CSC362 Operating System

Project 1 – System Call
Dr. Qixin Deng

Friday 14th February, 2025

System Call

Adding two system calls on Linux 0.11 and writing two simple application programs to test them.

1. Experiment Objectives

- Develop a deep understanding of system call interfaces.
- Master the basic process of system calls.
- Be able to exercise comprehensive control over system calls.
- Prepare for subsequent experiments.

2. Experiment Tasks

1. The first system call is **iam()**, and its prototype is:

```
1 int iam(const char * name);
```

   The completed functionality is to copy the content of the string parameter **name** into the kernel and save it. It is required that the length of **name** should not exceed 23 characters. The return value is the number of characters copied. If the number of characters in **name** exceeds 23, it returns "-1" and sets **errno** to **EINVAL**.

   This system call should be implemented in the **kernal/who.c** file.

2. The second system call is whoami(), and its prototype is:

```
1 int whoami(char* name, unsigned int size);
```

It copies the name saved in the kernel by **iam()** to the user address space pointed to by **name**, ensuring that there is no out-of-bounds memory access (the size of **name** is specified by **size**). The return value is the number of characters copied. If **size** is smaller than the required space, it returns "-1" and sets **errno** to **EINVAL**.

This system call should also be implemented in the **kernal/who.c** file.

3. Running Linux 0.11 with the newly added system calls, you would need to write two test programs, iam.c and whoami.c. The final expected results would be:

```
$ ./iam qixindeng
$ ./whoami
qixindeng
```

3. Instructions

1. Please restore the source code of Linux 0.11 to its original state. Section 5.5 of the A Heavily Commented Linux Kernel Source Code provides detailed information on how 0.11 handles system calls and is a valuable reference. The basic process of implementing system calls in an operating system is as follows:

   (a) An application program calls a library function (API).

   (b) The API places the system call number into EAX and transitions the system into kernel mode through an interrupt call.

   (c) In the kernel, the interrupt handler function, based on the system call number, invokes the corresponding kernel function (system call).

   (d) The system call performs the required functionality and stores the return value in EAX before returning to the interrupt handler.

   (e) The interrupt handler returns to the API.

   (f) The API returns the value in EAX to the application program

2. How applications invoke system calls:

   In general, invoking a system call and calling a regular custom function appear similar in code, but what happens after the invocation is quite different. When a custom function is called, it is done through a 'call' instruction that directly jumps to the function's address for continued execution. However, when a system call is invoked, an interface function written in the system library for that system call, known as an API (Application Programming Interface), is called. The API itself cannot perform the actual functionality of the system call. What it does is to invoke the real system call. The process involves:

   (a) Storing the system call number in the EAX register.

   (b) Placing function arguments into other general-purpose registers.

   (c) Triggering interrupt 0x80 (int 0x80)

   In the lib directory of version 0.11, there are some APIs that have already been implemented. Linus wrote them because, after the kernel is fully loaded, it switches to user mode to perform some initialization tasks, and then starts the shell. Many tasks in user mode depend on some system calls to be completed, so these APIs for system calls were implemented in the kernel. We can take a look at lib/close.c to study the API of close():

```
1 #define __LIBRARY__
2 #include <unistd.h>
3 _syscall1(int,close,int,fd)
```

   The _syscall1 is a macro defined in include/unistd.h. When the macro _syscall1(int, close, int, fd) is expanded, it can be resolved as follows:

```c
int close(int fd)
{
    long __res;
    __asm__ volatile ("int $0x80"
        : "=a" (__res)
        : "0" (__NR_close),"b" ((long)(fd)));
    if (__res >= 0)
        return (int) __res;
    errno = -__res;
    return -1;
}
```

This is the definition of the API. It first stores the macro __NR_close in the EAX register, stores the argument fd in the EBX register, and then executes the interrupt call 0x80. After the call returns, the return value is retrieved from EAX, stored in __res, and then a return value for the API caller is determined based on the evaluation of __res. Here, __NR_close is the system call number, which is defined in include/unistd.h:

```c
#define __NR_close    6
```

Therefore, when adding a system call, the file include/unistd.h needs to be modified to include __NR_whoami and __NR_iam. In the application, there should be:

```c
/* With it, _syscall1 and others become effective. See unistd.h for details.*/
#define __LIBRARY__
/* With it, the compiler can know the numbers of the custom system calls. */
#include <unistd.h>
/* iam() is the interface function in user space. */
_syscall1(int, iam, const char*, name);
/* whoami() is the interface function in user space. */
_syscall2(int, whoami,char*,name,unsigned int,size);
```

When compiling C programs in the Linux 0.11, all included header files are located in the /usr/include directory. The unistd.h in this directory is a standard header file (it is not the same file as the unistd.h in the 0.11 source code, although the content may be the same). It lacks the macros __NR_whoami and __NR_iam, so they need to be added manually. Alternatively, you can directly copy the modified unistd.h from the updated 0.11 source code.

