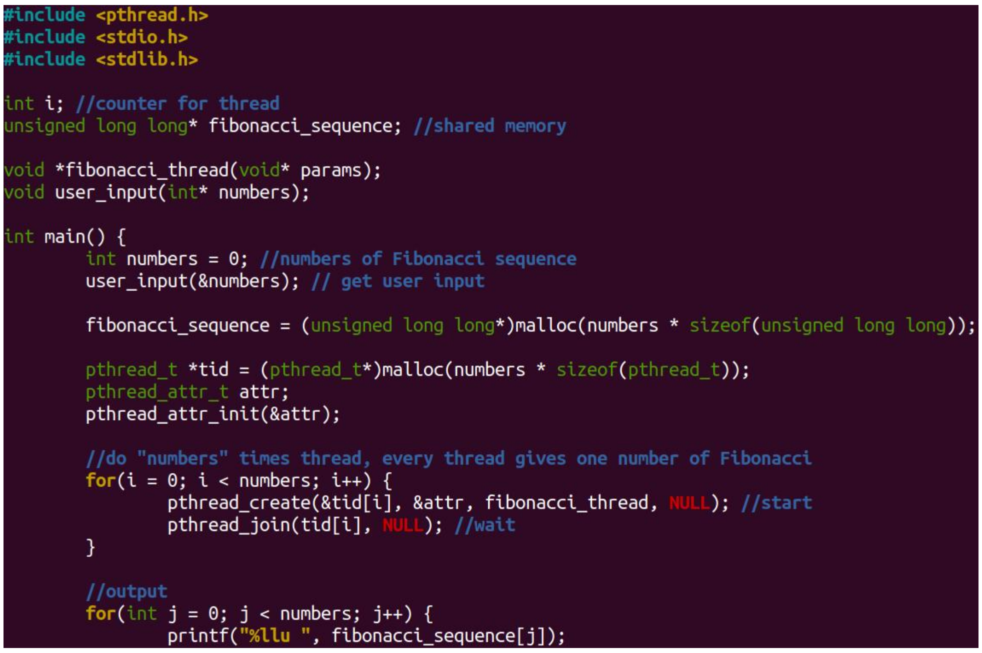
Report of Programming Projects

Description:

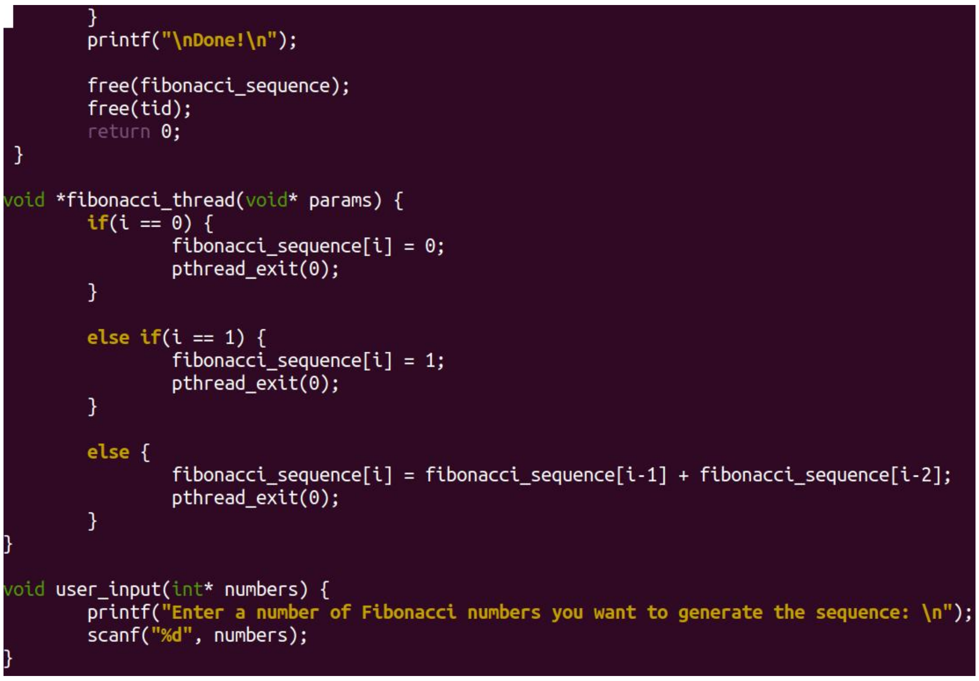
There are 3 programming projects: EX4.27, Unix Shell, and Kernel Module of Listing Tasks
And I will show you the codes, results, and explanation of each project in 3 parts here.
Part 1: p1 - p2, Part2: p3 - p8, Part3: p9 - p11.

Part 1: EX4.27 - Generate the Fibonacci Sequence.

Codes:



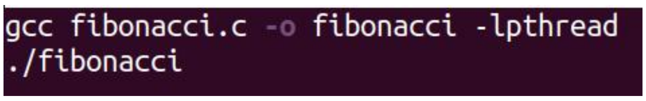
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int i; //counter for thread
unsigned long long* fibonacci_sequence; //shared memory

void *fibonacci_thread(void* params);
void user_input(int* numbers);

int main() {
        int numbers = 0; //numbers of Fibonacci sequence
        user_input(&numbers); // get user input

        fibonacci_sequence = (unsigned long long*)malloc(numbers * sizeof(unsigned long long));

        pthread_t *tid = (pthread_t*)malloc(numbers * sizeof(pthread_t));
        pthread_attr_t attr;
        pthread_attr_init(&attr);

        //do "numbers" times thread, every thread gives one number of Fibonacci
        for(i = 0; i < numbers; i++) {
                pthread_create(&tid[i], &attr, fibonacci_thread, NULL); //start
                pthread_join(tid[i], NULL); //wait
        }

        //output
        for(int j = 0; j < numbers; j++) {
                printf("%llu ", fibonacci_sequence[j]);
```



```
        }
        printf("\nDone!\n");

        free(fibonacci_sequence);
        free(tid);
        return 0;
 }

void *fibonacci_thread(void* params) {
        if(i == 0) {
                fibonacci_sequence[i] = 0;
                pthread_exit(0);
        }

        else if(i == 1) {
                fibonacci_sequence[i] = 1;
                pthread_exit(0);
        }

        else {
                fibonacci_sequence[i] = fibonacci_sequence[i-1] + fibonacci_sequence[i-2];
                pthread_exit(0);
        }
}

void user_input(int* numbers) {
        printf("Enter a number of Fibonacci numbers you want to generate the sequence: \n");
        scanf("%d", numbers);
}
```

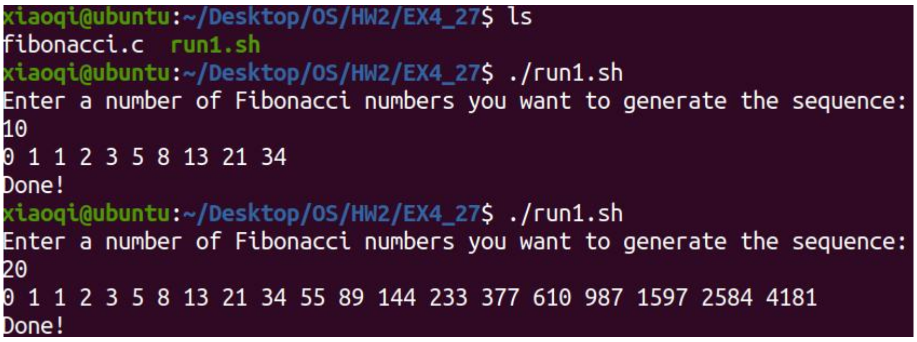
<1> I used the "unsigned long long" to define the Fibonacci sequences, because the number in the sequence increases so fast, if you use "int", it will reach the limit just in several iterations.
<2> I did "i" times "pthread_create" and "pthread_join", and I put these 2 iterations for i-th time together in a loop, otherwise, the sequence couldn't get update in time, and the data will be covered.
<3> In function "fibonacci_thread", I assign the sequence to be 0 when i is 0, and assign the sequence to be 1 when i is 1, thus the first 3 fibonacci numbers are assigned. And I did "pthread_exit" once they finish the assignment.
<4> In function "user_input", I scanf the input that you give, which is the number of Fibonacci numbers you want to generate the sequence.

Compile (run1.sh):



