

CSC3150 Operating Systems

Assignment1 Report

Name: Zekai Chen

Student ID: 120090539

Date: 8/10/2022

The Chinese University of Hong Kong, Shenzhen

In the Assignment 1, I finished all the three tasks. They are shown below.

- Program1: Create process from user mode
- Program2: Create process from kernel mode
- Bonus: pstree command

1. Program1: Create process from user mode

1.1 Design

In Program1, I use the output of the `fork()` function to divide the program into three parts. They are the error part, the child process part, and the parent process part. I use the `pid` parameter to store the output.

When the output is -1, we meet an error when using the `fork()` function.

When the output is 0, it is the child process part. Following the tutorial guide, I used the `execve()` function to execute test program. The new program will replace the child process. I also check whether it is replaced successfully by printing the error message in the following calling program.

When the output is 1, it means it is the parent process part. The parent and child process runs concurrently after forking. I use the `waitpid()` function to wait and receive output from child to parent. For the third argument in the `waitpid()` function, we need to choose `WUNTRACED` to cover the all fifteen cases. `WUNTRACED` reports on stopped child processes as well as terminated ones. If we choose 0, it will be some error when we execute the `abort.c` program. The parent process will keep waiting for the child process to terminate. At last, I check the child process' termination status by `WIFEXITED()`, `WIFSIGNALED()`, and `WIFSTOPPED()` and output the termination status.

Finally, I modified my output to follow the requirement in the demo.

For more details, please see the comments in the code.

1.2 Development Environment Set Up

For Program1, we do not need to compile the kernel since it is in user mode. I just ran the C program in Visual Studio Code. We should keep GCC Version above or equal to 4.9. Also, the Linux Distribution could be Ubuntu 20.04 or related version. Since the makefile is given, we need to type “make” in the terminal. Then all the test programs and the `program1.c` will be compiled. We could also type “`gcc -o Filename Filename.c`” to compile the specific program. Then we type “`./program1 ./TestFilename`”, and we see the output similar to the demo.

1.3 Screenshot

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./abort
Process start to fork
I'm the Parent Process, my pid = 2233
I'm the Child Process, my pid = 2234
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGABRT program

Parent process receives SIGCHLD signal
child process get SIGABRT signal
```

Figure 1.1 Output of abort.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./alarm
Process start to fork
I'm the Parent Process, my pid = 2321
I'm the Child Process, my pid = 2322
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGALRM program

Parent process receives SIGCHLD signal
child process get SIGALRM signal
```

Figure 1.2 Output of alarm.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./bus
Process start to fork
I'm the Parent Process, my pid = 2375
I'm the Child Process, my pid = 2376
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGBUS program

Parent process receives SIGCHLD signal
child process get SIGBUS signal
```

Figure 1.3 Output of bus.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./floating
Process start to fork
I'm the Child Process, my pid = 2442
Child process start to execute test program:
I'm the Parent Process, my pid = 2441
-----CHILD PROCESS START-----
This is the SIGFPE program

Parent process receives SIGCHLD signal
child process get SIGFPE signal
```

Figure 1.4 Output of floating.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./hangup
Process start to fork
I'm the Child Process, my pid = 2529
Child process start to execute test program:
I'm the Parent Process, my pid = 2528
-----CHILD PROCESS START-----
This is the SIGHUP program

Parent process receives SIGCHLD signal
child process get SIGHUP signal
```

Figure 1.5 Output of hangup.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./illegal_instr
Process start to fork
I'm the Parent Process, my pid = 2650
I'm the Child Process, my pid = 2651
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGILL program

Parent process receives SIGCHLD signal
child process get SIGILL signal
```

Figure 1.6 Output of illegal_instr.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./interrupt
Process start to fork
I'm the Parent Process, my pid = 2713
I'm the Child Process, my pid = 2714
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGINT program

Parent process receives SIGCHLD signal
child process get SIGINT signal
```

Figure 1.7 Output of interrupt.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./kill
Process start to fork
I'm the Parent Process, my pid = 2763
I'm the Child Process, my pid = 2764
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGKILL program

Parent process receives SIGCHLD signal
child process get SIGKILL signal
```

Figure 1.8 Output of kill.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./normal
Process start to fork
I'm the Parent Process, my pid = 2885
I'm the Child Process, my pid = 2886
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the normal program

