USENIX Association

Proceedings of BSDCon '03

San Mateo, CA, USA
September 8–12, 2003

Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions

Hideki Eiraku
College of Information Sciences, University of Tsukuba
hdk@coins.tsukuba.ac.jp, http://www.coins.tsukuba.ac.jp/~hdk/

Yasushi Shinjo
Institute of Information Sciences and Electronics, University of Tsukuba
yas@is.tsukuba.ac.jp, http://www.is.tsukuba.ac.jp/~yas/

Abstract

A user-level operating system (OS) can be implemented as a regular user process on top of another host operating system. Conventional user-level OSes, such as User Mode Linux, view the underlying host operating system as a specific hardware architecture. Therefore, the implementation of a user-level OS often requires porting of an existing kernel to a new hardware architecture. This paper proposes a new implementation method of user-level OSes by using partial emulation of hardware and static rewriting of machine instructions. In this method, privileged instructions and their related non-privileged instructions in a native operating system are statically translated into subroutine calls that perform emulation. The translated instructions of the user-level OS are executed by both the real CPU and the partial emulator. This method does not require detailed knowledge about kernel internals. By using the proposed method, NetBSD and FreeBSD are executed as user processes on NetBSD and Linux.

1 Introduction

Running multiple operating systems (OSes) simultaneously over a single hardware platform has recently become a popular system structuring approach that offers a number of benefits [SVL01] [Pap00] [Dik00]. First, application programs written for different operating systems, such as Unix and Windows, can be simultaneously executed on a single computer. Second, several versions of a single operating system, such as MacOS9 and MacOSX, can co-exist on the same platform. Other benefits include virtual hosting and easier system management and maintenance [HH79] [SVL01] [Pap00] [Dik00].

There are two prominent approaches to running multiple operating systems over a single hardware platform: Virtual machines [LDG+03] [Law03] and user-level operating systems [Dik00] [AAS94] [Tad92]. Virtual machines provide isolated execution environments for multiple operating system kernels, which can run over the native hardware. A user-level operating system is an operating system that runs as a regular user process on another host operating system. Conventional user-level OSes view the underlying host operating system as a specific hardware architecture. Therefore, the implementation of a user-level OS often requires porting of an existing kernel to a new hardware architecture. For example, User Mode Linux (UML) [Dik00], which is a user-level OS that runs on Linux and provides another Linux system image, adds a new architecture called *um* based on the i386 architecture. In general, such porting involves significant implementation effort, and requires detailed knowledge about the kernel and the base and new architectures. In porting of User Mode Linux, the size of the new *um*-dependent part is 33,000 lines while the size of the base i386-dependent part is 40,000 lines.

In this paper, we propose a new implementation method of user-level OSes with partial emulation of hardware and rewriting of machine instructions at compile time. The key idea is to enable the execution of most instructions by the real CPU with the exception of privileged instructions, hardware interrupts, and the interaction with some peripheral devices, which are emulated. We call this type of emulator a *partial emulator* or a *lightweight virtual machine* (LVM) because such a program does not have to emulate typical instructions, such as load, store, and arithmetic operations. In contrast, we refer to an emulator that executes all instructions as a *full emulator*.

In our implementation method, we emulate all privileged instructions. In addition, we emulate some non-privileged instructions that are tightly related with the privileged instructions. It is easy to detect execution of privileged instructions because the real CPU throws privilege violation exceptions. However, it is not trivial to detect execution of such non-privileged instructions.

To solve this problem, we use static rewriting of machine instructions at compile-time in two ways. One way is to insert an illegal instruction before each non-privileged instruction to be detected. Another way is to replace privileged instructions and related non-privileged instructions with subroutine calls for emulation. The translated instructions of a user-level OS are executed by both the real CPU and the partial emulator. By using our proposed method, we can generate a user-level OS based on a native OS without detailed knowledge about user-level OS internals. Furthermore, we can catch up the evolution of the base native OS easily. One disadvantage of our method is that we require source code of the user-level OS.

By using the proposed method, NetBSD and FreeBSD kernels are executed as user processes on NetBSD and Linux. Our user-level NetBSD on Linux is faster than NetBSD on Bochs [LDG$^+$03], a full PC emulator, by a factor of 10. However, our user-level NetBSD is slower than NetBSD on VMware and User Mode Linux on Linux. From the experiments results, we show that the main sources of slowdown are the emulation of memory mapping hardware and the redirections of system calls and page faults.

The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 describes the emulation of privileged and non-privileged instructions, the redirections of system calls and page faults, and the emulation of memory mapping hardware. Section 4 shows modifications of the NetBSD kernel for hosting our partial emulator. Section 5 describes modifications of the NetBSD kernel and the FreeBSD kernel for running as user processes. Section 6 shows the performance of the user-level NetBSD. Section 7 shows future directions, and Section 8 concludes the paper.

2 Related work

Running OSes as user-level processes has been proposed in the context of microkernel system research. For example, the Mach microkernel hosts BSDs, Linux, Hurd, and other systems [GDFR90]. In a microkernel-based system, the kernel provides primitive interprocess communication, memory management, and CPU scheduling. The OS servers outside the kernel implement file systems, network protocols, etc.

