操作系统(D)

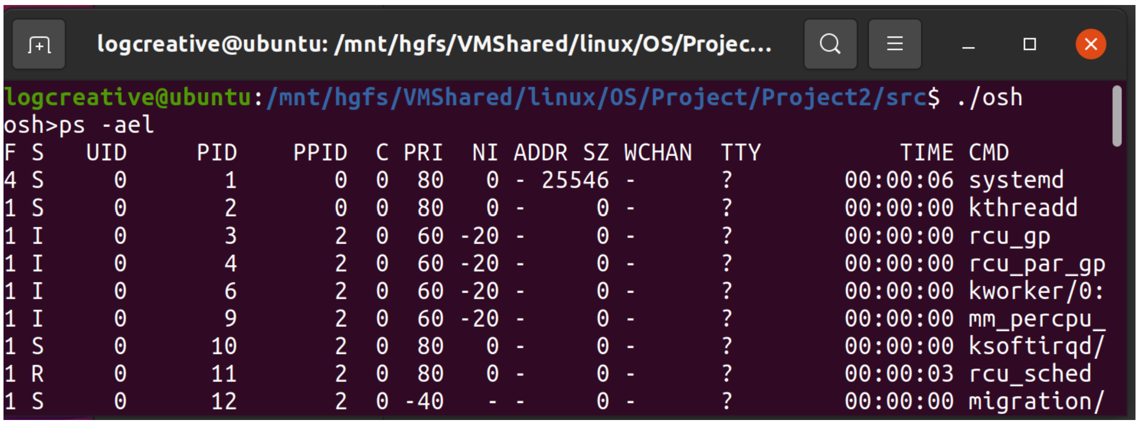
项目 2

Log Creative

2021 年 6 月 15 日

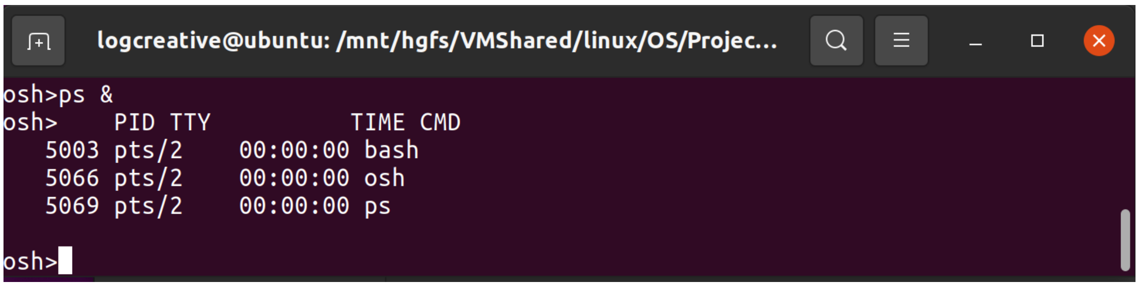
一 Unix 外壳程序

1. 在子进程中执行命令


logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Projec...

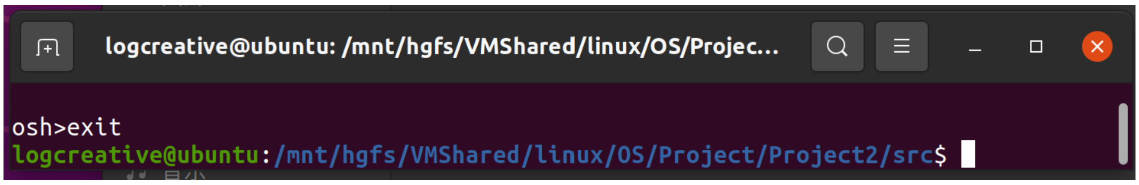
```
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ ./osh
osh>ps -ael
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0       1       0  0  80   0 - 25546 -      ?        00:00:06 systemd
1 S     0       2       0  0  80   0 -     0 -      ?        00:00:00 kthreadd
1 I     0       3       2  0  60 -20 -     0 -      ?        00:00:00 rcu_gp
1 I     0       4       2  0  60 -20 -     0 -      ?        00:00:00 rcu_par_gp
1 I     0       6       2  0  60 -20 -     0 -      ?        00:00:00 kworker/0:
1 I     0       9       2  0  60 -20 -     0 -      ?        00:00:00 mm_percpu_
1 S     0      10       2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/
1 R     0      11       2  0  80   0 -     0 -      ?        00:00:03 rcu_sched
1 S     0      12       2  0 -40   - -     0 -      ?        00:00:00 migration/
```

