香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Assignment 1 Report

CSC3150
Operating System

Name: *Derong Jin*
Student ID: *120090562*

Date: October 10, 2022

1 Overview

The assignment is divided into three tasks, which are written in three program files. The program in Task 1 calls `fork`, `wait`, etc. to create a child process in user space and execute the specified executable, while the parent process receives and analyzes the termination status of the child process after it finishes. The idea of task 2 is similar to that of task 1, except that task 2 requires writing a kernel module and calling the kernel functions. Since some APIs that have not been exported are to be used, it is necessary to modify and recompile parts of the linux kernel code before compiling and running the Task 2 code (see Section 2.2 for details). The program for the bonus task implements a simple `pstree` command, which reads and extracts information from the `/proc` directory, builds a tree of messages, organizes and merges the duplicate nodes according to the passed parameters, and finally uses recursion to output the message.

All programs in this assignment were implemented in `C`. And the source codes were formatted according to the standard `.clang_format` file in l*inux kernel 5.10.146 source code* via the `clang-format` command before submission.

2 Environment Setup

All the 3 programs in this assignment are developed and tested under the same environment as recommended. The information about the developing environment is listed below:

OS & Software name	version
Linux Distribution	Ubuntu 20.04 focal
Linux Kernel	x86_64 Linux 5.10.146
GCC	9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)
MAKE	GNU Make 4.2.1

Table 1: This list shows the basic information about the developing environment

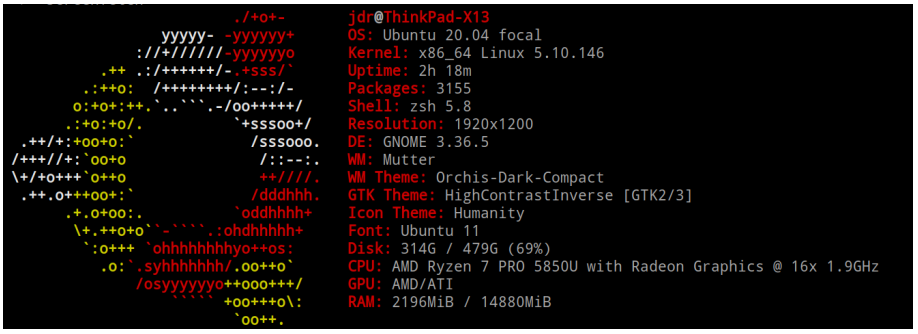


Figure 1: The linux OS information shown by screenfetch command

2.1 Set Up Linux OS

Due to personal reasons, I choose to install linux OS on my machine.

The Setup process consists of following steps:

1. Download the `.iso` file of appropriate linux distribution (in this assignment, I choose Ubuntu 20.04.4 LTS)
2. Create bootable USB flash drivers: I choose to use rufus to create the boot USB (rufus at `https://rufus.ie/en/#`)
3. Reboot the system from the USB, a installing GUI will be displayed.
4. Follow the guideline to set up Ubuntu system on your machine.

2.2 Modify, Compile, and Install Linux Kernel

The kernel in Linux `.iso` file is pre-compiled binary files. In Task II of this assignment, several kernel functions that are initially not accessible from externel programs are used. Hence, modifications on kernel source are needed to make these APIs accessible.

The modifications consists of several steps:

1. Download the source code of Linux Kernel (this assignment use Linux 5.10.146)
2. Install utility tools to compile the kernel (`libncurses-dev gawk flex bison openssl libssl-dev dkms libelf-devlibudev-dev libpci-dev libiberty-dev autoconf llvm dwarves`)
3. Open the source code and export the symbol:

Figure 2: This figure shows the example to export the function do_execve, which is originally unaccessible, after this modification, do_execve function can be accessed by external program with “GPL” licience

Function Name	File Path
kernel_thread	linux-5.10.146/kernel/fork.c
kernel_clone	linux-5.10.146/kernel/fork.c
do_wait	linux-5.10.146/kernel/exit.c
do_execve	linux-5.10.146/fs/exec.c
getname_kernel	linux-5.10.146/fs/namei.c

Table 2: The table exhibits all the symbols that needs exporting in this assignment

4. Clean previous settings and start configuration
   - `$make mrproper`
   - `$make clean`
   - `menuconfig`, save the configuration and exit
5. In Ubuntu OS on physical machine, the following configuration is somehow needed:
   - `$scripts/config --disable SYSTEM_TRUSTED_KEYS`
   - `$scripts/config --disable SYSTEM_REVOCATION_KEYS`
6. Build kernel using command: `$make -j16`
7. Install the modules & kernel:
   - `$sudo su`
   - `$make modules_install`
   - `$make install`
8. Set grub waiting time to enable kernel options when booting:
   - `$cd /etc/default`
   - `$cp grub grub.bak # create a backup`
   - `$vim grub`, and set `GRUB_TIMEOUT=5`
   - `$update-grub`
9. Reboot and select the newly installed kernel: `$reboot`

3 Task I

In this task, the implemented program forks child process to execute external programs, recieves and handles signals that are sent back by the child program.

