Project 2: UNIX Shell Programming and Linux Kernel Module for Task Information

CS307-Operating System (D), CS356-Operating System Course Design, Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang) Student ID: 518030910150 Email: galaxies@sjtu.edu.cn

1 UNIX Shell Programming

According to the project descriptions in textbook, we are required to write a UNIX Shell which provides the following functions.

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

Solution. Here is my method to implement the simple UNIX Shell program (`shell.c`).

- First, we implement a `standardlize_inst` function to standardlize the instruction in a standard form, which means there is no extra space and tab in the instruction.
- After standardlization, we can check whether this instruction asks for concurrent execution. If true, then we set the `concurrent` flag to 1, and the parent process do not have to execute `wait(NULL)` instruction to wait for its child process to finish execution since we can execute two processes concurrently.
- Then we can handle some special instructions such as `exit` and `!!`.
  - For `exit` instruction, we simply set the `should_run` flag to false, then it will jump out of the loop and finish executing the main function;
  - For `!!` instruction, we store the last instruction in the variable `last_inst` so we only need to replace the current instruction `inst` with the last instruction `last_inst`. To handle the error condition, we need to set a `have_last_inst` flag to check whether the current instruction do not have the previous instruction in the history.
- After that, we create a child process using `fork()` function, and implement a `parse` function in child process to parse the instruction and return several arguments `args[]` of the instruction. In parent process, we actually do nothing except waiting (if necessary).
- After parsing instruction, we can check whether the instruction needs a pipe. If true, then we can use `pipe(pipe_fd)` function to generate a pipe and use it to implement data transfer. The implementation of pipe transfer is similar to Section 3.7.4 in textbook.
- If the instruction does not need a pipe, then we can find out the input/output redirections of the instruction by examining the result of parsing. If the instruction needs redirections, then we can redirect the input/output using `dup2(fd, ...)` function.
- Then, we can execute the instruction using the `execvp(...)` function and the arguments `args[]` we have parsed before.

- After execution, we need to release the spaces of variables, close the files, and then we can exit from child process.

Here is the specific implementation of the UNIX Shell.

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

# define MAX_LINE 80
# define READ_END 0
# define WRITE_END 1

// Clear the string.
void clear_str(char *str);

// Check whether the instruction is concurrent.
int check_concurrent(char *inst);

// Standardlize the instruction.
void standardlize_inst(char *inst);

// Parse the instruction
int parse(char *inst, char **args);

// Debug program for parsing.
void debug_parse(char *args[], int argn);

int main(void) {
  // arguments, instruction, last instruction
  char *args[MAX_LINE / 2 + 1], *inst, *last_inst;
  // whether have the last instruction, cocurrent status
  int have_last_inst = 0, concurrent = 0;
  // input redirect filename, output redirect filename
  char *in_file, *out_file;

  inst = (char*) malloc(MAX_LINE * sizeof(char));
  last_inst = (char*) malloc(MAX_LINE * sizeof(char));
  in_file = (char*) malloc(MAX_LINE * sizeof(char));
  out_file = (char*) malloc(MAX_LINE * sizeof(char));

  clear_str(last_inst);

  int should_run = 1;

  pid_t pid;

```

```
  while(should_run) {
    printf("osh> ");
    fflush(stdout);
    if (concurrent) wait(NULL);

    concurrent = 0;
    clear_str(inst);

    fgets(inst, MAX_LINE, stdin);

    standardlize_inst(inst);
    concurrent = check_concurrent(inst);

    // exit shell
    if (strcmp(inst, "exit") == 0) {
      should_run = 0;
      continue;
    }

    // execute the last instruction
    if (strcmp(inst, "!!") == 0) {
      if (have_last_inst == 0) {
        fprintf(stderr, "Error: No commands in history.\n");
        continue;
      } else {
        printf("%s\n", last_inst);
        strcpy(inst, last_inst);
      }
    }

    pid = fork();
    if (pid < 0) fprintf(stderr, "Error: Fork failed!\n");
    else {
      if (pid == 0) {
        // child process
        // whether an error has occured
        int error_occur = 0;

        // allocate space for commands & arguments
        for (int i = 0; i <= MAX_LINE / 2; ++ i)
        args[i] = (char*) malloc(MAX_LINE * sizeof(char));

        // parse the instruction
        int argn = parse(inst, args);

        // free the space of extra commands & extra arguments
        for (int i = argn; i <= MAX_LINE / 2; ++ i) {
          free(args[i]);
          args[i] = NULL;
        }
```

