

上海交通大学

《操作系统》课程

学生实验报告

实验名称: _____ Project2 _____
姓 名: _____ 洪瑄锐 _____
学 号: _____ 517030910227 _____
班 级: _____ F1703302 _____
手 机: _____ 15248246044 _____
任课老师: _____ 吴晨涛 _____

2019 年 5 月 26 日

目录

1. 实验要求

2. 程序设计思想及代码解释

2-1 UNIX Shell

2-2 Linux Kernel Module for Task Information

3. 运行结果

一、实验要求

2-1 UNIX Shell

写一个 c 程序作为 shell 接口，接受用户命令，在单独的进程中执行每个命令。该 shell 需要支持输入和输出重定向，以及作为一对命令之间的 IPC 形式的管道。

2-2 Linux Kernel Module for Task Information

编写一个 Linux 内核模块，该模块使用 /proc 文件系统根据其进程标识符 pid 显示任务的信息。

二、程序设计思想及代码解释

2-1 UNIX Shell

1. 读入用户在命令行输入的命令

采用为 char 型指针动态分配内存的方法，按字符读入命令。

```
/*读入user命令行参数*/
char *user_input(void)
{
    tmp_length = COMMAND_BUFSIZE;
    char *tmp; //存储命令的数组
    int index = 0;
    tmp = malloc(sizeof(char) * tmp_length);
    int c;

    while (1)
    {
        c = getchar();

        if (c == EOF || c == '\n')
        {
            tmp[index] = '\0';
            return tmp;
        }
        else
        {
            tmp[index] = c;
        }
        index++;
        if (index >= tmp_length)
        {
            tmp_length += COMMAND_BUFSIZE;
            tmp = realloc(tmp, tmp_length);
        }
    }
}
```

2. 解析用户输入的命令

通过阅读教材，了解到应将用户输入命令进行拆分，比如用户输入 ps -ael，经过拆分后 args[0]=“ps”，args[1]=“-ael”，args[2]=“NULL”。读入用户命令所得为字符串，因此拆分可以利用 string 库的 strtok 函数，函数原型为 char* strtok (char* str, const char* delimiters); str 第一次传入时为要进行切割

的字符串的首地址，在后面调用时传入 NULL, delimiter 中的每个字符都会被当作分割符。此外，因项目要求，需要一个变量 whether_wait 表示父进程是否要等待子进程结束, whether_wait=1 表示要等待, whether_wait=0 表示不需等待。

```
/*拆分命令*/
char** user_input_split(char *command)
{
    tmp_length = MAX_LINE;
    int index = 0;
    char **tmp = malloc(tmp_length * sizeof(char*));
    char *token;

    token = strtok(command, " \n\t\r");
    while (token != NULL)
    {
        tmp[index] = token;
        index++;

        if (index >= tmp_length)
        {
            tmp_length += MAX_LINE;
            tmp = realloc(tmp, tmp_length * sizeof(char*));
        }

        token = strtok(NULL, " \n\t\r");
    }

    if (index > 0 && strcmp(tmp[index - 1], "&") == 0)
    {
        whether_wait = 0;
        tmp[index - 1] = NULL;
    }
    tmp[index] = NULL;
    return tmp;
}
```

3. 执行命令并将命令存入历史记录中

函数参数为用户所输的命令，返回值为是否要继续该程序，如果继续则返回 1，否则返回 0。首先判断所输的命令是否为“!!”，如果是，则调用执行历史记录的函数，如果不是，则判断是否为“exit”，如果是则直接 return 0，如果不是则首先将命令存入历史记录中，然后调用执行函数。

存入历史记录时利用 strcat 函数，注意这里存入的是命令字符串如“ps - ael”，而不是经拆分后的字符串数组。

```

int child_exec(char *command)
{
    char **args = user_input_split(command);

    if (args[0] == NULL)//空指令
    {
        return 1;
    }
    else if (strcmp(args[0], "!!") == 0)
    {
        return history_exec(args);
    }

    //将指令存入history缓冲区中
    cur_history_index = (cur_history_index + 1) % MAX_LINE;
    history[cur_history_index] = malloc(MAX_LINE * sizeof(char));
    char **tmp_command = args;

    while (*tmp_command != NULL)
    {
        strcat(history[cur_history_index], *tmp_command);
        strcat(history[cur_history_index], " ");
        tmp_command++;
    }

    if (strcmp(args[0], "exit") == 0)
        return 0;

    return child_work(args);
}

```

4. 创建子进程执行命令

1. 创建子进程

2. 识别命令中的“<” “>” “|”，并对三种情况分别进行处理。

• “>”

