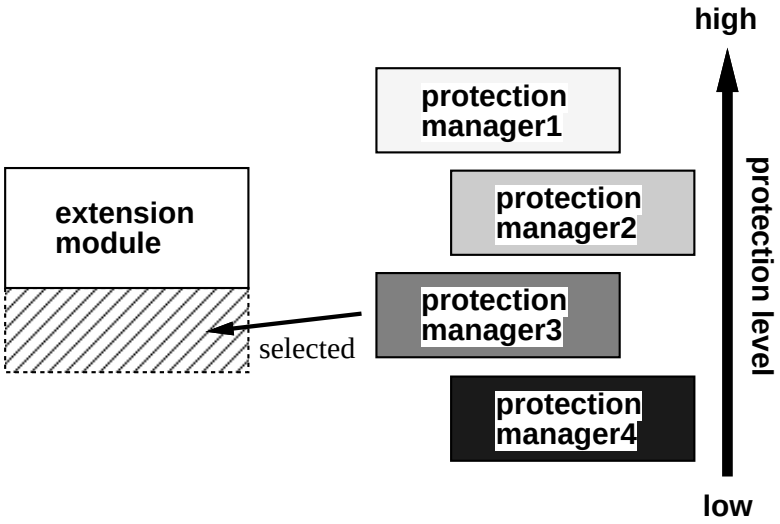

Figure 4.2: Changing the protection level of an extension module by selecting one of protection managers.

When users change the protection level of the extension module between the user levels, they only need to change its command line option and to restart it. In this case, relinking the extension module with a new protection manager is not necessary because an old protection manager and new one are involved in the same binary code. On the other hand, when users change the protection level of the extension module between the user level and the kernel level, they need to relink the extension module with a new protection manager. In case that users change the address space where the extension module runs from the user level to the kernel level, users can link the extension module with the kernel dynamically using the LKM mechanism after relinking the extension module with the kernel library. In any case, it is unnecessary to recompile the extension module. However, if recompiling the extension module is allowable, it can make the performance of the extension module better, in particular, when the protection level gets the lowest.

4.3 Protection Manager

The protection manager has three kinds of responsibilities to an extension module. First of all, the protection manager registers and unregisters the extension module to the kernel. In particular, it safely detaches the extension module from the kernel even if the module terminates abnormally.

Second, the protection manager plays a role of a gateway between the kernel and the extension module. If the protection manager receives upcalls from the kernel, it invokes callback functions of the extension module corresponding to the upcalls. Conversely, when the extension module needs to access the kernel functions or the kernel data, the protection manager does that instead. To play this role, the protection manager provides an API to the extension modules. Third, the protection manager protects the kernel from errors of the extension module. Depending on the protection level it provides, the protection manager protects the memory for the kernel data, or replicates the kernel data.

4.3.1 API

The protection manager provides an API to which all the extension modules must conform so that the extension modules can interact the kernel. Since all the protection managers the same API, the protection levels of the extension modules can be changed without modifying the extension modules. There is one API for callback functions and another for manipulation of the kernel data. Although forcing all the modules to conform to this API may restrict the programming of the extension modules, the advantages of changing the protection level without modification of the extension modules outweigh the disadvantages of this restriction. If the extension modules are malicious, the conformity to this API is not expected. However, we believe no extension modules are malicious. The details of this API are described in Appendix A.

4.3.1.1 Callback Functions

The protection manager provides an API for callback functions invoked by upcalls from the kernel. When an event hooked by an extension module occurs in the kernel, the kernel first notifies the protection manager of the event as an upcall. Second, the protection manager that received the upcall translates the arguments from data structure of low abstraction used in the kernel to one of high abstraction used in the extension modules. Finally, the protection manager invokes a callback function of the extension module.

To define programmers' own callback functions, they should override methods of classes for callback. CAPELA provides classes for callback and programmers can inherit them. For example, *FileSystem* class is a class for callback on file systems. To develop a new file system, programmers should inherit the class and override some methods such as *mount()* and *read()* if necessary.

4.3.1.2 Manipulation of the Kernel Data

The protection manager provides an API to manipulate the kernel data because the extension modules cannot directly access the kernel data in CAPELA for some reasons. First, this is because the protection manager prevents the kernel data from being illegally accessed. The kernel data is very complex since the execution efficiency is the most important and since pointers are used very frequently. For example, *mbuf* has very complicated structure for both the generality and the execution efficiency. Also, shared memory which the kernel data is put on is often protected by the protection manager. In this case, the kernel data is accessed only after the protection manager removes the protection of the shared memory.

The other reason is why CAPELA makes the extension modules deal with data structure of high abstraction. The protection manager translates the kernel data structure of low abstraction to one of high abstraction in order to hide the complexity of the kernel data structure. For instance, programmers can manipulate a chain of *mbuf*s as *MbufChain* class instead of manipulating multiple *mbuf*s with pointers. Also the API allows programmers only to increment and decrement a reference count one by one.

The protection manager also provides an API equivalent to one provided by the kernel since the extension modules running at the user level cannot directly use an API provided by the kernel. For example, an NFS [39] server needs to access a local file system like UFS, and therefore an API to access a local file system in the kernel is needed if the module of the NFS server is running at the user level. To enable a UDP [30] module and a TCP [32, 6] module at the user level to access an IP [31, 34] layer, an API for the operations to the IP layer is needed. The protection manager issues system calls or emulates these facilities to achieve them.

4.3.2 Protection Techniques

CAPELA uses some protection techniques to detect illegal memory accesses and deadlocks. Various combinations of these techniques enable the protection manager to provide various protection levels.

4.3.2.1 Detection of Illegal Memory Accesses

Hardware-Trapped Memory Accesses CAPELA uses switching address spaces to detect illegal memory accesses. The extension modules are located either in a user address space or in the kernel address space. If an extension module is located in a user address space, its illegal memory

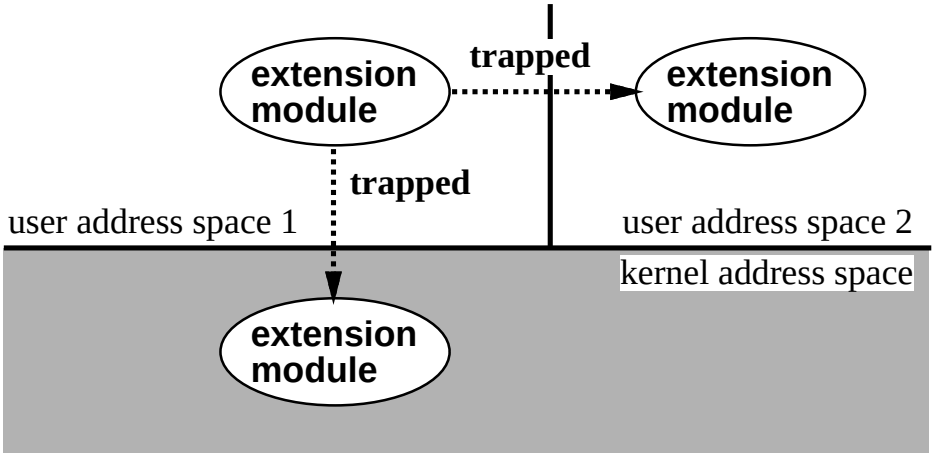

Figure 4.3: The memory protection by switching address spaces.

accesses to the kernel memory and the other processes' memory are trapped by hardware of the memory management unit (MMU) as illustrated in Figure 4.3. After the MMU detects such illegal memory accesses, they are notified to the CAPELA operating system and then CAPELA can terminate the extension module. The overheads for enabling this protection, however, are rather large due to increasing the number of context switches and the amount of data copies between address spaces. For instance, to pass data between an extension module at the user level and the other user process, two context switches and two copies of the data are needed at least; whereas only a context switch and a copy are needed at most between an extension module at the kernel level and a user process. Also, if the extension module is a user process, CAPELA can prevent the module from exhausting resources like CPU and memory by the limitation of the user process mechanism.

To decrease these overheads, CAPELA also allows users to locate the extension modules in the kernel address space. To embed an extension module into the kernel, the extension module is linked with the kernel dynamically using the mechanism of loadable kernel module.

When an extension module are located in a user address space, the module and the kernel use shared memory to communicate with each other. The important kernel data is put on this shared memory and therefore destroying the memory causes CAPELA to crash or to get unstable. To prevent the extension modules from illegally accessing the kernel data, CAPELA protects the shared memory using the virtual memory subsystem. While the code fragments of the extension

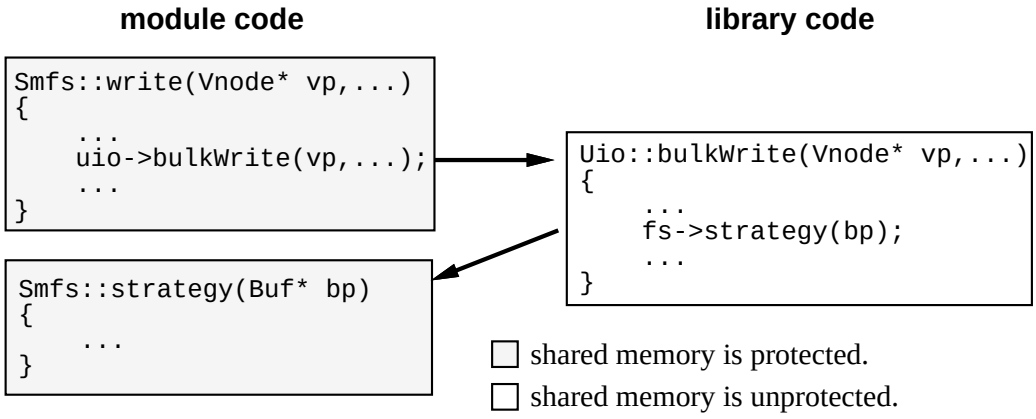

Figure 4.4: An example of changing the protection of shared memory according to the function calls.

modules is executed, the shared memory is protected. On the other hand, while the code fragments of the protection manager are executed, the protection is removed so that the protection manager can access the kernel data on shared memory. That is to say, the protection of the shared memory is removed when an extension module calls the functions of the protection manager, whereas the shared memory is protected again when the functions are exited. Figure 4.4 depicts in which code fragments shared memory is protected.

CAPELA allows the protection manager to select how the shared memory is protected. As the strongest protection, the protection manager can unmap the shared memory. This enables the protection manager to prevent the kernel data on the shared memory from being destroyed and from being illegally read. Trapping illegal reads to the kernel data helps errors be detected earlier although the illegal reads do not affect the system directly. As weaker protection, the protection manager can change the protection of the shared memory to read-only. This enables the protection manager to prevent the kernel data only from being illegally modified. In the SPARC architecture, changing the protection to read-only is faster than unmapping the memory and very useful because it sacrifices the ability of detection and can get better performance instead. In the Intel architecture, however, the overheads to change the protection to read-only are almost the same with the overheads to unmap the memory, and this protection appears to have both lower ability of detection and poorer performance. However, in fact, because it is not necessary to change the protection when the extension modules only read and do not modify the shared

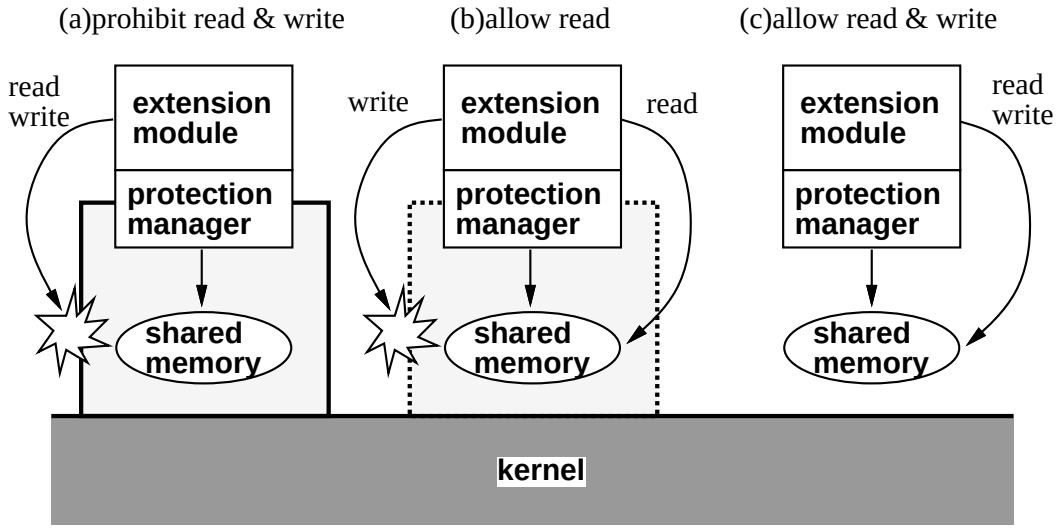

Figure 4.5: Three kinds of protections of shared memory.

memory, this protection is often useful in either architecture. As the weakest protection, the protection manager can also protect no shared memory. This leads good performance, sacrificing the protection completely. Figure 4.5 illustrates three kinds of protections of shared memory.

Semantically Illegal Data Modifications The protection manager replicates the kernel data and checks for the contents of replicas so that it can detect semantically illegal data modifications. The extension modules access the replicas instead of the raw kernel data. The modifications to the replicas are first checked and are then written back to the raw kernel data like Figure 4.6. To check for the replicas, the protection manager uses various knowledges on the kernel data structure that it supports. For example, the knowledges are: that the reference counts are 0 or positive, that the size of each buffer is often limited by the minimum and the maximum, and that some data are put on the shared memory necessarily.

CAPELA allows the protection manager to select what types of kernel data is checked and how the kernel data is checked. For example, if users give up checking for any loops by pointers, the protection manager does not need to traverse pointers and then the overheads of the traverse are reduced although it is possible that the errors are detected lately or cannot be detected.

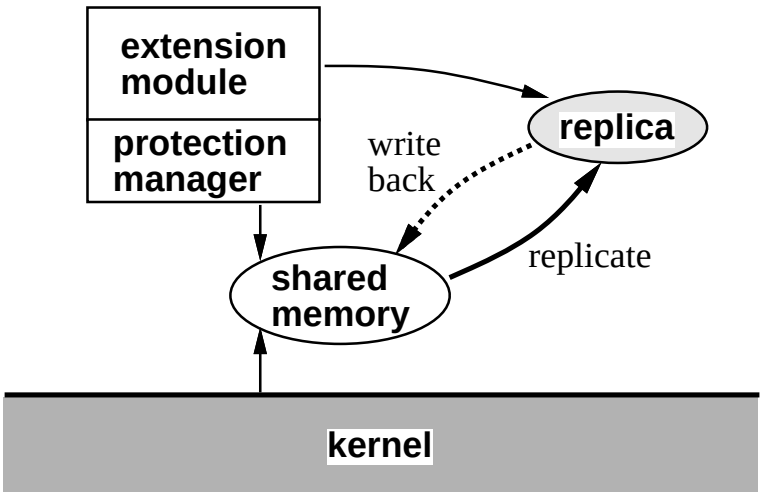

Figure 4.6: The replication and write-back of the kernel data.

4.3.2.2 Detection of Deadlocks

When the extension modules lock and unlock a resource, they should issue *lock_and_wait* system call with the argument of `LW_LOCK` and `LW_UNLOCK`, respectively. Likewise, when they wait for a locked resource, they should issue *lock_and_wait* system call with the argument of `LW_WAIT`. Using these information, the kernel of CAPELA detects whether deadlocks occur or not. Since deadlocks are caused by the interaction not only among multiple threads in an extension module but among multiple extension modules and the kernel, the kernel deals with the detection and recovery. The detection and recovery in the kernel is mentioned in Section 4.4.5.