同时执行命令：


logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Projec...

```
osh>ps &
osh>    PID TTY          TIME CMD
   5003 pts/2    00:00:00 bash
   5066 pts/2    00:00:00 osh
   5069 pts/2    00:00:00 ps

osh>
```

当输入 exit 时，会退出外壳程序：


logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Projec...

```
osh>exit
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$
```

可以看到外壳主进程在子进程尚未打印完成前就已经开始等待用户输入了，也就是两个进程同时进行。

该部分需要获取用户输入，这里采用了 fgets 函数获取用户的整行输入。

```
fgets(rline, MAX_LINE, stdin);
```

第二参数用于说明最大读入字符数，将会限定最大的范围防止越界。该函数也支援存储换行符 `\n`。

在遍历输入字符时，如果遇到 `&` 字符，将会对 `should_wait` 标志置为 0，

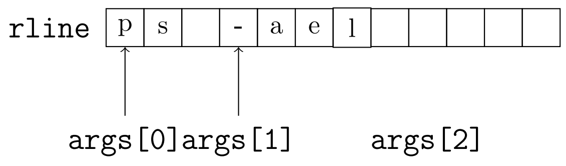
```
case '&':
    should_wait = 0;
```

在之后的子进程创建函数中将会根据这个标志判定是否等待：

```
pid_t pid;
pid = fork();

if (pid < 0) fprintf(stderr, "Fork Failed");
else if (pid == 0) return execvp(args[0],args);
else if (should_wait) wait(NULL);
```

对于 `args` 数组的存储，将会将每个字符串数组的起始位置指向 `rline` 用户输入数组的对应位置（该过程通过判定前一个字符是否是空格实现），以实现参数存储

```
default:
    if (prev == ' ') args[argc++] = rptr;
    prev = *rptr;
    break;
```



当遇到空格、换行、&符号时，都会被替换为 `\0`，从而限制每一个字符串数组的终止位置，读到换行符会强制终止循环

```
case '&':
    should_wait = 0;
case '\n':
    flag = 1;
case ' ':
    prev = *rptr;
    *rptr = '\0';
    break;
```

