Report for OS Project 2

UNIX Shell Programming & Linux Kernel Module for Task Information

陈文迪 519021910071

I. 实验任务

1. 了解Unix的一些基本系统调用接口如：`fork()`,`exec()`，`wait()`和`dup2()`等。
2. 使用C语言编写一个shell接口，该shell需要包含以下的基本功能：
    - 在子进程中执行命令
    - 支持父子进程并发执行
    - 支持历史命令功能
    - 输入/输出的重定向
    - 通过Pipe管道实现进程之间的通信—将一个进程的输出作为另一个进程的输入
3. 了解如何读/写`/proc`文件系统
4. 编写一个名为`pid`的内核模块，可通过与其交互得到指定进程的命令、进程号和状态等信息。

II. 实验思路

1. Unix Shell

编写一个shell的基本流程如下：接受并解析用户输入（构建一个Parser），保存历史命令以便调用历史时可以恢复 -> 依据用户的输入分几种情况来执行指令 -> 程序运行完成后的清理工作（释放动态分配的空间）。

因此，一个符合直觉的思路便是我们可以采用模块化开发的思路来构建shell，具体过程如下。

解析用户输入（Parser）与历史命令模块

为了实现历史命令功能，我们设计了一个**单缓冲**结构，维护两个字符串数组，当新命令输入后，我们首先将旧命令存入cache中，再将新命令拆分为一个个"词"并分别存入字符串数组。

```
char *args[MAX_LINE / 2 + 1];
char *args_cache[MAX_LINE / 2 + 1];
```

为了实现模块化设计的想法，我们设计了一个结构体来方便在各个模块之间传递参数。

```
struct status
{
    int if_wait;
    int if_history;
    int if_exit;
    int in_redirect;
    int out_redirect;
    int pipe;
};
```

在以上两个关键设计的基础上，我们可以非常方便地获取用户的输入并进行命令的解析：输入部分按照空白字符进行“拆词”，再根据换行符来判断一条指令的结束；依照一定的逻辑顺序判断该条指令的类型（是否并发执行，是否为历史命令指令，是否需要重定向，是否需要进行Pipe通信等），将这些信息记录到`status`结构体中。对于结构体的记录，我们可以采用一种巧妙的方法，在记录的同时保存参数在字符串数组中的位置，这便于后阶段进行系统调用。

该模块的实现如下：

```
struct status get_input(char *args[], char *args_cache[])
{
    char temp;
    int is_blank = 1;
    int i = -1;
    int j = 0;
    for (; i < MAX_LINE / 2 + 1;)
    {
        temp = getchar();
        if (temp == '\n')
        {
            for (int k = i + 1; k < MAX_LINE / 2 + 1; ++k)
            {
                if (args_cache[k])
                    free(args_cache[k]);
                args_cache[k] = args[k];
                args[k] = NULL;
            }
            break;
        }
        if (temp == '\t' || temp == ' ')
        {
            is_blank = 1;
        }
        else if (is_blank)
        {
            is_blank = 0;
            ++i;
            j = 0;
            if (args_cache[i])
                free(args_cache[i]);
            args_cache[i] = args[i];
            args[i] = malloc(sizeof(char) * 100);
            memset(args[i],0,sizeof(char) * 100);
            args[i][j++] = temp;
        }
        else
        {
            args[i][j++] = temp;
        }
    }
    struct status result;
    result.if_history = 0;
    result.if_exit = 0;
    result.if_wait = 1;
    result.out_redirect = -1;
    result.in_redirect = -1;
    result.pipe = -1;
    if (i >= 0 && strcmp(args[0], "!!") == 0)
```

```
    {
        for (int k = 0; k < MAX_LINE / 2 + 1; ++k)
        {
            if (args[k])
                free(args[k]);
            args[k] = args_cache[k];
            args_cache[k] = NULL;
        }
        result.if_history = 1;
    }
    else if (i >= 0 && strcmp(args[0], "exit") == 0)
    {
        result.if_exit = 1;
    }
    else if (i >= 0 && strcmp(args[i], "&") == 0)
    {
        free(args[i]);
        args[i] = NULL;
        result.if_wait = 0;
    }
    else
    {
    }

    for (int i = 0; i < MAX_LINE / 2 + 1; ++i)
    {
        if (args[i]&&strcmp(args[i], ">") == 0)
        {
            result.out_redirect = i;
        }
        if (args[i]&&strcmp(args[i], "<") == 0)
        {
            result.in_redirect = i;
        }
        if (args[i]&&strcmp(args[i], "|") == 0)
        {
            result.pipe = i;
        }
    }
    return result;
}
```