-----CHILD PROCESS END-----
Parent process receives SIGCHLD signal
Normal termination with EXIT STATUS = 0
```

Figure 1.9 Output of normal.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./pipe
Process start to fork
I'm the Child Process, my pid = 2936
Child process start to execute test program:
I'm the Parent Process, my pid = 2935
-----CHILD PROCESS START-----
This is the SIGPIPE program

Parent process receives SIGCHLD signal
child process get SIGPIPE signal
```

Figure 1.10 Output of pipe.c


```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./quit
Process start to fork
I'm the Parent Process, my pid = 2985
I'm the Child Process, my pid = 2986
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGQUIT program

Parent process receives SIGCHLD signal
child process get SIGQUIT signal
```

Figure 1.11 Output of quit.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./segment_fault
Process start to fork
I'm the Parent Process, my pid = 3060
I'm the Child Process, my pid = 3061
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGSEGV program

Parent process receives SIGCHLD signal
child process get SIGSEGV signal
```

Figure 1.12 Output of segment_fault.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./stop
Process start to fork
I'm the Parent Process, my pid = 3111
I'm the Child Process, my pid = 3112
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGSTOP program

Parent process receives SIGCHLD signal
child process get SIGSTOP signal
```

Figure 1.13 Output of stop.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./terminate
Process start to fork
I'm the Parent Process, my pid = 3161
I'm the Child Process, my pid = 3162
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGTERM program

Parent process receives SIGCHLD signal
child process get SIGTERM signal
```

Figure 1.14 Output of terminate.c

```
● vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./trap
Process start to fork
I'm the Parent Process, my pid = 3208
I'm the Child Process, my pid = 3209
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGTRAP program

Parent process receives SIGCHLD signal
child process get SIGTRAP signal
```

Figure 1.15 Output of trap.c

The figure above is the output of the Program1.

1.4 Learned from the task

I know the flow of creating a process from user mode. I deeply understand how to use fork() function, execve() function, and waitpid() function.

Everything begins from the fork(). The child process created by fork is an exact clone of the original parent process, except that it has its own process ID. Both parent and child processes start execution from the following statement after the fork call. When the child process executes a file, the file will overlay the process if the execution is successful. The child process use raise() function to raise the signal if needed. The process executes the last statement and asks the operating system to delete it with exit() function.

The parent process uses the wait() or waitpid() function to wait and receive output from the child. The child's termination process is a zombie if no parent is waiting. If the parent is terminated, the child processes are orphans, which the initial process will adopt.

After the parent process gets the information, I know how to evaluate the child process's status and the return value of the status argument with the specific function. This is basically what I learned in Program1.

2. Program2: Create process from kernel mode

2.1 Design

In Program2, I design the program according to the creating process inside the kernel mode.

At first, I export the function in the kernel source code, which I may need in the following program to my module. Then I compile and install the new kernel. As a result, I could use “extern” to clarify them in my program.

After everything is ready, I create a kernel thread to run the `my_fork()` function by using the `kthread_create()` function. The thread will not start running immediately. It will start to execute when returned `task_struct` is passed to `wake_up_process()`. Then I returned the value and activated the process.

After I read the source code, I find the `kernel_thread()` is similar to a wrapper version of `kernel_clone()`, and then I choose the `kernel_thread()` function in the `my_fork` execution. The process I forked will do the `my_execute()` function, which I wrote to do the execution part. The third argument in the `kernel_thread()` should be `SIGCHLD` which is related to the integer 17.

In the `my_execute()` function, I use the `getname_kernel()` function and absolute path to get the filename. After that, I use the `do_execve()` function to execute the corresponding file. If the return value of `do_execve()` function is 0, then the execution is successful.

I write a `my_wait()` function for the parent process to wait and receive the signals that the child process may raise. In the `my_wait()` function, I call the `do_wait()` function of the source code. In the `do_wait()` function, there is a struct called `wait_ops`. The struct could not be extended by the “extern” command. I copied the code from the source code and noted the source of the code. In the `wait_ops` struct, there is an integer argument called `wo_stat`. Its type differs from the argument in previous versions of Linux, which is the place to pay attention. Also, we need to set the `wo.wo_flags` as `(WEXITED | WUNTRACED)` in order to take `stop.c` file into consideration. The raised signal could be obtained by bit operation “`wo.wo_stat & 0x7f`” when `WIFSIGNALED(STATUS)` is true or process terminates normally, equivalent to clearing the highest bit without affecting the other lower 7 bits. When the situation is `WIFSTOPPED(STATUS)`, the raised signal should be `WEXITSTATUS(status)`.

In the end, I wrote a function called `LinuxSingal ()` which acted as a Linux signal dictionary to output the signal’s name to meet the requirement of the demo. When all tasks are over, I use the `put_pid()` function to decrease the count and free memory.

This is the whole process of Program2 that I designed.

For more details, please see the comments in the code.

2.2 Development Environment Set Up

For Program2, we need to properly set up the virtual machine and vagrant and set up the ssh to connect to the VM. Then we should set up Visual Studio Code and connect

VM with Visual Studio Code. This way, whenever we want to use a VM, we can open a remote terminal in Visual Studio Code.

Also, we need to compile the kernel. First, we need to download the source code from the corresponding website. The Linux kernel version should keep above or equal to 5.10.x. Next, we need to install dependency and development tools. Then, we unzip the source file to the specific folder.

After that, we should copy the config file from /boot to the kernel source dictionary. The next task is to log in to the root account and go to the kernel source directory to clean the previous setting and start the configuration. After everything above is done, we need to build kernel Images and modules, which will take hours. We should never forget to leave enough memory for this part. Finally, we need to install kernel modules and kernel and then reboot it.

We can go to the folder where we unzip the kernel file every time we want to program and type “vagrant up” to log in and “vagrant halt” to log out when we finish the task. Remember to log out when you do other work. It will consume a lot of computer performance.

When we want to use the function in the source code, we need to export it by `EXPORT_SYMBOL()` function after the code in the source code. Then we need to log in to the root account and type “`chmod 777 Filename.c`”. We should recompile the kernel from the “`make bzImage -j$(nproc)`” command. This time, it will be faster than the first time we compile.

We must compile the test. c by typing “`gcc -o test.c`”. Then we could use the command “`make`” to compile the program2.c or “`make clean`” to delete all the built files and leave the original c file and the makefile.

Next, we need to log in to the root account to insert and remove the kernel module. We could also use the “`lsmod`” command to list the module you insert and use the “`dmesg`” to see the output like the demo.

2.3 Screenshot

