Project2: UNIX Shell & Linux Kernel Module for Task Information

Yanjie Ze 519021910706

Project 2-1: UNIX Shell

0 Introduction

Project 2-1: Unix Shell guides us to finish a C program simulating **UNIX Shell**. We will make use of many system calls, including fork(), exec(), wait(), dup2(), and pipe().

Since we use the UNIX system call, the project can be finished in **Unix, macOS, Linux**.

My implementation is based on **Ubuntu 20.04**.

1 Overview

1.1 Real Shell

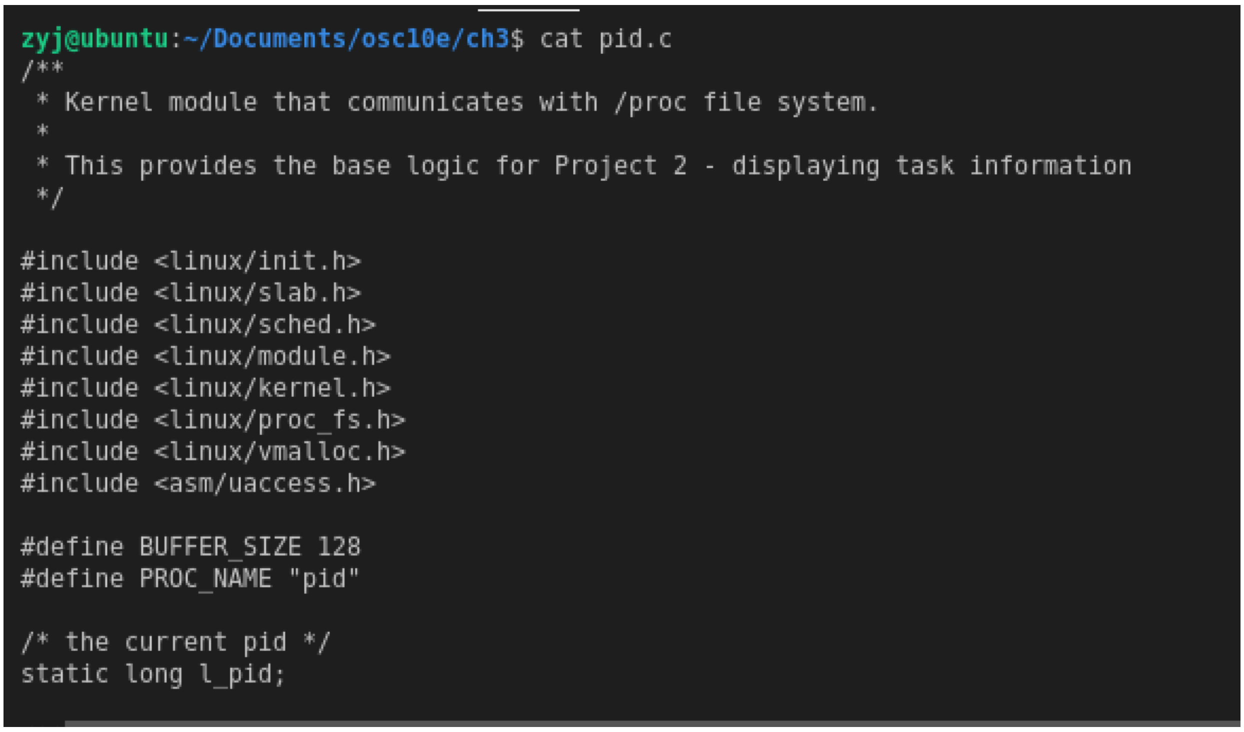
A shell interface gives the user a prompt, after which the next command is entered.

We enter:

```
cat pid.c
```

to make the shell display the contents of **pid.c** .

The result:

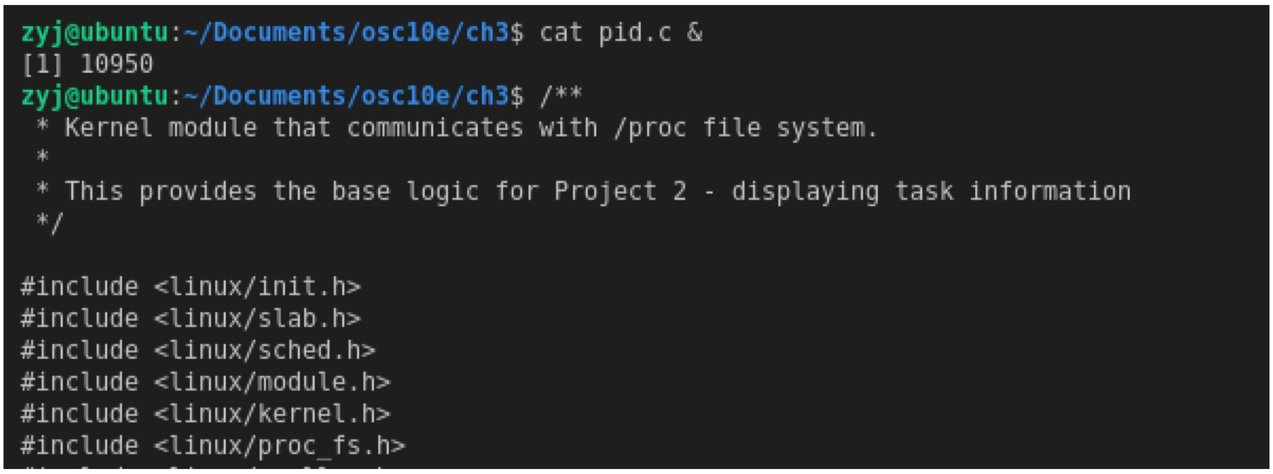


```
zyj@ubuntu:~/Documents/osc10e/ch3$ cat pid.c
/**
 * Kernel module that communicates with /proc file system.
 *
 * This provides the base logic for Project 2 - displaying task information
 */

#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;
```

We can also make the command run in the background, use **&**:

```
cat pid.c &
```

The result:



```
zyj@ubuntu:~/Documents/osc10e/ch3$ cat pid.c &
[1] 10950
zyj@ubuntu:~/Documents/osc10e/ch3$ /**
 * 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>
```

1.2 Implementation Requirements of Simple Shell

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, **cat pid.c**) and then create a separate child process that performs the command.

What's more, we can implement the function of **&**, which is to make the command run in the background.

The template of **Shell** is given:

```c
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE    80 /* 80 chars per line, per command */

int main(void)
{
  char *args[MAX_LINE/2 + 1]; /* command line (of 80) has max of 40
arguments */
      int should_run = 1;

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

          /**
           * After reading user input, the steps are:
           * (1) fork a child process
           * (2) the child process will invoke execvp()
           * (3) if command included &, parent will invoke wait()
           */
      }

  return 0;
}
```

In conclusion, we need to implement several parts:

1. Creating the child process and executing the command in the child
2. Providing a history feature
3. Adding support of input and output redirection
4. Allowing the parent and child processes to communicate via a pipe

2 Executing Command in a Child Process

2.1 Requirements

We need to modify **main()** so that a child process is forked and executes the command specified by the user.