指令执行模块

指令执行模块有以下两个难点。

第一，我们需要使用大量的Unix系统调用。在此处，我们首先将这些可能使用到的系统调用列出。

系统调用类型	具体系统调用
进程控制类	fork(); wait(); execvp(); exit();
管道通信类	pipe(); close();
输入输出重定向	dup2();

第二，我们需要合理地根据输入解析模块的结果安排各类型语句的判断顺序（例如，我们应该先判断指令是否为退出shell指令），由于该项目要求中表面重定向与Pipe不会同时出现，一定程度上减小了项目实现的难度。

在了解了上述难点后，我们便可以开始我们的实现：利用fork()指令创建子线程，依据是否并发来决定父线程是否需要wait()；重定向可通过dup2()将标准输入输出重定向到制定文件即可；Pipe可通过创建子进程的子进程，并在其与子进程之间建立管道，再通过重定向函数即可实现。

该模块的实现如下：

```
struct status result = get_input(args, args_cache);

if (result.if_exit)
{
    should_run = 0;
    continue;
}
else if (args[0] == NULL)
{
    if (result.if_history)
        printf("No commands in history.\n");
}
else
{
    if (result.in_redirect >= 0 )
    {
        strcpy(buffer,args[result.in_redirect+1]);
    }
    if (result.out_redirect >= 0 )
    {
        strcpy(buffer,args[result.out_redirect+1]);
    }

    pid_t pid;
    pid = fork();

    if (pid < 0)
    {
        fprintf(stderr, "Fork Failed.\n");
        return 1;
    }
    else if (pid == 0)
    {
        if (result.in_redirect >= 0)
        {
            int oldfd, fd;
            if ((oldfd = open(buffer, O_RDWR | O_CREAT, 0644)) == -1)
            {
                printf("open error\n");
                exit(-1);
            }
            fd = dup2(oldfd, fileno(stdin));
            if (fd == -1)
            {
                printf("dup2 error\n");
                exit(-1);
            }
```

```
        }

        if (result.out_redirect >= 0)
        {
            int oldfd, fd;
            if ((oldfd = open(buffer, O_RDWR | O_CREAT, 0644)) == -1)
            {
                printf("open error\n");
                exit(-1);
            }
            fd = dup2(oldfd, fileno(stdout));
            if (fd == -1)
            {
                printf("dup2 error\n");
                exit(-1);
            }
        }

        char *temp_args[MAX_LINE / 2 + 1];
        char *temp_child_args[MAX_LINE / 2 + 1];
        for (int i = 0;i<MAX_LINE / 2 + 1;++i){
            temp_args[i] = NULL;
            temp_child_args[i] = NULL;
        }
        if (result.in_redirect >= 0 )
        {
            for(int i = 0;i<result.in_redirect;++i){
                temp_args[i] = malloc(sizeof(char)*100);
                memset(temp_args[i],0,sizeof(char) * 100);
                strcpy(temp_args[i],args[i]);
            }
        }
        else if (result.out_redirect >= 0 )
        {
            for(int i = 0;i<result.out_redirect;++i){
                temp_args[i] = malloc(sizeof(char)*100);
                memset(temp_args[i],0,sizeof(char) * 100);
                strcpy(temp_args[i],args[i]);
            }
        }
        else if (result.pipe>=0)
        {
            for(int i = 0;i<result.pipe&& args[i];++i){
                temp_args[i] = malloc(sizeof(char)*100);
                memset(temp_args[i],0,sizeof(char) * 100);
                strcpy(temp_args[i],args[i]);
            }
            for(int i = result.pipe+1;i<MAX_LINE / 2 + 1 && args[i];++i){
                temp_child_args[i-result.pipe-1] = malloc(sizeof(char)*100);
                memset(temp_child_args[i-result.pipe-1],0,sizeof(char) * 100);
                strcpy(temp_child_args[i-result.pipe-1],args[i]);
            }
        }
        int a;
        if (result.in_redirect >= 0 || result.out_redirect >= 0)
            a = execvp(temp_args[0], temp_args);
        else if (result.pipe>=0){
            int fd[2];
```

```
            pid_t sub_pid;
            if(pipe(fd) == -1){
                fprintf(stderr,"Pipe failed.\n");
                return 1;
            }
            sub_pid = fork();

            if(sub_pid < 0){
                fprintf(stderr,"Fork failed.\n");
                return 1;
            }
            if(sub_pid > 0){
                wait(NULL);
                close(fileno(stdin));
                dup2(fd[READ_END], fileno(stdin));
                close(fd[WRITE_END]);
                a= execvp(temp_child_args[0],temp_child_args);
            }
            else{
                close(fileno(stdout));
                dup2(fd[WRITE_END], fileno(stdout));
                close(fd[READ_END]);
                execvp(temp_args[0],temp_args);
            }
        }
        else
            a = execvp(args[0], args);

        if (a < 0)
            fprintf(stderr, "No command found.\n");
        for(int i = 0;i<MAX_LINE / 2 + 1;++i){
            if(temp_args[i]) {
                free(temp_args[i]);
                temp_args[i] = NULL;
            }
            if(temp_child_args[i]) {
                free(temp_child_args[i]);
                temp_child_args[i] = NULL;
            }
        }
    }
    else
    {
        if (result.if_wait)
        {
            wait(NULL);
            printf("Child Complete!\n");
        }
        else
            printf("Concurrent Running!\n");
    }
}
```