```
    if (concurrent == 1) {
      -- argn;
      free(args[argn]);
      args[argn] = NULL;
    }

    // find pipe
    int pipe_pos = -1;
    for (int i = 0; i < argn; ++ i)
      if (strcmp(args[i], "|") == 0) {
        pipe_pos = i;
        break;
      }

    if(pipe_pos >= 0) {
      // pipe found
      if (pipe_pos == 0 || pipe_pos == argn - 1) {
        fprintf(stderr, "Error: Unexpected syntax '|'.\n");
        error_occur = 1;
      }

      // pipe fd
      int pipe_fd[2];

      if (pipe(pipe_fd) == -1) {
        fprintf(stderr, "Error: Pipe Failed!\n");
        error_occur = 1;
      }

      if(error_occur == 0) {
        // fork a grandson process
        pid = fork();
        if (pid < 0) {
          fprintf(stderr, "Error: Fork failed!\n");
          error_occur = 1;
        } else {
          if (pid == 0) {
            // grandchild process
            for (int i = pipe_pos; i < argn; ++ i) {
              free(args[i]);
              args[i] = NULL;
            }
            argn = pipe_pos;

            close(pipe_fd[READ_END]);
            if (error_occur == 0 && dup2(pipe_fd[WRITE_END], STDOUT_FILENO) < 0) {
              fprintf(stderr, "Error: dup2 failed!\n");
              error_occur = 1;
            }
```

```

            if(error_occur == 0 && argn > 0) execvp(args[0], args);

            close(pipe_fd[WRITE_END]);

            // free the spaces
            for (int i = 0; i < argn; ++ i) free(args[i]);
            free(inst);
            free(last_inst);
            free(in_file);
            free(out_file);

            exit(error_occur);
          } else {
            // child process
            wait(NULL);
            for (int i = 0; i <= pipe_pos; ++ i) free(args[i]);
            for (int i = pipe_pos + 1; i < argn; ++ i) args[i - pipe_pos - 1] = args[i];
            for (int i = argn - pipe_pos - 1; i < argn; ++ i) args[i] = NULL;
            argn = argn - pipe_pos - 1;

            close(pipe_fd[WRITE_END]);
            if (error_occur == 0 && dup2(pipe_fd[READ_END], STDIN_FILENO) < 0) {
              fprintf(stderr, "Error: dup2 failed!\n");
              error_occur = 1;
            }

            if(error_occur == 0 && argn > 0) execvp(args[0], args);

            close(pipe_fd[READ_END]);
          }
        }
      }
    } else {
      // find in_redirect or out_redirect
      int in_redirect = 0, out_redirect = 0, in_fd = -1, out_fd = -1;
      while (argn >= 2 && (strcmp(args[argn - 2], "<") == 0 || strcmp(args[argn - 2], ">") == 0)) {
        argn -= 2;
        if (strcmp(args[argn], "<") == 0) {
          in_redirect = 1;
          strcpy(in_file, args[argn + 1]);
        } else {
          out_redirect = 1;
          strcpy(out_file, args[argn + 1]);
        }
        free(args[argn]); args[argn] = NULL;
        free(args[argn + 1]); args[argn + 1] = NULL;
```

```
        }

        // redirect input
        if (error_occur == 0 && in_redirect == 1) {
          in_fd = open(in_file, O_RDONLY, 0644);
          if (error_occur == 0 && in_fd < 0) {
            fprintf(stderr, "Error: No such files.\n");
            error_occur = 1;
          }
          if (error_occur == 0 && dup2(in_fd, STDIN_FILENO) < 0) {
            fprintf(stderr, "Error: dup2 failed!\n");
            error_occur = 1;
          }
        }

        // redirect output
        if (error_occur == 0 && out_redirect == 1) {
          out_fd = open(out_file, O_WRONLY | O_TRUNC | O_CREAT, 0644);
          if (error_occur == 0 && out_fd < 0) {
            fprintf(stderr, "Error: No such files.\n");
            error_occur = 1;
          }
          if (error_occur == 0 && dup2(out_fd, STDOUT_FILENO) < 0) {
            fprintf(stderr, "Error: dup2 failed!\n");
            error_occur = 1;
          }
        }

        // not an empty instruction & no error occur, then execute the instruction
        if (error_occur == 0 && argn != 0)
          execvp(args[0], args);

        // close the files
        if (in_redirect == 1 && in_fd > 0) close(in_fd);
        if (out_redirect == 1 && out_fd > 0) close(out_fd);
      }