This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings.

Then we pass the command and the parameters to the function below to execute the command:

```
execvp(char *command, char *params[])
```

We also need to implement the function of **&**, ensuring the child process run in the background.

2.2 Implementation Details and Methods

2.2.1 Parsing

The basic thing of implementation is to parse the command into several seperating parts.

We use a single loop to parse the command based on whether the current character is **'\n'** or **' '** (newline and blank).

- **'\n'** is used as the break of a command.
- **' '** (blank) is used as the sepeation signal of a command.

We store the parsing results in a two-dimensional array . The first dimension is to represent each argument. The second dimension is to represent each character in one argument.

The code of parsing is shown below.

```
    //parsing
        int arg_num=0;
        for(int i=0;input_sentence[i]!='\n';)
        {
            args[arg_num] = (char*)malloc(sizeof(char)*20);
            int j=0;
            for(; input_sentence[i]!='\n'&&input_sentence[i]!=' ';++j,++i){
                args[arg_num][j]=input_sentence[i];
            }
            args[arg_num][j]='\0';

            arg_num++;
            if(input_sentence[i]=='\n')
                break;
            if(input_sentence[i]==' ')
                i ++;
        }
        arg_num--;
```

2.2.2 Code Frame

Then we can consider how to realize the child/parent process .

First, we use a simple logic **If Else** to determine whether the process is a child process or a parent process. This is also the basic frame of Shell.

```
  if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
                }
  else if (pid == 0) //child
  {
    /* execution*/
  }
else //parent
{
  /* wait or not */
}
```

2.2.3 Add '&' judgement

We add a judgement in the parent process:

```
  if(strcmp(args[arg_num],"&")!=0){
                    wait(NULL);
                }
```

If the command has a **&** in the **last argument**, the parent process will wait for the child process until the child process finishes execution.

2.2.4 Add "exit" command

Shell we implement supports **exit** command. We can implement this after parsing.

Define a variable called **stop_word** , to add the readability of codes.

```
 char stop_word[]="exit";
```

Add one judgment to judge whether the first argument is equal to **"exit"**.

```
if(strcmp(args[0], stop_word)==0) break;
```

2.2.5 Execute

Since there are many situations to be considered in the following part, the simplest case is shown here.

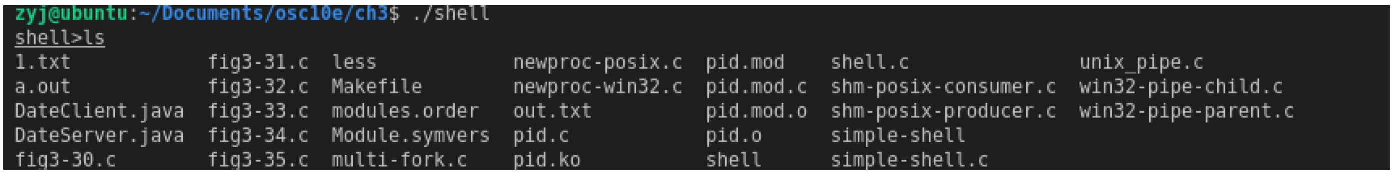
In this case, we just execute the command.

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

2.3 Program Results

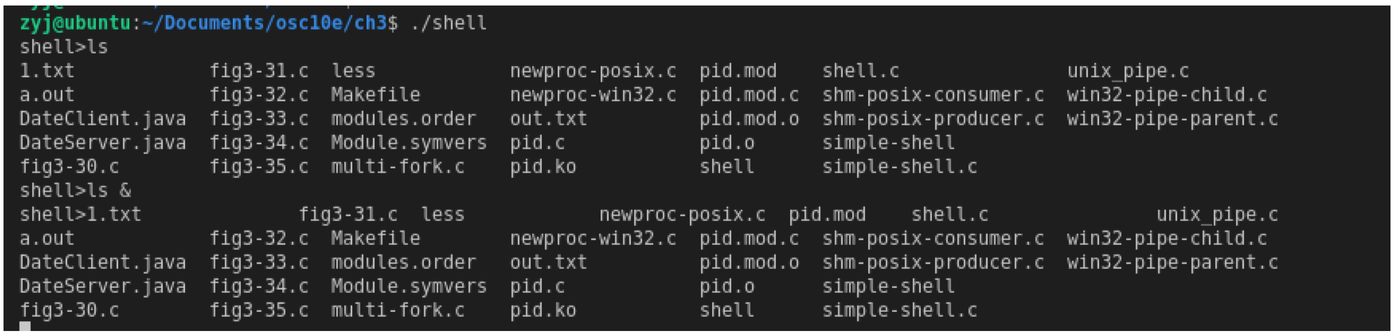
We test whether the simple commands can work and whether **&** can function properly.

```
ls
```



