HW #4. Interaction between Process and Kernel

Overview

Modern operating systems such as Windows and Linux are structured into two spaces: user space and kernel space. Most of the operating system functions are implemented in the kernel. Programs in the user space have to use appropriate system calls to invoke the corresponding kernel functions. In this homework, we will take a closer look at the system call mechanism by tracing system calls made by a user process calls. We will then demonstrate how to implement a new system call on Fedora Linux. We will also demonstrate how to copy data from kernel space to user space and vice versa.

Tasks

A. Use 'strace' to trace the system calls made by the 'ls' command

1. Use 'strace'

```
$ strace ls 2>& strace.txt
```

2. Open/Cat the output file 'strace.txt' (e.g. Figure 1)

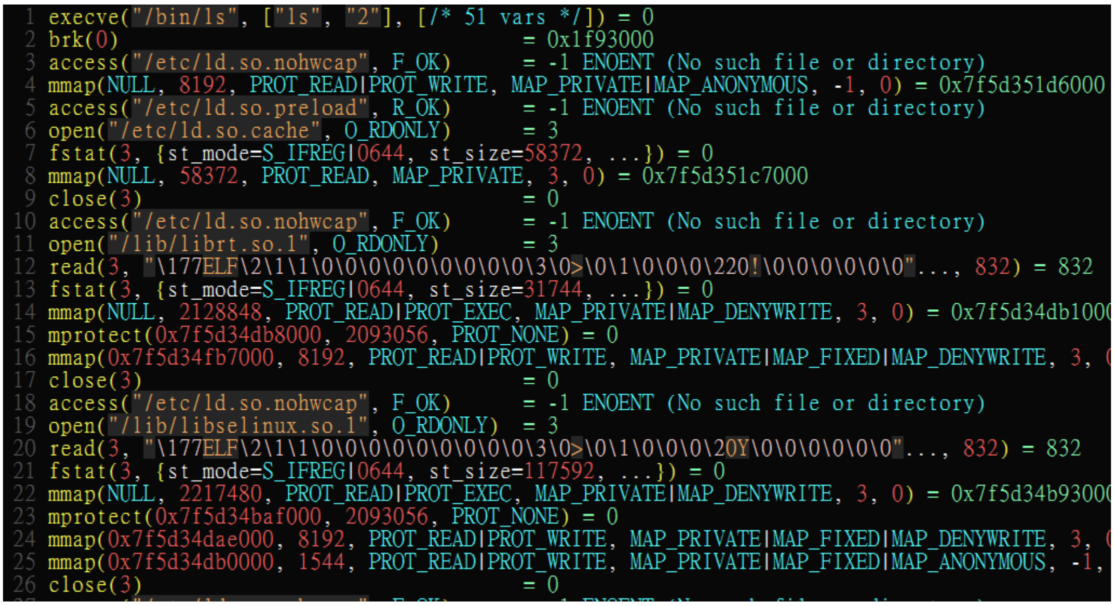


Figure 1. screenshot of strace command

3. You can see all the system calls made by the ls command in sequential

order. For instance, in Figure 1, we can see that the ls command has invoked the execve, brk, access, and mmap system calls

B. Add a custom system call

1. Download the kernel source (same steps as in Homework 2)
2. Add a custom system call to the syscall table (see Figure 2)

```
$ vim [source code directory]/arch/x86/syscall/syscall_64.tbl
```

```
304 common  open_by_handle_at     sys_open_by_handle_at
305 common  clock_adjtime         sys_clock_adjtime
306 common  syncfs                sys_syncfs
307 64  sendmmsg              sys_sendmmsg
308 common  setns                 sys_setns
309 common  getcpu                sys_getcpu
310 64  process_vm_readv      sys_process_vm_readv
311 64  process_vm_writev     sys_process_vm_writev

# simple system call
312 common  sayhello              sys_sayhello

#
# x32-specific system call numbers start at 512 to avoic
# for native 64-bit operation.
#
512 x32 rt_sigaction          sys32_rt_sigaction
513 x32 rt_sigreturn          stub_x32_rt_sigreturn
514 x32 ioctl             compat_sys_ioctl
```