      // free the spaces
      for (int i = 0; i < argn; ++ i) free(args[i]);
      free(inst);
      free(last_inst);
      free(in_file);
      free(out_file);

      // child process exit
      exit(error_occur);
    } else {
      // parent process
      if(concurrent == 0) wait(NULL);
    }
```

```
    }

    if(have_last_inst == 0) have_last_inst = 1;
    strcpy(last_inst, inst);
  }

  // free the spaces
  free(inst);
  free(last_inst);
  free(in_file);
  free(out_file);
  return 0;
}


// Clear the string.
void clear_str(char *str) {
  memset(str, 0, sizeof(str));
}

// Check whether the instruction is concurrent.
int check_concurrent(char *inst) {
  int len = strlen(inst);
  if(len && inst[len - 1] == '&') return 1;
  return 0;
}

// Standardlize the instruction.
// Specific function: clear the extra space & tab & enter in the instruction.
void standardlize_inst(char *inst) {
  int len = strlen(inst);

  char *temp = (char*) malloc(len * sizeof(char));
  for (int i = 0; i < len; ++ i) temp[i] = inst[i];
  clear_str(inst);

  int new_len = 0, last_blank = 1;
  for (int i = 0; i < len; ++ i) {
    if (temp[i] == ' ' || temp[i] == '\n' || temp[i] == '\t') {
      if (last_blank == 0) {
        inst[new_len ++] = ' ';
        last_blank = 1;
      }
    } else {
      inst[new_len ++] = temp[i];
      last_blank = 0;
    }
  }
  if(inst[new_len - 1] == ' ') inst[new_len - 1] = 0;

```

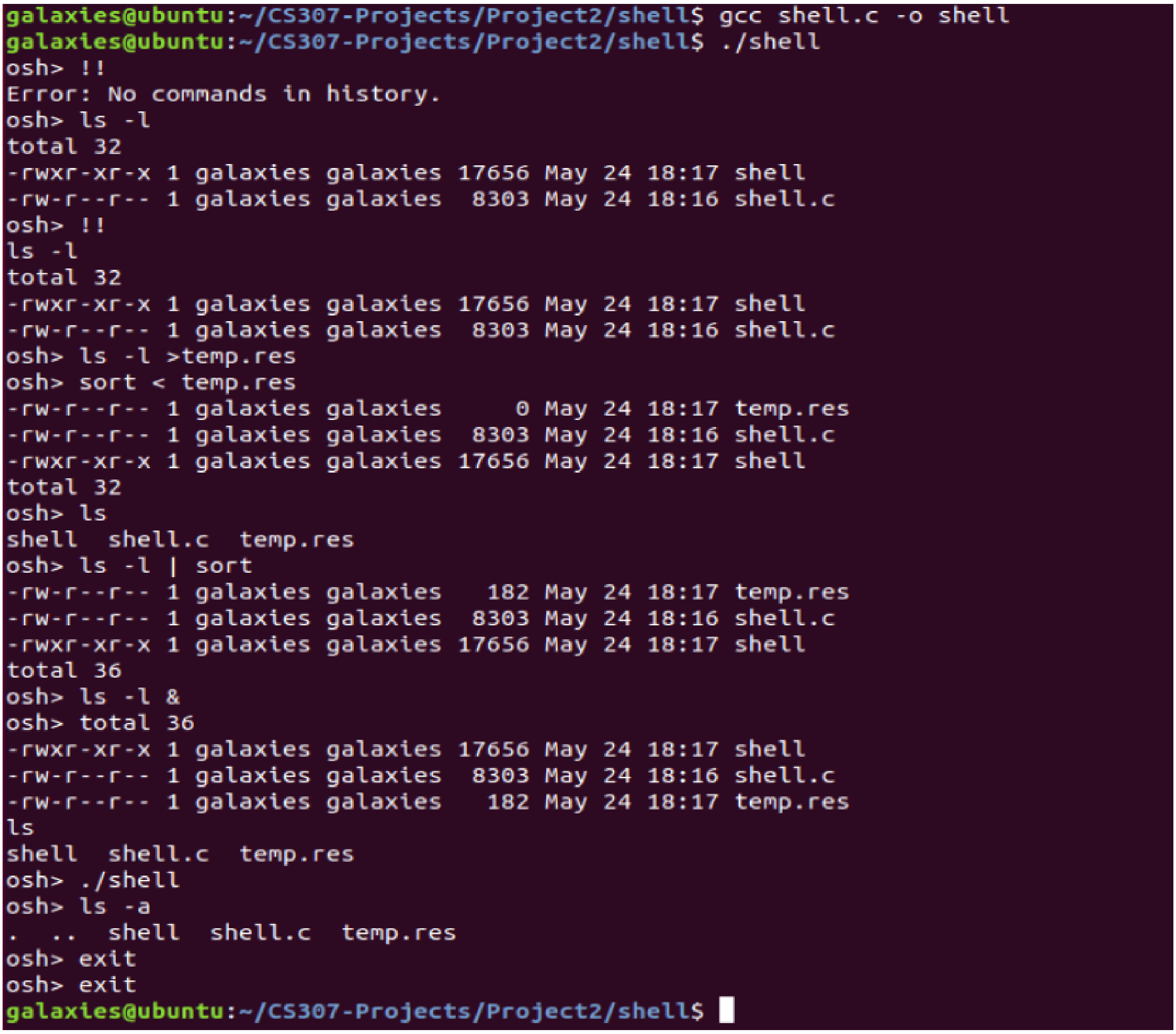
```
  free(temp);
}

