操作系统第二次实验实验报告

舒文炫

2021 年 5 月 7 日

目录

Chapter 1

实验介绍

1.1 实验目的

本次实验需要实现

- 添加 linux 系统调用，实现一个简单的 ps，即在用户态打印出系统进程的信息
- 使用 linux 系统调用，实现一个简单的 shell，shell 需要实现 exit，cd 两个内置命令，shell 上显示当前工作目录，外置命令以及管道

1.2 预备知识

- 系统调用是操作系统提供给用户程序访问内核空间的合法接口，应用程序一般通过应用编程接口即 API 来间接使用系统调用，比如在 unistd.h 头文件中定义了很多封装好的系统调用函数，API 将系统调用号存入 EAX，然后触发软中断使系统进入内核空间，内核的中断处理函数根据系统调用号，调用对应的内核函数，最后系统调用完成相应功能，将返回值存入 EAX, 返回到中断处理函数。
- 子命令通过；分隔，| 为管道符，本次实验使用匿名管道，是只能用于父子进程的，管道间的通信时单向的。匿名管道的使用方法：父进程调用 pipe 函数创建一个匿名管道，用 pipefd 数组表示，0 为读端，1 为写端。然后 fork 出一个子进程，复制了 pipefd 数组，父进程作为写端，子进程作为读端，需要关闭父进程读端，子进程写端，我们通过 close 和 open 来实现对应端口的开启和关闭。

1.3 实验环境

- OS: Ubuntu 18.04
- Linux kernel version :4.9.263

Chapter 2

实验设计

2.1 part1：添加系统调用

进程是操作系统结构的基础，大多数情况下我们需要了解当前系统正在运行那些进程，这些进程的状态是什么，这也是我们本次实验所添加的系统调用选择了实现打印进程信息。当然 linux 外部命令 ps 也可以打印出系统进程信息，这可以作为我们系统调用实现是否成功的参考。

添加系统调用大概需要以下几个步骤：

- 注册系统调用，即修改系统调用注册表，linux-4.9.263 下为 syscall_64.tbl ，我加入的系统调用名为 ps_info
- 定义函数原型，修改 syscall.h 头文件，加入系统调用函数原型的定义，因为要输出到用户态，需要使用 ___user 宏，表示该地址是用户空间的
- 实现函数，修改 sys.c 文件，添加函数的具体实现，我期望返回进程个数，进程的 PID，进程的运行时间，需要三个参数，counter 表示进程个数，pid 是一个整型数组存储进程 pid，time 也为整型数组，存储进程运行时间。
- 编写测试代码，用来测试系统调用是否成功，即在该代码打印出所添加系统调用函数所返回的三个参数：进程个数，PID，运行时间
- 编译，使用 gcc 编译器
- 运行，输出结果

2.2 part2: 使用系统调用

shell 是用户和 linux 操作系统之间的接口，我们在 shell 窗口输入命令，shell 将命令进行处理分出命令和参数，若只有单个命令，则判断是否为内置命令，是，则在父进程运行，否，在子进程运行。若有多个命令，用管道符相连，则需要在父子进程之间建立管道接口。然后调用对应系统调用来实现命令。通过实现 shell 窗口，可以更好的理解系统调用如何使用。

2.2.1 shell 实现

我的 shell 预计完成以下功能：

- 管道符 |，支持多个管道符
- exit，pwd，cd 三个 shell 内建命令
- shell 上实现显示当前所在目录
- 外置命令

代码中需要字符串分割函数，对多个管道符用 | 分割多条命令，对单个命令用空格分隔，分出命令和参数，这部分助教已经实现，函数名为 split_string, 输入字符串和分割符，返回分割出的字符串数组。对内建命令使用函数需实现 exec_builtin, 输入字符串数组和参数个数，判断是否为内建命令，是则执行，返回 0，否则返回-1 外置命令，需要使用 fork 生成子进程，在子进程中使用 exec 系统调用，替换为目标进程。对管道操作，需要用到 pipe 函数，并通过端口的开启和关闭实现父子进程之间信息传递