```
gcc fibonacci.c -o fibonacci -lpthread
./fibonacci
```

Results:



```
xiaoqi@ubuntu:~/Desktop/OS/HW2/EX4_27$ ls
fibonacci.c  run1.sh
xiaoqi@ubuntu:~/Desktop/OS/HW2/EX4_27$ ./run1.sh
Enter a number of Fibonacci numbers you want to generate the sequence:
10
0 1 1 2 3 5 8 13 21 34
Done!
xiaoqi@ubuntu:~/Desktop/OS/HW2/EX4_27$ ./run1.sh
Enter a number of Fibonacci numbers you want to generate the sequence:
20
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
Done!
```

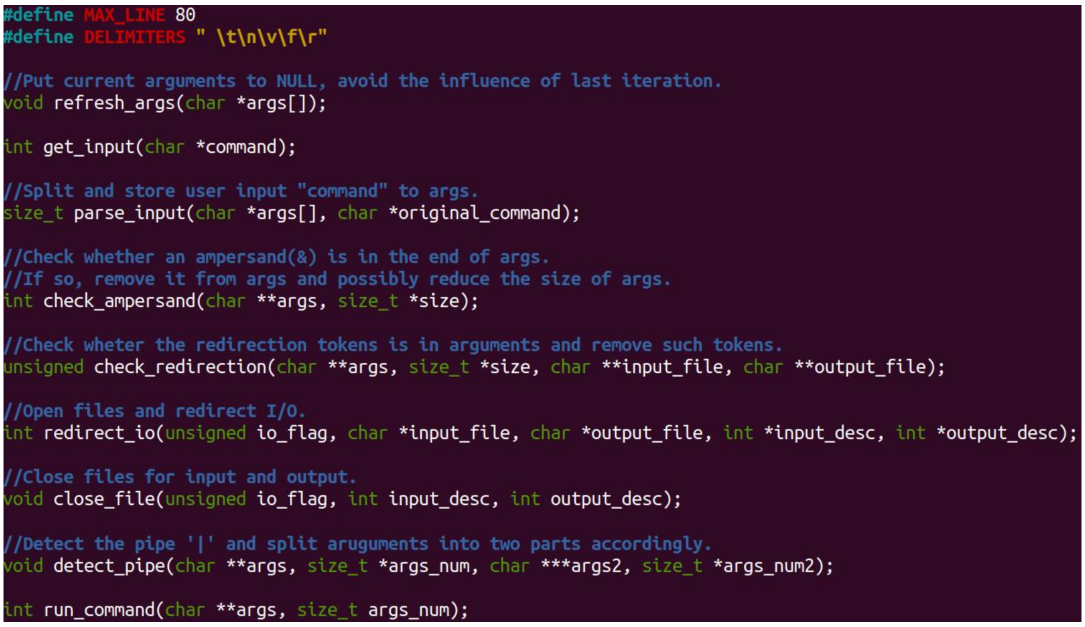
As you can see, after I execute the "run1.sh", the program will ask me for the number. After I entered 10 and 20 separately, the Fibonacci sequence is generated.

Part 2: Project - Unix Shell

Description:

This project is a quite big project, so it has many codes (200 lines). So I will show all the functions I used, explain what I did and results. And put the codes at the end.

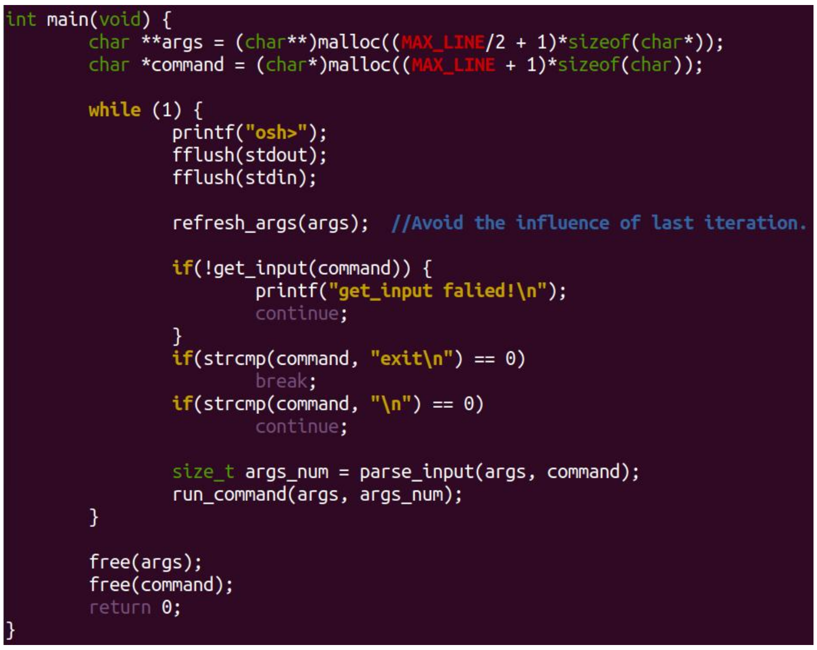
Functions:



```c
#define MAX_LINE 80
#define DELIMITERS " \t\n\v\f\r"

//Put current arguments to NULL, avoid the influence of last iteration.
void refresh_args(char *args[]);

int get_input(char *command);

//Split and store user input "command" to args.
size_t parse_input(char *args[], char *original_command);

//Check whether an ampersand(&) is in the end of args.
//If so, remove it from args and possibly reduce the size of args.
int check_ampersand(char **args, size_t *size);

//Check wheter the redirection tokens is in arguments and remove such tokens.
unsigned check_redirection(char **args, size_t *size, char **input_file, char **output_file);

//Open files and redirect I/O.
int redirect_io(unsigned io_flag, char *input_file, char *output_file, int *input_desc, int *output_desc);

//Close files for input and output.
void close_file(unsigned io_flag, int input_desc, int output_desc);

//Detect the pipe '|' and split aruguments into two parts accordingly.
void detect_pipe(char **args, size_t *args_num, char ***args2, size_t *args_num2);