```
[ 1302.219789] [program2] : module_init {Zekai Chen} {120090539}
[ 1302.219899] [program2] : module_init create kthread start
[ 1302.220674] [program2] : module_init kthread start
[ 1302.220963] [program2] : The child process has pid = 3724
[ 1302.221042] [program2] : This is the parent process, pid = 3723
[ 1302.221044] [program2] : child process
[ 1302.385769] [program2] : get SIGABRT signal
[ 1302.385772] [program2] : child process terminated
[ 1302.385773] [program2] : The return signal is 6
[ 1308.974445] [program2] : Module_exit
```

Figure 2.1 Output of abort.c

```
[ 1650.683433] [program2] : module_init {Zekai Chen} {120090539}
[ 1650.683435] [program2] : module_init create kthread start
[ 1650.683598] [program2] : module_init kthread start
[ 1650.683764] [program2] : The child process has pid = 4846
[ 1650.683765] [program2] : This is the parent process, pid = 4845
[ 1650.683765] [program2] : child process
[ 1652.685413] [program2] : get SIGALRM signal
[ 1652.685418] [program2] : child process terminated
[ 1652.685422] [program2] : The return signal is 14
[ 1657.965542] [program2] : Module_exit
```

Figure 2.2 Output of alarm.c

```
[ 1754.185030] [program2] : module_init {Zekai Chen} {120090539}
[ 1754.185032] [program2] : module_init create kthread start
[ 1754.187822] [program2] : module_init kthread start
[ 1754.187856] [program2] : The child process has pid = 5364
[ 1754.187857] [program2] : This is the parent process, pid = 5363
[ 1754.187858] [program2] : child process
[ 1754.436357] [program2] : get SIGBUS signal
[ 1754.436359] [program2] : child process terminated
[ 1754.436360] [program2] : The return signal is 7
[ 1760.015160] [program2] : Module_exit
```

Figure 2.3 Output of bus.c

