Report of OS Project 2 & 3

郭嘉宋 517030910374

Assignment

Chapter 2:

Project 1 — UNIX Shell

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. Completing this project will involve using the UNIX fork(), exec(), wait(), dup2(), and pipe() system calls and can be completed on any Linux, UNIX, or macOS system. This project is organized into several parts:

- 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

---

Project 2 — Linux Kernel Module for Task Information

In this project, you will write a Linux kernel module that uses the /proc file system for displaying a task's information based on its process identifier value pid. Before beginning this project, be sure you have completed the Linux kernel module programming project in Chapter 2, which involves creating an entry in the /proc file system. This project will involve 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 of how your kernel module will be accessed once loaded into the system is as follows:

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

The echo command writes the characters "1395" to the /proc/pid file. Your kernel module will read this value and store its integer equivalent as it represents a process identifier. The cat command reads from /proc/pid, where your kernel module will retrieve the three fields from the task struct associated with the task whose pid value is 1395.

Chapter 3:

Project 1 — Multithreaded Sorting Application

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term sorting threads) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread—a merging thread —which merges the two sublists into a single sorted list. Because global data are shared across all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured as in Figure 4.27. This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting. Refer to the instructions in Project 1 for details on passing parameters to a thread. The parent thread will output the sorted array once all sorting threads have exited.

Project 2 — Fork-Join Sorting Application

Implement the preceding project (Multithreaded Sorting Application) using Java's fork-join parallelism API. This project will be developed in two different versions. Each version will implement a different divide-and-conquer sorting algorithm:

- Quicksort
- Mergesort

The Quicksort implementation will use the Quicksort algorithm for dividing the list of elements to be sorted into a left half and a right half based on the position of the pivot value. The Mergesort algorithm will divide the list into two evenly sized halves. For both the Quicksort and Mergesort algorithms, when the list to be sorted falls within some threshold value (for example, the list is size 100 or fewer), directly apply a simple algorithm such as the Selection or Insertion sort. Most data structures texts describe these two well-known, divide-and-conquer sorting algorithms. The class SumTask shown in Section 4.5.2.1 extends RecursiveTask, which is a result-bearing ForkJoinTask. As this assignment will involve sorting the array that is passed to the task, but not returning any values, you will instead create a class that extends RecursiveAction, a non result-bearing ForkJoinTask (see Figure 4.19). The objects passed to each sorting algorithm are required to implement Java's Comparable interface, and this will need to be reflected in the class definition for each sorting algorithm. The source code download for this text includes Java code that provides the foundations for beginning this project.

Environment

- Virtual platform: VMware Workstation Pro 15.0
- Operating system: Ubuntu 18.04
- Kernel: Linux 5.3.1
- Local system: Windows 10
- Python: Python 3.7

Solution

Chapter 2:

Project 1 — UNIX Shell

```
#define MAX_LINE 50

int wait_or_not = 1;
char input[50];
char args[50][50];
char history[50] = {0};
int num = 0;
int detect()
{
                if(num == 0)
                        return 0;
                if(strcmp(args[0], "!!")==0)
                {
                        if(!history[0])
                        {
                                printf("No commands in history! \n");
                                return 0;
                        }

                                strcpy(input, history);
                        return detect();
                }

        int i = 0;
        while(input[i])
        {
                history[i] = input[i];
                i++;
        }
        strcpy(history,input);
        if(strcmp(args[0], "exit")==0)
                exit(0);
                if(strcmp(args[num-1],"&")==0)
                        {wait_or_not = 0;
                                num--;}
        return 1;
}

int execute()
{
        int type = 0,i,j;
        char *filename;
        char *tmp_1[MAX_LINE/2+1];
        char *tmp_2[MAX_LINE/2+1];
        for(i = 0;i<num;i++)
                tmp_1[i] = args[i];
        tmp_1[i] = NULL;
        for(i = 0;i < num;i++)
```

```
        {
                if(strcmp(args[i],">") == 0)
                {
                        filename = args[i+1];
                        type = 2;
                        tmp_1[i] = NULL;
                }
                else if(strcmp(args[i], "<")==0)
                {
                        filename = args[i+1];
                        type = 1;
                        tmp_1[i] = NULL;
                }
                else if(strcmp(args[i],"|")==0)
                {
                        type = 3;
                        tmp_1[i] = NULL;
                        for(j = i+1;j<num;j++)
                        {
                                tmp_2[j-i-1] = args[j];
                        }
                        tmp_2[j-i-1] = NULL;
                }
        }
        int pid = fork();
        int in, out;
        int fds[2];
        if(pid < 0)
        {
                perror("there is a fork error!!");
                exit(0);
        }
        else if(pid > 0)
        {
                if(wait_or_not == 1)
                        waitpid(pid,NULL,0);
                wait_or_not = 1;
        }
        else
        {
                switch(type)
                {
                        case 0:
                        execvp(tmp_1[0],tmp_1);
                        break;
                        case 1:
                        in = open(filename, O_RDONLY);
                        dup2(in,STDIN_FILENO);
                        execvp(tmp_1[0],tmp_1);
                        close(in);
                        break;
                        case 2:
                        out = open(filename, O_WRONLY|O_CREAT, 0666);
                        dup2(out, STDOUT_FILENO);
```

```
                        execvp(tmp_1[0],tmp_1);
                        close(out);
                        break;
                        case 3:
                        pipe(fds);
                        int pids = fork();
                        if(pids<0)
                        {
                                perror("there is a fork error!");
                                exit(0);
                        }
                        else if(pids>0)
                        {
                                close(fds[1]);
                                dup2(fds[0],STDIN_FILENO);
                                execvp(tmp_2[0],tmp_2);
                        }
                        else
                        {
                                close(fds[0]);
                                dup2(fds[1],STDOUT_FILENO);
                                execvp(tmp_1[0],tmp_1);
                        }
                break;
                }
        }
        return 1;
}

int main(void)
{


int should_run = 1;
while (should_run)
{
printf("osh > ");
fflush(stdout);
gets(input);
char *p = input;
strcat(input," ");
int t = 0, index = 0;
while(*p)
{
    if(!isspace(*p))
    {
        args[index][t] = *p;
        t++;
    }
    if(isspace(*p))
    {
        args[index][t] = '\0';
        index++;
        t = 0;
```

```
    }
    p++;

}
num = index;
int tmp = detect();
if(tmp == 0)
continue;
execute();

}
return 0;
}}
```

- The head files are showed in source code.
- "args" is the input array.
- Using "fork" to create a new process.
- Through "execvp" to use system calls, and return the result doing by kernel modes.
- "pipe(fds)" to use pipe. Be sure to use STDIN_FILENO in parent-process and STDOUT_FILENO in child-process.

Project 2 — Linux Kernel Module for Task Information

```
#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)
{
```

```
        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);

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

        k_mem = kmalloc(count, GFP_KERNEL);

        copy_from_user(k_mem,usr_buf,count);
        k_mem[count-1] = 0;
        sscanf(k_mem, "%ld", &l_pid);
        kfree(k_mem);
        return count;
}

module_init( proc_init );
module_exit( proc_exit );

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Module");
MODULE_AUTHOR("GUO");
```

- Just recording data to /proc/pid and read data from it.
- The read and write function are done in lab1, so project2 of lab2 is quite easy.

Chapter 3:

Project 1 — Multithreaded Sorting Application

```
import threading

def sort(nums, head, tail):
    if head+1 == tail or head == tail or head > tail:
        return
    mid = int((head+tail)/2)

    thread1 = threading.Thread(target=sort, args=(nums, head, mid))
    thread1.start()
    thread2 = threading.Thread(target=sort, args=(nums, mid, tail))
    thread2.start()
    thread1.join()
    thread2.join()

    nums.sort()
    return

def main():
    nums = [56, 234, 53, 633, 1, 6, 34, 763, 12, 5, 3245, 2352352, 4]
    sort(nums, 0, len(nums))
    for num in nums:
        print(num, end=" ")


if __name__ == "__main__":
    main()
```

- I use python in Windows to finish this project.
- Use "sort" function in python library to sort the list.
- Use join() and start() to create processes and combine processes.