Figure 2. add a system call 'sayhello' to syscall table

3. Add the system call definition to the syscall interface (see Figure 3)

```
$ vim [source code directory]/include/linux/syscalls.h
```

```
                              unsigned long
                              unsigned long

// simple system call
asmlinkage long sys_sayhello(void);

#endif
```

Figure 3. add the system call 'sayhello' definition to the syscall interface

4. Implement the custom system call (see Figure 4)

```
$ vim [source code directory]/kernel/sayhello.c
```

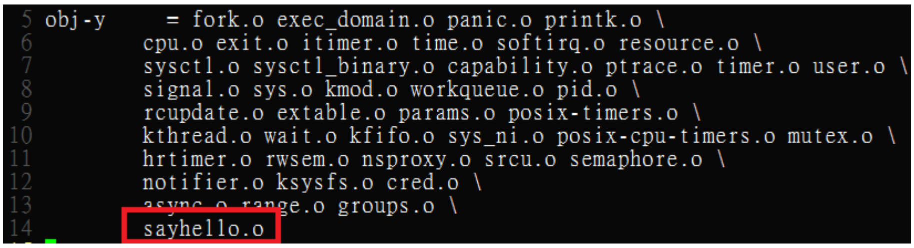
```c
#include <linux/kernel.h>

asmlinkage long sys_sayhello(void) {
    printk(KERN_DEBUG "Hello !\n");
    return 0;
}
```

Figure 4. the system call 'sayhello'

5. Modify the Makefile (e.g. Figure 5)

```
$ vim [source code directory]/kernel/Makefile
```



```
obj-y     = fork.o exec_domain.o panic.o printk.o \
            cpu.o exit.o itimer.o time.o softirq.o resource.o \
            sysctl.o sysctl_binary.o capability.o ptrace.o timer.o user.o \
            signal.o sys.o kmod.o workqueue.o pid.o \
            rcupdate.o extable.o params.o posix-timers.o \
            kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o \
            hrtimer.o rwsem.o nsproxy.o srcu.o semaphore.o \
            notifier.o ksysfs.o cred.o \
            async.o range.o groups.o \
            sayhello.o
```

Figure 5. modify the Makefile

6. Make the new kernel (steps like homework 2)

- For a multi-core PC, you can accelerate the kernel make process with the '-j [number of threads]' option.

```
$ make -j 4
```

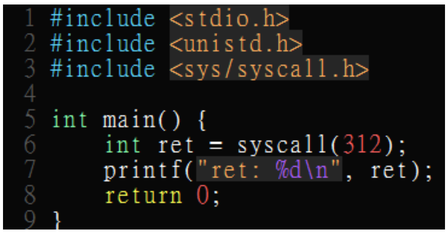
C. Invoke system call by the system all number (see Figure 6)

1. Include the needed libraries

```
#include <unistd.h>
#include <sys/syscall.h>
```

2. Use function 'syscall'

```
Usage: syscall(int [syscall number], [parameters to syscall])
```



```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main() {
    int ret = syscall(312);
    printf("ret: %d\n", ret);
    return 0;
}
```

Figure 6. call a system call in a program

- For detailed information of syscall, please check Linux man pages

```
$ man syscall
```

3. After running the code, you can use 'dmesg' to see the messages output from printk (e.g. Figure 7)

```
$ dmesg
```

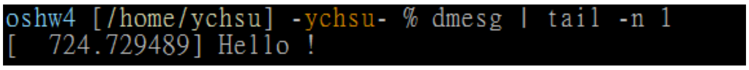


Figure 7. the 'printk' messages from 'sayhello' system call

※ You can download the full source code of the examples in the section B and section C here.

D. Copy data between user space and kernel space (Figure 8)

1. Include the header for XXX

```
#include <linux/uaccess.h>
```

2. copy_from_user
   'copy_from_user' is used to copy user space data to a kernel space buffer
   It is defined at '[source code directory]/include/asm-generic/uaccess.h'

```
Usage: copy_from_user (void* dst, void* src, unsigned long len)
```