```
[ 1846.524884] [program2] : module_init {Zekai Chen} {120090539}
[ 1846.524886] [program2] : module_init create kthread start
[ 1846.525198] [program2] : module_init kthread start
[ 1846.525681] [program2] : The child process has pid = 5834
[ 1846.525683] [program2] : This is the parent process, pid = 5833
[ 1846.525684] [program2] : child process
[ 1846.793358] [program2] : get SIGFPE signal
[ 1846.793364] [program2] : child process terminated
[ 1846.793367] [program2] : The return signal is 8
[ 1852.765779] [program2] : Module_exit
```

Figure 2.4 Output of floating.c

```
[ 1924.686222] [program2] : module_init {Zekai Chen} {120090539}
[ 1924.686224] [program2] : module_init create kthread start
[ 1924.686436] [program2] : module_init kthread start
[ 1924.686683] [program2] : The child process has pid = 6275
[ 1924.686685] [program2] : This is the parent process, pid = 6274
[ 1924.686685] [program2] : child process
[ 1924.688444] [program2] : get SIGHUP signal
[ 1924.688473] [program2] : child process terminated
[ 1924.688474] [program2] : The return signal is 1
[ 1929.998190] [program2] : Module_exit
```


Figure 2.5 Output of hangup.c

```
[ 2019.729264] [program2] : module_init {Zekai Chen} {120090539}
[ 2019.729267] [program2] : module_init create kthread start
[ 2019.729542] [program2] : module_init kthread start
[ 2019.729762] [program2] : The child process has pid = 6775
[ 2019.729767] [program2] : This is the parent process, pid = 6774
[ 2019.729769] [program2] : child process
[ 2020.068986] [program2] : get SIGILL signal
[ 2020.068991] [program2] : child process terminated
[ 2020.068994] [program2] : The return signal is 4
[ 2027.682393] [program2] : Module_exit
```

Figure 2.6 Output of illegal_instr.c

```
[ 2115.522713] [program2] : module_init {Zekai Chen} {120090539}
[ 2115.522716] [program2] : module_init create kthread start
[ 2115.522925] [program2] : module_init kthread start
[ 2115.523004] [program2] : The child process has pid = 7234
[ 2115.523005] [program2] : This is the parent process, pid = 7233
[ 2115.523005] [program2] : child process
[ 2115.523352] [program2] : get SIGINT signal
[ 2115.523353] [program2] : child process terminated
[ 2115.523354] [program2] : The return signal is 2
[ 2121.085922] [program2] : Module_exit
```

Figure 2.7 Output of interrupt.c

```
[ 2224.068050] [program2] : module_init {Zekai Chen} {120090539}
[ 2224.068052] [program2] : module_init create kthread start
[ 2224.068201] [program2] : module_init kthread start
[ 2224.068312] [program2] : The child process has pid = 7760
[ 2224.068314] [program2] : This is the parent process, pid = 7759
[ 2224.068314] [program2] : child process
[ 2224.069465] [program2] : get SIGKILL signal
[ 2224.069468] [program2] : child process terminated
[ 2224.069468] [program2] : The return signal is 9
[ 2230.217741] [program2] : Module_exit
```

Figure 2.8 Output of kill.c

```
[ 2294.873658] [program2] : module_init {Zekai Chen} {120090539}
[ 2294.873660] [program2] : module_init create kthread start
[ 2294.873929] [program2] : module_init kthread start
[ 2294.874008] [program2] : The child process has pid = 8250
[ 2294.874009] [program2] : This is the parent process, pid = 8249
[ 2294.874010] [program2] : child process
[ 2294.874363] [program2] : normal termination
[ 2294.874364] [program2] : child process terminated
[ 2294.874365] [program2] : The return signal is 0
[ 2299.616481] [program2] : Module_exit
```

Figure 2.9 Output of normal.c

```
[ 2419.993353] [program2] : module_init {Zekai Chen} {120090539}
[ 2419.993355] [program2] : module_init create kthread start
[ 2419.993538] [program2] : module_init kthread start
[ 2419.993766] [program2] : The child process has pid = 8669
[ 2419.993767] [program2] : This is the parent process, pid = 8668
[ 2419.993768] [program2] : child process
[ 2419.994714] [program2] : get SIGPIPE signal
[ 2419.994716] [program2] : child process terminated
[ 2419.994716] [program2] : The return signal is 13
[ 2427.477441] [program2] : Module_exit
```

Figure 2.10 Output of pipe.c