// Parse the instruction.
// Specific function: parse the instruction and find out the command & arguments.
int parse(char *inst, char **args) {
  int len = strlen(inst);

  // find out the arguments
  int argn = 0;
  for (int i = 0; i < len; ++ i) {
    clear_str(args[argn]);

    int j = i;
    while(j < len && inst[j] != ' ') {
      args[argn][j - i] = inst[j];
      ++ j;
    }
    if ((args[argn][0] == '<' || args[argn][0] == '>' || args[argn][0] == '|') &&
        j > i+1) {
      strcpy(args[argn + 1], args[argn] + 1);
      for (int k = 1; k < j - i; ++ k) args[argn][k] = 0;
      ++ argn;
    }

    i = j;
    ++ argn;
  }

  return argn;
}

// Debug program for parsing.
void debug_parse(char *args[], int argn) {
  fprintf(stderr, "Comm: %s, total %d arguments\n", args[0], argn);
  for (int i = 0; i < argn; ++ i)
    fprintf(stderr, "args[%d] = %s\n", i, args[i]);
```

We can test the UNIX Shell program by entering the following instructions. These instructions can test every required function of the UNIX Shell.

```
gcc shell.c -o ./shell
./shell
!!
ls -l
!!
ls -l > temp.res
sort < temp.res
ls
ls -l
ls -l | sort
```

```
ls -l &
ls
./shell
ls -a
exit
exit
```

Here is the execution result of my UNIX Shell program after entering the instructions (Fig. 1).



```
galaxies@ubuntu:~/CS307-Projects/Project2/shell$ gcc shell.c -o shell
galaxies@ubuntu:~/CS307-Projects/Project2/shell$ ./shell
osh> !!
Error: No commands in history.
osh> ls -l
total 32
-rwxr-xr-x 1 galaxies galaxies 17656 May 24 18:17 shell
-rw-r--r-- 1 galaxies galaxies  8303 May 24 18:16 shell.c
osh> !!
ls -l
total 32
-rwxr-xr-x 1 galaxies galaxies 17656 May 24 18:17 shell
-rw-r--r-- 1 galaxies galaxies  8303 May 24 18:16 shell.c
osh> ls -l >temp.res
osh> sort < temp.res
-rw-r--r-- 1 galaxies galaxies     0 May 24 18:17 temp.res
-rw-r--r-- 1 galaxies galaxies  8303 May 24 18:16 shell.c
-rwxr-xr-x 1 galaxies galaxies 17656 May 24 18:17 shell
total 32
osh> ls
shell  shell.c  temp.res
osh> ls -l | sort
-rw-r--r-- 1 galaxies galaxies   182 May 24 18:17 temp.res
-rw-r--r-- 1 galaxies galaxies  8303 May 24 18:16 shell.c
-rwxr-xr-x 1 galaxies galaxies 17656 May 24 18:17 shell
total 36
osh> ls -l &
osh> total 36
-rwxr-xr-x 1 galaxies galaxies 17656 May 24 18:17 shell
-rw-r--r-- 1 galaxies galaxies  8303 May 24 18:16 shell.c
-rw-r--r-- 1 galaxies galaxies   182 May 24 18:17 temp.res
ls
shell  shell.c  temp.res
osh> ./shell
osh> ls -a
.  ..  shell  shell.c  temp.res
osh> exit
osh> exit
galaxies@ubuntu:~/CS307-Projects/Project2/shell$
```

Figure 1: The execution result of my UNIX Shell program

□

2 Linux Kernel Module for Task Information

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

Solution. Here is my method to implement the required Linux kernel module.

- In `proc_write` function, we can use the `kstrtol()` function to read the value from a string. An important fact is that `usr_buf` may do not have an end-of-string sign '\0', therefore we need to add this sign to the end of the string manually.
- In `proc_write` function, we need to use `kmalloc()` and `kfree()` function to allocate and release memory; they are actually the kernel version of `malloc()` and `free()`.
- In `proc_read` function, we can use the `pid_task()` function to read the information in PCB, and we need to use `find_vpid()` to find the corresponding PCB using its `pid`. What we need are the `comm` and `state` value of the PCB.
- We also perform an error checking for invalid `pid`. This will cause the `pid_task()` function returns `NULL`. We simply report the error and return 0.
- The other parts of the program is very similar to project 1, we can use the similar methods to implement them.

Here is the specific implementation of the Linux kernel module for task information (`pid.c`).

