实验二 添加Linux系统调用

PB19051176 刘泽

2021年5月7日

第一部分 编写系统调用实现简单的PS

1.1代码设计

根据文档要求，需要从内核空间中传递出进程数量num、进程pid、进程运行时间time及进程名command共四个进程信息到用户空间。其中进程数量的传递完全按照文档示例代码来实现即可，而由于有多个进程,在一个系统调用函数中进行进程遍历(for_each_process)时相应会产生多个进程信息，它们不似num，仅需要传递一个变量的值，则必须用数组进行传递。

根据提示中task结构体中的信息，并经过查询，可以知道pid的类型为int，stime(此处仅传递了内核态运行时间，未考虑用户态运行时间utime,对于实验的完成影响不大)的类型为u64(即unsigned long long,但后来编译的时候系统提示类型为unsigned long,于是最终在代码中进行了修改)，故相应地创建int型数组和u64数组，对于遍历到的每一个进程，将其pid和stime传递到对应的地址空间即可。

进程名name为一个char型一维数组，多个进程名则组成了一个二维数组，于是在内核中创建了一个临时二维数组变量，并将遍历到的进程的名称储存于其中，在遍历结束后直接将整个二维数组传递到用户空间中去。

测试部分的代码较为简单，仅需要调用添加的系统调用，并打印传递到用户空间中的进程信息即可。

1.2核心代码

1.2.1 PS实现代码

- ```
  SYSCALL_DEFINE4(ps_info, int __user *, num, int __user *, pid, unsigned long __user *, time, char __user **, command){
          struct task_struct* task;
          int counter = 0;
          char temp_comm[80][20];

          printk("[Syscall] ps_info\n");

          for_each_process(task){
                int length = 0;
                copy_to_user(pid+counter, &(task->pid), sizeof(int));
                copy_to_user(time+counter, &(task->stime), sizeof(unsigned long));

                while(task->comm[length] != '\0' && length < 20){
                      temp_comm[counter][length] = task->comm[length];
                      length++;
                }
                temp_comm[counter][length] = '\0';
                counter ++;
          }
          copy_to_user(command, temp_comm, sizeof(char)*80*20);
          copy_to_user(num, &counter, sizeof(int));
          return 0;
  }
  ```
  ```

1.2.2 测试代码

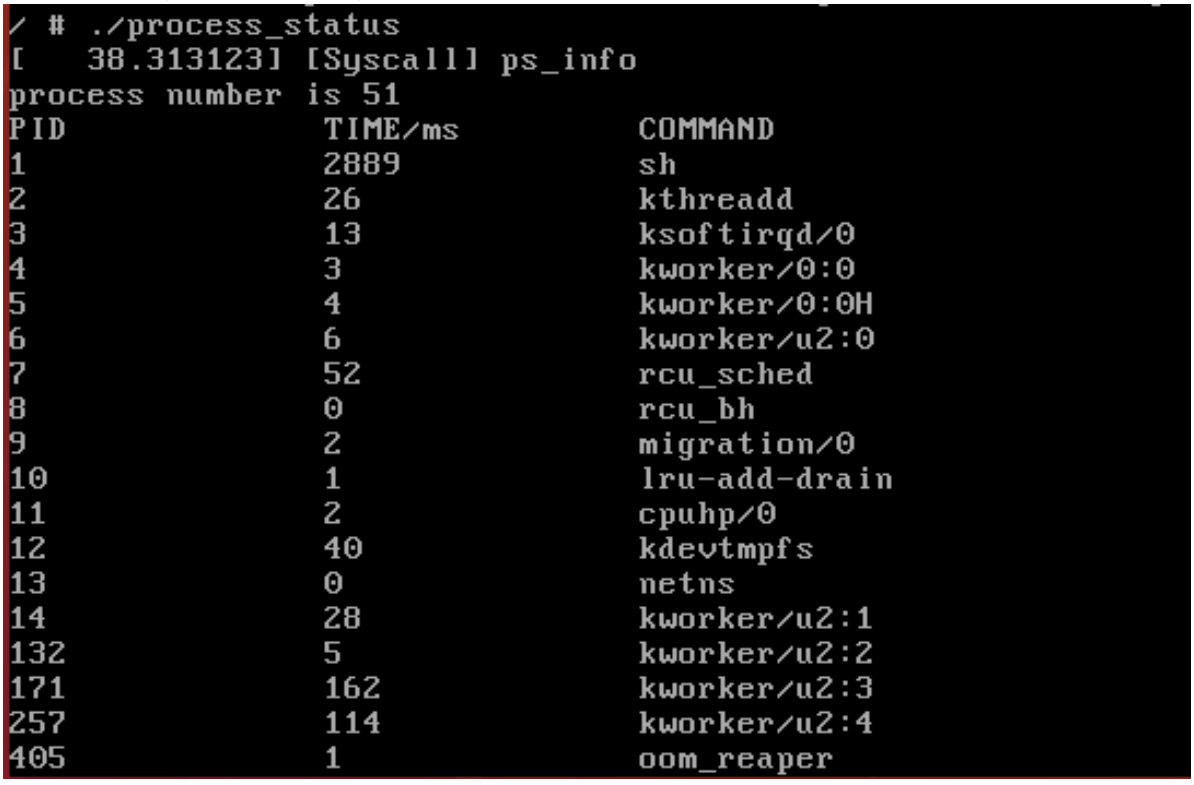
- ```
  #include<stdio.h>
  #include<unistd.h>
  #include<sys/syscall.h>