```
[ 2497.842457] [program2] : module_init {Zekai Chen} {120090539}
[ 2497.842460] [program2] : module_init create kthread start
[ 2497.842683] [program2] : module_init kthread start
[ 2497.842926] [program2] : The child process has pid = 9148
[ 2497.842928] [program2] : This is the parent process, pid = 9147
[ 2497.842929] [program2] : child process
[ 2498.131320] [program2] : get SIGQUIT signal
[ 2498.131326] [program2] : child process terminated
[ 2498.131329] [program2] : The return signal is 3
[ 2503.171006] [program2] : Module_exit
```

Figure 2.11 Output of quit.c

```
[ 2557.816801] [program2] : module_init {Zekai Chen} {120090539}
[ 2557.816804] [program2] : module_init create kthread start
[ 2557.816964] [program2] : module_init kthread start
[ 2557.817033] [program2] : The child process has pid = 9612
[ 2557.817035] [program2] : This is the parent process, pid = 9611
[ 2557.817035] [program2] : child process
[ 2558.079578] [program2] : get SIGSEGV signal
[ 2558.079580] [program2] : child process terminated
[ 2558.079581] [program2] : The return signal is 11
[ 2563.295675] [program2] : Module_exit
```

Figure 2.12 Output of segment_fault.c

```
[ 5425.206468] [program2] : module_init {Zekai Chen} {120090539}
[ 5425.206470] [program2] : module_init create kthread start
[ 5425.206709] [program2] : module_init kthread start
[ 5425.206857] [program2] : The child process has pid = 12976
[ 5425.206858] [program2] : This is the parent process, pid = 12975
[ 5425.206859] [program2] : child process
[ 5425.241689] [program2] : get SIGSTOP signal
[ 5425.241691] [program2] : child process terminated
[ 5425.241693] [program2] : The return signal is 19
[ 5430.607572] [program2] : Module_exit
```


Figure 2.13 Output of stop.c

```
[ 5606.656400] [program2] : module_init {Zekai Chen} {120090539}
[ 5606.656402] [program2] : module_init create kthread start
[ 5606.656729] [program2] : module_init kthread start
[ 5606.691171] [program2] : The child process has pid = 14018
[ 5606.691174] [program2] : This is the parent process, pid = 14015
[ 5606.691175] [program2] : child process
[ 5606.693608] [program2] : get SIGTERM signal
[ 5606.693610] [program2] : child process terminated
[ 5606.693611] [program2] : The return signal is 15
[ 5612.146506] [program2] : Module_exit
```

Figure 2.14 Output of terminate.c

