Project 2 -- Unix Shell Programming & Linux Kernel Module for Task

陈思远 518021910567

1. 实验内容和目标

1. 用C语言写一个shell interface，可以读入用户的指令并创建新的进程执行之。
2. shell interface需要支持输入输出重定向，并使用管道完成进程间通信。
3. 写一个Linux内核模块，输入进程标识符，输出该任务的信息。

2. Unix Shell Programming

2.1 要求与实现

1. Shell interface需要在打印出`osh>`后让用户输入指令，可以让父进程读入指令并解析指令，之后创建一个子进程来执行指令。此外，由于Unix通常允许子进程并发运行或后台运行，故指令末尾的'&'符号表示需要并发运行。由此，我们可以得到程序的大体框架如下。

```
while (should_run){
    printf("osh>");
    fflush(stdout);

    while(scanf("%s",args_copy[argindex])){ // read command
        scanf("%c",&space);
        args[argindex]=args_copy[argindex];
        argindex++;
        if(space=='\n') break;
    }
    if(strcmp(args[0],"exit")==0){  //退出
        return 0;
    }

    //TODO 历史指令功能

    if(strcmp(args[argindex-1],"&")==0){    //check whether wait
        should_wait=false;
        args[argindex-1]=NULL;
        argindex--;
    }

    pid_t pid=fork();
```

```
    if(pid==0){         //child process

        //TODO 输入输出重定向功能
        //TODO pipe功能

        execvp(args[0],args);
    }else{              //father process
        if(should_wait){
            wait(NULL);
        }
    }
}
```

之后，便可在此基础上不断增加新的功能。这里需要强调使用`execvp`函数的一些注意事项。

- execvp函数的第一个参数是要运行的文件，第二个参数是一个参数列表。
- argv列表的最后一位必须是NULL，因此我采取每次循环都将argv列表初始化为全NULL的方式。
- 如果execvp运行失败，则返回-1，回到原来的上下文；若成功，则无返回值，直接结束当前进程。所以execvp函数之后的语句，除了catch错误之外，没有意义。

2. Shell interface需要实现历史记录功能。当输入`!!`时，可以执行上一条指令；若没有上一条指令，则输出错误信息。为了实现该功能，可以添加一个存储上一条指令的二维字符数组`char hist_args[MAX_LINE/2+1][10]`，如果需要执行历史记录的指令，则把该数组中的值赋给当前执行的指令。历史记录功能的主要代码如下。

```
if(strcmp(args[0],"!!")==0){    //需要执行历史指令
    if(strlen(hist_args[0])==0){    //还没有历史指令，输出错误信息
        printf("No commands in history.\n");
        continue;
    }else{          //有历史指令，把历史指令作为当前指令来执行
        for(int i=0;i<hist_argindex;i++){
            args[i]=hist_args[i];
        }
        argindex=hist_argindex;
    }
}else{          //不需要执行历史指令，就把当前指令赋给历史指令，作为下一次的历史指令
    for(int i=0;i<argindex;i++){
        strcpy(hist_args[i],args[i]);
    }
    hist_argindex=argindex;
}
```

值得注意的是，处理历史指令功能的这部分代码必须放在其他功能之前实现。因为其他功能的实现可能会对当前指令作修改（比如带&的并发指令，就需要把&去掉再存入args数组），如果放在其他指令之后处理历史指令，就会导致原始的当前指令并没有被赋给下一条指令的历史指令，而修改后的当前指令却被赋给了下一条指令的历史指令，这显然是不对的。

