Separating Data Storage, Data Computation, and Resource Management One from Another in Operating Systems*

Fuyan Liu[1,2] and Jinyuan You[1]

[1] Dept. of Computer Sci. & Eng.,
Shanghai Jiaotong Univ.,
Shanghai 200030, China
[2] Dept. of Computer Sci. & Tech.,
North China Inst. of Tech.,
Taiyuan 030051, China

Abstract. It's impossible for traditional operating systems to separate the abstract for data storage (process virtual address space), the abstract for data computation (thread), and the abstract for resource management (process itself) one from another. In this paper, firstly we analyze the problems due to not separating these three abstracts. On the basis of the analysis, we propose the idea that these three abstracts should be separated, and on the basis of the idea, we propose Operating Systems Basing Virtual Address Spaces on Files (OS-BVASF). Then we investigate OS-BVASF architecture model, Thread Migration and Instructions Accessing Files Directly that implement the separation. In the end of this paper, we discuss its implementation and performance test.

1 Introduction

Constructing some abstracts, with system calls to operate them, can be thought as operating system's function. Operating Systems should construct at least four types of abstracts: the one for data storage, the one for data computation, the one for resource management, and the one for input and output. In a traditional operating system, the abstract for data storage is process virtual address space; the abstract for data computation is thread; the abstract for resource management is process, through which operating system implements the resource management and allocation; the abstract for input and output is file.

* This work was support by the National Natural Science Foundation of China (60173033, The Architecture Research of Operating System Basing Virtual Address Space on Files); also by the Youth Technology Research Foundation of Shanxi Province (20021016, The Architecture Research of Operating System Basing Virtual Address Space on Files), and by the Defense Science and Technology Key laboratory Foundation (51484030301JW0301, The Architecture Research of Operating System Basing Virtual Address Space on Files for Large Scale and Parallel Computer).

X. Zhou et al. (Eds.): APPT 2003, LNCS 2834, pp. 114–124, 2003.

Among the four types of abstracts constructed by operating systems, file is a special one. Only file is persistent, only file belongs to no process. The other three abstracts must belong to some process or even process itself, and cannot be separated one from another:

1. Thread and process cannot be separated: a thread can only use resources that belong to its process; threads that belong to the process can only use the resources of a process.
2. Thread and process virtual address space can not be separated: the virtual address space of a process can only be addressed by threads that belong to the process; threads that belong to a process can only address the virtual address space of the same process.
3. Process virtual address space and process itself cannot be separated: it's impossible to separate process virtual address space and process itself one from the other. Process virtual address space is part of the process itself.

Because data storage (process virtual address space), data computation (thread), and resource management (process) can not be separated one from another, when a thread that belongs to one process access services provided by other process, there exist some problems, for example,

1. Because thread and process cannot be separated, the client thread cannot enter directly into server process. When client thread need call a service in server, it has to employ client/server model and inter-process communication mechanisms, such as Message-Passing, semaphore, pipe, mail-box; socket, the service is called by a proxy thread in server process. Firstly client thread awakes server thread, then client thread get slept. When it finishes its task, the server thread awakes client thread, after that it gets slept. The client thread gets awaked and keeps on executing. For the client thread, it only calls a server service once, but for operating system, it need execute several times thread-switching as well as thread sleeping and waking up. Comparing with that of calling a function in libraries or a system call in operating system kernel, the performance of calling a service in server is poorer. This is one of the reasons why micro-kernel operating system is of poor performance [1].
2. Because thread and process virtual address space cannot be separated, server thread cannot access data in client process virtual address space. Client thread has to pass calling parameter through mechanism such as message passing, pipe, mail-box, socket, etc, so has to server thread return result to client thread. If there needs passing large mount of data, the performance may turn worse. This is another reason why micro-kernel operating is of poor performance [1].
3. Because process virtual address space and process itself can not be separated, there exists the following problems:
   - The problem of data persistency: the non-persistency of a process makes the data in process virtual address space lose its persistency. To implement data persistency, data has to be read from file before threads access them, also it has to be written to file after threads has finished accessing them. Process virtual address space is NOT physical memory, although it is an abstract constructed through memory and PERSIST swapping device (usually one or more disks), it has lost the persistency of swapping device.

- The problem of complex data structure preservation: the meaning of a pointer in high-level languages is essentially an address in process virtual address space. Even if we have saved a pointer persistently, once the process holding the pointed data has exited, the pointer turns out meaningless. This means that complex data structure cannot be persistently preserved in file directly, it need to be translated into another data structure without pointer before it is saved to file.
- The problem of complex data structure share: a pointer belongs to a unique process virtual address space, a pointer defined in one process virtual address space is meaningless to another process, complex data structure that comprise pointers can not be shared among different processes. Although it has implemented memory-sharing and memory mapping file, traditional operating system can only implement the sharing of some sections of process virtual address space, it can not implement the sharing of complex data structure that comprise pointers among different applications.

