



《操作系统》课第五次实验报告

学院:	软件学院
姓名:	杨万里
学号:	2013774
邮箱:	2013774@mail.nankai.edu.cn
时间:	2022/10/15

0. 开篇感言

所谓“纸上得来终觉浅，绝知此事要躬行”。

在上《操作系统》理论课之前，我对于进程的概念比较模糊，更不知道在系统中进程的具体组织和实现方式。上了理论课之后，明白了进程的组织与实现的理论。但是理论学的再好，也不如自己动手操作一个又一个的进程。本次实验，在实验指导书的指导下，我们得以亲自编码遍历系统的每个进程，查看进程的信息，这让我对于进程的组织实现方式有了更深刻的理解，也对进程结构体包含的信息有了更多的了解。

1. 实验题目

增加新的系统调用以列举所有进程的信息

2. 实验目标

- (1) 向 Linux 内核增加新的系统调用，实现列出每个进程的名字 (comm)，进



程 ID 号 (pid), 父进程 ID 号、进程状态、学号姓名等。

(2) 在用户模式测试该系统调用。

3. 原理方法

在上一次实验中, 我们已经实现了在 Linux 内核中增加一个简单的系统调用。其基本原理方法是先在操作系统中“注册”这一需要添加的系统调用 (类似函数声明), 接着需要在内核中实现该系统调用的具体内容, 上次实验的系统调用功能是打印“Hello new system call schello!**Your ID(2013774)**\n”。添加了系统调用的操作系统内核实际上已经改变, 因此需要重新编译安装, 从而用新的内核驱动 Ubuntu。

这一次同样是添加新的系统调用, 因此原理方法和上一次实验基本相同, 只是系统调用的具体代码实现不一样。实际上, 上一次实验已经为这一次实验做好了铺垫, 我们只需要把 /kernel/sys.c 文件中, 系统调用实现的具体代码修改即可。其余文件不需要改变。修改 sys.c 之后直接编译安装新内核即可 (不需要 make clean, 因此这一步会比之前快很多)。

功能实现的原理: 本次系统调用的目的是列出每个进程的名字 (comm), 进程 ID 号 (pid), 父进程 ID 号、进程状态、学号姓名等。进程的具体信息是由进程结构体 struct task_struct 实现的, 整个系统的进程由双向链表组织在一起, 系统的第一个进程是 init_task。函数 for_each_process(p) 的功能是从进程指针 p 出发, 遍历其下所有进程。因此我们首先获取到初始进程 init_task 的指针, 接着借用 for_each_process() 函数即可遍历所有进程。对于每个进程, 有一个成员 mm, 代表进程的内存空间, 内核进程没有内存空间、用户进程有内存空间, 因此可以