2.2.2 shell 测试

- 观察命令行上是否能显示当前目录，单条外置命令准备选择 ls，打印当前目录文件或目录列表
- 内建命令，cd 分别测试 cd，cd .. ,cd /, cd dir(某个确定目录) 以及 exit
- 管道操作考虑命令 ls | grep a | grep q, 即列出文件或目录列表，将结果中含 a 的文件或目录取出，作为输入再将该结果中含 n 的输出

Chapter 3

实验实现

3.1 part1: 添加系统调用

- 注册系统调用, 我向 syscall_64.tbl 添加了如下代码。表示调用号为 332，common 表示在 x86_64 和 x32 都适用，系统调用名为 ps_info, 后面的 sys_ps_info 即为要实现的函数原型。

```
332 common  ps_info         sys_ps_info
```

- 定义函数原型，仿造前面函数原型的定义，我向 syscall.h 中添加了如下代码，其中 int *___user num 用来指向进程个数，int *___ user pid1 用来指向进程 pid 数组 int *___ user time1, 用来指向进程运行时间数组。___ user 表示数据在用户态。

```
asmlinkage long sys_ps_info(int __user * num,int __user * pid1,int __user * time1);
```

- 实现函数在 sys.c 中添加函数，如下。仿造该文件前面定义的形式，由于需要三个参数，这里定义宏函数 SYSCALL_DEFINE3,3 表示参数个数为 3，第一个调用名为 ps_info, 后面分别为所传参数的类型和名称。我定义了 counter 用来储存进程数目，pid2[500] 用来储存进程 pid，time2[500] 用来储

```
SYSCALL_DEFINE3(ps_info,int __user *,num,int __user *,pid1,int __user *,time1){
    struct task_struct* task;
    int counter=0;
    int pid2[500];
    int time2[500];
    printk("[system call] ps_info\n");
    for_each_process(task){
        pid2[counter]=task->pid;
        time2[counter]=task->utime+task->stime;
        counter++;
}
    copy_to_user(num,&counter,sizeof(int));
    copy_to_user(pid1,pid2,counter*sizeof(int));
    copy_to_user(time1,time2,counter*sizeof(int));
    return 0;
}
```

存进程运行时间，调用 ps -e | wc -l，我们知道进程数为 50 长度实际上取超过 50 的都行，我取了 500 printk 是内核中实现 print 的函数，会将信息写入到系统记录中，这里用 printk 实现系统调用名的输出函数主体部分，需要使用 task_struct 结构体来访问进程信息，其中包含进程 pid，utime 用户态时间，stime 内核态时间，总运行时间为 utime+stime 循环遍历当前进程需要用到 for_each_process 函数，传入结构体指针 task，该指针用来遍历当前进程，在循环体内每进行一次循环，counter 都有 +1，且可以顺便用 counter 当前的值，对应数组的下标，来储存进程第 counter 个 pid 和 time（数组下标从 0 开始）。最后返回到用户态，需要使用 copy_to_user 函数，将 counter 内容复制到 num，长度为 1 个 int 变量大小，将 pid2 内容复制到 pid1，长度为 500 个 int 变量大小，将 time2 内容复制到 time1，长度为 500 个 int 变量大小。

- 编写测试代码。我的代码命名为 get_ps_info, 如下图 这个代码在用户态，红色波浪线是我在 win10

```c
#include<unistd.h>
#include<stdio.h>
#include<sys/syscall.h>
#include<malloc.h>
int main(void){
    int result;
    int i=0;
    int pid3[500];
    int time3[500];
    syscall(332,&result,pid3,time3);
    printf("process number is %d\n",result);
    printf("PID\tTIME\n");
    for(i=0;i<result;i++){
        printf("%d\t%d\n",pid3[i],time3[i]);
    }
    return 0;
}
```

系统下打开了这个代码，检测不到该头文件，linux 下是正常的。我用 result，pid3，time3 分别储存调用系统调用所返回的进程个数，进程 pid 和进程运行时间，因为用到了系统调用需要使用 syscall 函数，该函数在 syscall.h 里面，先打印出进程个数，然后循环打印出所以进程 pid。运行时间。