3. Entering the kernel function from 'int 0x80'.

   After 'int 0x80' is triggered, what follows is the kernel's interrupt handling. First, let's understand the process of how 0.11 handles the 0x80 interrupt.

During kernel initialization, the main function (located in init/main.c, named main() in the Linux experimental environment) calls the sched_init() initialization function:

```
void main(void)
{
    ......
    time_init();
    sched_init();
    buffer_init(buffer_memory_end);
    ......
}
/*sched_init() is defined in kernel/sched.c as:*/

void sched_init(void)
{
    ......
    set_system_gate(0x80,&system_call);
}
/*set_system_gate, which is a macro defined in include/asm/system.h. */

#define set_system_gate(n,addr) \
    _set_gate(&idt[n],15,3,addr)

/*The definition of _set_gate is:*/

#define _set_gate(gate_addr,type,dpl,addr) \
__asm__ ("movw %%dx,%%ax\n\t" \
    "movw %0,%%dx\n\t" \
    "movl %%eax,%1\n\t" \
    "movl %%edx,%2" \
    : \
    : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
    "o" (*((char *) (gate_addr))), \
    "o" (*(4+(char *) (gate_addr))), \
    "d" ((char *) (addr)),"a" (0x00080000))
```

Although it seems quite complicated, it's actually quite simple. It involves populating the IDT (Interrupt Descriptor Table) by writing the address of the system_call function into the interrupt descriptor corresponding to 0x80. This means that after interrupt 0x80 occurs, the system_call function is automatically called. For specific details, please refer to Chapter 4 of the A Heavily Commented Linux Kernel Source Code.

Next, let's look at system_call. This function is purely crafted in assembly and is defined in kernel/system_call.s:

```
    ......
//This represents the total number of system calls.
//If any system calls are added or removed, corresponding modifications must be made.
nr_system_calls = 72
    ......
.globl system_call
.align 2
system_call:
    //It checks whether the system call number is within the legal range.
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    //'push %ebx, %ecx, %edx' passes arguments to the system call.
    pushl %edx
    pushl %ecx
    pushl %ebx
    //It sets ds, es to point to the GDT, which is the kernel address space.
    movl $0x10,%edx
    mov %dx,%ds
    mov %dx,%es
    //It sets fs to point to the LDT, which is the user address space
    movl $0x17,%edx
    mov %dx,%fs
    call sys_call_table(,%eax,4)
    pushl %eax
    movl current,%eax
    cmpl $0,state(%eax)
    jne reschedule
    cmpl $0,counter(%eax)
    je reschedule
```

system_call is made visible to other functions with the .globl modifier. Before 'call sys_call_table(,%eax,4)', there are some stack operations for protection and modifications of the segment selector to the kernel segment. After 'call sys_call_table(,%eax,4)', there is an assessment to see if there is a need for rescheduling. These aspects are not directly related to this experiment, so we will focus only on the 'call sys_call_table(,%eax,4)' part. According to assembly addressing methods, it actually translates to:

```
/*The EAX register contains the system call number, which is __NR_xxxxxx.*/
call sys_call_table + 4 * %eax
```

To understand this instruction:

(a) call: This is an assembly instruction used to call a function. In this context, it's used to call a function pointer located in a table (in this case, the sys_call_table).

(b) sys_call_table: This is the name of the system call table, which is an array of pointers to system call functions. Each entry in this table corresponds to a different system call.

(c) (,%eax,4): This part is a bit tricky as it's using x86 assembly syntax for indirect addressing.

(d) %eax is a register in the x86 architecture. Here, it's used to store the system call number. The value in %eax is used to index into the sys_call_table.

(e) The (,%eax,4) syntax is an example of scaled indexing addressing mode in x86 assembly. This mode allows you to access an array of elements where each element is a certain number of bytes. The general form is (base, index, scale). Here, base is omitted (hence the leading comma), %eax is the index, and 4 is the scale. The scale value of 4 is used because each pointer in the system call table is likely 4 bytes (32 bits) in size.

(f) So, the address calculated for the call instruction is the start of sys_call_table plus four times the value in %eax. This effectively jumps to the specific system call function that corresponds to the system call number stored in %eax.

Clearly, sys_call_table must be the starting address of an array of function pointers. It is defined in include/linux/sys.h:

```
/*The EAX register contains the system call number, which is __NR_xxxxxx.*/
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,......
```

To add the system calls required by the assignment, two function references – sys_iam and sys_whoami – need to be added to this function table. Of course, the position of these functions in the sys_call_table array must correspond to the value of __NR_xxxxxx. At the same time, following the format of the other system calls in this file, the following should be added:

```
extern int sys_whoami();
extern int sys_iam();
```

Otherwise, there will be compilation errors.