最后一个参数根据规定，必须被赋值为空指针

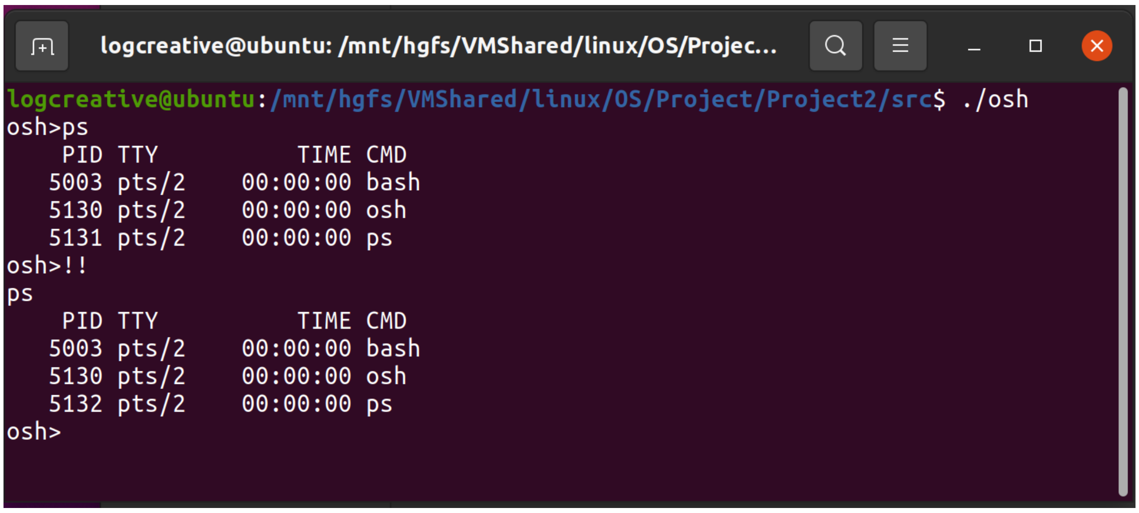
```
args[argc] = NULL;
```

退出的判定是通过 `strcmp` 函数实现的

```
should_run = strcmp(args[0],"exit");
```

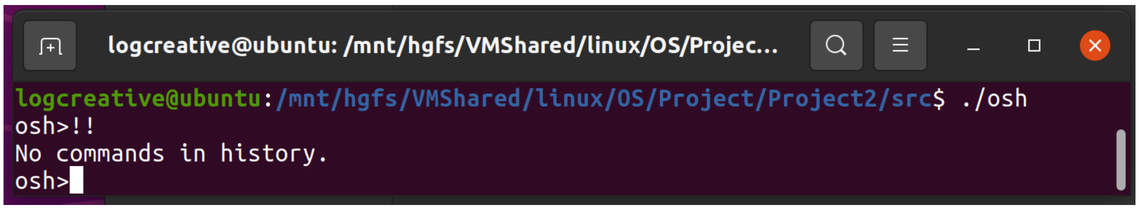
2. 历史记录存储

当输入 `!!` 时将会回显并运行上一条指令



```
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ ./osh
osh>ps
    PID TTY          TIME CMD
   5003 pts/2    00:00:00 bash
   5130 pts/2    00:00:00 osh
   5131 pts/2    00:00:00 ps
osh>!!
ps
    PID TTY          TIME CMD
   5003 pts/2    00:00:00 bash
   5130 pts/2    00:00:00 osh
   5132 pts/2    00:00:00 ps
osh>
```

当没有上一条指令时，将会返回提示



```
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ ./osh
osh>!!
No commands in history.
osh>
```

这一部分需要将上一条指令存储在 buf 中，开始时置为空

```
        char buf[MAX_LINE] = "\0";
```

当连续输入两个 ! 时，将会首先检查 buf[0] 是否是空字符，否则将通过 memcpy 将缓存复制到当前的阅读行数组中，注意这里不能使用 strcpy，因为其的复制到 \0 就会立刻停止，导致后面的部分没有被有效复制。如果是其他指令，则会被存储到缓冲区中去。

```
        if (strcmp(rline,"!!\n")==0){
            if(!buf[0]){
                fprintf(stdout, "%s\n", "No commands in history.");
                flag = -1;
            } else {
                memcpy(rline, buf, MAX_LINE*sizeof(char));
                fprintf(stdout, "%s", rline);
            }
        } else memcpy(buf, rline, MAX_LINE*sizeof(char));
```