4.3.3 Other Topics

4.3.3.1 Emulation of Interrupt Level

The protection manager emulates interrupt disabling so that an upcall is not interrupted by another upcall. In the kernel, the invocation of routines of subsystems is done from not only system calls but also interrupt handlers. If an interrupt handler invokes a routine of a subsystem when a system call has invoked the same routine, the integrity of the kernel states may be lost. To prevent this, the kernel can disable interrupts by changing a CPU interrupt level. However, since the extension modules at the user level cannot change a CPU interrupt level appropriately, the protection manager changes an upcall enable flag in the kernel. If the protection manager needs

to disable interrupts, it sets the upcall enable flag in the kernel, and the kernel does not issue any upcalls while the flag is set. This is not complete emulation of an interrupt level, but enough to run subsystems at the user level.

4.3.3.2 Thread

The protection manager uses threads so that it can process multiple upcalls which depend on each other. For example, consider that, after an upcall which waits for a vnode of UFS is done to an NFS module, the other upcall which waits for the same vnode to the same module is done. Since the upcalls put data in the same stack, the growth of the stack of the former upcall destroys the neighboring stack of the latter upcall if the former upcall continues first. Threads can prevent this problem because each thread has an independent stack.

The threads that CAPELA provides are different from normal threads in two points. First of all, the contexts of our threads are not switched periodically. Since each upcall should be executed exclusively like when the functions of subsystems are executed in the kernel, context switches between upcalls are done only when a thread yields CPU to sleep. Likewise, when a thread are woken up by the kernel or the other thread, the thread is first entered into the runnable queue and then switched after the current running thread terminates or sleeps. Second, our threads change an upcall enable flag in the kernel for emulation of interrupts when the context is switched.

4.4 The Kernel

The kernel of the CAPELA operating system provides several facilities for enabling to create the extension modules at the user level and to achieve the fail-safe mechanism.

4.4.1 Register and Unregister of Extension Modules

CAPELA provides the facility to register and unregister the extension modules. When an extension module at the user level is registered, the protection manager issues *modregist* system call. In this system call, the kernel first constructs information for the management of the extension module. The information consists of an upcall handler, a process ID, a module name, an upcall enable flag, the information of shared memory, and so on. The upcall handler is used when the extension module is upcalled by events in the kernel. The process ID and the module name is used for identifying the module. If the upcall enable flag is 0, upcalls to the extension module is disabled.

The information of shared memory includes the address of the shared memory for communication between the kernel and the extension module, the size, and so on. Finally, the system call invokes a different routine for initialization depending on a type of the extension module, e.g. a file system module, a network subsystem module, and so on.

When an extension module terminates normally or abnormally, the protection manager issues *modunregist* system call. This system call first cleans up the kernel to get rid of the influence of the extension module. This clean-up routine is the same with one for recovery described in Section 4.4.4.

4.4.2 Shared Memory

CAPELA provides memory shared between the kernel and each extension module. System V shared memory is provided in NetBSD 1.3.2, which is the base of CAPELA, but it is for general use. This general shared memory allows users to limit the access right to the memory only in the style of the UNIX access control. Therefore users can set readable, writable, and executable to only three types of owner, group, and others. To make matters worse, checking for the access right to the shared memory is done only once when it is mapped, and thereafter anyone can access the memory with the specified access mode such as read-only or read-write. Such an access model is not enough to keep shared memory safe from malicious user processes.

To solve this problem, we have implemented a new shared memory with fine-grain access control. An extension module can access its own memory shared with the kernel freely. Our shared memory appears to be the same with the general shared memory from the extension module, but it has two different points. One is to be automatically mapped when the protection manager calls *modregist* system call. Therefore our shared memory is not mapped illegally. The other point is to protect the shared memory using *shmprotect* system call instead of *mprotect* system call. Since CAPELA allows only the protection manager owning the shared memory to issue *shmprotect* system call, the protection of our shared memory is not removed illegally.

CAPELA disables the shared memory to be accessed by the other user processes as illustrated in Figure 4.7. It seems to be easy because CAPELA can make the other user processes not map the shared memory of the extension module. However, it is necessary for the memory to be accessed from the kernel even in these processes' context. For instance, an user process issues *read* system call and then the system call may access the shared memory of an extension module from the

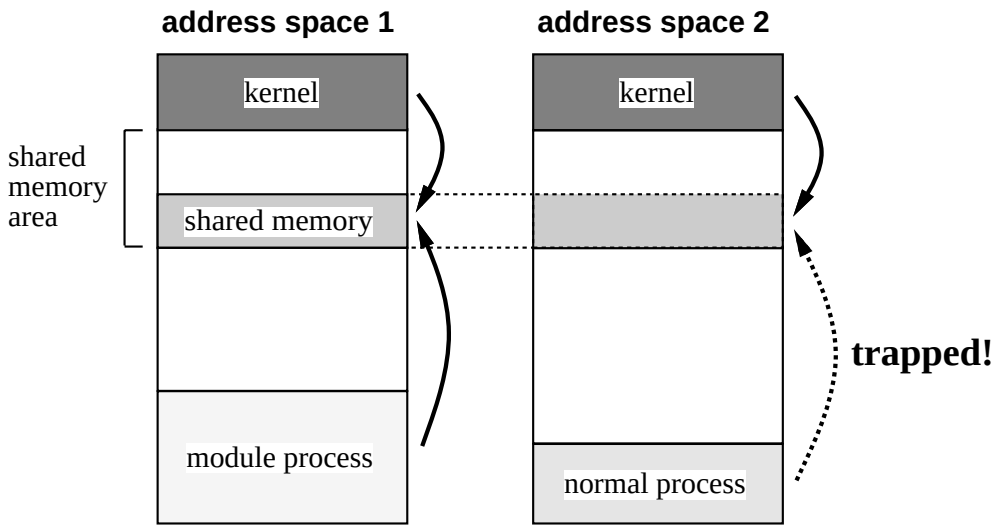

Figure 4.7: Shared memory with unique access control for communication between an extension module and the kernel.

kernel. Since the kernel is permanently mapped into the high part of every process's address space in CAPELA, allowing the kernel to access to the shared memory simply causes to allow all the user processes to access it. To enable only the kernel to access it, all the pages of the shared memory is mapped as the pages inaccessible from the user address space. This is done by setting a bit of a page table entry so that the page allows only the kernel to access. This fashion is general since most of latest hardware have that bit.

Our shared memory has the limitation that it cannot be accessed in interrupt handlers. During interrupts, a page table is undefined since any interrupt handlers are executed without relation to any processes. Our shared memory can be swapped out unlike the kernel permanently resided on physical memory, and then if a page fault occurs, it leads system crash.

Why does CAPELA use shared memory for communication between the kernel and each extension module? Many systems like Mach use IPC instead. The reason is why shared memory has two advantages against IPC. First of all, communication using shared memory gets more efficient than that using IPC. IPC needs to copy data from the kernel address space to an extension module's address space (the reverse is not always necessary), whereas shared memory needs no copy. Second, shared memory enables to communicate complicated data structure more easily and more efficiently. For example, shared memory allows data structure with pointers as long as the pointers

point to the data on the same shared memory. In case of using IPC, the system must traverse pointers and copy the data to which they point, or do that lazily when the data is needed.

Shared memory, however, has a disadvantage. Shared memory is dangerous because it can be destroyed by an extension module owning it if the module has errors. On the other hand, IPC is safer because the data for communication is copied from an extension module's address space to the kernel address space and because the extension module cannot destroy the kernel data directly. But we believe that the dangerousness can be avoided. Our shared memory does not allow the other user processes to access the memory illegally. Destruction of the shared memory by the extension module owning it is prevented if the protection manager makes the protection level high and protects the shared memory. We believe that the extension module is not malicious, and therefore this is sufficient to avoid danger of the shared memory.

4.4.3 Upcall

When a hooked event occurs in the kernel, CAPELA issues an upcall to an extension module hooking the event if the module is running at the user level. Exactly the upcall is received the protection manager. Upcalls are implemented using a similar mechanism to signals. A routine for calling an upcall handler, called trampoline code, are put on a user-level stack of every process on initializing it like that for signals. There are two differences between upcalls and signals. From this two points of view, we have implemented this new notification mechanism. First of all, upcalls enable CAPELA to pass arguments to an extension module, whereas signals pass an only signal number. Second, upcalls are a special mechanism that the only kernel can issue them, whereas signals can be issued by not only the kernel but also all user processes and the system can be confused.

Upcalls are done as follows:

1. When a process raises an event hooked by an extension module, CAPELA makes the process sleep and then forces context switching to the process of the extension module to be upcalled. Since a routine for context switching to an arbitrary process is not provided in NetBSD 1.3.2 of the base of CAPELA, we have implemented it newly.

2. An upcall routine puts an upcall handler, the arguments of the upcall and other information for returning from the upcall on the user-level stack of the extension module. The arguments

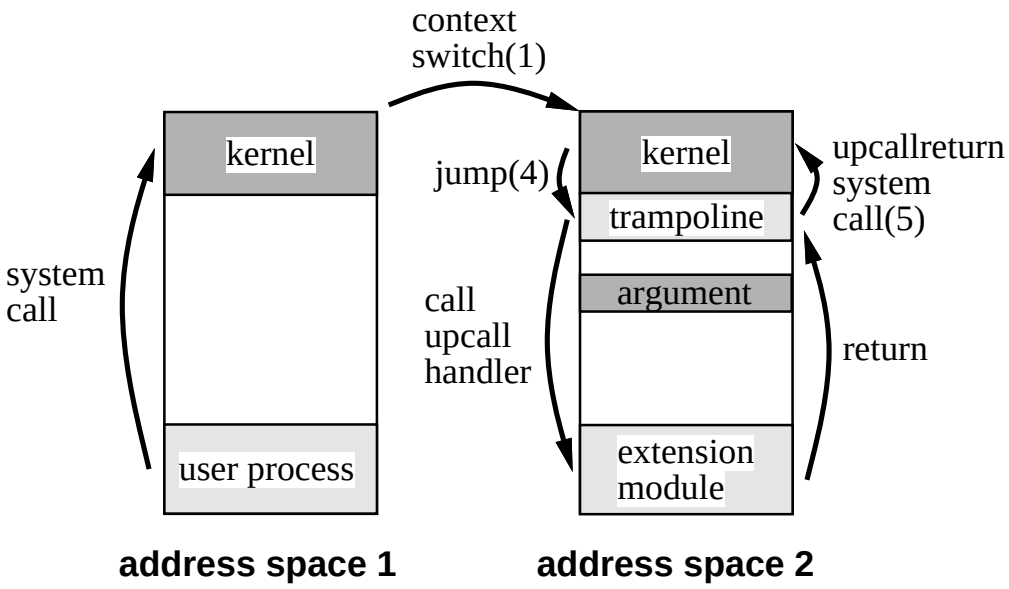

Figure 4.8: The movement of the control of the execution on upcall.

are allocated on the shared memory so that the extension module at the user level can access them.

3. The upcall routine rewrites a return address from the kernel to the extension module of a user process so that a new return address points to the trampoline code on the user-level stack.

4. After the control of execution moves to the trampoline code, the routine invokes the upcall handler passed by the argument.

5. After the upcall handler returns, the trampoline code issues *upcallreturn* system call, which restores an original return address from the kernel and terminates the upcall.

Figure 4.8 shows the movement of the control of the execution.

Upcalls cannot be done during interrupts because it is impossible to switch the context from an interrupt handler to the other processes, and because CAPELA does not allow an interrupt handler to access the shared memory in order to allocate the arguments of upcalls. To avoid this problem, CAPELA delays the upcalls that are issued during interrupts and the upcalls are done after the interrupt handler finishes.

4.4.4 Recovery from Illegal Memory Accesses

CAPELA has the ability to recover from an error of the extension modules so that the kernel is not affected by the abnormal termination of erroneous modules. For the causes to make the kernel unstable in the abnormal termination, it is considered (1)that the kernel data refers the shared memory owned by the erroneous modules and (2)that the erroneous modules have modified kernel states so as to get unstable.

In the former case, the kernel must modify the references to the shared memory so that the kernel does not access non-existing memory and does not occurs a kernel fault. First of all, the kernel removes all entries for the extension module. In the file system modules, for example, the mount structure for the management of mount information is removed from the mount list. At the same time, the kernel changes the vnodes for the current directory held by the process structure if the directory is on the file systems. In the current implementation, the current directory is changed to `/`. In the network subsystem modules, the protocol switch structure is removed from the protocol switch table.

In the latter case, on the other hand, the kernel must restore the kernel states modified by the erroneous modules so that the kernel is kept stable. To restore the kernel states, the kernel prepares a log per extension module and records the operations with which kernel states are modified by the extension modules in the log. In the current implementation, four operations are recorded: increment and decrement of the reference count of vnodes, and lock and unlock of vnodes. To keep the size of the log small, an operation recorded in the log is deleted when a new operation can cancel out the previous operation. For instance, the increment and the decrement of the reference count of the same vnode are cancelled out. Likewise, the lock and the unlock of the same vnode are also cancelled out. When restoring the unstable kernel states, the kernel examines the log and executes operations reverse to those in the log. For example, the reverse operation of the increment of the reference count is the decrement, and that of the lock is the unlock. Figure 4.9 illustrates the record of a lock operation in a log and the recovery based on the log.

4.4.5 Detection and Recovery of Deadlocks

CAPELA periodically checks whether the system falls into a deadlock state. When locking, unlocking, and waiting for resources, the extension modules notify the system of that, so CAPELA creates a wait-for-graph (WFG) on the basis of these information like Figure 4.10. Then CAPELA

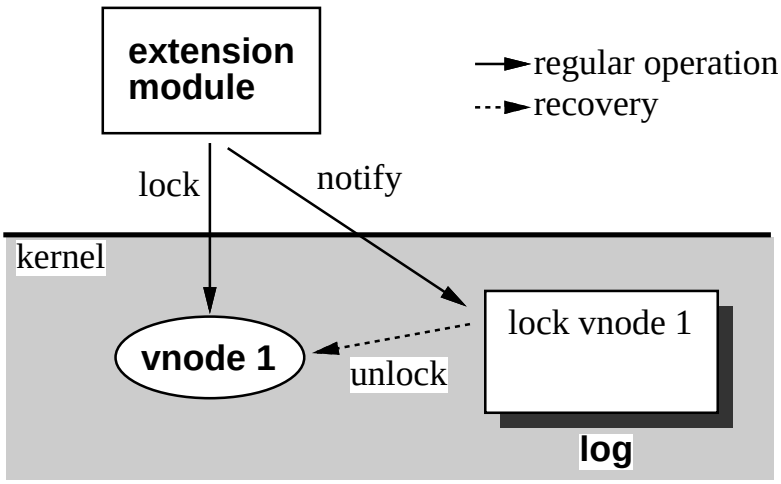

Figure 4.9: The record of operations in a log and the recovery based on the log.

checks for the WFG by examining the dependencies of resources. The algorithm for check are as follows:

1. CAPELA marks all resources in the wait list of each thread of extension modules `SL_UNTOUCH`.
2. For each wait list, CAPELA pushes all the resources into a stack and marks them `SL_INSTACK`.
3. CAPELA peeks the top of the stack, and if the mark of the resource is `SL_EXTRACT`, CAPELA pops it from the stack, marks it `SL_DELETE`, and repeat this operation of 3. Otherwise, CAPELA marks it `SL_EXTRACT` and search the thread of extension module that locks this resource.
4. If there is no such thread, CAPELA backtracks previous operations so that all the resources with `SL_EXTRACT` mark on top of the stack are poped and are marked `SL_DELETE`.
5. If such a thread exists, CAPELA examines each resource of the wait list of the thread. If CAPELA finds out a resource with `SL_EXTRACT` mark, there is a loop in the WFG, that is, the system is in a deadlock state. If CAPELA finds out a resource with `SL_UNTOUCH`, CAPELA marks it `SL_INSTACK`. If any resources of the wait list is not marked as `SL_UNTOUCH`, CAPELA backtracks.