To solve the problems listed above, we propose and are implementing a new operating system, which constructs abstracts different from the ones in traditional operating systems. In our operating system:

1. Only three abstracts are constructed, the three abstracts include the one for data computation (thread), the one for resource management (process), and the one for input and output (file). The function implemented by data storage abstract (process virtual address space) in traditional operating systems is implemented by files. Similar to memory mapping file in traditional operating systems, in our operating system, instructions access files directly, threads run directly on files.
2. The three abstracts, including the abstract for data computation (thread), abstract for resource management (process), and the abstract for input and output (file), are separated one from another. Operating system kernel provides Thread Migration system call, by calling it, a thread can enter from current process to another process, calling services in server process directly, and the thread can also return to original process and keep on running, just as calling system calls in traditional operating systems.

As our operating system integrates process virtual address space and file together, applications run directly on files, the virtual address spaces accessed by threads are file spaces; we call it Operating Systems Basing Virtual Address Spaces on Files (OS-BVASF). In OS-BVASF, when a thread need access some services implemented by other processes, the thread can get into other process directly by calling thread migration system call implemented by operating system kernel. When completed, the thread can also return to original process and keep on running by calling thread return system call.

Fig. 1 shows the abstracts constructed in OS-BVASF, as well as their relations. OS-BVASF can solve the problems due to not separating the three abstracts.

1. Because thread and process are separated, thread can enter directly into server process by thread migration system call, avoiding operations such as thread-switching, thread-sleeping and thread- waking-up that are necessary in traditional operating systems.

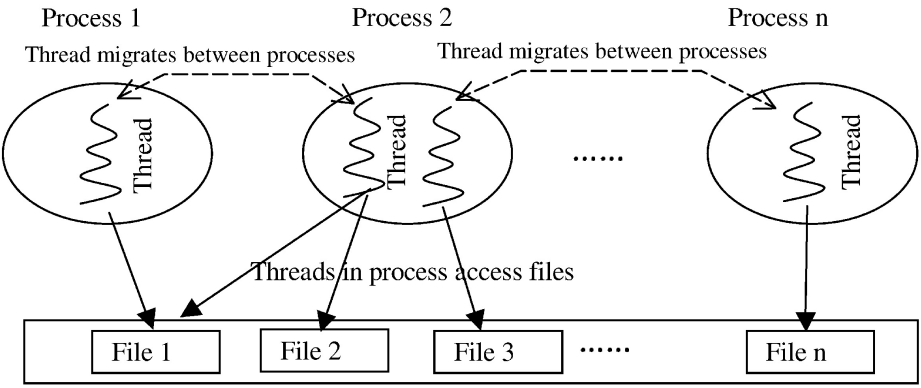

Fig. 1. The abstracts built by operating system and their relations

2. Because process virtual address space and file are integrated, all threads run on files, a thread can access all data in all files provided it has the accessing right. Safety and protection is implemented through file right control. The data that a thread can access is not confined within its process virtual address space. It's unnecessary to copy calling parameters and returning results between client and server when a client calls a service in server. This can help improve the performance of micro-kernel operating systems.
3. Because all threads run on files, the meaning of a pointer in high-level language is an address in file space, not the one in process virtual address space. A pointer is not confined within a process virtual space; it can be saved persistently in files directly, as well as be shared among different applications. This can avoid the problem of complex data structure preservation and the problem of complex data structure share.

2 The Architecture of OS-BVASF

2.1 The Hierarchy of OS-BVASF

Fig. 2 illustrates the hierarchy of OS-BVASF. In Fig. 2, the middle layer is OS-BVASF kernel, which implements thread migration, semaphore management, processor management and memory management. The System Call Interface, Interruption Handler, Address Mapping Component, and Switch Component in Fig. 2 is related to hardware platform, which should be re-implemented when OS-BVASF is migrated to other platforms. Upon the kernel are all sorts of servers, including communication server, file server, directory server, Linux emulator, etc. Applications are also servers existing in the form of process; any application can be called by other applications as a server.

