Report-Project2

付昊源
517021910753

2019 年 6 月 10 日

1 UNIX Shell Programming

1.1 程序设计思想

1.1.1 Overview

命令行的作用是对输入的指令进行处理，所以我们首先应读入一行命令，并用数组分别记录每一个指令，以及记录是否有"&" 符号。如果有"&" 符号，则父进程与子进程同时进行，否则父进程需要等待子进程结束。课程文件中已经给出了本 project 的基本框架，接下来的部分需要完成四个功能：创建子进程并执行命令行命令、提供命令历史、增加输入输出重定向、允许父子进程通过管道进行沟通。

1.1.2 Excuting Command in a Child Process

首先我们在数组 $args[]$ 中储存命令行每一个指令的值。然后通过调用 $execvp(char*command, char*params[])$ 函数来模拟命令行的执行。但是在这里需要注意的是，该函数并不能识别"&" 符号，所以需要程序去识别最后一个命令行指令是否为"&" 并据此调试父子进程的关系。

1.1.3 Creating a History Feature

当用户输入"!!" 时，我们需要去给出上一条执行的指令并将其执行。因此，需要添加"!!" 的判断条件。若程序在此之前没有输入指令，则返回提示字符串，若有，则执行之。因此需要添加一个数组以储存上一条指令，并实时更新。

1.1.4 Redirecting Input and Output

与"&" 符号类似，*execvp* 函数也不能识别"<"、">" 符号，这两个符号后接一个文件名。因此，我们需要判断倒数第二个指令是否为"<"、">" 并单独处理。这里需要用到 $dup2(fd, STDOUT_FILENO)$ 函数，将输入输出方向由 stdio 重定向于文件。

1.1.5 Communication via a Pipe

与重定向类似，我们需要单独处理"|" 符号，并且它的前后分别为两条不同的指令，相当于"|" 前的指令先把结果输出到 Pipe 中，"|" 后的指令再从 Pipe 中读取内容，并执行。这相当于执行一次输出重定向，一次输入重定向。我们采用在子进程中再创建一个子进程的方式执行这两条指令。在这里我将 Pipe 选定为一个文件 pipe.txt，每次都是输出到该文件或者从该文件输入。

1.2 运行结果截图

```
fuhaoyuan@fuhaoyuan-virtual-machine:~/STUDY/OS/project2$ ./simple-shell
osh>ls
DateClient.java  fig3-35.c        newproc-win32.c  shm-posix-consumer.c
DateServer.java  in.txt           out.txt          shm-posix-producer.c
fig3-30          Makefile         pid.c            simple-shell
fig3-30.c        modules.order    pid.ko           simple-shell.c
fig3-31.c        Module.symvers   pid.mod.c        unix_pipe.c
fig3-32.c        multi-fork       pid.mod.o        win32-pipe-child.c
fig3-33.c        multi-fork.c     pid.o            win32-pipe-parent.c
fig3-34.c        newproc-posix.c  pipe.txt
osh>ls &
osh>DateClient.java  fig3-35.c    newproc-win32.c  shm-posix-consumer.c
DateServer.java  in.txt           out.txt          shm-posix-producer.c
fig3-30          Makefile         pid.c            simple-shell
fig3-30.c        modules.order    pid.ko           simple-shell.c
fig3-31.c        Module.symvers   pid.mod.c        unix_pipe.c
fig3-32.c        multi-fork       pid.mod.o        win32-pipe-child.c
fig3-33.c        multi-fork.c     pid.o            win32-pipe-parent.c
fig3-34.c        newproc-posix.c  pipe.txt
```

图 1: 执行命令行

```
fuhaoyuan@fuhaoyuan-virtual-machine:~/STUDY/OS/project2$ ./simple-shell
osh>!!
NO commands in history
osh>ps
   PID TTY          TIME CMD
  2485 pts/0    00:00:00 bash
  2623 pts/0    00:00:00 simple-shell
  2625 pts/0    00:00:00 simple-shell
  2626 pts/0    00:00:00 ps
osh>!!
ps
   PID TTY          TIME CMD
  2485 pts/0    00:00:00 bash
  2623 pts/0    00:00:00 simple-shell
  2625 pts/0    00:00:00 simple-shell
  2627 pts/0    00:00:00 ps
osh>
```

图 2: 命令历史”!!”

```
osh>ls > out.txt
osh>sort < out.txt
DateClient.java
DateServer.java
fig3-30
fig3-30.c
fig3-31.c
fig3-32.c
fig3-33.c
fig3-34.c
fig3-35.c
in.txt
Makefile
modules.order
Module.symvers
multi-fork
multi-fork.c
newproc-posix.c
newproc-win32.c
out.txt
pid.c
pid.ko
pid.mod.c
pid.mod.o
pid.o
```

图 3: 重定向

```
fuhaoyuan@fuhaoyuan-virtual-machine:~/STUDY/OS/project2$ ./simple-shell
osh>ls -l | less
```