3. copy_to_user
   'copy_to_user' is used to copy kernel space data to a user space buffer
   It is defined at '[source code directory]/include/asm-generic/uaccess.h'

```
Usage: copy_to_user(void* dst, void* src, unsigned long len)
```

```c
#include <asm/uaccess.h>
#include <linux/kernel.h>
#include <linux/string.h>

#define BUF_SIZE 1000

asmlinkage long sys_sayhello2(char *ptr) {

    char name[BUF_SIZE];
    char buf[BUF_SIZE];
    unsigned long len;
    memset(name, 0, BUF_SIZE);
    memset(buf, 0, BUF_SIZE);

    len = strlen(ptr);
    if(copy_from_user(name, ptr, len)) {
        return -EFAULT;
    }
    printk(KERN_DEBUG "syscall get name: %s(len: %lu)\n", name, len);

    snprintf(buf, BUF_SIZE, "Hello, %s !", name);
    if(copy_to_user(ptr, buf, strlen(buf))) {
        return -EFAULT;
    }
    printk(KERN_DEBUG "%s\n", buf);

    return 0;
}
```

Figure 8. the 'sayhello2' system call uses 'copy_from_user' to get a name and uses 'copy_to_user' to store the hello message to user space buffer

※ You can download the full source code of the examples in the section D here.

E. Manipulate the task_struct (program control block) of a process

In Linux, each process has a data structure 'task_struct' to store its information(process id, process state, page table, etc.), and there is a global variable 'current' which points to the current process' task_struct'. As a result, you can get the current process's task_struct information easily through the *current* pointer.(e.g. see Figure 9 to get the current process's state)

```c
    long state = current->state;
    printk(KERN_DEBUG "get state: %ld\n", state);

    if(copy_to_user(dst, &state, sizeof(state))) {
        return -EFAULT;
    }
```

Figure 9. get process state

The 'task_struct' is defined in '[source code directory]/include/linux/sched.h' (e.g. Figure 10), you can trace the structure and know more about the process.

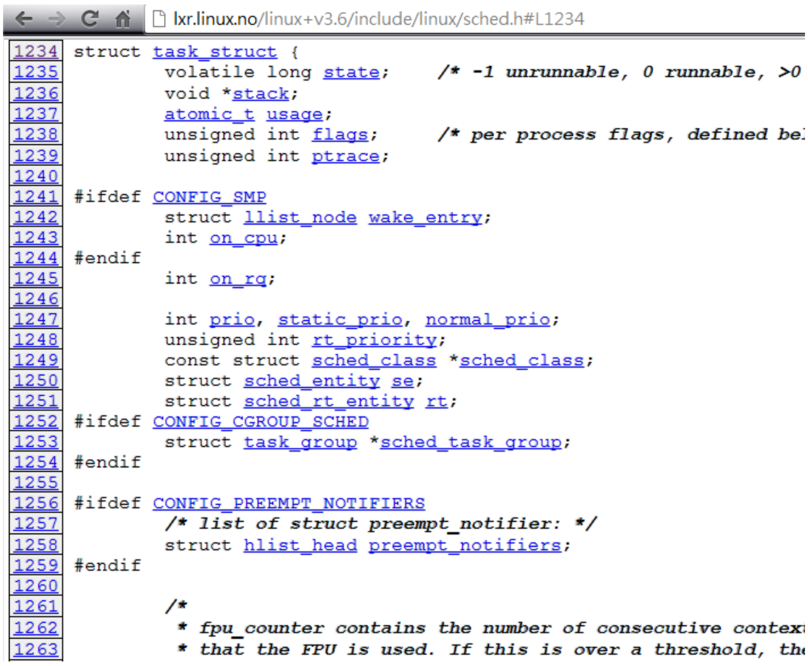


Figure 10. the definition of ‘task_struct’ in file ‘sched.h’

※ You can download the full source code of the examples in the section E here and the other similar example source code here.

F. Send a signal from kernel space to user space

- The kernel space

1. Include the required headers

```
#include <asm/siginfo.h>
#include <linux/sched.h>
```

4. Declare and initialize a signal structure

```
// declare a signal structure
struct siginfo info;
// initialization
memset(&info, 0, sizeof(struct siginfo));
info.si_signo = SIGUSR1;
info.si_code = SI_KERNEL;
```

The variable 'si_signo' is the signal number and 'si_code' presents how to send the signal, we set it as 'SI_KERNEL' to indicate that the signal is sent from the kernel.

5. Get the 'task_struct' of a process by process id

```
struct task_struct* task;
task = find_task_by_vpid(pid);
```

The 'find_task_by_vpid' function will return the process task structure with a given process id.

