计算机系统体系结构 Project2

517030910326 王孝诚

2019.10.31

实验环境

Windows 10 下使用 VMWare Workstation 15 Player 创建和运行虚拟机，虚拟机环境是 Linux 发行版 Ubuntu16.04.6 LTS。

1 Shell

1.1 设计思路

一个 shell 首先会读入用户的输入，然后将输入分割成 execvp() 函数可处理的 char* args[] 形式，最后根据不同的 args 情形执行指令。需要完成以下功能：

1. 在子进程中执行指令，这里直接使用 fork 的一系列操作即可。
2. 实现指令的异步和同步执行，与 1 类似，根据指令类型决定父进程是否等待子进程。
3. 保存输入指令的历史记录，用全局的 commands 的数组存储即可。
4. 实现输入输出的重定向，要用到 dup2() 函数，其可以将已有的文件描述符复制给另一个文件描述符，如将本地文件的描述复制给标准输入、输出，即可将标准输出输出到文件或者将文件输入的标准输出。
5. 用管道进行两条命令间的通信，要用到 UNIX 中的 pipe() 函数，同样需要 dup2()。由于一般是 | 前的命令的输出作为其后命令的输入，故而将 | 后的一条命令放在子进程中执行，同时其创建一个子子进程，| 前的一条命令在其中执行，| 后的命令等到子子进程结束、将输出放入管道后，再从管道中获得结果并执行命令。

1.2 核心代码解释

1. 首先是程序的主体，当未发生错误时，每次循环（对应到 shell 中即提示符后的一行）重复读取输入、提取命令、执行命令、获得运行的结果（是否出错）、然后再通过结果判断进入下一次循环。args 变量也在主体中声明好了以便作为传入获取命令和执行命令步骤的参数。

```
1  int main(void)
2  {
3      int cmd_flag;
4      char *args[MAX_LINE/2 + 1];
5      int should_run = 1;
6          while (should_run){
7              printf("osh>");
8              fflush(stdout);
9              fflush(stdin);
10             wait(NULL);
11             cmd_flag = get_cmd(args);
```

```
            should_run = execute_cmd(args, cmd_flag);
        }
    return 0;
}
```

2. 读取输入提取命令的函数对传入的 args 进行定义并存储命令历史，返回反映命令类型的整型。需要特别处理空输入和 history 类型指令输入，如果是空输入，则直接返回 EMPTY_CMD；如果是 history，则不用存储 history 命令到 history array 中；其他的命令则一律处理后每一个 arg 作为字符串存入 args，并且根据命令尾有无 & 返回 CONC_CMD（表示并行）或者 ACON_CMD（表示不并行）。

```
int get_cmd(char *args[]){
    char line[MAX_LINE];
    int pos = 0;
    char c;
    char blk = '␣';
    int is_his = 0;

    while(1){//read input
        c = getchar();
        if(c=='\n'){
            line[pos] = '\0';
            break;
        }
        else{
            line[pos++] = c;
        }
    }

    //empty cmds
    if(line[0]=='\0'){
        return EMPTY_CMD;
    }

    //don't store history cmds
    if(line[pos]=='!' || strcmp(line,"history")==0){
        is_his = 1;
    }
    if(!is_his){
        hstry_count++;
        history[hstry_count] = malloc(MAX_LINE);
    }

    pos = 0;
    int count = 0;
    while(line[pos]!='\0'){//split input to arguments, also store into history
        char arg[MAX_LINE/4];
        int cp = 0;
        while((line[pos] != blk) && (line[pos] != '\0')){
            arg[cp++] = line[pos++];
        }