图 4: 父子进程管道沟通-1

```
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    710 1月   4  2018 DateClient.java
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    810 6月  19  2018 DateServer.java
-rwxr-xr-x 1 fuhaoyuan fuhaoyuan   8416 5月  22 19:45 fig3-30
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    361 6月  19  2018 fig3-30.c
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    121 1月   4  2018 fig3-31.c
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    136 1月   4  2018 fig3-32.c
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    500 6月  19  2018 fig3-33.c
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    680 6月  19  2018 fig3-34.c
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    534 6月  19  2018 fig3-35.c
-rw-r--r-- 1 fuhaoyuan fuhaoyuan      8 5月  16 20:00 in.txt
-rwxr--r-- 1 fuhaoyuan fuhaoyuan    151 5月  18 17:06 Makefile
-rw-r--r-- 1 fuhaoyuan fuhaoyuan     43 5月  18 17:08 modules.order
-rw-r--r-- 1 fuhaoyuan fuhaoyuan      0 5月  18 17:08 Module.symvers
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan   8712 1月  31  2018 multi-fork
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    257 1月  31  2018 multi-fork.c
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan    780 1月  29  2018 newproc-posix.c
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan   1413 1月   4  2018 newproc-win32.c
-rw-r--r-- 1 fuhaoyuan fuhaoyuan    378 6月  10 14:55 out.txt
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan   3346 5月  18 17:08 pid.c
-rw-r--r-- 1 fuhaoyuan fuhaoyuan 282192 5月  18 17:08 pid.ko
-rw-r--r-- 1 fuhaoyuan fuhaoyuan    646 5月  18 17:08 pid.mod.c
-rw-r--r-- 1 fuhaoyuan fuhaoyuan 103936 5月  18 17:08 pid.mod.o
-rw-r--r-- 1 fuhaoyuan fuhaoyuan 179776 5月  18 17:08 pid.o
-rw------- 1 fuhaoyuan fuhaoyuan      0 6月  10 14:57 pipe.txt
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan   1115 6月  19  2018 shm-posix-consumer.c
-rwxrw-rw- 1 fuhaoyuan fuhaoyuan   1434 6月  19  2018 shm-posix-producer.c
:
```

图 5: 父子进程管道沟通-2

1.3 核心数据结构及代码解释

代码中一共使用了四个数组，分别是 args，arg，hisBuf，pipes。其中，args 是读入命令行命令的指针数组，arg 储存读入命令，hisBuf 用于储存历史命令（只能保留上一次的命令），pipes 用于父子进程管道沟通时"|" 后指令的储存。每一个数组有其对应的 num 变量，用于记录该数组有多少个有效指令。

每次读入一个命令并将其拆分，存入 arg 和 args 后，先判断输入是否为"!!"，即是否为选择历史指令。若选择了历史指令，则需判断是否存在历史指令。若未选择历史指令，判断是否让程序结束。未结束，判断有无"&"，即父子进程的同步性，用 ifWait 标记。之后判断"|" 的存在，即是否需要 pipe，并用 needPipe 标记。

以上判断阶段结束后，进入执行命令行指令阶段。首先需 fork 一个子进程，在子进程中，判断有无重定向的存在，以及 needPipe 的标记。在重定向中，只需要通过 *dup*2() 函数进行重定向，但在父子进程沟通中，则需要在创建一个子进程，在原子进程和新子进程中分别进行输出和输入的重定向，最终使用 *execvp*() 函数进行命令行的执行。

父进程中，我们只需要通过 ifWait 的标记，来判断父进程是否需要等待即可。

2 Linux Kernel Module for Task Information

2.1 程序设计思想

本 project 的设计原理与 project1 类似，都是通过在/proc 文件夹中添加文件夹而进行的。在这个 project 中，我们需要添加 pid 文件夹并通过 echo 指令在其中写入一个进程号，使用 cat 读取/proc/pid 时要返回该进程号所对应进程的指令、进程号和状态。

2.1.1 Writing to the /proc File System

下载好的 pid.c 文件已经将 proc_write() 函数写的差不多，唯一需要修改的地方是我们不能使用 kstrtol() 函数，而应使用 sscanf() 函数来读取命令中的进程号。

2.1.2 Reading from the /proc File System

在 proc_read() 函数中，首先需要通过文件中的 pid 读取到它对应的 task_struct 类型变量，并从中获取所需要的内容，再输出到命令行中。

2.2 运行结果截图

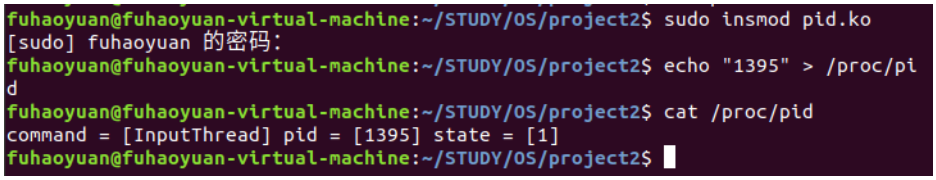

图 6: 写入存在的 pid 并读取

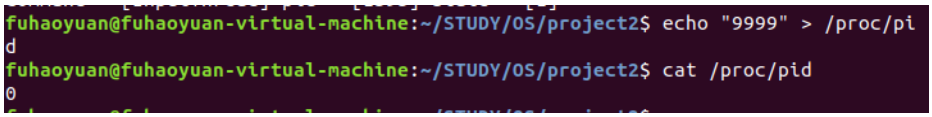

图 7: 写入不存在的 pid 并读取

2.3 核心数据结构及代码解释

首先，需要在 file_operations 中添加.write = proc_write，表示每次写操作调用 proc_write() 函数。

在 proc_read() 函数中，当我们获取了指向该进程的指针 tsk 时，我们需要到 linux/sched.h 中查找结构体 task_struct 的定义，以确定我们所需的 command，pid 和 state 的变量都是什么。通过查找，三个变量名分别为 comm, pid 以及 state。最终只需要将待输出的字符串存入 buffer 中并调用 raw_copy_to_user() 函数。

在 proc_write() 函数中，功能基本已经实现完成，需要改变的即是从用户的输入中得到 pid 的值，这一步需要使用 sscanf() 函数格式化输入并存于 l_pid 变量中。