CAPELA allows the protection manager to change the interval between checks for deadlocks. If users want to detect deadlocks earlier and to prevent the suspension of the services by deadlock-

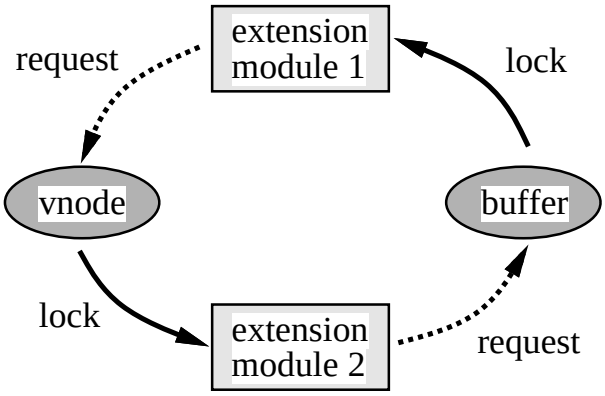

Figure 4.10: The example of a wait-for-graph in CAPELA.

ing modules as short as possible, they can make this interval shorter although this degrades the performance of the whole system. On the other hand, if they want to decrease the overheads, they can make the interval longer although it takes more time to detect deadlocks.

Because the occurrence of deadlocks depends on timing, CAPELA tries to resolve deadlocks so that the deadlocking modules can continue to run. To destroy a loop in the WFG to resolve deadlocks, CAPELA first finds out where the loop is and then temporarily releases one of the locks in the loop. Although CAPELA should release the only lock that causes the deadlock, because finding that lock is difficult, CAPELA releases one of locks in the loop randomly and stops the module whose lock is released temporarily until it can obtain the lock again.

4.4.6 Special System Calls

CAPELA provides several special system calls for the protection manager to use internally. These system calls are used to create extension modules at the user level and to achieve a part of facilities of a fail-safe mechanism. Although not only the protection manager provided by CAPELA but also the extension modules written by programmers can issue these system calls, we believe that non-malicious extension modules do not do that.

modregist system call registers the process that issues this system call as an extension module to the kernel. For the arguments, the type of the extension module, its name, an entry point for receiving upcalls, and private data are passed. The type must be `MLP_FS` for a file system module and/or `MLP_NS` for a network subsystem module in the current implementation. The name must be

one that can identify the extension module. The private data is passed to an initialization routine for each module type. CAPELA allows the only super user, that is the system administrators, to use this system call so that normal user processes cannot install any extension modules illegally.

modunregist system call unregisters the extension module that issues this system call from the kernel. In this system call, kernel states are cleaned up so that the unregistered module does not leave a bad influence to the kernel. Since this system call allows the only process that has registered as an extension module by *modregist* system call to unregister itself, the other user processes cannot unregister any extension modules illegally.

shmalloc system call allocates the area of shared memory and returns the address. The maximum size is limited by the size of the shared memory. The only extension modules registered by *modregist* system call can use this system call.

shmfree system call frees the area of shared memory indicated by the argument. If the address to be freed does not point to the shared memory owned by the extension module, this system call returns an error. Like *shmalloc* system call, this system call does not also allow to be used by normal processes.

shmprotect system call changes the protection of shared memory. The argument is the combination of `PROT_READ` for read permission and `PROT_WRITE` for write permission. The protection of shared memory provided by CAPELA must be changed by this system call instead of *mprotect* system call. *mprotect* system call does not allow to change the protection of the shared memory. This system call are also secure because the only process owning the shared memory can use it.

upcallsuspend system call is used to wait for a signal or an upcall. On receiving a signal or an upcall, the extension module first processes it and then return.

kernfunc system call executes kernel functions specified by the argument. Since this system call may destroy the kernel data by the execution of kernel functions, normal user processes cannot use it. Some examples of the defined kernel functions are as follows:

KF_UCINTR This changes an upcall enable flag according to the argument.

KF_WAKEUPALL This wakes up a kernel thread waiting for the identifier specified by the argument.

KF_GET_NEW_FSID This returns an unique file system identifier.

KF_VREF This increments a reference count of a vnode of a file system in the kernel. This is used for NFS to access file systems in the kernel.

KF_KERNFS_GETATTR This returns an attribute of a vnode of a file system in the kernel. This is used for NFS to access file systems in the kernel.

KF_IP_OUTPUT This calls the output routine of an IP layer.

KF_SOSEND This sends a packet. The mbuf chain passed as the argument is copied to the kernel memory so that it can be accessed in interrupt handlers of a network device driver.

KF_RTALLOC This allocates a new routing table entry.

KF_SOSETUPCALL This sets an upcall handler for a socket. The upcall is done when the socket receives packets.

logctl system call controls the log for recovery. If the argument is `RECLOG_CHECK`, the kernel checks for the log to examine whether the kernel is unstable or not. If the argument is `RECLOG_ROLLBACK`, the kernel checks for the log and rolls back if the kernel is unstable. If the size of the log is not zero, it is determined that the kernel states is unstable.

lock_and_wait system call notifies the kernel of locking, unlocking, waiting for, and waking up on resources of the kernel such as *vnode*. When `LW_LOCK` for locking and the pointer to a resource are passed as the arguments, the kernel inserts the pointer in the lock list. The pointer inserted in the lock list is removed when this system call is issued with the arguments of `LW_UNLOCK` and the same pointer. On the other hand, when `LW_WAIT` for waiting for and the pointer to a resource, the kernel inserts the pointer in the wait list. The pointer inserted in the wait list is removed when this system call is issued with the arguments of `LW_WAKEUP` or `LW_WAKEUP_ALL` and the same pointer. In addition, waking up on resources are also done from wakeup routine in the kernel.

4.4.7 Other Topics

4.4.7.1 Insertion of Check Code

To protect the kernel from erroneous modules, code for check is inserted in the kernel. The code is mainly inserted in the head of kernel functions, and checks for the validity of the arguments. This validity check examines if the addresses that the arguments point to are safe. That an address is safe means that it points to the kernel memory or shared memory of existing extension modules.

If the validity check fails, the functions return an error if possible. The overheads of the check is almost negligible since the comparison of addresses takes less time than the execution of a function in many cases.

4.4.7.2 Optimized Memory Allocation

The kernel provides an optimized memory allocation routine for the extension modules at the kernel level. Our extension modules allocate and destroy more memory than ones hand-crafted in the kernel from the beginning because they frequently use temporary instances of classes, which are data structure of high abstraction. Since the memory allocation routine has rather large overheads, it becomes performance bottleneck. To solve this problem, the kernel allocates memory of a fixed size before the extension modules start, and does not use the general memory allocation routine. If the memory gets short, the kernel allocates more memory again for the extension modules.

4.5 Examples of Extension Modules

CAPELA allows users to extend the operating system every subsystem. The extension modules are therefore relatively course-grain. We believe, however, that the suppression of extensibility makes it easier to extend the operating system. If programmers want to change only a part of a subsystem, they can create a new extension module that delegates the execution to the original subsystem for the same routines. In the current implementation, CAPELA supports the development of file systems and network subsystems.

4.5.1 File System Module

The file systems are one of subsystems that are developed the most. Since the file systems affects the system performance largely, it is meaningful to improve the file systems and to develop new file systems. In particular, distributed file systems such as AFS [40] and Coda [41] are researched recently. CAPELA helps developers create such distributed file systems as well as local file systems.

In the current implementation of CAPELA, the file systems are implemented on top of the virtual file system (VFS) [25]. The file system modules serve the applications while they are mounted on directories. When they are mounted, *FileSystem::mount* method is called and then it prepares for the file systems. When they are unmounted from the directories, *FileSystem::unmount* method is called and then it cleans up the file systems. The most primitive operations for file

systems are read and write of files. When a file is read, *FileSystem::read* method is called. This method should read the contents of the file into a file buffer and copy it to a universal I/O buffer. Conversely, when a file is written, *FileSystem::write* method is called. This method should copy the contents of a universal I/O buffer to a file buffer and write it to a file on a physical device. Additionally, the file system modules should support changing a directory, getting a file status, getting directory entries, creating a file at least.

In distributed file systems, some of file operations like read and write are delegated to the server. The server performs requested operations to the local file system like UFS. Since the server may modify the states of the local file system in the kernel, the kernel records accesses to the local file system in a log, preparing an unexpected accident.

Programmers must develop a mount(8) program specific to their file system. The mount program should pass the unique name to identify the file system as an argument. The name is the same with one used when the file system module is registered to the kernel, and is used to find out a file system to be mounted. Also, the mount program can pass private data to *FileSystem::mount* method of the file system. The starting four bytes of the private data must be the size of the data because the data is copied between a process of the mount program and one of the file system module if the module is a user process.

4.5.2 Network Subsystem Module

The network subsystems are indispensable to recent operating systems. When programmers develop a new distributed file system, they often want to develop a new network protocol so that the distributed file system can communicate between clients and servers most efficiently. As a more close example to us, the present IP version 4 is being changed IP version 6 in order to prepare for the shortage of IP address in the near future.

In the current implementation of CAPELA, the network subsystems are implemented on top of an IP layer. The network subsystem modules serve the applications while they are bound to a socket. When they are bound to a socket, *NetworkSystem::attach* method is called and then it prepares for the network protocol. When they are detached from a socket, *NetworkSystem::detach* method is called and then it cleans up the network protocol. The most primitive operations for network subsystems are send and receive of packets. When a packet is passed from a socket layer above, *NetworkSystem::send* method is called. This method should divide a packet if necessary,

attach a protocol header, and call an output routine of an IP layer. Conversely, when a packet is received on a network device driver, *NetworkSystem::input* method is called. This method should control sequence of packets if necessary, and append it to a socket buffer. The data in a socket buffer is read by *recv* system call.

Chapter 5

Experiments

We experimented to make sure of the usefulness of the multi-level protection. We have three purposes in the experiments. The first purpose is to make sure that the execution performance is improved when users lower the protection level of the extension modules. The second is to measure the overheads of the maximum protection level. These overheads are not too important because the maximum protection level is used on debugging, but it is significant that the overheads are not too large. The third purpose is to measure the overheads of the minimum protection level. These overheads is caused by making the API of multiple protection managers same, and it is considered that the multi-level protection enables the extension modules to be efficient if these overheads are enough small.

For the experiments, we used two PCs, which have a Pentium II processor running at 400MHz. Each PCs was equipped with 128MB of RAM and a 10Mbps Ethernet. All experiments with network were made between two machines on the same Local Area Network (LAN). All measurements were done using the CAPELA operating system.

Since it would have been too difficult to experiment with all combinations of the protection techniques our system provides, we selected five characteristic combinations, listed in Table 5.1, and experimented with them. These combinations can detect the errors on memory protection listed in Table 5.2. We call the combination yielding the highest protection level *the first level* and call the combination yielding the lowest level *the fifth level*.

The first level locates the extension modules in user address spaces, protects shared memory from illegal memory reads and writes, and replicates the kernel data. The second level is different from the first level in that it protects shared memory only from illegal writes. The third level does

Protection technique	1st	2nd	3rd	4th	5th
Shared memory protection	√*	√**			
Kernel data replication	√	√	√		
Address space switch	√	√	√	√	

Table 5.1: Five characteristic combinations of protection techniques. (*Unmap shared memory **Change the protection of shared memory to read-only)

Type of error	1st	2nd	3rd	4th	5th
Illegal reads on shared memory	√				
Illegal writes on shared memory	√	√			
Semantically illegal data modifications	√	√	√		
Illegal accesses to the kernel data	√	√	√	√	

Table 5.2: Detectable errors on memory protection at each protection level.

not protect shared memory, but illegal accesses to shared memory may be able to be detected by checking for the replicated kernel data. The fourth level does not even replicate the kernel data, so illegal accesses to shared memory cannot be detected at all. However, it still receives a benefit of the protection as a user process at least. Finally, at the fifth level, the extension modules are embedded into the kernel without any protection. The extension modules at this level are exactly the same with ones hand-crafted in the kernel from the beginning except the overheads for using the same API among all protection levels.

For comparison, we have also measured hand-crafted version of extension modules in the kernel. These extension modules have exactly the same performance with subsystems in NetBSD 1.3.2, which has the traditional monolithic kernel.

5.1 File System Module

We have developed two file system modules: SMFS (Simple Memory File System) and SNFS (Simple Network File System). SMFS is a RAM disk, whose files reside in memory. The block size that SMFS can read and write from the memory at a time is 512 bytes. SNFS is a simplified

NFS [39], which consists of some clients and a server and communicates between the client and the server using remote procedure call (RPC) [47]. The RPC is done using UDP, and the block size that SNFS can read and write with RPC at a time is 512 bytes. The server reads and writes real files from a local file system UFS.

5.1.1 File Copy

We measured the time needed to copy a file on our file system. Since copying a file is one of the most fundamental operations to file systems, we can obtain the potential overheads of each of the protection techniques used in CAPELA. A file was copied from our file system to the same file system. The size of the copied file was 64KB and the block size for each *read* and *write* system call was 8KB. In SNFS, the client communicated with the server through a 10Mbps network.

Figure 5.1 and Figure 5.2 show the results of this experiment. These two figures mean that the performance of the file systems is improved when the protection level is lowered. The reason that the second level suffers larger overheads than the first level is probably why changing the protection of memory pages often takes more time than unmapping memory pages.

The ratio of the overheads of each protection level to the fifth level is described in Table 5.3. In SMFS, the first level is 211% slower than the fifth level; and in SNFS, the first level is 70% slower. These overheads are not small and the performance is rather degraded, but we think that it is acceptable for debugging the file systems. The overheads of the fifth level to the hand-crafted version are 3.7% and 1.4% in SMFS and SNFS, respectively, so they are almost negligible.

Since the above measurements were done just after booting computers, any caches like a file buffer cache and a vnode cache almost never hit. When all the contents of a file to be copied were cached and other caches were also hot, the result of the measurements on SMFS was like Table 5.4. For all protection levels, the penalties due to cache miss reduce, so the apparent ratio of the overheads increases. The first level is 640% slower than the fifth level, and the fifth level is

	1st	2nd	3rd	4th	5th
SMFS	3.11	3.26	1.94	1.62	1.00
SNFS	1.70	1.74	1.25	1.09	1.00

Table 5.3: The ratio of the overheads to the fifth level in SMFS and SNFS.

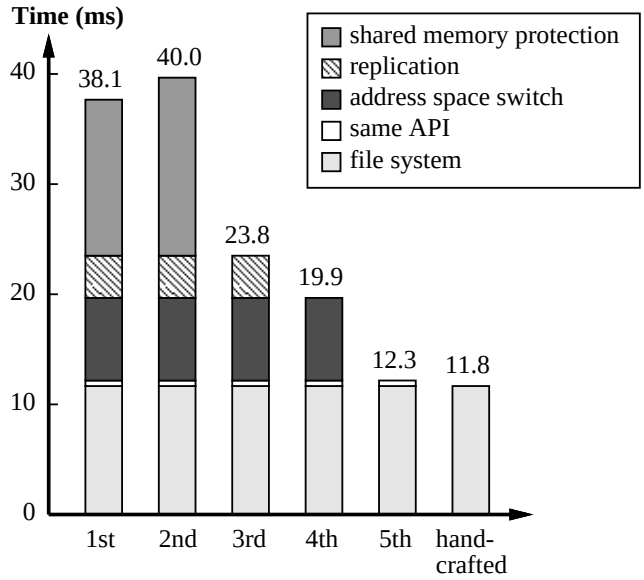

Figure 5.1: The time needed to copy a 64KB file on SMFS and the breakdown of the overheads.