4. Implement sys_iam() and sys_whoami().

   The last step in adding a system call is to implement the functions sys_iam() and sys_whoami() in the kernel.

   Each system call has a corresponding sys_xxxxxx() function, and they are good examples for us to learn from and emulate. For example, in fs/open.c, there is sys_close(int fd):

```
int sys_close(unsigned int fd)
{
    ......
    return (0);
}
```

There's nothing particularly special about it; it simply performs the necessary actions for close(). So, as long as you create a file: kernel/who.c and implement the two functions, you'll be all set.

5. Modify Makefile.

   In order to make our added kernel/who.c compile and link with the rest of the Linux code, it's necessary to modify the Makefile file. The Makefile contains the compilation and linking rules for all source code files, and there's a brief introduction to it in A Heavily Commented Linux Kernel Source Code section 3.6. The reason we can simply run 'make' to compile the entire codebase is because 'make' operates according to the instructions in the Makefile.

   There are many Makefiles in the codebase, each responsible for compiling different modules. The one we need to modify is kernel/Makefile. There are two places that need to be modified. One of them is:

```
OBJS  = sched.o system_call.o traps.o asm.o fork.o \
        panic.o printk.o vsprintf.o sys.o exit.o \
        signal.o mktime.o
```

   Change it to:

```
OBJS  = sched.o system_call.o traps.o asm.o fork.o \
        panic.o printk.o vsprintf.o sys.o exit.o \
        signal.o mktime.o who.o
```

   Another place is:

```
### Dependencies:
exit.s exit.o: exit.c ../include/errno.h ../include/signal.h \
  ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
  ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
  ../include/asm/segment.h
```

   Change it to:

```
### Dependencies:
who.s who.o: who.c ../include/linux/kernel.h ../include/unistd.h
exit.s exit.o: exit.c ../include/errno.h ../include/signal.h \
  ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
  ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
  ../include/asm/segment.h
```

After modifying the Makefile, you can automatically include who.c into the kernel by running "make all" as usual. If you receive an error message indicating issues with who.c during compilation, it means the modifications have taken effect. So, intentionally or unintentionally introducing one or two errors isn't necessarily a bad thing; it can at least validate that the Makefile is correct.

6. Debugging the kernel using printk(). It's important to note that printf() is a function that can only be executed in user mode, whereas system calls run in kernel mode, so printf() is not available, and instead, you should use printk(). The interface and functionality of printk are essentially the same as printf, with just a few code-level differences. printk() requires special handling of the fs register, which is a segment register dedicated to user mode. Take a look at the code for printk (found in kernel/printk.c) to see this:

```
int printk(const char *fmt, ...)
{
    ......
    __asm__("push %%fs\n\t"
            "push %%ds\n\t"
            "pop %%fs\n\t"
            "pushl %0\n\t"
            "pushl $buf\n\t"
            "pushl $0\n\t"
            "call tty_write\n\t"
            "addl $8,%%esp\n\t"
            "popl %0\n\t"
            "pop %%fs"
            ::"r" (i):"ax","cx","dx");
    ......
}
```

Clearly, printk() first pushes %fs to save this register pointing to the user segment and later pops %fs to restore it. The core of printk still involves calling tty_write(). If you examine printf(), you'll see that it ultimately boils down to this function as well.

7. Write a test program.

Write a simple test application for testing purposes. For example, in sys_iam(), use printk() to print some information to the terminal. Have the application call iam(). The results will indicate whether the system call is actually being invoked.

You can write this program directly in the Linux 0.11 environment using vi (don't forget to regularly execute "sync" to ensure data in memory buffers is written to the disk), or you can write it on Ubuntu and then transfer it to Linux 0.11. Regardless of the approach, you must ultimately compile it in Linux 0.11. The compilation command is:

```
# gcc -o iam iam.c -Wall
```

The "-Wall" parameter in GCC provides all compilation warning messages, and the "-o" parameter specifies the name of the generated executable file as "iam." Run it using the following command:

```
# ./iam
```

If your information is successfully printed as expected, it means your added system call is effective. Otherwise, you'll need to continue debugging.

8. Passing data between user mode and kernel mode.

   Pointer parameters pass the logical addresses of the application's address space. In the kernel, if you directly access this address, you'll access data in the kernel space, not the user space. So, some special work is needed to obtain data from the user space within the kernel.