```

```
42            args[count] = malloc((1+cp) * sizeof(char*));
43            if(!is_his){
44                history[hstry_count][count] = malloc((1+cp) * sizeof(char*));
45            }
46            for(int i=0;i<cp;i++){
47                args[count][i] = arg[i];
48                if(!is_his){
49                    history[hstry_count][count][i] = arg[i];
50                }
51            }
52            args[count][cp] = '\0';
53            if(!is_his){
54                history[hstry_count][count][cp] = '\0';
55            }
56            count++;
57            if(line[pos]!='\0')pos++;
58        }
59        if(!is_his){
60            history[hstry_count][count] = NULL;
61        }
62
63        // check whether repeated cmd is stored
64        int rpt = 1;
65        pos = 0;
66        if(hstry_count>0){
67            while(rpt&&history[hstry_count-1][pos]&&history[hstry_count][pos]){
68                if(strcmp(history[hstry_count-1][pos],history[hstry_count][pos])!=0){
69                    rpt = 0;
70                    break;
71                }
72                pos++;
73            }
74            if(history[hstry_count-1][pos]||history[hstry_count][pos]){
75                rpt = 0;
76            }
77            if(rpt){
78                hstry_count--;
79            }
80        }
81
82        // check concurrency
83        if(args[count-1][0]=='&'){
84            args[count-1] = NULL;
85            return CONC_CMD;
86        }
87        else{
88            args[count] = NULL;
89            // for(int i=0;i<count+1;i++){printf(args[i]);}
90            return ACON_CMD;
91        }
92    }
```

3. 然后是执行命令，首先要判断命令类型是空、执行上一条指令、查看指令历史、重定向输入输出、pipe 还是普通指令。如果是空，直接返回 1，使 shell 进入下一次循环（shell 提示符换行）；如果是查看指令历史，则打印输出 history array 的内容；如果是普通指令，则新建子进程并调用 execvp() 函数，然后根据同步/异步 flag 决定父进程是否等待；如果是其他的，则调用相应功能的实现。

```
int execute_cmd(char *args[], int flag){
    // redirect output
    int i=0;
    while (args[i])
    {
        if (strcmp(args[i], ">") == 0)
            return redirect_output(args, flag);
        i++;
    }

    // redirect input
    i=0;
    while (args[i])
    {
        if (strcmp(args[i], "<") == 0)
            return redirect_input(args, flag);
        i++;
    }

    // pipe
    i=0;
    while (args[i])
    {
        if (strcmp(args[i], "|") == 0)
            return piped_cmd(args, flag);
        i++;
    }

    // !! cmd
    if (args[0][0]=='!'){
        if(hstry_count==-1){
            printf("No commands in history.\n");
            return 1;
        }else{
            return run_history();
        }
    }

    // show_history cmd
    if (strcmp(args[0],"history")==0){
        for(int i=0;i<=hstry_count;i++){
            printf("%d  ",i+1);
            int c=0;
            while(history[i][c]){
                printf("%s ",history[i][c++]);
            }
```

```
47                 printf("\n");
48             }
49             return 1;
50         }
51 
52         if(flag==EMPTY_CMD){// 空指令
53             return 1;
54         }else  if(flag==ACON_CMD){
55 
56             // common cmd not concurrency
57             pid_t pid;
58             pid = fork();
59             if(pid < 0){
60                 perror("Fork failed!\n");
61                 return 1;
62             }
63             else  if(pid == 0){//child process
64                 if(execvp(args[0],args)==-1){
65                     perror(args[0]);
66                     exit(1);
67                 }
68             }
69             else{//parent process
70                 wait(NULL);
71                 return 1;
72             }
73         }
74         else  if(flag==CONC_CMD){
75 
76             // common cmd concurrency
77             pid_t pid;
78             pid = fork();
79             if(pid < 0){
80                 perror("Fork failed!\n");
81                 return 1;
82             }
83             else  if(pid == 0){//child process
84                 if(execvp(args[0],args)==-1){
85                     perror(args[0]);
86                     exit(1);
87                 }
88             }
89             else{//parent process
90                 return 1;
91             }
92         }
93         return 1;
94     }
```