It is much easier to implement an OS server on a microkernel than to implement a monolithic kernel for bare hardware. However, in the case when we already have a native kernel for bare hardware, we have to port the native kernel to the microkernel. This porting sometimes involves significant effort. A native kernel accesses hardware directly and uses interrupts and privileged instructions. Accessing hardware should be replaced with using microkernel's facilities. In this paper, we show a method that translates a native kernel for bare hardware into a user-level OS with less effort.

The idea of nesting operating systems or virtual machines had appeared even in early virtual machines for mainframes [HH79] [LW73]. Apertos is a modern object-oriented operating system that supports nesting of operating systems or *meta-objects*[Yok92]. Fluke is another modern operating system [FHL$^+$96]. Fluke also supports efficient nesting or *recursion* with a microkernel technology. Both Apertos and Fluke have been designed to support nesting from scratch. Our method deals with commodity operating systems that are designed to run on bare hardware.

Instructional operating systems are often designed as user-level operating systems. SunOS Minix [AAS94] and VXinu [Tad92] have different structures from their native systems, PC Minix and PDP-11 Xinu, respectively. In our method, a user-level operating system has the same structure as the corresponding native operating system.

Plex86 is a virtual machine for Pentium [Law03].

Plex86 uses a special protection mechanism of Pentium, which is also known as Protected Mode Virtual Interrupts (PVI). To use this mechanism, Plex86 needs a kernel module. Plex86 provides hardware access, such as disks and networks, via a Hardware Abstraction Layer (HAL). Compared with Plex86, our approach differs in that we use a language processor (the assembler preprocessor), and we rewrite machine instructions of a user-level OS statically.

Some BSD kernels [HMM03] [LF03] have the facility to emulate other operating systems, such as Linux. In such environments, application programs written for different operating systems can be simultaneously executed on a single computer. Virtual machines and user-level OSes, including our approach, allow executing not only application programs but also operating system kernels.

Rewriting of machine instructions is used for address sandboxing with software [WLAG93]. To enforce a module to access a range of memory, this method inserts some masking instructions before each load or store instruction. In this paper, we use rewriting of machine instructions for realizing user-level OSes.

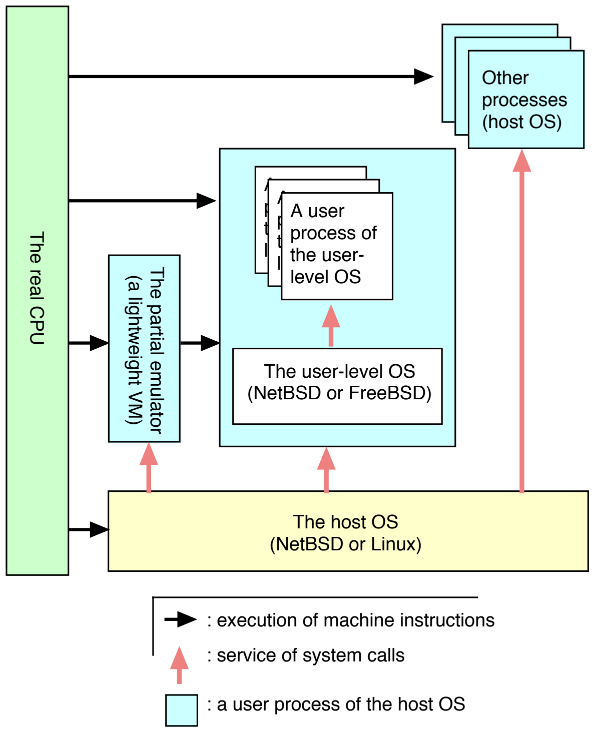
The real CPU
The partial emulator (a lightweight VM)
A user process of the user-level OS
The user-level OS (NetBSD or FreeBSD)
Other processes (host OS)
The host OS (NetBSD or Linux)
: execution of machine instructions
: service of system calls
: a user process of the host OS

Figure 1: Running an user-level operating system and its user process by the real CPU and a partial emulator.

3 Running a user-level OS by partial emulation and rewriting of machine instructions

In this section, we propose a new implementation method of a user-level OS by using partial emulation of hardware and static rewriting of machine instructions. In this method, we have to solve the following problems:

- Detect and emulate privileged instructions and some non-privileged instructions.
- Redirect system calls and page faults to the user-level OS.
- Emulate Memory Management Unit (MMU).
- Emulate essential peripheral devices.