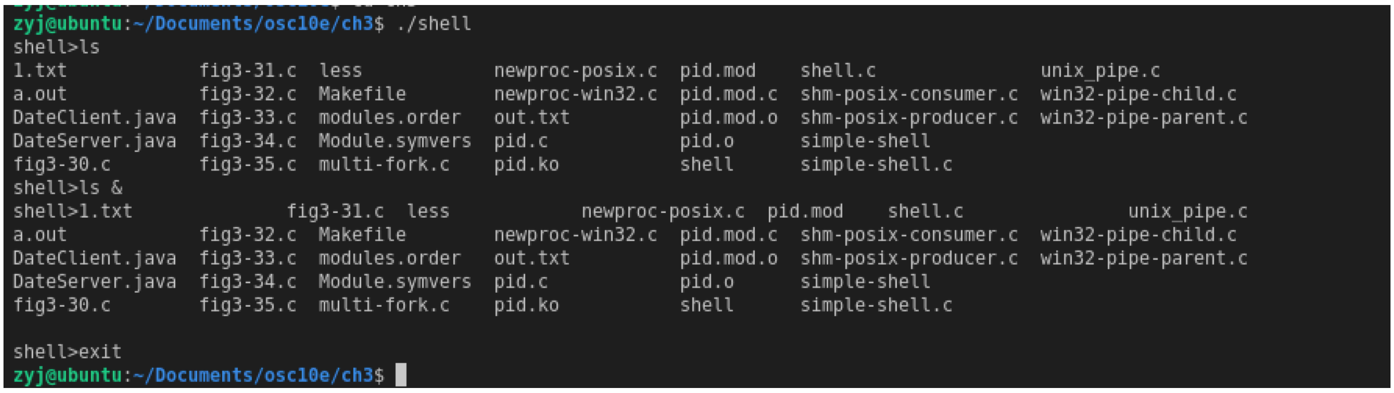
```
zyj@ubuntu:~/Documents/osc10e/ch3$ ./shell
shell>ls
1.txt           fig3-31.c  less           newproc-posix.c  pid.mod    shell.c               unix_pipe.c
a.out           fig3-32.c  Makefile       newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java fig3-33.c  modules.order  out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java fig3-34.c  Module.symvers pid.c            pid.o      simple-shell
fig3-30.c       fig3-35.c  multi-fork.c   pid.ko           shell      simple-shell.c
```

```
ls &
```



```
zyj@ubuntu:~/Documents/osc10e/ch3$ ./shell
shell>ls
1.txt           fig3-31.c  less           newproc-posix.c  pid.mod    shell.c               unix_pipe.c
a.out           fig3-32.c  Makefile       newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java fig3-33.c  modules.order  out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java fig3-34.c  Module.symvers pid.c            pid.o      simple-shell
fig3-30.c       fig3-35.c  multi-fork.c   pid.ko           shell      simple-shell.c
shell>ls &
shell>1.txt           fig3-31.c  less           newproc-posix.c  pid.mod    shell.c               unix_pipe.c
a.out           fig3-32.c  Makefile       newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java fig3-33.c  modules.order  out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java fig3-34.c  Module.symvers pid.c            pid.o      simple-shell
fig3-30.c       fig3-35.c  multi-fork.c   pid.ko           shell      simple-shell.c
```

```
exit
```



```
zyj@ubuntu:~/Documents/osc10e/ch3$ ./shell
shell>ls
1.txt           fig3-31.c  less           newproc-posix.c  pid.mod    shell.c             unix_pipe.c
a.out           fig3-32.c  Makefile       newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java  fig3-33.c  modules.order  out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java  fig3-34.c  Module.symvers  pid.c            pid.o      simple-shell
fig3-30.c       fig3-35.c  multi-fork.c   pid.ko           shell      simple-shell.c
shell>ls &
shell>1.txt             fig3-31.c  less           newproc-posix.c  pid.mod    shell.c             unix_pipe.c
a.out           fig3-32.c  Makefile       newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java  fig3-33.c  modules.order  out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java  fig3-34.c  Module.symvers  pid.c            pid.o      simple-shell
fig3-30.c       fig3-35.c  multi-fork.c   pid.ko           shell      simple-shell.c

shell>exit
zyj@ubuntu:~/Documents/osc10e/ch3$
```

3 Creating a History Feature

3.1 Requirements

We need to modify the shell interface program so that it provides a **history** feature to allow a user to execute the most recent command by entering **!!** . For example, if a user enters the command ls -l, she can then execute that command again by entering !! at the prompt. Any command executed in this fashion should be echoed on the user's screen, and the command should also be placed in the history buffer as the next command.

If there is no history, entering !! should result in a message **"No commands in history."**

3.2 Implementation Details and Methods

3.2.1 Save History Command

We need to create a new array to store the history command:

```
char history_sentence[81];
```

Then at the end of the command's execution, we will store the command in the history command:

```
 //save history
        int m = 0;
        for(; input_sentence[m]!='\n';
++m)history_sentence[m]=input_sentence[m];
        history_sentence[m]='\n';
        history_sentence[m+1]='\0';
        history_exist = 1;
```

3.2.2 Use History Command

When we enter "!!" Command, Shell should run the command in the history.

Note that when there exists no history, entering "!!" will make Shell output "No commands in history!\n"

Therefore, in the implementation, we first detect whether the command is "!!".

If so, we judge whether the history command exists based on the variable **hitory_exit** .

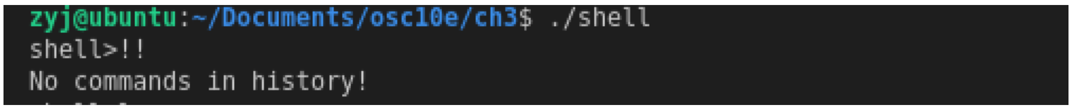
If the history command exists, we can copy the history command into the current command. E**ach time we do a copy, we display the copy**. This makes us know the history command that "!!" command calls.

The implementation is shown below.

```
if(input_sentence[0]=='!'&&input_sentence[1]=='!'&&input_sentence[2]=='\n')
{
            if(history_exist==0){
                printf("No commands in history!\n");
                continue;
                }
            else{
                int k=0;
                for(int w=0;history_sentence[w]!='\n';++w)
                {
                    printf("%c", history_sentence[w]);
                }
                printf("\n");
                for(;history_sentence[k]!='\n';k++)
                    input_sentence[k]=history_sentence[k];
                input_sentence[k]='\n';
                input_sentence[k+1]='\0';
            }
        }
```

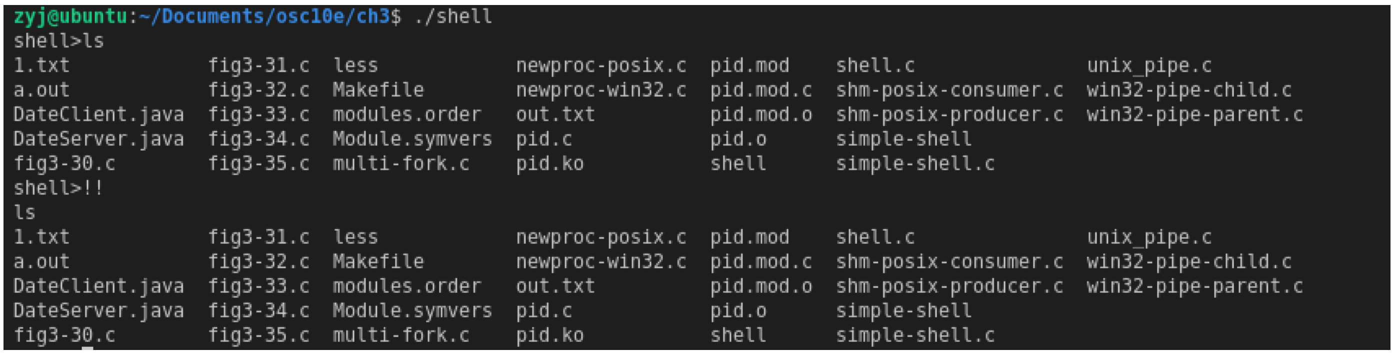
3.3 Program Results

When there's no history command:



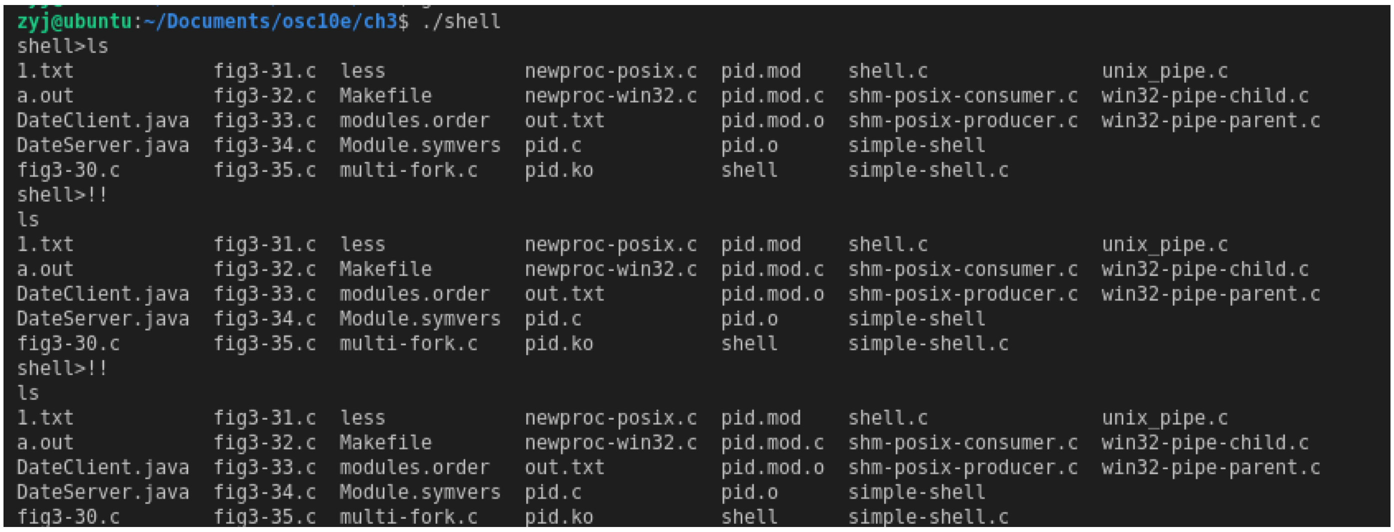
```
zyj@ubuntu:~/Documents/osc10e/ch3$ ./shell
shell>!!
No commands in history!
```

When there exists a history command:



```
zyj@ubuntu:~/Documents/osc10e/ch3$ ./shell
shell>ls
1.txt            fig3-31.c  less            newproc-posix.c  pid.mod    shell.c               unix_pipe.c
a.out            fig3-32.c  Makefile        newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java  fig3-33.c  modules.order   out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java  fig3-34.c  Module.symvers  pid.c            pid.o      simple-shell
fig3-30.c        fig3-35.c  multi-fork.c    pid.ko           shell      simple-shell.c
shell>!!
ls
1.txt            fig3-31.c  less            newproc-posix.c  pid.mod    shell.c               unix_pipe.c
a.out            fig3-32.c  Makefile        newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java  fig3-33.c  modules.order   out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java  fig3-34.c  Module.symvers  pid.c            pid.o      simple-shell
fig3-30.c        fig3-35.c  multi-fork.c    pid.ko           shell      simple-shell.c
```

When we enter "!!" for two times:



```
zyj@ubuntu:~/Documents/osc10e/ch3$ ./shell
shell>ls
1.txt            fig3-31.c  less            newproc-posix.c  pid.mod    shell.c               unix_pipe.c
a.out            fig3-32.c  Makefile        newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java  fig3-33.c  modules.order   out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java  fig3-34.c  Module.symvers  pid.c            pid.o      simple-shell
fig3-30.c        fig3-35.c  multi-fork.c    pid.ko           shell      simple-shell.c
shell>!!
ls
1.txt            fig3-31.c  less            newproc-posix.c  pid.mod    shell.c               unix_pipe.c
a.out            fig3-32.c  Makefile        newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java  fig3-33.c  modules.order   out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java  fig3-34.c  Module.symvers  pid.c            pid.o      simple-shell
fig3-30.c        fig3-35.c  multi-fork.c    pid.ko           shell      simple-shell.c
shell>!!
ls
1.txt            fig3-31.c  less            newproc-posix.c  pid.mod    shell.c               unix_pipe.c
a.out            fig3-32.c  Makefile        newproc-win32.c  pid.mod.c  shm-posix-consumer.c  win32-pipe-child.c
DateClient.java  fig3-33.c  modules.order   out.txt          pid.mod.o  shm-posix-producer.c  win32-pipe-parent.c
DateServer.java  fig3-34.c  Module.symvers  pid.c            pid.o      simple-shell
fig3-30.c        fig3-35.c  multi-fork.c    pid.ko           shell      simple-shell.c
```

4 Redirecting Input and Output

4.1 Requirements

Shell should then be modified to support the ‘>’ and ‘<’ redirection where ‘>’ redirects the output of a command to a file and ‘<’ redirects the input to a command from a file.

For example, if a user enters:

```
osh>ls > out.txt
```

the output from the ls command will be redirected to the file out.txt. Similarly, input can be redirected as well. For example, if the user enters:

```
osh>sort < in.txt
```

the file in.txt will serve as input to the sort command.

Managing the redirection of both input and output will involve using the **dup2()** function, which duplicates an existing file descriptor to another file descriptor. For example, if fd is a file descriptor to the file out.txt, the call:

```
dup2(fd, STDOUT_FILENO);
```

duplicates fd to standard output (the terminal). This means that any writes to standard output will in fact be sent to the out.txt file.

4.2 Implementation Details and Methods

There's two situation: **>(greater than) and <(less than)**. We need to specify the solution to each situation.

4.2.1 Redirect Output

The command with ">" redirects the output into a file.

The file name is the last argument of the input command, which is **args[arg_num]**.

We use **int origin_point = dup(STDOUT_FILENO)** to record the output point in the command line. The origin point records the original position of **STDOUT_FILENO**. After we redirect all the output, we need to recover the position of **STDOUT_FILENO**.

We use **int file_write = open(args[arg_num], O_CREAT|O_RDWR|O_TRUNC, S_IRUSR|S_IWUSR)** to record the file descriptor. The parameter description of **open** can be seen in this website.

Then the process is:

1. We redirect **STDOUT_FILENO** to **file_write**, using **dup2(file_write, STDOUT_FILENO)**.
2. We make **args[arg_num-1]=NULL** and **args[arg_num]=NULL**, which is ">" and the file name.
3. We execute the command using **execvp(args[0], args)**.
4. We redirect **STDOUT_FILENO** back to **origin_point**, using **dup2(origin_point, STDOUT_FILENO)**.

The implementation code is shown below.

```
if(strcmp(args[arg_num-1],">")==0)
                    {
                        //situation >
                        int origin_point = dup(STDOUT_FILENO);
                        int file_write = open(args[arg_num],
O_CREAT|O_RDWR|O_TRUNC, S_IRUSR|S_IWUSR);
                        if (file_write==-1)
                            fprintf(stderr, "Create file Failed");
                        dup2(file_write, STDOUT_FILENO);
                        args[arg_num-1]=NULL;
                        args[arg_num]=NULL;
                        execvp(args[0], args);
                        close(file_write);
                        dup2(origin_point, STDOUT_FILENO);
                        exit(0);
                    }
```

4.2.2 Redirect Input

The process of redirect input is almost the same as the process of redirecting output.

The difference is that we use **STDIN_FILENO** to redirect the input.

Therefore, we will not repeat it again.

