DTM: A New Mechanism for Extensible Operating Systems

Shu Liu，Xuetao Guan, Xu Cheng
{liushu, guanxuetao, chengxu}@mprc.pku.edu.cn

ABSTRACT

Traditional operating system designers define a fixed set of flat interfaces which provide the same set of services to all applications. This approach, however, is now problematic and showing its limitations in the face of diverse applications which demand extensibility and configurability. In this paper, we describe the design and implementation of a new interface mechanism called DTM (Dynamic Trunk Mechanism) in which related services are treated as a *trunk*. The trunk can be registered dynamically to the system, and be requested, used and released by the applications that have privilege to it. System designers can configure the system according to application requirements by registering a set of necessary trunks when system starts. It requires only trivial operating system changes and enables a large class of extensions or flexible configurations to existing operating systems while retaining full compatibility. We have implemented DTM based on Linux 2.4.17 and a number of useful trunks have been extended. Performance measurements of DTM demonstrate that even the mechanism is more flexible than traditional system calls, it introduces little of overhead.

Keywords: Dynamic Trunk Mechanism, extensibility, configurability, operating system

1. Introduction

General purpose systems such as Linux are increasingly being used for a number of diverse applications, especially various embedded environments. However, services provided by the system are often ill-suited to the specific needs of sophisticated applications [8, 26, 29]. In response to this problem, researchers have studied various methods by which operating system kernels can be extended with application-specific functionality [16, 19, 22, 30, 32] and services can be tailored and customized for specific purpose [9, 11, 12].

Undoubtedly, the interfaces exposed by operating systems to provide certain services to programs and to the users of those programs play an important role in operating system extension. There are at least three conflicting challenges to system interfaces.

First, the interface mechanism must have good support for extension and configuration of system services. The specific services required by applications, of course, differ from one to another. So, multiple extensions or configuration are needed to support specific applications domain [3, 26]. Whether the extension or configuration can be implemented easily depends on the interface mechanism.

Second, the interface should be helpful to ensure the security of extension services [4, 23]. If multiple application-specific functions are extended to system, it must be guaranteed that the extensions are protected from malicious applications and enforced upon uncooperative applications.

Third, the most important one, extensibility and configurability of operating systems should be achieved with few or no modification to existing operating systems or applications. And it should introduce little of overhead.

Traditional operating system designers define a fixed set of flat interfaces which provide the same set of services to all applications [6, 19] and allow extensions to a certain extent. Although this paradigm has worked well for decades, it is now coming under increasing pressure from the varied system extension requirements of a growing diversity of applications areas and hardware platforms [35]. For example, if kernel services for multimedia applications to utilize the hardware decoder and for web server to support its own buffer cache algorithm will be extended in Linux, system designers usually tail and modify the kernel functions manually. Although specific extension can be simply implemented by extending system call table, it is not a viable option when multiple extensions are required. The modifying of operating systems is complex and difficult and would cause serious compatibility problems [21]. In addition, the flat interfaces provide applications with more services than they need, resulting in larger kernel footprints which are generally detrimental to performance. The traditional interface also grants privilege with coarse granularity. So, many applications acquire more privileges than they require [23]. Specific extension can not be protected from malicious applications and enforced upon uncooperative applications.

In this paper, we propose a new interface mechanism named DTM (Dynamic Trunk Mechanism) to support operating system extension which exhibits the characteristics mentioned above. In DTM, related services are treated as a *trunk*, the basic unit in operating system interface. Compared with traditional fixed flat interface, all the system services are divided into different trunks. a trunk can be registered dynamically to the system to avoid compatibility problems, and be requested, used and released by the applications that have privilege to access it. System designer can configure the system according to application requirements by registering a set of trunks when system starts. This mechanism has been implemented based on Linux2.4.17 and widely used on our platform. In our experience, DTM provides a flexible, efficient and safe operating system extension and configuration solution. We believe that the mechanism we have described in this paper can provide a substantial benefit to users of existing operating systems, enabling a viable developing operating system with extensibility and configurability.

The rest of this paper is organized as follows. Section 2 provides a brief overview of system extension and identifies related work. Section 3 describes the basic idea of DTM and Section 4 states the implementation in more detail. Evaluation is then presented in Section 5. Finally, conclusions are described in Section 6.

