EI338 Lab Report Chapter3&4

Bao Xiaoyi 517030910306

November 1, 2019

1 Project 3-1 UNIX Shell

1.1 Environment

- VirtualBox 5.2.18
- Ubuntu 14.04 (64 bit) running on the virtual machine

1.2 Assignment

Design a C program to serve as a shell interface. It gives the user a prompt, then executes each command in the way the user asks. To be more specific, the following functions should be provided:

- Get the command from the user and decompose it.
- Create the child process and executing the command in the child
- Provide history feature
- Add support of input and output redirection
- Allow the parent and child process to communicate via a pipe.

1.3 Decomposition and Analysis

Here, we try to decompose each of the original task into smaller parts, and solve the sub problems one by one.

1. keep asking the user for input command, until the user wants to end the process.

   Solution. *We keep a bool variable to record whether the user wants to continue. If the user doesn't, the variable is set to 0 and the whole program ends.*

2. decompose the command from the command into separate arguments.

   Solution. *First,* `fget` *function helps us get one line of input. After that,* `strtok` *in* `string` *module helps us get one word at a time, and we store the pointer to each word in (char**) args. Finally, special space is open for the input, and we store all the words. Make sure that we keep track of the number of arguments provided.*

3. pass all the arguments and execute with all the parameters.

   Solution. `execvp` *can do the work for us. We just pass the command(the first argument), then pass the pointer to all the arguments(including command). We need to set the last argument as* `NULL` *to indicate the end of parameters. Note that if we direct a (char*) pointer to* `NULL`*, the space should be reallocated after performing* `execvp`*. A child process is created for each command, and & means that the parent process doesn't wait for its child to finish.*

4. allow user to execute the most recent command.

   Solution. *As we know, the most recent command has been stored in special space. Thus, we only need to check whether there is any history command(using counter to indicate the number of commands). If the user asks to execute the most recent one, then we just show the command and execute it again.*

5. output the result of a command to a file.

   Solution. *Managing the redirection involves using* `dup2` *function. We can simply redirect the output of the terminal (*`STDOUT_FILENO`*) to certain file descriptor, then execute the command.*

6. get the input fo command line from a file.

   Solution. *In this part, I read the content of the file, decipher it, then execute. However, a more clever way is to redirect the input of terminal (*`STDIN_FILENO`*) to the given file.*

7. allow the output of one command to serve as input to another.

   Solution. *Actually, it seems like a combination of task 5 and 6. The arguments before "l" is treated as a grandchild, and the arguments after it is treated as a child. The grandchild redirect output of command line as input of a pipe, and execute its command. The child waits for the grandchild to complete. After that, the child redirect output from pipe as input of command line, and execute the command with the arguments provided, and the output from the pipe.*

1.4 Details

In this part, some codes are shown for better illustration.

- If we use the content of file instead of file descriptor, we need to pay special attention to `EOF` at the end of the file

```
/* remove \n and EOF from input file */
while (one_line[i] == '\n' || one_line[i] == EOF)
{
        one_line[i] = '\0';
        i -= 1;
}
```

- Although maybe not necessary, we'd better restore the way of input and output after communication or redirection.

```
/* store file descriptor at the beginning */
int out_fd = dup(STDOUT_FILENO);
int in_fd = dup(STDIN_FILENO);
...
/* restore_in and restore_out indicates whether we need to restore */
if (restore_out){
                dup2(out_fd, STDOUT_FILENO);
                restore_out=FALSE;
}
if (restore_in){
                dup2(in_fd, STDIN_FILENO);
                restore_in=FALSE;
}
```

- After setting the end of argument array pointers as `NULL` for the execution of `execvp`, we need to reallocate space for it.

```
storage[num_of_args] = NULL;
execvp(storage[0],storage);
storage[num_of_args] =  malloc (20 * sizeof(char));      /* restore space */
```

- Free the allocated memory in the end.

```
int i;
for (i=0; i<MAX_LINE/2 + 1; i++){
    free(storage[i]);
}
```

1.5 Result

Here are the results of our experiment:

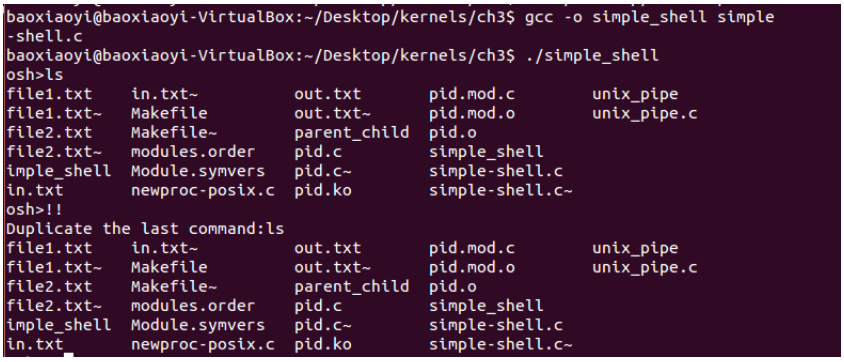

Figure 1: Result1: execution and history

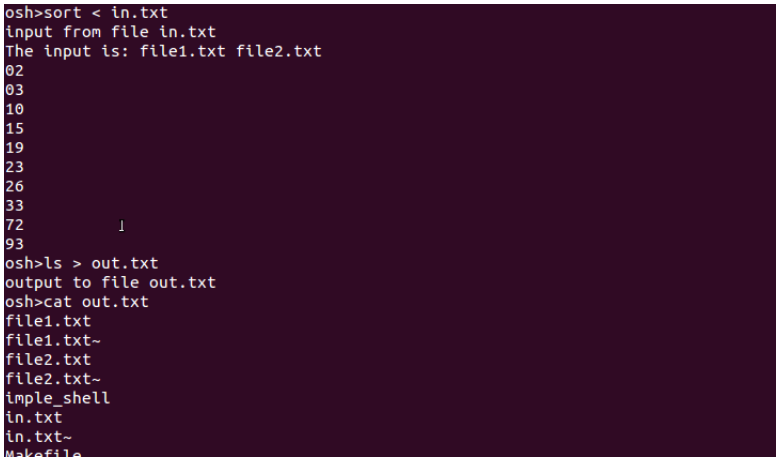

Figure 2: Result2: redirecting input and output

1.6 Future work

There are some points that we may improve in the future:

- In the part of redirecting input, actually we do not need to decode the input. Leave it in the input buffer and `execvp` will do it automatically.
- The interface of some functions can be improved, so that in the `main` function we do not need to do too much modification to input and output.

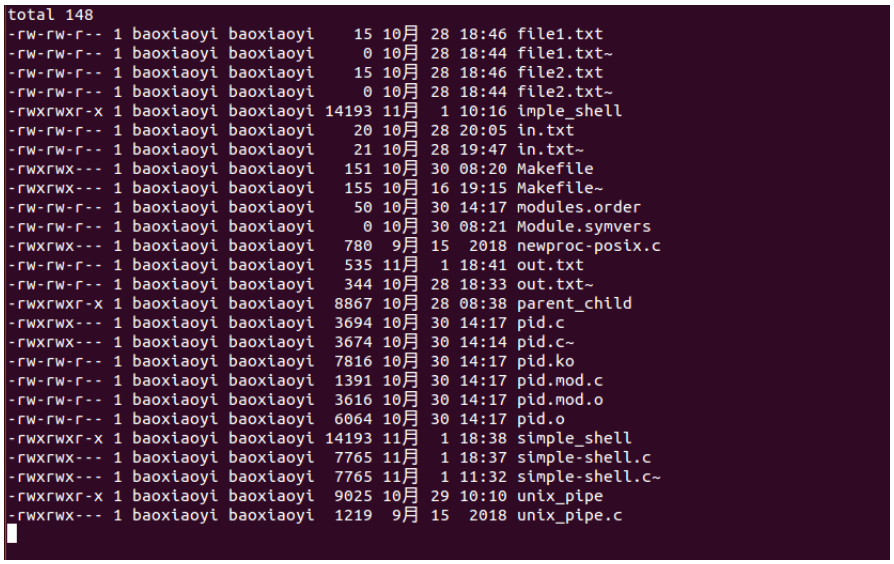

Figure 3: Result3:communication after executing"ls -l — less"

- It takes me a lot of time to implement the management of memory in the current code. However, it does not have a satisfying output. Perhaps it would be better to initialize after each execution, and store the last command in another place.