3. Shell interface需要实现输入输出重定向。`>`支持将指令的输出写入一个文件，而`<`则支持把文件作为指令的输入。该功能的实现可以使用`dup2`函数。该函数的作用是把已存在的文件描述符复制赋给另一个文件描述符，比如`dup2(fd,STDOUT_FILENO)`，就会把写入standard output的内容写入到文件描述符为fd的文件中。具体实现如下。

```
if(argindex>=3 && strcmp(args[argindex-2],">")==0){     //输出重定向
    //打开文件，若文件不存在则创建
    int fd=open(args[argindex-1],O_CREAT|O_RDWR|O_TRUNC,S_IRUSR|S_IWUSR);
```

```
    if(fd<0){
        perror("open file failed!");
        exit(1);
    }
    if(dup2(fd, STDOUT_FILENO)<0){  //STDOUT_FILENO是标准输出的文件描述符
        perror("dup2 failed!");
        exit(1);
    }
    args[argindex - 2] = NULL;
    args[argindex - 1] = NULL;c
    argindex -= 2;
    close(fd);      //及时关闭文件
}else if(argindex>=3 && strcmp(args[argindex-2],"<")==0){   //输入重定向
    //打开文件，此时文件必须存在，不存在就报错
    int fd=open(args[argindex-1],O_RDONLY);
    if(fd<0){
        perror("open file failed!");
        exit(1);
    }
    if(dup2(fd, STDIN_FILENO)<0){   //STDIN_FILENO是标准输入的文件描述符
        perror("dup2 failed!");
        exit(1);
    }
    args[argindex - 2] = NULL;
    args[argindex - 1] = NULL;
    argindex -= 2;
    close(fd);      //及时关闭文件
}
```

4. Shell interface需要通过管道实现进程间通信，即可以通过管道，把一个指令的输出作为另一个指令的输入。这部分的实现需要用到`pipe()`函数，并创建一个子进程。具体代码和注释如下。

```
for (int i=0;i<argindex;i++){
    if(strcmp(args[i],"|")==0){     //判断是否需要进程间通信
        should_pipe=true;
        args[i]=NULL;           //args中仅保留'|'之前的部分
        for(int j=i+1;j<argindex;j++){
            pipe_args[pipe_argindex]=args[j];   //pipe_args中存储'|'之后的部分
            args[j]=NULL;
            pipe_argindex++;
        }
        argindex=i;
        break;
    }
}
if(should_pipe){    //需要进程间通信
    int pip_fd[2];
    pid_t pid1;

    pipe(pip_fd);   //建立管道，pip_fd[0]为管道读取端，pip_fd[1]为管道写入端

    pid1=fork();
    if(pid1<0){
        perror("pid1 fork failed!");
        exit(1);
    }else{
        if(pid1==0){    //子进程处理'|'之前的部分
```

```
            close(pip_fd[0]);
            dup2(pip_fd[1],STDOUT_FILENO);  //输出写入管道
            execvp(args[0],args);
            close(pip_fd[1]);
        }else{      //父进程处理'|'之后的部分
            wait(NULL);     //等待写入的子进程结束
            close(pip_fd[1]);
            dup2(pip_fd[0],STDIN_FILENO);   //从管道中读出作为输入
            execvp(pipe_args[0],pipe_args);
            close(pip_fd[0]);
        }
    }
}
```

此处需要考虑的一个问题是"到底把哪个进程作为父进程？"，最初我采取了把执行'|'之前的指令作为父进程，把执行'|'之后的指令作为子进程的方法，但这个方法会导致父进程的`wait(NULL)`函数等到的并不是自己的子进程，而是其他进程，从而造成错误。思考过后，我将父子进程调换，并让父进程一开始就`wait(NULL)`，等子进程执行完再执行，这样就可以保证等到的一定是自己的子进程。

2.2 结果

首先，我测试了普通指令、输入输出重定向指令和需要进程间通信的指令，结果如下。

```
chesiy@ubuntu:~/osProject/Proj2$ gcc simpleshell.c
chesiy@ubuntu:~/osProject/Proj2$ ./a.out
osh>ls -al
total 92
drwxrwxr-x 3 chesiy chesiy  4096 May 23 15:37 .
drwxrwxr-x 4 chesiy chesiy  4096 May 22 21:13 ..
-rwxrwxr-x 1 chesiy chesiy 13376 May 23 15:37 a.out
-rw-rw-r-- 1 chesiy chesiy 47835 May 23 10:11 .cache.mk
-rw-rw-r-- 1 chesiy chesiy   159 May 23 10:11 Makefile
-rw------- 1 chesiy chesiy    46 May 23 11:58 out.txt
-rw------- 1 chesiy chesiy   285 May 23 15:34 pipe.txt
-rw-rw-r-- 1 chesiy chesiy  3265 May 23 15:36 simpleshell.c
drwxrwxr-x 2 chesiy chesiy  4096 May 23 10:11 .tmp_versions
osh>ls > out.txt
osh>sort < out.txt
a.out
Makefile
out.txt
pipe.txt
simpleshell.c
osh>ls -l | less
```

```
chesiy@ubuntu: ~/osProject/Proj2
total 28
-rwxrwxr-x 1 chesiy chesiy 13376 May 23 15:37 a.out
-rw-rw-r-- 1 chesiy chesiy   159 May 23 10:11 Makefile
-rw------- 1 chesiy chesiy    46 May 23 15:37 out.txt
-rw------- 1 chesiy chesiy     0 May 23 15:38 pipe.txt
-rw-rw-r-- 1 chesiy chesiy  3265 May 23 15:36 simpleshell.c
(END)
```