3.1 Program Design & implementaion

To run an external program in user mode and analysis it running status, the program will not `exec` it directly, instead, it (the main process) will first fork a child process and do `exec` in the child process. After execution, the main process will wait until the external program exits, stops or is terminated. Finally, the main process prints out the information about signals recieved.

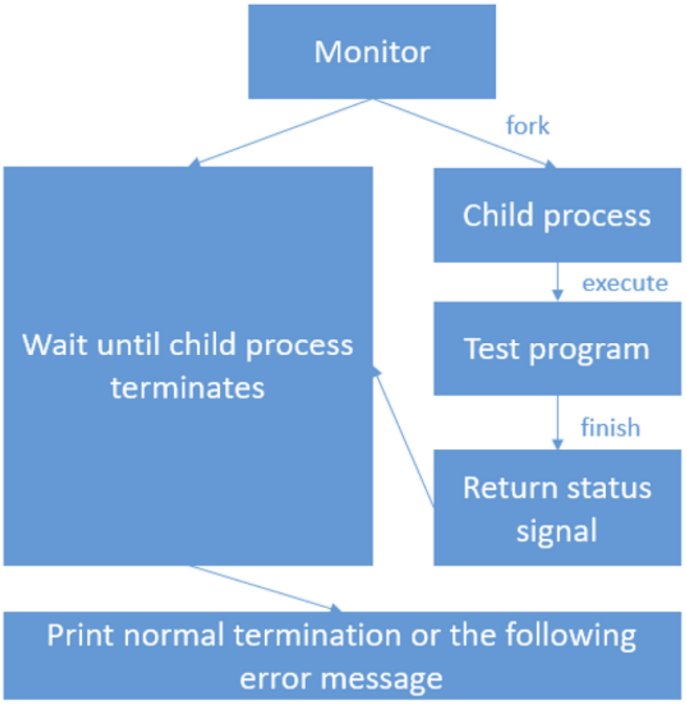


Figure 3: The main flow chart for Task 1 in the assignment guideline. The implementation of this task is stick to this design.

The implementaion is done with the help of necessary system functions:

Function Name	Description
`fork`	create a child process that is identical to main process (i.e., copy entire resources of parent process to its child process)
`waitpid`	wait until child process finish or enter specific modes
`execve`	substitute the origin process with external program

At the bigining, the main process calls `fork()` to create a child process, and then the main process performs `waitpid(-1, &status, WUNTRACED)` to wait for the child process. The third argument is used to set keep track of stopped/untraced child process. Since `waitpid` has default wait option of `WEXITED` by the defination, so the third option `WUNTRACED` itself has the same effect as `WUNTRACED|WEXITED`, which is the desired wait option.

After child process lauched, it calls `execve(path, arg, env)` to substitute itself with external program at `path`. Then the external program started as child process of the main process. In case there is error in starting execution, an error detection is added.

The remaining job of the main process after wait its child process is to decode the signals from the child and print out the stauts. This step is implemented with macros in header `wait.h`.

3.2 Program Usage

Compiling

- `$cd program1`
- `$make`

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  make
cc -o abort abort.c
cc -o alarm alarm.c
cc -o bus bus.c
cc -o floating floating.c
cc -o hangup hangup.c
cc -o illegal_instr illegal_instr.c
cc -o interrupt interrupt.c
cc -o kill kill.c
cc -o normal normal.c
cc -o pipe pipe.c
cc -o program1 program1.c
cc -o quit quit.c
cc -o segment_fault segment_fault.c
cc -o stop stop.c
cc -o terminate terminate.c
cc -o trap trap.c
```

Running

run with the command: `./program1 <executable-file-path>`

3.3 Output

Following are 15 output examples:

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./abort                                                    127 ↵
Process start to fork
I'm the Parent Process, my pid = 6797
I'm the Child Process, my pid = 6798
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
```

(a) SIGABRT

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./alarm
Process start to fork
I'm the Parent Process, my pid = 6812
I'm the Child Process, my pid = 6813
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
```

(b) SIGALRM

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./bus
Process start to fork
I'm the Parent Process, my pid = 6818
I'm the Child Process, my pid = 6819
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
```

(c) SIGBUS

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./floating
Process start to fork
I'm the Parent Process, my pid = 6999
I'm the Child Process, my pid = 7000
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGFPE program

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

(d) SIGFPE

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./hangup
Process start to fork
I'm the Parent Process, my pid = 7014
I'm the Child Process, my pid = 7015
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGHUP program

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

(e) SIGHUP

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./illegal_instr
Process start to fork
I'm the Parent Process, my pid = 7020
I'm the Child Process, my pid = 7021
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
```

(f) SIGILL

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./interrupt
Process start to fork
I'm the Parent Process, my pid = 7080
I'm the Child Process, my pid = 7081
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
```

(g) SIGINT

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./kill
Process start to fork
I'm the Parent Process, my pid = 7087
I'm the Child Process, my pid = 7088
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
```

(h) SIGKILL

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./normal
Process start to fork
I'm the Parent Process, my pid = 7093
I'm the Child Process, my pid = 7094
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
```