- 编译，gcc 编译命令如下，之后将生成的 get_ps_info 文件复制到 busybox-1.32.1/_install 下，进行 make，至此我们实现了这个系统调用的添加和使用测试结果我放在实验测试部分一并展出。

3.2 part2: 使用系统调用

- 这里我对 lab2_shellwithTODO.c 文件进行修改，大体框架助教已经实现，我在这里把我所作的工作展出，完整版在对应的文件夹里
- exec_builtin 函数，用于判断是否为内置命令，是，则直接运行，否则返回-1，这里我们只需要实现 cd，pwd，exit 所以只需要判断传入第一个参数是否为这三个字符串，然后分别处理即可对 cd，我使用了 mkdir() 这个系统调用 API，该函数可以传入路径，然后将当前工作路径切换到传入的路径，其中缺省值为 home 目录，所以加了一行判断是否只有一个 cd，若是，切换到家目录。对 exit 直接调用 exit() 函数即可。对 pwd，我使用了 getcwd() 函数，传入 path 指针和长度，将当前路径返回到 path 所指向的地址，然后打印 path 即可

```c
int exec_builtin(int argc, char**argv) {
    if(argc == 0) {
        return 0;
    }
    /* TODO: 添加和实现内置指令 */

    if (strcmp(argv[0], "cd") == 0) {
    if(argc==1){
    chdir("/home");
    }
    else {
    chdir(argv[1]);
    }
    return 0;

    } else if (strcmp(argv[0], "pwd") == 0) {
    char path[500];
    getcwd(path,500*sizeof(char));
    printf("%s \n",path);
    return 0;


    } else if (strcmp(argv[0], "exit") == 0){
    exit(0);

    } else {
        // 不是内置指令时
        return -1;
    }
}
```

- execute 函数，用于运行指令，如果是内置命令此时运行了该命令，返回 0，然后退出，如果是外部命令，调用 execvp(char *file,char *argv[]) 选用该函数，是因为所传入的参数长度并不相同，execvp 可以实现变长的参数的进程切换，第一个参数是所调用的命令文件，后面的参数是该命令的参数执行成功则将该子进程替换为命令文件的进程，失败返回-1，最后用 exit 结束进程。

```
int execute(int argc, char** argv) {
    if(exec_builtin(argc, argv) == 0) {
        exit(0);
    }
    /* TODO:运行命令 */
    else{
    execvp(argv[0],argv);
    exit(0);
}
```

- 打印当前工作目录。这里我考虑使用 getcwd 函数，前面介绍过了，最后使用 printf 进行格式化输出即可

```
char path[500];
getcwd(path,500*sizeof(char));
    /* TODO:增加打印当前目录，格式类似"shell:/home/oslab ->"，你需要改下面的printf */
    printf("shell: %s-> ",path);
```

- 处理参数，分出命令和参数，这一个理解了 split_string 函数即可，命令使用空格隔开，每一个命令都存在 command[i] 中，代码如下

```
    int argc = split_string(commands[0], " ", argv);
```

- 只有一个命令时，内置命令在主进程完成，这个通过 exec_builtin 函数完成，前面也介绍了。外部命令我们使用 pid=fork() 创建了子进程,fork 成功返回进程的 pid 在父进程中子进程 pid 为 0，所以通过 if(pid==0) 这个条件进入子进程，在子进程中执行 execute 函数，来执行命令，然后调用 exit() 结束子进程 pid>0, 在父进程执行 wait(NULL), 等待子进程运行完成。