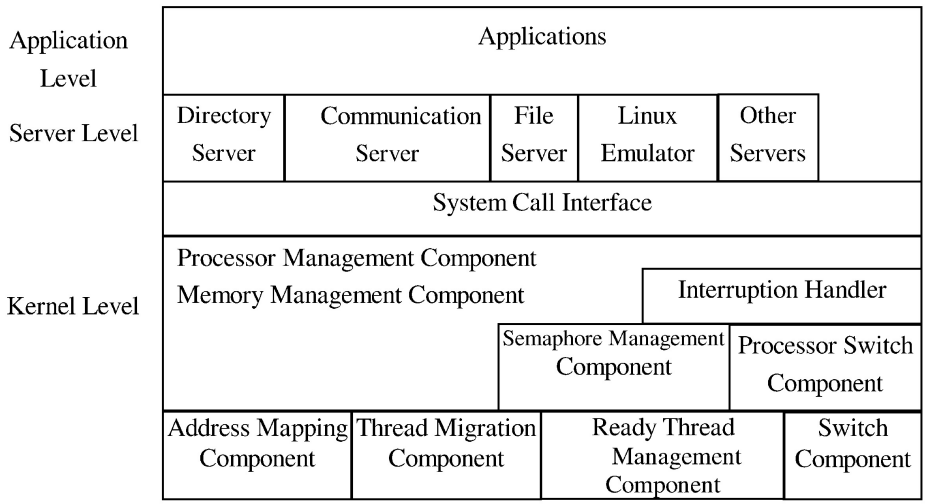

Fig. 2. The Architecture and hierarchy of OS-BVASF

2.2 The Function of OS-BVASF Kernel

In OS-BVASF, the following functions are implemented in the kernel: Thread-migration and thread-return, semaphore management, processor management and memory management. Through Thread-migration and thread-return, a thread can migrate between processes; through semaphore P operation and V operation, the mechanisms such as thread synchronization and thread mutex, are implemented. Processor management (process creation and exit, thread creation and exit, setting thread priority, etc.) and memory management (storage domain creation and exit, file open and close, etc.) are also implemented in kernel. File systems can exist either in the kernel as device drivers or outside the kernel as applications (servers).

2.3 File Servers, Communication Servers, and Applications

In OS-BVASF, all applications exist outside the kernel; they can be treated as virtual file servers, as a program module that can be called by other applications, and also as a program unit that implements application division and protection. Through thread migration, a thread can enter directly into another application, starting from the Initial Execution Point, the thread begin run within the new process. When completed, the thread can also return to its original process and keeps on running.

The function of all sorts of applications (also can be treated as virtual file servers) is to manage persistent objects inside and provides virtual file call interfaces to other applications. By treating all applications as virtual file servers, OS-BVASF implements the function of file systems in traditional operating system, as well as the integration of file server and communication servers.

2.4 Linux Emulator

OS-BVASF implements Linux compatibility through Linux emulator as Mach does. A Linux thread calls Linux emulator through thread migration system call. Just as a Linux thread enters Linux operating system kernel, a thread in OS-BVASF enters Linux emulator, calls it execute function of Linux kernel. When completed, it will return to original application.

3 Thread Migration – Separating Data Computation and Resource Management One from the Other

3.1 The Conception of Thread Migration

In OS-BVASF, two processes relate to a thread, one is called owner process, it identifies to which process the thread belongs, the other is called current process, it identifies within which process the thread currently exists, its also identifies which process's resource the thread currently uses. Within its whole lifetime, thread can not change its owner process, whereas at any time, thread can change its current process through thread migration. When it calls thread migration system call, a thread changes its current process and resource environment, enters from one process to another, executes program within the new current process, when completed, it can also return to its original process. Thread migration is one of the main functions implemented within operating system kernel. One of the differences between OS-BVASF and traditional operating system is that through thread migration instead of message-passing or pipe, etc., a thread calls functions within others processes, thus prevents thread switching, thread sleeping and waking up, and improves performance.

Thread migration implements the separation of data computation (thread) and resource management (process).

3.2 Some Techniques Related to Thread Migration

Many operating system designers recognize the worse performance due to client/server model and adopt some techniques to solve this problem. From these techniques and systems, we propose the thread migration:

- Sun's Spring OS introduces a new conception: the Door, through which thread can enter from one domain to another. In Spring OS, Door is defined in IDL language, which makes threads be possible to enter into other process even on different computer or heterogeneous architecture. Sun's Spring OS is one of the operating systems from which we propose OS-BVASF thread migration technique.
- Grasshopper operating system constructs Loci (similar to thread in traditional operating system) which can migrate from one Container (similar to process in traditional operating system) to another. Loci abstract in Grasshopper operating system is persistent. It is from the Loci abstract in Grasshopper operating system that we propose the separation of data computation (thread)

and resource management (process), the implementing technique is thread migration

- MIT's Aegis operating system adopts the idea of Exokernel. Through exception forwarding and upcall, execution in Aegis can go from one process to another. From MIT's Aegis operating system, we propose exception processing in OS-BVASF.