(i) normal case

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./pipe
Process start to fork
I'm the Parent Process, my pid = 7161
I'm the Child Process, my pid = 7162
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGPIPE program

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

(j) SIGPIPE

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./quit
Process start to fork
I'm the Parent Process, my pid = 7167
I'm the Child Process, my pid = 7168
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
```

(k) SIGQUIT

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./segment_fault
Process start to fork
I'm the Parent Process, my pid = 7174
I'm the Child Process, my pid = 7175
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
```

(l) SIGSEGV

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./stop
Process start to fork
I'm the Parent Process, my pid = 7250
I'm the Child Process, my pid = 7251
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
```

(m) SIGSTOP

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./terminate
Process start to fork
I'm the Parent Process, my pid = 7257
I'm the Child Process, my pid = 7258
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
```

(n) SIGTERM

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program1
└─➤  ./program1 ./trap
Process start to fork
I'm the Parent Process, my pid = 7263
I'm the Child Process, my pid = 7264
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
```

(o) SIGTRAP

4 Task II

4.1 Program Design & implementaion

The main idea of this program is similar to that of Task I. However, program of this task is substantially a kernel module, which runs only in kernel mode. So the parent and child processes become kernel threads. The main difference this time is to utilize kernel functions instead of user functions/syscalls to conduct `fork`, `wait`, and `exec`.

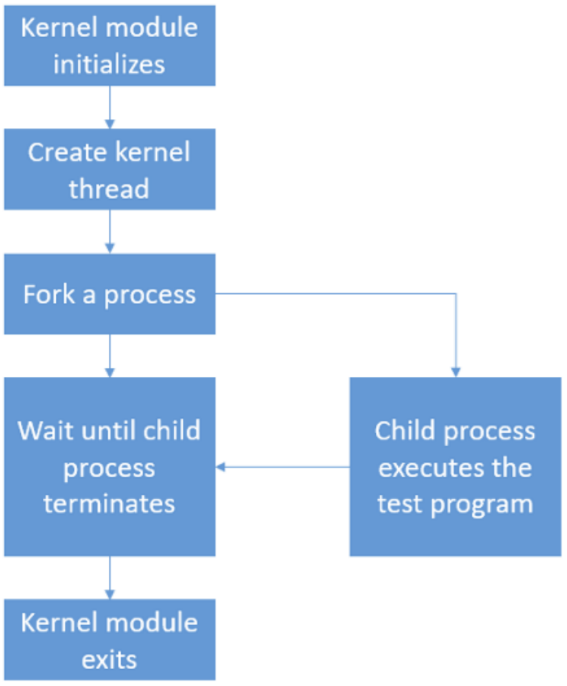


Figure 4: The main flow chart for Task 2 in the assignment guideline. The implementation of this task is stick to this design.

1. When initialization, the module create a kernel_thread by calling `kthread_create`, and wake up the process by calling `wake_up_process(kthr)`.
2. When doing fork, the module calls `kernel_thread(&my_exec, NULL, SIGCHILD)` to perform a function similar to fork.
3. When waiting the child process, a function called `ex_kernel_wait` is implemented to act the role of `wait` This function is based on kernel function `do_wait`, and to use desired waiting option, this newly implemented function simulates the implementaion of `kernel_wait` which only allows `WEXITED` option, and extended a new parameter `flags` to pass in desired options.
4. When executing, `do_execve` is called to run (but not substitute with) external program.

4.2 Program Usage

Compiling

Compiling the kernel module:

- `$cd program2`
- `$make`

After compilation, `program2.ko` is created

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program2
└─➤ make
make -C /lib/modules/5.10.146/build M=/home/jdr/Desktop/CSC3150/Assignment_1_120090562/source/program2
modules
make[1]: Entering directory '/home/jdr/Documents/dev/linux-5.10.146'
  CC [M]  /home/jdr/Desktop/CSC3150/Assignment_1_120090562/source/program2/program2.o
  MODPOST /home/jdr/Desktop/CSC3150/Assignment_1_120090562/source/program2/Module.symvers
  CC [M]  /home/jdr/Desktop/CSC3150/Assignment_1_120090562/source/program2/program2.mod.o
  LD [M]  /home/jdr/Desktop/CSC3150/Assignment_1_120090562/source/program2/program2.ko
make[1]: Leaving directory '/home/jdr/Documents/dev/linux-5.10.146'
```

Compiling the test executable: `gcc test.c -o /tmp/test`

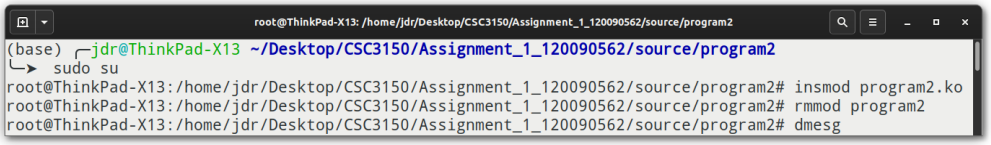
```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program2
└─➤ gcc test.c -o /tmp/test
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/CSC3150/Assignment_1_120090562/source/program2
└─➤ ls /tmp/test
/tmp/test
```

Running

The program is run by kernel module installation:

- `$sudo su`

- `$insmod program2.ko`
- `$rmmod program2`
- `$dmesg # to view kernel log`



4.3 Output

Following are 15 output examples:

```
[  860.343103] [program2] : module_init Derong Jin 120090562
[  860.343106] [program2] : module_init create kthread start
[  860.343403] [program2] : module_init kthread start
[  860.343508] [program2] : The child process has pid = 4267
[  860.343583] [program2] : This is the parent process, pid = 4266
[  860.343609] [program2] : child process
[  860.479196] [program2] : get SIGABRT signal
[  860.479198] [program2] : child process terminated
[  860.479199] [program2] : The return signal is 6
[  863.246811] [program2] : module_exit
```

(a) SIGABRT

```
[  904.539946] [program2] : module_init Derong Jin 120090562
[  904.539950] [program2] : module_init create kthread start
[  904.540107] [program2] : module_init kthread start
[  904.540153] [program2] : The child process has pid = 4311
[  904.540154] [program2] : This is the parent process, pid = 4310
[  904.540156] [program2] : child process
[  904.547634] [program2] : get SIGALRM signal
[  904.547636] [program2] : child process terminated
[  904.547636] [program2] : The return signal is 14
[  909.250669] [program2] : module_exit
```

(b) SIGALRM

```
[  924.450350] [program2] : module_init Derong Jin 120090562
[  924.450354] [program2] : module_init create kthread start
[  924.450603] [program2] : module_init kthread start
[  924.450700] [program2] : The child process has pid = 4358
[  924.450703] [program2] : This is the parent process, pid = 4357
[  924.450785] [program2] : child process
[  924.586476] [program2] : get SIGBUS signal
[  924.586478] [program2] : child process terminated
[  924.586479] [program2] : The return signal is 7
[  926.480452] [program2] : module_exit
```

(c) SIGBUS

```
[  947.993667] [program2] : module_init Derong Jin 120090562
[  947.993671] [program2] : module_init create kthread start
[  947.993878] [program2] : module_init kthread start
[  947.993927] [program2] : The child process has pid = 4392
[  947.993930] [program2] : This is the parent process, pid = 4391
[  947.993981] [program2] : child process
[  948.130625] [program2] : get SIGFPE signal
[  948.130627] [program2] : child process terminated
[  948.130628] [program2] : The return signal is 8
[  950.464850] [program2] : module_exit
```

(d) SIGFPE

```
[  965.110548] [program2] : module_init Derong Jin 120090562
[  965.110553] [program2] : module_init create kthread start
[  965.110796] [program2] : module_init kthread start
[  965.110878] [program2] : The child process has pid = 4428
[  965.110885] [program2] : This is the parent process, pid = 4427
[  965.110985] [program2] : child process
[  965.111744] [program2] : get SIGHUP signal
[  965.111746] [program2] : child process terminated
[  965.111747] [program2] : The return signal is 1
[  966.875111] [program2] : module_exit
```

(e) SIGHUP

```
[  975.568376] [program2] : module_init Derong Jin 120090562
[  975.568379] [program2] : module_init create kthread start
[  975.568586] [program2] : module_init kthread start
[  975.568646] [program2] : The child process has pid = 4463
[  975.568648] [program2] : This is the parent process, pid = 4462
[  975.568768] [program2] : child process
[  975.701216] [program2] : get SIGILL signal
[  975.701218] [program2] : child process terminated
[  975.701219] [program2] : The return signal is 4
[  977.059413] [program2] : module_exit
```

(f) SIGILL

```
[  987.719083] [program2] : module_init Derong Jin 120090562
[  987.719085] [program2] : module_init create kthread start
[  987.719277] [program2] : module_init kthread start
[  987.719387] [program2] : The child process has pid = 4501
[  987.719389] [program2] : This is the parent process, pid = 4500
[  987.719451] [program2] : child process
[  987.719895] [program2] : get SIGINT signal
[  987.719900] [program2] : child process terminated
[  987.719903] [program2] : The return signal is 2
[  989.390366] [program2] : module_exit
```

(g) SIGINT

```
[ 1011.302170] [program2] : module_init Derong Jin 120090562
[ 1011.302173] [program2] : module_init create kthread start
[ 1011.302361] [program2] : module_init kthread start
[ 1011.302437] [program2] : The child process has pid = 4535
[ 1011.302442] [program2] : This is the parent process, pid = 4534
[ 1011.302593] [program2] : child process
[ 1011.303192] [program2] : get SIGKILL signal
[ 1011.303194] [program2] : child process terminated
[ 1011.303195] [program2] : The return signal is 9
[ 1012.951816] [program2] : module_exit
```

(h) SIGKILL

```
[ 1258.734440] [program2] : module_init Derong Jin 120090562
[ 1258.734447] [program2] : module_init create kthread start
[ 1258.734644] [program2] : module_init kthread start
[ 1258.734801] [program2] : The child process has pid = 4844
[ 1258.734811] [program2] : This is the parent process, pid = 4843
[ 1258.734867] [program2] : child process
[ 1258.743327] [program2] : child process ended normally with exit status 100
[ 1261.555590] [program2] : module_exit
```

