实验二 添加 Linux 系统调用

PB19030888 张舒恒

第一部分：编写系统调用实现一个 Linux Process Status

1.1 设计思路

（1）注册系统调用

选择一个没有用过的数字作为调用号，这里我选择 333 作为 ps 的系统调用号，然后类型为 common，最后系统调用名是 ps_status，后面的函数原型是 sys_ps_status。

（2）定义函数原型

系统调用函数 sys_ps_status 需要三个用户态参数：进程数量 int *num，进程 id int *pid，进程名 char **comm。这里需要获取一系列进程的进程名，所以使用二维字符指针。函数原型为 asmlinkage long sys_ps_status(int __user * num,int __user * pid, char __user ** comm);

（3）编写实现函数

这里系统调用函数 sys_ps_status 有三个参数，所以应写成 SYSCALL_DEFINE3，定义结构体 task_struct* task，先用 printk 打印[Syscall] ps_status;对于每个 task,记录下 task 的 pid 和 comm，再让计数变量 counter 加 1，最后使用 copy_to_user 函数从内核空间向用户空间传数据。

1.2 具体实现

```
542     x32     getsockopt              compat_sys_getsockopt
543     x32     io_setup                compat_sys_io_setup
544     x32     io_submit               compat_sys_io_submit
545     x32     execveat                compat_sys_execveat/ptregs
546     x32     preadv2                 compat_sys_preadv64v2
547     x32     pwritev2                compat_sys_pwritev64v2
332     common  ps_counter              sys_ps_counter
333     common  ps_status               sys_ps_status
```

```
//这里是我们新增的系统调用
asmlinkage long sys_ps_counter(int __user * num);
asmlinkage long sys_ps_status(int __user * num,int __user * pid, char __user ** comm);
#endif
```

```
SYSCALL_DEFINE3(ps_status, int __user *, num ,int  __user *, pid, char  __user **, comm){
        struct task_struct* task;
        int counter = 0,i;
        int pid2[100];
        char comm2[100][16];
        printk("[Syscall] ps_status\n");
        for_each_process(task){
                pid2[counter]=task->pid;
                i=0;
                while(task->comm[i]!='\0'){
                        comm2[counter][i]=task->comm[i];
                        i++;
                }

                counter ++;

}
copy_to_user(num, &counter, sizeof(int));
copy_to_user(pid, pid2, 100*sizeof(int));
copy_to_user(comm, comm2, 100*16*sizeof(char));
return 0;
}
```

如图所示分别为注册系统调用，定义函数原型，编写实现函数。

1.3 功能测试

```
#include<linux/unistd.h>
#include<sys/syscall.h>
#include<stdio.h>
int main(void) {
        int result,i;
        int pid[100];
        char comm[100][16];
        syscall(333, &result,pid,comm);
        printf("pid   command\n");
        for(i=0;i<result;i++){
                printf("%d   %s\n",pid[i],comm[i]);
        }
        printf("process number is %d\n",result);
        return 0;
}
```

如图所示，先调用系统调用获取进程数量，进程 id，进程名，再输出提示行“pid　　command”，对于每个进程输出 pid 和 comm，最后输出进程数量。

```
772   acpi_thermal_pm
796   bioset
799   bioset
802   bioset
805   bioset
808   bioset
811   bioset
814   bioset
817   bioset
835   scsi_eh_0
836   scsi_tmf_0
839   scsi_eh_1
840   scsi_tmf_1
843   kworker/u2:3
848   kworker/u2:4
851   bioset
856   kworker/0:2
941   ipv6_addrconf
942   kworker/0:3
967   kworker/0:1H
969   test
process number is 51
```

如图所示为测试结果

第二部分 实现一个 linux shell

2.1 内建命令的执行

```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) {
        chdir(argv[1]);
        return 0;
    } else if (strcmp(argv[0], "pwd") == 0) {
        char buf[255];
        getcwd(buf,255);
        printf("%s",buf);
        return 0;
    } else if (strcmp(argv[0], "exit") == 0){
        exit(0);
    } else {
        // 不是内置指令时
        return -1;
    }
}
```

首先需要判断参数个数 argc 是否为 0,如果是 cd 命令,则 argv[1]是目标目录,可以调用 chdir 函数定位到这个目录,如果是 pwd 命令,则需先定义字符串数组 buf，然后调用 getcwd 函数获取当前目录，最后输出当前目录 buf，如果是 exit 函数则直接调用 exit 函数来退出 shell。当命令是上述内建命令时需要返回 0，不是内建命令时需要返回-1。

2.2 外部命令的执行

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

    return execvp(argv[0],argv);
}
```

先要判断是不是内部命令，如果是则退出，然后调用 execvp 函数，其中 execvp 函数的函数原型是 int execvp（const char *file，char *const argv[]），这里的指令的第一个参数 argv[0]就是*file 命令名。

2.3 打印当前目录

```
/* TODO:增加打印当前目录，格式类似"shell:/home/oslab ->"，你需要改下面的printf */
char cwd[255];
getcwd(cwd,255);
printf("shell:%s -> ",cwd);
fflush(stdout);

fgets(cmdline, 256, stdin);
strtok(cmdline, "\n");
```

定义字符数组 cwd，然后调用 getcwd 函数获取当前目录，再输出当前目录，调用 fflush 清除标准输出缓冲区，再输入命令行。

2.4 单个命令的执行

