System Call Inheritance

Yuanting Luo, Feifei Ji, and Junxing Yang
Stony Brook University

Abstract

From Linux 2.6, system call table cannot be exported and accessed by kernel loadable modules, which makes it difficult to override a system call. But system developers still want to provide additional functionality without recompiling the kernel. Some Unix systems, such as BSD, allows it by inheriting the system call table per process so that each process can have a entirely new set of system calls. In this paper, we use an idea similar to BSD implementations and provide a solution in Linux kernel 3.2.2. A kernel loadable module can now register its own system calls in customizable system call tables maintained by our helper module. A user program, with an additional system call table field in its task structure, can use this registered system call by setting its system call table to the corresponding one. APIs for those actions are also provided by the helper module. We present the design and implementation of the solution as well as its feasibility and performance.

1 Introduction

Unix systems implement most interfaces between User Mode processes and hardware devices by means of system calls issued to the kernel [2].A system call provides an interface for user level program to request a service from an operating system's kernel. There are many predefined system calls in Linux kernel, yet some processes would request system calls with customized features which are not supported by the kernel, such as encryption before writing to a file. Up until Linux kernel 2.4, the default system call table can be exported and accessed by kernel loadable modules so that modules can easily override system calls. But this flexiblity has been removed from the kernel since Linux 2.6, as a consideration for security issues. So developers need to change the kernel code and recompile the kernel to add or override system calls, which makes it hard to distribute one's own code and may cause some compatibility problems.

However, BSD allows users to have modules that can override or add a new system call to the kernel by inheriting the system call table per process, which means that each process can use a entirely new set of system calls in BSD. In this case, a process can change or add entries in its system call table with values of entries for some other functions provided by other modules so that new functionalities are easily deployed and used. We use an idea similar to BSD implementations and provide a solution in Linux kernel 3.2.2 [5].

In our framework, a helper module is plugged into the kernel, which maintains new system call table structures and provides APIs for a loadable module to register its own system calls and for a user process to use the registered system calls in the table. We use $syscall\_tables$ to refer to these system call table structures. A loadable module can now provide its own system call functions and write the entry addresses of the funtions to one of the $syscall\_tables$ or set the entry to 0 to unregister the system calls when it is removed.

We modify the task structure so that each process will have a new field in its $task\_struct$ which can be set to point to one of the $syscall\_tables$. A user-level program can now set the system call table that it wants to use before it invokes a system call. If it sets its system call table legally to one of the $syscall\_tables$ and invokes a system call that has been overridden and registered into that table by some loadable module, the new system call will be invoked instead of the one predefined in the kernel. This is done by modifying the interrupt service routine (ISR). The ISR now first tries to see if the current process's system call table is overridden by one of the $syscall\_tables$, then decides to call the system call from either $syscall\_tables$ or the system's default system call table.

The rest of this paper is organized as follows. Section 2 provides the background information of this paper, focusing on the details of invoking system calls. In Section 3, we present and explain the main design decisions that we have made. In Section 4, we discuss the details of our implementation of our framework. Section 5 contains the test results of our framework to show the feasibility and preformace. We summarize and discuss future work in Section 6.

2 Background

Our implementation is built on x86_32 architecture with Linux kernel 3.2.2.

A typical procedure of a system call in this system is as follows:

1. During the building of vmlinux, linker ld computes the entry address of a system call and store it in the default system call table;
2. A user program invokes a wrapper function for a

system call provided by GNU C library $glibc$;
3. $glibc$ finds the system call number($syscall\_no$), and a stub function will copy the parameters to registers or RAM. Especially, $syscall\_no$ is stored in $\%eax$;
4. The stub function interrupts CPU with INT 0x80;
5. ISR saves the registers using SAVE_ALL and stores them in the stack;
6. ISR calls $sys\_call\_table(syscall\_no)$, which jumps to the entry address of the system call;
7. The system call funcion will get paremeters from the stack, and ISR will get the return value and place it in the register, and then return to the stub function;
8. The stub function gets the return value and returns to user program.

The declaration of system calls uses a macro $asmlinkage$ which is defined as CPP_ASMLINKAGE __attribute__((regparm(0))). This indicates that the system calls get all parameters from the stack instead of registers. Thus, a loadable module needs to declare its own system call using the same macro to make sure the parameters can be passed correctly.