如果不是正常指令，将不会执行任何指令。

```
        if(flag == -1) continue;
```

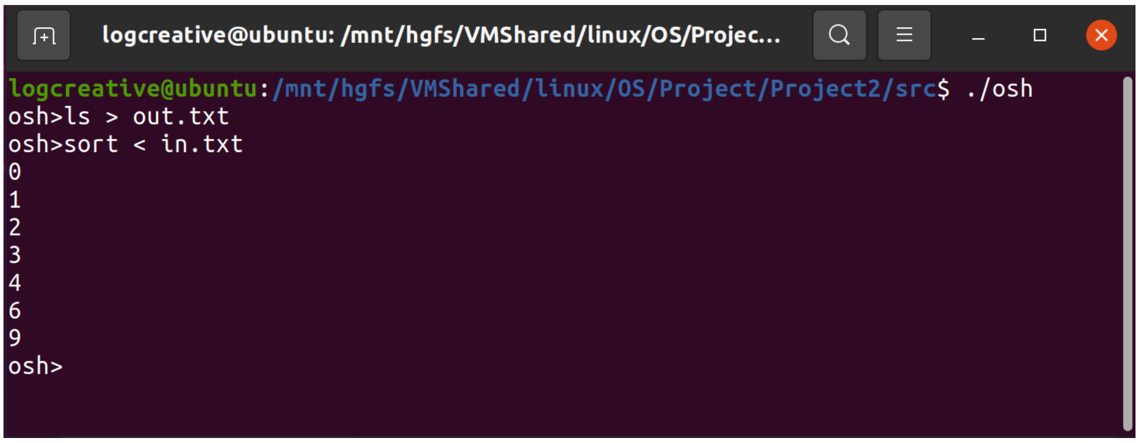
3. 重定向输入输出

in.txt 里面存有一些乱序的数字

Listing 1: src/in.txt

```
3
4
2
```

```
9
6
1
0
```



```
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ ./osh
osh>ls > out.txt
osh>sort < in.txt
0
1
2
3
4
6
9
osh>
```

out.txt 的文件将命令的输出存储。

Listing 2: src/out.txt

```
in.txt
Makefile
osh
osh.c
osh.exe
out.txt
```

首先需要识别输入输出符号

```
        case '>':
            file_mode = O_WRONLY;
            args[argc++] = NULL;
            prev = ' ';
            file_arg = argc;
            break;
        case '<':
            file_mode = O_RDONLY;
            args[argc++] = NULL;
            prev = ' ';
            file_arg = argc;
            break;
```

其中 file_mode 存储打开文件的模式：O_WRONLY 以只写方式打开文件，等价为 1；O_RDONLY 以只读方式打开文件，等价为 0。file_arg 存储文件名的位置，便于后续调用。而作为命令而言，命令应当在输入输出符号之前终结，因此这里需要添加示意终结的符号 NULL；并且将输入输出符号识别为分隔符号，令为空格。

当识别文件模式大于等于 0 时，就会在子进程中截获控制台输入与输出。在写入模式下，如果文件不存在，文件会被创建。值得一提的是，使用 dup2 截获子进程的控制台输入输出并不会影响主进程的输入输出，所以就没有必要去使用一个变

量存储主进程的输入输出文件描述符，再赋值回去。子进程需要在完成使用后关闭文件。

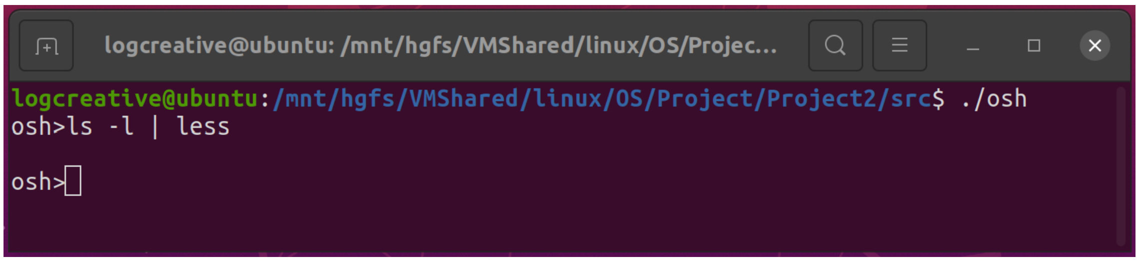
```
// child process
if(file_mode >= 0){
    fd = open(args[file_arg], file_mode, PERMS);
    if(file_mode == O_WRONLY){
        if(fd == -1) fd = creat(args[file_arg], PERMS);
        if(fd == -1) fprintf(stderr, "%s %s.\n", "Cannot write file", args[file_arg]);
        else dup2(fd, STDOUT_FILENO);
    } else if (file_mode == O_RDONLY){
        if(fd == -1) fprintf(stderr, "%s %s.\n", "Cannot read file",args[file_arg]);
        else dup2(fd, STDIN_FILENO);
    }
}
execvp(args[0],args);
if(fd != -1) close(fd);
return 0;
```

