Project2: UNIX Shell Programming & Linux Kernel Module for Task Information

518021911193 刘昊林

1 使用到的功能、函数和命令

1.1 创建子进程的基本代码结构

```
int main()
{
        pid_t pid;
        pid=fork();
        if (pid<0) {
                fprintf(stderr,"Fork Failed");
                return 1;
        }
        else if (pid==0) {
                /*child process*/
                do something here
        }
        else {
                /*parent process*/
                wait(NULL);
                do something here
        }
}
```

1.2 使用 pipe 创建进程间通信的基本代码结构

```
int main()
{
        pid_t pid;
        int fd[2];
        if (pipe(fd)==-1) {
```

```
6                       fprintf(stderr,"Pipe Failed");
7                       return 1;
8               }
9
10              pid=fork();
11              if (pid<0) {
12                      fprintf(stderr,"Fork Failed");
13                      return 1;
14              }
15              else if (pid==0) {
16                      /*child process*/
17                      /* close the unused end of the pipe */
18                      close(fd[WRITE_END]);
19
20                      /* read from the pipe */
21                      read(fd[READ_END], read_msg, BUFFER_SIZE);
22                      printf("child read %s\n",read_msg);
23
24                      /* close the write end of the pipe */
25                      close(fd[READ_END]);
26              }
27              else {
28                      /*parent process*/
29                      /* close the unused end of the pipe */
30                      close(fd[READ_END]);
31
32                      /* write to the pipe */
33                      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
34
35                      /* close the write end of the pipe */
36                      close(fd[WRITE_END]);
37              }
38      }
```

1.3 编写/proc 文件系统的基本结构

与 project1 中的结构类似，但是在 struct file_operations proc_ops 中加入了.write=proc_write 这一参数，以及增加了一个 proc_write() 函数，用来实现对文件系统的写入功能。

1.4 使用到的函数

1. fork() 函数，用来创建单独的子进程。

2. wait() 函数，用来使父进程等待子进程运行完成。

3. dup2(int oldfd, int newfd) 函数，将现有文件描述符复制到另一个文件描述符，使得两个文件描述符都指向同一个文件，并且是函数第一个参数指向的文件。

4. pipe() 函数，创建两个进程间的通信管道。

5. execvp(char *command, char *params[]) 函数，其中 comand 表示要执行的命令，params 表示将参数存储到该命令中。

6. strcmp() 函数，比较两个字符串是否一样。

7. strcpy() 函数，进行字符串间的复制。

8. snprintf() 函数，将可变个参数按照 format 格式化成字符串。

9. int kstrtol(const char *str, unsigned int base, long *res) 函数，将参数 str 字符串根据参数 base 来转换成长整型数 (long)。

10. kmalloc() 和 kfree() 函数，用来在内存中动态分配内存和释放内存。

11. struct task_struct pid_task(struct_pid *pid,enum pid_type type) 函数，根据进程标识符，获取进程信息 task_struct。

12. find_vpid(int pid) 函数，根据 pid 找到对应的 struct pid 结构体。

1.5 使用到的命令

1. make，利用 Makefile 编译文件。

2. ./shell_simulator，运行编译好的可执行文件。

3. ls -l，列出当前目录下的文件信息。

4. !!，使用上次使用的命令。

5. ls -l > output.txt，将 ls -l 命令的输出内容重定向到 output.txt 中。

6. sort < output.txt，使用 output.txt 中的内容作为 sort 排序的对象。

7. ls -l |sort，将 ls -l 命令的结果作为 sort 排序的对象。

8. ./shell_simulator &，两个进程同时运行 shell 模拟程序。

9. sudo insmod simple.ko，加载 simple 内核模块。

10. sudo rmmod simple，删除 simple 内核模块

11. dmesg，在内核日志缓冲区中检查输出的消息。

12. dmesg -c，清空缓冲区。

13. echo ”1” > /proc/pid，将 1 输入到/proc 文件系统中去。

14. cat /proc/hello，连接/proc/hello 文件并打印到输出设备上。

2 实现思路

2.1 Part 1-UNIX Shell Programming

首先设置一个字符串数组 input,存放输入的原始命令字符串数组;一个动态二维数组 *args[MAX_LINE/2 + 1] 用来存放划分好的命令字符串；整型数 num_of_args，用来记录命令字符串可以划分为多少部分。然后我设计了一个命令字符串预处理函数 int init_args(char *input,char *args[],int num_of_args)，在 input 数组中以空格为依据，对命令进行划分并且记录在 args 数组中，同时返回划分的数量 num_of_args。考虑到命令中不同部分之间可能有不止一个空格，所以在函数中我采用了双标记位来记录下非空格与空格的位置来解决这个问题。