清理模块

该模块虽然简单，但却十分重要，对于shell这样基础并经常使用的软件，若总是发生内存泄漏是十分糟糕的。

```
for (int i = 0; i < MAX_LINE / 2 + 1; ++i)
{
    if (args[i])
    {
        free(args[i]);
        args[i] = NULL;
    }
    if (args_cache[i])
    {
        free(args_cache[i]);
        args_cache[i] = NULL;
    }
}
```

注意：在编写shell时，也要非常关注异常处理，因为fork()和pipe()函数并不总是成功的。

```
if (pid < 0)
{
    fprintf(stderr, "Fork Failed.\n");
    return 1;
}
```

2. Linux Kernel Module for Task Information

该部分类似于我们在Project 1中编写的内核模块，不同的是我们增加了一个`proc_write()`函数，该函数会记录用户向`/proc`文件系统写入的pid值。此外，我们还利用了一个`pid_task()`函数来获取对应pid的进程的`task_struct`。`task_struct`是一个内核数据结构，我们可以通过查阅内核源码来获得它的结构。

关键函数的实现如下：

```
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) return 0;
        rv = sprintf(buffer, "command = [%s] pid = [%ld] state = [%ld]\n",tsk-
>comm,tsk->pid,tsk->state);

        if (copy_to_user(usr_buf, buffer, rv)) {
                rv = -1;
        }
```

```
        return rv;
}

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

        if (copy_from_user(k_mem, usr_buf, count)) {
        printk( KERN_INFO "Error copying from user\n");
                return -1;
        }

        sscanf(k_mem,"%ld",&l_pid);

        kfree(k_mem);

        return count;
}
```