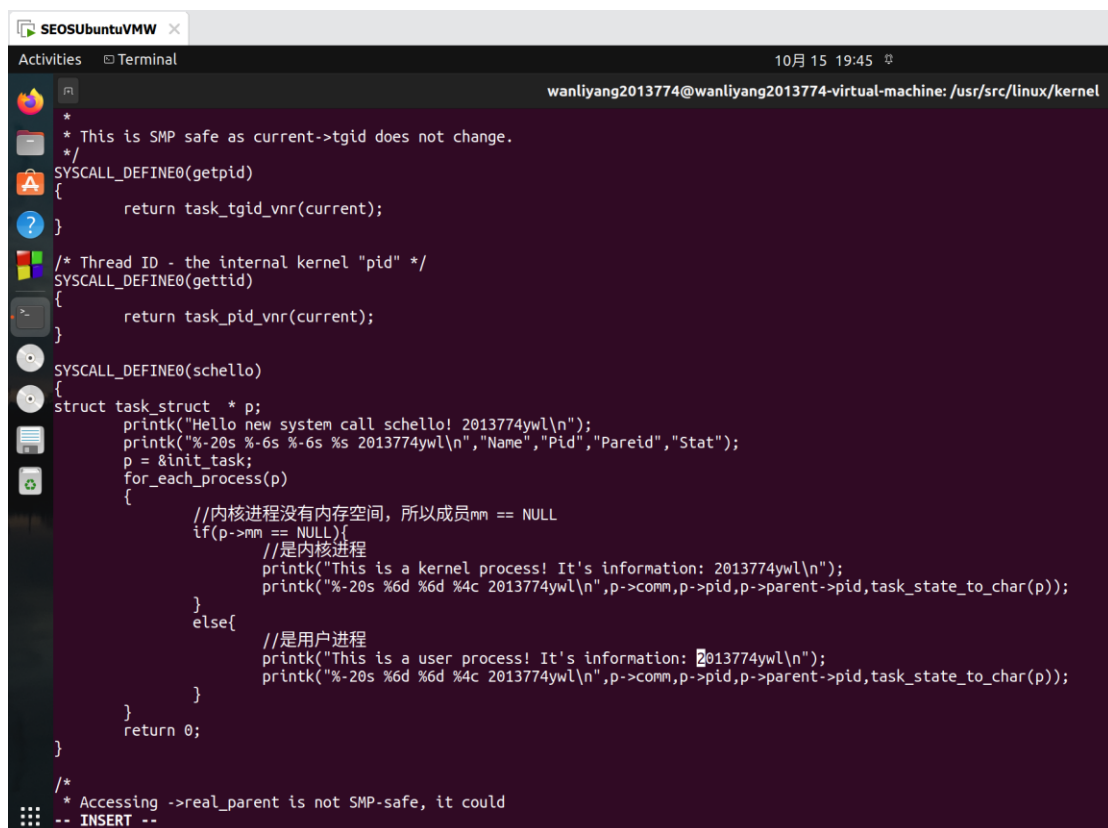


用这个成员是否为 NULL 判断进程属于内核进程还是用户进程。用户状态可以用 `static inline char task_state_to_char(struct task_struct *tsk)` 函数获得，该函数将进程状态用单个字符表示。进程名： `p->comm`，进程 ID： `p->pid`，父进程 ID： `p->parent->pid` 表示即可。

最后的测试程序和上一次实验一样，只需要使用该系统调用即可。为了获取系统调用的输出信息，我在每个需要打印的字符串之后加上了我的学号姓名“2013774ywl”作为标识。

4. 具体步骤

(1) 在 `kernel/sys.c` 文件的 `SYSCALL_DEFINE0(gettid)` 函数之后，添加如下代码，实现新的系统调用功能。（代码原理在上一小节已经阐述）。其中 **2013774ywl** 是我的学号姓名。



```
SEOSUbuntuVMW x 10月15 19:45
wanliyang2013774@wanliyang2013774-virtual-machine: /usr/src/linux/kernel

* This is SMP safe as current->tgid does not change.
*/
SYSCALL_DEFINE0(getpid)
{
    return task_tgid_vnr(current);
}

/* Thread ID - the internal kernel "pid" */
SYSCALL_DEFINE0(gettid)
{
    return task_pid_vnr(current);
}

SYSCALL_DEFINE0(schello)
{
    struct task_struct * p;
    printk("Hello new system call schello! 2013774ywl\n");
    printk("%-20s %-6s %-6s %-6s 2013774ywl\n", "Name", "Pid", "Parent", "Stat");
    p = &init_task;
    for_each_process(p)
    {
        //内核进程没有内存空间，所以成员mm == NULL
        if(p->mm == NULL){
            //是内核进程
            printk("This is a kernel process! It's information: 2013774ywl\n");
            printk("%-20s %d %d %4c 2013774ywl\n", p->comm, p->pid, p->parent->pid, task_state_to_char(p));
        }
        else{
            //是用户进程
            printk("This is a user process! It's information: 2013774ywl\n");
            printk("%-20s %d %d %4c 2013774ywl\n", p->comm, p->pid, p->parent->pid, task_state_to_char(p));
        }
    }
    return 0;
}

/*
 * Accessing ->real_parent is not SMP-safe, it could
 * -- INSERT --
 */
```