2 Project 3-2 Linux Kernel Module for Task Information

2.1 Environment

- VirtualBox 5.2.18
- Ubuntu 14.04 (64 bit) running on the virtual machine

2.2 Assignment

Write a Linux kernel module that uses the `/proc` file system for displaying a task's information based on its process. To be more specific, the following functions should be provided:

- allow user to write pid to the `/proc` file system
- read from the `/proc` file system for some information about a task.

2.3 Decomposition and Analysis

Here, we try to decompose each of the original task into smaller parts, and solve the sub problems one by one.

1. copy input from user space to kernel space, and convert it into the type `long`.

   Solution. *Suppose that the user offers us standard input.*
   We modify the function for writing. `kmalloc` *is the counterpart for malloc in user model, and we copy the content from user buffer to the allocated memory. Notice that we need to curtail the non-digit part. After that,* `ksrtol` *assists in converting* `str` *into* `long`*. Pid is stored as static variable in the data part of the process.*

2. consider the case that the user does not provide a valid pid.

   Solution. *If it is not valid, then we inform user of his or her error. Whatever the case is, we need to copy our output from dmesg to terminal.*

2.4 Details

In this part, some codes are shown for better illustration.

- Only the digital part should be saved, thus we need to examine the input.

```
for (i=j-1; i>=0; i--){
        if (k_mem[i] - '0' >= 0 && k_mem[i] - '9' <= 0) {
            j=i;
                break;
        }
}
k_mem[j+1] = '\0';   /* end of string */
```

- We need to obtain task information and check whether the given pid is valid. No matter what the result is, the program should end.

```
tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);
if (tsk==NULL){...}
else{...}   /* invalid case */
```

2.5 Result

Here is the result of our experiment:

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch3$ ps
  PID TTY          TIME CMD
 2883 pts/1    00:00:00 bash
 3268 pts/1    00:00:00 simple_shell
 3585 pts/1    00:00:00 simple_shell
 3586 pts/1    00:00:00 less
 3616 pts/1    00:00:00 ps
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch3$ sudo insmod pid.ko
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch3$ echo "2883" > /proc/pid
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch3$ cat /proc/pid
command = [bash] pid = [2883] state = [1]
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch3$ echo "1234" > /proc/pid
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch3$ cat /proc/pid
pid = [1234], it is not a valid process id.
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch3$
```

Figure 4: Result: read and write `/proc`

2.6 Future work

We may use `sscanf` or regular expression to format the input.

3 Project 4-1 Multi-threaded Sorting Application

3.1 Environment

- VirtualBox 5.2.18
- Ubuntu 14.04 (64 bit) running on the virtual machine

3.2 Assignment

Design a multi-threaded C program that works as follows: it divides a list of integers into two separate smaller lists of equal size. Two separate threads sort each of them, then a third thread merge the result of the first two threads.To be more specific, the following functions should be provided:

- Partition the original array
- Design function to solve sub-problems
- Implement an algorithm to merge
- Managing different threads.

3.3 Decomposition and Analysis

Here, we try to decompose each of the original task into smaller parts, and solve the sub problems one by one.

1. Partition the original array.

   Solution. *The start position and end position are passed as parameters to the sorting algorithm. After calculation middle point, the two are derived.*

2. Design function to solve sub-problems

   Solution. *Quicksort can solve the problem, and it is relatively efficient.*

3. Implement an algorithm to merge

   Solution. *Suppose both of the sub arrays are sorted in ascending order. Compare the leading element of the two add the smaller to the result array, until at least one is empty. After that, concatenate the remaining part to the result array.*

4. Managing different threads.

   Solution. *The two sorting threads may execute concurrently. Yet, the third can not start until the first two have finished.*

3.4 Details

In this part, some codes are shown for better illustration.

- Because of the existence of threads, we need to use `gcc -o executable file name original file name.c -lpthread` when compiling. Pay special attention to `-lpthread`,otherwise it may reports fatal error.
- The input parameter of the running function for threads can only be of type (void*). Consider the fact that we need to pass more than one parameter, we may compact them in a `struct`, transfer it as (void*) and decode it when we need information.

```
typedef struct
{
        int sp1;
        int sp2;
        int ep;
}param_merge;  /* struct for merge thread */
...
param_merge param_m;
param_m.sp1 = sp1;
param_m.sp2 = sp2;
param_m.ep = ep;  /* initialization */
...
pthread_create(&tid_m, &attr, runner_merge, (void*)&param_m); /* encode */
...
void *runner_merge(void *argv)
{
param_merge* param = (param_merge*)argv;  /* decode */
...
}
```

- The shared data should be defined outside the main function.