要求将输出重定向到某个文件，文件名在命令行被输入，如果 args[i]='>'，那么文件名为 args[i+1]。利用 dup2 (fd, 1)，fd 是文件的文件描述符，该函数将 fd 复制到标准输出端，这意味着对标准输出的任何写入都将发送到文件中。此外，在执行结束后，要恢复对终端的输出。

```

for (; args[i] != NULL; i++)
{
    if (strcmp(">", args[i]) == 0)
    {
        args[i] = NULL;
        close(1); //关闭标准输出端
        int fd = open(args[i + 1], O_WRONLY | O_CREAT, 0777);
        copyfd = dup2(fd, 1);

        flag = 1;
        break;
    }
}

execvp(args[0], args);
if (flag)
{
    close(1);
    dup2(copyfd, 1);
}

```

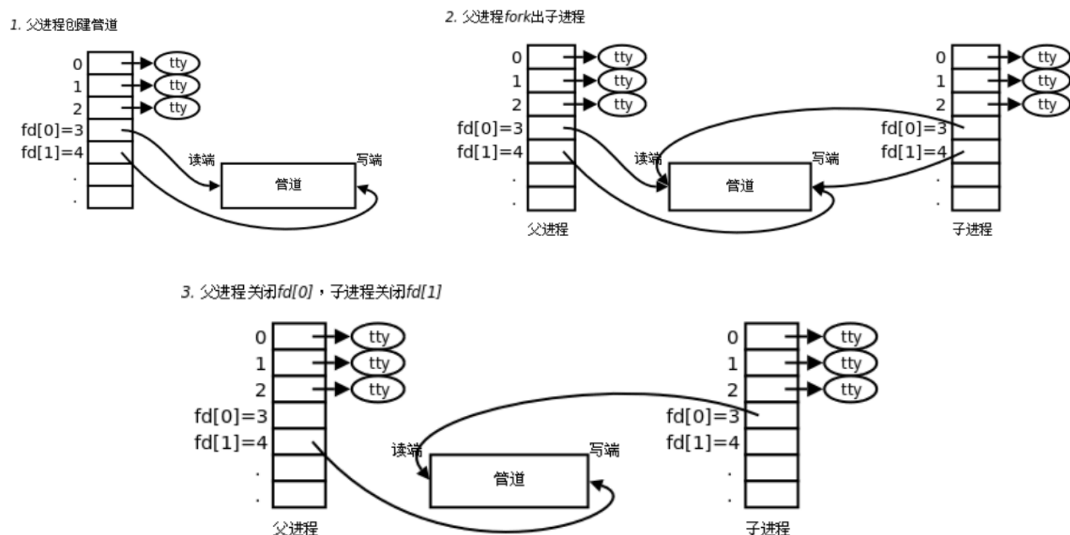
- “<”

将对一个命令的输入重定向到文件中，利用 `dup2(fd, 0)`，将文件描述符 `fd` 复制到标准输入端。

```
if (strcmp("<", args[i]) == 0)
{
    args[i] = NULL;
    int fd = open(args[i + 1], O_RDONLY, 0666);
    dup2(fd, 0);
    close(fd);
    break;
}
```

- “|”

允许一个命令的输出是另一个命令的输入，使用 `pipe` 完成。通过网上查询有关 `pipe` 的信息了解到以下过程：



如图所示为将父进程向管道中写入数据，子进程向管道中读出数据。本项目要求将子进程的输出作为父进程的输入，那么应该与上图相反，父进程关闭 `fd[1]`，子进程关闭 `fd[0]`。此外，还需利用 `dup2` 进行标准输入输出的重定向。

```
if (whether_pipe) //子进程的输出作为父进程的输入
{
    args[i] = NULL;
    int fd[2];
    int res = pipe(fd);

    pid_t pid2 = fork();
    if (pid2 < 0)
    {
        perror("child process create failed\n");
        exit(1);
    }
    if (pid2 == 0) //子进程
    {
        close(fd[0]); //关闭读端
        dup2(fd[1], 1); //标准输出定向到写端
        execvp(args[0], args);
        exit(1);
    }
    if (pid2 > 0) //原父进程的子进程
    {
        close(fd[1]); //关闭写端
        dup2(fd[0], 0); //将标准输入重定向到读端
        execvp(args[i + 1], args);
        exit(1);
    }
}
```

- 其他普通命令，直接执行即可

```
else
{
    execvp(args[0], args);
}
```

3. 对父进程的处理

因为项目要求针对父进程是否等待子进程结束再进行自己的工作在命令后设置了“&”，所以要对命令中是否含有“&”进行识别和处理。在命令拆分函数中设置了对 whether_wait 的识别，在子进程执行函数中设置了对 whether_wait 的处理。

```
if (index > 0 && strcmp(tmp[index - 1], "&") == 0)
{
    whether_wait = 0;
    tmp[index - 1] = NULL;
}

else if (pid > 0)
{
    if (whether_wait)
    {
        printf("Parent waits\n");
        wait(NULL);
    }
    else
    {
        printf("Parent doesn't wait\n");
    }
}

whether_wait = 1;
```