(2) 编译系统内核，这一次不需要 make clean，只需要 make 即可，因此会快很多。

```
wanliyang2013774@wanliyang2013774-virtual-machine: /usr/src/linux$ cd kernel
wanliyang2013774@wanliyang2013774-virtual-machine: /usr/src/linux/kernel$ vi sys.c
wanliyang2013774@wanliyang2013774-virtual-machine: /usr/src/linux/kernel$ cd ..
wanliyang2013774@wanliyang2013774-virtual-machine: /usr/src/linux$ make -j5
DESCEND objtool
CALL scripts/atomic/check-atomics.sh
CALL scripts/checksyscalls.sh
CHK include/generated/compile.h
CC kernel/sys.o
AR kernel/built-in.a
GEN .version
CHK include/generated/compile.h
...
OBJCOPY arch/x86/boot/setup.bin
BUILD arch/x86/boot/bzImage
Kernel: arch/x86/boot/bzImage is ready (#3)
wanliyang2013774@wanliyang2013774-virtual-machine: /usr/src/linux$
```

(3) 编译完成之后安装新的内核。(这一步可能都不需要，但是因为这一步用时很短，保险起见还是做了)

```
wanliyang2013774@wanliyang2013774-virtual-machine: /usr/src/linux$ sudo make modules_install
[sudo] password for wanliyang2013774:
INSTALL /lib/modules/5.19.10/kernel/arch/x86/crypto/aesni-intel.ko
SIGN /lib/modules/5.19.10/kernel/arch/x86/crypto/aesni-intel.ko
INSTALL /lib/modules/5.19.10/kernel/arch/x86/crypto/crc32-pclmul.ko
SIGN /lib/modules/5.19.10/kernel/arch/x86/crypto/crc32-pclmul.ko
INSTALL /lib/modules/5.19.10/kernel/arch/x86/crypto/crct10dif-pclmul.ko
SIGN /lib/modules/5.19.10/kernel/arch/x86/crypto/crct10dif-pclmul.ko
...
wanliyang2013774@wanliyang2013774-virtual-machine: /usr/src/linux$ sudo make install
INSTALL /boot
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 5.19.10 /boot/vmlinuz-5.19.10
update-initramfs: Generating /boot/initrd.img-5.19.10
run-parts: executing /etc/kernel/postinst.d/unattended-upgrades 5.19.10 /boot/vmlinuz-5.19.10
run-parts: executing /etc/kernel/postinst.d/update-notifier 5.19.10 /boot/vmlinuz-5.19.10
run-parts: executing /etc/kernel/postinst.d/xx-update-initrd-links 5.19.10 /boot/vmlinuz-5.19.10
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 5.19.10 /boot/vmlinuz-5.19.10
```

(4) 安装完成之后重启。

(5) 编写测试程序 testschello.c，和上次实验一样。

```
打开(O)  [icon] *testschello.c
~ /桌面
1 #include <unistd.h>
2 #include <sys/syscall.h>
3 #include <sys/types.h>
4 #include <stdio.h>
5 #define _NR_schello 448
6 int main(int argc, char *argv[])
7 {
8     syscall(_NR_schello);
9     printf("ok! run dmesg | grep hello in terminal!\n");
10    return 0;
11 }
12
```



(6) 编译、执行测试程序。

```
wanliyang2013774@wanliyang2013774-virtual-machine:~/Desktop$ vi testschello.c
wanliyang2013774@wanliyang2013774-virtual-machine:~/Desktop$ gcc -o testsc testschello.c
wanliyang2013774@wanliyang2013774-virtual-machine:~/Desktop$ sudo dmesg -C
[sudo] password for wanliyang2013774:
wanliyang2013774@wanliyang2013774-virtual-machine:~/Desktop$ ./testsc
ok! run dmesg | grep hello in terminal!
```