3.1 Detecting execution of privileged instructions and some non-privileged instructions

Although an operating system kernel includes privileged instructions, most parts are built from non-privileged instructions. If we can prepare an appropriate address space for the kernel, we can execute the most parts of the kernel by the real CPU, directly. The rest of the tasks are emulation of hardware and execution of privileged instructions. In this case, we have to emulate only a small number of CPU instructions and peripheral devices because most CPU instructions are executed by the real CPU. We call this type of emulator *a partial emulator* or *a lightweight virtual machine.*

Figure 1 shows execution of a user-level OS by the real CPU, the host OS, and the partial emulator. The user-level OS and its own user processes are included in a regular process of the host OS. Regular processes of the host OS are served with the real CPU and the host OS. The real CPU executes their machine instructions, and the host OS handles system calls. In addition to the real CPU and the host OS, the user-level OS and its user processes are interpreted by the partial emulator. This partial emulator emulates privileged instructions and some peripheral devices, but it does not emulate normal instructions, such as arithmetic operations, load, store, and branch instructions. This is in contrast to *full emulators*, such as Bochs, which execute all CPU instructions.

A full emulator is independent of the underlying CPU, so it can execute machine instructions of another CPU. On the other hand, a partial emulator is dependent on the underlying CPU to directly execute machine instructions. Within this limitation, a partial emulator has an advantage over a full emulator in that application programs with no privileged instructions can run as fast as the real CPU. Furthermore, it is much easier to implement a partial emulator than to implement a full emulator.

The real problem on emulation is to detect the executions of some *non-privileged* instructions that are tightly coupled with corresponding privileged instructions. For example, `ltr` (load task register) of IA-32 is a privileged instruction while `str` (store task register) is a non-privileged instruction. We have to detect the executions of both `ltr` and `str`.

To detect execution of such non-privileged instructions, we use the following two methods:

insertion: Insert an illegal instruction statically before every non-privileged instruction to be detected.

rewriting: Rewrite the non-privileged and privileged instructions with subroutine calls that emulate these instructions.

3.1.1 Inserting an illegal instruction

In the insertion method, we insert an illegal instruction for each non-privileged instruction to be detected. The following is an example of insertion:

```
Before:
        str     %eax
After:
        .byte   0x8e, 0xc8
        str     %eax
```

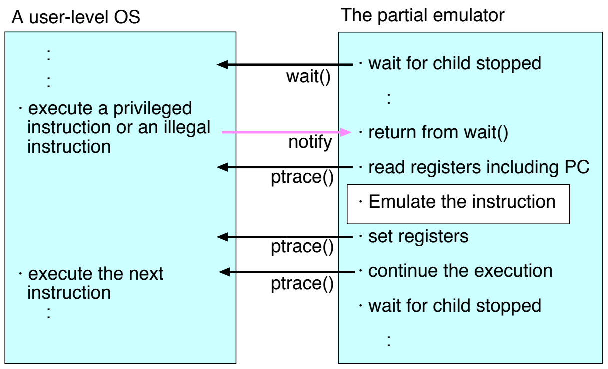

Figure 2: Execution of a privileged instruction or an illegal instruction by the partial emulator in the insertion method.

Since the byte sequence "`0x8e, 0xc8`" works as an illegal instruction in IA-32, we can detect the execution of the `str` instruction in the parent process of the user-level OS through the signal facility and the process trace facility. The parent process is the partial emulator.

Figure 2 shows execution of a privileged instruction or an inserted illegal instruction by the partial emulator. The partial emulator is a parent process of the user-level OS, and it is usually waiting for the child process being stopped with the `wait()` system call. When the user-level OS executes a privileged instruction or an inserted illegal instruction, the partial emulator is notified as if the child process received a signal (`SIGILL`). Next, the partial emulator reads the registers including the program counter with the `ptrace()` system call, and fetches the instruction pointed by the program counter. If the instruction is a privileged instruction, it is handled by the partial emulator. If the instruction is an inserted illegal instruction, the partial emulator fetches and handles the next instruction. After that, the partial emulator adjust the program counter by setting the registers with the `ptrace()` system call. Finally, the partial emulator continues the execution with the `ptrace()` system call, and is going to wait for the child process being stopped again. In summary, the partial emulator has to issue four system calls for each execution of a privileged or non-privileged instruction to be detected.

We have implemented a partial emulator for IA-32 based on the proposed method. The partial emulator consists of 1,500 lines of C code and 30 lines of assembly language code. Our partial emulator is much smaller than Bochs that consists of 50,000 lines of C++ code.

We have also implemented an assembler preprocessor for rewriting machine instructions. For IA-32, this preprocessor rewrites the following instructions:

- The `mov`, `push`, and `pop` instructions that manipulate segment registers (`%cs`, `%ds`, `%es`, `%fs`, `%gs`, and `%ss`).
- The `call`, `jmp`, and `ret` instructions that cross a segment boundary.
- The `iret` instruction.
- The instructions that manipulate special registers, such as the task register.
- The instructions that read and write the flag register.

3.1.2 Rewriting with subroutine calls

In the insertion method, the partial emulator has to issue four system calls for each execution of a privileged or non-privileged instruction to be detected, as described in Section 3.1.1. We eliminate this overhead by rewriting the privileged and non-privileged instructions with subroutine calls that emulate these instructions. An example of rewriting follows:

```
Before:
        mov     %eax, %cr3
After:
        call    mov_eax_cr3
```

The subroutine `mov_eax_cr3` performs emulation of the instruction. We also perform inlining for simple instructions. Our newer partial emulator consists of 800 lines of C code in the different address space, and 250 lines of C code and 300 lines of assembly language code in the same address space as a user-level OS.