其中 PERMS 被定义为对于所有者、所有者组和其他成员均可读写。

```
#define PERMS 0666
```

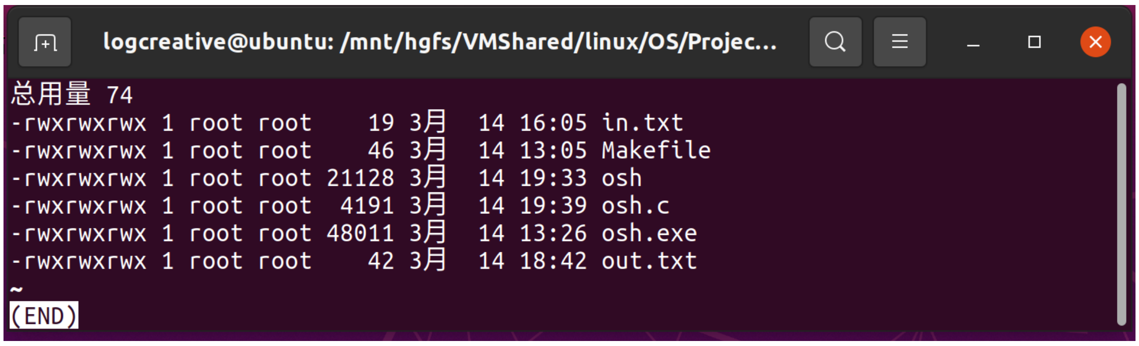
4. 通过管道建立通信

输入管道命令



```
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ ./osh
osh>ls -l | less

osh>
```

可以看到数值被正确传入了后一个命令中。



```
总用量 74
-rwxrwxrwx 1 root root    19 3月  14 16:05 in.txt
-rwxrwxrwx 1 root root    46 3月  14 13:05 Makefile
-rwxrwxrwx 1 root root 21128 3月  14 19:33 osh
-rwxrwxrwx 1 root root  4191 3月  14 19:39 osh.c
-rwxrwxrwx 1 root root 48011 3月  14 13:26 osh.exe
-rwxrwxrwx 1 root root    42 3月  14 18:42 out.txt
~
(END)
```

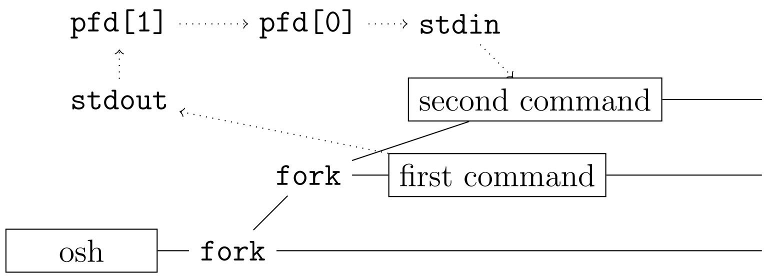
首先仍然是识别管道符号：

```
case '|':
    args[argc++] = NULL;
    prev = ' ';
    pipe_arg = argc;
    break;
```

其中 pipe_arg 将会存储管道命令开始的位置，当其不为小于 0 的值时，将会触发子进程再次新建一个子进程，并通过管道连接两者。

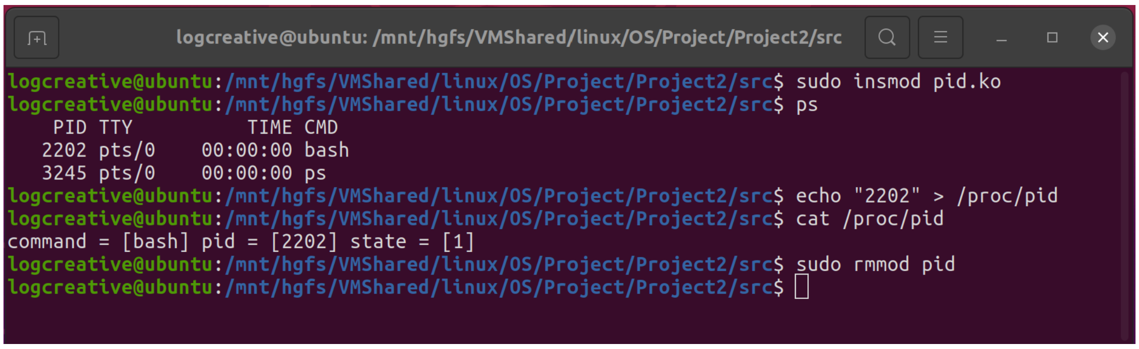
```
        if(pipe_arg < 0){
            // original child commands ...
        } else {
            int pfd[2];
            pid_t ppid;

            if (pipe(pfd) == -1){
                fprintf(stderr, "Pipe Failed");
                continue;
            }

            ppid = fork();

            if(ppid < 0){
                fprintf(stderr, "Child Fork Failed");
                continue;
            }

            if(ppid > 0){
                close(pfd[READ_END]);
                dup2(pfd[WRITE_END], STDOUT_FILENO);
                execvp(args[0],args);
                close(pfd[WRITE_END]);
            } else {
                close(pfd[WRITE_END]);
                dup2(pfd[READ_END], STDIN_FILENO);
                execvp(args[pipe_arg],&args[pipe_arg]);
                close(pfd[READ_END]);
            }
        }
```