int run_command(char **args, size_t args_num);
```

As you can see, I put the “blue” explanation on each of the functions. For example: the function “refresh_args” is used to put current arguments to NULL, in order to avoid the influence of last iteration. The functions “parse_input” is used to split and store user input “command” to “args”, etc...

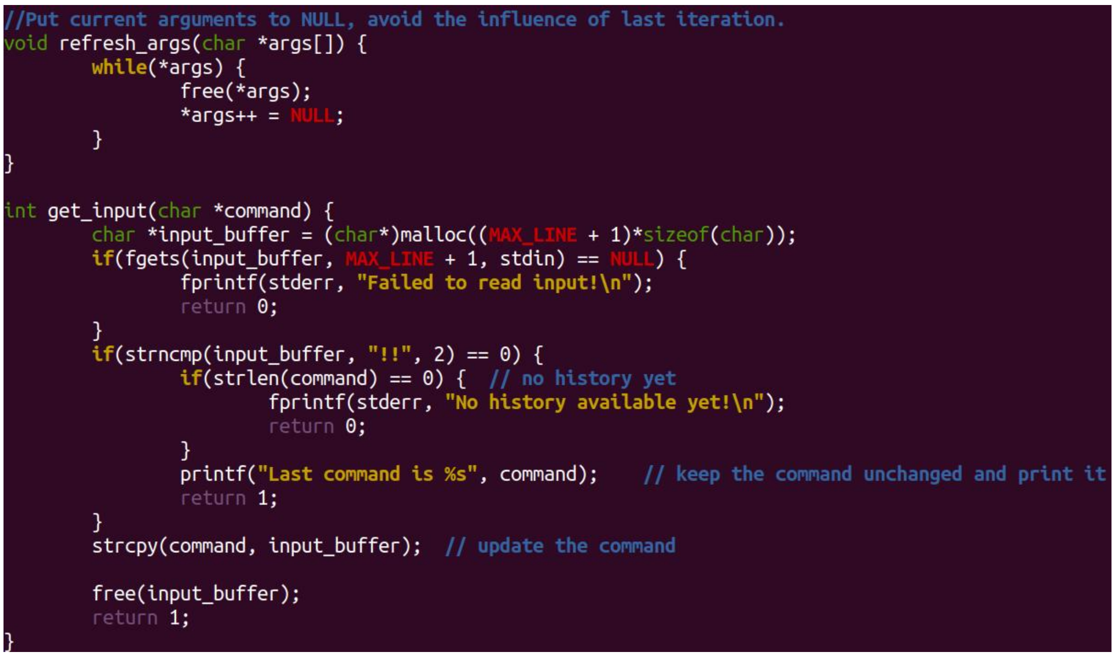
Main:



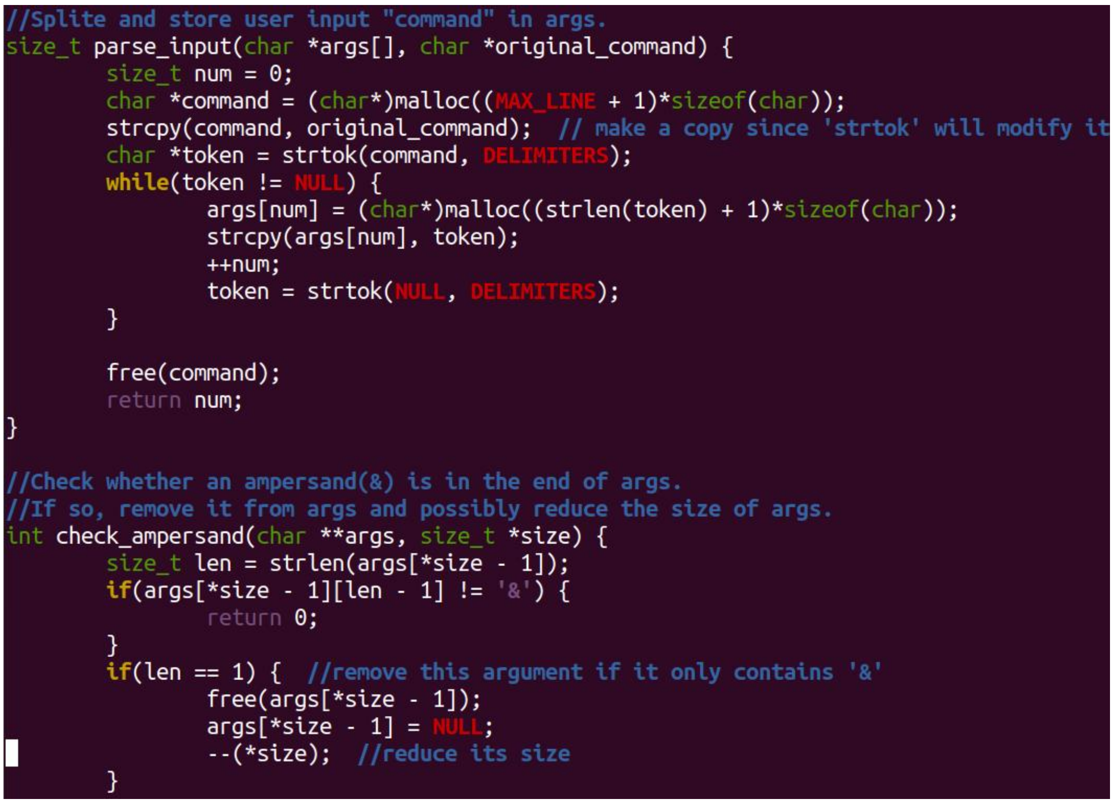
```c
int main(void) {
        char **args = (char**)malloc((MAX_LINE/2 + 1)*sizeof(char*));
        char *command = (char*)malloc((MAX_LINE + 1)*sizeof(char));

        while (1) {
                printf("osh>");
                fflush(stdout);
                fflush(stdin);

                refresh_args(args);  //Avoid the influence of last iteration.

                if(!get_input(command)) {
                        printf("get_input falied!\n");
                        continue;
                }
                if(strcmp(command, "exit\n") == 0)
                        break;
                if(strcmp(command, "\n") == 0)
                        continue;

                size_t args_num = parse_input(args, command);
                run_command(args, args_num);
        }

        free(args);
        free(command);
        return 0;
}
```

In the “main” function, I refreshed the “args” first, then I used “get_input” to get the user input in the command line of my unix shell. Then, for command == “exit\n”, the shell will be closed, and for command == “\n”, which means the “Enter in the keyboard”, it will continue working.

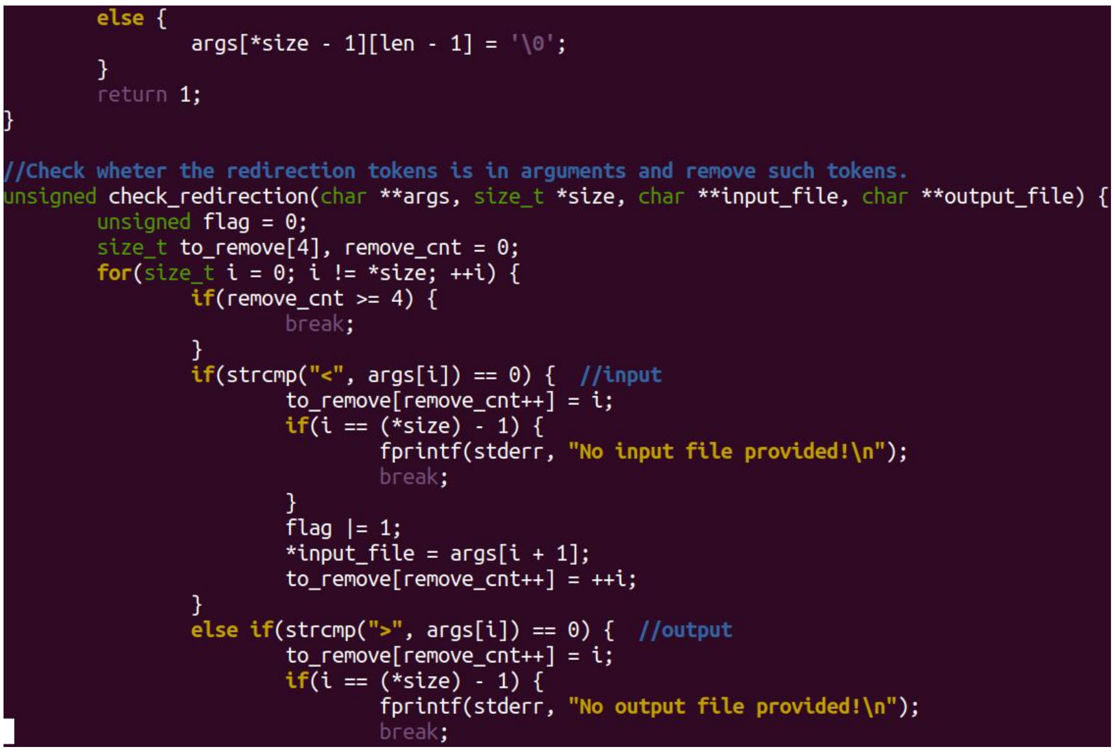
Other Codes:



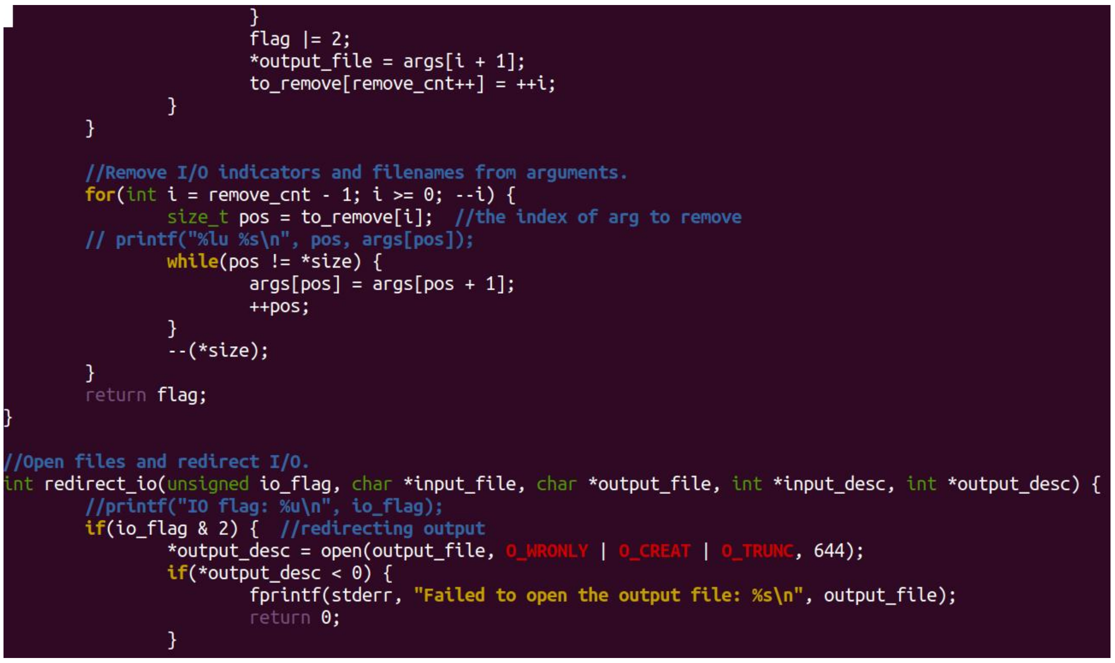
```c
//Put current arguments to NULL, avoid the influence of last iteration.
void refresh_args(char *args[]) {
        while(*args) {
                free(*args);
                *args++ = NULL;
        }
}