(i) normal case

```
[ 1021.275096] [program2] : module_init Derong Jin 120090562
[ 1021.275099] [program2] : module_init create kthread start
[ 1021.275373] [program2] : module_init kthread start
[ 1021.275444] [program2] : The child process has pid = 4569
[ 1021.275448] [program2] : This is the parent process, pid = 4568
[ 1021.275556] [program2] : child process
[ 1021.276105] [program2] : get SIGPIPE signal
[ 1021.276107] [program2] : child process terminated
[ 1021.276108] [program2] : The return signal is 13
[ 1022.504451] [program2] : module_exit
```

(j) SIGPIPE

```
[ 1029.715000] [program2] : module_init Derong Jin 120090562
[ 1029.715001] [program2] : module_init create kthread start
[ 1029.715107] [program2] : module_init kthread start
[ 1029.715157] [program2] : The child process has pid = 4583
[ 1029.715158] [program2] : This is the parent process, pid = 4582
[ 1029.715160] [program2] : child process
[ 1029.848729] [program2] : get SIGQUIT signal
[ 1029.848730] [program2] : child process terminated
[ 1029.848732] [program2] : The return signal is 3
[ 1030.850694] [program2] : module_exit
```

(k) SIGQUIT

```
[ 1040.643325] [program2] : module_init Derong Jin 120090562
[ 1040.643327] [program2] : module_init create kthread start
[ 1040.643464] [program2] : module_init kthread start
[ 1040.643508] [program2] : The child process has pid = 4599
[ 1040.643509] [program2] : This is the parent process, pid = 4598
[ 1040.643547] [program2] : child process
[ 1040.775689] [program2] : get SIGSEGV signal
[ 1040.775690] [program2] : child process terminated
[ 1040.775691] [program2] : The return signal is 11
[ 1042.143827] [program2] : module_exit
```

(l) SIGSEGV

```
[ 1074.395614] [program2] : module_init Derong Jin 120090562
[ 1074.395617] [program2] : module_init create kthread start
[ 1074.395869] [program2] : module_init kthread start
[ 1074.396031] [program2] : The child process has pid = 4636
[ 1074.396036] [program2] : This is the parent process, pid = 4635
[ 1074.396055] [program2] : child process
[ 1074.396612] [program2] : get SIGSTOP signal
[ 1074.396615] [program2] : child process STOPPED
[ 1076.029773] [program2] : module_exit
```

(m) SIGSTOP

```
[ 1085.403235] [program2] : module_init Derong Jin 120090562
[ 1085.403238] [program2] : module_init create kthread start
[ 1085.403476] [program2] : module_init kthread start
[ 1085.403525] [program2] : The child process has pid = 4670
[ 1085.403528] [program2] : This is the parent process, pid = 4669
[ 1085.403530] [program2] : child process
[ 1085.404046] [program2] : get SIGTERM signal
[ 1085.404048] [program2] : child process terminated
[ 1085.404049] [program2] : The return signal is 15
[ 1086.559024] [program2] : module_exit
```

(n) SIGTERM

```
[ 1121.099953] [program2] : module_init Derong Jin 120090562
[ 1121.099956] [program2] : module_init create kthread start
[ 1121.100210] [program2] : module_init kthread start
[ 1121.100304] [program2] : The child process has pid = 4724
[ 1121.100307] [program2] : This is the parent process, pid = 4723
[ 1121.100394] [program2] : child process
[ 1121.238779] [program2] : get SIGTRAP signal
[ 1121.238780] [program2] : child process terminated
[ 1121.238781] [program2] : The return signal is 5
[ 1123.365021] [program2] : module_exit
```

(o) SIGTRAP

5 Bonus: implementation of pstree

The software, or program `pstree` is a helpful tool in monitoring ongoing process. It displays processes in tree structure in terminal window and supports multiple options.

```
Usage: pstree [-acglpsStTuZ] [ -h | -H PID ] [ -n | -N type ]
              [ -A | -G | -U ] [ PID | USER ]
   or: pstree -V

Display a tree of processes.

  -a, --arguments     show command line arguments
  -A, --ascii         use ASCII line drawing characters
  -c, --compact-not   don't compact identical subtrees
  -C, --color=TYPE    color process by attribute
                      (age)
  -g, --show-pgids    show process group ids; implies -c
  -G, --vt100         use VT100 line drawing characters
  -h, --highlight-all highlight current process and its ancestors
  -H PID, --highlight-pid=PID
                      highlight this process and its ancestors
  -l, --long          don't truncate long lines
  -n, --numeric-sort  sort output by PID
  -N TYPE, --ns-sort=TYPE
                      sort output by this namespace type
                              (cgroup, ipc, mnt, net, pid, user, uts)
  -p, --show-pids     show PIDs; implies -c
  -s, --show-parents  show parents of the selected process
  -S, --ns-changes    show namespace transitions
  -t, --thread-names  show full thread names
  -T, --hide-threads  hide threads, show only processes
  -u, --uid-changes   show uid transitions
  -U, --unicode       use UTF-8 (Unicode) line drawing characters
  -V, --version       display version information
  -Z, --security-context
                      show SELinux security contexts

  PID    start at this PID; default is 1 (init)
  USER   show only trees rooted at processes of this user
```

Figure 5: Usage table of full-featured pstree command in linux shell

5.1 Program Design & implementaion

To implemente the program of `pstree`, the information of processes' status and their relationship with other processes is needed.