  #define MAXSIZE 80
  #define LEN_MAX 20
  int main(void){
      int num;
      int pid[MAXSIZE];
      unsigned long time[MAXSIZE];
      char command[MAXSIZE][LEN_MAX];

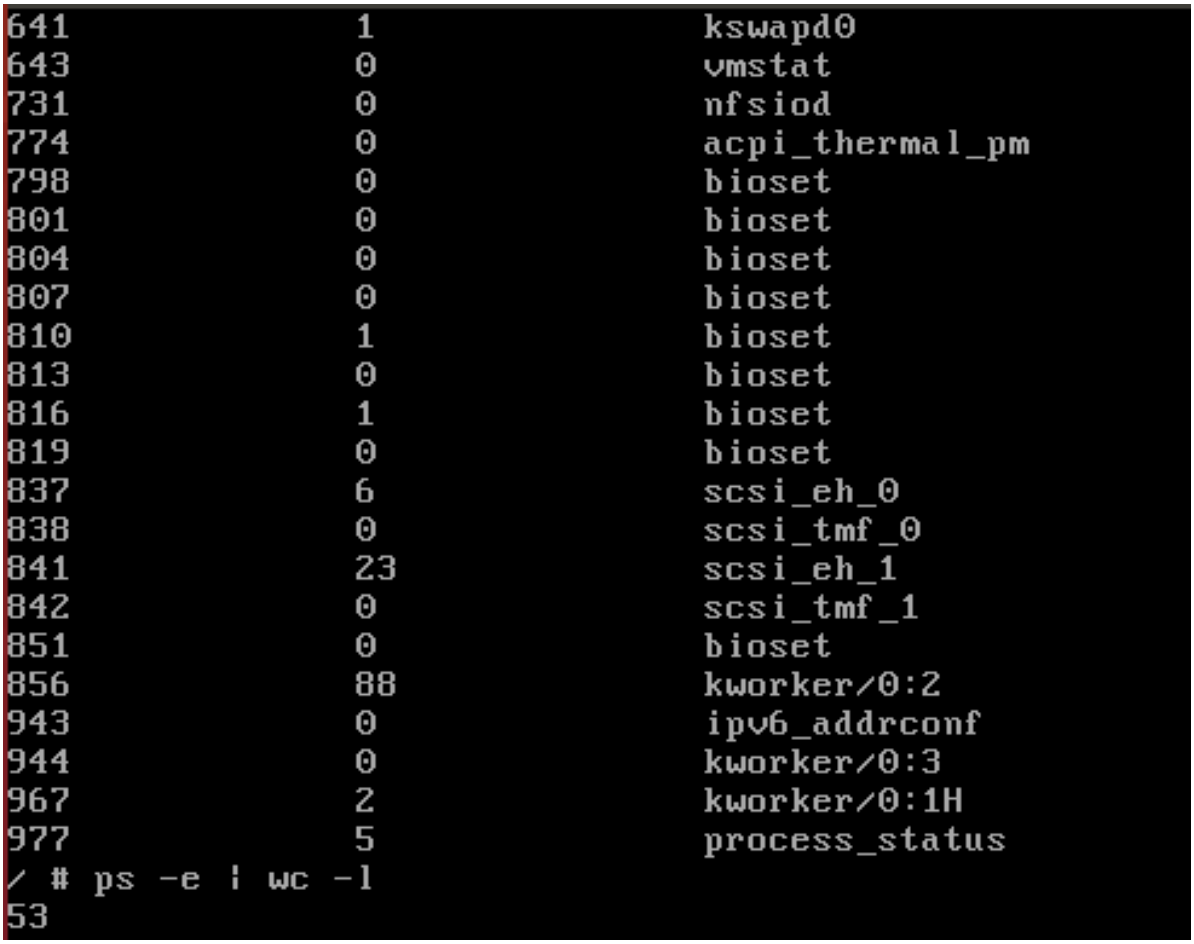
      syscall(333, &num, pid, time, command);
      printf("process number is %d\n", num);
      printf("PID     TIME/ms     COMMAND\n");
      for(int i = 0; i < num; i++)
          printf("%-3d        %-7lu       %s\n",pid[i], time[i], command[i]);
  }
  ```

1.3运行结果

在打开的qemu界面中执行process_status命令，得到如下结果:



执行ps -e | wc -l 命令得到如下结果:
  ```