int get_input(char *command) {
        char *input_buffer = (char*)malloc((MAX_LINE + 1)*sizeof(char));
        if(fgets(input_buffer, MAX_LINE + 1, stdin) == NULL) {
                fprintf(stderr, "Failed to read input!\n");
                return 0;
        }
        if(strncmp(input_buffer, "!!", 2) == 0) {
                if(strlen(command) == 0) {  // no history yet
                        fprintf(stderr, "No history available yet!\n");
                        return 0;
                }
                printf("Last command is %s", command);    // keep the command unchanged and print it
                return 1;
        }
        strcpy(command, input_buffer);  // update the command

        free(input_buffer);
        return 1;
}
```



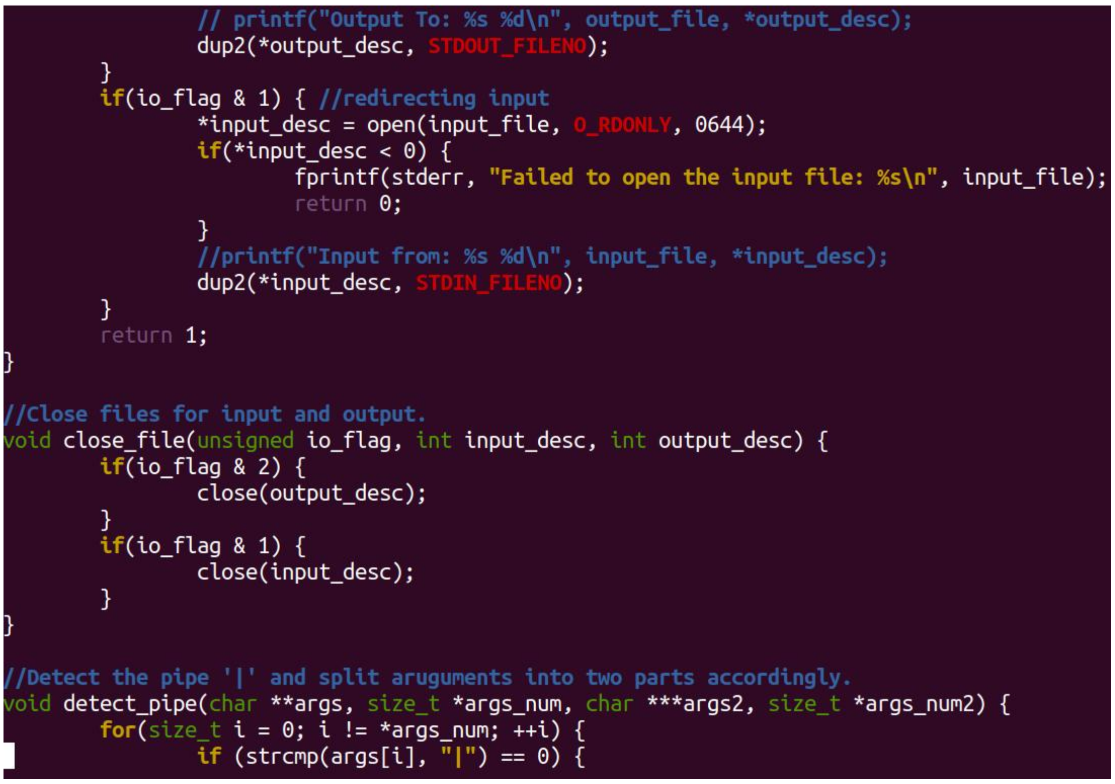
```c
//Splite and store user input "command" in args.
size_t parse_input(char *args[], char *original_command) {
        size_t num = 0;
        char *command = (char*)malloc((MAX_LINE + 1)*sizeof(char));
        strcpy(command, original_command);  // make a copy since 'strtok' will modify it
        char *token = strtok(command, DELIMITERS);
        while(token != NULL) {
                args[num] = (char*)malloc((strlen(token) + 1)*sizeof(char));
                strcpy(args[num], token);
                ++num;
                token = strtok(NULL, DELIMITERS);
        }

        free(command);
        return num;
}

//Check whether an ampersand(&) is in the end of args.
//If so, remove it from args and possibly reduce the size of args.
int check_ampersand(char **args, size_t *size) {
        size_t len = strlen(args[*size - 1]);
        if(args[*size - 1][len - 1] != '&') {
                return 0;
        }
        if(len == 1) {  //remove this argument if it only contains '&'
                free(args[*size - 1]);
                args[*size - 1] = NULL;
                --(*size);  //reduce its size
        }
```



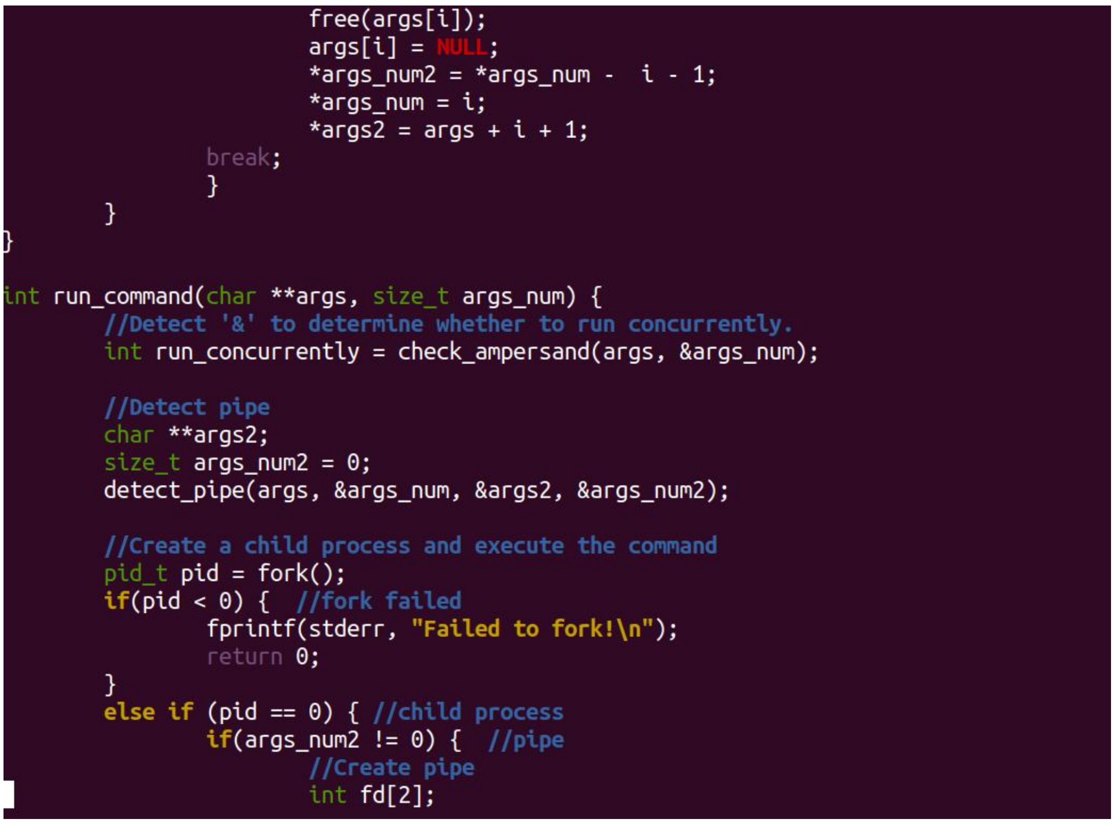
```
        else {
                args[*size - 1][len - 1] = '\0';
        }
        return 1;
}