Project 2 — Fork-Join Sorting Application

```
# quick sort
import threading

def quick_sort(nums, head, tail):
    if head+1 == tail or head == tail or head > tail:
        return
    i, j = head, tail-1
    while (i < j):
        while (i < j and nums[j] >= nums[i]):
            j -= 1
        nums[i], nums[j] = nums[j], nums[i]
```

```
        while (i < j and nums[i] <= nums[j]):
            i += 1
        nums[i], nums[j] = nums[j], nums[i]

    thread1 = threading.Thread(target=quick_sort, args=(nums, head, j))
    thread1.start()
    thread2 = threading.Thread(target=quick_sort, args=(nums, j+1, tail))
    thread2.start()
    thread1.join()
    thread2.join()


def main():
    nums = [56, 234, 53, 633, 1, 6, 34, 763, 12, 5, 3245, 2352352, 4]
    quick_sort(nums, 0, len(nums))
    for num in nums:
        print(num, end=" ")


if __name__ == "__main__":
    main()
```

```
import threading

def merge_sort(nums, head, tail):
    if head+1 == tail or head == tail or head > tail:
        return
    mid = int((head+tail)/2)

    thread1 = threading.Thread(target=merge_sort, args=(nums, head, mid))
    thread1.start()
    thread2 = threading.Thread(target=merge_sort, args=(nums, mid, tail))
    thread2.start()
    thread1.join()
    thread2.join()

    tmp, i, j = list(), head, mid
    while i < mid and j < tail:
        if nums[i] < nums[j]:
            tmp.append(nums[i])
            i += 1
        else:
            tmp.append(nums[j])
            j += 1
    while i < mid:
        tmp.append(nums[i])
        i += 1
    while j < tail:
        tmp.append(nums[j])
        j += 1
```

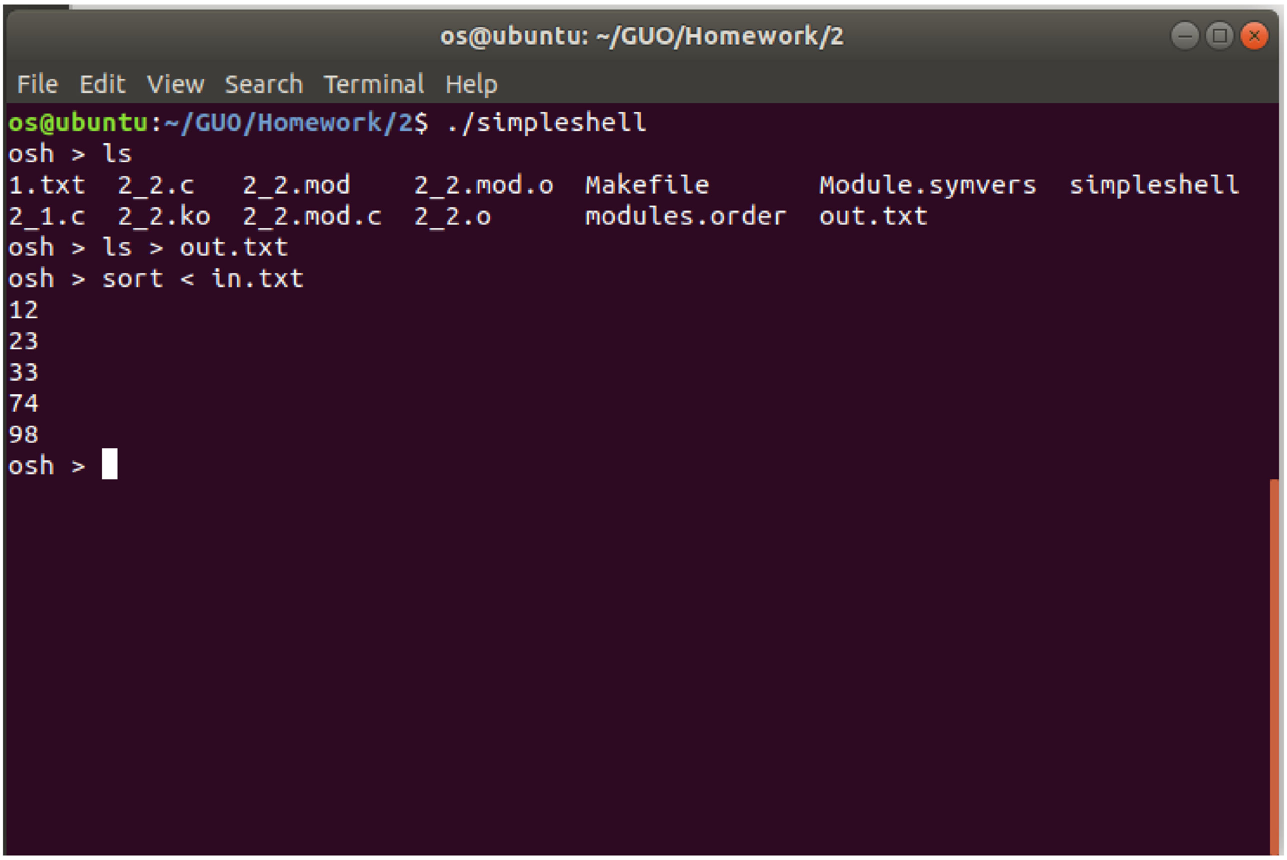
```
        for k in range(tail - head):
            nums[head+k] = tmp[k]
        return



def main():
    nums = [56, 234, 53, 633, 1, 6, 34, 763, 12, 5, 3245, 2352352, 4]
    merge_sort(nums, 0, len(nums))
    for num in nums:
        print(num, end=" ")


if __name__ == "__main__":
    main()
```