In some Unix implementations such as FreeBSD [3], each process has a $sysentvec$ structure in its $proc$ structure which is similar to $task\_struct$ in Linux). The $sysentvec$ structure contains system call dispatch infomation and has a pointer to a $sysent$ structure which contains the whole set of system calls that this process will use. The $sysentvec$ structure is copied during a fork so that the child process inherits the parent's system call table. The system has a default $sysent$ structure which contains the predefined system calls and is used by most processes, but a process can also set its own system call table. When a user program invokes a system call, the entry address of the system call will be found through its $sysentvec$ structure in its $proc$ structure. This feature enables a loadable modules to provide additional functionality by overriding system calls and a user program to use this functionality by setting its own system call table.

3 Design

The purpose of this project is to provide a framework that users can override the default system call table for a process. We can see from the syscall process that every time it will cause a software interrupt 0x80, and served by an ISR $system\_call()$ that will lead to the entry point of the syscall. In order to let the process call customized syscall, we must find a way to intercept syscalls somehow.

The approach we took is to intercept syscalls in the interrupt service routing. Since the new policy for system call must not interfere with the original calling process, we need to perform some checking before doing the real work. Instead of calling the default syscall table directly, the interrupt handler checks if the current process requests to use its own syscall table. If a syscall table is loaded by that process and the requested function is provided by a module, the interrupt handler will find the address of the new syscall. Otherwise the default syscall will be used. The process of intercepting a system call is depicted in Figure 1.

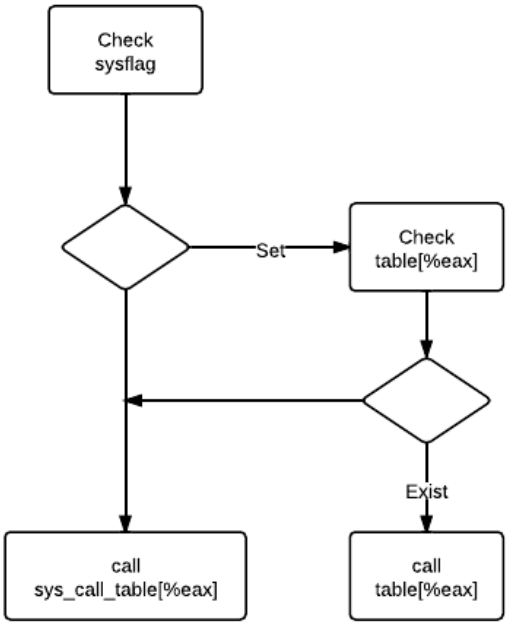

Figure 1: Intercepting a system call

Information about syscall table needs to be stored in memory and managed properly. In our framework, we define a new struct $syscall\_t$ which stores syscall table, vector name, reference count and a spinlock that protects reference count. We define syscall table as an array, and all its elements are initialized to 0 meaning it is not overridden. Default syscalls will be called when 0 is encounter in the table.

We also propose a register module to manipulate all syscall tables in system and provide access for system call modules and processes. Register module exports registering and setting interface, so a new syscall module could register itself to a syscall table, and a process could set which syscall table it needs to use.

The register module provides register API to new syscall modules that want to override existing functions. Table name, syscall number and overriding function need to be passed to register API. Table name specifies which syscall vector it wants to use, since register module maintains several syscall vectors. Syscall number specifies which syscall it will override. Every syscall has a unique number, so it is enough to just pass the number. The function passed to API will take the place of table[num] after registration finished. Register API is usually called when

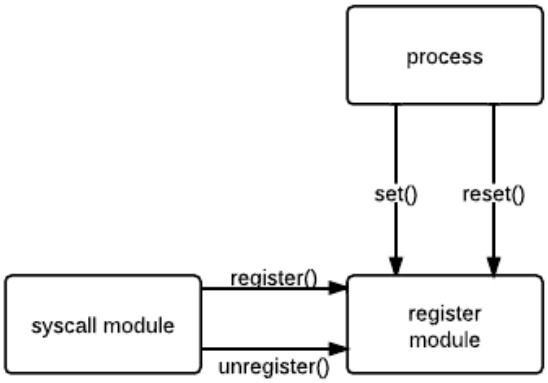

Figure 2: Register module APIs

a syscall module is loaded.

If a syscall wants to unregister itself, it simply sets the function address to 0, so the entry address of syscall will be reset to 0, and default syscall will be used when this system call is invoked.

We propose a new system call $set\_syscall\_table$ for the register module so that running processes could set its syscall table by invoking it. Table name and a control flag must be passed to the register module. Table name is used to specify which syscall table to use; Control flag tells $set\_syscall\_table$ either to use the syscall vector or stop using it.