//Check wheter the redirection tokens is in arguments and remove such tokens.
unsigned check_redirection(char **args, size_t *size, char **input_file, char **output_file) {
        unsigned flag = 0;
        size_t to_remove[4], remove_cnt = 0;
        for(size_t i = 0; i != *size; ++i) {
                if(remove_cnt >= 4) {
                        break;
                }
                if(strcmp("<", args[i]) == 0) {  //input
                        to_remove[remove_cnt++] = i;
                        if(i == (*size) - 1) {
                                fprintf(stderr, "No input file provided!\n");
                                break;
                        }
                        flag |= 1;
                        *input_file = args[i + 1];
                        to_remove[remove_cnt++] = ++i;
                }
                else if(strcmp(">", args[i]) == 0) {  //output
                        to_remove[remove_cnt++] = i;
                        if(i == (*size) - 1) {
                                fprintf(stderr, "No output file provided!\n");
                                break;
```



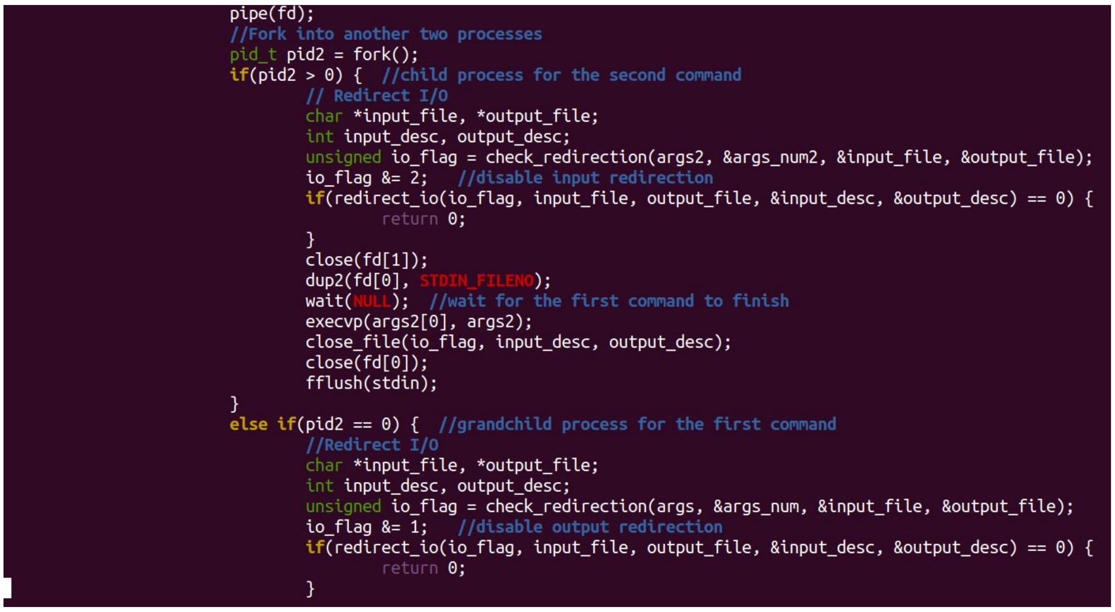
```
                        }
                        flag |= 2;
                        *output_file = args[i + 1];
                        to_remove[remove_cnt++] = ++i;
                }
        }

        //Remove I/O indicators and filenames from arguments.
        for(int i = remove_cnt - 1; i >= 0; --i) {
                size_t pos = to_remove[i];  //the index of arg to remove
        // printf("%lu %s\n", pos, args[pos]);
                while(pos != *size) {
                        args[pos] = args[pos + 1];
                        ++pos;
                }
                --(*size);
        }
        return flag;
}

//Open files and redirect I/O.
int redirect_io(unsigned io_flag, char *input_file, char *output_file, int *input_desc, int *output_desc) {
        //printf("IO flag: %u\n", io_flag);
        if(io_flag & 2) {  //redirecting output
                *output_desc = open(output_file, O_WRONLY | O_CREAT | O_TRUNC, 644);
                if(*output_desc < 0) {
                        fprintf(stderr, "Failed to open the output file: %s\n", output_file);
                        return 0;
                }
```



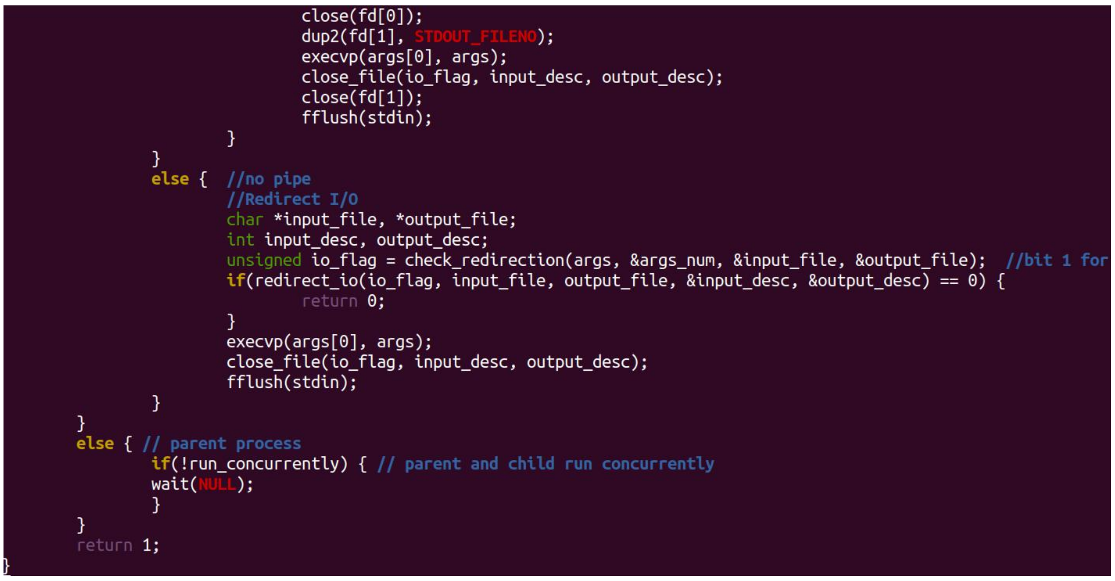
```
                // printf("Output To: %s %d\n", output_file, *output_desc);
                dup2(*output_desc, STDOUT_FILENO);
        }
        if(io_flag & 1) { //redirecting input
                *input_desc = open(input_file, O_RDONLY, 0644);
                if(*input_desc < 0) {
                        fprintf(stderr, "Failed to open the input file: %s\n", input_file);
                        return 0;
                }
                //printf("Input from: %s %d\n", input_file, *input_desc);
                dup2(*input_desc, STDIN_FILENO);
        }
        return 1;
}

//Close files for input and output.
void close_file(unsigned io_flag, int input_desc, int output_desc) {
        if(io_flag & 2) {
                close(output_desc);
        }
        if(io_flag & 1) {
                close(input_desc);
        }
}

//Detect the pipe '|' and split aruguments into two parts accordingly.
void detect_pipe(char **args, size_t *args_num, char ***args2, size_t *args_num2) {
        for(size_t i = 0; i != *args_num; ++i) {
                if (strcmp(args[i], "|") == 0) {
```



```
                        free(args[i]);
                        args[i] = NULL;
                        *args_num2 = *args_num -  i - 1;
                        *args_num = i;
                        *args2 = args + i + 1;
                break;
                }
        }
}

int run_command(char **args, size_t args_num) {
        //Detect '&' to determine whether to run concurrently.
        int run_concurrently = check_ampersand(args, &args_num);

        //Detect pipe
        char **args2;
        size_t args_num2 = 0;
        detect_pipe(args, &args_num, &args2, &args_num2);