3.2 Redirection of system calls and page faults

When a user process on a user-level OS issues a system call or causes a page fault, the host OS should not interpret the event by itself. Instead, the host

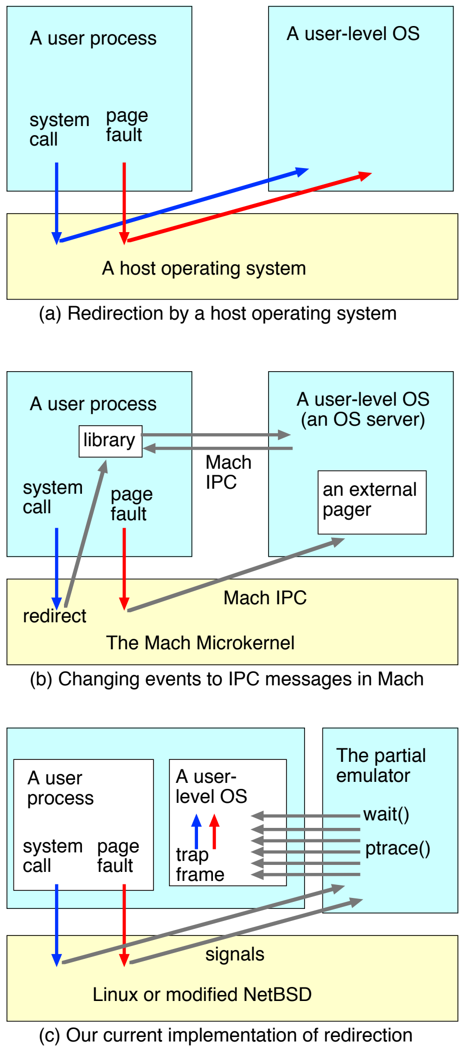

Figure 3: Redirection of system calls and page faults.

OS should notify the user-level OS of the event (a system call or page fault). If the host OS provides a redirection mechanism, this is an ideal mechanism for a user-level OS (Figure 3 (a)).

The Mach microkernel [GDFR90] includes a redirection mechanism of system calls for executing Unix binaries. When a user process (task) of Unix issues a system call, the Mach microkernel sends back a message to the same task (Figure 3 (b)). The user task includes the support module that receives the message from the microkernel, and performs an RPC to the remote Unix server. The Mach microkernel has a similar mechanism for handing page faults. This mechanism is called an external pager [GDFR90].

To implement the system call redirection, we use the process trace facility of Linux at first. In Linux, if a parent (or tracing) process specifies PTRACE_SYSCALL to the system call ptrace(), the child (or traced) process continues execution as for PTRACE_CONT[1], but it will stop on entry or exit of the system call.

When the child process is stopped, the parent process can examine and modify the CPU registers and arguments or results of the system call. Solaris and other Unix System V also have a similar facility through the /proc filesystem.

The procedure for redirecting system calls is similar to that for handing privileged and non-privileged instructions described in Section 3.1.1. When a user process of a user-level OS issues a system call, the process of the host OS is stopped on entry of the system call (Figure 3 (c)). The partial emulator changes the system call number with an illegal one and continues the execution. Next, the host OS tries to execute the body of the system call in a regular way. However, the host OS cannot execute it because the system call number is wrong. Therefore, the host OS sets an error value and notifies the partial emulator of exiting of the system call. Next, the partial emulator prepares an interrupt frame on the user-level OS. Finally, the partial emulator changes the program counter and switches to the user-level OS.

Page faults (SIGSEGV) are handled in a similar way as system calls. A difference is that the partial emulator has to send a signal to get values of some control registers when the host OS is Linux. In the partial emulator described in Section 3.1.2, the signals (SIGSEGV) are handled by the partial emulator code in the user-level OS. Therefore, we can reduce the overhead of context switches between the user-level OS and the partial emulator.

3.3 Emulation of Memory Management Unit

In IA-32, operations of MMU (Memory Management Unit) are performed thorough the register cr3 (Control Register 3). IA-32 uses two-level page tables. The register cr3 holds the physical address of the top level page table called *Page Directory* [Int97].

To emulate MMU and build address spaces for the user-level OS and its user processes, we use the system calls mmap() and munmap(). When the content of the register cr3 is changed, the partial emulator first compares each entry of the new page table with that of the previous page table. If a new page table entry no longer has a page, the partial emulator unmaps the corresponding page with the system call munmap(). If a new page table entry has a new page, the partial emulator maps the page with the system call mmap(). Otherwise, the partial emulator does nothing.

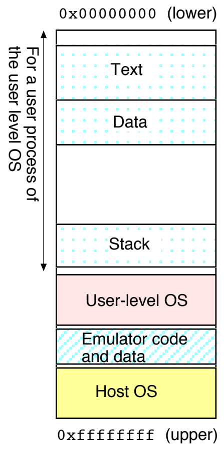

Figure 4: The address space of the user-level OS.