(7) 通过我之前标记的“2013774ywl”得到所有进程的信息。

```
wanliyang2013774@wanliyang2013774-virtual-machine:~/Desktop$ sudo dmesg | grep 2013774ywl
[ 347.272580] Hello new system call schello! 2013774ywl
[ 347.272592] Name      Pid      Pareid Stat 2013774ywl
[ 347.272598] This is a user process! It's information: 2013774ywl
[ 347.272600] systemd      1        0      S 2013774ywl
[ 347.272605] This is a kernel process! It's information: 2013774ywl
[ 347.272606] kthreadd      2        0      S 2013774ywl
[ 347.272609] This is a kernel process! It's information: 2013774ywl
[ 347.272611] rcu_gp         3        2      I 2013774ywl
[ 347.272614] This is a kernel process! It's information: 2013774ywl
[ 347.272616] rcu_par_gp     4        2      I 2013774ywl
[ 347.272619] This is a kernel process! It's information: 2013774ywl
[ 347.272620] netns         5        2      I 2013774ywl
[ 347.272623] This is a kernel process! It's information: 2013774ywl
[ 347.272625] kworker/0:0    6        2      I 2013774ywl
[ 347.272629] This is a kernel process! It's information: 2013774ywl
[ 347.272630] kworker/0:0H   7        2      I 2013774ywl
[ 347.272633] This is a kernel process! It's information: 2013774ywl
[ 347.272634] kworker/0:1H   9        2      I 2013774ywl
[ 347.272637] This is a kernel process! It's information: 2013774ywl
[ 347.272638] mm_percpu_wq  10       2      I 2013774ywl
[ 347.272642] This is a kernel process! It's information: 2013774ywl
[ 347.272644] rcu_tasks_kthre 11       2      I 2013774ywl
[ 347.272646] This is a kernel process! It's information: 2013774ywl
[ 347.272648] rcu_tasks_rude 12       2      I 2013774ywl
[ 347.272651] This is a kernel process! It's information: 2013774ywl
[ 347.272652] rcu_tasks_trace 13       2      I 2013774ywl
[ 347.272655] This is a kernel process! It's information: 2013774ywl
[ 347.272656] ksoftirqd/0    14       2      S 2013774ywl
[ 347.272658] This is a kernel process! It's information: 2013774ywl
[ 347.272660] rcu_preempt    15       2      I 2013774ywl
[ 347.272663] This is a kernel process! It's information: 2013774ywl
[ 347.272664] migration/0    16       2      S 2013774ywl
[ 347.272667] This is a kernel process! It's information: 2013774ywl
[ 347.272668] idle_inject/0  17       2      S 2013774ywl
[ 347.272672] This is a kernel process! It's information: 2013774ywl
[ 347.272673] cpuhp/0        19       2      S 2013774ywl
[ 347.272676] This is a kernel process! It's information: 2013774ywl
[ 347.272678] cpuhp/1        20       2      S 2013774ywl
```

.....

```
[ 347.274508] This is a user process! It's information: 2013774ywl
[ 347.274509] gnome-terminal 2627    2100    S 2013774ywl
[ 347.274512] This is a user process! It's information: 2013774ywl
[ 347.274513] gnome-terminal. 2628    2627    S 2013774ywl
[ 347.274515] This is a user process! It's information: 2013774ywl
[ 347.274517] gnome-terminal- 2633    1049    S 2013774ywl
[ 347.274519] This is a user process! It's information: 2013774ywl
[ 347.274520] bash           2663    2633    S 2013774ywl
[ 347.274523] This is a user process! It's information: 2013774ywl
[ 347.274524] update-notifier 2672    1385    S 2013774ywl
[ 347.274527] This is a kernel process! It's information: 2013774ywl
[ 347.274528] kworker/1:0     2768     2      I 2013774ywl
[ 347.274531] This is a user process! It's information: 2013774ywl
[ 347.274532] systemd-timedat 2771     1      S 2013774ywl
[ 347.274534] This is a user process! It's information: 2013774ywl
[ 347.274535] testsc         2772    2663    R 2013774ywl
```

可以发现最后一个进程（用户进程）就是我们的测试程序 testsc。

至此，成功完成本次实验。



5. 总结心得

- (1) 不管是写代码实现一个应用, 还是更改操作系统的内核, 都讲究循序渐进, 讲究由最小可用的框架开始迭代。如果一上来就要增加系统调用以实现列举所有进程的信息, 也许我们会很懵, 无从下手。但是老师先让我们通过增加简单的系统调用 (打印一条信息) 来掌握增加系统调用的基本步骤, 再要求我们实现更复杂的功能 (列举进程信息), 这样循序渐进的学习节奏, 使得我们能很好地掌握这部分的知识。
- (2) 实际上一个进程所要记录的信息比想象的要多很多, 没想到光是进程结构体的代码就有十几页 ppt 的量, 难以想象 Linux 内核背后的代码是何其庞大。
- (3) 进程以双向链表的结构组织, 使得我们能很好地获取进程信息。
- (4) 初见这个题目觉得特别困难, 但是具体开始动手实现就好多了。

6. 参考资料

实验指导书《lab05addnewsyscallbREADME.pdf》