```
if(cmd_count == 0) {
    continue;
} else if(cmd_count == 1) {// 没有管道的单一命令
    char *argv[MAX_CMD_ARG_NUM];
    int argc=split_string(cmdline," ",argv);// TODO:处理参数，分出命令名和参数
    /* 在没有管道时，内建命令直接在主进程中完成，外部命令通过创建子进程完成 */
    if(exec_builtin(argc, argv) == 0) {
        continue;
    }
    int pid=fork();// TODO:创建子进程，运行命令，等待命令运行结束
    if(pid==0)
        execute(argc,argv);
     while(wait(NULL)>0);
```

cmd_count 为 1 表示单个命令，以空格作为分割符，分出命令名和后面的参数，调用 exec_builtin 函数，如果其返回值是 0 表明是内建命令且运行成功，则跳到下一轮循环执行下一条命令行，如果其

返回值不是 0 则创建子进程，在子进程里面调用 execute 函数运行命令，父进程等待子进程结束。

2.5 两条命令间的管道操作

```
int pipefd[2];
int ret = pipe(pipefd);
if(ret < 0) {
    printf("pipe error!\n");
    continue;
}
// 子进程1
int pid = fork();
if(pid == 0) {
    /*TODO:子进程1 将标准输出重定向到管道，注意这里数组的下标被挖空了要补全*/
    close(pipefd[0]);
    dup2(pipefd[1], STDOUT_FILENO);
    close(pipefd[1]);
    /*
        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
     */
    char *argv[MAX_CMD_ARG_NUM];
    int argc = split_string(commands[0], " ", argv);
    execute(argc, argv);
    exit(255);
}
// 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
// 子进程2
pid = fork();
if(pid == 0) {
    /* TODO:子进程2 将标准输入重定向到管道，注意这里数组的下标被挖空了要补全 */
    close(pipefd[1]);
    dup2(pipefd[0], STDIN_FILENO);
    close(pipefd[0]);

    char *argv[MAX_CMD_ARG_NUM];
    int argc = split_string(commands[1], " ", 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
}
close(pipefd[WRITE_END]);
close(pipefd[READ_END]);
while (wait(NULL) > 0);
```

cmd_count 为 2 表示有两条命令，创建一条管道。在子进程 1 中关闭管道读口，将标准输出重定向到管道写口，再关闭管道写口，最后以空格为分割符分出第一条命令的命令名和参数，调用 execute 函数执行命令。在子进程 2 中关闭管道写口，将标准输入重定向到管道读口，再关闭管道读口，最后以空格为分割符分出第二条命令的命令名和参数，调用 execute 函数执行命令。父进程需要关闭没用的读口和写口，并且等待子进程结束。

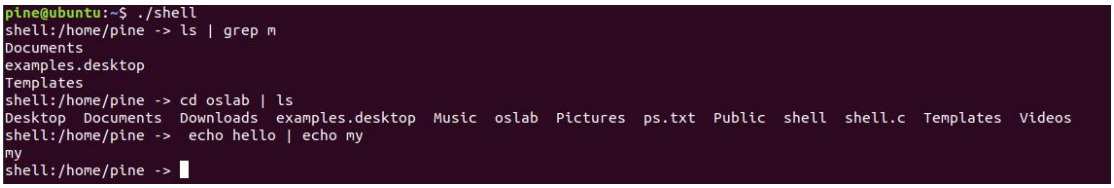
2.6 多条命令间的管道操作

```
} 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];
        // 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 ret = pipe(pipefd);
         if(ret < 0) {
        printf("pipe error!\n");
        continue;
        }
        int pid = fork();
        if(pid == 0) {
            // 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!=cmd_count-1){
                dup2(pipefd[1], STDOUT_FILENO);
                close(pipefd[1]);
            }
            // TODO:除了第一条命令外，都将标准输出重定向到上一个管道入口
            if(i){
                dup2(read_fd, STDIN_FILENO);
                close(read_fd);
            }
            char *argv[MAX_CMD_ARG_NUM];
            int argc = split_string(commands[i], " ", argv);
            execute(argc, argv);
            // 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(i!=cmd_count-1)
            read_fd=pipefd[0];
        close(pipefd[WRITE_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
```

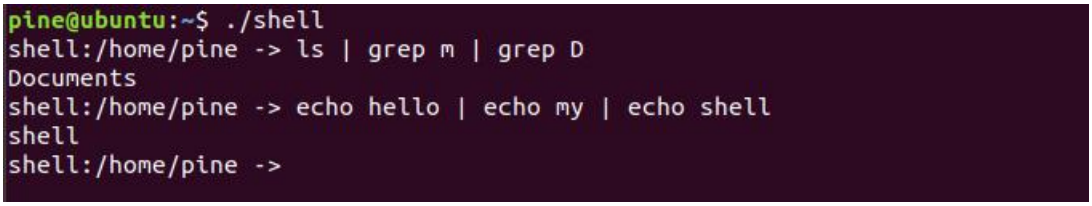
定义 read_fd 用来保存上一条管道的读口，对于每条命令创建一条管道，再创建子进程。在子进程中判断如果不是最后一条命令就将标准输出重定向到当前管道的写口，再关闭当前管道的写口，如果不是第一条命令就将就将标准输入重定向到上一个管道的读口，再关闭上一个管道的读口，最后以空格为分割符分出第一条命令的命令名和参数，调用 execute 函数执行命令。在父进程中用 read_fd 保存当前管道的读口，还需要关闭没用的管道写口。最后等待所有子进程结束。

2.7 功能测试

测试单条指令，包括内部命令 cd，pwd，exit，外部命令 ls。



测试两条指令，包括 ls | grep m，cd oslab | ls， echo hello | echo my。



测试多条指令，包括 ls | grep m | grep D，echo hello | echo my | echo shell。

总结

通过这次实验我初步学会了如何添加系统调用编写系统调用实现函数，初步理解了如何编写支持单条两条多条命令执行的 shell。