子进程将前一个命令的输出输入管道的写入端，子子进程将管道端的读入部分作为自己的输入。这样就完成了管道通信。



二 任务信息内核模块

安装模块 pid 后，将进程号输入文件，读取时将会展示进程的相关信息。


logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Project2/src

```
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ sudo insmod pid.ko
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ ps
    PID TTY          TIME CMD
   2202 pts/0    00:00:00 bash
   3245 pts/0    00:00:00 ps
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ echo "2202" > /proc/pid
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ cat /proc/pid
command = [bash] pid = [2202] state = [1]
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ sudo rmmod pid
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project2/src$ 
```

1. 写入文件

   读取到用户信息后，使用 sscanf 获取变量并赋值到模块变量 l_pid 中。

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

2. 读取文件

   当获取到 tsk 变量时，将会返回进程的相关信息，通过 sprintf 读取到 buffer 中，稍后将会被返回到用户区，注意此处需要暂存返回的字符串长度到变量 kv 中以便返回用户区时系统知晓需要读取的字符数。

```
        if(tsk!=NULL)
            rv = sprintf(buffer,"command = [%s] pid = [%ld] state = [%ld]\n",tsk->comm,l_pid,tsk->
                state);
        else
            rv = sprintf(buffer,"pid [%ld] is not found!\n",l_pid);
```

A 全部代码

Listing 3: src/Makefile

```
obj-m:=pid.o
KDIR:=/lib/modules/$(shell uname -r)/build
PWD:=$(shell pwd)

all:
    gcc -g osh.c -o osh
    make -C $(KDIR) M=$(PWD) modules
clean:
    rm *.o *.ko *.mod.c Modules.symvers modules.order -f
```

Listing 4: src/osh.c

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

#define MAX_LINE 80
#define PERMS 0666
```

```c
#define READ_END 0
#define WRITE_END 1

int main(void){
    char *args[MAX_LINE/2 + 1];
    int should_run = 1;

    char buf[MAX_LINE] = "\0";

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

        int should_wait = 1;
        char rline[MAX_LINE];
        fgets(rline, MAX_LINE, stdin);

        int flag = 0;

        if (strcmp(rline,"!!\n")==0){
            if(!buf[0]){
                fprintf(stdout, "%s\n", "No commands in history.");
                flag = -1;
            } else {
                memcpy(rline, buf, MAX_LINE*sizeof(char));
                fprintf(stdout, "%s", rline);
            }
        } else memcpy(buf, rline, MAX_LINE*sizeof(char));

        int pipe_arg = -1;

        int fd = -1;
        int file_mode = -1;
        int file_arg = -1;

        int argc = 0;
        char prev = ' ';

        for(char* rptr = rline; rptr < rline + MAX_LINE && !flag; ++rptr){
            switch (*rptr) {
                case '&':
                    should_wait = 0;
                case '\n':
                    flag = 1;
                case ' ':
                    prev = *rptr;
                    *rptr = '\0';
                    break;
                case '>':
                    file_mode = O_WRONLY;
                    args[argc++] = NULL;
                    prev = ' ';
                    file_arg = argc;
                    break;
                case '<':
                    file_mode = O_RDONLY;
                    args[argc++] = NULL;
                    prev = ' ';
                    file_arg = argc;
                    break;
                case '|':
                    args[argc++] = NULL;
                    prev = ' ';
                    pipe_arg = argc;
                    break;
                default:
                    if (prev == ' ') args[argc++] = rptr;
                    prev = *rptr;
                    break;
            }
        }