3.3 Thread Initial Executing Context (TIEC) and Thread Initial Executing Point (TIEP)

When a thread enters into target process through thread migration, it starts running with a special CPU context called Thread Initial Executing context (TIEC). TIEC's instruction pointer is address of the first instruction that thread executes within target process, TIEC's registers store calling-parameters from source process.

In traditional operating systems, when a thread enters into kernel, it will start running at a special address, at which operating system kernel does switch and checks if the thread owns the accessing rights. If it does own, operating system kernel executes the called function, if not, operating system kernel refuses. This mechanism ensures the security and safety. Just as in traditional operating systems, in OS-BVASF, when a thread enters target process through thread migration system call, it will start running at Thread Initial Executing Point (TIEP), where target process will do some checks to ensure the thread is not a malicious one, and guarantee security and safety.

3.4 The Returning of Migrated Thread

When a thread has finished its task within target process, it can return to its original process through calling thread return system call. The execution flow of thread migration and thread return is similar to that of system call in traditional operating system. The difference is that in traditional operating system, the called object is operating system kernel, whereas in OS-BVASF, the called objects are processes outside operating system kernel. Instead of client/server model and message passing in traditional micro-kernel operating systems, OS-BVASF employs thread migration and thread return to put some function of traditional operating system outside operating system kernel. This mechanism can improve extensibility and performance of operating system.

4 Instructions Access Files Directly – Separating Data Storage from Data Computation and Resource Management

Similar to Memory Mapping File in traditional operating system [6], in OS-BVASF instructions access files directly, all applications run on files. Through Instructions Access Files Directly, OS-BVASF implements separating data storage (file) from data computation (thread) and resource management (process).

In computer systems, instructions access virtual address space. For example, at the protection mode of X86 processor, when it executes an assembly instruction MOV DS:[2000H],AX, X86 CPU will save the value in register EAX to a virtual address unit whose segment is identified by DS register and offset is 2000H. This unit may be either in physical memory or on exterior storage device. If it is not in physical memory, the executions of the instruction will trigger a memory fault exception; the exception handler will load data of the unit into physical memory, modifies segment table and page table, and restarts the instruction execution. The relation between virtual address space and physical memory as well as exterior storage device is maintained by operating system. If we define it as the relation between file address space and physical memory as well as exterior storage device, operating system will implement Instructions Access Files Directly. When programs are running, it seems that instructions access files directly, it also seems to the users that there exist no process virtual address space; programs are not running in process virtual address space, but in file address space. Instructions Access Files Directly can be implemented in the following method:

1. CPU employs segmentation or segmentation with paging virtual memory management, which a lot of CPU chip support. The virtual address space is two-dimensional, one is segment, and the other is offset within the segment.
2. Logically a segment in virtual address space corresponds to a section of file on file server.
3. Before an application tries to access data, file should first be opened. When open file, operating system constructs segment table and page table, return the segment ID to application as file handler. When application access data through the segment ID (file handler), it will actually access file data.
4. When thread is running and accessing a segment identified by the segment ID (file handler), if data does not exist in physical memory, a memory fault exception occurs, the exception handler will call file server to load data into physical memory, modifies segment table and page table, restart the excepted instruction to keep on running.
5. If memory resource gets scarce, operating system can select some page frames, write data in them to files on file servers, and releases these page frames.
6. In the end, applications close files, write all data in physical memory to file on file server, release memory resource as well as segment table and page table.

By the method described above, operating system implements Instructions Access Files Directly. When instructions access segments, it seems that the instructions access files. To applications, there is only conception of file, without that of process virtual address space. Applications do not run among process virtual address space, but on files.

Because the function of data storage is implemented not through process virtual address space, but through file abstract, and because files belong to no thread (data computation) or process (resource management), OS-BVASF separates data storage from data computation and resource management through Instructions Access Files Directly.

5 OS-BVASF Implementation and Performance Test

Upon X86 PC, we have implemented an OS-BVASF kernel. Its architecture is based upon Thread Migration and Instructions Access Files Directly. Upon the kernel, OS-BVASF applications run directly on files. Our development environment is Redhat Linux 7.1, Linux kernel version is 2.4.2, CPU is Pentium-IV, Clock Frequency is 1.8G, main memory capacity is 512M, and hard disk capacity is 40G.

The development language of OS-BVASF kernel is GCC under Linux environment. We also employ Linux LILO to install OS-BVASF kernel. But OS-BVASF kernel and exterior kernel applications run directly upon hardware, without the support from Linux. Our implement is not an emulator upon Linux environment; Linux is only our development environment.