接下来，我测试了历史记录功能和并行（后台运行）功能，结果如下。

```
chesiy@ubuntu:~/osProject/Proj2$ gcc simpleshell.c
chesiy@ubuntu:~/osProject/Proj2$ ./a.out
osh>ls -al
total 96
drwxrwxr-x 3 chesiy chesiy  4096 May 23 16:03 .
drwxrwxr-x 4 chesiy chesiy  4096 May 22 21:13 ..
-rw------- 1 chesiy chesiy   550 May 23 15:56 1.txt
-rwxrwxr-x 1 chesiy chesiy 13376 May 23 16:03 a.out
-rw-rw-r-- 1 chesiy chesiy 47835 May 23 10:11 .cache.mk
-rw-rw-r-- 1 chesiy chesiy   159 May 23 10:11 Makefile
-rw------- 1 chesiy chesiy    46 May 23 15:37 out.txt
-rw------- 1 chesiy chesiy   285 May 23 15:38 pipe.txt
-rw-rw-r-- 1 chesiy chesiy  3321 May 23 16:03 simpleshell.c
drwxrwxr-x 2 chesiy chesiy  4096 May 23 10:11 .tmp_versions
osh>ls -al &
osh>total 96
drwxrwxr-x 3 chesiy chesiy  4096 May 23 16:03 .
drwxrwxr-x 4 chesiy chesiy  4096 May 22 21:13 ..
-rw------- 1 chesiy chesiy   550 May 23 15:56 1.txt
-rwxrwxr-x 1 chesiy chesiy 13376 May 23 16:03 a.out
-rw-rw-r-- 1 chesiy chesiy 47835 May 23 10:11 .cache.mk
-rw-rw-r-- 1 chesiy chesiy   159 May 23 10:11 Makefile
-rw------- 1 chesiy chesiy    46 May 23 15:37 out.txt
-rw------- 1 chesiy chesiy   285 May 23 15:38 pipe.txt
-rw-rw-r-- 1 chesiy chesiy  3321 May 23 16:03 simpleshell.c
drwxrwxr-x 2 chesiy chesiy  4096 May 23 10:11 .tmp_versions
!!
osh>total 96
drwxrwxr-x 3 chesiy chesiy  4096 May 23 16:03 .
drwxrwxr-x 4 chesiy chesiy  4096 May 22 21:13 ..
-rw------- 1 chesiy chesiy   550 May 23 15:56 1.txt
-rwxrwxr-x 1 chesiy chesiy 13376 May 23 16:03 a.out
-rw-rw-r-- 1 chesiy chesiy 47835 May 23 10:11 .cache.mk
-rw-rw-r-- 1 chesiy chesiy   159 May 23 10:11 Makefile
-rw------- 1 chesiy chesiy    46 May 23 15:37 out.txt
-rw------- 1 chesiy chesiy   285 May 23 15:38 pipe.txt
-rw-rw-r-- 1 chesiy chesiy  3321 May 23 16:03 simpleshell.c
drwxrwxr-x 2 chesiy chesiy  4096 May 23 10:11 .tmp_versions
!!
osh>total 96
drwxrwxr-x 3 chesiy chesiy  4096 May 23 16:03 .
drwxrwxr-x 4 chesiy chesiy  4096 May 22 21:13 ..
-rw------- 1 chesiy chesiy   550 May 23 15:56 1.txt
-rwxrwxr-x 1 chesiy chesiy 13376 May 23 16:03 a.out
-rw-rw-r-- 1 chesiy chesiy 47835 May 23 10:11 .cache.mk
-rw-rw-r-- 1 chesiy chesiy   159 May 23 10:11 Makefile
```