To emulate MMU with the system calls mmap() and munmap(), we have to solve the following problem in the insertion method (Section 3.1.1). These system calls should be invoked by the process of the user-level OS. These system calls manipulate the address space of the issuing process itself, and the parent or tracing process cannot manipulate the child or traced process.

To solve this problem, we embedded a support module in the address space of the user-level OS. Figure 4 shows the address space of the user-level OS. The lower regions of the address space are for user processes of the user-level OS (Text, Data, and Stack). The upper end is used by the host OS, and cannot be used by the user-level OS. Below the host OS region, there is a region for the user-level OS. We allocate a space for the mmap/munnap module between the host OS region and the user-level OS region.

When the partial emulator detects the manipulation of MMU (setting a value to the register cr3), the partial emulator compares the old and the new

[1]PT_CONTINUE in BSD.

page table. Based on the differences between the old and the new page table, the partial emulator makes an issuing plan of the system calls `mmap()` and `munmap()`, and stores the plan to the region of the emulator code and data in the address space of the user-level OS (Figure 4). After that, the partial emulator changes the program counter of the child process (the process executing the user-level OS) to the code of the partial emulator in the address space of user-level OS, and switches to the user-level OS. In the user-level OS, the emulator code issues the system calls `mmap()` and `munmap()` according to the plan. The partial emulator (the parent process) does not intercept these system calls and allows passing them to the host OS. Finally, the code executes a special instruction (`int $3`) to switch to the partial emulator. The partial emulator changes the program counter to the next instruction of the MMU operation, and continues execution.

We have described the procedure for the insertion method (Section 3.1.1). In the rewriting method (Section 3.1.2), MMU emulation is performed by the subroutines in the same address space as the user-level OS. The partial emulator in the separated address space does nothing about MMU emulation.

In both procedures, comparing the old and the new page table and issuing the system calls `mmap()` and `munmap()` are heavy tasks, and they are big sources of performance degradation. We will show the experimental results about MMU emulation in Section 6.

3.4 A console, a timer and a disk

The current partial emulator provides minimum peripheral devices: the keyboard for input, the video RAM for console output, and the timer for periodic interrupt.

For persistent storage, we have developed a small device driver that is derived from the memory disk driver of NetBSD. The memory disk of NetBSD is a block device, and it does not use interrupt for I/O. Instead, the memory disk reads and writes a memory reason in the kernel. We have changed these operations with the partial emulator calls (Section 3.1.1) or the system calls for the host OS (Section 3.1.2).

4 Modifying NetBSD for hosting the partial emulator

At first, we settled our goal to implement our partial emulator to run NetBSD on Linux. We chose NetBSD because it supports many hardware architectures, and each architecture-specific part seems small. We chose Linux because Linux already had User Mode Linux, so it was obvious that Linux has enough facilities for running a user-level OS. However, porting the i386-dependent part of NetBSD to the Linux architecture was not easy for us. Therefore, we have developed the techniques with partial emulation and rewriting of machine instructions.

After we had succeeded in running NetBSD on Linux, we started running NetBSD on NetBSD. We found that the unmodified NetBSD does not provide enough facilities to implement our partial emulator. The *ktrace* facility of NetBSD can be used to record events of entering and exiting of system calls. However, the ktrace facility does not allow stopping traced processes and changing registers and memory.

To run NetBSD on NetBSD, we decided to add a new facility to the host NetBSD kernel. The basic task is to introduce the `PTRACE_SYSCALL` facility of Linux to NetBSD. In the following subsections, we will show our modifications to NetBSD 1.6.1 in detail.

4.1 Modifications of the core part of NetBSD

We have modified the following three files in the core part of NetBSD (the architecture-independent part of NetBSD):

- `sys/proc.h`
- `sys/ptrace.h`
- `kern/sys_process.c`

We have added the following flag to the `struct proc` in `proc.h`:

```
#define P_SYSTRACED 0x800000
   /* System call is traced. */
```

We have added the following request for the system call ptrace() in the file ptrace.h:

```
#define PT_SYSCALL 12 /* Continue and stop
  at the next (return from) syscall. */
```

The flag and the request are used by the function sys_ptrace() of sys_process.c in the core part and the function syscall_plain() of syscall.c in the i386 architecture-specific part.

To the function sys_ptrace() in sys_process.c, we have added some lines for the request PT_SYSCALL next to the request PT_CONTINUE. The essential difference between PT_SYSCALL and PT_CONTINUE follows:

```
  if (SCARG(uap, req) == PT_SYSCALL) {
    SET(t->p_flag, P_SYSTRACED);
  } else {
    CLR(t->p_flag, P_SYSTRACED);
  }
```

If the request for the system call ptrace() is PT_SYSCALL, we set the flag P_SYSTRACED to the traced process. Otherwise, we clear the flag.