Since we are trying to provide new system calls at a per-process level, we add $sysflag$ and $syscall\_table$ to $task\_struct$. $sysflag$ is used to indicate if current process is using a overridden syscall table. And $syscall\_table$ is a pointer that points to the syscall table which the current process is using.

4 Implementation

We implement a new syscall handler and a register module on Linux 3.3.2 kernel for our syscall framework. Syscall modules will register itself to the syscall table we maintain, and a process could set which syscall it wants to use. For every system call, the kernel jumps to the syscall handler that we modified, check the status of current process and find the address of the system call function.

4.1 Syscall handler

[1] describe details about system call interrupt handler. Since we are implementing a per process level syscall, every $task\_struct$ maintains a $syscall\_t$ struct it is using, and we need to call the functions in that table instead of the default sys_call_table. Kernel provides a macro GET_THREAD_INFO to get the $thread\_info$ struct of current process. Figure 3 illustrates how we find the entry address of a syscall. We can see from the figure that $thread\_info$ has a field struct $task\_struct *task$ which points to the $task\_struct$ of the current process. GET_THREAD_INFO($\%ebp$) will put the start address of struct $thread\_info$ which is $*task$ into ebp register. We use indirect address mode to find the actual memory that store $*task$, then check if sysflag is set. If current process is using a customized syscall, then the ebp register first moves down four bytes, and use the same address mode to find the start address of the syscall table. We call the function by call ($\%ebp, \%eax, 4$). eax stores the syscall number and the address of every system call takes 4 bytes. Therefore the address of the syscall we want is the start address of the syscall table plus offset of $4 * \%eax$.

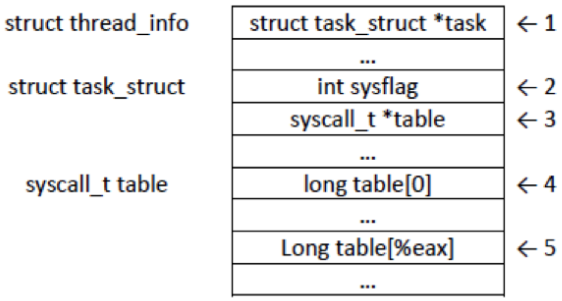

Figure 3: Entry address of system calls

4.2 Parameter Passing

The parameters of a syscall are stored in edi, esi, edx, ecx, and ebx. Before calling syscall interrupt handler, the current state of a process needs to be saved, and all registers are pushed to the stack.

4.3 Register Module

Register module exports two functions to user.

int register_syscall(char $*tname$, int num, long fn);

int set_syscall_table(char $*tname$, int flag);

$register\_syscall$ is used by a syscall loadable module to add a new system call to the internal list. We do not provide another function for unregister, since the process is basically the same. A syscall loadable module could simply set fn to 0 when it is unloaded. Register module helps maintain a list of $syscall\_t$ struct, and all structs are not used when it is initialized. When a new syscall loadable module tries to register to a certain syscall table, we go through the whole list and find the syscall ta-

ble requested by the syscall module. In the case when no $syscall\_t$ struct is found by the name pass to register module, then an unused $syscall\_t$ struct will be arranged for register.

$set\_syscall\_table$ is a new system call we add to kernel that provides syscall table setting to running processes. $flag$ is set to 1 for using a syscall table, and set to 0 for stop using it. Figure 4 illustrates how different components in the system interact with each other.

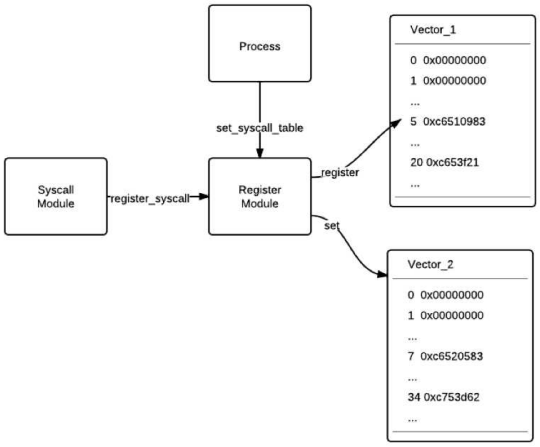

Figure 4: Interacting between components in the system

4.4 Reference count

The reference count in every $syscall\_t$ struct is used to count how many times a vector is used by processes. Reference count increases every time a process calls $set\_syscall\_table$ and decrease every time a process exits. When reference count decreases to 0, the corresponding $syscall\_t$ will be marked inactive, so it can be reused by other syscall modules.