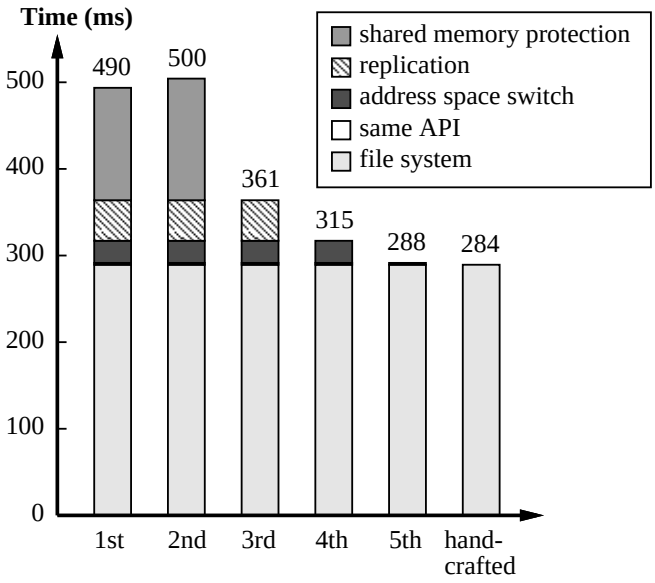

Figure 5.2: The time needed to copy a 64KB file on SNFS through a 10Mbps network and the breakdown of the overheads.

	1st	2nd	3rd	4th	5th	hand-crafted
SMFS (hot cache)	29.1	30.8	14.1	10.4	4.0	3.3

Table 5.4: The time needed to copy a 64KB file on SMFS (hot cache). (*msec*)

	1st	2nd	3rd	4th	5th	hand-crafted
SNFS (localhost)	284	282	149	114	67.6	63.4

Table 5.5: The time needed to copy a 64KB file on SNFS without network. (*msec*)

18% slower than the hand-crafted version.

In SNFS, for comparison, we also measured the time needed to copy a 64KB file at a single host because debugging of network file systems is often done at a single host. The server and the client communicated without using Ethernet. Table 5.5 shows the result. The first level is 350% slower than the fifth level, and the fifth level is 6.3% slower than the hand-crafted version. The overheads are larger than SNFS through network, but we think that they are not too large to debug a SNFS module.

5.1.2 Compile

We measured the time needed to compile a small program on our file system. Since compiling a program is one of the most frequently used applications, we can obtain the realistic overheads of our approach. The program consists of five source files and two header files of the C language and had about 2,000 lines. We compiled the program using `gcc` 2.7.2.2. But temporary files were put on the local file system.

Figure 5.3 and Figure 5.4 show the results of this experiment. In practical circumstances like this experiment, the first level is 16% and 19% slower than the fifth level in SMFS and SNFS, respectively. The performance is good enough even for normal use. In addition, the overheads of the fifth level to the hand-crafted version are just 3.1% and 1.4% in SMFS and SNFS, respectively, and are enough small. Also, in SNFS without network for comparison, the overheads of each protection level are almost as efficient as in SNFS through network. (Table 5.6)

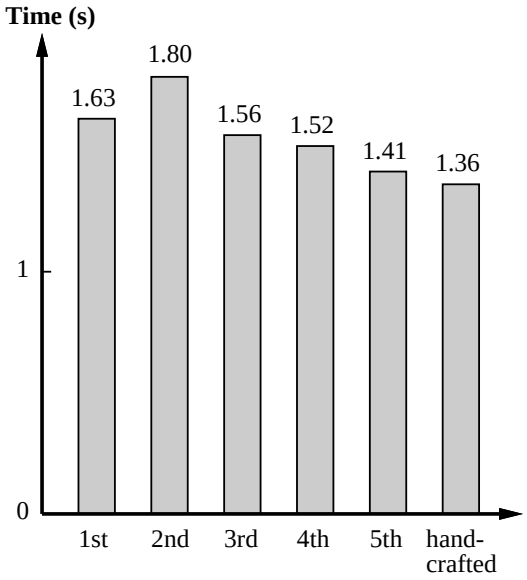

Figure 5.3: The time needed to compile `ps` program on SMFS.

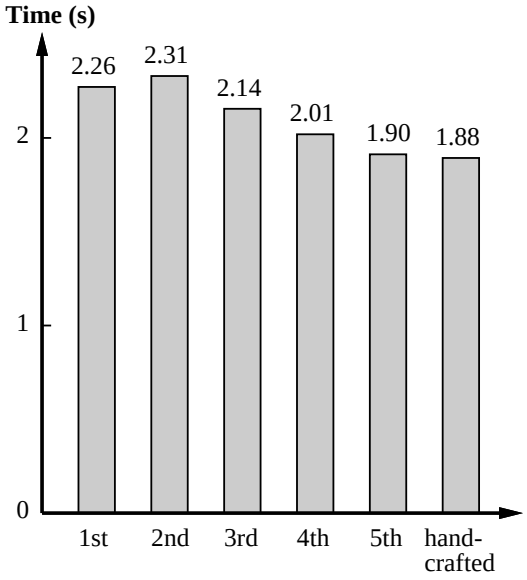

Figure 5.4: The time needed to compile `ps` program on SNFS through a 10Mbps network.

	1st	2nd	3rd	4th	5th	hand-crafted
SNFS (localhost)	2.21	2.31	1.95	1.90	1.82	1.79

Table 5.6: The time needed to compile `ps` program on SNFS without network. (*sec*)

5.2 Network System Module

We have developed two network subsystem modules: SUDP (Simple User Datagram Protocol) and STCP (Simple Transmission Control Protocol). SUDP and STCP are exactly the same with UDP [30] and TCP [32, 6], respectively, except that control operations have not implemented.

5.2.1 Round-Trip Latency

Round-trip latency reflects the overhead induced by a protocol when it transfers a packet from sender to receiver. We sent a packet with data of 1 byte and measured the time needed from sending a packet to receiving a packet to obtain round-trip latency. We repeated sending and receiving a packet 1,000 times and divided the elapsed time by 1,000 to obtain an average round-trip latency.

Figure 5.5 and Figure 5.6 show the results of this experiment. These two figures mean that the performance of network subsystems is improved when the protection level is lowered. The ratio of the overheads of each protection level to the fifth level is described in Table 5.7. In SUDP, the first level is 182% slower than the fifth level; and in STCP, the first level is 220% slower. These overheads of the maximum protection are, we think, acceptable for debugging. The overheads of the fifth level to the hand-crafted version are 12% and 2.8%. It is considered that the overheads in SUDP are a little large because the send and input routines of SUDP are small and the overheads for using the same API relatively get large.

For comparison, we also measured round-trip latency in SUDP and STCP at a single host

	1st	2nd	3rd	4th	5th
SUDP	2.82	2.79	1.68	1.56	1.00
STCP	3.20	2.72	1.55	1.39	1.00

Table 5.7: The ratio of the overheads to the fifth level of round-trip latency in SUDP and STCP.

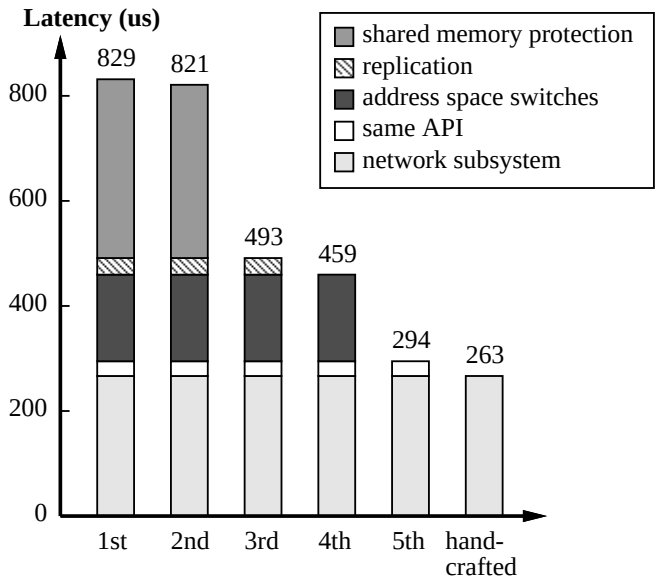

Figure 5.5: Round-trip latency in SUDP through a 10Mbps network and the breakdown of the overheads.

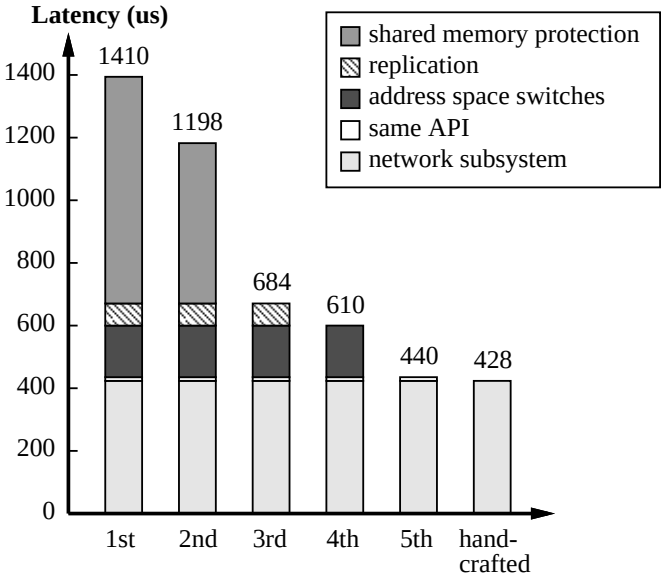

Figure 5.6: Round-trip latency in STCP through a 10Mbps network and the breakdown of the overheads.

	1st	2nd	3rd	4th	5th	hand-crafted
SUDP (localhost)	1,152	1,079	419	355	103	74
STCP (localhost)	1,851	1,564	594	520	141	101

Table 5.8: Round-trip latency in SUDP and STCP without network. (μs)

without network. Table 5.8 shows the results. Both the overheads for the maximum protection and for using the same API are rather large. This is because this measurement is the worst case that it sends only one byte.

5.2.2 Throughput

Throughput is the other indicator to measure the execution performance of network subsystems. Throughput indicates how much data is sent using a protocol per unit time. We did not measure throughput for SUDP because it depends on the windowing and acknowledgment strategies. STCP uses the same strategies with TCP, but SUDP does not provide the strategies like UDP. We calculated throughput from the time needed from sending packets of total 1MB to receiving all the packets. The buffer size of both *send* and *recv* system calls is 8KB.

Figure 5.7 shows the result of this experiment. Throughput of each protection level is near the hand-crafted version. The overheads for the maximum protection are 10% and those for using the same API are only 1.3%. However, when the packet transfer is done without network, throughput is very low as shown in Table 5.9. Throughput of the first level is one tenth of that of the fifth level, and that of the fifth level is still 35% lower than that of the hand-crafted version.

	1st	2nd	3rd	4th	5th	hand-crafted
STCP (localhost)	19.0	21.3	51.8	72.9	194	262

Table 5.9: Throughput in STCP without network. ($Mbps$)

5.2.3 SNFS with SUDP

We measured the performance of SNFS, which we have developed as a file system module, with SUDP instead of UDP. Network file systems are one of the most usual and important applications

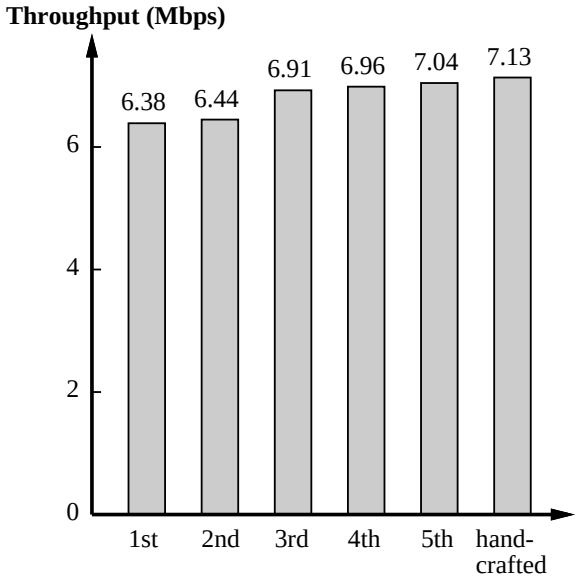

Figure 5.7: Throughput in STCP through a 10Mbps network.

of datagram protocol, so we can obtain practical overheads of SUDP from this experiment. We measured the time needed to copy a 64KB file on SNFS with SUDP. A file was copied from SNFS to the same SNFS, and the block size for each *read* and *write* system call was 8KB. We experimented using SNFS of the fifth level, which is embedded into the kernel.

Figure 5.8 shows the results of this experiment. Through a 10Mbps network, the first level is 75% slower than the fifth level, and the fifth level is 2.7% slower than the hand-crafted version. These overheads are enough small for the purpose of each protection level. The result of this experiment without network is shown in Table 5.10.

We also measured for SNFS of the fourth level, which is running at the user level. The results are shown in Table 5.11. From this result, it is confirmed that the interaction between the two

	1st	2nd	3rd	4th	5th	hand-crafted
SNFS/5 (localhost)	336	312	140	115	47	42

Table 5.10: The time needed to copy a 64KB file on SNFS of the fifth level with SUDP without network. (*msec*)

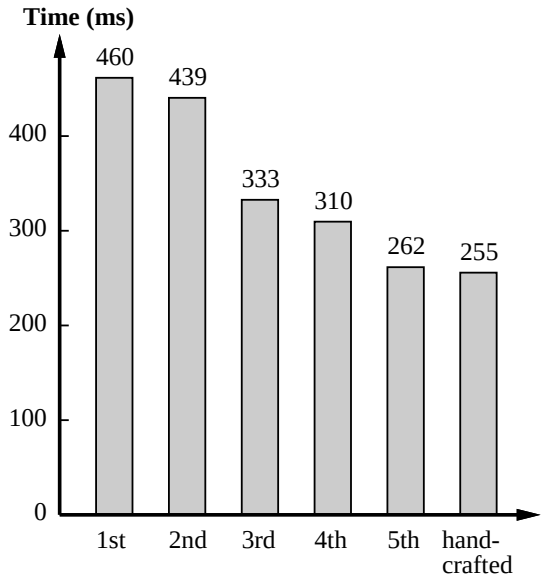

Figure 5.8: The time needed to copy a 64KB file on SNFS of the fifth level with SUDP through a 10Mbps network.

	1st	2nd	3rd	4th	5th	hand-crafted
SNFS/4 (Ethernet)	517	526	400	351	297	290
SNFS/4 (localhost)	341	350	212	157	87	81

Table 5.11: The time needed to copy a 64KB file on SNFS of the fourth level with SUDP. ($msec$)

user-level modules can work properly, and that the overheads do not increase dramatically.

5.2.4 FTP with STCP

We measured the file transfer rate using FTP [33] with STCP instead of TCP. FTP is one of the most frequently used applications of data-stream protocol, so we can obtain practical overheads of STCP from this experiment. We received a 1MB file by `get` command of FTP and measured the transfer rate.

