بسمه تعالی

گزارش تمرین عملی اول – سوال سوم – درس سیستم‌های عامل دکتر اسدی – نیمسال اول 405-404

نویسندگان:

1. سیداحمد موسوی‌اول – 402106648
2. عرفان تیموری – 402105813

گزارشکار.

قسمت اول

در این تمرین قصد داریم به هسته سیستم عامل خودمان فراخوانی های سیستمی اضافه کنیم . در این قسمت می‌خواهیم فراخوانی سیستمی اضافه کنیم که در لاگ کرنل **hello world** چاپ کند. برای این کار ابتدا در فایل **./kernel/sys.c**

یک فراخوانی سیستمی اضافه می‌کنیم.

در ادامه باید در فایل **./arch/x86/entry/syscalls/syscall_64.tbl** مطابق تمپلیت جدولی که دارد اضافه کنیم. توجه داریم که در ستون اول شماره سیسکال را اضافه می‌کنیم که اولین شماره آزاد آن 548 است . ستون دوم **common** نشان دهنده این است که این فراخوانی سیستمی برای همه برنامه های 64 بیتی قابل انجام است. در ستون های دیگر نیز نام فراخوانی سیستمی و همچنین نام آن به همراه **sys** در ستون بعدی اضافه کنیم:

```
548     common  my_hello                 sys_my_hello
549     common  set_my_field             sys_set_my_field
550     common  get_my_field             sys_get_my_field
551     common  get_time_ns              sys_get_time_ns
552     common  get_last_syscall_times   sys_get_last_syscall_times
# This is the end of the legacy x32 range.  Numbers 548 and above are
# not special and are not to be used for x32-specific syscalls.
```

در مرحله آخر در فایل **./include/linux/syscalls.h** پروتوتایپ فراخوانی سیستمی زده شده را قرار می‌دهیم :

```
asmlinkage long __x64_sys_my_hello(void);

asmlinkage long __x64_sys_set_my_field(long new_val);
asmlinkage long __x64_sys_get_my_field(void);
```

توجه داریم که چون در این فراخوانی سیستمی از **printk** به همراه **kernel_info** استفاده کردیم برای مشاهده پیام باید از **dmesg** استفاده کنیم :

```
osvm@Teymouri-Mossaviawal:~/Desktop$ nano test1.c
osvm@Teymouri-Mossaviawal:~/Desktop$ gcc test1.c -o test1
osvm@Teymouri-Mossaviawal:~/Desktop$ ./test1
System call returned 0
osvm@Teymouri-Mossaviawal:~/Desktop$ dmesg | tail
dmesg: read kernel buffer failed: Operation not permitted
osvm@Teymouri-Mossaviawal:~/Desktop$ sudo dmesg | tail
[sudo] password for osvm:
[    6.117265] iTCO_wdt iTCO_wdt.1.auto: initialized. heartbeat=30 sec (nowayout=0)
[    6.267947] snd_hda_codec_generic hdaudioC0D0: autoconfig for Generic: line_outs=1 (0x3/0x0/0x0/0x0/0x0) type:line
[    6.267952] snd_hda_codec_generic hdaudioC0D0:    speaker_outs=0 (0x0/0x0/0x0/0x0/0x0)
[    6.267955] snd_hda_codec_generic hdaudioC0D0:    hp_outs=0 (0x0/0x0/0x0/0x0/0x0)
[    6.267958] snd_hda_codec_generic hdaudioC0D0:    mono: mono_out=0x0
[    6.267959] snd_hda_codec_generic hdaudioC0D0:    inputs:
[    6.267961] snd_hda_codec_generic hdaudioC0D0:      Line=0x5
[    6.619106] NET: Registered PF_QIPCRTR protocol family
[    9.235189] rfkill: input handler disabled
[   95.881544] hello world
```

قسمت دوم :

در این قسمت می‌خواهیم پیام را به جای لاگ کرنل در **stdout** چاپ کنیم برای اینکار مطابق فراخوانی سیستمی **write** در هسته سیستم عامل خودمان عمل می‌کنیم. یعنی **syscall** را به صورت زیر تعریف می‌کنیم :

```
ers > Laptopkaran > Downloads > hello.c > SYSCALL_DEFINE0(my_hello)
SYSCALL_DEFINE0(my_hello)
{
    struct file *file;
    char msg[] = "Hello from kernel to stdout!\n";
    size_t msg_len = strlen(msg);
    long ret;
    loff_t pos = 0;

    file = fget(1);

    if (!file) {
        printk(KERN_WARNING "my_hello: Could not get file for stdout (fd 1)\n");
        return -EBADF;
    }

    if (!file->f_op->write && !file->f_op->write_iter) {
        printk(KERN_WARNING "my_hello: stdout does not have a write/write_iter operation\n");
        fput(file);
        return -EACCES;
    }

    ret = kernel_write(file, msg, msg_len, &pos);

    fput(file);

    if (ret < 0) {
        printk(KERN_WARNING "my_hello: kernel_write to stdout failed with error %ld\n", ret);
        return ret;
    }

    return ret;
}
```