process_status输出了进程的pid、运行时间和名称，对比两次的运行结果，process_status输出的进程数为51，ps -e | wc - l 输出的进程数为53，多出的两个进程数正是ps -e 和 wc - l，由此可验证实现了PS的系统调用。

第二部分 使用系统调用实现简单的Shell

2.1代码设计

根据文档要求，要摸拟实现一个简单的Shell，给出的代码中已经完成了对于输入的字符串进行拆分的工作(split_string函数)，则只需要对于拆分后得到的子串进行一个字符串匹配以确定其对应的命令，并将之实现。

Linux命令分为内置命令和外部命令，根据文档要求此次要实现的内置命令有cd、pwd及exit,查询给出的系统调用函数，chdir()可以用于改变工作目录，可用于实现cd，同时为了更加真实地模拟cd命令，增加了一些关于输入参数和运行结果的分支，具体可见代码；getcwd()可用于获取当前的工作目录，其可用于实现pwd命令，同时可以用于在命令行上显示当前目录；exit()可以结束进程，用于退出模拟的shell。

外部命令的实现通过execvp()函数实现，它可以根据传递进来的字符参数在PATH下进行搜索，找到对应二进制命令后即可执行之。(execvp中的p即指代path,若是没有p则需要认为指定搜索目录)。同时由于execvp()函数会发生二进制文件替代，需要在子进程中调用它。

此次实验要求实现管道符，以实现将标准输入或标准输出重定向。关于管道，个人认为关键一点是要明确仅有父进程会创建管道，子进程不过是对父进程进行复制，新产生文件描述符，即一个管道，多个文件描述符。实现重定向的关键函数是dup2，它可以将标准输出重定向到管道写端口，将标准输入重定向到管道读端口。还要关闭多余的端口，最终只余下前一条命令重定向过的标准输出和后一条命令重定向过的标准输入。此外，多条管道和单条管道的实现并无本质的区别，只需要对于每个管道的端口的重定向和关闭处理好即可。

2.2核心代码

2.2.1内置命令实现

内置命令cd、pwd、exit通过系统调用chdir()、getpwd()和exit()来实现，具体代码如下：

```
int exec_builtin(int argc, char**argv) {
    if(argc == 0) {
        return 0;
    }
    if (strcmp(argv[0], "cd") == 0) {
        if(argc > 2)
            printf("bash: cd: too many arguments\n");
        else if(chdir(argv[1]) == -1)
            printf("bash: cd: %s: No such file or directionary\n",argv[1]);
        return 0;
    } else if (strcmp(argv[0], "pwd") == 0) {
              char buf[MAX_BUF_SIZE];
              getcwd(buf, sizeof(buf));
              printf("%s\n",buf);
              return 0;
    } else if (strcmp(argv[0], "exit") == 0){
               exit(0);
    } else {
        // 不是内置指令时
        return -1;
    }
}
```