获取到已经划分好的数组 args 后，首先判断数组的最后一个元素是否为'exit'，如果是，则直接退出程序；否则，就继续运行下去。再判断一下最后一个元素是否为'!!'，如果是，则输出上一条执行的指令并且再执行一次；否则，就继续运行下去。在创建历史功能时，需要一个字符串数组 last_input 用来记录上一条指令。每当输入一个指令时，last_input 都会记录下这个指令，作为下一条指令的历史记录。如果命令为'!!'，但是 last_input 数组为空，那么输出错误信息；否则，就执行上一条指令，并且用一个新数组 *cmd_args[MAX_LINE/2 + 1] 记录去掉最后一项的数组 args。

接下来遍历一遍数组 args。如果数组最后一项为'&'，那么设置 wait_flag=0，表明父进程与子进程并发执行，并且用一个新数组 cmd_args 记录去掉最后一项 & 后的数组 args; 如果在数组中找到了'>',那么用字符串数组 redirect_path 记录下重定向后的位置,用数组 out_args 记录下重定向之前的命令，设置 out_redirect_flag=1; 如果在数组中找到了'<'，那么用字符串数组 redirect_path 记录下重定向后的位置，用数组 in_args 记录下重定向之前的命令, 设置 in_redirect_flag=1; 如果在数组中找到了'|'，那么设置 pipe_flag=1，表示需要进行进程间通信，再分别用 first_args 和 second_args 数组，记录下管道两侧的命令。

在对命令的预处理完成后，创建一个子进程来处理具体的指令。在子进程中，如果 out_redirect_flag=1 或者 in_redirect_flag=1，那么先把输入输出文件根据 redirect_path 进行重定向。然后判断 pipe_flag 的值，如果为 0，那么就根据 out_redirect_flag，in_redirect_flag 的值，运行 out_args 或 in_args 或 cmd_args; 如果为 1，那么在子进程中再创建一个孙子进程，并且用 pipe 管道连接，同时用 dup2() 函数对输入输出重定向，最后在子进程中运行 second_args，在孙子进程中运行 first_args 即可。而在父进程中则根据 wait_flag 的值决定是等待子进程完成再执行还是与子进程并行执行。

具体的代码实现在文件 shell_simulator.c 中。

2.2 Part 2-Linux Kernel Module for Task Information

大体思路与上一个 project 相似，只要在 struct file_operations proc_ops 中加入了.write=proc_write 这一参数，以及增加了一个 proc_write() 函数，用来实现对文件系统的写入功能即可。