Firstly, information related to current porcesses is stored under `/proc`, and information about specific process with id `pid` is stored under the directory `/proc/[pid]/`. Programs is allowed to acccess to file `/proc/[pid]/stat` in order to retireve demanding information about process `pid`'s status (the pattern of these files can be found by `man proc` command). Besides, child threads status can be access through file system `/proc/[pid]/task/[tid]/stat`. The parsing pattern is the same as that of process `stat`'s. Having all information collected, the only thing `pstree` should do is build the tree and displays it.

The implementaion details are introduced as follow:

To implemente in C language, several auxiliary `struct` are declared.

Function Name	Description
`proc_stat_t`	provide a copy of file `stat`.
`process_t`	stores the status & relationship of a process
`message_node_t`	represent the message of a pstree ndoe

Firstly, the definition of `proc_stat_t` follows the guide of `man proc` about `stat`. In other words, `proc_stat_t` provide a type to load content of `stat` into. Besides, a function `fread_proc_stat(FILE *fp, struct proc_stat_t *stat)` provide the method to read content in file `stat` into `proc_stat_t`.

C structure of `process_t` adn `task_t` stores information related to process & tasks using structure of linked list. Comparing to using static array, the usage of linked list improves the robustness of the entire program.

C structure `message_node_t` and `edge_t` use a graph to store the process tree. For conciseness, there is child-node merging in `pstree` program. So the structure of message tree might not be identical to that of a process tree. In merging procedure, child nodes without children and with the same name will be merged, and parent nodes with only one child will be merged with their unique child node.

The printing procedure is done with recurrence `print_tree(node, ..., cache)` where `cache` stores messages to be printed.

5.2 Program Usage

Comparing to full-featured `pstree` command, this implementaion only provide small subset of available options:

```
Usage: pstree [-cgptT] [ -n ]
              [ -A ] [ PID ]
   or: pstree -V

Display a tree of processes.

  -A, --ascii         use ASCII line drawing characters
  -c, --compact-not   don't compact identical subtrees
  -g, --show-pgids    show process group ids; implies -c
  -n, --numeric-sort  sort output by PID
  -p, --show-pids     show PIDs; implies -c
  -t, --thread-names  show full thread names
  -T, --hide-threads  hide threads, show only processes
  -V, --version       display version information

  PID    start at this PID; default is 1 (init)
```

Figure 6: Usage table of my pstree command

Compiling

compile the program (2 ways, either is ok):

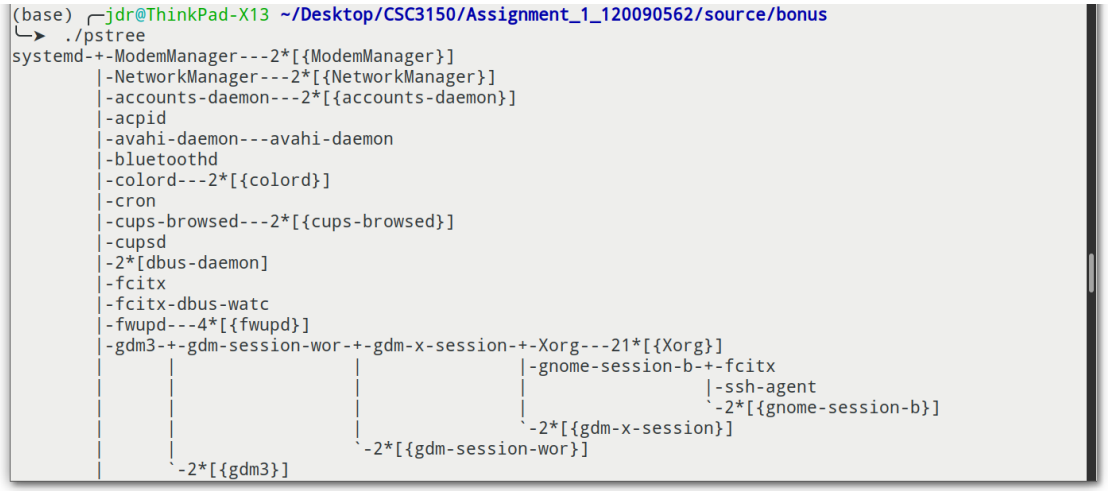
- option 1: `$make`
- option 2: `$gcc pstree.c -o pstree`