5.1 Performance Test of Thread Migration

To appraise performance of Thread Migration, we test the time length needed for thread migration between two different processes. On the same hardware platform and Linux operating system, using pipe and message buffer, we also test the time length to do an execution switching between two different Linux processes. Test result is listed in Tab. 1.

OS-BVASF threads can execute at either kernel mode or user mode. If the source process and target process mode are different, so is the test result. That is the reason why Tab. 1 includes four items. But our test result doesn't show the difference too much.

Table1. Execution switching time length in OS-BVASF and Linux (unit: μs)

In OS-BVASF through Thread Migration			
User-User	User-Kernel	Kernel-User	Kernel-Kernel
3.2	2.9	3.3	3.4
In Linux through Message Buffer and Pipe			
Through Message Buffer		Through Pipe	
6.3		6.4	

From Tab. 1 we can conclude that OS-BVASF execution switching time between two different processes is shorter than that of Linux.

5.2 Performance Test of Instructions Access Files Directly

The mechanism of Instructions Access Files Directly is very similar to that of Memory Mapping Files in traditional operating systems [6]. To appraise the performance of OS-BVASF memory management, we test the performance of Linux Memory Mapping File, and compare the test result to that of Instructions Access Files Directly in OS-BVASF. The test method is:

- Under Linux environment, we map file /dev/zero to a process virtual address space. Through repeatedly accessing mapped virtual pages, we test the time length for handling a page fault exception. File /dev/zero is a special character device that supports memory mapping file, all data read from it is zero.
- Under OS-BVASF environment, we open a file. Also through repeatedly accessing mapped virtual file pages, we test the time length for handling a page fault exception. In order to make the test comparable, we also set all data in file zero. We do the test when file is both managed by device driver within memory manager in kernel and file system outside kernel.

The time length for handling a page fault exception under Linux environment is 19.8ms under out test platform and environment. The test result under OS-BVASF environment is listed in Tab. 2. Once one memory fault exception occurs, OS-BVASF memory manager can set up address mapping for multi-page frames when handles the exception. The number of frames is identified when file is opened. In Tab. 2, the first row is the number of frames, the second row is test result when device driver within memory manager in kernel manages file, and the third row is test result when file is managed by file system outside kernel.

Table 2. Time length for handling memory fault exception (unit: μs)

Mapped Page Frame Number	File is managed by device driver within memory manager in kernel	File is managed by file system outside kernel
1	15.7	37.6
2	13.3	29.6
4	12.2	28.8
8	11.6	28.7
16	11.5	27.8
32	11.4	27.7
64	11.4	27.7

From Tab. 2, we can conclude that if file is managed by device driver within memory manager in kernel, the performance for handling memory fault exception is better than that of Linux, if by file system outside kernel, the performance is worse.

6 Conclusions

By replacing message-passing and client/server model with Thread Migration, and also by Instructions Access Files Directly, OS-BVASF abandons process virtual address space, it makes all programs run directly on files, implements Separation of data storage, data computation and resource management one from another in operating systems. Comparing with traditional operating systems, OS-BVASF can solve the three problems due to not separating data storage, data computation and resource management. Our implementation shows OS-BVASF feasible and of better performance.

References

1. Wang, Shiyou, Guo, Fushun. The performance effect of Microkernel operating system architecture, Computer Research and Development, Jan, 1999 (Chinese).
王世铀 郭福顺 等，微核心操作系统的结构对性能的影响，计算机研究与发展，1999年1月
2. James G. Mitchell, Jonathan J.Gibbons: An Overview of the Spring System Sun Microsystems Inc. 2550 Garcia Avenue, Mountain View Ca 94303.
3. Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, Francis Vaughan. Grasshopper: An Orthogonally Persistent Operating System. Computing Systems, 7(3), pp 289–312, Summer 1994
4. Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, Francis Vaughan. Grasshopper: An Orthogonally Persistent Operating System. Grasshopper Technical Report GH-10. Department of Computer Science University of Adelaide S.A., 5001, Australia, 1994
5. Dawson R. Engler. The exokernel operating system architecture. Ph.D. thesis, Massachusetts Institute of Technology, October 1998.
6. M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceno, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In the Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97), Saint-Mal, France, October 1997, pages 52–65.
7. W. Richard Stevens, Advanced Programming in the UNIX Environment, Addison Wesley Publishing Company, 1992, Translate by You, Jinyuan, China Machine Press, 1999.
尤晋元等译，UNIX 环境高级编程，机械工业出版社，2000.2，pp.307–311