2. Related Work

Various solutions have been proposed to extend the operating system. Related work can be categorized by their underlying implementation.

Some researches on extensible operating systems focus on OS structure modifications [1, 4, 16, 17, 18]. For example, micro-kernels such as Mach [1] offer a few basic abstractions while moving the implementation of more complex services and policies into application-level components. These systems have either suffered from significant communication costs or lack of portability. In Exokernel [16], Lipto [14], and the Cache Kernel [7] etc, applications linked with library which provide the rich semantics expected of a full operating system by utilizing the low level kernel services. In addition, dynamically extensible systems such as SPIN [4] and VINO, allow application to download code into the system address space. Configurable operating systems are also designed to allow system administrators who select appropriate modules for the bootable instance off-line. Examples include Microkernel [24], uChoices [33], and DEIMOS [9]. All these projects improve extensibility and configurability to some extent. But the initial cost of replacing existing commodity operating systems is prohibitive. Consequently, these extensible or configurable operating system architectures will remain unavailable to the average users for the foreseeable future.

Alternatively, operating system extension can be carried out at the user level [5, 35], such works include process migration[13, 34], fast communication primitives [15], upcalls [10] and kernel events notifications [2，25] etc.. Most of them are concerned with the way to trigger services or handlers due to some conditions or events in the kernel. The user-level ones provide flexible techniques to support system extension with little modification to existing systems. However, it suffers from the overhead due to multilayer functionality [21] in order to adapt to application requirements.

Another related work is researches on traditional interface design [21, 23, 27]. The fixed and flat interface is problematic for diverse application areas. M. B. Jones [21] pointed out the flat interface can not utilize enhanced or application specific implementations. He constructed objects which correspond to the abstractions present in the original interface to provide different sets of objects and interfaces appropriate to different applications. M. Krohn etc. [23] investigated the security problem induced by traditional interface. The traditional interface grants privilege with coarse granularity. He proposed a new capability management mechanism for secure application construction.

Our approach, Dynamic Trunk Mechanism, aims to provide a mechanism of evolving existing operating systems in interface-level by enabling application-specific extension and configuration. We have implemented it based on Linux kernel by replacing traditional system call table and extending the kernel services for diverse applications on our platform. Experience and experimental evaluation show that the new interface mechanism is as efficient as traditional system call interface. In fact, the DTM places no specific requirements on the underlying OS structure. As a consequence, it is possible to be used as a replacing mechanism in existing operating systems.

3. DTM Overview

As stated above, the basic idea of DTM is to overcome the limitation of traditional interface in application-specific extension and configuration and security problems. We achieve this goal by introduce the *trunk* concept and trunk management mechanism. The key design issues for DTM are as follows.

Related system services are organized as a *trunk* in DTM, which is the basic unit to be extended or customized. The interface provided to applications is structured as a set of trunks. DTM allows:

- the custom trunks which contains the necessary services for target system, including functions for system running and extensions for specific applications to register to the kernel.
- providing different applications with appropriate sets of system services, not all the registered services. Applications should apply for the trunks it wants to use.

Compared with traditional interface, there are several notably differences. First, traditional interface consists of fixed functions to all applications, but DTM supports dynamically register of necessary trunks. So, designers can custom the interface flexibly for diverse application environments. The extensibility, compatibility and configurability of systems are easily achieved. Second, traditional interface provides general functions for all applications. Nevertheless, the trunks set for applications may be different in DTM, which can implement kernel service efficiently for diverse applications and grants privilege with fine-grained granularity. Take the Linux system as an example. The kernel functions are mapped to certain system call numbers in system call table in advance and are available for all applications. Designers may modify the system call table manually to customize services. However, the position sensitivity mechanism tampers portability and compatibility. Using DTM, the trunks registered to the system call table get the system call number dynamically with less manual action. In addition, the set of services are divided into different subsets corresponding to diverse applications, which gives the application the minimal set of services to perform its task. It is helpful for system security.

DTM can be implemented with less modification to existing systems. It consists of two main components, kernel-level trunk management mechanism to organize the trunks and user-level support for applications to call system service via trunks.

