CSE 422S: Studio 4

Linux System Calls

Xingjian Xuanyuan

March 11, 2022

In this studio, we will:

1. *Make system calls with the* `libc` *wrapper.*
2. *Make system calls with the native Linux interface.*
3. *Write your own system calls.*

On a Linux machine, create a new file called `lib_call.c` under the directory for our own user space programs. In that file, write a short C program that (i) reads the user ID and prints it out, (ii) attempts to set it to zero (the root `uid`), (iii) indicates whether or not that attempt is successful, and then again (iv) reads and prints out the user ID.

```c
/* My lib_call.c program */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

int main()
{
    unsigned int real = getuid();
    printf("The current user ID is %u.\n", real);

    int set_return = setuid(0);
    if (set_return != 0) {
        printf("Error: setuid failed! Reason: %s.\n", strerror(errno));
    }

    unsigned int root = getuid();
    printf("The new user ID is %u.\n", root);

    return 0;
}
```

The `stdio.h`, `string.h`, and `errno.h` header files should be included for `printf`, `strerror`, and `errno` function calls respectively. The prototypes of the two system calls are given by:

```c
/* getuid() returns the real user ID of the calling process.
 * setuid() sets the effective user ID of the calling process.
 * In the GNU C Library, uid_t is an alias for unsigned int.
 */
```

```
#include <unistd.h>

uid_t getuid(void);
int setuid(uid_t uid);
```

According to The Open Group, if the process has appropriate privileges, `setuid()` shall set the real user ID, effective user ID, and the saved set-user-ID of the calling process to `uid`. If the process does not have appropriate privileges, but `uid` is equal to the real user ID or the saved set-user-ID, `setuid()` shall set the effective user ID to `uid`; the real user ID and saved set-user-ID shall remain unchanged. The term “appropriate privileges” has caused confusion among Unix systems. On Linux[1], it means that the process has `CAP_SETUID` and that `setuid(geteuid())` will fail if the effective user ID is not equal to 0, the real user ID, or the saved set-user-ID. For more details, please go visit CMU Software Engineering Institute website. The output is:

```
$ gcc lib_call.c -o lib_call
$ ./lib_call
The current user ID is 2014487.
Error: setuid failed! Reason: Operation not permitted.
The new user ID is 2014487.
```

Here, `setuid(0)` failed because the executable file is not owned by the root. Running the same executable file on Raspberry Pi 3 Model B+ produces:

```
$ ./lib_call
The current user ID is 1000.
Error: setuid failed! Reason: Operation not permitted.
The new user ID is 1000.
```

Note that the only difference between the two results is the user ID assigned to the running process.

When user space makes a system call, it passes a system call number, defined in `unistd.h`[2], as argument to the kernel. The kernel holds system call table, which contains the addresses of routines to be called based on the system call number. According to the Linux manual page, `syscall()` is a small library function that invokes the system call whose assembly language interface has the specified number with the specified arguments. Symbolic constants for system call numbers can be found in the header file `<sys/syscall.h>`. In `arch/arm/include/uapi/asm/unistd.h`, `__NR_SYSCALL_BASE` is defined as 0.

```
cd /project/scratch01/compile/x.xingjian/linux_source
cd linux/arch/arm/include/generated/uapi/asm/unistd-common.h
```

we can find four lines:

```
#define __NR_setuid (__NR_SYSCALL_BASE + 23)
#define __NR_getuid (__NR_SYSCALL_BASE + 24)
#define __NR_getuid32 (__NR_SYSCALL_BASE + 199)
#define __NR_setuid32 (__NR_SYSCALL_BASE + 213)
```

which means that the manifest constants for the two system calls actually equal to 23 and 24, respectively (for 16-bit, or 199 and 213 for 32-bit). The `native_call.c` program presented below replaces the calls to `getuid` and `setuid` with calls to `syscall`.

[1] Linux introduces a capability model for finer-grained control of privileges. Instead of a single level of privilege determined by the effective user ID, there are a number of capability bits each of which is used to determine access control to certain resources. Whenever the effective user ID becomes zero, the `SETUID` capability is set; whenever the effective user ID becomes non-zero, the `SETUID` capability is cleared. See Chen et al. (2002).