        if(flag == -1) continue;

        args[argc] = NULL;
        if (!args[0]) continue;
        should_run = strcmp(args[0],"exit");

        pid_t pid;
        pid = fork();

        if (pid < 0) fprintf(stderr, "Fork Failed");
        else if (pid == 0) {

            if(pipe_arg < 0){

                if(file_mode >= 0){
                    fd = open(args[file_arg], file_mode, PERMS);
                    if(file_mode == O_WRONLY){
                        if(fd == -1) fd = creat(args[file_arg], PERMS);
                        if(fd == -1) fprintf(stderr, "%s %s.\n", "Cannot write file", args[file_arg]);
                        else dup2(fd, STDOUT_FILENO);
                    } else if (file_mode == O_RDONLY){
                        if(fd == -1) fprintf(stderr, "%s %s.\n", "Cannot read file",args[file_arg]);
                        else dup2(fd, STDIN_FILENO);
                    }
                }

                execvp(args[0],args);
                if(fd != -1) close(fd);

            } else {

                int pfd[2];
                pid_t ppid;

                if (pipe(pfd) == -1){
                    fprintf(stderr, "Pipe Failed");
                    continue;
                }

                ppid = fork();
```

```c
                if(ppid < 0){
                    fprintf(stderr, "Child Fork Failed");
                    continue;
                }

                if(ppid > 0){
                    close(pfd[READ_END]);
                    dup2(pfd[WRITE_END], STDOUT_FILENO);
                    execvp(args[0],args);
                    close(pfd[WRITE_END]);
                } else {
                    close(pfd[WRITE_END]);
                    dup2(pfd[READ_END], STDIN_FILENO);
                    execvp(args[pipe_arg],&args[pipe_arg]);
                    close(pfd[READ_END]);
                }

            }
            return 0;
        }
        else if (should_wait) wait(NULL);

    }

    return 0;
}
```

Listing 5: src/pid.c

```c
/**
 * Kernel module that communicates with /proc file system.
 *
 * This provides the base logic for Project 2 - displaying task information
 */

#include <linux/init.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/vmalloc.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "pid"

/* the current pid */
static long l_pid;

/**
 * Function prototypes
 */
static ssize_t proc_read(struct file *file, char *buf, size_t count, loff_t *pos);
static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t count, loff_t *pos);
static struct proc_ops proc_ops = {
    .proc_read = proc_read,
    .proc_write = proc_write
};

/* This function is called when the module is loaded. */
static int proc_init(void)
{
        // creates the /proc/procfs entry
        proc_create(PROC_NAME, 0666, NULL, &proc_ops);

        printk(KERN_INFO "/proc/%s created\n", PROC_NAME);

    return 0;
}

/* This function is called when the module is removed. */
static void proc_exit(void)
{
        // removes the /proc/procfs entry
        remove_proc_entry(PROC_NAME, NULL);

        printk( KERN_INFO "/proc/%s removed\n", PROC_NAME);
}

/**
 * This function is called each time the /proc/pid is read.
 *
 * This function is called repeatedly until it returns 0, so
 * there must be logic that ensures it ultimately returns 0
 * once it has collected the data that is to go into the
 * corresponding /proc file.
 */
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!=NULL)
                rv = sprintf(buffer,"command = [%s] pid = [%ld] state = [%ld]\n",tsk->comm,l_pid,tsk->state);
        else
                rv = sprintf(buffer,"pid [%ld] is not found!\n",l_pid);

        completed = 1;

        // copies the contents of kernel buffer to userspace usr_buf
        if (copy_to_user(usr_buf, buffer, rv)) {
```

```c
                rv = -1;
        }

        return rv;
}

/**
 * This function is called each time we write to the /proc file system.
 */
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;
        }

    /**
     * kstrol() will not work because the strings are not guaranteed
     * to be null-terminated.
     *
     * sscanf() must be used instead.
     */

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

        kfree(k_mem);

        return count;
}

/* Macros for registering module entry and exit points. */
module_init( proc_init );
module_exit( proc_exit );

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Pid Module");
MODULE_AUTHOR("LogCreative");
```