2.2.2外部命令实现

外部命令通过系统调用execvp()函数实现，具体代码如下:

```
int execute(int argc, char** argv) {
    if(exec_builtin(argc, argv) == 0) {
        exit(0);
    }
    execvp(argv[0], argv);
    exit(0);
}
```

2.2.3打印当前目录

打印当前目录通过系统调用getpwd()来实现，具体代码如下：

```
char buf[MAX_BUF_SIZE];
getcwd(buf, sizeof(buf));
printf("shell:%s -> ", buf);
fflush(stdout);
```

2.2.4实现单个命令

判断待执行的命令是内置命令还是外部命令，内置命令可直接调用相关子函数执行，外部命令需先创建子进程，再调用相关子函数，同时父进程需要等待子进程完成后开始运行，具体代码如下：

```
if(cmd_count == 1) {      // 没有管道的单一命令
        char *argv[MAX_CMD_ARG_NUM];
        int  argc = split_string(commands[0], " ", argv);
        if(exec_builtin(argc, argv) == 0 {
               continue;
        }
        int pid = fork();
        if( pid == 0 )
             execute(argc, argv);
        wait(NULL);
}
```

2.2.5实现只有一个管道的命令

对于只有一个管道的命令，其实是拆分成两个命令来处理，每个命令均在新创建的子进程中实现，基本类似于2.2.4。而管道由父进程创建，创建的子进程继承该管道，利用dup2函数实现将前一命令的标准输出重定向到管道写端口，将后一条命令的标准输入重定向到管道读端口，并关闭多余端口，具体代码如下:

```
if(cmd_count == 2) {      // 两个命令间的管道
            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) {
                close(pipefd[READ_END]);
                dup2(pipefd[WRITE_END], STDOUT_FILENO);
                close(pipefd[WRITE_END]);

                char *argv[MAX_CMD_ARG_NUM];
                int argc = split_string(commands[0], " ", argv);
                execute(argc, argv);
                exit(255);
            }
            // 子进程2
            pid = fork();
            if(pid == 0) {
                close(pipefd[WRITE_END]);
                dup2(pipefd[READ_END], STDIN_FILENO);
                close(pipefd[READ_END]);

                char *argv[MAX_CMD_ARG_NUM];
                int argc = split_string(commands[1], " ", argv);
                execute(argc, argv);
                exit(255);
            }
            close(pipefd[WRITE_END]);
            close(pipefd[READ_END]));
            while (wait(NULL) > 0);
    }
```