注意：一定要记得通过`kfree()`来释放内核分配的内存空间，内核内存泄漏相当危险。

III. 实验过程

1. Unix Shell

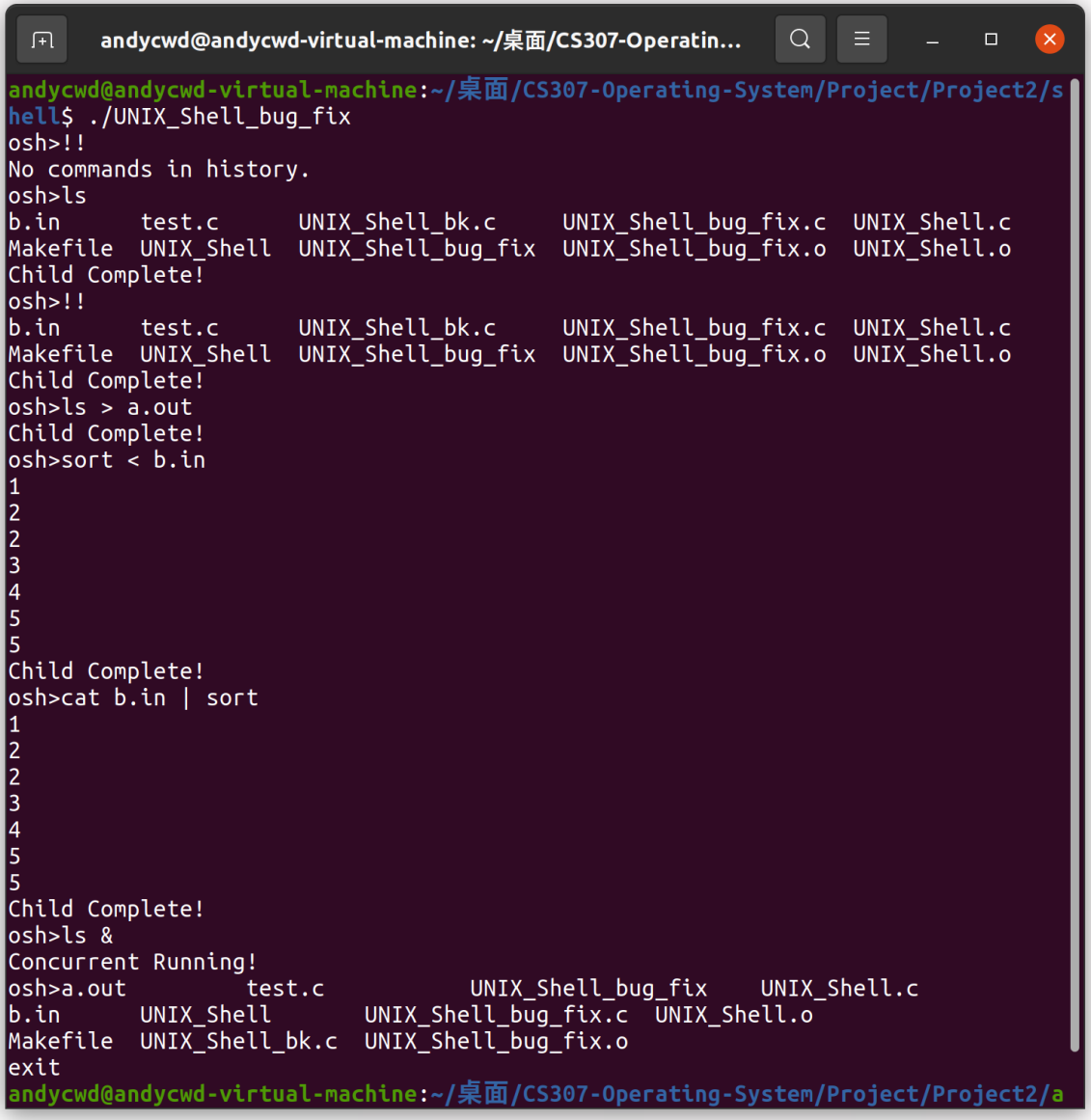
我们通过以下指令来测试我们所编写shell的鲁棒性。

```
!!                              //测试没有历史指令时调用历史指令会产生什么结果
ls
!!
ls > a.out                      //测试输出重定向
sort < b.in                     //测试输入重定向
cat b.in | sort                 //测试管道
ls &                            //测试父子进程并发执行
exit
```