There is a special case we need to handle with reference count. Since we allow a process to set its syscall table when it is running, a process can also decide to use another syscall table. In this case, when a process call $set\_syscall\_table$, old table will be released first, and $syscall\_t$ struct will be set to the requested syscall table.

5 Evaluation

Functionality test

We write 3 test modules to register syscalls **getpid**, **getuid**, **open** to a new syscall table(table name vector_1).

We changed the syscalls slightly so that we can see their differences with the default syscalls. When we invoke a syscall, interrupt will be disabled. So we cannot invoke a syscall in a syscall, which means we have to *reimplement* the new syscall.

For example, in order to test the functionality of the new **open** syscall, we exported function **do_sys_open** in **fs/open.c**.

Also we tested the case when there are more than one customized syscall tables.

```
// add 1 to the original uid
asmlinkage int getuid2(void) {
    return current_uid() + 1;
}
// add 1 to the original pid
asmlinkage int getpid2(void) {
    return current->pid() + 1;
}
//printk some file information before actually open file
asmlinkage int open2(const char* filename, int flags, int
mode) {
    long ret = 0;
    printk("Calling open2");
    printk("filename:%s,flag:%d,mode:%d",filename,flags,mode);
    // do sys_sys_open file
}
```

These three customized syscalls worked correctly. open2 shows that syscalls with parameters work properly.

We also tested fork to see that child process does inherit the syscall table from its father process. It worked correctly.

Stability

LTP was used to test the stability of our system. Since we only change the syscall module significantly, we run LTP only for the syscall test.

Overhead

We used a performance analysis tool lmbench [4] to evaluate the overhead of our system over vanilla kernel. We run the test in the virtual machine. for vanilla kernel, we used the kernel from homework1. Due to the instability of the load of the server, there are some noise in the tests. But in general the overhead is negligible. Below is the table and chart for one of test results.

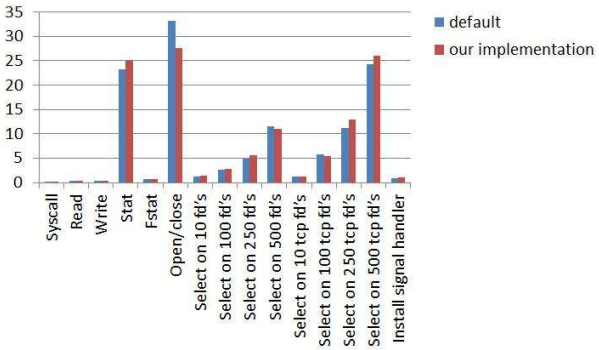

Figure 5: Evaluation

Operations	Default	Inherit
Syscall	0.2315	0.2472
Read	0.3511	0.4043
Write	0.3010	0.3434
Stat	23.2901	25.0170
Fstat	0.6173	0.6647
Open/close	33.1503	27.6306
Select on 10 fds	1.2588	1.3880
Select on 100 fds	2.6006	2.8415
Select on 250 fds	4.9112	5.6030
Select on 500 fds	11.4812	10.9898
Select on 10 tcp fds	1.2585	1.3094
Select on 100 tcp fds	5.7925	5.3705
Select on 250 tcp fds	11.1194	12.9255
Select on 500 tcp fds	24.3534	25.9720
Install signal handler	0.8358	1.0086

Table 1: Evaluation

6 Conclusions

We propose a framework that maintains several syscall tables and provides a way for users to override system calls with loadable modules. Thus a running process could set and change which syscall table it wants to use. We export two API functions to user level, so users could easily override default system call by loadable modules without recompiling the whole kernel. And the evaluation shows that the overhead with our new syscall interrupt handler is negligible.

Future Work. System call tables are maintained by our register module and all structures are stored in memory. This design requires that users have a very clear idea what they have done with system call tables. A better design could be exporting syscall information to users, and keep syscall tables on disk.

The syscall table structure is declared as an array in our system, its size is static in our framework. If more syscall tables are required, the framework should be able to resize the array.

Security is also a concern in future design. Since we do not perform any permission check when a process is setting a syscall table, a malicious program could easily crush the kernel by using some other syscall tables. Therefore, permissions should be associated with every syscall table and can only be used by process with a right permission.

References

[1] Intel Corporation. Intel 80386 programmer's reference manual.

[2] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel*. O'Reilly, 2005.

[3] Freebsd. *http://www.freebsd.org/*.

[4] Carl Staelin Larry McVoy. Lmbench - tools for performance analysis. *http://www.bitmover.com/lmbench/*.

[5] Linux Cross Reference. *http://lxr.linux.no/linux+v3.2.2/*.