The results are shown in Figure 5.9. Like throughput of STCP, influences due to the overheads for the maximum protection level and for using the same API are small when a file is transfered through a 10Mbps network. The transfer rate of the first level is 16% lower throughput than that

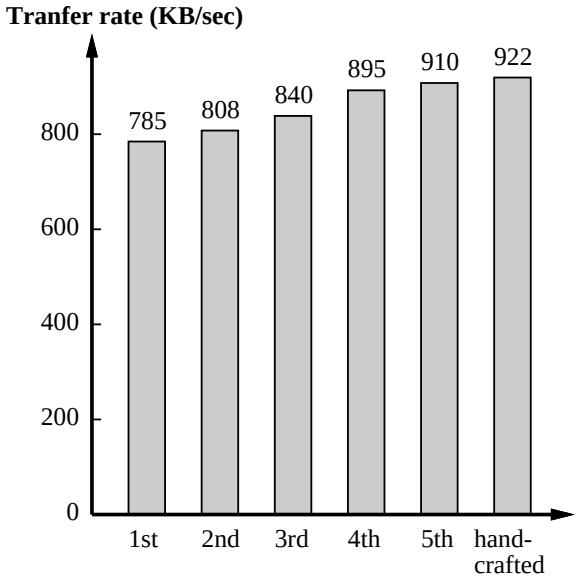

Figure 5.9: The file transfer rate of FTP with STCP through a 10Mbps network.

	1st	2nd	3rd	4th	5th	hand-crafted
FTP (localhost)	2.24	2.47	5.76	7.77	16.0	19.4

Table 5.12: The file transfer rate of FTP with STCP without network. (MB/sec)

of the fifth level, and that of the fifth level is only 1.3% lower than that of the hand-crafted version. The result of this experiment without network is shown in Table 5.12.

5.3 Source Code Translation

At the fifth level, which is the lowest protection level, the overheads of some of our extension modules are not enough small due to the overheads for using the same API and for modifying no binary codes of the extension modules among protection levels. The details of the overheads are extra memory allocation for instances of classes, dispatches of virtual functions of the C++ language, extra function calls by the fact that inline extraction is impossible, and so on.

If the extension modules are allowed to translate the source codes and recompile them, these overheads are reduced. To translate source codes, preprocessing of source codes is generally used.

Macro is very helpful to preprocess source codes for programs of the C language, but it is not enough powerful for C++ programs because C++ method invocation is difficult to be translated by macro. Therefore OpenC++ [8] is useful to translate source codes of C++ programs. Using OpenC++, we can translate C++ classes of higher abstraction to the kernel data structure. This reduces the overheads of runtime translation between them. For example, the following code fraction

```
int Smfs::write(Vnode* vp, Uio* uio, int ioflag, Ucred* cred)
{
    uio->bulkWrite(vp, DEV_BSIZE, size, cred);
}

int Uio::bulkWrite(Vnode* vp, int blksize, int filesize, Ucred* cred)
{
}
```

is translated to

```
int Smfs::write(struct vnode* vp, struct uio* uio, int ioflag,
                struct ucred* cred)
{
    Uio_bulkWrite(uio, vp, DEV_BSIZE, size, cred);
}

int Uio_bulkWrite(struct uio* uio, struct vnode* vp, int blksize,
                  int filesize, struct ucred* cred)
{
}
```

We have translated the source code of the SUDP module and the SMFS module. But we have done the translation by hand within the limits of OpenC++. For the SUDP module, we measured latency when sending 1 byte packet through a 10Mbps network. For the SMFS module, we measured the time needed to copy a 64KB file with all caches hot.

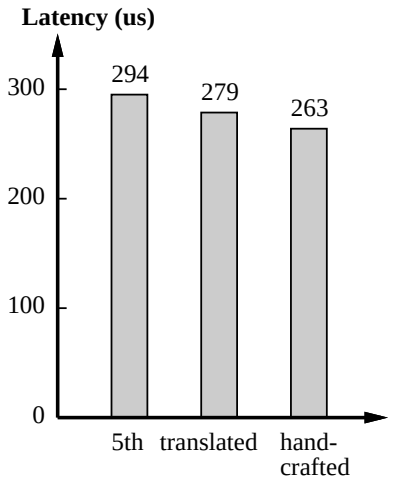

Figure 5.10: Comparison of the latency in SUDP after source code translation.

The results are shown in Figure 5.10 and Figure 5.11. The translated SUDP module is 5.4% faster than SUDP of the fifth level, and the overheads are 6.1% compared with the hand-crafted one. For the SMFS module, the translated module is 10% faster than the hand-crafted one. It is considered that this is due to CPU caches. In fact, when L1 and L2 caches of CPU are not used, the hand-crafted SMFS module is 4% faster than the translated one. These results indicate that source code translation makes the extension modules almost as efficient as hand-crafted ones.

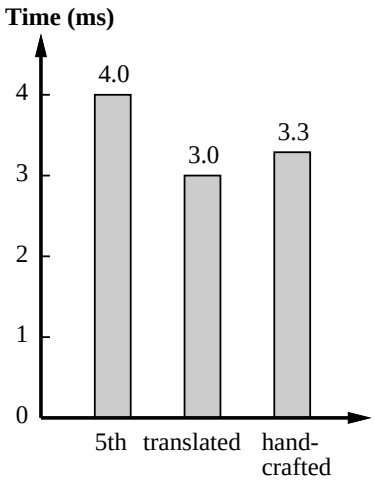

Figure 5.11: Comparison of the time needed to copy a 64KB file on SMFS with all caches hot after source code translation.

Chapter 6

Conclusion

6.1 Summary

We proposed the multi-level protection, which enables users to install the extension modules at various protection levels without modifying the binary code. Users can change the protection level depending on the stability of the extension modules. We have implemented the CAPELA operating system, which is an extensible operating system with the multi-level protection. CAPELA provides multiple protection managers, each of which provides a different protection level, and allows users to change the protection level of the extension modules by exchanging the protection managers. From some experiments, we confirmed that the performance of the extension modules is improved if the protection level is lowered. Also, when the protection level is maximum, the overheads is not too large to debug the extension modules. On the other hand, when the protection level is minimum, the overheads is enough small to use them normally. As a result, it is considered that the multi-level protection can support debugging erroneous extension modules and make stable modules run efficiently.

6.2 Future Directions

- Kernel-level extension modules with protection

  CAPELA should allow the extension modules to run at various protection levels at the kernel level as well as at the user level. In the current implementation, the extension modules embedded in the kernel run without any protection. However, since the differences between the user level and the kernel level are large, it is useful for debugging to enable users to change

the protection level at the kernel level. For instance, if the extension modules run with the least protection in the kernel, the fail-safe mechanism may be able to detect timing-critical errors other than deadlocks. DECADE [29] allows users to protect the extension modules in the kernel and to change the protection levels to some degree, and then we believe that we can use that framework for our purpose.

- Automatic change of protection levels

  It is convenient for CAPELA to change the protection levels of the extension modules depending to the stability automatically. To achieve this facility, there are two problems to be solved. One is that CAPELA exchanges the protection managers without restarting the extension modules. When the extension modules continue to run at the user level after the protection level is changed, CAPELA must keep integrity of kernel data in regard to data replication. If replication is stopped, all replicas must be written back to the raw kernel data. Conversely, if replication is continued, all the kernel data must be replicated. When the address space where the extension modules are located changes between a user address space and the kernel address space, CAPELA must migrate the code and data of the extension modules.

  The other problem is when CAPELA changes the protection levels of the extension modules. If CAPELA lowers the protection level in spite of an unstable module, the module would make the whole system crash. But if CAPELA does not change the protection level in spite of a stable module, the system performance would not be improved. We believe that this problem can be solved using statistics information of the extension module only when users desire changing the protection level automatically. If an extension module runs for a longer time than certain threshold without detecting any errors, CAPELA should lower the protection level. On the other hand, if an extension module crashed, CAPELA should start the module with a higher protection level next time.

- Reducing the overheads for interaction between extension modules

  The extensible operating systems suffer the overheads for interaction between their functions since each function is modularized and that prevents efficient communication between each other. For example, disk I/O buffers and network buffers have different data structure each other, and therefore extra copies are needed between the two data structure. Some techniques

have been proposed for the optimization specific to particular fields [9, 12]. We believe that such inefficiency is generally solved by technologies of source code translation. The source code translation enables different data structure to be dealt transparently. The translation fundamentally requires recompilation of extension modules, but we think that it can be done dynamically when the modules are installed in the operating systems.

References

[1] Accetta, M., 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*, pp. 93–112, June 1986.

[2] Armand, F., "Give a Process to your Drivers!," in *Proceedings of the EurOpen Autumn 1991 Conference*, Sep. 1991.

[3] Bershad, B. N., T. E. Anderson, E. D. Lazowska, and H. M. Levy, "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systems*, vol. 8, pp. 37–55, Feb. 1990.

[4] Bershad, B. N., 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 of the 15th ACM Symposium on Operating Systems Principles*, pp. 267–284, Dec. 1995.

[5] Cao, P., E. Felten, and K. Li, "Implementation and Performance of Application-Controlled File Caching," in *Proceedings of the 1st Usenix Symposium on Operating System Design and Implementation*, pp. 165–177, Nov. 1994.

[6] Cerf, V. and R. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Transactions on Communications*, vol. 22, pp. 637–648, May 1974.

[7] Chen, J. and B. N. Bershad, "The Impact of Operating System Structure on Memory System Performance," in *Proceedings of the 14th Symposium on Operating System Principles*, pp. 120–133, Dec. 1993.

[8] Chiba, S., "A Metaobject Protocol for C++," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 285–299, Oct. 1995.

[9] Druschel, P. and L. L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 189–202, Dec. 1993.

[10] Engler, D. and M. F. Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 53–59, Aug. 1996.

[11] Engler, D. R., M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 251–266, Dec. 1995.

[12] Fall, K. and J. Pasquale, "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability," in *Proceedings of the 1993 USENIX Technical Conference*, pp. 327–333, Winter 1993.

[13] Fiuczynski, M. E. and B. N. Bershad, "An Extensible Protocol Architecture for Application-Specific Networking," in *Proceedings of the USENIX 1996 Annual Technical Conference*, pp. 55–64, Jan. 1996.

[14] Golub, D., R. Dean, A. Forin, and R. Rashid, "Unix as an Application Program," in *Proceedings of the Summer 1990 USENIX Conference*, pp. 87–95, June 1990.

[15] Guedes, P. and D. Julin, "Object-Oriented Interfaces in the Mach 3.0 Multi-Server System," in *Proceedings of IEEE Second International Workshop on Object Orientation in Operating Systems*, Oct. 1991.

[16] Kaashoek, M. F., D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie, "Application Performance and Flexibility on Exokernel Systems," in *Proceedings of the 16th Symposium on Operating Systems Principles*, pp. 52–65, Oct. 1997.

[17] Kourai, K., S. Chiba, and T. Masuda, "Necessity of Multi-Level Protection for Developing New File Systems," in *SIG notes of Information Processing Sciety of Japan (97-OS-76)*, pp. 37–42, Aug. 1997.

[18] Kourai, K., S. Chiba, and T. Masuda, "Fail-Safe Mechanism for Extensible Operating Systems," in *SIG notes of Information Processing Sciety of Japan (98-OS-77)*, pp. 197–202, Feb. 1998.

[19] Kourai, K., S. Chiba, and T. Masuda, "Multi-Level Protection: A New Fail-Safe Mechanism for Extensible Operating Systems," *Journal of Information Processing Society of Japan*, vol. 39, pp. 3054–3064, Nov. 1998.

[20] Kourai, K., S. Chiba, and T. Masuda, "Operating System Support for Easy Development of Distributed File Systems," Tech. Rep. TR–98–01, Department of Information Science, University of Tokyo, 1998.

[21] Kourai, K., S. Chiba, and T. Masuda, "Operating System Support for Easy Development of Distributed File Systems," in *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 551–554, Oct. 1998.

[22] Liedtke, J., "On μ-Kernel Construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles*, pp. 237–250, Dec. 1995.

[23] Maeda, C. and B. N. Bershad, "Protocol Service Decomposition for High-Performance Networking," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 244–255, Dec. 1993.

[24] McCanne, S. and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," in *Proceedings of the Winter 1993 USENIX Conference*, pp. 259–269, Winter 1993.

[25] McKusick, M. K., "The Virtual Filesystem Interface in 4.4BSD," *Computing Systems*, vol. 8, pp. 3–25, Winter 1995.

[26] Mogul, J. C., R. F. Rashid, and M. J. Accetta, "The Packet Filter: an Efficient Mechanism for User-Level Network Code," in *Proceedings of the 11th Symposium on Operating System Principles*, pp. 39–51, Nov. 1987.

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

[28] http://www.netbsd.org.

[29] Nishimura, T., "Extensible OS with In-Kernel Protection for Kernel Modules," Master's thesis, Department of Information Science, University of Tokyo, Feb. 1998.

[30] Postel, J., "User Datagram Protocol." RFC 768, Aug. 1980.

[31] Postel, J., "Internet Protocol." RFC 791, Sep. 1981.

[32] Postel, J., "Transmission Control Protocol." RFC 793, Sep. 1981.

[33] Postel, J. and J. Reynolds, "File Transfer Protocol (FTP)." RFC 959, Oct. 1985.

[34] Postel, J., C. Sunshine, and D. Cohen, "The ARPA Internet Protocol," *Computer Networks*, vol. 5, pp. 261–271, July 1981.

[35] Reynolds, F. and J. Heller, "Kernel Support for Network Protocol Servers," in *Proceedings of the USENIX Mach Symposium*, pp. 149–162, Nov. 1991.

[36] Ritchie, D. and K. Thompson, "The UNIX Time-Sharing System," *Communications of ACM*, vol. 17, pp. 365–375, July 1974.

[37] Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "Overview of the Chorus Distributed Operating System," in *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pp. 39–69, Apr. 1992.

[38] Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "CHORUS Distributed Operating System," *Computing Systems*, vol. 1, pp. 305–370, Dec. 1988.

[39] Sandberg, R., 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*, pp. 119–130, June 1985.

[40] Satyanarayanan, M., J. 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*, pp. 35–50, Dec. 1985.

[41] Satyanarayanan, M., 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*, vol. 39, pp. 447–459, Apr. 1990.

[42] Schulzrinne, H., "Operating System Issues for Continuous Media," *Multimedia Systems*, vol. 4, pp. 269–280, Oct. 1996.

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

[44] Seltzer, M. I., Y. Endo, C. Small, and K. A. Smith, "Dealing With Disaster: Surviving Misbehaved Kernel Extensions," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 213–227, Oct. 1996.

[45] Small, C., "MiSFIT: A Minimal i386 Software Fault Isolation Tool," Tech. Rep. TR–07–96, Harvard University Computer Science, 1996.

[46] Stonebraker, M., "Operating System Support for Database Management," *Communications of the ACM*, vol. 24, pp. 412–418, July 1981.

[47] Sun Microsystems, Inc., *Remote Procedure Call Protocol Specification*, Feb. 1986.

[48] Thekkath, C. A., T. D. Nguyen, E. Moy, and E. D. Lazowska, "Implementing Network Protocols at User Level," in *Proceedings of the ACM SIGCOMM'93 Symposium on Communications Architectures and Protocols*, pp. 64–73, Sep. 1993.

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

[50] Yarvin, C., R. Bukowski, and T. Anderson, "Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space," in *Proceedings of the 1993 Summer USENIX Conference*, pp. 175–186, June 1993.

Appendix A

API Reference

The following API is one provided by the protection manager. Programmers of the extension modules should conform to the API so that they can change the protection level of the extension modules without modifying the binary code.

A.1 API for Callback Functions

API for callback functions is invoked to execute the functions of the extension modules by the kernel. All the methods are virtual functions, and programmers should override them if necessary.

A.1.1 FileSystem class

All file system modules must inherit this class.

`void init()`

This is invoked to initialize the file system. It is used to initialize global data for the file system.