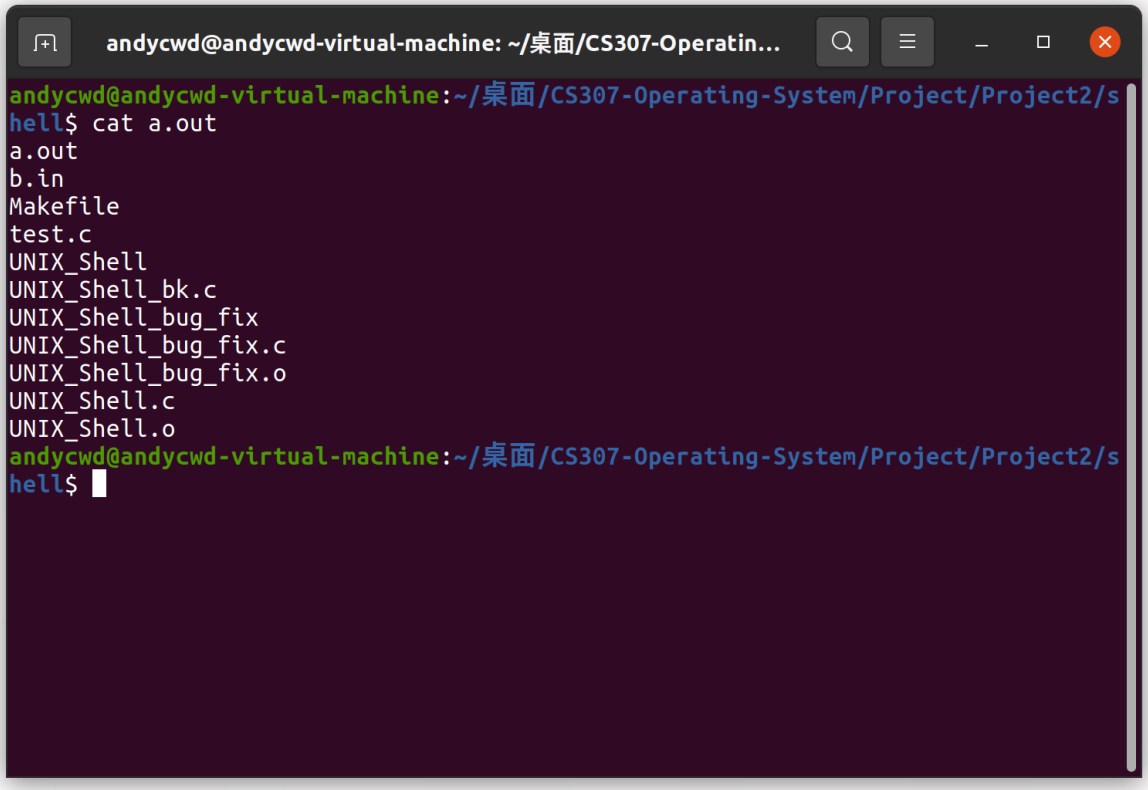
其中，`b.in`的内容如下：

```
5
4
3
2
1
2
5
```

测试结果如下：


andycwd@andycwd-virtual-machine: ~/桌面/CS307-Operatin...
andycwd@andycwd-virtual-machine:~/桌面/CS307-Operating-System/Project/Project2/shell$ ./UNIX_Shell_bug_fix
osh>!!
No commands in history.
osh>ls
b.in test.c UNIX_Shell_bk.c UNIX_Shell_bug_fix.c UNIX_Shell.c
Makefile UNIX_Shell UNIX_Shell_bug_fix UNIX_Shell_bug_fix.o UNIX_Shell.o
Child Complete!
osh>!!
b.in test.c UNIX_Shell_bk.c UNIX_Shell_bug_fix.c UNIX_Shell.c
Makefile UNIX_Shell UNIX_Shell_bug_fix UNIX_Shell_bug_fix.o UNIX_Shell.o
Child Complete!
osh>ls > a.out
Child Complete!
osh>sort < b.in
1
2
2
3
4
5
5
Child Complete!
osh>cat b.in | sort
1
2
2
3
4
5
5
Child Complete!
osh>ls &
Concurrent Running!
osh>a.out test.c UNIX_Shell_bug_fix UNIX_Shell.c
b.in UNIX_Shell UNIX_Shell_bug_fix.c UNIX_Shell.o
Makefile UNIX_Shell_bk.c UNIX_Shell_bug_fix.o
exit
andycwd@andycwd-virtual-machine:~/桌面/CS307-Operating-System/Project/Project2/a