Running

the program can be execute using: `$./pstree [-cgptT] [-n] [-A] [PID]`

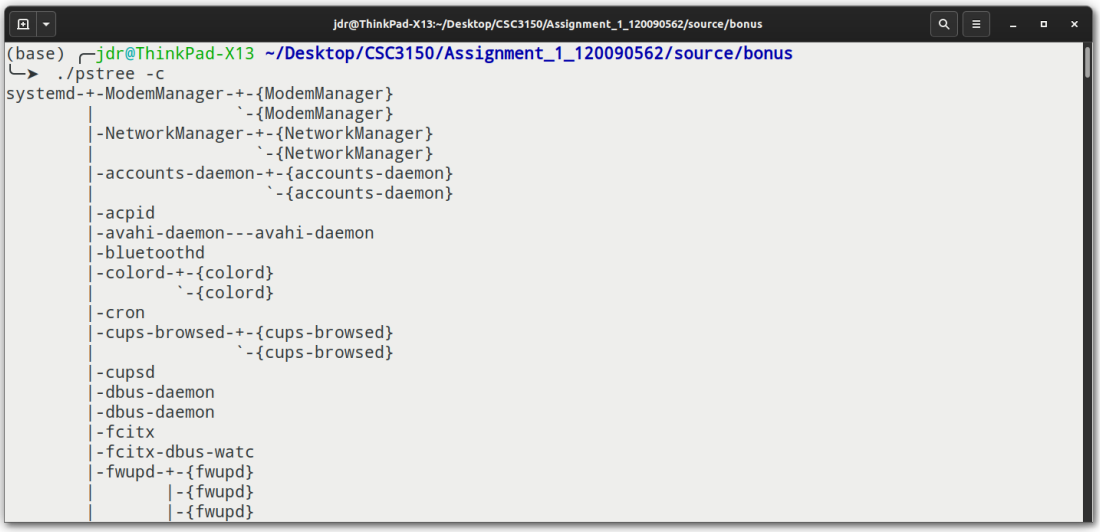
5.3 Output

Figure 7: Examples showing basic function without parameters



(a) Basic function with no parameters

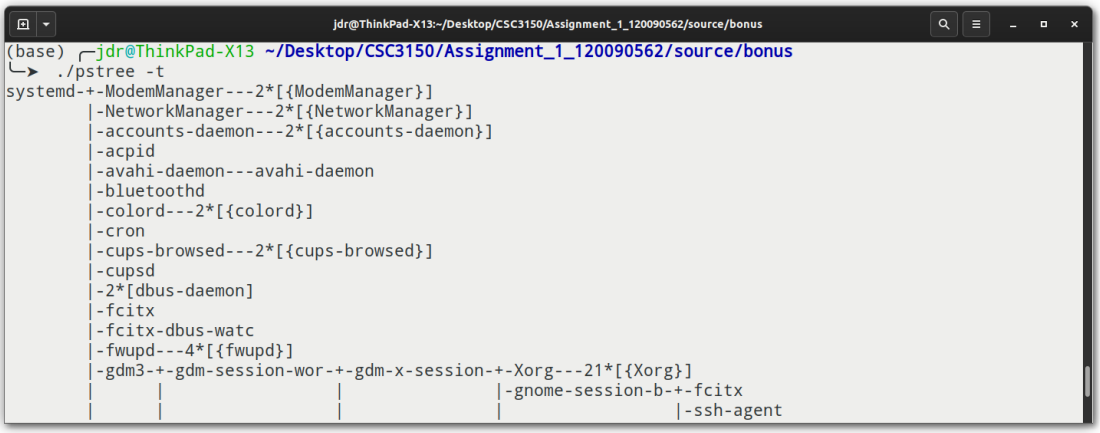
Figure 8: Example with options



(a) -c: without compacting identical subtrees