[2] To determine the ARM architecture-specific system call numbers, one should look at the Linux source file `arch/arm/include/uapi/asm/unistd.h` and the generated header file `arch/arm/include/generated/uapi/asm/unistd-common.h` in the kernel source directory.

```c
/* My native_call.c program */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <asm/unistd.h>

int main()
{
    unsigned int real = syscall(__NR_getuid);
    printf("The current user ID is %u.\n", real);

    int set_return = syscall(__NR_setuid);
    if (set_return != 0) {
        printf("Error: setuid failed! Reason: %s.\n", strerror(errno));
    }

    unsigned int root = syscall(__NR_getuid);
    printf("The new user ID is %u.\n", root);

    return 0;
}
```

Whenever it runs on the Linux PC desktop or Raspberry Pi, the output is exactly the same as `lib_call.c`.

There are five distinct tasks we need to accomplish to write new system calls into the Linux kernel:

a. Declare a C function prototype inside the kernel for each new syscall

b. Write a C function implementation for each new syscall

c. Define a new system call number for each new syscall

d. Update the ARM architecture system call table with each new syscall

e. Update the total number of syscalls value that stored by the kernel

In the `kernel/sys.c` section of `syscalls.h`, we can find

```c
asmlinkage long sys_setuid(uid_t uid);
```

And in the `kernel/timer.c` section of `syscalls.h`, we can find

```c
asmlinkage long sys_getuid(void);
```

Here, the `asmlinkage` tag is a `#define` for some gcc magic that tells the compiler that the function should not expect to find any of its arguments in registers, but only on the CPU's stack; it is also used to allow calling a function from assembly files. We can declare our own function prototypes, one that take no arguments and one that takes a single `int` argument, at the bottom of `include/linux/syscalls.h`:

```c
asmlinkage long sys_noargs(void);
asmlinkage long sys_onearg(int arg);
```

Inside `/include/linux/syscalls.h`, we may find the following lines of code:

```c
#ifndef SYSCALL_DEFINE0
#define SYSCALL_DEFINE0(sname)                              \
        SYSCALL_METADATA(_##sname, 0);                      \
        asmlinkage long sys_##sname(void);            \
        ALLOW_ERROR_INJECTION(sys_##sname, ERRNO);    \
        asmlinkage long sys_##sname(void)
#endif /* SYSCALL_DEFINE0 */
```

which define the SYSCALL_DEFINE0 macro. We can implement our own syscalls in arch/arm/kernel/:

```
/* sys_noargs(void) */
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/syscalls.h>

SYSCALL_DEFINE0( noargs ) {
    printk("Someone invoked the sys_noargs system call");
    return 0;
}

/* sys_onearg(int arg) */
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/syscalls.h>

SYSCALL_DEFINE1( onearg, int, arg ) {
    printk("Someone invoked the sys_onearg system call");
    return 0;
}
```

We now need to make sure the two new files are included in the Linux build process. Edit the file named Makefile in the arch/arm/kernel directory and add our two new files to the end of the object file list, which starts on the line with obj-y. Then, change the file extensions for your files from .c to .o. Finally, at the bottom of the arch/arm/tools/syscall.tbl file, create two new lines, following the scheme used throughout the file. Increment the value in the first column to allocate unique numbers for each of the new system calls. Note the format of the last two columns: the system call name, followed by the actual function definition, which by convention is the name prefixed with sys_.