然后看一下各个特定指令的代码实现。

(a) history：history 相关指令有两条：!! 用来执行前一条指令，history 用来查看指令记录。前

者提取全局 history array 的最后一条记录，将其作为 execute_cmd() 的参数进行调用即可；后者打印输出 history array 的内容，已集成在 execute_cmd() 函数中了。

```
int run_history(){
    int flag;
    char *his_args[MAX_LINE/2+1];
    int len = 0;

    // 将history的最后一条取出
    while(history[hstry_count][len]){
        his_args[len] = malloc(20*sizeof(char*));
        strcat(his_args[len], history[hstry_count][len]);
        len++;
    }

    // 判断同/异步
    if(strcmp(his_args[len-1],"&")==0){
        flag = CONC_CMD;
        his_args[len-1] = NULL;
    }else{
        his_args[len] = NULL;
    }
    return execute_cmd(his_args,flag);
}
```

(b) 重定向输入输出：两者的实现十分类似，关键步骤在于用 open() 函数打开文件，获得其返回的文件描述符，然后用 dup2() 将该描述符与标准输入/输出绑定，然后执行命令即可。需要注意打开文件的模式（可否读写执行），以及及时关闭之。

```
int redirect_output(char **args, int flag){
    pid_t pid, wpid;
    int status;
    pid = fork();

    if(pid<0){
        perror("Fork failed!\n");
        return 1;
    }else if(pid==0){
        char **ori_args = malloc(MAX_LINE * sizeof(char*));
        int i=0;
        while(args[i][0]!='>'){
            ori_args[i] = args[i];
            i++;
        }
        char *out_path = args[i+1];
        int out_put = open(out_path, O_WRONLY | O_TRUNC |
            O_CREAT, S_IRUSR | S_IRGRP | S_IWGRP | S_IWUSR);//
            第二个参数关于打开的模式，第三个参数设置若新建文件的权限
        dup2(out_put, STDOUT_FILENO);
        close(out_put);
        if(execvp(ori_args[0],ori_args)==-1){
            perror(ori_args[0]);
```

```
            exit(1);
        }
    }else{
        // 决定父进程是否等待
        if(flag==ACON_CMD){
            do{
                wpid = waitpid(pid, &status, WUNTRACED);
            }while(!WIFEXITED(status) && !WIFSIGNALED(status));
            return 1;
        }else{
            return 1;
        }

    }
}
```

(c) pipe：先切分出左右指令，创建一个管道，创建子进程并在其中创建子子进程执行左指令，子子进程中将标准输出与管道输入端绑定，子进程中标准输入与管道输出端绑定，子子进程执行左指令，子进程等待子子进程结束后执行右指令。

```
int piped_cmd(char **args, int flag){
    pid_t cpid;
    cpid = fork();
    if(cpid<0){
        perror("Fork failed!\n");
        return 1;
    }else if(cpid == 0){
        char *child_args[MAX_LINE/2 + 1];
        char *grand_args[MAX_LINE/2 + 1];
        int pos=0;
        int i=0;
        while(args[i][0]!='|'){
            grand_args[i]=args[i];
            i++;
            pos++;
        }
        i++;
        grand_args[pos] = NULL;
        while(args[i] != NULL){
            child_args[i-pos-1]=args[i];
            i++;
        }
        child_args[i-pos-1] = NULL;
        int pipefd[2];
        if(pipe(pipefd)==-1){
            perror("pipe error!");
            exit(1);
        }
        int gpid;
        gpid = fork();
        if(gpid<0){
            perror("Fork failed!\n");
```

```
33                  exit(1);
34              }else  if(gpid>0){// 子进程
35                  //right  cmd, read from pipe[0]/output
36                  wait(NULL);
37                  dup2(pipefd[0],STDIN_FILENO);//read from pipe
38                  close(pipefd[1]);
39                  if(execvp(child_args[0],child_args)==-1){
40                      perror(child_args[0]);
41                      exit(1);
42                  }
43              }else{// 子子进程
44                  //left  cmd, output to pipe[1]/input
45                  dup2(pipefd[1],STDOUT_FILENO);
46                  close(pipefd[0]);
47                  if(execvp(grand_args[0],grand_args)==-1){
48                      perror(grand_args[0]);
49                      exit(1);
50                  }
51              }
52          }else{
53              if(flag==ACON_CMD){
54                  wait(NULL);
55              }
56          }
57          return 1;
58      }
```