Kernel-level trunk management mechanism provides functions for trunk register and facilities for user to apply for, inquire and release trunk. All trunks used by applications should be registered to the system and preserved in kernel space. Application maintains a subset of registered trunks which it has access to in its process space. When a system function belonging to a certain trunk is called the first time by an application, the application must apply for the trunk to check whether it is privileged to this trunk. If the function call is privileged, then the user process copies the trunk to its own process space to use. Applications can inquire the trunks in the process space which it can use, and release the trunk if it will not be used any more. All the details will be explained in next section.

The interface to users is structured as a set of trunks in DTM. But usually, applications call system functions directly. So, it is necessary to keep the requisition of trunk and privilege checking is transparent to existing applications. Facility which consists of dynamic jump table described in 4.3 is provided in user-level to transform the functions call in application to exact service of a trunk.

The general situation and process of operating system using DTM can be depicted as Figure 1.

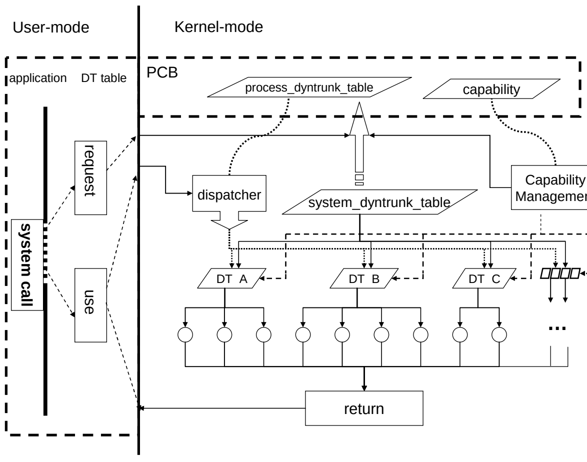


Figure 1. The system call process of DTM

4. Implementation Details

4.1 Basic data structure

There are three important data structures in DTM, named *system_dyntrunk_table*, *process_dyntrunk_table* and *dyntrunk*.

Dyntrunk is the denotation of *trunk*. *System_dyntruk_table* and *process_dyntrunk_table* are both an array of pointers which point to *dyntrunk*. *System_dyntrunk_table* records all the information of trunks registered to the system while *process_dyntrunk_table* knows only the trunks which the process has successfully applied for. The structure *dyntrunk* defines a trunk as follows:

```
struct dyntrunk{
        int dtid;
        int dtver;
        int dtno;
        int dtflags;
        int dtcap;
        int dtresearved;
        char[8] name;
};
```

The *dtid* and *dtver* entries are used to identify the trunk. Two trunks are considered to be the same if they have uniform *dtid* and *dtver*. When a trunk is registered to the system, *dtno*, like system call number is assigned and *dtflags* is updated. In our design, the upper 16 bits of *dtno* map to the index of the *dyntrunk_table* (both *sytem_dyntrunk_table* and *process_dyntrunk_table*) and the lower 16 bits indicate the number of functions that the trunk provides. Each bit of *dtflags* represents a flag, including trunk status (DTM_INVALID, DTM_RUNNING and DTM_DElETED), request conditions (DTM_CHECK_ID, DTM_CHECK_NAME, DTM_CHECK_VER, DTM_VER_EQ, DTM_VER_GE, DTM_VER_LE, and DTM_VER_NE) and substitute flags (DTM_ALTERNATED). The *dtcap* is capability of this trunk and the consecutive one is reserved for extension. The last entry defines the name of the trunk.

The relationship of *dyntrunk* and *dyntrunk_table* can be described as Figure 2. Each entry in *dyntrunk_table* contains the starting address of memory space allocated to registered trunk. The memory space stores the trunk structure, the number of functions in the trunk and entry to these functions in sequence.

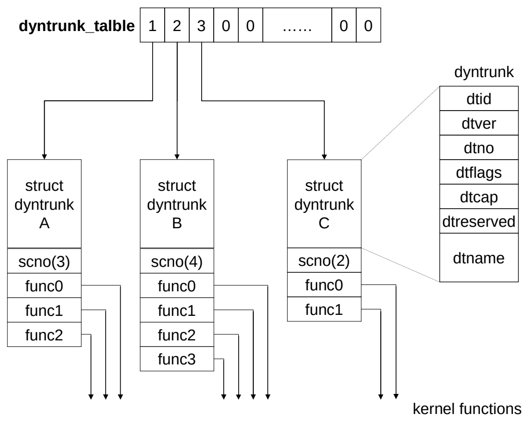


Figure 2. The relationship of dyntrunk_table and dyntrunk

There is only one *system_dyntrunk_table* in system, and each process has its own *process_dyntrunk_table* in its PCB. Child process has a copy of its parent process's *process_dyntrunk_table* so that it has access to the privileged trunks of its parent.