最后我测试了退出功能，结果如下。

```
chesiy@ubuntu:~/osProject/Proj2$ gcc simpleshell.c
chesiy@ubuntu:~/osProject/Proj2$ ./a.out
osh>sort < out.txt
a.out
Makefile
out.txt
pipe.txt
simpleshell.c
osh>exit
chesiy@ubuntu:~/osProject/Proj2$
```

至此，shell interface要求的所有功能都已实现，该部分告一段落。

3. Linux Kernel Module for Task

3.1 要求和实现

通过`echo`指令将进程标识符写入/proc/pid文件，该内核模块需要读入进程标识符并将一些进程信息写入内核提示信息中。

首先，需要在`file_operations`中加入write的函数，表示每次写入时都会调用proc_write()函数。

```
static struct file_operations proc_ops = {
        .owner = THIS_MODULE,
        .read = proc_read,
        .write = proc_write,
};
```

接下来，就需要完成proc_write()函数。在该函数中，需要首先分配内存给即将读入的字符串，然后调用`copy_from_user`函数从用户空间将字符串传入内核空间，存入刚刚分配的内存中。之后再调用`sscanf`函数将字符串转化为长整型，存入变量`l_pid`，最后释放空间。具体代码如下。

```
static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t
count, loff_t *pos)
{
        char *k_mem;

        // allocate kernel memory
        k_mem = kmalloc(count, GFP_KERNEL);

        /* copies user space usr_buf to kernel buffer */
        if (copy_from_user(k_mem, usr_buf, count)) {
        printk( KERN_INFO "Error copying from user\n");
                return -1;
        }

        sscanf(k_mem,"%d",&l_pid);  //string turn into long integer.

        kfree(k_mem);

        return count;
}
```

当读文件/proc/pid时，需要把一些进程信息打印出来，因此也需要完成proc_read()函数。完成此部分的重点在于从task_struct的源代码中找到command、pid和state的变量名，分别为comm、pid和state。此外，假如输入的进程号是非法的，`pid_task()`函数将会返回NULL。这样就可以根据这个条件来判断进程是否合法，若非法，则输出提示信息后返回0；若合法则输出进程信息。具体实现如下。

```
static ssize_t proc_read(struct file *file, char __user *usr_buf, size_t count,
loff_t *pos)
{
    int rv = 0;
    char buffer[BUFFER_SIZE];
    static int completed = 0;
    struct task_struct *tsk = NULL;

    if (completed) {
        completed = 0;
        return 0;
    }

    tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);

    completed = 1;
    if(tsk==NULL){
        rv=sprintf(buffer,"This pid does not exit.\n");
    }else{
```

```
        rv=sprintf(buffer,"command = [%s] pid = [%d] state = [%d]\n",
                   tsk->comm,tsk->pid,tsk->state);
    }

    // copies the contents of kernel buffer to userspace usr_buf
    if (copy_to_user(usr_buf, buffer, rv)) {
        // return 0 if copy is successful
        rv = -1;//the copy fails.
        if(tsk==NULL){          //如果该进程非法，则返回0
            rv= 0;
        }
    }

    return rv;
}
```

3.3 结果

首先加载内核模块pid，之后将2写入该模块，输出pid=2的进程信息；将4写入模块，输出pid=4的进程信息。最后将不存在的进程999写入模块，输出提示信息，程序返回0。截图如下。

4.总结与思考

在我看来，这个project的难度与上一个相比有了显著的提升，尤其是第一部分实现一个shell interface。我按照课本的指导，从最初的框架开始，一步步实现新的功能；在这个过程中，我对pipe(), dup2(), fork(), execvp() 等函数的理解更加清晰，也对进程部分的知识有了更深刻的理解。在debug的过程中，我也体会到了输出错误提示的重要性，它使我们的程序更加健壮，也降低了debug的难度。此外，这次project也涉及不少给指针数组分配内存空间、释放内存空间等操作，使我对`malloc`和`free`函数的掌握更加熟练。但是，在最终的实现中，我没有采用给指针数组分配空间的方式，而是新定义一个二维数组args_copy存放args，这样就只需把args的指针指向args_copy中的对应项即可。我认为这种方式相比指针分配空间更加安全，不易泄露，实现代码也更加简洁易懂。