- 管道操作，由于两个管道的操作被后面多个管道操作所包含，这里我只列出多个管道的代码。这里由于有多个管道，我们要将其连接起来，需要使用一个变量保存上一个管道的读端，起名为 read_fd ，使用 for 循环创建管道（这里只要创建 n-1 个）我貌似创建了 n 个，不过对结果没有影响 orz。除了最后一个命令，都将标准输出重定向到当前管道入口，这里需要用到 dup2 函数，其原型为 int dup2(int oldfd, int newfd); 即将 newfd 标识符变成 oldfd 的一个拷贝，使得输入/输出变成 newfd 的输入/输出, 标准输入输出简单理解就是屏幕，这里就是将屏幕输出的东西，传到管道写端输入的东西传到管道读端。从而屏幕本来直接输出的结果，写入到管道，从而在读端可以读取这些结果，不过这里我们通过 read_fd 让下一个管道去读取上一个管道输出的内容，以此传递下去。保存 read_fd 的工作需要在父进程进行，由于子进程变量时父进程的 copy，所以 read_fd 也可以被子进程用到。父进程的管道读和写端都没有用，从而我们把它关掉。以及父进程需要等待子进程运行结束，由于有多个子进程，我们要全部等待结束，这里运用 while(wait(NULL)); 循环等待（busywaiting），若有子进程结束 wait 返回结束子进程的 pid，等全部子进程结束后，返回 0，这个循环结束。

```
            if(exec_builtin(argc, argv) == 0) {
                continue;
            }

    int pid=fork();
    if(pid==0){
    execute(argc,argv);
    exit(255);
}
            /* TODO:创建子进程，运行命令，等待命令运行结束
             *
             *
             *
             *
             */
    if(pid>0){
    wait(NULL);
}
```

图 3.1: 只有一个命令

```
        } else {    // 三个以上的命令
            int read_fd;    // 上一个管道的读端口（出口）
            for(int i=0; i<cmd_count; i++) {
                int pipefd[2];
                int ret=pipe(pipefd);

        if(ret < 0) {
                printf("pipe error!\n");
                continue;
            }
                /* TODO:创建管道，n条命令只需要n-1个管道，所以有一次循环中是不用创建管道的
                 *
                 *
                 *
                 */
                int pid = fork();
                if(pid == 0) {
                if(i<cmd_count-1){
                close(pipefd[READ_END]);
                dup2(pipefd[WRITE_END], STDOUT_FILENO);
                close(pipefd[WRITE_END]);
 }
                    /* TODO:除了最后一条命令外，都将标准输出重定向到当前管道入口
                     *
                     *
                     *
                     */
        if(i>0){
                close(pipefd[WRITE_END]);
                dup2(read_fd, STDIN_FILENO);
                close(read_fd);
 }
```

(a) 多管道 part1

```
        char *argv[MAX_CMD_ARG_NUM];
        int argc = split_string(commands[i], " ", argv);
                execute(argc, argv);
                exit(255);
                    /* TODO:处理参数，分出命令名和参数，并使用execute运行
                     * 在使用管道时，为了可以并发运行，所以内建命令也在子进程中运行
                     * 因此我们用了一个封装好的execute函数
                     *
                     *
                     */
                }
        if(pid>0){
        close(pipefd[WRITE_END]);
            if(i>0){
        close(read_fd);
}
        if(i<cmd_count-1){
        read_fd=pipefd[READ_END];
}

}

                /* 父进程除了第一条命令，都需要关闭当前命令用完的上一个管道读端口
                 * 父进程除了最后一条命令，都需要保存当前命令的管道读端口
                 * 记得关闭父进程没用的管道写端口
                 *
                 */
                // 因为在shell的设计中，管道是并发执行的，所以我们不在每个子进程结束后才运行下一个
                // 而是直接创建下一个子进程
            }
    while(wait(NULL)>0);
            // TODO:等待所有子进程结束
```