Raspberry Pi 3 with its ARM Cortex-A53 processor supports various performance monitoring registers. For this final task, we would like to use a system call that allows a program to request the value of the Cycle Count Register (CCNT) from the kernel. First, we need a header file (perfmon.h as listed below) that defines architecture-specific functions for interacting with the Raspberry Pi's performance monitors.

```
/* perfmon.h */
#ifndef __ASMARM_ARCH_PERFMON_H
#define __ASMARM_ARCH_PERFMON_H

/*************************************************************************
*
* perfmon.h
*
* Provides functions and constant flags
* for interracting with the ARM Cortex-A53
* performance monitors in AARCH32 mode.
*
```

```
* This is the architecture in use on the Raspberry Pi 3 Model B series.
*
* Written October 9, 2020 by Marion Sudvarg
*
*****************************************************************************/

//PMCR: Performance Monitor Control Register

#define PMCR_ENABLE_COUNTERS (1 << 0)
#define PMCR_EVENT_COUNTER_RESET (1 << 1)
#define PMCR_CYCLE_COUNTER_RESET (1 << 2)
#define PMCR_CYCLE_COUNT_64 (1 << 3) //Increment cycle count every 64
↪  cycles
#define PMCR_EXPORT_ENABLE (1 << 4)
#define PMCR_CYCLE_COUNTER_DISABLE (1 << 5)
#define PMCR_CYCLE_COUNTER_64_BIT (1 << 6) //Cycle counter overflows at 64
↪  bits instead of 32


const unsigned PMCR_WRITE = PMCR_ENABLE_COUNTERS | PMCR_EVENT_COUNTER_RESET
↪  |  PMCR_CYCLE_COUNTER_RESET | PMCR_CYCLE_COUNT_64 | PMCR_EXPORT_ENABLE
↪  | PMCR_CYCLE_COUNTER_DISABLE | PMCR_CYCLE_COUNTER_64_BIT;
const unsigned PMCR_READ = PMCR_ENABLE_COUNTERS | PMCR_CYCLE_COUNT_64 |
↪  PMCR_EXPORT_ENABLE | PMCR_CYCLE_COUNTER_DISABLE |
↪  PMCR_CYCLE_COUNTER_64_BIT;

static inline void pmcr_set(unsigned x) {
        asm volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r" (x));
}

static inline char pmcr_set_check(unsigned x) {
        if ((x & PMCR_WRITE) == x) {
                pmcr_set(x);
                return 1;
        }
        return 0;
}

static inline unsigned long pmcr_get(void) {
        unsigned long x = 0;
        asm volatile ("MRC p15, 0, %0, c9, c12, 0\t\n" : "=r" (x));
        return x;
}

//CNTENS: Count Enable Set Register

const unsigned CNTENS_CTR0 = 1 << 0;
const unsigned CNTENS_CTR1 = 1 << 1;
const unsigned CNTENS_CTR2 = 1 << 2;
const unsigned CNTENS_CTR3 = 1 << 3;
const unsigned CNTENS_CYCLE_CTR = 1 << 31;

static inline void cntens_set(unsigned x) {
```

```c
        asm volatile ("MCR p15, 0, %0, c9, c12, 1\t\n" :: "r" (x));
}

static inline unsigned long cntens_get(void) {
        unsigned long x = 0;
        asm volatile ("MRC p15, 0, %0, c9, c12, 1\t\n" : "=r" (x));
        return x;
}

//PMCCNTR: Performance Monitors Cycle Count Register

static inline unsigned long long pmccntr_get(void) {
        register unsigned long pmcr = pmcr_get();
        if(pmcr & PMCR_CYCLE_COUNTER_64_BIT) {
                unsigned long low, high;
                asm volatile ("MRRC p15, 0, %0, %1, c9" : "=r" (low), "=r"
                 ↪  (high));
                return ( (unsigned long long) low) |
                                ( ( (unsigned long long) high ) << 32 );

        }
        else {
                unsigned long cycle_count;
                asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r"
                 ↪  (cycle_count));
                return (unsigned long long) cycle_count;
        }
}

static inline void pmcr_enable_pmccntr(char cycles_64_bit, char
 ↪  count_every_64) {
        unsigned long flags = pmcr_get();
        flags = flags | PMCR_ENABLE_COUNTERS | PMCR_CYCLE_COUNTER_RESET;
        if (cycles_64_bit) flags |= PMCR_CYCLE_COUNTER_64_BIT;
        else flags = flags & ~PMCR_CYCLE_COUNTER_64_BIT;
        if (count_every_64) flags |= PMCR_CYCLE_COUNT_64;
        else flags = flags & ~PMCR_CYCLE_COUNT_64;
        pmcr_set(flags);
}

static inline void cntens_enable_pmccntr(void) {
        unsigned long flags = cntens_get();
        flags = flags | CNTENS_CYCLE_CTR;
        cntens_set(CNTENS_CYCLE_CTR);
}

static inline void pmccntr_enable(char cycles_64_bit, char count_every_64)
 ↪  {
        pmcr_enable_pmccntr(cycles_64_bit, count_every_64);
        cntens_enable_pmccntr();
}

```

```
static inline void pmccntr_enable_user(char cycles_64_bit, char
↪  count_every_64) {
        pmccntr_enable(cycles_64_bit, count_every_64);
        asm volatile ("MCR p15, 0, %0, c9, c14, 0" :: "r" (1));
}

static inline void pmccntr_enable_once(char cycles_64_bit, char
↪  count_every_64) {
        unsigned long flags;
        flags = pmcr_get();
        if ( !(flags & PMCR_ENABLE_COUNTERS) )
         ↪  pmcr_enable_pmccntr(cycles_64_bit, count_every_64);;
        flags = cntens_get();
        if ( !(flags & CNTENS_CYCLE_CTR) ) cntens_enable_pmccntr();
}

static inline void pmccntr_enable_once_user(char cycles_64_bit, char
↪  count_every_64) {
        pmccntr_enable_once(cycles_64_bit, count_every_64);
        asm volatile ("MCR p15, 0, %0, c9, c14, 0" :: "r" (1));
}

static inline void pmccntr_reset(void) {
        unsigned long flags = pmcr_get();
        flags = flags | PMCR_CYCLE_COUNTER_RESET;
        pmcr_set(flags);
}

#endif //__ASMARM_ARCH_PERFMON_H
```