6. Send a signal to the process by its 'task_struct'

```
int ret = send_sig_info(SIGUSR1, &info, task);
```

The first parameter of send_sig_info is the signal number, the second parameter is a pointer to the signal structure, and the last parameter is a pointer to the specified task structure.

- The user space program

1. Include the required header

```
#include <signal.h>
```

2. Declare and initialize a signal handler structure

```
struct sigaction sig;
sig.sa_sigaction = receiveData;
sig.sa_flags = SA_SIGINFO;
```

The ‘sa_sigaction’ variable is the signal handler function and ‘sa_flag’ is the signal flags which modify the behavior of the signal.

3. Regist the signal handler when receive the specific signal

```
sigaction(SIGUSR1, &sig, NULL);
```

The first parameter is the signal number for the signal that the program is interested in, the second is the new sigaction structure pointer, and the last is the old sigaction structure pointer.

4. Define the signal handler function(e.g. function receiveData)

```
void receiveData(int signo, siginfo_t *info) {
        // do something when receive the signal
}
```

The first parameter is the received signal number, the second is the siginfo structure from the sender, it is an optional parameter.

※ You can download the full source code of the examples in the section F here.

- For more detailed information, you can use command ‘man 7 signal’

Homework Submission

Usually, we use the command ‘ps aux’ to get a list of the running processes on a system. By using the ‘ps aux’ command, we can see the command line strings of the running processes (e.g. the ‘./a.out’ in Figure 11). Internally, the command line string for each process is stored as a field in the process’s task_struct in the kernel.

```
oshw4 [/home/ychsu] -ychsu- % ps aux |grep a.out
ychsu     25789  0.0  0.0   4108   312 pts/1    S+   20:16   0:00 ./a.out
ychsu     26539  0.0  0.0 109400   920 pts/2    S+   20:16   0:00 grep --color a.out
```

Figure 11. program a.out is running

The memory address of the command line string can be acquired from the variable ‘arg_start’. Correspondingly, the variable ‘arg_end’ stores the memory address to the tail of the command line string. These two variables are kept in the ‘mm_struct’ structure (Figure 12), and the mm_struct structure appears in ‘task_struct’ as the variable ‘mm’ (Figure 13).
‘task_struct’ is defined at ‘[source code directory]/include/linux/sched.h’, and ‘mm_struct’ is defined at ‘[source code directory]/include/linux/mm_types.h’.

```
lxr.linux.no/linux+v3.5.4/include/linux/mm_types.h#L299

struct mm_struct {
        struct vm_area_struct * mmap;           /* list of VMAs */
        struct rb_root mm_rb;
        struct vm_area_struct * mmap_cache;     /* last find_vma result */
#ifdef CONFIG_MMU
        unsigned long (*get_unmapped_area) (struct file *filp,
                                unsigned long addr, unsigned long len,
                                unsigned long pgoff, unsigned long flags);
        void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
#endif
        unsigned long mmap_base;                /* base of mmap area */
        unsigned long task_size;                /* size of task vm space */
        unsigned long cached_hole_size;         /* if non-zero, the largest hole
        unsigned long free_area_cache;          /* first hole of size cached_hole_
        pgd_t * pgd;
        atomic_t mm_users;                      /* How many users with user space
        atomic_t mm_count;                      /* How many references to "struct
        int map_count;                          /* number of VMAs */

        spinlock_t page_table_lock;             /* Protects page tables and some
        struct rw_semaphore mmap_sem;

        struct list_head mmlist;                /* List of maybe swapped mm's.  T
                                                 * together off init_mm.mmlist, a
                                                 * by mmlist_lock
                                                 */


        unsigned long hiwater_rss;      /* High-watermark of RSS usage */
        unsigned long hiwater_vm;       /* High-water virtual memory usage */

        unsigned long total_vm;         /* Total pages mapped */
        unsigned long locked_vm;        /* Pages that have PG_mlocked set */
        unsigned long pinned_vm;        /* Refcount permanently increased */
        unsigned long shared_vm;        /* Shared pages (files) */
        unsigned long exec_vm;          /* VM_EXEC & ~VM_WRITE */
        unsigned long stack_vm;         /* VM_GROWSUP/DOWN */
        unsigned long reserved_vm;      /* VM_RESERVED|VM_IO pages */
        unsigned long def_flags;
        unsigned long nr_ptes;          /* Page table pages */
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
```