        //Create a child process and execute the command
        pid_t pid = fork();
        if(pid < 0) {  //fork failed
                fprintf(stderr, "Failed to fork!\n");
                return 0;
        }
        else if (pid == 0) { //child process
                if(args_num2 != 0) {  //pipe
                        //Create pipe
                        int fd[2];
```



```
            pipe(fd);
            //Fork into another two processes
            pid_t pid2 = fork();
            if(pid2 > 0) {  //child process for the second command
                // Redirect I/O
                char *input_file, *output_file;
                int input_desc, output_desc;
                unsigned io_flag = check_redirection(args2, &args_num2, &input_file, &output_file);
                io_flag &= 2;   //disable input redirection
                if(redirect_io(io_flag, input_file, output_file, &input_desc, &output_desc) == 0) {
                    return 0;
                }
                close(fd[1]);
                dup2(fd[0], STDIN_FILENO);
                wait(NULL);  //wait for the first command to finish
                execvp(args2[0], args2);
                close_file(io_flag, input_desc, output_desc);
                close(fd[0]);
                fflush(stdin);
            }
            else if(pid2 == 0) {  //grandchild process for the first command
                //Redirect I/O
                char *input_file, *output_file;
                int input_desc, output_desc;
                unsigned io_flag = check_redirection(args, &args_num, &input_file, &output_file);
                io_flag &= 1;   //disable output redirection
                if(redirect_io(io_flag, input_file, output_file, &input_desc, &output_desc) == 0) {
                    return 0;
                }
```



```
                close(fd[0]);
                dup2(fd[1], STDOUT_FILENO);
                execvp(args[0], args);
                close_file(io_flag, input_desc, output_desc);
                close(fd[1]);
                fflush(stdin);
            }
        }
        else {  //no pipe
            //Redirect I/O
            char *input_file, *output_file;
            int input_desc, output_desc;
            unsigned io_flag = check_redirection(args, &args_num, &input_file, &output_file);  //bit 1 for
            if(redirect_io(io_flag, input_file, output_file, &input_desc, &output_desc) == 0) {
                return 0;
            }
            execvp(args[0], args);
            close_file(io_flag, input_desc, output_desc);
            fflush(stdin);
        }
    }
    else { // parent process
        if(!run_concurrently) { // parent and child run concurrently
        wait(NULL);
        }
    }
    return 1;
}
```

<1> The separate of "command" into args[] is implemented in the function "parse_input".

<2> The "Creating the child process and executing the command in the child" is implemented in the function "run_command".

<3> The "History feature !!" is implemented in the function "get_input".

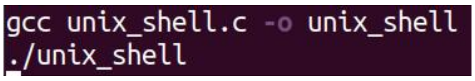
<4> The "Redirecting Input and Output" is implemented in the function "check_redirection", "redirect_io", and "chose_files".

<5> The "Communicating via a Pipe" is implemented in the function "detect_pipe".

<6> The "Ampersand &" is implemented in the function "check_ampersand".

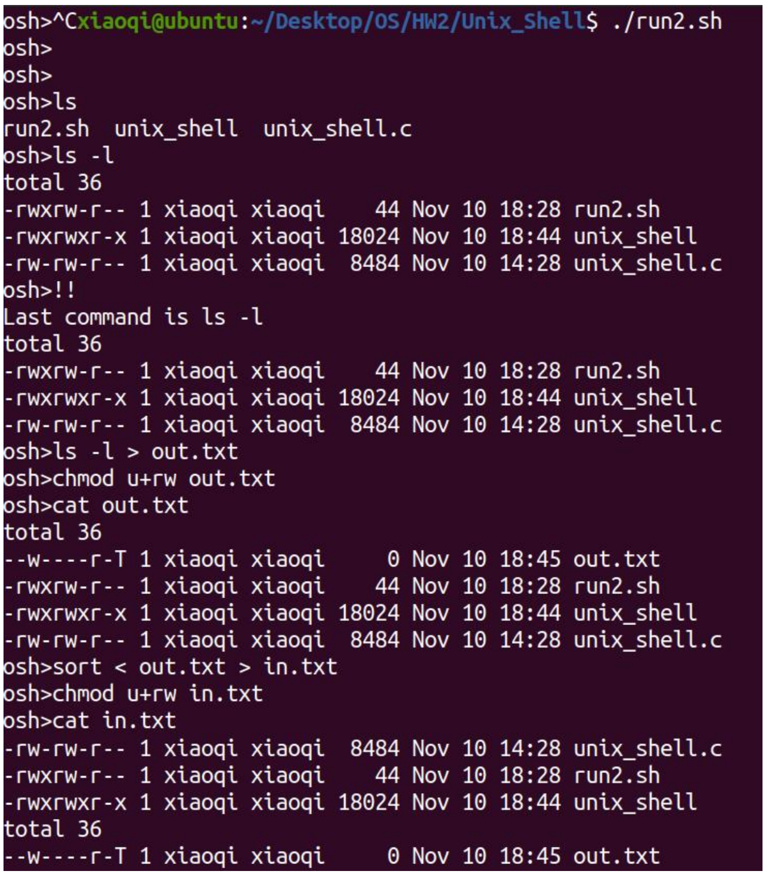
<7> When we start to run the command "run_command", the program will check all the information and status above, and combine them together to execute the input of the unix shell.

Compile (run2.sh):

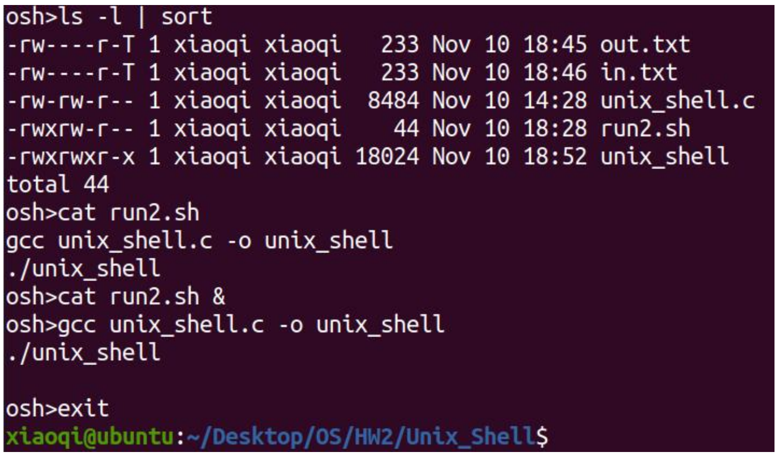


```
gcc unix_shell.c -o unix_shell
./unix_shell
```

Results:



```
osh>^Cxiaoqi@ubuntu:~/Desktop/OS/HW2/Unix_Shell$ ./run2.sh
osh>
osh>
osh>ls
run2.sh  unix_shell  unix_shell.c
osh>ls -l
total 36
-rwxrw-r-- 1 xiaoqi xiaoqi    44 Nov 10 18:28 run2.sh
-rwxrwxr-x 1 xiaoqi xiaoqi 18024 Nov 10 18:44 unix_shell
-rw-rw-r-- 1 xiaoqi xiaoqi  8484 Nov 10 14:28 unix_shell.c
osh>!!
Last command is ls -l
total 36
-rwxrw-r-- 1 xiaoqi xiaoqi    44 Nov 10 18:28 run2.sh
-rwxrwxr-x 1 xiaoqi xiaoqi 18024 Nov 10 18:44 unix_shell
-rw-rw-r-- 1 xiaoqi xiaoqi  8484 Nov 10 14:28 unix_shell.c
osh>ls -l > out.txt
osh>chmod u+rw out.txt
osh>cat out.txt
total 36
--w----r-T 1 xiaoqi xiaoqi     0 Nov 10 18:45 out.txt
-rwxrw-r-- 1 xiaoqi xiaoqi    44 Nov 10 18:28 run2.sh
-rwxrwxr-x 1 xiaoqi xiaoqi 18024 Nov 10 18:44 unix_shell
-rw-rw-r-- 1 xiaoqi xiaoqi  8484 Nov 10 14:28 unix_shell.c
osh>sort < out.txt > in.txt
osh>chmod u+rw in.txt
osh>cat in.txt
-rw-rw-r-- 1 xiaoqi xiaoqi  8484 Nov 10 14:28 unix_shell.c
-rwxrw-r-- 1 xiaoqi xiaoqi    44 Nov 10 18:28 run2.sh
-rwxrwxr-x 1 xiaoqi xiaoqi 18024 Nov 10 18:44 unix_shell
total 36
--w----r-T 1 xiaoqi xiaoqi     0 Nov 10 18:45 out.txt
```