In addition to the above modification, we have defined a new function as follows:

```
process_systrace(p)
        struct proc *p;
```

This function is called from the entry point and the exit point of the system call when the flag P_SYSTRACED is set. This function stops the current process (the traced process) as if it would receive the signal SIGTRAP. If the tracing process is sleeping typically by issuing the system call wait(), the current process wakes up the tracing process.

4.2 Modifications of the i386-dependent part of NetBSD

We have modified the following three files in the i386-specific part of NetBSD:

- arch/i386/i386/syscall.c

```
void syscall_plain(frame)
  struct trapframe frame;
{
  ...
  p = curproc;
  if (ISSET(p->p_flag, P_SYSTRACED)) {
    CLR(p->p_flag, P_SYSTRACED);
    process_systrace(p);
  }
  code = frame.tf_eax;
  callp = p->p_emul->e_sysent;
  ...
  code &= (SYS_NSYSENT - 1);
  callp += code;
  ...
  error = (*callp->sy_call)(p, args, rval);
  ...
  if (ISSET(p->p_flag, P_SYSTRACED)) {
    CLR(p->p_flag, P_SYSTRACED);
    process_systrace(p);
  }
  userret(p);
}
```

Figure 5: Checking of the P_SYSTRACED flag on entry and exit of the system call.

- arch/i386/i386/process_machdep.c
- arch/i386/include/reg.h

We have inserted the invocation of the function process_systrace() to the function syscall_plain() in the file syscall.c, as shown in Figure 5. This function fetches the system call number in the register eax and calls the body of the system call by consulting the jump table in the p->p_emul->e_systent. Before getting the system call number from the register eax, we check if the flag P_SYSTRACED is set. If it is set, we call the function process_systrace() which is described in Section 4.1. The same function is also called at the end of the function before returning to the user mode.

Moreover, we have changed the files process_machdep.c and reg.h, and extended struct reg for the PT_GETREGS request of the system call ptrace(). In addition to regular registers for debugging, we need other values in control registers and the trap frame. For example, the register cr2 in PCB (Process Control Block) and the trap number are needed by the partial emulator. In Linux, we used a signal facility to get these values. If we set the program counter to an illegal address, the process receives a signal. At this time, the values that are needed by the partial emulator are pushed on the stack. In NetBSD, we did not use a signal facility. Instead, we extended struct reg in the request PT_GETREGS of the system call `ptrace()`.

5 Modifications to NetBSD and FreeBSD for running as user-level operating systems

In the implementation of a user-level OS, the final goal is to generate the user-level OS from the corresponding native OS for the bare hardware automatically. However, we had to slightly modify the native NetBSD and FreeBSD. In this section, we show the modifications to NetBSD and FreeBSD for running them as user-level OSes.

5.1 Modifications to NetBSD for running as a user process

We ran NetBSD 1.5.2 as a user process by changing 6 constants to adjust the address space and removing device drivers from the configuration file. The base address of NetBSD is changed from `0xc0000000` to `0xa000000` because the memory region after `0xc0000000` is occupied by the host operating system kernel.

Note that this modification is achieved without detailed knowledge about the NetBSD kernel. This is the significant difference from conventional user-level OSes, such as User Mode Linux. User Mode Linux requires adding a new architecture called *um*. The code size under the `um` directory is 33,000 lines, and this is comparable with the code size of the native i386 architecture (44,000 lines). This porting may cause a maintenance problem. When the native i386 architecture gets a new facility, the *um* architecture has to catch up the facility manually. In contrast, the core of our user-level NetBSD is automatically generated from the native i386 NetBSD. Therefore, we can follow the evolution of native i386 NetBSD more easily.

5.2 Modifications to FreeBSD for running as a user process

We have also executed the FreeBSD 4.7 kernel as a user process. In addition to address constants, we have changed the places that call BIOS. We have simply commented out the places and replaced with the code that returns parameters, such as the size of memory and the type of CPU. Since we did not have BIOS code, changing the FreeBSD kernel was much easier than implementing BIOS. Furthermore, changing the kernel reduces the effort to implement the partial emulator. Calling BIOS requires emulation of *the virtual 8086 mode* of Pentium. Our partial emulator does not have that facility. If we have BIOS code and a more powerful emulator, we do not have to modify these places.

6 Performance

We made experiments to measure the performance of our user-level OS (NetBSD 1.5.2). In this section, we show the results of microbenchmarks and an application benchmark.

All experiments were performed on a PC with a Pentium III 1GHz and 512M bytes of main memory. The host operating system for our partial emulator is Debian 3.0 with the Linux kernel 2.4.20 [2].

6.1 Microbenchmarks

As microbenchmarks, we use the following user programs:

`loop`: This program increments a variable in a loop. This program does not issue any system call during the experiments although the execution is interrupted by the timer.

`getpid`: This program issues the system call `getpid()`, repeatedly.

`pipesw`: This program creates two processes which are connected with two pipes. Each process writes and reads a byte for each step in a loop, so two context switches and four system calls are performed in a step.

`fork`: This program creates and terminates processes repeatedly. The parent process issues the system calls `fork()` and `wait()`, and the child processes issue the system call `exit()`.