(b) 多管道 part2

Chapter 4

测试

4.1 part1: 添加系统调用

我们先调用 ps 函数输出结果如下：ps -e | wc -l 结果如下：进程数为 53 我定义的函数 get_ps_info

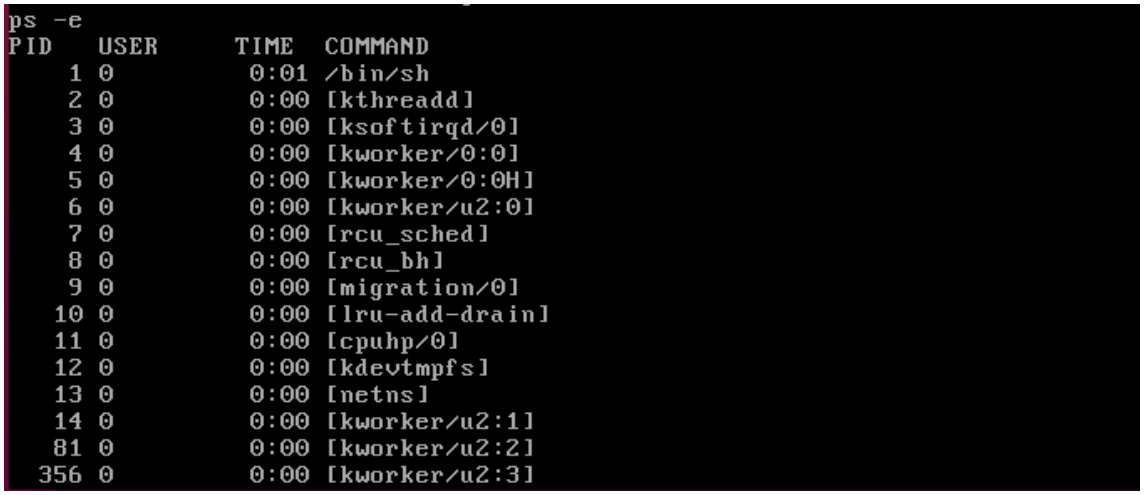

图 4.1: ps -e

图 4.2: ps -s | wc -l

输出结果如下：用 printk 输出了系统调用名，进程数为 51，每个进程成功输出了进程 pid 和运行时间 remark :wc -l 实际上统计的不是进程数目，而是输出结果的行数，所以最上面一行输出也会被统计进去，同时 wc 本身也是一个进程，也会被考虑进去，所以考虑了 ps 进程还会多算两行，不考虑 ps 进程的情况下，我们可以得到进程运行数都为 50，结果正确。

4.2 part2: 使用系统调用

下图时我的 shell 测试结果：我们可以看到，输出结果符合预期，cd .. 返回上级目录，cd /进入 /目录，cd 进入 home 目录（此处缺省值为 home），与此同时命令行的当前目录也被显示出来 exit 直接退出 ls 输出了 oslab2 的文件和目录，被 grep a 之后剩下两个，然后 grep q 剩下输出 qaq 从而完成了简易 shell 的编写。

```
[   13.503906] [system call] ps_info
process number is 51
PID     TIME
1       1387
2       10
3       1
4       2
5       0
6       2
7       19
8       0
9       3
10      0
11      0
12      30
13      0
14      65
268     49
403     0
404     0
```

图 4.3: get_ps_info

```
swx@ubuntu:~/oslab2$ ./shell
shell: /home/swx/oslab2-> cd ..
shell: /home/swx-> cd /
shell: /-> cd
shell: /home-> ls
swx
shell: /home-> cd swx/oslab2
shell: /home/swx/oslab2-> ls
1.txt         get_ps_info.c  get_ps_num.c          qaq   qwq.txt  we
get_ps_info   get_ps_num     lab2_shellwithTODO.c  qwq   qwq.zip
get_ps_info1  get_ps_num1    ps.txt                qwq1  shell
shell: /home/swx/oslab2-> ls | grep a
lab2_shellwithTODO.c
qaq
shell: /home/swx/oslab2-> ls | grep a | grep q
qaq
shell: /home/swx/oslab2-> exit
swx@ubuntu:~/oslab2$
```

图 4.4: shell

Chapter 5

实验总结

本次实验，我实现了系统调用的添加，以及使用系统调用实现一个简易的 shell。通过此次实验，我对 linux 系统调用的理解更加深刻了，也初步熟悉了管道的操作，了解到的一个小知识点：cd 的缺省值为家目录。知道了 shell 的原理，还想实现其他的功能，也能类似的添加。最开始时，由于关错了管道的端口，导致进程死掉了，无法输出结果，经过助教提醒，修改了过来。总而言之收获还是很多的。