2 pid 模块

2.1 设计思路

需要创建一个名为/proc/pid 的/proc 文件。通过将 echo 的输出重定向 write 到文件中，可以存储 echo 的内容，这里我们将保存特定的进程 id。当被执行 read 操作时，pid 模块通过 find_vpid() 函数找到相对应进程的信息并打印输出，同时要处理异常信息；当移除 pid 模块时，/proc/pid 文件也被移除。通过在 proc_ops 中将.read 和.write 设定为对应的 proc_read 和 proc_write 函数，并在相应函数中实现功能即可。

2.2 核心代码解释

首先是 write 函数，要用 kmalloc() 申请内核的内存空间，然后用 copy_from_user 将命令行 echo 指令输入存到申请的内核内存中，再用 sscanf() 将之赋给全局变量 l_pid 进行保存。

```
1  static  ssize_t  proc_write(struct  file  * file , const char ___user *usr_buf, size_t count,
       loff_t *pos)
2  {
3          char *k_mem;
4
5          // allocate  kernel  memory
6          k_mem = kmalloc(count, GFP_KERNEL);
7
8          /* 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,"%ld",&l_pid);
        kfree(k_mem);
        return count;
}
```

接下来是 read 函数，当命令行读取文件的时候被调用。因为 find_vpid() 返回的是 pid_task 结构体的指针，通过指针打印对应进程的各种信息即可。

```
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);
        if(tsk){
                rv = sprintf(buffer, "command = [%s] pid = [%ld] state =
                    [%ld]\n",tsk->comm,l_pid,tsk->state);
        }else{
                rv = sprintf(buffer, "Invalid pid = [%ld], no such task!\n",l_pid);
        }
        completed = 1;

        // copies the contents of kernel buffer to userspace usr_buf
        if (copy_to_user(usr_buf, buffer, rv)) {
                rv = -1;
        }
        return rv;
}
```

3 实验结果

```
osh>ps
   PID TTY          TIME CMD
  2326 pts/17   00:00:00 bash
  3741 pts/17   00:00:00 simple-shell2
  3782 pts/17   00:00:00 ps
osh>ls
out  simple.c  simple-shell2  simple-shell.c
osh>cat out &
osh>out
simple.c
simple-shell2
simple-shell.c

osh>history
1       ps
2       ls
3       cat out &
osh>ps
   PID TTY          TIME CMD
  2326 pts/17   00:00:00 bash
  3741 pts/17   00:00:00 simple-shell2
  3831 pts/17   00:00:00 ps
osh>!!
   PID TTY          TIME CMD
  2326 pts/17   00:00:00 bash
  3741 pts/17   00:00:00 simple-shell2
  3838 pts/17   00:00:00 ps
osh>ls > ls.txt
osh>sort < ls.txt
ls.txt
out
simple.c
simple-shell2
simple-shell.c
osh>cat ls.txt | sort
ls.txt
out
simple.c
simple-shell2
simple-shell.c
osh>cat ls.txt
ls.txt
out
simple.c
simple-shell2
simple-shell.c
osh>
```

Figure 1: Shell

```
xcwang@ubuntu:~/Documents/project2/2$ echo "4300" > /proc/pid
xcwang@ubuntu:~/Documents/project2/2$ cat /proc/pid
command = [kworker/1:0] pid = [4300] state = [1026]
```

Figure 2: Linux Kernel Module for Task Information