4.2 Dynamic Trunk Management

The dynamic trunk management components are implemented with responsibility for the init of Dynamic Trunk Mechanism, and provide functions and facilities for trunk register and use.

The init route sets up *system_dynrunk_table* and init it. Then, the Dynamic trunk mechanism itself, registers to the system as the first trunk to provide basic trunk related services (called DTM trunk). In our implementation, it is fixed in the first entry of *system_dyntrunk_table*. Also, the first entry of *init* process, the parent process of all processes, should be filled with the address of DTM trunk, so that the services are available to all processes.

Basic functions are implemented as the members of DTM trunk to provide operations on trunk, including *dyntrunk_register*, *dyntrunk_request*, *dyntrunk_release* and *dyntrunk_info*.

The function *dyntrunk_register* is called by init function of each trunk. The trunk passes its *dyntrunk* structure and function list to *dynrunk_register*. The *dtcap* of dyntrunk is obligatory. This function works according to following rules:

1. Check the trunk information. If the *dtid* and *dtver* is not valid, or *dtcap* has not been set, return a parameter error.
2. Traverse the *system_dyntrunk_table*. If the same trunk has been registered, return a repeated register error.

3. Handle the function list to get the number of functions in the trunk. If the list is too long, more than the maximum value defined by designer, an error will arise.

4. Allocate memory space for the *dyntrunk* structure and function list as described above. If failed, return an error.

5. Find an empty entry in *system_dyntrunk_table* and fill it with the starting address of allocated memory in preceding step. If no free entry exists, free allocated memory. Then, return an error.

6. Set the *dtno* according to the index of *system_dyntrunk_table* and the number of functions and update the *dtflags* (usually, DTM_RUNNING and DTM_CHECK_ID are set) to indicate that the trunk is registered successfully.

The functions *dyntrunk_request*, *dyntrunk_release* and *dyntrunk_info* are called by user applications.

As mentioned above, child process copies the *process_dyntrunk_table* of its parent to its own PCB. All the trunks which the parent process has access to can be used by the child process. If other services are needed, it should apply for corresponding trunk. Maybe more than one similar service is provided by different trunks to diverse applications. So, user should declare a *dyntrunk* structure with the qualification of trunk, such as its *dtid*, *dtver*, or *dtname*. The conditions can be set as bits of *dtflags*. The *dyntrunk_request* handles the request for a trunk in the following order:

1. We first search the *system_dyntrunk_table* for the registered trunk which matches the trunk user required. For example, check the *dtid* if DTM_CHECK_ID is set, compare *dtver* according to the flag DTM_VER_EQ, DTM_VER_LE, DTM_VER_GE or DTM_VER_NE.
2. Next, if the requested trunk exists in the *system_dyntrunk_table*, we check the capability of the trunk to make sure it is privileged to use.

3. If the trunk with appropriate capability is found, copy it to the *process_dyntrunk_table* and update *dtflags*. Otherwise, set the *dtflags* of dyntrunk declared by user to DTM_INVALID to indicate the service user request is not available. If the DTM_ALTERNATE is set, a function can be called as substitute.

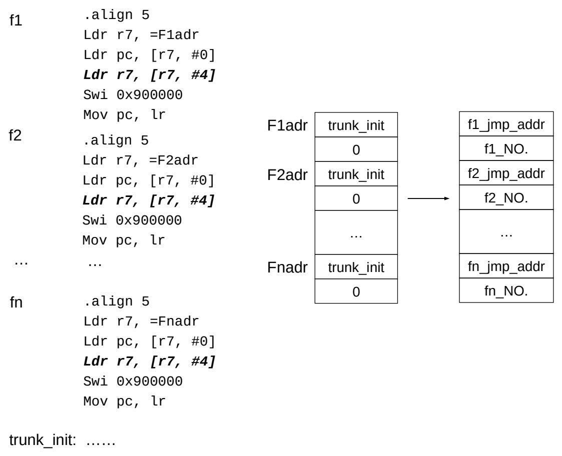
So, the *process_dyntrunk_table* is usually a subset of *system_dyntrunk_table*. It only contains the trunks the process has access to, and needs to use. The entry in *process_dytrunk_table* can be deleted via *dyntrunk_release* if the process doesn't use the trunk any more. *Dyntruk_info* just prints all the information about trunks which can be used by the process.