```
osh>ls -l | sort
-rw----r-T 1 xiaoqi xiaoqi   233 Nov 10 18:45 out.txt
-rw----r-T 1 xiaoqi xiaoqi   233 Nov 10 18:46 in.txt
-rw-rw-r-- 1 xiaoqi xiaoqi  8484 Nov 10 14:28 unix_shell.c
-rwxrw-r-- 1 xiaoqi xiaoqi    44 Nov 10 18:28 run2.sh
-rwxrwxr-x 1 xiaoqi xiaoqi 18024 Nov 10 18:52 unix_shell
total 44
osh>cat run2.sh
gcc unix_shell.c -o unix_shell
./unix_shell
osh>cat run2.sh &
osh>gcc unix_shell.c -o unix_shell
./unix_shell

osh>exit
xiaoqi@ubuntu:~/Desktop/OS/HW2/Unix_Shell$
```

<1>You can see when I did “ls”, “ls -l”, and “!!”, it works well.

<2> “ls -l > out.txt” put the result of “ls -l” to the file.

<3> “sort < out.txt > in.txt”, sort the data in out.txt, and put the result into in.txt.

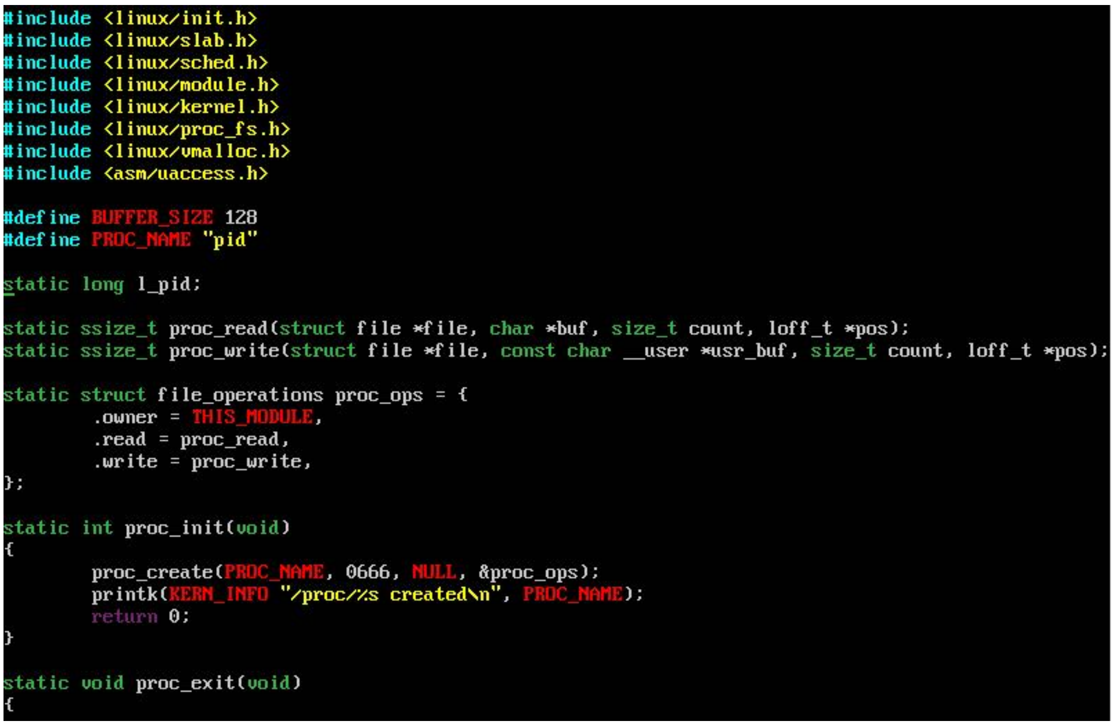
<4> “ls -l | sort”, put the result of “ls -l” to “sort”.

<5> “cat run2.sh &” run the parent and child concurrently.

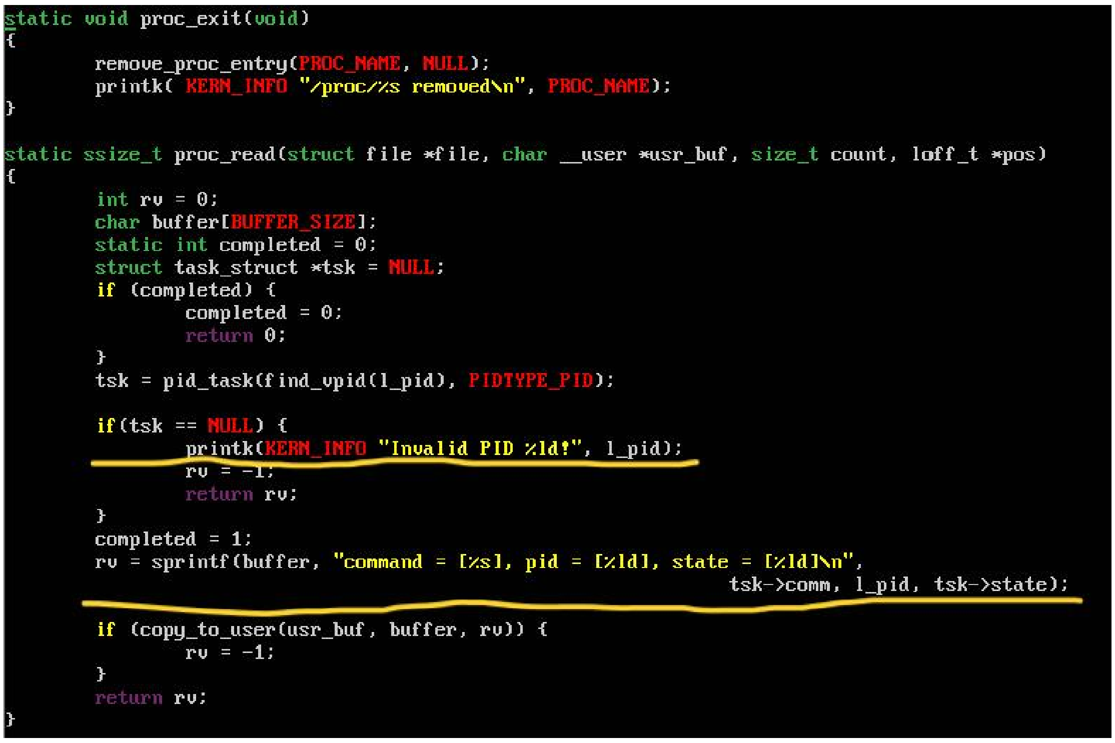
<6> All processes above shows that my Unix Shell reach the requirements of the project.

Part 3: Project - Kernel Module for Task Information

Codes:

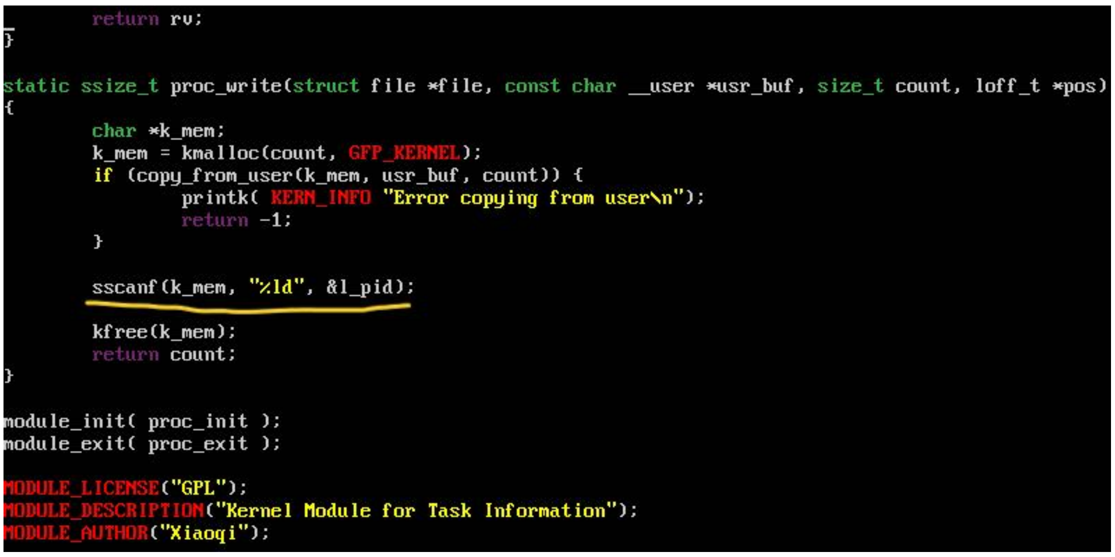