`int mount(Mount* mp, const String& path, FsOption* opt, Nameidata* ndp, Proc* p)`

This is invoked to mount the file system. The argument *path* indicates a directory on which the file system is mounted. Specific data for the file system is passed by *opt*, whose structure is defined by mount(8) program for the file system. In this method, programmers should create the root vnode and private data of the file system.

`int start(Mount* mp, int flags, Proc* p)`

This is invoked after *mount()*.

`int unmount(Mount* mp, int mntflags, Proc* p)`

This is invoked to unmount the file system. In this method, programmers should call *vflush()* for *mp* to remove all vnodes in the vnode list of the file system, and then free the root vnode and private data of the filesystem, if necessary.

`int statfs(Mount* mp, Statfs* sbp, Proc* p)`

In this method, programmers should set statistics information of the file system to *sbp*.

`int root(Mount* mp, Vnode** vpp)`

In this method, programmers should set a pointer to the root vnode to **vpp*. A reference count of the root vnode must be increased.

`int vget(Mount* mp, int num, Vnode** vpp)`

In this method, programmers should set to **vpp* a pointer to a vnode whose has a node number of *num*. A reference count of the vnode must be increased.

`int sync(Mount* mp, int waitfor, Ucred* cred, Proc* p)`

In this method, programmers should make all vnodes in the vnode list of the file system synchronize using *fsync()*.

```
int fhtovp(Mount* mp, struct fid* fhp, MbufChain* nam, Vnode** vpp, int* exflags,
        Ucred** credanon)
```

In this method, programmers should translate a file handle *fhp* to a vnode and set a pointer to it to **vpp*. This is basically used in network file systems.

`int vptofh(Vnode* vp, struct fid* fhp)`

In this method, programmers should translate a vnode *vp* to a file handle *fhp*. This is basically used in network file systems.

`int quotactl(Mount* mp, int cmds, uid_t uid, caddr_t arg, Proc* p)`

This is invoked by *quotactl* system call. The high 24-bit of *cmds* indicates a quota main command: `Q_QUOTAON` (enable quotas), `Q_QUOTAOFF` (disable quotas), `Q_GETQUOTA` (get limits and usage), `Q_SETQUOTA` (set limits and usage), `Q_SETUSE` (set usage), and `Q_SYNC` (sync disk copy of the file system's quotas). The low 8-bit of *cmds* indicates the type of quota: a user quota or a group quota.

`int access(Vnode* vp, mode_t mode, Ucred* cred, Proc* p)`

In this method, programmers should check an access right of a file specified by the vnode *vp*.

`int getattr(Vnode* vp, Vattr* vap, Ucred* cred, Proc* p)`

In this method, programmers should set an attribute of a file specified by the vnode *vp* to *vap*.

`int setattr(Vnode* vp, Vattr* vap, Ucred* cred, Proc* p)`

In this method, programmers should change the vnode *vp* according to the new attribute *vap*. For example, if the size is changed, they should call *truncate()*. if the access time or the modified time is changed, they should call *update()*.

`int lock(Vnode* vp)`

In this method, programmers should set a flag for lock if necessary. The flag is often in private data structure of the vnode *vp*.

`int unlock(Vnode* vp)`

In this method, programmers should clear a flag for lock if necessary.

`int islocked(Vnode* vp)`

In this method, programmers should check whether a flag for lock is set if necessary. If the vnode is locked, return 1; otherwise, return 0.

`int advlock(Vnode* vp, caddr_t id, int op, struct flock* fl, int flags)`

This is invoked to do advisory locks. Advisory locks mean that locks are enforced for only processes that request locks; whereas locks enforced for every process without choice are said to be mandatory locks.

`int lookup(Vnode* dvp, Vnode** vpp, CompName* cnp)`

In this method, programmers should look up a pathname specified by *cnp* and set a pointer to a vnode looked up to **vpp*. The argument *dvp* is the vnode of a directory on which the file system is mounted. If the lookup successes, return 0; otherwise, return `ENOENT`. But if the lookup fails and the lookup operation is for `CREATE` or `RENAME`, return `EJUSTRETURN`.

`int open(Vnode* vp, int mode, Ucred* cred, Proc* p)`

This is invoked to open a file specified by the vnode *vp*. In this method, programmers should do something specific to the file system if necessary.

`int close(Vnode* vp, int fflag, Ucred* cred, Proc* p)`

This is invoked to close a file specified by the vnode *vp*. In this method, programmers should do something specific to the file system if necessary.

`int create(Vnode* dvp, Vnode** vpp, CompName* cnp, Vattr* vap)`

This is invoked to create a file. The argument *dvp* is the vnode of a directory to create a new file. The argument *cnp* holds the name of a newly created file. The attribute of the file is passed by *vap*. In this method, programmers should set a pointer to a newly created vnode to **vpp*.

`int remove(Vnode* dvp, Vnode* vp, CompName* cnp)`

This is invoked to remove a file by *unlink* system call. The argument *vp* is the vnode for a removed file and *dvp* is the vnode for a parent directory of the file. The argument *cnp* indicates the name of a file to be removed. In this function, programmers should call *vrele()* to *dvp* and *vp*.

`int read(Vnode* vp, Uio* uio, int ioflag, Ucred* cred)`

This is invoked by *read* and *readv* system calls to read data from a file. In this method, programmers should copy the buffer of the vnode *vp* to *uio*. They may be able to use *bulkRead()* for *uio*.

`int write(Vnode* vp, Uio* uio, int ioflag, Ucred* cred)`

This is invoked by *write* and *writev* system calls to write data to a file. The argument *uio* holds data to be written. The size is returned by *getResident()* for *uio*. If the append flag `IO_APPEND` is set in the argument *ioflag*, programmers should append data to the end of a file specified by the vnode *vp*. In this method, programmers should copy data of *uio* to the buffer of the vnode *vp*. They may be able to use *bulkWrite()* for *uio*.

`int bwrite(Buf* bp)`

This is invoked to write the contents of the buffer *bp* to a file. Programmers can call only *bwrite()* for `bp`.

`int strategy(Buf* bp)`

This is invoked to read from or write to a file. If the read flag `B_READ` is set, programmers should read from a file to the buffer *bp*; otherwise, they should write the contents of buffer *bp* to a file. They can read from or write to devices, memory, network, and so on.

`int bmap(Vnode* vp, daddr_t bn, Vnode** vpp, daddr_t* bnp, int* runp)`

In this function, the logical block number *bn* should be translated to the physical block number and a pointer to it is set to **bnp*.

`int truncate(Vnode* vp, off_t length, int flags, Ucred* cred, Proc* p)`

This is invoked to change the length of a file. The argument *length* indicates a new length of a file. The argument *vp* is the vnode for a file whose size is changed. If `IO_SYNC` is set in the argument *flags*, programmers should make the change in a synchronous mode. In this method, programmers should allocate some disk blocks filled with zero if *length* is larger than the length of the file. Otherwise, they should free some disk blocks and fill the contents of the partial block following the end of the file with zero.

`int fsync(Vnode* vp, Ucred* cred, int waitfor, Proc* p)`

In this method, programmers should make dirty buffers on the vnode *vp* synchronize.

```
int update(Vnode* vp, struct timespec* access, struct timespec* modify,
           int waitfor)
```

In this method, programmers should change an access time and a modified time of a file specified by *vp*.

`int inactive(Vnode* vp)`

This is invoked to inactivate the vnode *vp* so that it is not used. In this method, programmers should write dirty buffer back by *vgone()* for *vp* if necessary.

`int reclaim(Vnode* vp)`

This is invoked to reuse the vnode *vp*. In this method, programmers should call *cachePurge()* and *freePrivateData()* for `vp`.

`int abortop(Vnode* dvp, CompName* cnp)`

This is invoked to abort an operation of the file system due to errors. In this method, programmers should free memory for the pathname of *cnp* using *freePathname()* for *cnp* if `HASBUF` is set and `SAVESTART` is not set in a flag of *cnp*.

`int mkdir(Vnode* dvp, Vnode** vpp, CompName* cnp, Vattr* vap)`

This is invoked to make a new directory. The argument *dvp* is the vnode for a directory where a new directory is made. *cnp* indicates the new directory name to be created and *vap* holds the attribute of a new directory, e.g. access permission. In this method, programmers should set a pointer to a vnode for a new directory to **vpp*.

`int rmdir(Vnode* dvp, Vnode* vp, CompName* cnp)`

This is invoked to remove a directory. The argument *dvp* is the vnode for a directory where a directory is made. *vp* is the vnode to be removed and *cnp* indicates the directory name to be removed.

```
int readdir(Vnode* vp, Uio* uio, Ucred* cred, int* eofflag, off_t* cookies,
            int ncookies)
```

This is invoked by *getdent* system call to read a block of directory entries. In this method, programmers should copy directory entries from the vnode *vp* for a directory to *uio* and should set 1 to *eofflag* if all entries have been read.

`int link(Vnode* dvp, Vnode* vp, CompName* cnp)`

This is invoked to make a hard link to a file. The argument *dvp* is the vnode for a directory where a hard link is created. *vp* is the vnode for an existing file to which a hard link is made. *cnp* indicates the new file name for a hard link. In this method, programmers should make sure that *dvp* and *vp* are on the same mount point, and should create a new file for a hard link.

`int symlink(Vnode* dvp, Vnode** vpp, CompName* cnp, Vattr* vap, char* target)`

This is invoked to make a symbolic link. The argument *dvp* is the vnode for a directory where a symbolic link is created. *cnp* indicates the new file name for a symbolic link and *vap* holds the attribute of a new file for a symbolic link. *target* is the string used in creating the symbolic link. In this method, programmers should create a new file for a symbolic link and write *target* to the file.

`int readlink(Vnode* vp, Uio* uio, Ucred* cred)`

This is invoked to read the contents of a symbolic link. The argument *vp* is the vnode for a symbolic link to be read. In this method, programmers should copy the contents to *uio*.

`int mknod(Vnode* dvp, Vnode** vpp, CompName* cnp, Vattr* vap)`

This is invoked by *mknod* and *mkfifo* system calls in order to create a special file. The argument *dvp* is the vnode of a directory to create a new file. The argument *cnp* holds the name of a newly created file. The attribute of the file is passed by *vap*. In this method, programmers should set a pointer to a newly created vnode to **vpp*.

`int ioctl(Vnode* vp, u_long command, caddr_t data, int fflag, Ucred* cred, Proc* p)`

This is invoked by *ioctl* and *fcntl* system calls. The arguments *command* and *data* are passed by system calls. *fflag* is `O_RDONLY`, `O_WRONLY`, and so on.

`int pathconf(Vnode* vp, int name, register_t* retval)`

This is invoked by *fpathconf* system call. The argument *name* is `_PC_NAME_MAX`, `_PC_PATH_MAX`, and so on. In this method, programmers should set the return value to *retval*.

A.1.2 NetworkSystem class

All network subsystem modules must inherit this class. In the current implementation, programmers can write network protocols over IP such as UDP and TCP.

`void init()`

This is invoked to initialize the network subsystem. It is used to initialize global data for the network subsystem.

`int attach(Socket* so, long proto, Proc* p)`

This is invoked by *socket* and *accept* system calls. In this method, programmers should create a new protocol control block using *allocateInPcb()* for *so*.

`int detach(Socket* so, Proc* p)`

This is invoked by *close* system call. In this method, programmers should destroy the protocol control block returned by *getInPcb()* for *so* using the destructor for the protocol control block.

`int bind(Socket* so, SockAddr* nam, Proc* p)`

This is invoked by *bind* system call. The argument *nam* indicates the local address to be bound to a socket *so*. In this method, programmers should call *bind()* for the protocol control block returned by *getInPcb()* for *so*.

`int listen(Socket* so, Proc* p)`

This is invoked by *listen* system call. In this method, programmers should call *bind()* if the local port has not been allocated yet. Next, they should change the protocol state to a listen state.

`int connect(Socket* so, SockAddr* nam, Proc* p)`

This is invoked by *connect* system call. The argument *nam* indicates the address of a socket with which *so* is connected. In this method, programmers should call *bind()* if the local port has not been allocated yet. Next, they should call *connect* for the protocol control block returned by *getInPcb()* for *so*.

`int disconnect(Socket* so, Proc* p)`

This is invoked by *close* system call. In this method, programmers should close the socket *so* soon if the connection is not established. If `SO_LINGER` is set for the socket option and the linger interval is 0, they should close the socket after they send pending data. Otherwise, they should make the socket a disconnecting state using *setDisconnecting()*, flush the receive buffer of the socket, and then close the socket.

`int accept(Socket* so, SockAddr* nam, Proc* p)`

This is invoked for a newly created socket *so* by *accept* system call. In this method, programmers should set the address of the socket with which *so* has been connected to *nam* using *setPeerAddr()* for the protocol control block returned by *getInPcb()* for *so*.

`int shutdown(Socket* so, Proc* p)`

This is invoked by *shutdown* system call. In this method, programmers should make the socket *so* a state where the socket cannot send more packet, and close it.

`int recvd(Socket* so, long flags, Proc* p)`

This is invoked when a packet is received with `PR_WANTRCVD` set in the protocol. In this method, programmers can send an acknowledgment to the received packet.

`int send(Socket* so, MbufChain* m, SockAddr* nam, MbufChain* ctrl, Proc* p)`

This is invoked by *send*, *sendto*, and *sendmsg* system calls. The argument *m* is the data to be sent and *nam* indicates the address to which the data is sent. In connection-oriented protocols, *nam* is `NULL`. In this method, programmers should call *connect()* for the protocol control block returned by *getInPcb()* for *so* if *nam* is not `NULL`. Next, they should attach a protocol header to *m* and call *IP::output()* to pass the packet to a lower layer.

`int recvOOB(Socket* so, MbufChain* m, long flags, Proc* p)`

This is invoked to read out-of-band data present on the socket *so*. In this method, programmers should copy the data to *m*.

`int sendOOB(Socket* so, MbufChain* m, SockAddr* nam, MbufChain* ctrl, Proc* p)`

This is invoked to send the out-of-band data *m* by *send*, *sendto*, and *sendmsg* system calls. Currently, this method is supported only by connection-oriented protocols. In this method, programmers should attach a protocol header to *m* and call *IP::output()*.

`int sockAddr(Socket* so, SockAddr* nam, Proc* p)`

This is invoked to the local address of the socket *so*. In this method, programmers should set the address to *nam* using *setSockAddr* for the protocol control block returned by *getInPcb()* for *so*.

`int peerAddr(Socket* so, SockAddr* nam, Proc* p)`

This is invoked to the address with which the socket *so* is connected. In this method, programmers should set the address to *nam* using *setPeerAddr* for the protocol control block returned by *getInPcb()* for *so*.

`int abort(Socket* so, Proc* p)`

This is invoked to abort sending packets. In this method, programmers should drop pending packets.

`int control(Socket* so, u_long cmd, caddr_t data, Proc* p)`

This is invoked to process protocol specific ioctl by *ioctl* and *fcntl* system calls. The arguments *cmd* and *data* are passed by system calls. In this method, programmers can call *Ip::control()*.

`int sense(Socket* so, struct stat* ub, Proc* p)`

This is invoked by *fstat* system call. In this method, programmers should set the socket status to *ub*.

`int connect2(Socket* so, Socket* so2, Proc* p)`

This is invoked by *socketpair* system call. In this function, programmers should connect *so* and *so2* if necessary.

`int input(MbufChain* m, int hlen)`

This is invoked when the IP layer receives a packet. The argument *m* is the packet with an IP header, IP options, and a header and options of this protocol. *hlen* indicates the length of the IP header and IP options. In this method, programmers should trim all headers and options and chain it to the receive buffer of the socket. The socket is returned by *getSocket()* for the protocol control block which is looked up by *InPcb::lookup()* from the IP addresses and ports of the source and the destination.