Furthermore, a necessary facility to support the use of DTM is *dyntrunk dispatcher* which provides the entrance to dynamic trunk management components. In our implementation, user calls the dispatcher via "swi". Then the dispatcher finds system services via the *process_dyntrunk_table* according to the parameters passed by user-level function.

4.3 User-level Supporting

All system services are provided as trunks. Therefore, the interface between system and upper level is the trunk. Application which calls a system service must request corresponding trunk (call *dyntrunk_request*) if the trunk is first used. Then, the service can be used directly. As described above, a *dispatcher* in kernel is responsible for kernel functions calls. But user-level support is required to implement the trunk request task and set appropriate parameters for the dispatcher.

A header file consisting of transform function for each trunk is provided to achieve this goal. The template is defined as follow (Figure 3).


```
f1      .align 5
        Ldr r7, =F1adr
        Ldr pc, [r7, #0]
        Ldr r7, [r7, #4]
        Swi 0x900000
        Mov pc, lr
f2      .align 5
        Ldr r7, =F2adr
        Ldr pc, [r7, #0]
        Ldr r7, [r7, #4]
        Swi 0x900000
        Mov pc, lr
…       …

fn      .align 5
        Ldr r7, =Fnadr
        Ldr pc, [r7, #0]
        Ldr r7, [r7, #4]
        Swi 0x900000
        Mov pc, lr

trunk_init: ……
```

Figure 3. The DTM call header file template

A mechanism similar to relocation in dynamic library is used for flexibility and efficiency. There is a table for each trunk to record information of its functions, including the jump address and function NO. When the table is initialized, the function address is set to the init function of the trunk, which calls *dyntrunk_request* to the use of trunk to kernel and the function number is set to zero. If the trunk is used by a process at the first time, the second and third states call the *trunk_init* function. Once the trunk is requested successfully, the entries of the table are modified. The f1_jmp_addr is filled with the address of the third state (in italic type) and the f1_NO is set. The upper 16bits of f1_NO is the same as *dtno* while the lower 16 bits are the index of the function in the function list of trunk, not the total number. Therefore, the template executes in sequence to call the dispatcher with parameter fn_NO.

The template is implemented in assemble language for efficiency. The portability will not be prohibited though assemble language is architecture depended. Because the template only consists of five states which can be implemented manually or by compiler easily.

4.4 Access control mechanism

All programmers agree in theory: an application should have the minimal privilege needed to perform its task [23]. If we treat all the system services as elements of Set *S*, the ones application can have access to is Set P, and the ones they acquired is set R. Then P is the subset of S and R is the subset of P. Most operating systems in general favor the subtractive model, which provided applications with services all they can use, set P. This module exposes more services to applications than they require. Capability-based operating systems [20, 31] use additive module which is similar to what we used in DTM. The *process_dyntrunk_table* stores the trunk the process has access to. When a trunk is called for the first time, the capability is checked. If it is privileged, the information of the trunk is copied to the *process_dyntrunk_table*. Compared with traditional interface access control mechanism, the one used in DTM is more flexible and helpful to achieve principle of least privilege [28].

5. Evaluation

All DTM related issues are detailed until now. In this section, we describe a reality extension system in our lab first, then, several experiment results about DTM got on the system.