The implementation is shown below:

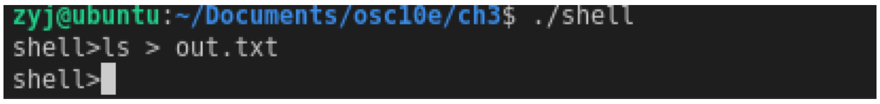
```
else if(strcmp(args[arg_num-1],"<")==0)
                    {
                        //situation <
                        int origin_point = dup(STDIN_FILENO);
                        int file_read = open(args[arg_num], O_RDONLY);
                        dup2(file_read, STDIN_FILENO);
                        args[arg_num-1]=NULL;
                        args[arg_num]=NULL;
                        execvp(args[0], args);
                        close(file_read);
                        dup2(origin_point, STDIN_FILENO);
                        exit(0);
                    }
```

4.3 Program Results

Enter:

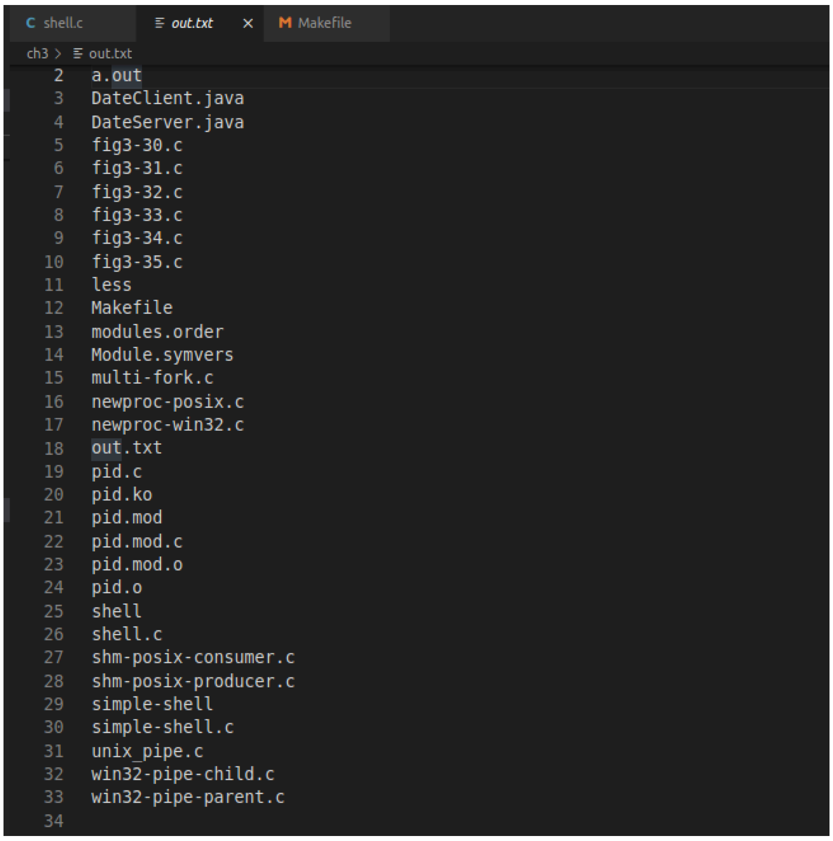
```
ls > out.txt
```

Result of Shell:



```
zyj@ubuntu:~/Documents/osc10e/ch3$ ./shell
shell>ls > out.txt
shell>
```

The file **out.txt**:

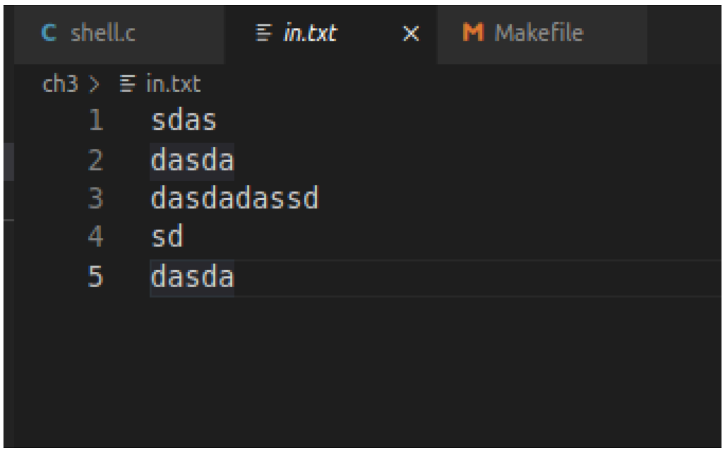


```
a.out
DateClient.java
DateServer.java
fig3-30.c
fig3-31.c
fig3-32.c
fig3-33.c
fig3-34.c
fig3-35.c
less
Makefile
modules.order
Module.symvers
multi-fork.c
newproc-posix.c
newproc-win32.c
out.txt
pid.c
pid.ko
pid.mod
pid.mod.c
pid.mod.o
pid.o
shell
shell.c
shm-posix-consumer.c
shm-posix-producer.c
simple-shell
simple-shell.c
unix_pipe.c
win32-pipe-child.c
win32-pipe-parent.c
```

Enter:

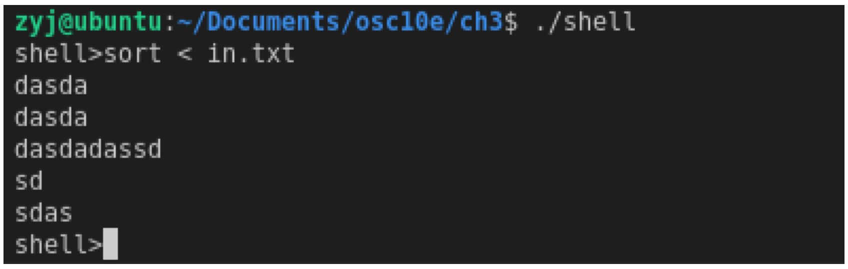
```
sort < in.txt
```

The file **in.txt** is:



```
sdas
dasda
dasdadassd
sd
dasda
```

The result of Shell:



```
zyj@ubuntu:~/Documents/osc10e/ch3$ ./shell
shell>sort < in.txt
dasda
dasda
dasdadassd
sd
sdas
shell>
```

5 Communication via a Pipe

5.1 Requirements

The final modification to Shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

```
osh>ls -l | less
```

has the output of the command ls -l serve as the input to the less command. Both the ls and less commands will run as separate processes and will communicate using the UNIX pipe() function described in Section 3.7.4.

5.2 Implementation Details and Methods

5.2.1 Divide Pipe Arguments

Since there are two commands actually, we divide them and use an extra string to store another command.

We use the following code to detect whether there's a "|" symbol.

- **pipe_exist** to represent the existence of "|"
- **pipe_position** to represent the position of "|"

```
// divide pipe args
        int pipe_exist=0;
        int pipe_position;
        for(int n=0;n<=arg_num;++n)
        {
            if(strcmp(args[n],"|")==0)
                {
                    pipe_exist = 1;
                    pipe_position = n;
                }

        }
```

If **pipe_exist==1**, we need to divide the second command out.