4. 对“!!”执行历史命令的处理

设置执行历史命令的函数，利用之前处理过的历史记录缓冲区得到刚刚运行过的命令，再调用子进程执行函数执行。注意如果历史记录缓冲区为空，则要输出提示信息。

```
/*执行历史命令*/
int history_exec()
{
    if (cur_history_index == -1 || history[cur_history_index] == NULL)
    {
        fprintf(stderr, "No command in history\n");
        exit(EXIT_FAILURE);
    }

    char **history_command;
    char *tmp = malloc(sizeof(history[cur_history_index]));
    strcat(tmp, history[cur_history_index]);
    printf("%s\n", tmp);
    history_command = user_input_split(tmp);
    return child_work(history_command);
}
```

5. 主函数

```
int main(void)
{
    int should_run = 1;
    char *command;

    while (should_run)
    {
        printf("osh>");
        fflush(stdout); //更新输出缓冲区
        command = user_input();
        should_run = child_exec(command);
        free(command);
    }
    return EXIT_SUCCESS;
}
```

2-2 Linux Kernel Module for Task Information

1. Writing to the /proc File System

由课本提示可知需要将 struct file 操作中的字段.write 设置为.write=proc_write，这使得每次对 /proc/pid 执行写操作时都会调用 proc_write() 函数。

对于 proc_write 函数，首先为字符型指针 k_mem 动态分配内存，然后将写入 usr_buf 中的内容复制到刚刚分配的内核内存中，再利用 simple_strtol 函数获取输入内容中的十进制数，最后释放内存。

```
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 */
    copy_from_user(k_mem, usr_buf, count);

    l_pid = simple_strtol(k_mem, NULL, 10);
    kfree(k_mem);

    return count;
}
```

2. Reading from the /proc File System

利用 struct task_struct pid_task(struct pid*pid, enum pid_type type) 获取任务信息，pid 由 find_vpid(int pid) 获取 struct pid，pid_type 为 PIDTYPE_PID。

如果 pid_task 返回为 NULL，则进行相应的错误信息输出。

否则输出所有的任务信息。

```
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(1_pid), PIDTYPE_PID);

    completed = 1;


    if (tsk == 0)
        rv = sprintf(buffer, "0\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)) {
        rv = -1;
    }

    return rv;
}
```

三、运行结果截图

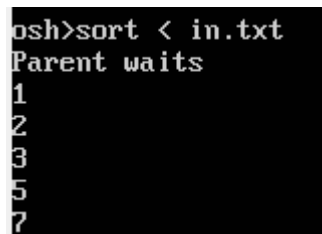
1. UNIX Shell

- 测试普通命令



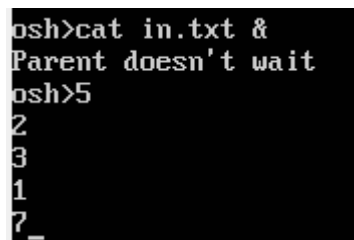
```
osh>cat in.txt
Parent waits
5
2
3
1
7osh>_
```

- 测试输入重定向



```
osh>sort < in.txt
Parent waits
1
2
3
5
7osh>_
```

- 测试&



```
osh>cat in.txt &
Parent doesn't wait
osh>5
2
3
1
7_
```

‘_’ 表示父进程等待命令输入

- 测试输出重定向

开始 out.txt 为空，输入命令 `ls > out.txt`，out.txt 不为空

```
osh>cat out.txt
Parent waits
osh>ls > out.txt
Parent waits
osh>cat out.txt
Parent waits
osh>a.out
data.txt
DateClient.java
DateServer.java
fig3-30.c
fig3-31.c
fig3-32.c
fig3-33.c
fig3-34.c
fig3-35.c
file
in.txt
lxl
Makefile
modules.order
Module.symvers
multi-fork
multi-fork.c
newproc-posix.c
newproc-win32.c
out.c
out.txt
p1
p1.c
pid.c
```

- 测试|

```
osh>ls -l | less
Parent waits
a.out
data.txt
DateClient.java
DateServer.java
fig3-30.c
fig3-31.c
fig3-32.c
fig3-33.c
fig3-34.c
fig3-35.c
file
in.txt
lxl
Makefile
modules.order
Module.symvers
multi-fork
multi-fork.c
newproc-posix.c
newproc-win32.c
out.c
```

可进行翻页操作。

2. Linux Kernel Module for Task Information

```
osc@ubuntu:~/final-src-osc10e/ch3$ sudo insmod pid.ko
[sudo] password for osc:
osc@ubuntu:~/final-src-osc10e/ch3$ echo "4" > /proc/pid
osc@ubuntu:~/final-src-osc10e/ch3$ cat /proc/pid
command=[kworker/0:0] pid=[4] state=[0]
osc@ubuntu:~/final-src-osc10e/ch3$ _
```