5.1 An Extension System

The Dynamic Trunk Mechanism has been implemented based on the Linux 2.4.17 kernel. Several application-specific kernel services are extended as trunks. One of the kernel extension functionalities is for a web-browser to provide its own memory-management mechanism (called unimem). The functions of this trunk are:

Table 1. Functions of unimem trunk

Function name	**description**
Sys_unimem_map	Map the Dynamic Memory Space to the process space
Sys_unimem_allo c	Allocate part of Dynamic Memory Space
Sys_unimem_free	Free part of Dynamic Memory Space
Sys_unimem_exit	Free allocated Dynamic Memory Space

First of all, the DTM trunk must be registered to the system. We call the init route of Dynamic Trunk Mechanism in *main.c*. Then each trunk has the responsibility to register itself. The unimem trunk is registered by following codes.

```
int __init unimem_dtinit(void)
{
       struct dyntrunk umimem_dt =
            {
                   0x10001,            /* dtid */
                   0x1,                /* dtver */
                   0,                  /* dtno */
                   0,                  /* dtflags */
                   CAP_DT_UNIMEM,      /* dtcap      */
                   0,                  /* dtreserved */
                   “unimem"};          /* dtname     */
       return(dyntrunk_register(&unimem_dt,
                          sys_unimem_map,
                          sys_unimem_alloc,
                          sys_unimem_free,
                          sys_unimem_exit,
                          0 ));
}
#ifdefine CONFIG_UNIMEM
__initcall(unimem_dtinit);
#endif
```

A file named unimem.S is implemented as Figure 3 and a header file named unimem.h is provided to user to call the functions in trunk unimem. Following is a sample code which calls sys_unimem_map.

```
#include <dyntrunk.h>
#include <unimem.h>

void function(……)
{
       ……
       unimem_map(……)
       ……
}
```

Kernel services for multimedia and other applications are implemented similarly. System designers can configure the trunks according to their application environment just by modifying a configure file.

5.2 Experiments

Extensible operation systems should provide flexible, safe extension technology without little overhead.

It is obvious that DTM is easily to be implemented with little modification of existing operating system. Furthermore, as shown in above example, the extension and configuration of system services can be easily implemented via DTM. Also, the access control mechanism in DTM guarantee that the extensions are protected from malicious applications and enforced upon uncooperative applications. For example, the s*ys_unimem_alloc* of unimem trunk is specifically for web-browser, the multimedia media applications have no access to this kernel service unless they are privileged.

In order to evaluate the overhead of DTM and impact to applications, measurements have been performed on our thin-client platform with 200MHz and 128MB RAM

We compare the Dynamic Trunk Mechanism with traditional system call table extension. The details of testing files are as follows:

- system call table extension: A function which simply return a integer value is added as a benchmark to system call table . This function is called via “swi” directly by user application without any modification to the rules of system call.
- dynamic trunk mechanism: The same benchmark is extended to kernel using DTM(the example system described above). This function is called via *dispatcher* of the dynamic trunk mechanism.

Table 2. Comparison of traditional system call and DTM

	Traditional System call	**Dynamic Trunk Mechanism**
sys_time	**417.3 ns**	**468.3 ns**
user_time	**943.2 ns**	**899.0 ns**
real_time	**1360 ns**	**1367 ns**

The function has been looped for 1×10^9 times. The per-execution time (Table 2) of function call in DTM invocate overhead commensurate with traditional system call. DTM is as efficiency as traditional system call mechanism.

To demonstrate the functionality and performance of DTM in real application, we test the performance of the web-browser described in Section 5.1. Also, kernel services have been implemented in two ways: one is to extend the system call table directly, and the other is to use DTM. The benchmark i-bench 5.0 was used. Results show that DTM has no obvious impact on applications.

6. Conclusion

In this paper, we describe the design and implementation of Dynamic Trunk Mechanism for making operating systems extensible and configurable.

Compared with traditional operating system, the interface provided to user has been structured as a set of trunks which consist of related kernel services. As basic components in operating system, the trunks are optional. Designers can register necessary trunks to system and extend kernel services for specific-application in the form of trunk. In DTM, applications only have access to the trunks it has been privileged to. This is helpful to system security to some extent. In addition, the implementation of DTM requires only trivial operating system changes and the result shows that the performance of this mechanism is commensurate with traditional system call. It has been used in our platform. In our experience, DTM provides a flexible, efficient and safe operating system extension and configuration method.

In fact, the DTM places no specific requirements on the underlying OS structure. As a consequence, it is possible to be used in two-level operating systems. We believe that the mechanism described in this paper can provide benefit to users of existing operating systems, such as Linux, enabling them to extend and configure the operating system to diverse application environments.