`a.out`中的内容为：

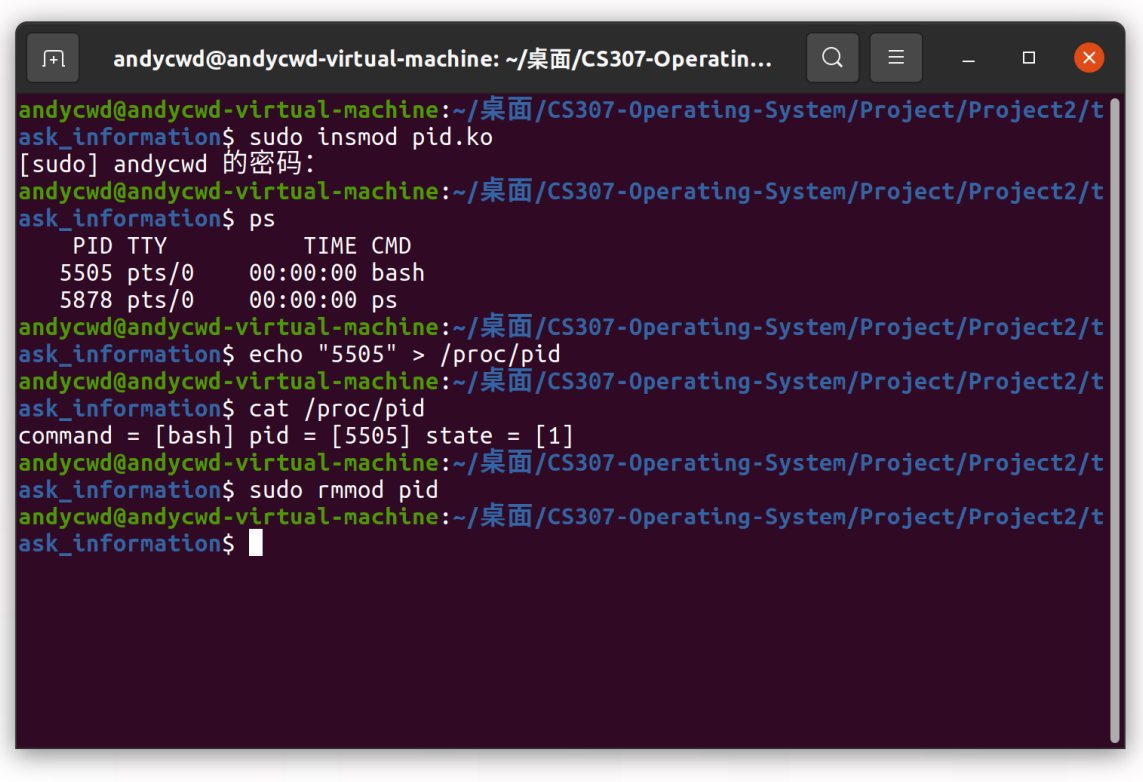


结果均符合我们的设计预期。

2. Linux Kernel Module for Task Information

我们可以先用`ps`命令查看当前的进程状态，再利用编写好的`pid`模块查询进程信息并验证。

测试结果如下：



符合我们的设计预期。

IV. 遇到的问题

在进行管道IPC时，我们应该如何安排父子进程关系?

按照书本上给与的提示，我们应该将管道符号`|`之前的进程作为父进程，而管道符号之后的进程作为子进程。这样的想法是合理的，但其更适用于父子进程并发执行的情况，而父子进程并发执行会导致我们编写的shell界面混乱，不便于用户体验。因此，我们可以考虑让父进程等待子进程执行完再执行自身代码。若采用这样的设计，我们可以将管道符号`|`之前的进程作为子进程，而管道符号之后的进程作为父进程，父进程使用`wait()`等待子进程的完成。这是自然的，因为我们可以让子进程将全部的输出产生之后再经由管道、重定向，传入父进程。采用这样的设计后，经测试，我们所设计的shell的稳定性得到了极大地提高。