```
[ 5677.556720] [program2] : module_init {Zekai Chen} {120090539}
[ 5677.556723] [program2] : module_init create kthread start
[ 5677.556939] [program2] : module_init kthread start
[ 5677.591468] [program2] : The child process has pid = 14540
[ 5677.591469] [program2] : This is the parent process, pid = 14539
[ 5677.591470] [program2] : child process
[ 5678.005284] [program2] : get SIGTRAP signal
[ 5678.005287] [program2] : child process terminated
[ 5678.005288] [program2] : The return signal is 5
[ 5682.494112] [program2] : Module_exit
```

Figure 2.15 Output of trap.c

2.4 Learned from the task

I know the flow of creating a process from kernel mode. Also, I learned to read and use the kernel's source code, which significantly improved my engineering ability. The code in the PPT may not always be valid. We must learn to draw inferences from one case and find the answer in the source code. In Program2, I meet tons of problems. However, most of them could be solved by searching on the Google, which improves my self-study ability.

Before programming this project, I was unfamiliar with Linux instructions and C language. To finish this project, I have learned much-related knowledge. I may do better next time I meet a similar problem.

In the end, I learned how to modified a file to clang-format. According to TA, writing code in format is very important in industrial use. The learning process was a bit tortuous, but it also added a skill to me.

3. Bonus: pstree command

3.1 Design

I would like first to introduce the high-level idea of the program. The first thing I did was read the status of each folder. We can find the status is the filename in the "proc." Then I read all process information and store it in the HARR[], the helper node array(Double linked list). Then I traverse the HARR and stuff the information into the tree. The root node of the tree is pROOT. At last, I print the tree by traversing the tree using the recursion tool.

In the bonus folder, I wrote six files. One is a makefile which is needed. The Qsort.c and Qsort.h is to achieve the function of the quick sort known worldwide. The DuLinkedList.c and DuLinkedList.h is to achieve the function of creating a double linked list. This is similar to a template. The head file has a three-layers nested structures (DuLinkedList.h). I did not want to be so troublesome, but if I do not add it, it will report an error, so I add it. I don't currently know the reason.

There is no doubt that the doubly linked list is very suitable for this project. It can be used by every parent node that has child nodes to store child nodes. The relationship between processes could be easily simulated by it. To sort the list, I first convert the linked list to an array. Then it is the place to use the quick sort. After the quick sort, I convert the array into a linked list again.

When parsing the two functions of the status file, I use the techniques from the source code of tinyxml. I learned the ability to read source code in Program2. It comes in handy now. I think the technique is compelling. I use SkipWhiteSpace() to skip consecutive spaces. Also, I use ReadElem() to read a non-whitespace string. We take the first line as an example, and the first element is Name, which we do not need. We only need to get the second one. We go to the next line as soon as we get it. Also, as I comment in the code, our goal is in lines 1,7,9, and 10, which stands for name, ppid, pid, and gid. Username is queried by function based on uidToName().

I also use some tips in the other places. The length of the HARR is 100000. I set the MAX_PID to reduce the number of traversals. The flag_active shows whether the node of HARR has been activated. The status activated it since the index of HARR is the pid. If it is not activated, the corresponding file has not been read. When traversing, we should skip them.

The option I achieved is "-n", "-V", "-u", "-g", and "-p". Some options are obvious after writing the tree. However, I did not think much about it at first. Fortunately, "-V" is, without a doubt, the simplest of them all. We need to print the version. For "-n", it is to achieve sorting all processes in numerically ascending order. For "-p", it is to display pid after the name. For "-u", we need to display the user the process belongs to after the name. For "-g", we need to display the process group id after the name. They are all stored in the "data" part.

I find it troublesome to draw special symbols. With the permission of the USTF, I use spaces and indents for the printing the tree.

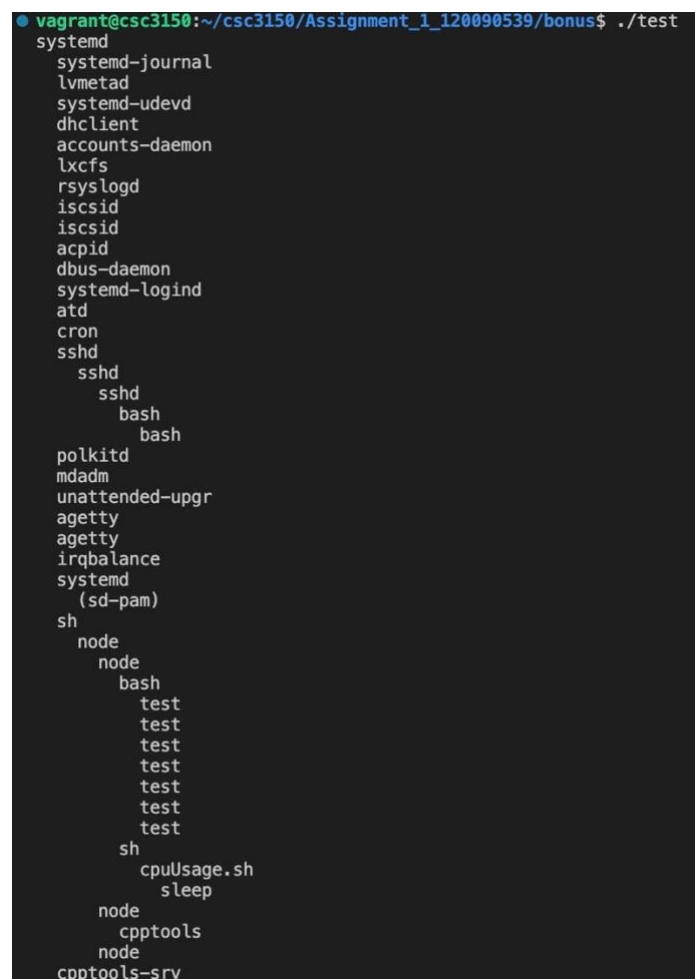
For the makefile, I read lots of related information to achieve it. I wrote my first makefile by imitating and learning previous makefiles from other computer disciplines.

There is too much to cover when talking about the ideas when designing. For more details, please see the comments in the code. I also write enough comments to explain quick sort and double linked list, though they are familiar.

3.2 Development Environment Set Up

For the bonus problem, no special requirement is needed to set up. I just ran the C program in Visual Studio Code as we did before. A makefile is provided to make the compile process easier. Teacher just needs to type “make”, a file called test will be made. Then he just need to type “./test” or “./test -V” and so on. The option I achieve is “-n”, “-V”, “-u”, “-g”, and “-p”. Also, “make clean” is provided to clean the file we make.

3.3 Screenshot