7. References

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. “Mach: A New Kernel Foundation for UNIX Development”.Proc. of the USENIX Summer Conference, GA, USA, pp.93-113, 1986.

[2] Banga, G., Mogul,J. C., and Druschel, P. “A scalable and explicit event delivery mechanism for UNIX”. Proc. of the USENIX Annual Technical Conference, Monterey, CA, pp. 53-266, 1999.

[3] Batory, D. “Subjectivity and GenVoca generators”. Proc.of the Fourth International Conference on Software Reuse, pp. 166-175,1996.

[4] Bershad, B.N., Savage, and S., Pardyak, P. et.al. “Extensibility safety and performance in the SPIN operating system”. Proc. of the fifteenth ACM symposium on Operating systems principles. Colorado, United States, pp.267-283, 1995.

[5] Bestavros, A., Carter, R. L., Crovella, M. E., Cunha, C. R., Heddaya, A., and Mirdad, S. A. “Application-level document caching in the Internet”. Proc. of Second International Workshop on Services in Distributed and Networked Environments, BC, Canada, pp. 166-173, 1995,.

[6] Butner, M., Benson, G.D., Padden, S., and Fedosov, A. “The Virtual Processor Interface: Linux Kernel Support for User-level Thread Systems”. Proc. of 15th International conference on Parallel and Distributed Computing and Systems, Marina del Rey, CA, pp.115-120, 2003.

[7] Cheriton, D. R. and Duda, K. J. “A caching model of operating system kernel functionality”.ACM SIGOPS Operating System Review, vol. 29,NO.1, pp. 83-86, 1995.

[8] Clarke, M. “Operating System Support for Emerging Application Domains”. Ph.D. Thesis, Lancaster University, UK, 2000.

[9] Clarke M. and Coulson, G. “An Architecture for Dynamically Extensible Operating Systems”. Proc. of the 4th International Conference on Configurable Distributed Systems, MA, USA, pp.145-155, 1998.

[10] Clark, D. D. “The structuring of systems using upcall”. Proc. of the 10th Symposium on Operating Systems Principles. WA, USA, pp.171-180, 1985.

[11] Craig, A.N. Soules, Appavoo, J., Hui, K. “System Support for Online Reconfiguration”. Proc. of USENIX 2003 Annual Technical Conference, Texas, USA, pp.141-154, 2003.

[12] DeTreville, J. “Making System Configuration More Declarative”. Proc. of the Tenth Workshop on Hot Topics in Operating Systems, NM,USA, pp.15-20, 2005.

[13] Douglis, F., and Ousterhout, J. K. “Transparent process migration: Design alternatives and the Sprite implementation”. Software: Practice and Experience, vol. 21, No.8, pp. 757-785, 1991.

[14] Druschel, P. “Efficient Support for Incremental Customization of OS Services”. Proc. of the Third International Workshop on Object Orientation in Operating Systems, NC,USA, pp.186-190,1993.

[15] Eicken, T. v., Culler, D.E., Goldstein, S.C., and Schauser, K.E. “Active Messages: a Mechanism for Integrating Communication and Computation”. Proc. of the 19th Annual International Symposium on Computer Architecture, Queensland, Australia, pp.256-266, 1992.

[16] Engler, D. R., Kaashoek, M. F. et.al. “Exokernel: an operating system architecture for application-level resource management”. Proc. of the fifteenth ACM symposium on Operating systems principles. Copper Mountain, Colorado, United States, pp.251-266, 1995.

[17] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G. and Clawson, S. “Microkernels Meet Recursive Virtual Machines”. University of Utah Technical Report UUCS-96-004, 1996.

[18] Gabber, E., Small, C., Bruno, J., Brustoloni, J., and Silberschatz A. “The Pebble Component-Based Operating System”. Proc. of Usenix 1999 Annual Technical Conference, California, USA, pp.267-282,1999.

[19] Ghormley, D. P., Rodrigues, S. H., Petrou, et.al. “SLIC: An Extensibility System for Commodity Operating Systems”. Proc. of USENIX 1998 Annual Technical Conference, BerKeley, CA, pp.39-48, 1998.