`void* ctlinput(int cmd, SockAddr* sa, void* data)`

This is invoked to process control information from lower layers. The argument *cmd* is the command, for example, `PRC_IFDOWN` for interface transition or `PRC_REDIRECT_HOST` for host routing redirect. *sa* is the address in the lower layer.

`int ctloutput(int op, Socket* so, int level, int optname, MbufChain*& mp)`

This is invoked to process control information by *setsockopt* and *getsockopt* system calls. The argument *level* indicates the layer to process control information. *op* is `PRCO_SETOPT` to set an option or `PRCO_GETOPT` to get an option. *optname* indicates the name of the option and *mp* indicates the value of the option to be set. In this method, programmers should process control information if *level* indicates this protocol, e.g. `IPPROTO_TCP` for TCP. They should set the value contained by *mp* to the option if *op* is `PRCO_SETOPT`, whereas they should set *mp* to the value of the option. If *level* does not indicate this protocol, they should call *Ip::ctloutput()*.

`void drain()`

This is invoked when memory is in short supply. In this method, programmers should free memory as much as possible.

```
void sysctl(int* name, u_int namelen, void* oldp, size_t* oldlenp, void* newp,
            size_t newlen)
```

This is invoked by *__sysctl* system call.

`void sysFastTimeout()`

This is invoked every 200 miliseconds.

`void sysSlowTimeout()`

This is invoked every 500 miliseconds.

`void soUpcall(Socket* so)`

This is invoked when the socket *so* is woken up for read by *input()* and so on. In this method, programmers can read packets without blocking by *receive()* for *so*.

A.2 API for Manipulating the Kernel Data

We describe an API for manipulating the kernel data in the current implementation. Basically, we explain all methods, but the description of some methods are omitted. Also, the below is an API necessary for developing sample modules, and is not complete.

A.2.1 Common

The following classes are commonly used in every subsystem module.

A.2.1.1 System class

This is the class for the whole system.

`struct timeval getCurrentTime()`

This is a static method. This returns the current time in the timeval format.

`struct timespec getCurrentTimeSpec()`

This is a static method. This returns the current time in the timespec format.

`int sleep(caddr_t id, int priority, const char* msg)`

This is a static method. This makes the current thread of the extension module sleep with the priority *priority* until it is woken up on *id* by *wakeup()*. *msg* is used for debugging.

`void wakeup(caddr_t id)`

This is a static method. This wakes up the thread of the extension module made slept by *sleep()* on *id*.

`void timeout(void (*ftn)(void *), void* args, int msec)`

This is a static method. This creates a new thread that executes the function *ftn* with the argument *args* and makes it sleep for *msec* miliseconds.

A.2.1.2 Interrupt class

This is the class for handling the level of interrupts.

`void disableSoftNet()`

This disables interrupts from protocol stacks.

`void enableAll()`

This enables interrupts by disabling function like *disableSoftNet()*.

A.2.1.3 Proc class

This is the class for handling a process. This class is currently used only for an identifier and has no methods.

A.2.1.4 String class

This is the class for handling strings.

methods: `getChar`, `substring`, `length`, `compare`

A.2.1.5 PList class

This is the class for handling a pointer list.

methods: `head`, `tail`, `size`, `insertHead`, `insertTail`, `remove`, `removeHead`, `removeTail`

This class has an internal class *Iterator*. It is used to traverse the list.

methods: `head`, `tail`, `prev`, `next`, `current`, `insertBefore`, `insertAfter`, `remove`

A.2.1.6 List class

This is the class for a special list. Programmers cannot directly use or inherit this class.

methods: `head`, `hasNode`, `insertHead`, `remove`

This class has an internal class *Iterator*. The methods are the same with *PList* class.

A.2.1.7 TailQueue class

This is the class for a special tail queue. Programmers cannot directly use or inherit this class. This class has the same methods with *List* class, but the internal algorithm is different.

A.2.1.8 CircleQueue class

This is the class for a special circular queue. Programmers cannot directly use or inherit this class. This class has the same methods with *List* class, but the internal algorithm is different.

A.2.2 File System

The following classes are used in file system modules.

A.2.2.1 Mount class

This is the class for handling the mount structure per file system.

`void setPrivateData(void* data) const`

This sets the file system specific data to *data*.

`void setFlags(int flag)`

This sets the bits of *flag* in the flag. The type of the flag is `MNT_RDONLY` for a read-only file system, `MNT_LOCAL` for a local file system, and so on.

`VnodeList* getVnodeList() const`

This returns the list of vnodes that the file system has.

`void getNewFsid(const String& name)`

This generates a new identifier of the file system from *name* and sets it to the structure for statistics information of the file system.

`Vnode* getNewVnode()`

This creates a new vnode and return it. The vnode is inserted in the vnode list. To create a new vnode, programmers cannot use the constructor of *Vnode* directly.

`int vflush(Vnode* skipvp, int flags)`

This removes all vnodes in the vnode list except *skipvp*. If `FORCECLOSE` is set in *flags*, the vnode is written back. For *skipvp*, the root vnode of the file system is often specified.

`Mount* getVfs(fsid_t* fsid)`

This is a static method. This returns the mount structure of the file system identified by *fsid*.

others: `getPrivateData`, `freePrivateData`, `getStatfs`, `clearFlags`, `checkFlags`

A.2.2.2 Statfs class

This is the class for handling the statistics information of the file system.

`void setName(const String& name)`

This sets the name of the file system to *name*. The length of the name must be less than 16.

`void setPath(const String& path)`

This sets the directory on which the file system is mounted to *path*. The length of the pathname must be less than 90.

`void setFsName(const String& fsname)`

This sets the file system specific name to *fsname*. For example, *fsname* is a hostname from which the file system is mounted for NFS.

`void setFlags(short flags)`

This sets the bits of *flags* in the flag. The type of the flag is the same with one of Mount class.

`void setBlockSize(long size)`

This sets the data block size to *size*.

`void setIOSize(long size)`

This sets I/O block size to *size*. This size is often the same with the data block size.

`void setBlockNum(long size)`

This sets the number of total data blocks to *size*.

`void setFreeBlockNum(long size)`

This sets the number of free data blocks to *size*.

`void setAvailBlockNum(long size)`

This sets the number of available data blocks to *size*.

`void setNodeNum(long size)`

This sets the number of total file nodes to *size*.

`void setFreeNodeNum(long size)`

This sets the number of free file nodes to *size*.

`long getFsid() const`

This returns a part of the identifier for the file system.

`long getFullFsid() const`

This returns the full identifier for the file system.

others: `getName`, `getPath`, `getFsName`, `getFlags`, `changeFlags`, `getBlockSize`, `getIOSize`, `getBlockNum`, `getFreeBlockNum`, `getAvailBlockNum`, `getNodeNum`, `getFreeNodeNum`

A.2.2.3 Vnode class

This is the class for handling the file node.

`void setPrivateData(void* data)`

This sets the file system specific data for this vnode to *data*.

`void setType(enum vtype type)`

This sets the type of this vnode. For example, `VREG` is for a regular file, `VDIR` is for a directory, and so on.

`void setTag(enum vtagtype type)`

This sets the tag type of this vnode to *type*. The tag type stands for the type of file systems.

`void setFlags(u_long flag)`

This sets the bits of *flag* in the flag. The type of the flag is `VROOT` for the root vnode, and so on.

`int getUseCount() const`

This returns the reference count for the number of users that are open for reading and/or writing. Use *vref()*, *vput()*, and *vrele()* if you want to increment or decrement this count.

`int getNumOutput()`

This returns the number of writes in progress.

`int getWriteCount() const`

This returns the reference count for the number of writers which reference the vnode.

`int getHoldCount() const`

This return the reference count for the number of pages and buffers which are associated with the vnode.

`void lock()`

This sets the lock flag.

`void wait(int priority = -1, const char* msg = "")`

This makes the current thread sleep until *wakeup()* is called for the vnode. *priority* and *msg* are passed to *System::sleep()*.

`void wakeup()`

This wakes up the thread made sleep by *wait()* for the vnode.

`void vref()`

This checks that the reference count of users is positive and then increments it.

`void vput()`

This calls *FileSystem::unlock()* and then calls *vrele()*.

`void vrele()`

This decrements the reference count of users. If the count gets 0, it calls *FileSystem::inactive()*.

`void vhold()`

This increments the reference count of pages and buffers.

`void holdRele()`

This checks that the reference count of pages and buffers is positive and then decrements it.

`int vget(int lockflag)`

This increments the reference count of users and calls *FileSystem::lock()* if *lockflag* is 1.

`int vinvalBuf(int flags, Ucred* cred, Proc* p)`

This flushes out and invalidate all buffers associated with the vnode. If `V_SAVE` is set, this calls *FileSystem::fsync()* and calls *FileSystem::bwrite()* for buffers with the flag `B_DELWRI`.

`void vclean(int flags)`

This disassociates the file system from the vnode. If `DOCLOSE` is set in *flags*, this calls *vinvalBuf()* and *FileSystem::close()*. This calls *FileSystem::inactivate()* if necessary, and then calls *FileSystem::reclaim()*.

`void vgone()`

This calls *vclean* with the argument of `DOCLOSE`.

`void cachePurge()`

This flushes the cache for the vnode.

`void vwakeup(Buf* bp)`

This updates outstanding I/O count and does wakeup if requested.

```
int vaccess(mode_t file_mode, uid_t uid, gid_t gid, mode_t acc_mode, Ucred* cred)
            const
```

This checks for the access right of the vnode.

`BufList* getCleanList() const`

This returns the list of clean buffers associated with the vnode.

`BufList* getDirtyList() const`

This returns the list of dirty buffers associated with the vnode.

`int lookup(Nameidata* ndp)`

This looks up the pathname. Programmers should pass *ndp* whose vnode for a starting directory and component name are set.

others: `getPrivateData`, `freePrivateData`, `getType`, `getTag`, `clearFlags`, `checkFlags`, `getMount`, `getDevice`, `unlock`, `isLocked`, `isWaiting`, `incrUseCount`, `decrUseCount`, `incrNumOutput`, `decrNumOutput`, `decrHoldCount`, `destroy`

A.2.2.4 VnodeList class

This is the class for a list of vnodes. The methods are the same with *List* class.

A.2.2.5 Vattr class

This is the class for an attribute of a vnode.

`void setUid(uid_t uid)`

This sets the owner user ID to *uid*.

`void setGid(gid_t gid)`

This sets the owner group ID to *gid*.

`void setBytes(u_quad_t bytes)`

This sets the size of disk space held by a file to *bytes*.

`void setBlockSize(long size)`

This sets the block size perferred for I/O to *size*.

`void setGeneration(u_long gen)`

This sets the generation number of a file to *gen*.

`void setFlags(u_long flags)`

This sets the bits of *flags* in the flag. The type of the flag is `APPEND`, `IMMUTABLE`, and so on.

`void setType(enum vtype type)`

This sets the vnode type to *type*.

`void setMode(u_short mode)`

This sets the file access mode to *mode*. The mode is `VREAD`, `VWRITE`, and/or `VEXEC`.

`void setNlink(short nlink)`

This sets the number of references to a file to *nlink*.

others: `setUid`, `getUid`, `setGid`, `getGid`, `getBytes`, `getBlockSize`, `setGeneration`, `changeFlags`, `setFsid`, `setDevice`, `getDevice`, `getMode`, `getNLink`, `setFileid`, `getFileid`, `setSize`, `getSize`, `setAccessTime`, `getAccessTime`, `setChangeTime`, `getChangeTime`, `setModifiedTime`, `getModifiedTime`, `create`, `destroy`

A.2.2.6 Nameidata class

This is the class for pathname lookup.

`void setPathLength(int len)`

This sets the length of a pathname to *len*.

`void setStartDir(Vnode* vp)`

This sets the vnode for a starting directory.

`Vnode* getVnode()`

This returns the vnode of the result of pathname lookup.

`Vnode* getParentVnode()`

This returns the vnode of a directory where the result vnode is.

others: `getCompName`, `getStartDir`, `create`, `destroy`

A.2.2.7 CompName class

This is the class for a component name used in pathname lookup.

`const String& getName() const`

This returns the pathname to be looked up.

`void setFlags(u_long flags)`

This sets the bits of *flags* in the flag. The type of the flag is `LOCKLEAF` for locking a vnode, `LOCKPARENT` for locking aparent vnode.

`void setNameiOp(u_long op)`

This sets the operation for which the pathname lookup is done. The operation is `LOOKUP`, `CREATE`, `DELETE`, or `RENAME`.

`void setPathName(const String& path)`

This sets the pathname to *path*.

others: `getNameLen`, `checkFlags`, `changeFlags`, `setUcred`, `getUcred`, `setProc`, `getProc`, `freePathname`

A.2.2.8 Buf class

This is the class for handling the file buffer.

`void lock()`

This sets the lock flag.

`void setBusy()`

This sets the busy flag.

`void wait(int priority = -1, const char* msg = "")`

This makes the current thread sleep until *wakeup()* is called for this buffer. *priority* and *msg* are passed to *System::sleep()*.

`void wakeup()`

This wakes up the thread made sleep by *wait()* for this buffer.

`void setFlags(long flag)`

This sets the bits of *flag* in the flag. The type of the flag is `B_ASYNC` for asynchronous I/O, `B_DELWRI` for delay I/O, `B_READ` for reading a buffer, and so on.

`void setReadUcred(Ucred* cred)`

This sets the credentials for read to *cred*.

`void setResident(long resid)`

This sets the size of data remaining in this buffer.

`void setDirtyOffset(int off)`

This sets the offset of dirty region in this buffer.

`void setDirtyEnd(int end)`

This sets the end of dirty region in this buffer.

`void setValidOffset(int off)`

This sets the offset of valid region in this buffer.

`void setValidEnd(int end)`

This sets the end of valid region in this buffer.

`long getSize() const`

This returns the size of this buffer.

`void setError(int error)`

This sets the error for this buffer.

`void bremFree()`

This removes this buffer from the free list.

`void brelse()`

This release this buffer on to the free list.

`void brelVp()`

This disassociates this buffer from a vnode.

`void bgetVp(Vnode* vp)`

This associates this buffer with the vnode *vp*.

`int bwrite()`

This writes out the contents of this buffer.

`int bioWait()`

This waits for operations on this buffer to complete. This method returns an error if it fails the operations.

`void bioDone()`

This marks I/O complete on this buffer. If the operations are not asynchronous, it wakes up a thread waiting for this buffer.

`int bread(Ucred* cred, int async = 0)`

This reads a disk block to this buffer.

`void clrBuf()`

This clears the data area of this buffer by zero.

`void copyIn(caddr_t addr, int size = -1, off_t offset = 0)`

This copies memory of *addr* to this buffer with the offset *offset* by *size*. If *size* is -1, the size of copy is the buffer size.

`void copyOut(caddr_t addr, int size = -1, off_t offset = 0)`

This copies the data of this buffer with the offset *offset* to memory of *addr* by *size*. If *size* is -1, the size of copy is the buffer size.

`caddr_t getData() const`

This returns the pointer to the data of this buffer.

`Buf* getBlock(Vnode* vp, daddr_t lblkno, int size)`

This is a static method. This returns a new buffer with the size of *size*. If a buffer associated with the vnode *vp* and the logical block number *lblkno* exists, this returns it.

others: `unlock`, `isLocked`, `clearBusy`, `isBusy`, `isWaiting`, `setPhysicalBlockNum`, `getPhysicalBlockNum`, `setLogicalBlockNum`, `getLogicalBlockNum`, `setVnode`, `getVnode`, `getReadUcred`, `setWriteUcred`, `getWriteUcred`, `getResident`, `setProc`, `setDevice`, `getDirtyOffset`, `getDirtyEnd`, `getValidOffset`, `getValidEnd`

A.2.2.9 BufList class

This is the class for a list of buffers. The methods are the same with *List* class.