```
int arr_size = 8;
int arr[10] = {2, 8, 5, 7, 1, 4, 0, 9}; /* this data is shared by the thread
```

3.5 Result

Here is the result of our experiment:

3.6 Future work

Nothing to do.

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch4$ gcc -o thread thrd-posix.c -lpthread
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch4$ ./thread
original array:
2 8 5 7 1 4 0 9
starting point1 = 0, starting point2 = 4, end point = 7, array size = 8
after executing two sorting threads:
2 5 7 8 0 1 4 9
after executing the merging thread:
0 1 2 4 5 7 8 9
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch4$
```

Figure 5: Result: multi-thread in C

4 Project 4-2 Fork-Join Sorting Application

4.1 Environment

- Pycharm 2017.2.4
- Python 3.6.2(with module `threading` and `random`

4.2 Assignment

Implementing the preceding project in another language (we choose python here). This project should include quick_sort and merge_sort. In addition, a recursive version for sorting should be provided. To be exact, it should continue divide-and-conquer until the length of sub-array is below some threshold value.

4.3 Decomposition and Analysis

Here, we try to decompose each of the original task into smaller parts, and solve the sub problems one by one.

1. Design a class that supports thread creation and execution .

   Solution. *Inherit the* `Thread` *class in module* `threading`*. Its function for execution should support different types according to the catagory that it falls in(sort or merge, partition or sort).*

2. Generate test example.

   Solution. *With the help of the* `random` *module, we can shuffle a list that contains all the integers from* 0 *to* $n-1$*. Solidify its seed helps us check our result.*

4.4 Details

In this part, some codes are shown for better illustration. As the two are similar, we just take *merge_sort.py* for example.

- The type of a thread should be provided upon initialization.

```
def __init__(self, name="thread", thread_type="s"):
    threading.Thread.__init__(self)
    self.name = name
    if thread_type == "s":
        self.is_sort_thread = True
    else:
        self.is_sort_thread = False
```

- A threshold is given to check whether it is necessary to further split the array.

```
def merge_sort(arr, lb, ub, threshold=10):
    if lb < ub:
        if ub - lb < threshold:
            temp = arr[lb: ub+1]
            temp.sort()
            for i in range(lb, ub+1):
                arr[i] = temp[i-lb]
        else:
            ...
```

- The execution order requires a little more thinking.

```
thread_s1.start()
thread_s2.start()
thread_s1.join()
thread_s2.join()  # jam the process
...

thread_m.start()
thread_m.join()
```

4.5 Result

Here are the results of our experiment:

4.6 Future work

Nothing to do.

```
original array:
[2, 4, 15, 12, 6, 7, 10, 18, 17, 1, 0, 19, 8, 9, 14, 11, 13, 5, 16, 3]
partition thread starts:
end of partition thread.
[2, 1, 0, 3, 6, 7, 10, 18, 17, 4, 15, 19, 8, 9, 14, 11, 13, 5, 16, 12]
two sorting threads start:
partition thread starts:
end of partition thread.
[0, 1, 2, 3, 6, 7, 10, 4, 8, 9, 11, 5, 12, 18, 14, 15, 13, 19, 16, 17]
two sorting threads start:
end of sorting threads
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
end of sorting threads
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Process finished with exit code 0
```

Figure 6: Result: quick_sort(pivot :last element)

```
original array:
[5, 18, 19, 6, 1, 3, 16, 2, 8, 0, 12, 9, 7, 10, 17, 11, 15, 13, 14, 4]
two sorting threads start:
two sorting threads start:
end of two sorting threads.
[1, 2, 3, 5, 6, 8, 16, 18, 19, 0, 7, 9, 10, 12, 4, 11, 13, 14, 15, 17]
merging thread start:
end of merging threads
[1, 2, 3, 5, 6, 8, 16, 18, 19, 0, 4, 7, 9, 10, 11, 12, 13, 14, 15, 17]
end of two sorting threads.
[1, 2, 3, 5, 6, 8, 16, 18, 19, 0, 4, 7, 9, 10, 11, 12, 13, 14, 15, 17]
merging thread start:
end of merging threads
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Process finished with exit code 0
```

Figure 7: Result: merge_sort