```
vagrant@csc3150:~/csc3150/Assignment_1_120090539/bonus$ ./test
systemd
systemd-journal
lvmetad
systemd-udevd
dhclient
accounts-daemon
lxcfs
rsyslogd
iscsid
iscsid
acpid
dbus-daemon
systemd-logind
atd
cron
sshd
sshd
sshd
bash
bash
bash
polkitd
mdadm
unattended-upgr
agetty
agetty
irqbalance
systemd
(sd-pam)
sh
node
node
bash
test
test
test
test
test
test
test
sh
cpuUsage.sh
sleep
node
cpptools
node
cpptools-srv
```

Figure 3.1 Output of command “./test”

```
● vagrant@csc3150:~/csc3150/Assignment_1_120090539/bonus$ ./test -u
systemd(user=root)
systemd-journal(user=root)
lvmtools(user=root)
systemd-udev(user=root)
dhclient(user=root)
accounts-daemon(user=root)
lxcfs(user=root)
rsyslogd(user=syslog)
iscsid(user=root)
iscsid(user=root)
acpid(user=root)
dbus-daemon(user=messagebus)
systemd-logind(user=root)
atd(user=root)
cron(user=root)
sshd(user=root)
sshd(user=root)
sshd(user=vagrant)
bash(user=vagrant)
bash(user=vagrant)
polkitd(user=root)
mdadm(user=root)
unattended-upgr(user=root)
agetty(user=root)
agetty(user=root)
irqbalance(user=root)
systemd(user=vagrant)
(sd-pam)(user=vagrant)
sh(user=vagrant)
node(user=vagrant)
node(user=vagrant)
bash(user=vagrant)
test(user=vagrant)
test(user=vagrant)
test(user=vagrant)
test(user=vagrant)
test(user=vagrant)
test(user=vagrant)
test(user=vagrant)
node(user=vagrant)
cpptools(user=vagrant)
node(user=vagrant)
cpptools-srv(user=vagrant)
```

Figure 3.2 Output of command “./test -u”

```
● vagrant@csc3150:~/csc3150/Assignment_1_120090539/bonus$ ./test -g
systemd(gid=0)
systemd-journal(gid=0)
lvmtools(gid=0)
systemd-udev(gid=0)
dhclient(gid=0)
accounts-daemon(gid=0)
lxcfs(gid=0)
rsyslogd(gid=108)
iscsid(gid=0)
iscsid(gid=0)
acpid(gid=0)
dbus-daemon(gid=111)
systemd-logind(gid=0)
atd(gid=0)
cron(gid=0)
sshd(gid=0)
sshd(gid=0)
sshd(gid=1000)
bash(gid=1000)
bash(gid=1000)
polkitd(gid=0)
mdadm(gid=0)
unattended-upgr(gid=0)
agetty(gid=0)
agetty(gid=0)
irqbalance(gid=0)
systemd(gid=1000)
(sd-pam)(gid=1000)
sh(gid=1000)
node(gid=1000)
node(gid=1000)
bash(gid=1000)
test(gid=1000)
test(gid=1000)
test(gid=1000)
test(gid=1000)
test(gid=1000)
test(gid=1000)
test(gid=1000)
test(gid=1000)
node(gid=1000)
cpptools(gid=1000)
node(gid=1000)
cpptools-srv(gid=1000)
```

Figure 3.3 Output of command “./test -g”