A.2.2.10 BufQueue class

This is the class for a queue of buffers. The methods are the same with *TailQueue* class.

A.2.2.11 Uio class

This is the class for handling a universal I/O buffer.

```
int read(Vnode* vp, daddr_t lblkno, int xfersize, off_t offset, int blksize,
        Ucred* cred)
```

This calls *FileSystem::strategy* and then copies the contents of the file buffer read to this buffer. The arguments *vp* and *lblkno* indicate the file and the logical block number to be read, respectively. The file is read from *offset* by *xfersize*. *blksize* is the size of a file buffer read at a time.

```
int bulkRead(Vnode* vp, int blksize, int filesize, Ucred* cred)
```

This reads the contents of a file to this buffer by *filesize*, repeating to call *read()*.

```
int write(Vnode* vp, daddr_t lblkno, int xfersize, off_t offset, int blksize,
         int filesize, Ucred* cred)
```

This fills the contents of this buffer to a file buffer and then calls *FileSystem::strategy* to write out the buffer to a file.

```
int bulkWrite(Vnode* vp, int blksize, int filesize, Ucred* cred)
```

This writes out the contents of this buffer to a file by *filesize*, repeating to call *write()*.

```
void setOffset(off_t off)
```

This sets the offset at which the operation should start.

```
void setResident(int resid)
```

This sets the size of data remaining in this buffer.

`void setIovec(struct iovec* iov, int iovcnt = 1)`

This sets the I/O vector array to this buffer. The size of array is *iovcnt*.

`int copyIn(caddr_t cp, int size)`

This fills this buffer with the contents of *cp* by *size*.

`int copyOut(caddr_t cp, int size)`

This copies the contents of this buffer to *cp* by *size*.

others: `getOffset`, `getResident`, `setProc`, `getProc`, `getIovec`, `getIovCount`

A.2.2.12 DirEntry class

This is the class for a directory entry.

`DirEntry(u_int32_t fileno, u_int16_t reclen, u_int8_t type, const String& name)`

This is a constructor of *DirEntry* class.

`int pack(Uio* uio) const`

This copies the directory entry to *uio*.

`u_int16_t length() const`

This returns the length of this directory entry.

`u_int8_t getType() const`

This returns the file type. For example, `DT_REG` is for a regular file.

`const char* getName() const`

This returns the file name.

others: `getFileNo`, `getNameLen`

A.2.2.13 Ucred class

This is the class for credentials.

`void setGroupMembers(gid_t* groups, int ngroups)`

This sets the members of the group to the array *groups* with the size of *ngroups*.

`int getGroupNum() const`

This returns the number of members of the group.

`Bool isGroupMember(gid_t gid) const`

This returns True if *gid* is a member of the group.

`void crHold()`

This increments the reference count.

others: `setUid`, `getUid`, `setGid`, `getGid`, `getGroupMembers`

A.2.3 Network Subsystem

The following classes are used in network subsystem modules.

A.2.3.1 MbufChain class

This is the class for handling a mbuf chain.

`int length() const`

This returns the length of this mbuf chain.

`int remain() const`

This returns the remaining size for dissect.

`void resetDissect()`

This resets the counters for dissect.

`int build(caddr_t cp, int size)`

This fills this mbuf chain with the data *cp* with the size of *size*.

`int build(u_int32_t tl)`

This fills this mbuf chain with one word value *tl*.

`int build(MbufChain* m, int size)`

This fills this mbuf chain with the specified mbuf chain *m*. The data is copied from the dissect point of *m* by *size*.

`int build(SockBuf* sb, int off, int size)`

This fills this mbuf chain with the data of the socket buffer *sb*. The data is copied from the offset *off* by *size*.

`int buildAsBytes(caddr_t* cp, int size)`

This reserves the space for data of *size* in this mbuf chain and returns a pointer to the data to **cp*. The maximum size is limited to 2,048.

`int buildAsWord(u_int32_t** tl)`

This reserves the space for data of one word in this mbuf chain and returns a pointer to the data to **tl*.

`int buildAsIovec(struct iovec** iov, int size)`

This reserves the space for I/O vector array of *size* in this mbuf chain and returns a pointer to the array to **iov*.

`int padding(int alignment)`

This pads this mbuf chain with alignment.

`int append(MbufChain* mc)`

This appends the mbuf chain *mc* to the end of this mbuf chain.

`int prepend(MbufChain* mc)`

This appends this mbuf chain to the end of the mbuf chain *mc*.

`int dissect(caddr_t cp, int size)`

This copies the data in this mbuf chain from the dissect point by *size*. The dissect point is advanced by *size*.

`int dissect(u_int32_t* tl)`

This copies the data in this mbuf chain from the dissect point to *tl*. The dissect point is advanced by one word.

`int dissectAsPointer(caddr_t* cp, int size)`

This sets the dissect point to **cp*, and advances the dissect point by *size*. The maximum of *size* is 2,048.

`int advance(int size)`

This advances the dissect point by *size*.

`int trimHead(int size)`

This trims the data from the head of this mbuf chain by *size*.

`int trimTail(int size)`

This trims the data from the tail of this mbuf chain by *size*.

`int checkSum(int len)`

This calculates checksum for the front of this mbuf chain by *len*.

`MbufChain* create(int type = MT_DATA)`

This is a static method. This creates a new mbuf chain with the type of *type*. If creating a packet header, users must pass `MT_HEADER` as *type*.

others: `destroy`

A.2.3.2 InPcb class

This is the class for a internet protocol control block.

`void setLocalAddr(struct in_addr addr)`

This sets the local address of the socket with this protocol control block to *addr*.

`void setForeignAddr(struct in_addr addr)`

This sets the address of a socket connected with the socket with this protocol control block to *addr*.

`void setLocalPort(short port)`

This sets the local port of the socket with this protocol control block to *port*.

`void setForeignPort(short port)`

This sets the port of a socket connected with the socket with this protocol contorol block to *port*.

`void setPrivatePcb(caddr_t ppcb)`

This sets the protocol specific protocol control block to *ppcb*.

`void setState(int state)`

This sets the state of this protocol control block to *state*. The state is `INP_ATTACHED`, `INP_BOUND`, or `INP_CONNECTED`.

`void setOptions(MbufChain* m)`

This sets the IP options to *m*.

`void setSockAddr(SockAddr* nam)`

This fills *nam* with the address and port of the socket with this protocol control block.

`void setPeerAddr(SockAddr* nam)`

This fills *nam* with the address and port of a socket connected with the socket with this protocol control block.

`RtEntry* getRtEntry()`

This returns the routing table entry. If the entry does not created, it creates a new routing table entry.

`int bind(SockAddr* nam, Proc* p)`

This binds *nam* to the socket with this protocol control block.

`int connect(SockAddr* nam)`

This connects the socket with this protocol control block with a socket specified by *nam*.

`void disconnect()`

This disconnects the socket with this protocol control block from the connected socket.

`InPcbQueue* getInPcbQueue()`

This is a static method. This returns the queue of protocol contorol blocks of the network subsystem.

`InPcb* lookup(struct in_addr faddr, u_int16_t fport, struct in_addr laddr, u_int16_t lport)`

This is a static method. This looks up a protocol control block using a foreign address *faddr*, a foreign port *fport*, a local address *laddr*, and a local port *lport*. The search is done in the order of (1)a lookup by connection, (2)a lookup by binding.

others: `getLocalAddr`, `getForeignAddr`, `getLocalPort`, `getForeignPort`, `getSocket`, `getIp`, `getRoute`, `getOptions`

A.2.3.3 Ip class

This is the class for handling IP.

`void setSrcAddr(struct in_addr src)`

This sets the source IP address to *src*.

`void setDstAddr(struct in_addr dst)`

This sets the destination IP address to *dst*.

`void setTtl(u_int8_t ttl)`

This sets the value of time to live to *ttl*.

`u_int8_t getTos() const`

This gets the type of service, i.e. the protocol number.

`int output(MbufChain* m, MbufChain* opt, Route* ro, int flags, MbufChain* mopt)`

This is a static method. This attaches an IP header and the IP options *opt* to the packet *m* and passes it to a lower network layer, e.g. ethernet device driver.

`int ctloutput(int op, Socket* so, int level, int optname, MbufChain*& mp)`

This is a static method. This processes a socket option for IP if *level* is `IPPROTO_IP`. The argument *op* is `PRCO_SETOPT` or `PRCO_GETOPT`, *optname* is `IP_TTL`, `IP_TOS`, and so on.

`void stripOptions(MbufChain* m)`

This is a static method. This strips out IP options from the packet *m*.

`u_int getOptLen(InPcb* inp)`

This is a static method. This returns the maximum length of IP options that *inp* holds.

`int control(Socket* so, u_long cmd, caddr_t data, IfNet* ifp, Proc* p)`

This is a static method. This processes generic internet control operations. The arguments *cmd* and *data* are the same with those of *ioctl* system call.

others: `getSrcAddr`, `getDstAddr`, `getTtl`

A.2.3.4 Socket class

This is the class for handling a socket.

`void setOptions(short options)`

This sets the bits of *options* in the socket option. The type of the option is `SO_DONTROUTE`, `SO_REUSEADDR`, and so on.

`void setState(short state)`

This sets the bits of *state* in the socket state. The type of the state is `SS_NBIO` for non-blocking operations, `SS_ASYNC` for asynchronous I/O, and so on.

`Bool hasQueueingSpace() const`

This returns True if there is a space to queue a new connection. The limit is determined by *listen* system call.

`InPcb* allocateInPcb()`

This allocates a new protocol control block.

`SockBuf* getSndBuf() const`

This returns the send buffer.

`SockBuf* getRcvBuf() const`

This returns the receive buffer.

`Socket* getNewConn(int connstatus)`

This creates a new socket with the state of *connstatus*.

`void setError(u_short errno)`

This sets the error of socket operations to *errno*.

`int reserve(u_long sndcc, u_long rcvcc)`

This reserves buffer spaces for send and receive to *sndcc* and *rcvcc*, respectively. It must be called before using this socket.

`int abort()`

This aborts sending packets.

`void setConnecting()`

This makes the socket state connecting.

`void cantSendMore()`

This makes this socket not send any more packets.

`void wakeupRead()`

This wakes up a thread waiting for socket read.

`int connect(SockAddr* addr)`

This connects this socket with a socket specified by *addr*.

`int shutdown(int how)`

This shuts down part of a full-duplex connection. The argument *how* is the same with one passed to *shutdown* system call.

`int close()`

This disconnects if this socket is connected and then close this socket.

`int send(SockAddr* addr, MbufChain* m, int flags)`

This sends a packet *m* to the address *addr*. If *addr* is `NULL` and this socket is not connected, it returns an error.

`int receive(SockAddr** paddr, MbufChain** m, int* flagsp)`

This reads a pakcet from the receive buffer and the pointer is set in *m*.

`void setUpcall(NetworkSystem* ns)`

This sets up so that *NetworkSystem::soUpcall()* for *ns* is called by a socket upcall. The socket upcall is done when this socket is woken up for read.

`Socket* create(int dom, int type, int proto)`

This is a static method. This creates a new socket with the arguments of *socket* system call and returns it.

`Socket* create(int fd)`

This is a static method. This returns a socket corresponding to *fd*.

others: `getOptions`, `checkOptions`, `clearState`, `checkState`, `getProtoSw`, `setConnected`, `setDisconnecting`, `setDiconnected`, `cantRecvMore`, `wakeupWrite`, `clearUpcall`, `destroy`

A.2.3.5 SockBuf class

This is the class for handling a socket buffer.

`void setHighWatermark(u_long hiwat)`

This sets the maximum buffer size to *hiwat*.

`void setLowWatermark(u_long lowat)`

This sets the low watermark to *lowat*.

`int size() const`

This returns the size of data in this buffer.

`int getSpace() const`

This returns the size of space in this buffer.

`MbufChain* getMbufChain()`

This returns data in this buffer as a mbuf chain.

`int reserve(u_long cc)`

This checks for the value of *cc* and sets the maximum buffer size to it if acceptable.

`void drop(int len)`

This drops data from the front of this buffer by the size of *len*.

`Bool needNotify()`

This returns True if I/O is possible. It is used to to notify the other sockets.

`Bool appendAddr(SockAddr* asa, MbufChain* m0, MbufChain* ctrl)`

This appends the address *asa*, the data *m0*, and control data *ctrl* to this buffer. It is fundamentally used for a receive buffer.

`void append(MbufChain* m)`

This appends the data *m* to this buffer. It is used for data-stream protocol.

others: `getHighWatermark`, `getLowWatermark`

A.2.3.6 SockAddr class

This is the class for an socket address.

`int length() const`

This returns the length of this socket address.

`void setAddr(in_addr addr)`

This sets the IP address to *addr*.

`void setPort(u_int16_t port)`

This sets the port to *port*.

`Bool isNullHost() const`

This returns True if this socket address points to null address.

others: `getAddr`, `getPort`, `isMulticast`, `isBroadcast`

A.2.3.7 IfAddr class

This is the class for an interface address of internet.

`struct in_addr getAddr() const`

This returns the internet address.

`struct in_addr getBroadAddr() const`

This returns the broadcast address.

others: `getIfNet`, `getSockAddr`, `create`, `destroy`

A.2.3.8 IfNet class

This is the class for a network interface.

`Bool checkFlags(short flags) const`

This returns True if the bit of *flags* is set in the flag.

`u_long getMtu() const`

This returns the value of maximum transmission unit (MTU).

A.2.3.9 Route class

This is the class for a route.

`SockAddr* getDstAddr() const`

This returns the destination address.

`void allocateRtEntry()`

This allocates a new routing table entry.

others: `setRtEntry`, `getRtEntry`, `freeRtEntry`

A.2.3.10 RtEntry class

This is the class for a routing table entry.

`Bool checkFlags(short flags) const`

This returns True if the bits of *flags* are set in the flag. The type of the flag is `RTF_UP` for an usable route, `RTF_HOST` for a host entry, and so on.

`SockAddr* getGwAddr() const`

This returns the gateway address.

`SockAddr* getDstAddr() const`

This returns the destination address.

`SockAddr* getNetmask() const`

This returns the network mask.

others: `getFlags`, `getIfNet`, `getIfAddr`, `getRtMetrics`

A.2.3.11 RtMetrics class

This is the class for route metrics.

`u_long getMtu()`

This returns the maximum packet size called maximum transmission unit (MTU).

`u_long getSendPipe()`

This returns the outbound delay-bandwidth product.

`u_long getRecvPipe()`

This returns the inbound delay-bandwidth product.

`void setThreshold(u_long ssthresh) const`

This returns the outbound gateway buffer limitation.

`void setRtt(u_long rtt)`

This sets the round trip time to *rtt*.

`void setRttVar(u_long rttvar)`

This sets the variance of the round trip time to *rttvar*.

`Bool checkLocks(u_long locks) const`

This returns True if the bits of *locks* are set in the lock flag. The type of the flag is `RTV_MTU`, `RTV_RPIPE`, `RTV_RTT`, and so on. When these lock flags are set, programmers cannot access the corresponding metrics.

others: `getThreshold`, `getRtt`, `getRttVar`

A.2.3.12 ProtoSw class

This is the class for a protocol switch.

`Bool checkFlags(short flags) const`

This returns True if the bits of *flags* are set in the flag. The type of the flag is `PR_ATOMIC` for exchanging atomic message, `PR_CONNREQUIRED` for connection required by protocol, and so on.