   The two system calls you want to implement have string pointers as parameters, similar to open(char *filename, ...). So, let's take a look at how the open() system call is handled.

```
int open(const char * filename, int flag, ...)
{
    ......
    __asm__("int $0x80"
            :"=a" (res)
            :"0" (__NR_open),"b" (filename),"c" (flag),
            "d" (va_arg(arg,int)));
    ......
}
```

It can be seen that system calls use the eax, ebx, ecx, and edx registers to pass parameters. Among them, eax passes the system call number, while ebx, ecx, and edx are used to pass the function's arguments. ebx corresponds to the first argument, ecx to the second argument, and so on. For example, the pointer to the file name passed by open is transmitted through ebx, which means that after entering the kernel, the file name string is extracted through ebx. The data

pointed to by open's ebx is in user space, while the code currently being executed is in kernel space. How is data passed between user mode and kernel mode? Let's continue to look at the handling of open.

```
int open(const char * filename, int flag, ...)
{
//All system calls start from system_call.
system_call:
    ......
    /* push %ebx, %ecx, %edx are the parameters passed to the system call.*/
    pushl %edx
    pushl %ecx
    pushl %ebx
    movl $0x10,%edx
    /*et ds and es to point to the Global Descriptor Table (GDT),
    which points to the kernel address space.*/
    mov %dx,%ds
    mov %dx,%es
    /*Set fs to point to the Local Descriptor Table (LDT),
    which points to the user address space.*/
    movl $0x17,%edx
    mov %dx,%fs
    //call sys_open
    call sys_call_table(,%eax,4)
}
```

From the above code, it can be seen that accessing data in the user address space (user data segment) relies on the fs segment register. Now, we will proceed to execute sys_open, which is located in the fs/open.c file.

```
//Where do these parameters like filename come from?
int sys_open(const char * filename,int flag,int mode)
/*Do you remember the pushl %edx, pushl %ecx, pushl %ebx that we saw earlier?
  In fact, when one C language function calls another C language function,
  during compilation, the parameters to be passed are pushed onto the stack
  (with the first parameter pushed last, and so on),
  and then a call ... instruction is used.
  So, when an assembly program calls a C function,
  it needs to manually write the code to push these parameters onto the stack.*/
{
    ......
    if ((i=open_namei(filename,flag,mode,&inode))<0) {
        ......
    }
    ......
}
```

It passes the parameters to open_namei(). Continuing the investigation within open_namei(), the filename is subsequently passed to dir_namei() and get_dir(). In get_dir(), you can observe:

```
static struct m_inode * get_dir(const char * pathname)
{
    ......
    if ((c=get_fs_byte(pathname))=='/') {
        ......
    }
    ......
}
```

The approach is quite evident: use get_fs_byte() to retrieve a byte of data from the user space. Therefore, when implementing iam(), you can simply call get_fs_byte(). But how can you implement whoami()? How can you copy data from kernel mode to user mode memory space? Take a guess, is there a put_fs_byte()? Yes, let's take a look at include/asm/segment.h:

```
static struct m_inode * get_dir(const char * pathname)
extern inline unsigned char get_fs_byte(const char * addr)
{
    unsigned register char _v;
    __asm__ ("movb %%fs:%1,%0":"=r" (_v):"m" (*addr));
    return _v;
}

extern inline void put_fs_byte(char val,char *addr)
{
    __asm__ ("movb %0,%%fs:%1"::"r" (val),"m" (*addr));
}
```

Both of them, as well as all put_fs_xxx() and get_fs_xxx() functions, serve as bridges between user space and kernel space, and they will be frequently used in subsequent assignments.

9. errno

   errno is a traditional mechanism for returning error codes. When a function call encounters an error, it typically returns -1 to the caller. However, -1 only indicates an error occurred but doesn't specify what the error is. To address this issue, the global variable errno was introduced. Error values are stored in errno, allowing the caller to determine how to handle the error by checking its value. Various systems have standardized definitions for the meaning of errno values. On Linux, you can view these definitions by using the "man errno" command.

Submission

1. The Linux 0.11 kernel you modified.(30%)
2. A detailed report. (70%) Inside the report, you need to finish three parts:
   (a) "Goal". Explain the tasks and the expectation. (10%)
   (b) "Implementation Details". You need to show every place that is added, deleted, or modified by you (C code). You need to give all of the intermediate results and final results and discuss them. (40%)
   (c) "Challenges". Put everything you think is challenging but not mentioned in the assignment document. Explain why it is challenging and how to overcome it.(20%)

One group needs to submit at least one copy to Canvas. **I do not accept late submissions.** Please put all group members' names in the report.

Academic Integrity

1. Collaborate honestly with your group members, ensuring that everyone contributes fairly to the project. Encourage and ensure equal participation from all group members, avoiding situations where one or two members do all the work.
2. Ensure that all sources used in the assignment are properly cited. This includes text, images, and ideas that are not your own.
3. Work should be original and not reused from past assignments, either your own or someone else's.
4. Use technology ethically, avoiding the temptation to use tools or resources in a manner not permitted by the professor, such as letting AI write code for the assignment.
5. Be aware of the consequences of violating academic integrity policies, which can range from grade penalties to more severe academic sanctions.