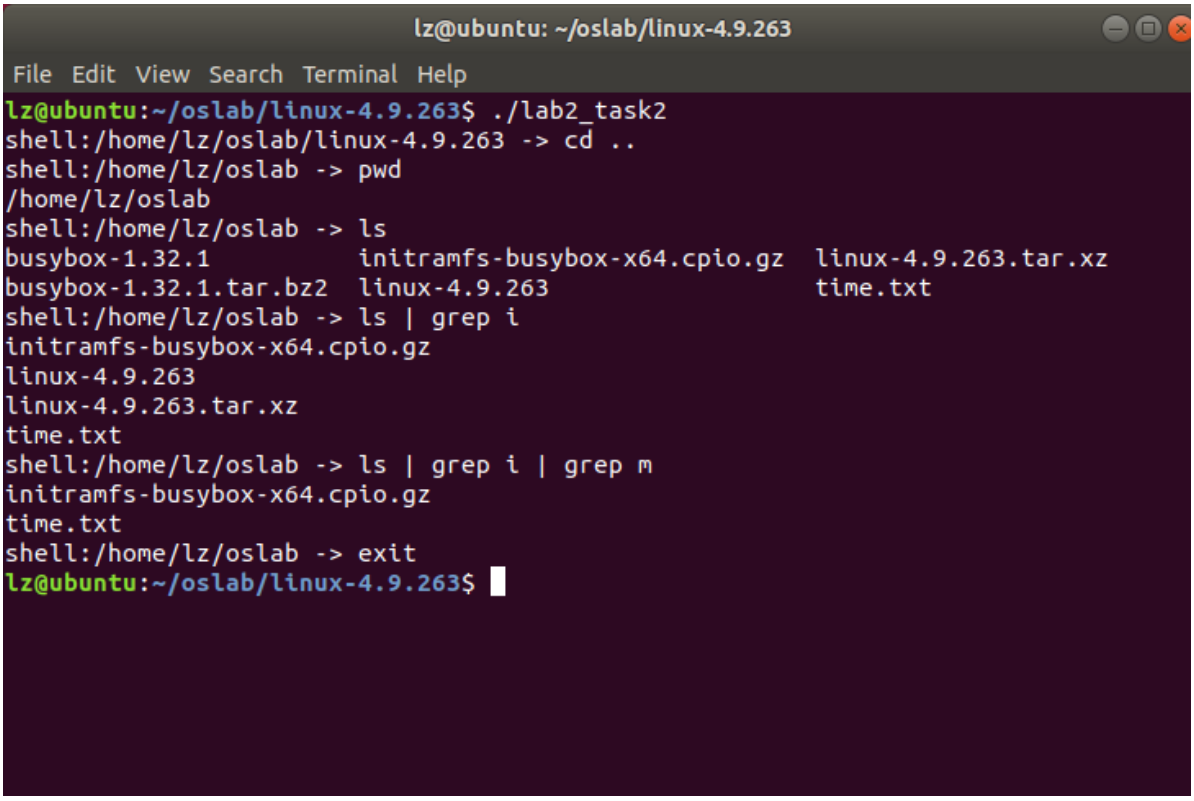
2.2.6实现多个管道的命令

对于多个管道符的命令，由于无法确定具体命令的个数，故采用循环来实现，循环参数为读入的命令个数。每次循环创建一个子进程，n个命令仅需要n-1个管道，最后一次循环不创建管道。创建的n个子进程用于执行n个命令，而管道的操作基本类似于2.2.5，但要注意此时命令是第几条，并执行相应的重定向和关闭端口操作，同时还要设置中间变量read_fd用于存储上一个管道的读端口，并注意最后一条命令没有创建管道，不要对其进行关闭端口操作,具体代码如下:

```
    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];
                 if(i != cmd_count - 1){ //最后一次循环不创建管道
                     int ret = pipe(pipefd);
                     if(ret < 0) {
                         printf("pipe error!\n");
                         continue;
                     }
                 }
                 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]);
                    }
                     if( i != 0 ){
                         dup2(read_fd, STDIN_FILENO);
                         close(read_fd);
                     }
                     char *argv[MAX_CMD_ARG_NUM];
                     int argc = split_string(commands[i], " ", argv);
                     execute(argc, argv);
                     exit(255);
                 }
                 if(i != 0)
                          close(read_fd);
                 if(i != cmd_count -1){
                          read_fd = pipefd[READ_END];
                          close(pipefd[WRITE_END]);
                 }
            }
            while (wait(NULL) > 0);
    }
```

2.3运行结果

在自己实现的Shell中运行以下命令:cd .. ; pwd ; ls ; ls | grep i ; ls | grep i | grep m ; exit，运行结果如下图所示，可以验证编写的Shell实现了命令行上显示当前目录、支持基本的单条命令运行，支持两条命令间的管道和内置命令以及支持多条命令间的管道操作。



第三部分 总结&建议

本次实验有一定的难度，我大概花了一天半的时间去完成，中间犯了一些比较低级的错误，比如移动可执行文件后未重新制作initramfs文件、有关数组之间传递信息的知识未把握好，导致耗费了比较多的时间。整个实验二完成后收获很大，对于系统调用及一些进程方面的知识有了更深的理解。特别是当真正看到编写的Shell能实现时，还是非常兴奋的。文档编写得很好，可以看出助教们费了一番心思了，不过有一点小小的建议，一是管道重定向及dup2函数在文档里面介绍得不是很清楚，看了很长时间都未曾理解，在看了PPT中的几张配图后就恍然大悟了，这些配图可以放到文档中；二是给出的系统调用函数如果能够对其功能做一个大致的提示(不是直接点出功能)，让同学们有一个大致的方向去搜索，能更大的提高完成实验和学习相关知识的效率，而单纯地把这些系统调用函数列举在这会给人一种无从下手的感觉(太多了，不知道用哪个)。 总之，实验体验很棒，绝对好评。