[2] Currently, NetBSD on NetBSD is not stable enough to measure performance.

Table 1: The execution times of microbenchmark programs.

OS/Environment	program loop (n sec)	getpid (u sec)	pipesw (u sec)	fork (m sec)
NetBSD/PE-Insert/Linux	2.04	136	2880	89.9
NetBSD/PE-Rewrite/Linux	2.00	23.0	1030	19.0
NetBSD/Physical	1.99	0.360	19.8	0.380
NetBSD/Bochs/NetBSD	288	68.0	1600	34.9
NetBSD/VMware/Linux	2.01	3.53	83.7	2.550
Linux/Physical	1.99	0.299	5.53	0.114
User Mode Linux/Linux	1.99	44.1	665	31.7
Linux/Plex86/Linux	1.99	20.0	346	1.76

In these experiments, we measured the peak performance. In each execution, the task is repeated from 100 to 10,000,000 times. The number of trials is determined to be high enough to reach several tents of milliseconds to several seconds. The execution times were obtained using the system call `gettimeofday()` for the host OS, and divided by the number of iterations.

The result is shown in Table 1. In Table 1, "PE-" sands for our partial emulator. "Insert" means the insertion method (Section 3.1.1), and "Rewrite" means the rewriting method (Section 3.1.1), respectively. For reference, we include the results of the following operating systems and environments:

- NetBSD 1.6.1 on the physical PC
- NetBSD 1.5.2 on Bochs 2.02 on NetBSD 1.6.1
- NetBSD 1.5.2 on VMware 4.0 on Linux 2.4.20
- User Mode Linux 2.4.20-uml-6 on Linux 2.4.20
- Linux 2.4.20 on the physical PC
- Linux on Plex86 2003-02-16 on Linux 2.4.20 with NFS [3]

For the program `loop`, both of our partial emulators ("PE-Insert" and "PE-Rewrite") produced almost same performance as the physical PC because the measured part of the benchmark program is executed by the real CPU directly. By the same reason, our user-level NetBSD is faster than NetBSD on Bochs by a factor of 100.

[3]The current Plex86 uses a small RAM disk as a root device. We mounted the host file system with NFS and ran the benchmarks after changing the root directory to the host root.

In the cases of `getpid`, `pipesw` and `fork`, our partial emulator is slower than the physical machine by a factor of 100 and VMware by a factor of 10. This slowdown is cased by overheads of the system call redirection (Section 3.2) and the MMU emulation (Section 3.3). We got performance improvement by a factor of 2.8 to 5.9 between "PE-Insert" and "PE-Rewrite".

We cannot simply compare our partial emulator with User Mode Linux and Plex86 because the user-level OSes are different. As shown in Table 1, NetBSD/Physical is slower than Linux/Physical in those microbenchmarks. If we ignore the difference of user-level OSes, NetBSD on our partial emulator of "PE-Rewrite" is faster than User Mode Linux in the cases of `getpid` and `fork`. Our partial emulators are slower than Plex86 because Plex86 uses an efficient hardware mechanism called PVI, as described in Section 2.

6.2 An application benchmark

We ran the make command for compiling the GNU patch command (Version 2.5.4), and measured the execution times. The source code of the patch command consists of 15 C files and 17 headers. Total length of those files is 9200 lines or 244 k bytes [4]. The result is shown in Table 2.

NetBSDs on our partial emulators were faster than NetBSD on Bochs by a factor of 10. However, they were slower than NetBSD on the physical PC

[4]Although the source files are same on NetBSD and Linux, the header files in `/usr/include are` different. In this compilation, total 460 header files (2 M bytes) are included in NetBSD while 770 header files (6 M bytes) are included in Linux. Therefore, the execution time on NetBSD/Physical is shorter than that on Linux/Physical.

and VMware by a factor of 15 and 4, respectively. The ratios are smaller than the results of the microbenchmarks `getpid`, `pipesw`, and `fork` because this application benchmark includes a CPU workload. Our user-level NetBSD is slower than User Mode Linux and Linux/Plex86 because of the MMU emulation overhead.

7 Future directions

Our projects began on June 2002, and we have many tasks to be accomplished. Those tasks are classified into two categories:

- Adding new functions.
- Improving performance.

For each of functionality and performance, we have to choose or combine the following strategies:

- To modify the partial emulator.
- To modify host OSes.
- To modify user-level OSes.

The first strategy is best because it is independent of host and user-level OSes.

The first priority task on functionality is to add a networking facility. We are implementing a pseudo serial device for communicating between a user-level OS and a host OS. This serial device can be used for passing PPP packets. We also have a plan to implement an Ethernet-like device.

Table 2: The execution times of compilation in seconds.

OS/Environment	make (sec)
NetBSD/PE-Insert/Linux	52.5
NetBSD/PE-Rewrite/Linux	13.7
NetBSD/Physical	3.6
NetBSD/Bochs/Linux	550
NetBSD/VMware/Linux	3.9
Linux/Physical	4.1
User Mode Linux/Linux	9.5
Linux/Plex86/Linux	13.0

We are also interested in a function to access host file systems, as similar to *the host file system* of User Mode Linux. A straightforward implementation method is to insert a module to the VFS layer while we have to implement the module for each host OS. By using the networking facility, we can access host file systems through the NFS protocol and the SMB protocol.

