SHANGHAI JIAO TONG UNIVERSITY

IEEE CLASS

Computer Systems Engineering

The report of Project 2 and 3

Author:
Yang Nianzu

Student Number:
517030910301

October 23, 2019

Contents

1 The environment of the experiment

I finish this experiment on a Linux virtual machine through VMware. And the version of my Linux is 14.04.5. In order to make it more convenient to edit files, I installed Sublime on my virtual machine.

2 Project2-1:UNIX Shell

2.1 Analysis

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. Our implemention will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. And it can remember the last command we input.

The main idea is that we have the parent process first read what the user enters on the command line and then create a separate child process that performs the command. And we have to judge what is the type of the command first before it make response. The codes of the judgement will not be demonstrated due to that it's not so difficult. You can see them in the .c files.

First, we need to judge whether a command has ended and record the command. So I design a function which can satisfies the two goals. It uses "
n" as a signal that a command has ended. Besides, it uses a char array to store the whole command and return it. We call this function "get_the_input". Here is its code.

```
/* judge whether a command has ended and record the command*/
void *get_the_input(char *input){
    int i, enter, first_non_space, last_non_space;
    char *output;
    first_non_space = 0;
    last_non_space = 0;
    output = malloc(sizeof(char) * MAX_LINE);
    for(i=0;i<MAX_LINE;i++){
        if(input[i] == '\n'){
            enter = i;
            break;
        }
    }
    for(i=0;i<MAX_LINE && input[i] == ' ';i++)  ;
    first_non_space = i;
```

```
    for(i=enter-1;i>=0 && input[i] == ' ';i--)    ;
    last_non_space = i;
    strcpy(output, input + first_non_space);
    output[last_non_space - first_non_space+1] = '\0';
    input = output;
}
```

Listing 1: function "get_the_input"

Then we begin to realize its different responses corresponding to different types of commands through modifying the main() function provided in the textbook. We need to notice that we have to reset the input initially in every loop through the following code.

```
        for(i=0;i<MAX_LINE/2+1;i++)
            memset(args[i], 0, sizeof(char) * MAX_LINE);
```

Listing 2: reset the input

UNIX shells typically allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand & at the end of the command. So we need to check whether the user included & to determine whether or not the parent process is to wait for the child to exit. Judgement is easy to finish. For this problem, I defined a bool value named wait_flag. Its initial value is 1. But if we input & at the end of the command, its value and turn to 0. According to different values, we make different responses.

We asked to creating a history feature. It's quite easy. We only need to define a char array to store the last command. If we enter "!!", we just copy the history command as current input.

The shell should also be modified to support the ">" and "<" redirection operators, where ">" redirects the output of a command to a file and "<" redirects the input to a command from a file. It seems difficult to cope with. However, fortunately, the textbook has given us a hint that managing the redirection of both input and output will involve using the dup2() function, which duplicates an existing file discriptor to another file descriptor. How to use this function can be found easily on the Internet. The following codes are the part concerning handling these two kind of redirection operations.

```
            if(out_redirect_flag){
                fd = open(redirect_path, O_WRONLY | O_TRUNC | O_CREAT,
                    0644);
```

```
            if(fd < 0)  {perror("open error!"); exit(1);}
            /* dup2(oldfd, newfd) (close the stdout) */
            if(dup2(fd, 1) < 0) {perror("dup2 error!"); exit(1);}
            close(fd);
        }else if(in_redirect_flag){
            fd = open(redirect_path, O_RDONLY, 0644);
            if(fd < 0)  {perror("open error!"); exit(1);}
            /* dup2(oldfd, newfd) (close the stdin) */
            if(dup2(fd, 0) < 0) {perror("dup2 error!");exit(1);}
            close(fd);
        }
```

Listing 3: redirection

In the project 2-1, now only one problem remains unsolved. It's about establishing a pipe to allow the output of one command to serve as input to another using a pipe. Let's take the child process as an example. First, we close the stdin and the write port of the pipe. Then we use dup() function to get the 0 file descriptor and set its pointer to the read port of the pipe. For parent process, it's almost the same. Here are the codes realizing the function.

```
        if(!pipe_flag)  res = execvp(valid_args[0], valid_args);
        else{
            /* for a cmd1 | cmd2 simple pipeline problem */
            /* we create a child for the child and use IPC between
               them to finish the pipeline */
            int p[2], ppid;
            pipe(p);
            ppid = fork();
            if(ppid < 0)    {perror("ppid create error!");exit(1);}
            else if(ppid == 0){
                /* for the child of the child process, set it as
                   the executor of the second commmand */
                close(0);
                close(p[1]);
                if(dup(p[0]) < 0)   {perror("dup error!");exit(1);}
                res = execvp(pipe_args[0], pipe_args);
                if(res < 0) { perror("cmd2 execution error!"); exit
                    (1); }
            }else{
                /* for the parent process, it will execute the
                   first command */
                /* close stdout */
                close(1);
                /* close one port */
                close(p[0]);
                if(dup(p[1]) < 0)   {perror("dup error!");exit(1);}
```

```
23                      res = execvp(valid_args[0], valid_args);
                        if(res < 0) { perror("cmd1 execution error!"); exit
                            (1); }
25                  }
                }
```

Listing 4: pipe

Now, the whole problem has been solved. I write a .c file named Unix_shell for this project.