```c
# 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 <linux/uaccess.h>
# include <asm/uaccess.h>

# define BUFFER_SIZE 128
# define PROC_NAME "pid"

// the current pid
static long cur_pid;

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 file_operations proc_ops = {
  .owner = THIS_MODULE,
  .read = proc_read,
  .write = proc_write,
};

```

```c
static int proc_init(void) {
  /* create /proc files */
  proc_create(PROC_NAME, 0666, NULL, &proc_ops);
  printk(KERN_INFO "/proc/" PROC_NAME " is created!\n");
  return 0;
}

static void proc_exit(void) {
  /* remove /proc files */
  remove_proc_entry(PROC_NAME, NULL);
  printk(KERN_INFO "/proc/" PROC_NAME " is removed!\n");
}

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 *PCB = NULL;

  if (completed) {
    completed = 0;
    return 0;
  }

  PCB = pid_task(find_vpid(cur_pid), PIDTYPE_PID);
  if (PCB == NULL) {
    printk(KERN_INFO "Invalid PID!\n");
    return 0;
  }

  completed = 1;

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

  copy_to_user(usr_buf, buffer, rv);

  return rv;
}

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;
  }

  // the end of string k_mem
  k_mem[count] = 0;

  // extract the number into the variable pid using kstrtol
  kstrtol(k_mem, 10, &cur_pid);

  // free the memory
  kfree(k_mem);

  return count;
}

module_init(proc_init);
module_exit(proc_exit);