این قطعه کد هسته لینوکس با هدف نوشتن یک پیام ثابت **hello from kernel to stdout** به خروجی استاندارد طراحی شده است. در ابتدا کد ساختار فایل مربوط به توصیفگر فایل **1** که در لینوکس معادل **stdout** است را با استفاده از **fget(1)** بازیابی می‌کند. پس از انجام بررسی های ضروری برای اطمینان از اینکه ساختار فایل معتبر باشد و عملیات نوشتن مجاز باشد تابع هسته ای **kernel_write** فراخوانی می‌شود تا رشته پیام را از فضای هسته به آن فایل بنویسد. در نهایت با استفاده از **fput(file)** منابع فایل ازاد شده و در صورت بروز خطا در حین نوشتن یک پیام هشدار با استفاده از **printk** در لاگ هسته ثبت و کد با کد خطای مربوطه برگردانده می‌شود. در غیر این صورت تعداد بایت های نوشته شده بازگردانده می‌شود.

در ادامه با استفاده از روشی که گفتیم فراخوانی سیستمی را اضافه می‌کنیم.

حال تابع تستی برای آن می‌نویسیم و فراخوانی سیستمی **549** را صدا می‌زنیم. خروجی به صورت زیر است.

قسمت سوم :

در این قسمت می‌خواهیم متغیر هایی به **pcb** یک پردازه اضافه کنیم. می‌دانیم **pcb** یک پردازه داده ساختاری است که در آن متغیرهای مناسب یک پردازه مانند تعداد فایل های باز یا **pid** آن ذخیره شده است. این داده ساختار در فایل **./include/linux/sched.h** به عنوان **task_struct** تعریف شده است. در این استراکت متغیر جدید خود را اضافه می‌کنیم :

```
long my_custom_field;
```

در ادامه فراخوانی سیستمی تعریف می‌کنیم که یکبار این متغیر را بخواند و در فراخوانی سیستمی دیگری متغیر ست شده را خروجی دهد.

```
SYSCALL_DEFINE1(set_my_field, long, new_val)
{
    current->my_custom_field = new_val;

    printk(KERN_INFO "PID %d set my_custom_field to %ld\n", current->pid, new_val);
    return 0;
}

SYSCALL_DEFINE0(get_my_field)
{
    long val = current->my_custom_field;

    printk(KERN_INFO "PID %d read my_custom_field: %ld\n", current->pid, val);
    return val;
}
```

pcb در سیستم‌عامل مخفف **Process Control Block**(بلوک کنترل فرآیند) است و مهم‌ترین ساختار داده‌ای است که هسته **(Kernel)** سیستم‌عامل برای مدیریت هر فرآیند **(Process)** در حال اجرا استفاده می‌کند. این بلوک شامل تمام اطلاعات ضروری مربوط به فرآیند، از جمله وضعیت جاری فرآیند در حال اجرا، آماده، منتظر، شماره شناسایی**(PID)** ، مقادیر ثبات‌های **CPU** و شمارنده برنامه (آدرس دستورالعمل بعدی)، و اطلاعات مدیریت حافظه آن است . **pcb** امکان عملیات حیاتی مانند زمان‌بندی **CPU** و تعویض زمینه **(Context Switching)** را فراهم می‌کند؛

به این صورت که هنگام جابجایی بین فرآیندها، اطلاعات فرآیند فعلی در **PCB** آن ذخیره و اطلاعات فرآیند جدید از **PCB** آن بارگذاری می‌شود تا فرآیندها بتوانند بدون از دست دادن اطلاعات از سر گرفته شوند.

حال برنامه **user** ای می‌نویسیم یک مقدار در متغیر تعریف شده ست می‌کنیم و سپس این مقدار را می‌خوانیم.
خروجی تست به صورت زیر است :

```
osvm@Teymouri-Mossaviawal:~/Desktop$ ./test2
Initial value of my_custom_field: 0
Writing 12345...
New value of my_custom_field: 12345
SUCCESS!
```