[20] Hardy, N. “KeyKOS architecture”. ACM SIGOPS Operating Systems Review, vol. 19, No.4 pp. 8 - 25 1985.

[21] Jones, M. B. “Inheritance in unlikely places: using objects to build derived implementations of flat interfaces”. Proc. of

the Second International Workshop on Object Orientation in Operating Systems, Dourdan, France, pp.342-345,1992.

[22] Krell E., and Krishnamurthy ,B. “COLA: customized overlaying”. Proc. of USENIX San Francisco Winter 1992 Conference, TN, USA, pp.3-7,1992.

[23] Krohn, M., Efstathopoulos,P., et.al. “Make Least Privilege a Right (Not a Privilege)”. Proc.of The Tenth Workshop on Hot Topics in Operating Systems,NM, USA,2005.

[24] Liedtke, J. “On micro-kernel construction”. Proc. of the fifteenth ACM symposium on Operating systems principles. Colorado, United States, pp.237-250, 1995.

[25] Lemon, J. “Kqueue: A generic and scalable event notification facility”. Proc. of the FREENIX Track *(USENIX-01),*California, USA, pp.141-153, 2001.

[26] Mogul, J. C. “Operating Systems Should Support Business Change”. Proc. of The Tenth Workshop on Hot Topics in Operating Systems, NM, USA, 2005.

[27] Rousseau L. and Natkin, S. “A framework of secure object system architecture”. Proc. of the Third International Workshop on Object-Oriented Real-Time Dependable Systems, California, USA, pp. 108-115,1997.

[28] Saltzer J. H. and Schroeder, M. D. “The protection of information in computer systems”. Proceedings of the IEEE, vol. 63, No. 9, pp. 1278-1308, 1975.

[29] Seltzer, M. I., Endo, Y., Small,C., and Smith, K.A. “Issues in Extensible Operating Systems”. Harvard University Technical Report TR-18-97, 1997.

[30] Serra, A., Navarro, N., and Cortes, T., “DITools: Application-level Support for Dynamic Extension and Flexible Composition”. Proceeding of USENIX 2000 Annual Technical Conference, California, USA, pp.1-12, 2000.

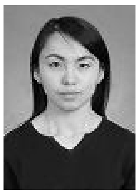
[31] Shapiro, J. S.J. M. Smith, and D. J. Farber, “EROS: a fast capability system”. ACM SIGOPS Operating Systems Review, Vol.33, No.5, pp.170-185, 1999.

[32] Small C. and Seltzer, M. “A Comparison of OS Extension Technologies”. Proceedings of USENIX 1996 Annual Technical Conference, CA, USA, pp.41-54, 1996.

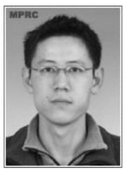
[33] Tan, S.-M., Raila, D. K., and Campbell, R. H. “An object-oriented nano-kernel for operating system hardware support”. Fourth International Workshop on Object-Orientation in Operating Systems, Lund, Sweden, pp.220-223, 1995.

[34] Theimer, M.M., Lantz, K.A., and Cheriton, D.R. “Preemptable remote execution facilities for the V-system”. Proc. of the tenth ACM symposium on Operating systems principles, Washington, United States, pp.2-12,1985.

[35] West, R., Gloudon, J., Qi, X., and Parmer, G. An Efficient User-Level Shared Memory Mechanism for Application-Specific Extensions. Computer Science Department. Boston University 2005.



Liu Shu: was born in 1982. She received B.S. degree in computer science from Beijing Institute of Technology in 2004. Now she is a doctor candidate in Micro Processor Research and Development Center of Peking University. Her research interests include operating system, embedded system and system software.



Guan Xuetao： was born in 1974. He received B.S. degree in computer science from Shandong University in 1997. And he received his doctor degree in 2006. He has taken part in several national projects related to microprocessor and embedded system software. His research interests include operating system, embedded system, system software and system chips.

Cheng Xu: was born in 1964. He is a professor and doctoral supervisor of Peking University. He is also the member of VLSI Design Expert Group of National 863 Program. He received Ph.D. degree from Harbin Institute of Technology in 1994 and Post Doctor from Computer Science Department of Peking University in 1996. His main research interests are high performance microprocessor design, SoC design, embedded system, instruction level parallelism, compiler optimization and hardware software co-design.