As shown in Section 6, the main sources of overhead are the MMU emulation and the system call/page fault redirection.

To enhance the MMU emulation, we can cache address spaces as user processes of the host OS. As similar to the hardware context table of SPARC [SPA92], we can preserve and reuse user processes as page tables. In other words, the partial emulator forks when the MMU register of the page table gets a new value. If we cache page tables, we have to discard unused page tables or user processes of the host OS. If some LRU algorithm works well, we do not have to modify the user-level OS. Otherwise, we should modify the user-level OS to invalidate the cache on termination of its user processes.

We are also studying to introduce a new kernel level abstraction called *a virtual page table*. With this facility, a user-level OS can manipulate its page tables by issuing a new system call for the host OS. Unlike regular system calls, this system call for virtual page tables should depend on underlying hardware because we can preserve the structure and semantics of the base native OS. If we use a different facility, such as the external pager of the Mach microkernel, we have to change the base native OS more.

In the current implementation, the partial emulator does not handle the segment facility of IA-32 completely because most operating systems including NetBSD do not make use of the segment facility. The partial emulator interprets only address translation and write protection in the two-level page table. The partial emulator does not interpret other bits, such as Accessed and User/Supervisor for performance. Therefore, a user-level OS cannot perform page replacement efficiently. Moreover, the partial emulator does not change the memory protection on switching from the kernel mode to the user mode for performance reason. In IA-32, most operating systems including NetBSD do not change the MMU setting when the context is transferred from the user mode to the kernel mode or vice versa. Therefore, user processes can access the kernel memory. We would like to add a protection facility of the kernel memory after implementing the caching facility.

8 Conclusion

In this paper, we have proposed an implementation method of user-level operating systems based on partial emulation of PC hardware and rewriting of machine instructions at compile time. Unlike conventional methods, user-level operating systems are generated from the native operating systems. Therefore, no detailed knowledge about the native operating systems is needed to implement the user-level operating system. Based on the proposed method, we have executed NetBSD and FreeBSD kernels as user processes on Linux and NetBSD with small changes from the corresponding native systems.

The partial emulator on Linux can be used for running NetBSD and FreeBSD applications on Linux. We have nested operating system environments for NetBSD, so we can use several versions of NetBSD co-exist on the same platform.

We hope that our partial emulator will be one of the most popular tools for nested BSD operating systems. For developers, nested operating systems will be an essential facility to experiment with new kernels or new release while keeping the base environment safely. We are working on adding a networking facility to our partial emulator. With the networking facility, we can execute Internet servers on user-level operating systems, and we can couple user-level operating systems with the host operating system more tightly.

References

[AAS94] P. Ashton, D. Ayers, and P. Smith. SunOS Minix: a tool for use in operating system laboratories. *Australian Computer Science Communications*, 16(1):259–269, 1994.

[Dik00] Jeff Dike. A user-mode port of the linux kernel. In *the 4th Annual Linux Showcase & Conference*, 2000.

[FHL$^+$96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Operating Systems Design and Implementation*, pages 137–151, 1996.

[GDFR90] David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. UNIX as an application program. In *USENIX Summer*, pages 87–95, 1990.

[HH79] E. C. Hendricks and T. C. Hartmann. Evolution of a virtual machine subsystem. *IBM System Journal*, 18(1):111–142, 1979.

[HMM03] Brian N. Handy, Rich Murphey, and Jim Mock. *FreeBSD Handbook, Linux Binary Compatibility*, 2003.

[Int97] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 1997.

[Law03] Kevin P. Lawton. *The Plex86 x86 Virtual Machine Project*, 2003. `http://plex86.sourceforge.net/`.

[LDG$^+$03] Kevin Lawton, Bryce Denney, N. David Guarneri, Volker Ruppert, Christophe Bothamy, and Michael Calabrese. *Bochs x86 PC emulator Users Manual*, 2003. `http://bochs.sourceforge.net/`.

[LF03] Federico Lupi and The NetBSD Foundation. *The NetBSD Operating System, A Guide, Chapter 14. Linux emulation*, 2003.

[LW73] Hugh C. Lauer and David Wyeth. A recursive virtual machine architecture. In *Proceedings of the ACM SIGOPS/SIGARCH workshop on virtual computer systems*, pages 113–116, 1973.

[Pap00] A Connectix White Paper. *The Technology of Virtual PC*, 2000.

[SPA92] SPARC International. *The SPARC architecture manual: Version 8*. Prentice-Hall, 1992.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, 2001.

[Tad92] Yoshikatsu Tada. A virtual operating system VXinu — its implementation and problems. *Transactions of the Institute of Electronics, Information and Communication Engineers*, J75-D-1(1):10–18, 1992.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[Yok92] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 414–434, 1992.