在 proc_read() 函数中，要添加根据进程的 pid 获取进程以 task_struct 结构体存储的进程具体信息，并且把信息写入用户缓冲区的功能。

在 proc_write() 函数中，要把输入的进程 pid，存储进入内核的缓冲区和全局变量 current_pid 中即可。

具体的代码实现在文件 pid.c 中。

3 步骤截图

3.1 Part 1-UNIX Shell Programming

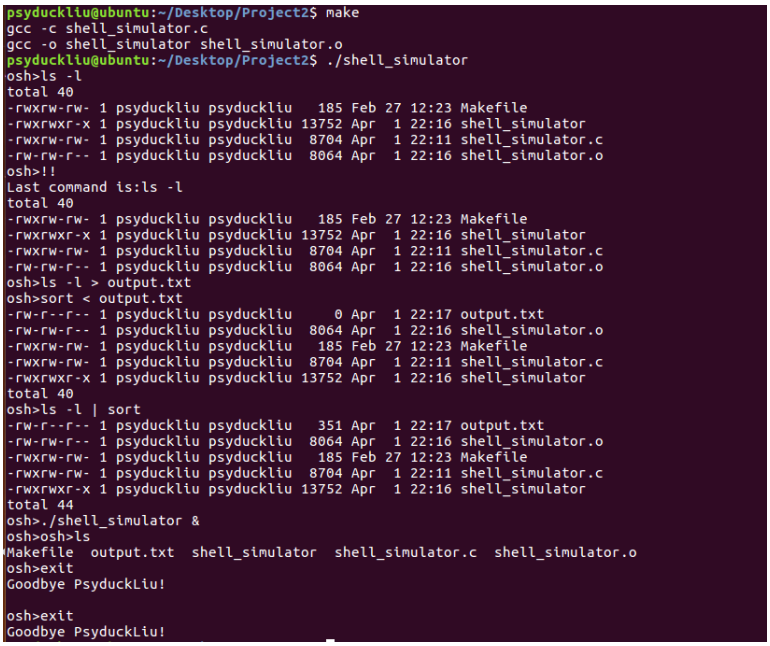

图 1: Part 1-UNIX Shell Programming

3.2 Part 2-Linux Kernel Module for Task Information

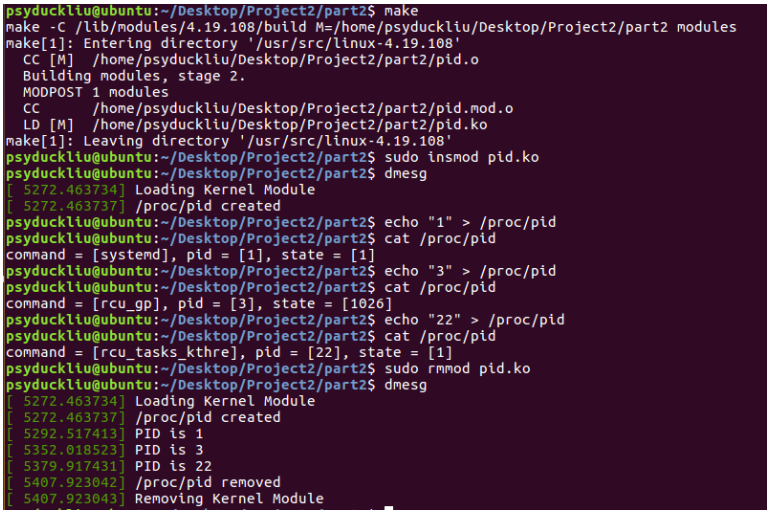

图 2: Part 2-Linux Kernel Module for Task Information

4 实验中遇到的问题

在 Part1 中，面对动态内存的分配与回收我遇到了一些问题。理论上来讲每当 malloc() 函数分配内存出去，一定要有 free() 函数回收内存，但是编译器却提示我有这样一个错误：

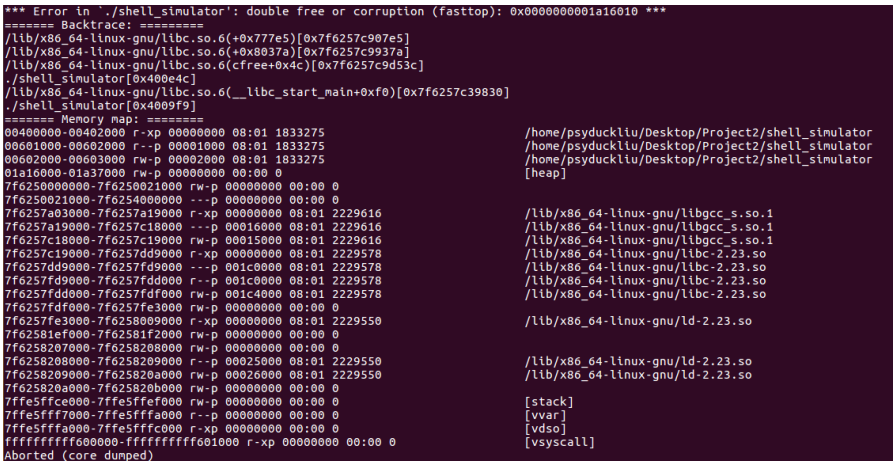

图 3: error

提示我两次释放同一空间，我猜测可能是多个子进程之间或者并行运行的进程之间的数据共享问题。所以我删去了 free() 函数，这个问题便消失了。但是，这种做法是将内存的管理交给了编译器自行处理，我觉得并不是很好。希望老师可以给我一些帮助。