```
gnment_1_120090539/bonus$ ./test -p
systemd(pid=1)
  systemd-journal(pid=402)
  lvmtools(pid=408)
  systemd-udevd(pid=430)
  dhclient(pid=871)
  accounts-daemon(pid=994)
  lxcfs(pid=995)
  rsyslogd(pid=998)
  iscsid(pid=1000)
  iscsid(pid=1001)
  acpid(pid=1003)
  dbus-daemon(pid=1005)
  systemd-logind(pid=1023)
  atd(pid=1028)
  cron(pid=1031)
  sshd(pid=1035)
    sshd(pid=7401)
      sshd(pid=7436)
        bash(pid=7437)
          bash(pid=7441)
polkitd(pid=1042)
mdadm(pid=1047)
unattended-upgr(pid=1049)
agetty(pid=1109)
agetty(pid=1111)
irqbalance(pid=1114)
systemd(pid=1452)
  (sd-pam)(pid=1453)
sh(pid=1539)
  node(pid=1549)
    node(pid=1616)
      bash(pid=5969)
        test(pid=6219)
        test(pid=6314)
        test(pid=6380)
        test(pid=6484)
        test(pid=6542)
        test(pid=6790)
        test(pid=11398)
      node(pid=5917)
        cpptools(pid=6629)
      node(pid=5928)
        cpptools-srv(pid=10017)
```

Figure 3.4 Output of command “./test -p”

```
● vagrant@csc3150:~/csc3150/Assignment_1_120090539/bonus$ ./test -V
pstree (PSmisc) 22.21
Copyright (C) 1993-2009 Werner Almesberger and Craig Small

PSmisc comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it under
the terms of the GNU General Public License.
For more information about these matters, see the files named COPYING.
```

Figure 3.5 Output of command “./test -V”

```
● vagrant@csc3150:~/csc3150/Assignment_1_120090539/bonus$ ./test -n
systemd
  systemd-journal
  lvm2metad
  systemd-udevd
  dhclient
  accounts-daemon
  lxcfs
  rsyslogd
  iscsid
  iscsid
  acpid
  dbus-daemon
  systemd-logind
  atd
  cron
  sshd
    sshd
      sshd
        bash
          bash
            polkitd
            mdadm
            unattended-upgr
            agetty
            agetty
            irqbalance
            systemd
              (sd-pam)
            sh
              node
                node
                  bash
                    test
                    test
                    test
                    test
                    test
                    test
                    test
                  node
                    cpptools
                  node
                    cpptools-srv
                    cpptools-srv
```

Figure 3.6 Output of command “./test -n”

```
● vagrant@csc3150:~/csc3150/Assignment_1_120090539/bonus$ make
gcc -c DuLinkedList.c -o DuLinkedList.o
gcc -c Qsort.c -o Qsort.o
gcc -c pstree.c -o pstree.o
pstree.c: In function '_PrintTree':
pstree.c:271:7: warning: implicit declaration of function 'Qsort' [-Wimplicit-function-declaration]
    Qsort(&(proot->data.childsList), proot->data.childsList.cursize);
    ^
gcc DuLinkedList.o Qsort.o pstree.o -o test
```

Figure 3.7 Output of command “make”

```
● vagrant@csc3150:~/csc3150/Assignment_1_120090539/bonus$ make clean
rm -rf DuLinkedList.o Qsort.o pstree.o
rm -rf test
```

Figure 3.8 Output of command “make clean”

3.4 Learned from the task

This is the first time I tried to write a simple Linux command. I also read tons of information related to pstree before I wrote. Compared to Program2, the bonus problem is more maneuverability. It will not be stuck as long as the Program2 problem does when we meet the error. However, on the contrary, the Bonus problem has an enormous amount of code to implement.

For the big project, we always need to break it into a small functional snippet and achieve it one by one. At the same time, we need to start from the simplest case, take it as the backbone, and derive various additional functions or extreme cases. Finally, we must assemble them and modify the possible bug.

In the end, I also tried to write my makefile for the first time, which makes more manageable for me to compile.

4. Conclusion

In this assignment, I finished projects about creating processes from user and kernel modes. Also, I tried to program a simple pstree command and write the makefile by myself. I have devoted an incalculable amount of time to this assignment. As a reward, I feel my abilities have grown in many ways.

I am very grateful for TA's and USTF's help in my project. I am also very grateful to the classmates who shared their questions on the piazza. Their questions and the teacher's answers have greatly helped my assignment.

I am looking forward to the following few assignments. This should be a challenging but rewarding journey.

References:

1. CSC3150 Assignment 1 Homework Requirements
2. Tutorial2's PPT
3. <https://en.wikipedia.org/wiki/Procfs> The website in the hint about pstree command.