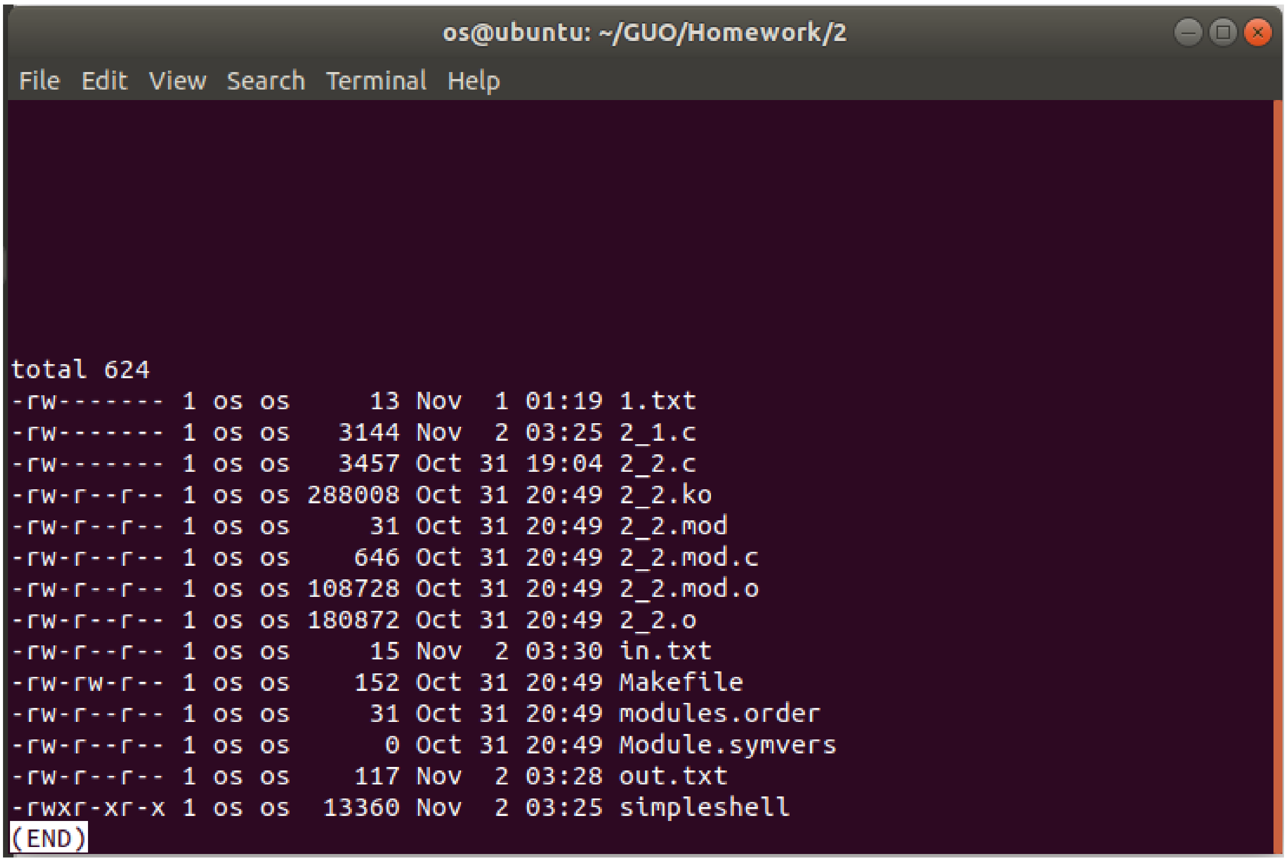
- Using the same structure as project 1.
- This time use "quick_sort" and "merge_sort" instead of "sort".