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

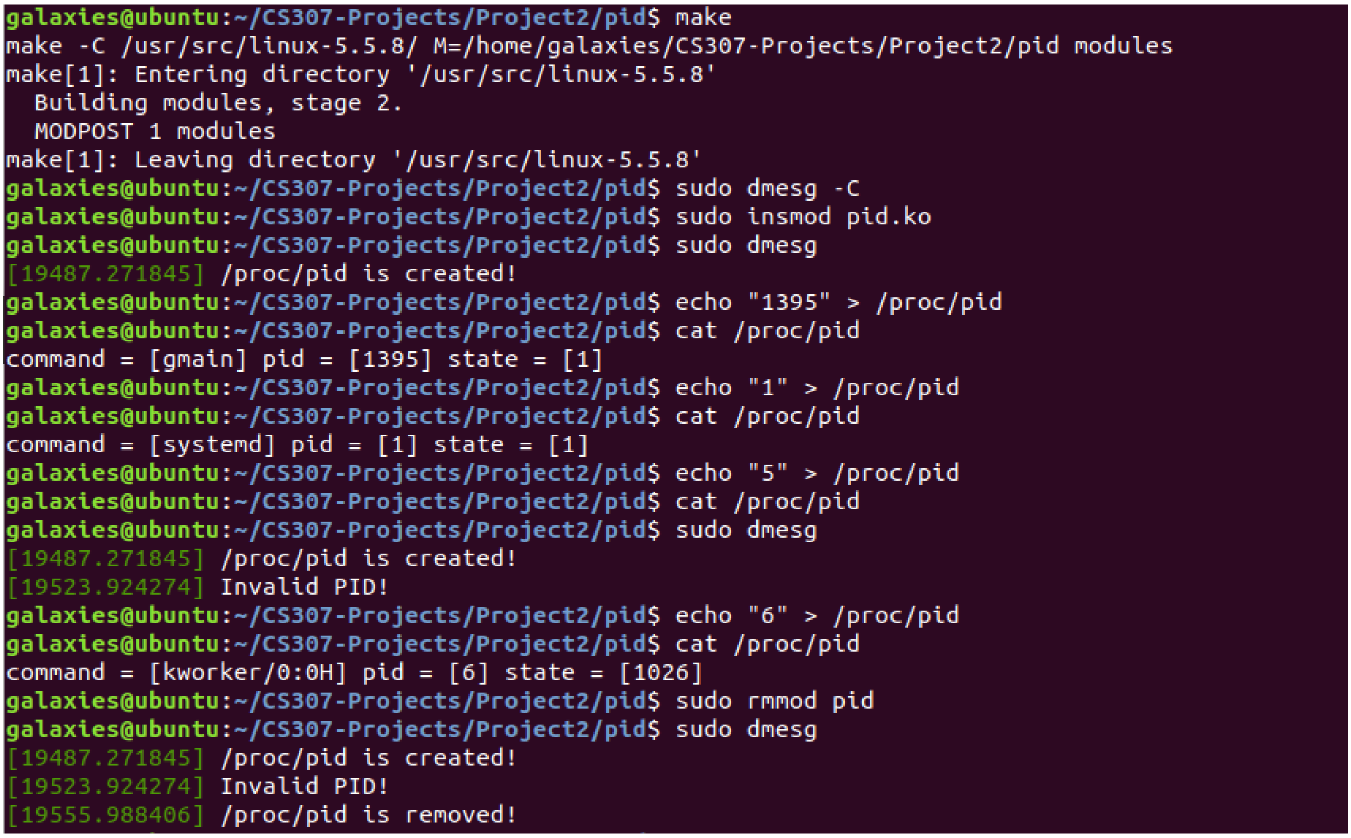
And here is the `Makefile` file of the project.

```
obj-m := pid.o

all:
  make -C /usr/src/linux-5.5.8/ M=$(shell pwd) modules
clean:
  make -C /usr/src/linux-5.5.8/ M=$(shell pwd) clean
```

We can enter the following instructions to test the kernel module.

```
make
sudo dmesg -C
sudo insmod pid.ko
sudo dmesg
echo "1395" > /proc/pid
cat /proc/pid
echo "1" > /proc/pid
cat /proc/pid
echo "5" > /proc/pid
cat /proc/pid
echo "6" > /proc/pid
cat /proc/pid
sudo rmmod pid
sudo dmesg
```

Here is the execution result of the program (Fig. 2).



```
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ make
make -C /usr/src/linux-5.5.8/ M=/home/galaxies/CS307-Projects/Project2/pid modules
make[1]: Entering directory '/usr/src/linux-5.5.8'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-5.5.8'
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ sudo dmesg -C
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ sudo insmod pid.ko
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ sudo dmesg
[19487.271845] /proc/pid is created!
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ echo "1395" > /proc/pid
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ cat /proc/pid
command = [gmain] pid = [1395] state = [1]
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ echo "1" > /proc/pid
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ cat /proc/pid
command = [systemd] pid = [1] state = [1]
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ echo "5" > /proc/pid
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ cat /proc/pid
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ sudo dmesg
[19487.271845] /proc/pid is created!
[19523.924274] Invalid PID!
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ echo "6" > /proc/pid
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ cat /proc/pid
command = [kworker/0:0H] pid = [6] state = [1026]
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ sudo rmmod pid
galaxies@ubuntu:~/CS307-Projects/Project2/pid$ sudo dmesg
[19487.271845] /proc/pid is created!
[19523.924274] Invalid PID!
[19555.988406] /proc/pid is removed!
```

Figure 2: The execution result of my Linux Kernel module for task information

□

3 Personal Thoughts

During the first UNIX Shell project, I've experienced the process of implementing a simple UNIX Shell program. The implementation enhances my understandings of the pipe and input/output redirections. Actually the implementations of the UNIX Shell program is complicated than I thought before, which takes me about 4 hours to finish, and the amount of code reaches 8 KB. After finishing the project, I feel very fulfilled and I become more familiar with the Linux C instructions. I'm also getting well with the C language programming in Linux.

The Linux kernel module for task information project enhances my understandings of `/proc` file system, and it strengthen my knowledge about it, which is important in Linux system.

By the way, you can find all the source codes in the "src" folder. You can also refer to my github to see my codes of this project, and they are in the `Project2` folder.