2.2 Result

The results will be demonstrated as follows.

```
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project2$ ./Unix_shell
osh>ls
Makefile  Unix_shell  Unix_shell.c  Unix_shell.o
osh>!!
Makefile  Unix_shell  Unix_shell.c  Unix_shell.o
```

Figure 1: Creating a history feature

```
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project2$ ./Unix_shell
osh>ls > output
capture the output redirection...
osh>sort < output
capture the input redirectoion...
Makefile
output
Unix_shell
Unix_shell.c
Unix_shell.o
```

Figure 2: Redirecting input and output

```
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project2$ ./Unix_shel
osh>ls | sort
osh>Makefile
output
Unix_shell
Unix_shell.c
Unix_shell.o
```

Figure 3: Communication via a pipe

```
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project2$ ./Unix_shell
osh>cat Makefile &
osh>CC=gcc
SRC_FILE=$(wildcard *.c)
OBJ_FILE=$(patsubst %.c, %.o ,$(SRC_FILE))
target=Unix_shell
all : $(target)
$(target) : $(OBJ_FILE)
        $(CC) -o $@ $^
$(OBJ_FILE) : $(SRC_FILE)
        $(CC) -o $@ -c $^
clean:
        $(RM) $(target) $(OBJ_FILE)
```

Figure 4: Command including &

3 Project2-2:Linux Kernel Module for Task Information

3.1 Analysis

In this project, we 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. 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.

In the kernel module project in last project, we have learned how to read from /proc file system. This time, we should cover how to write to /proc. We just set the field .write in struct file_operations to ./write=proc_write. And the proc_write() function will be called. Then it can get the process identifier user input.

As ilustrated in Section 3.1 in the textbook, the PCB in Linux is represented by the structure task_struct, which is found in the linux/sched.h include file.

For a vlid pid in the system, pid_task() function will return its tast_struct. But if pid_task() is not passed a valid pid, it returns NULL. So we should be careful. The textbook has given detailed enough instructions on this project. Therefore, we won't talk about it too much.

3.2 Result

The result will be demonstrated as follows.

```
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project2/2$ sudo insmod PCB_info.ko
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project2/2$ echo "1" > /proc/PCB_info
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project2/2$ cat /proc/PCB_info
command = init pid = 1 state = 1
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project2/2$ sudo rmmod PCB_info
```

Figure 5: Linux Kernel Module for Task Information

4 Project3-1:Multithread Sorting Application

4.1 Analysis

In this project, we are asked to write a multithreaded sorting problem that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads(which will term sorting threads) sort each sublist using a sorting algorithm. The two sublists are then merged by a third thread —— a merging thread —— which merges the two sublists into a single sorted list.

It's not difficult for us to realize a sorting algorithm. I use bubble sort here.

The true difficulty lies in that how we use threads to achieve our goal. In deed, the textbook has told us the method already. I choose to use Pthreads. So we need include pthread.h header file in our .c file. And its related functions will be listed as follow.

There are three functions about Pthreads which we use. They're pthread_attr_init(), pthread_create() and pthread_join(). The first one is used to set the default attributes of the thread. The second is used to create the thread. The last one is used to wait for the thread to exit. We can use these three function to realize multithread sorting application. How to use them has been introduced in detail in textbook. So I won't demonstrated their usage here.

4.2 Result

I input a array where the intergers ranging from 1 to 80 are in the descending order. We hope after sorting, they will be ordered in ascending order.The result will be demonstrated as follows.

```
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project3/1$ ./sort
Before sorting, the input array is:
80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 1
5 14 13 12 11 10 9 8 7 6 5 4 3 2 1

After the sorting, the output array is:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 6
9 70 71 72 73 74 75 76 77 78 79 80
```

Figure 6: Multithread Sorting Application

5 Project3-2:Fork-Join Sorting Application

5.1 Analysis

We are asked to implement the preceding project(Multithread 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 and Mergesort.

Fortunately, the textbook also have provided us with how to use Java threads. However, I did't learn Java before. But we can learn it online. And many sort algorithms written in the language of Java can be found on the Internet. Only we need to do is combine the existing sort algorithms with Jave threads. It's very similiar to last project in deed.

5.2 Result

For each algorithm, we generate a input array randomly whose size is 50 first. The result of Quicksort algorithm will be demonstrated as follows.

```
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project3/2$ javac *.java
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project3/2$ java Quicksort
Quicksort:
Before sorting, input array is:
6 8 37 2 48 37 45 18 25 34 48 32 42 24 11 33 23 10 23 22 33 7 6 46 41 49 40 39 23 5 14 28 45 25 18 17 36 39
 28 4 47 43 4 49 2 24 5 11 4 14
After sorting, the array turns to:
2 2 4 4 4 5 5 6 25 6 7 8 10 11 11 14 23 14 23 17 18 18 22 23 24 24 25 28 28 32 33 33 34 45 36 37 37 39 39 4
5 40 41 42 43 46 47 48 48 49 49
```

Figure 7: Quicksort

The result of Mergesort algorithm will be demonstrated as follows.

```
yangnianzu@yangnianzu-virtual-machine:~/桌面/EI338/Project3/2$ java Mergesort
Mergesort:
Before sorting, input array is:
9 0 18 0 37 29 30 22 8 0 44 1 45 25 0 5 38 29 47 8 17 7 43 31 8 27 47 28 33 11 2 41 22 35 27 10 36 31 20 26
45 3 47 4 28 15 29 44 44 5
After sorting, the array turns to:
0 0 0 0 1 2 3 4 5 5 7 8 8 8 9 10 11 15 17 18 20 22 22 25 26 27 27 28 28 29 29 29 30 31 31 33 35 36 37 38 41
43 44 44 44 45 45 47 47 47
Finish sorting!
```

Figure 8: Mergesort

6 Acknowledgement

Thanks for the guidance of teacher Wu Chentao. Without enough knowledge we learned in class, we cann't finish this project easily. Also thanks for assitants' efforts.