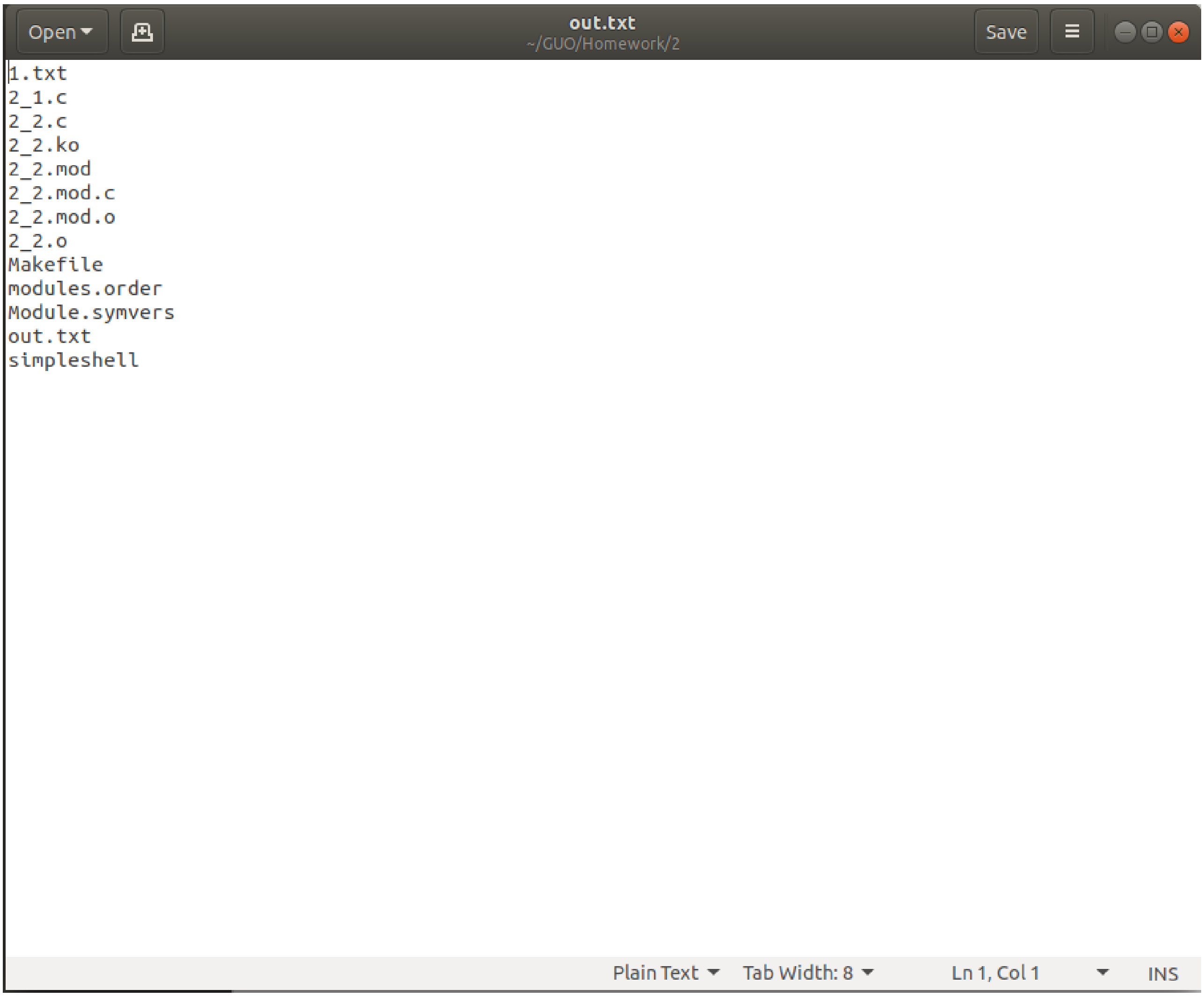
Experiment Result

Chapter 2:

- Project 1

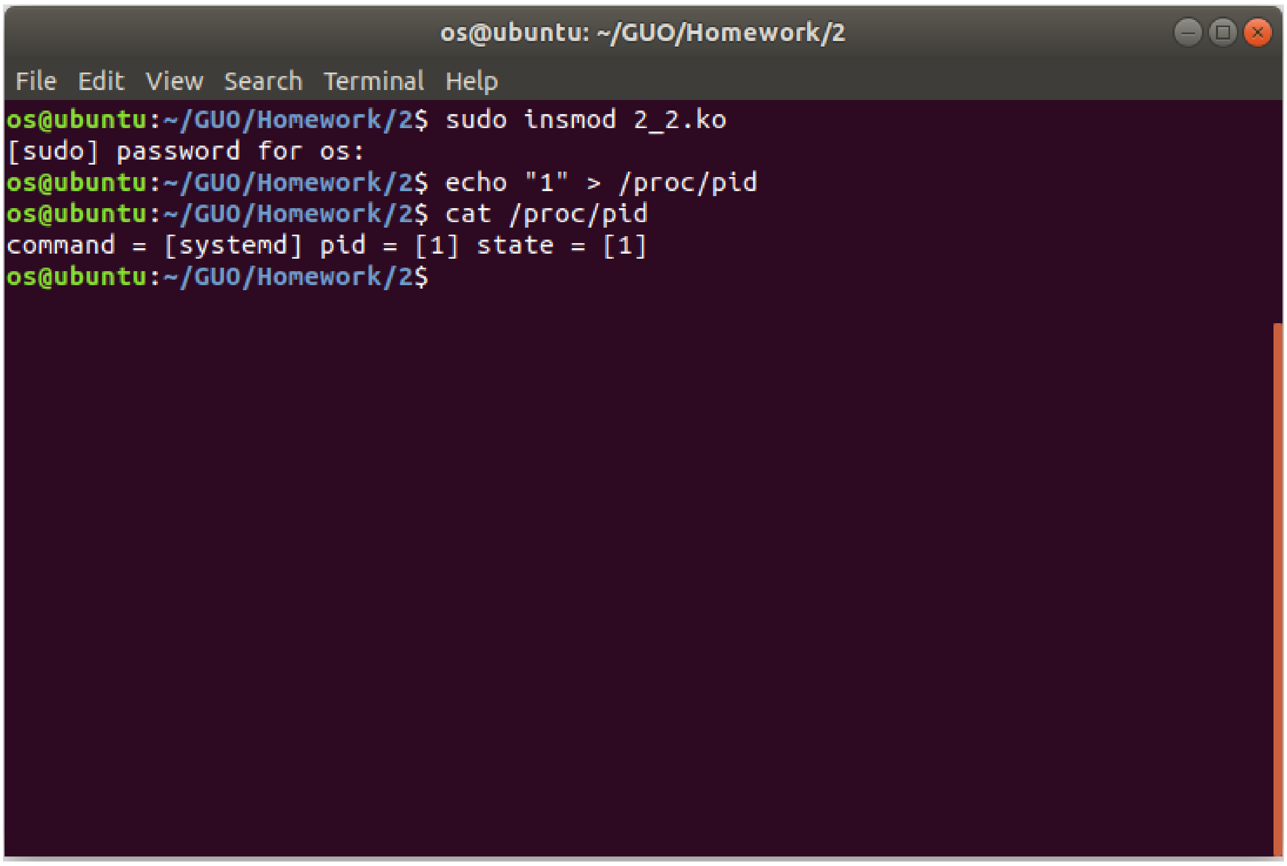






Simpleshell is running successfully. And remember to use gcc to compile.

- Project 2



Information of "pid" is showed successfully.

Chapter 3:

```
C:\python\python.exe C:/Users/GUO/Desktop/EI338-maste
1 4 5 6 12 34 53 56 234 633 763 3245 2352352
Process finished with exit code 0
```

- All 3 py files have the same input, so the outputs are the same.

Logs

This is my second time to develop a Linux kernel and my first time to learn process. I learned how to use "fork", "pipe" and so many other commands. I felt satisfied and I hope I can learn more in further study. Finally, thanks teacher and TA for reading my report and code.

517030910374 郭嘉宋