Simple copy works.

```
    char *pipe_args[MAX_LINE/2+1];

        if(pipe_exist==1)
        {

            // create pipe args
            for(int w=0;w<MAX_LINE/2+1;w++)pipe_args[w]=NULL;
            int total_num = arg_num;
            for(int w=pipe_position+1,i=0;w<=total_num;++w,++i)
            {
                int q=0;
                pipe_args[i]=(char*)malloc(sizeof(char)*20);
                for(;args[w][q]!='\0';++q)
                {
                    pipe_args[i][q]=args[w][q];
                }
                pipe_args[i][q]='\0';

                free(args[w-1]);
                args[w-1]=NULL;
                arg_num--;
            }

            free(args[total_num]);
            args[total_num]=NULL;
            arg_num--;
        }
```

5.2.2 Create Pipe and Execute

To execute the two command, we need to use **fork** to create a new child process.

```
                        // new  child
                        pid2 = fork();
```

The rest part is similar to what we do in the former part. We need to seperate the child process and the parent process and execute them individually.

For the child process:

1. Close the read end of the pipe
2. Redirect **STDOUT_FILENO** to **WRITE_END**

3. Execute
4. Close the write end
5. Exit

```
else if (pid2 == 0) {
                        //child do former
                        close(fd[READ_END]);
                        dup2(fd[WRITE_END], STDOUT_FILENO);
                        execvp(args[0], args);
                        close(fd[WRITE_END]);
                        exit(0);
                        }
```

For the parent process:

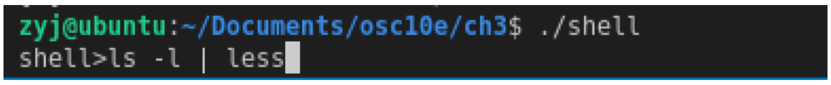
1. Close the write end of the pipe
2. Redirect **STDOUT_FILENO** to **READ_END**
3. Execute
4. Close the read end
5. Wait the child process

```
else {
                            //parent do latter
                        close(fd[WRITE_END]);
                        dup2(fd[READ_END], STDIN_FILENO);
                        execvp(pipe_args[0], pipe_args);
                        close(fd[READ_END]);
                        wait(NULL);
                        }
```

5.3 Program Results

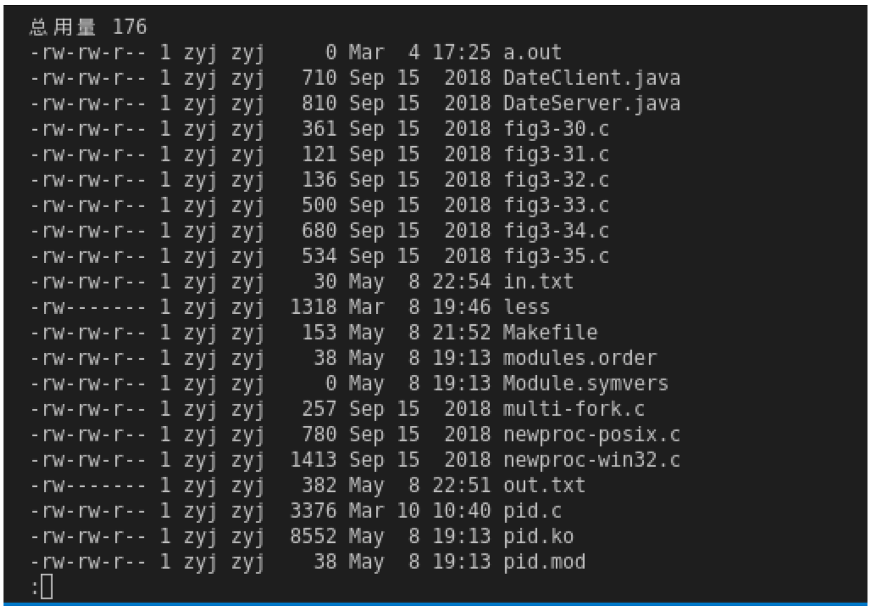
Enter:

```
ls -l | less
```



```
zyj@ubuntu:~/Documents/osc10e/ch3$ ./shell
shell>ls -l | less
```

The result of Shell:



```
总用量 176
-rw-rw-r-- 1 zyj zyj    0 Mar  4 17:25 a.out
-rw-rw-r-- 1 zyj zyj  710 Sep 15  2018 DateClient.java
-rw-rw-r-- 1 zyj zyj  810 Sep 15  2018 DateServer.java
-rw-rw-r-- 1 zyj zyj  361 Sep 15  2018 fig3-30.c
-rw-rw-r-- 1 zyj zyj  121 Sep 15  2018 fig3-31.c
-rw-rw-r-- 1 zyj zyj  136 Sep 15  2018 fig3-32.c
-rw-rw-r-- 1 zyj zyj  500 Sep 15  2018 fig3-33.c
-rw-rw-r-- 1 zyj zyj  680 Sep 15  2018 fig3-34.c
-rw-rw-r-- 1 zyj zyj  534 Sep 15  2018 fig3-35.c
-rw-rw-r-- 1 zyj zyj   30 May  8 22:54 in.txt
-rw------- 1 zyj zyj 1318 Mar  8 19:46 less
-rw-rw-r-- 1 zyj zyj  153 May  8 21:52 Makefile
-rw-rw-r-- 1 zyj zyj   38 May  8 19:13 modules.order
-rw-rw-r-- 1 zyj zyj    0 May  8 19:13 Module.symvers
-rw-rw-r-- 1 zyj zyj  257 Sep 15  2018 multi-fork.c
-rw-rw-r-- 1 zyj zyj  780 Sep 15  2018 newproc-posix.c
-rw-rw-r-- 1 zyj zyj 1413 Sep 15  2018 newproc-win32.c
-rw------- 1 zyj zyj  382 May  8 22:51 out.txt
-rw-rw-r-- 1 zyj zyj 3376 Mar 10 10:40 pid.c
-rw-rw-r-- 1 zyj zyj 8552 May  8 19:13 pid.ko
-rw-rw-r-- 1 zyj zyj   38 May  8 19:13 pid.mod
:
```

6 The Full Implementation(shell.c)

Simple use **gcc** to complie this program **shell.c**:

```
gcc shell.c -o shell
```

The full implementation is shown here for the reference.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>

#define MAX_LINE    80
#define READ_END    0
```

```
#define WRITE_END   1


int main()
{
    char *args[MAX_LINE/2 + 1];
    for(int k=0;k<MAX_LINE/2+1;k++)args[k]=NULL;

    char input_sentence[81];
    char history_sentence[81];
    char stop_word[]="exit";


    int should_run = 1;
    int history_exist = 0;
    int fd[2];

    while (should_run)
    {
       printf("shell>") ;