```
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/vmalloc.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "pid"

static long l_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,
};

static int proc_init(void)
{
        proc_create(PROC_NAME, 0666, NULL, &proc_ops);
        printk(KERN_INFO "/proc/%s created\n", PROC_NAME);
        return 0;
}

static void proc_exit(void)
{
```



```
static void proc_exit(void)
{
        remove_proc_entry(PROC_NAME, NULL);
        printk( KERN_INFO "/proc/%s removed\n", PROC_NAME);
}

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) {
                printk(KERN_INFO "Invalid PID %ld!", l_pid);
                rv = -1;
                return rv;
        }
        completed = 1;
        rv = sprintf(buffer, "command = [%s], pid = [%ld], state = [%ld]\n",
                                                        tsk->comm, l_pid, tsk->state);

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

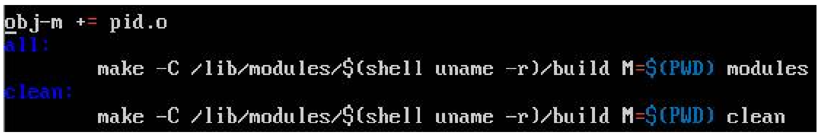
<1> I wrote this program based on the “pid.c” of the book.
The codes with yellow mark are the only places I made the changes.
<2> Here, I add a “if(tsk == NULL)” situation, to let the program break once the PID which the user give is an invalid PID. After we find the task, I output the results by “sprintf()”.



```
        return rv;
}

static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t count, loff_t *pos)
{
        char *k_mem;
        k_mem = kmalloc(count, GFP_KERNEL);
        if (copy_from_user(k_mem, usr_buf, count)) {
                printk( KERN_INFO "Error copying from user\n");
                return -1;
        }

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

        kfree(k_mem);
        return count;
}

module_init( proc_init );
module_exit( proc_exit );

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Kernel Module for Task Information");
MODULE_AUTHOR("Xiaoqi");
```

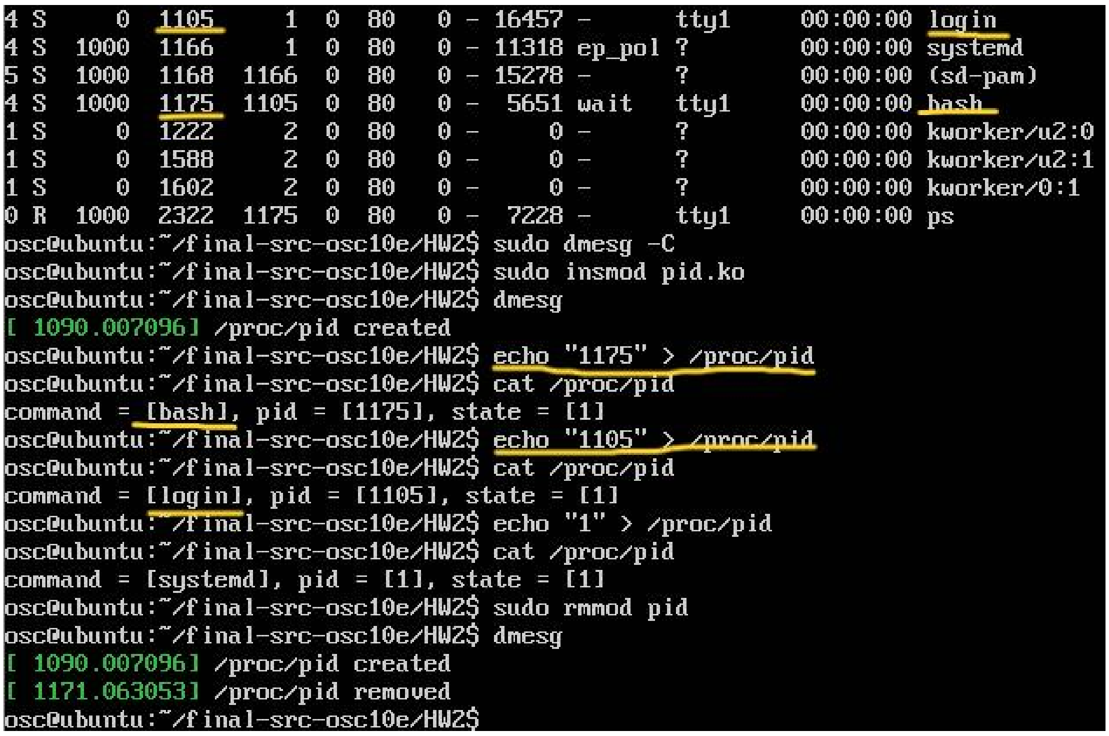
<3> Here I used "sscanf()", since the "kstrl()" will not work because the strings are not guaranteed to be null_terminated.

Compile (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
```

Results:



```
4 S     0  1105     1  0  80   0 - 16457 -      tty1     00:00:00 login
4 S  1000  1166     1  0  80   0 - 11318 ep_pol ?        00:00:00 systemd
5 S  1000  1168  1166  0  80   0 - 15278 -      ?        00:00:00 (sd-pam)
4 S  1000  1175  1105  0  80   0 -  5651 wait   tty1     00:00:00 bash
1 S     0  1222     2  0  80   0 -     0 -      ?        00:00:00 kworker/u2:0
1 S     0  1588     2  0  80   0 -     0 -      ?        00:00:00 kworker/u2:1
1 S     0  1602     2  0  80   0 -     0 -      ?        00:00:00 kworker/0:1
0 R  1000  2322  1175  0  80   0 -  7228 -      tty1     00:00:00 ps
osc@ubuntu:~/final-src-osc10e/HW2$ sudo dmesg -C
osc@ubuntu:~/final-src-osc10e/HW2$ sudo insmod pid.ko
osc@ubuntu:~/final-src-osc10e/HW2$ dmesg
[ 1090.007096] /proc/pid created
osc@ubuntu:~/final-src-osc10e/HW2$ echo "1175" > /proc/pid
osc@ubuntu:~/final-src-osc10e/HW2$ cat /proc/pid
command = [bash], pid = [1175], state = [1]
osc@ubuntu:~/final-src-osc10e/HW2$ echo "1105" > /proc/pid
osc@ubuntu:~/final-src-osc10e/HW2$ cat /proc/pid
command = [login], pid = [1105], state = [1]
osc@ubuntu:~/final-src-osc10e/HW2$ echo "1" > /proc/pid
osc@ubuntu:~/final-src-osc10e/HW2$ cat /proc/pid
command = [systemd], pid = [1], state = [1]
osc@ubuntu:~/final-src-osc10e/HW2$ sudo rmmod pid
osc@ubuntu:~/final-src-osc10e/HW2$ dmesg
[ 1090.007096] /proc/pid created
[ 1171.063053] /proc/pid removed
osc@ubuntu:~/final-src-osc10e/HW2$
```

<1> As you can see, I used “ps -el” first to see the actual PID of each process, and then I used my program to check.
<2> The PID of “bash” is 1175, the PID of “login” is 1105. After I write the number into /proc/pid and read it, the results are the same, which means my program works well.
<3> I used to want to use my own Virtual Machine to do this Kernel Module project, but due to some problem, maybe the difference of version and environment, there are a lot of problems such as: When I did “echo ‘1111’ > /proc/pid” in my computer, it said “bash: echo: input/output error”, when I did “cat /proc/cat”, it said “no permission”. After fixing the bugs for several hours and nothing works, finally I choose to use the source virtual machine of the book, and the bugs just disappeared. ^^

Reference:
Part2: https://github.com/forestLoop/Learning-EI338/tree/master/Project-2-1

That’s the end of this report, thank you very much for your attention!
Xiaoqi LIU 999009335