On the Linux machine, place this file in the `arch/arm/include/asm` directory in the Linux kernel source tree. Modify the `include/linux/syscalls.h` header to declare another function prototype, which takes a single `unsigned long long *` argument:

```
asmlinkage long sys_read_ccnt(unsigned long long * cc);
```

Inside the `arch/arm/kernel` directory create a new file `sys_read_ccnt.c` that implements the syscall:

```
/******************************************************************************
*
* sys_read_ccnt.c
*
* Implements a system call to retrieve the current
* cycle count on an ARM Cortex-A53 in AARCH32 mode.
*
* Takes a pointer to an unsigned long long as an argument.
* Writes the cycle count value to the pointer's location.
*
* Written October 9, 2020 by Marion Sudvarg
*
******************************************************************************/

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

```c
#include <linux/sched.h>
#include <linux/syscalls.h>
#include <asm/perfmon.h>

//Define a system call implementation that takes one argument
SYSCALL_DEFINE1( read_ccnt, unsigned long long *, cc ){

        //Enable cycle counter, if not yet enabled
        pmccntr_enable_once(1, 0);

        //Retrieve the cycle count
        *cc = pmccntr_get();

        return 0;
}

//End of file
```

Add this new file to the end of the object file list in `arch/arm/kernel/Makefile`. Add this new system call to `arcg/arm/tools/syscall.tbl`. At this point, the kernel is fully modified and can be recompiled. In the base `linux` source directory, issue the command `make clean` to remove the artifacts from your previous build. Then, to differentiate this new kernel from the previously built ones, edit the `.config` file, modify the `CONFIG_LOCALVERSION` string. Issue the following command to compile the kernel:

```
make -j8 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs
```

Then issue the command:

```
make -j8 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
 ↪  INSTALL_MOD_PATH=../modules modules_install
```

Navigate to the `linux_source` directory, create two archives with the following commands:

```
tar -C modules/lib -czf modules.tgz modules
tar -C linux/arch/arm -czf boot.tgz boot
```

Use `sftp` to transfer these file to the `linux_source` directory on the Raspberry Pi. Back up the directories `/usr/lib/modules` and `/boot`. Install the kernel just built:

```
tar -xzf modules.tgz
tar -xzf boot.tgz
cd modules
sudo cp -rd * /usr/lib/modules (or sudo cp -rd * /lib/modules if /usr/lib
 ↪  does not exist)
cd ..
sudo cp boot/dts/*.dtb /boot/
sudo cp boot/dts/overlays/*.dtb* /boot/overlays
sudo cp boot/dts/overlays/README /boot/overlays

## If using Raspberry Pi 3B+
sudo cp boot/zImage /boot/kernel7.img
## If using Raspberry Pi 4 or 4B
sudo cp boot/zImage /boot/kernel7l.img
```

After `reboot`, this new custom kernel will be running on the Raspberry Pi. Use `sftp` to retrieve `arch/arm/include/generated/uapi/asm/unistd-common.h` from the kernel source directory of the Linux machine. Right before `#endif`, there exists this new line corresponding to the new system call:

```
#define __NR_READ_CCNT (__NR_SYSCALL_BASE + 438)
```

Copy this file to the appropriate directory for GCC:

```
sudo cp unistd-common.h /usr/include/arm-linux-gnueabihf/asm
```

Now, let's write a simple program that makes the system call twice to read cycle counts in a row:

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <asm/unistd.h>

int main()
{
    unsigned long long count1, count2;

    syscall(__NR_read_ccnt, &count1);
    printf("Cycle count: %llu.\n", count1);

    syscall(__NR_read_ccnt, &count2);
    printf("Cycle count: %llu.\n", count2);

    return 0;
}
```

The output is:

```
Cycle count: 0.
Cycle count: 66376.
```

References

Chen, H., Wagner, D. & Dean, D. (2002), Setuid demystified, *in* 'Proceedings of the 11th USENIX Security Symposium', San Francisco.

Love, R. (2010), *Linux Kernel Development*, 3 edn, Addison-Wesley.