        //input
       fgets(input_sentence, 81, stdin);
       if(input_sentence[0]=='\n')
               continue;


 if(input_sentence[0]=='!'&&input_sentence[1]=='!'&&input_sentence[2]=='\n'
) {
            if(history_exist==0){
                printf("No commands in history!\n");
                continue;
                }
            else{
                int k=0;
                for(int w=0;history_sentence[w]!='\n';++w)
                {
                    printf("%c", history_sentence[w]);
                }
                printf("\n");
                for(;history_sentence[k]!='\n';k++)
                    input_sentence[k]=history_sentence[k];
                input_sentence[k]='\n';
                input_sentence[k+1]='\0';
            }
        }

        //parsing
        int arg_num=0;
        for(int i=0;input_sentence[i]!='\n';)
        {
            args[arg_num] = (char*)malloc(sizeof(char)*20);
            int j=0;
            for(; input_sentence[i]!='\n'&&input_sentence[i]!=' ';++j,++i){
                args[arg_num][j]=input_sentence[i];
            }
            args[arg_num][j]='\0';

            arg_num++;
            if(input_sentence[i]=='\n')
                break;
            if(input_sentence[i]==' ')
                i ++;
        }
        arg_num--;


        if(strcmp(args[0], stop_word)==0) break;

        if(strcmp(args[0], "!!")==0)
        {
            printf("No commands in history!!\n");
            continue;
        }
```

```
// divide pipe args
int pipe_exist=0;
int pipe_position;
for(int n=0;n<=arg_num;++n)
{
    if(strcmp(args[n],"|")==0)
        {
            pipe_exist = 1;
            pipe_position = n;
        }

}
char *pipe_args[MAX_LINE/2+1];
if(pipe_exist==1)
{

    // create pipe args
    for(int w=0;w<MAX_LINE/2+1;w++)pipe_args[w]=NULL;
    int total_num = arg_num;

    for(int w=pipe_position+1,i=0;w<=total_num;++w,++i)
    {
        int q=0;
        pipe_args[i]=(char*)malloc(sizeof(char)*20);
        for(;args[w][q]!='\0';++q)
        {
            pipe_args[i][q]=args[w][q];
        }
        pipe_args[i][q]='\0';

        free(args[w-1]);
        args[w-1]=NULL;
        arg_num--;
    }
```

```
            free(args[total_num]);
            args[total_num]=NULL;
            arg_num--;

        }

        //process
        pid_t pid;
        pid = fork();
        if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
                }
        else if (pid == 0) {

                if(strcmp(args[arg_num],"&")==0)
                        args[arg_num]=NULL, arg_num--;

                if(arg_num>=1)
                {
                    if(strcmp(args[arg_num-1],">")==0)
                    {
                        //situation >
                        int origin_point = dup(STDOUT_FILENO);
                        int file_write = open(args[arg_num],
O_CREAT|O_RDWR|O_TRUNC, S_IRUSR|S_IWUSR);
                        if (file_write==-1)
                            fprintf(stderr, "Create file Failed");
                        dup2(file_write, STDOUT_FILENO);
                        args[arg_num-1]=NULL;
                        args[arg_num]=NULL;
                        execvp(args[0], args);
                        close(file_write);
                        dup2(origin_point, STDOUT_FILENO);
                        exit(0);
                    }
                    else if(strcmp(args[arg_num-1],"<")==0)
                    {
                        //situation <
                        int origin_point = dup(STDIN_FILENO);
                        int file_read = open(args[arg_num], O_RDONLY);
                        dup2(file_read, STDIN_FILENO);
                        args[arg_num-1]=NULL;
                        args[arg_num]=NULL;
                        execvp(args[0], args);
                        close(file_read);
                        dup2(origin_point, STDIN_FILENO);
                        exit(0);
                    }
                    else if(pipe_exist==1)
                    {


                        //situation using pipe

                        pid_t pid2;
                        if (pipe(fd) == -1)
                        {
                            fprintf(stderr,"Pipe failed");
                            return 1;
                        }
                        // new  child
                        pid2 = fork();
                        if (pid2 < 0)
                        {
                            fprintf(stderr, "Fork Failed");
                            return 1;
                        }
                        else if (pid2 == 0) {
                        //child do former
                        close(fd[READ_END]);
                        dup2(fd[WRITE_END], STDOUT_FILENO);
                        execvp(args[0], args);
                        close(fd[WRITE_END]);
                        exit(0);
                        }
                        else {
                            //parent do latter
                        close(fd[WRITE_END]);
                        dup2(fd[READ_END], STDIN_FILENO);
                        execvp(pipe_args[0], pipe_args);
                        close(fd[READ_END]);
                        wait(NULL);
                        }
                    }
                    else
                    {
                        execvp(args[0], args);
                    }
                }
                else
                    execvp(args[0], args);
                }
        else { //parent process
                if(strcmp(args[arg_num],"&")!=0){
                    wait(NULL);
                }
            }

        //clear
        for(int i = 0;i < MAX_LINE/2 + 1; ++i) free(args[i]);

        //save history
        int m = 0;
        for(; input_sentence[m]!='\n';
++m)history_sentence[m]=input_sentence[m];
        history_sentence[m]='\n';
        history_sentence[m+1]='\0';
        history_exist = 1;

    }
```

```
}
```

Project 2-2 **Linux Kernel Module for Task Information**

0 Introduction

In this project, we write a Linux kernel module that uses the /proc file system for displaying a task's information based on its process identifier value pid.

We strating from writing a process identifier to the file /proc/pid.

Once a pid has been written to the /proc file, subsequent reads from /proc/pid will report (1) the command the task is running, (2) the value of the task's pid, and (3) the current state of the task.

An example:

```
echo "1395" > /proc/pid
cat /proc/pid
command = [bash] pid = [1395] state = [1]
```

1 **Writing to the** /proc **File System**

1.1 Requirements

We need to use several functions provided by the system, such as **kmalloc()** , **kstrtol()** , **kfree()** , to write to the /proc system.

1.2 Implementation Details and Methods

First, we need to add **.proc_write** in the struct **my_fops**. This will support us to write to the /proc file system.

```
static struct proc_ops my_fops={
    .proc_read = seq_read,
    .proc_write = seq_write
};
```

Second, we need to finish the function **seq_write**.

1. Allocate the kernel memory. The kmalloc() function is the kernel equivalent of the user-level malloc() function for allocating memory, except that kernel memory is being allocated. The **GFP_KERNEL** flag indicates routine kernel memory allocation.

   ```
           char *k_mem;
           char buffer[BUFFER_SIZE];

           // allocate kernel memory
           k_mem = kmalloc(count, GFP_KERNEL);
   ```

2. Copy the user buffer to the kernerl memory. If this fails, the function **copy_from_user** returns 1.

   ```
   /* 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;
           }
   ```

3. Transfer the string into a new temporary variable **buffer**, to enable the use of **kstrtol**.

   The function **kstrtol** is shown below, which gets a string into the long integer number.

   ```
   int kstrtol(const char *str, unsigned int base, long *res)
   ```

Note that we need to use **sscanf** to read the kernel memory into the **buffer**.

Finally, we transfer **buffer** into **l_pid**, a value that stores the pid of the process we input.

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

      sscanf(k_mem,"%s", buffer);
      kstrtol(buffer, 10, &l_pid);
      kfree(k_mem);
```

2 Reading from the /proc File System

2.1 Requirements

Once the process identifier has been stored, any reads from /proc/pid will return the name of the command, its process identifier, and its state.