Figure 12. arg_start, arg_end in mm_struct

```
lxr.linux.no/linux+v3.6/include/linux/sched.h#L1234

#endif /* #ifdef CONFIG_TREE_PREEMPT_RCU */
#ifdef CONFIG_RCU_BOOST
        struct rt_mutex *rcu_boost_mutex;
#endif /* #ifdef CONFIG_RCU_BOOST */

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
        struct sched_info sched_info;
#endif

        struct list_head tasks;
#ifdef CONFIG_SMP
        struct plist_node pushable_tasks;
#endif

        struct mm_struct *mm, *active_mm;
#ifdef CONFIG_COMPAT_BRK
        unsigned brk_randomized:1;
#endif
#if defined(SPLIT_RSS_COUNTING)
        struct task_rss_stat    rss_stat;
#endif
/* task state */
        int exit_state;
```

Figure 13. mm in task_struct

From the section E, we know the usage of the 'current' variable. Now you can simply use the global variable 'current' and access the inner variable 'mm'. You can then get the command line string’s address, which is stored in the ‘arg_start’ variable After you locate the string, you can use copy_to_user function to copy the string to user space memory.

In this homework, you need to complete two tasks.

Task 1

You have to implement a custom system call to support two features: the first is to get the command line string of the calling process and copy it to a user space variable; the second is to modify the command line string of the calling process. For demonstration, you have to implement a user-space program to invoke the custom system call and show both of the features are working.
You must follow the steps described below.

At the start of your user-space program, you have to delay it and use ‘ps aux’ to show the original command line string as shown in Figure 14.
After calling your custom system call, your user space program must print out the original command line string like Figure 15 (the value must be the same as the one shown in Figure 14), and then you have to delay the program and use ‘ps aux’ again to confirm that the modified command line string like Figure 16, and then capture the screenshots for homework submission.

For example, you implement a system call ‘sys_change_cmdline’ and a user space program ‘change_cmdline’.
At first, execute ‘change_cmdline’, and use ‘ps aux’ before calling the system call ‘sys_change_cmdline’ to show the original command line string ‘./change_cmdline’(e.g. Figure 14)

```
oshw4 [/home/ychsu] -ychsu- % ps aux | grep change_cmdline
ychsu       953  0.0  0.0   4108   308 pts/0    S+   01:40   0:00 ./change_cmdline
ychsu       955  0.0  0.0 109400   904 pts/1    S+   01:40   0:00 grep --color change_cmdline
```

Figure 14. before the syscall, the command line of the ‘change_cmdline’ is ‘./change_cmdline’

After the system call ‘sys_change_cmdline’ is called, program ‘change_cmdline’ will print out the original command line string is ‘./change_cmdline’ (e.g. Figure 15)

```
oshw4 [/home/ychsu] -ychsu- % ./change_cmdline
call syscall 'sys_change_cmdline'
origin cmdline: ./change_cmdline
```

Figure 15. after the syscall, 'change_cmdline' will get and print out the original command line string is './change_cmdline'

And then use 'ps aux' to show the current command line string is 'Yen-Chun Hsu 0056021' (e.g. Figure 16)

```
oshw4 [/home/ychsu] -ychsu- % ps aux | grep 953
ychsu       953  0.0  0.0   4112   308 pts/0    S+   01:40   0:00 Yen-Chun Hsu 0056021
ychsu       966  0.0  0.0 109400   904 pts/1    S+   01:43   0:00 grep --color 953
```

Figure 16. after the syscall, the command line of 'change_cmdline' will become 'Yen-Chun Hsu 0056021'

Try to implement the custom system call, and then show the 'ps aux' result to demonstrate your work. Also, please briefly describe how you implement it.

※ For convenience, you can download the user space program prototype of this task here, and the system call program prototype here.

Task 2

Think about why we cannot just use memcpy, strcpy, etc. function to copy data from kernel space memory to user space memory?

Please archive your system call program, your user space program and the assignment document in PDF in an RAR file. Submit this RAR file to E3.

You can download all the example source code from github.

You can search and trace the kernel source code at lxr website.

It will take a long time to build the kernel, so you should start working on the homework early on.