2.2 Implementation Details and Methods

First, get the PCB of the process, which is represented as **task_struct** in Linux:

```
struct task struct pid task(struct pid *pid, enum pid type type)
```

We utilize the function **find_vpid** . *find_vpid*() finds the pid by its virtual id.

Then we can have the **task_struct**.

```
tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);
```

Second, read the contents of the **task_struct**, including **command, pid, state**:

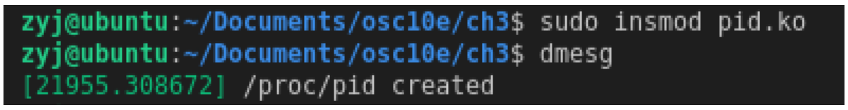
```
rv = sprintf(buffer,"command = [%s], pid = [%ld], state = [%ld]\n", tsk->comm, l_pid, tsk->state);
```

Third, we get the contents we read into the user buffer, to make us see in the command line.

```
if (copy_to_user(usr_buf, buffer, rv)) {
                rv = -1;
        }
```

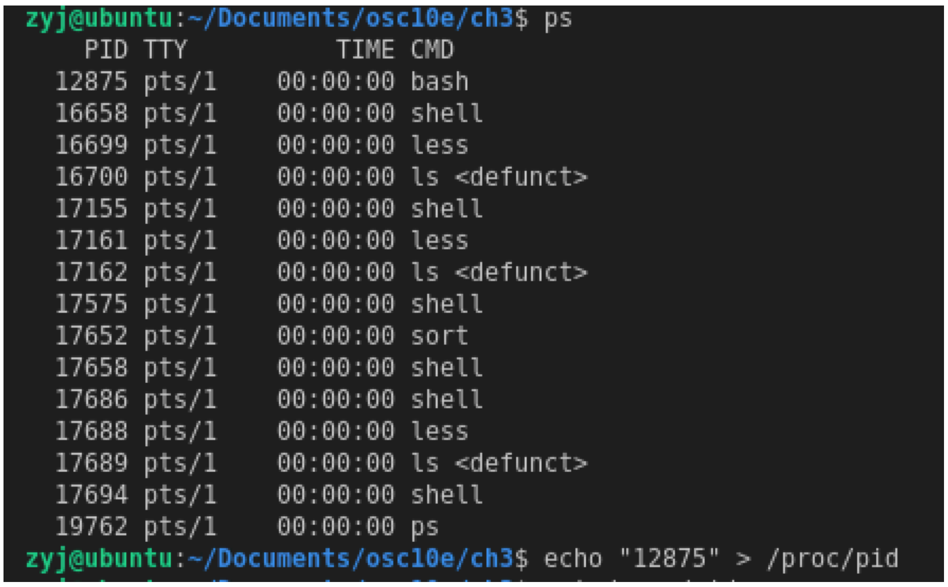
3 Program Results

We first insert the kernel module:



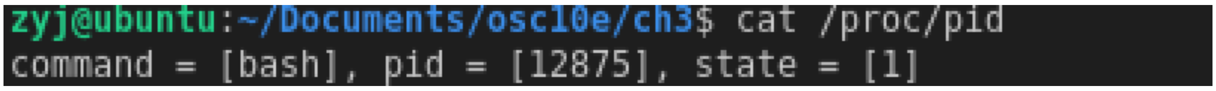
```
zyj@ubuntu:~/Documents/osc10e/ch3$ sudo insmod pid.ko
zyj@ubuntu:~/Documents/osc10e/ch3$ dmesg
[21955.308672] /proc/pid created
```

Select the **pid** of one process:



```
zyj@ubuntu:~/Documents/osc10e/ch3$ ps
    PID TTY          TIME CMD
  12875 pts/1    00:00:00 bash
  16658 pts/1    00:00:00 shell
  16699 pts/1    00:00:00 less
  16700 pts/1    00:00:00 ls <defunct>
  17155 pts/1    00:00:00 shell
  17161 pts/1    00:00:00 less
  17162 pts/1    00:00:00 ls <defunct>
  17575 pts/1    00:00:00 shell
  17652 pts/1    00:00:00 sort
  17658 pts/1    00:00:00 shell
  17686 pts/1    00:00:00 shell
  17688 pts/1    00:00:00 less
  17689 pts/1    00:00:00 ls <defunct>
  17694 pts/1    00:00:00 shell
  19762 pts/1    00:00:00 ps
zyj@ubuntu:~/Documents/osc10e/ch3$ echo "12875" > /proc/pid
```

Show the information of this process:



```
zyj@ubuntu:~/Documents/osc10e/ch3$ cat /proc/pid
command = [bash], pid = [12875], state = [1]
```

4 The Full Implementation (pid.c)

The full implementation is shown here for the inference.

To run the program, first **make** and then insert the new kernel module **pid.ko**.

Makefile:

```
obj-m += pid.o
all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

pid.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 seq_read(struct file *file, char *buf, size_t count, loff_t
*pos);
static ssize_t seq_write(struct file *file, const char __user *usr_buf,
size_t count, loff_t *pos);
/**
static struct file_operations proc_ops = {
        .owner = THIS_MODULE,
        .read = proc_read,
        .write = proc_write
};
**/
static struct proc_ops my_fops={
    .proc_read = seq_read,
    .proc_write = seq_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, &my_fops);

        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 seq_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)
                return -1;
        rv = sprintf(buffer,"command = [%s], pid = [%ld], state = [%ld]\n",
tsk->comm, l_pid, tsk->state);


        completed = 1;

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

        return rv;
}

/**
 * This function is called each time we write to the /proc file system.
 */
static ssize_t seq_write(struct file *file, const char __user *usr_buf,
size_t count, loff_t *pos)
{
        char *k_mem;
        char buffer[BUFFER_SIZE];

        // 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,"%s", buffer);
        kstrtol(buffer, 10, &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("Module");
MODULE_AUTHOR("YanjieZe");
```

Conclusion and Thoughts

The first project **UNIX Shell** enables us to know deeper into the **Shell**, and is an very interesting project to make us implement four functions step by step. What's more, We can practice the use of **pipe** and **redirection opreation**.

The second project **Linux Kernel Module for Task Information** goes deeper into the Linux /proc system, and get us to make use of the knowledge about **Process Control Block**.

Both projects are great fun. By finishing the two projects, I train the programming skills of myself and review the knowledge we study in the class, having a deeper knowledge of these concepts.