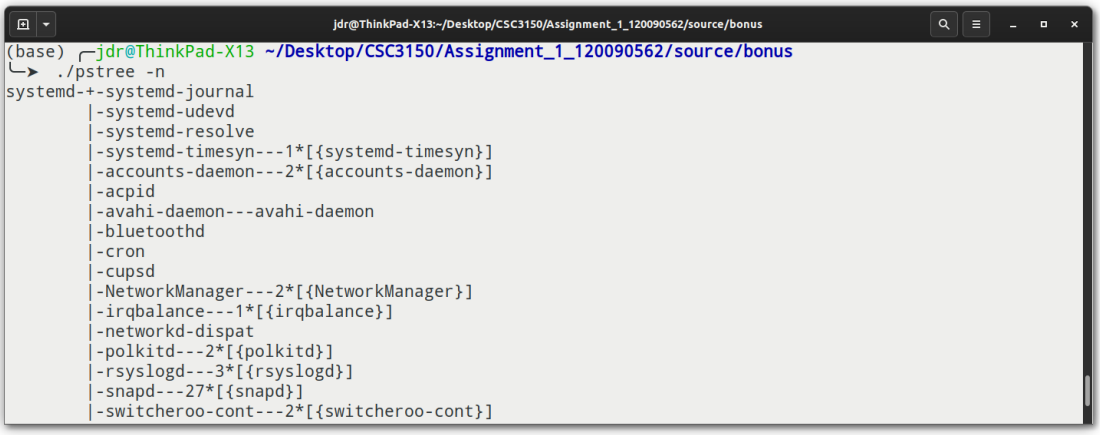
(b) -g: showing group id

(c) -p: showing process id

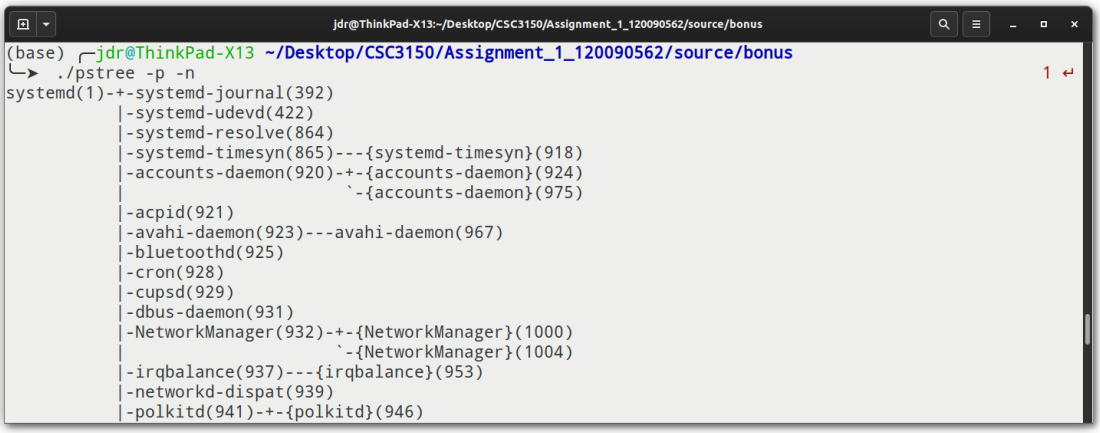


(d) -t: showing full thread names

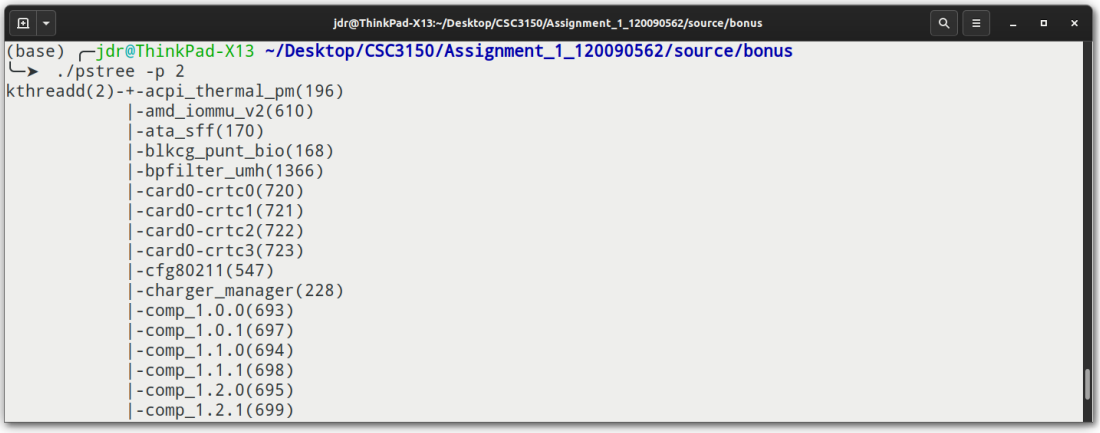
(e) -T: hiding threads, showing only processes



(f) -n: sort by PID

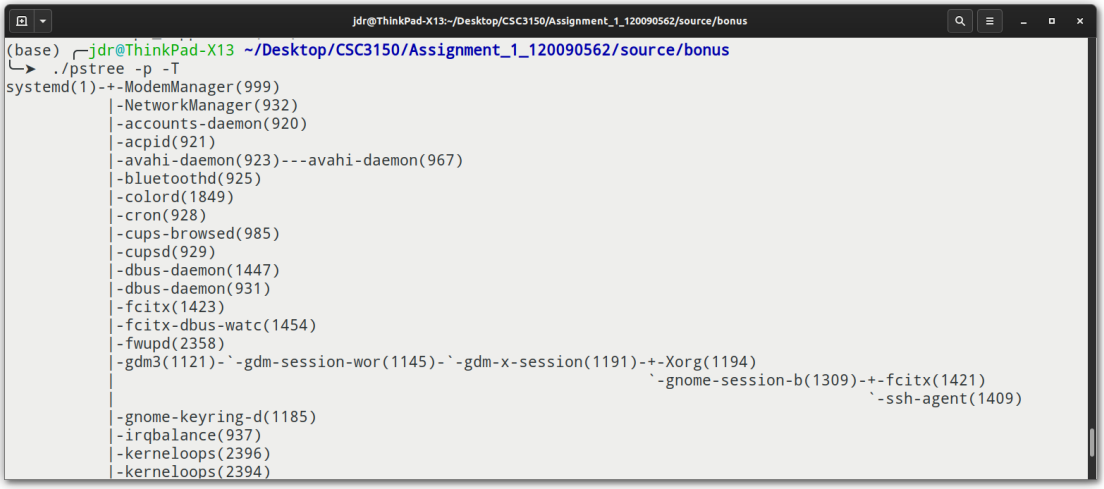


(g) -n -p: sort by PID, PID shown



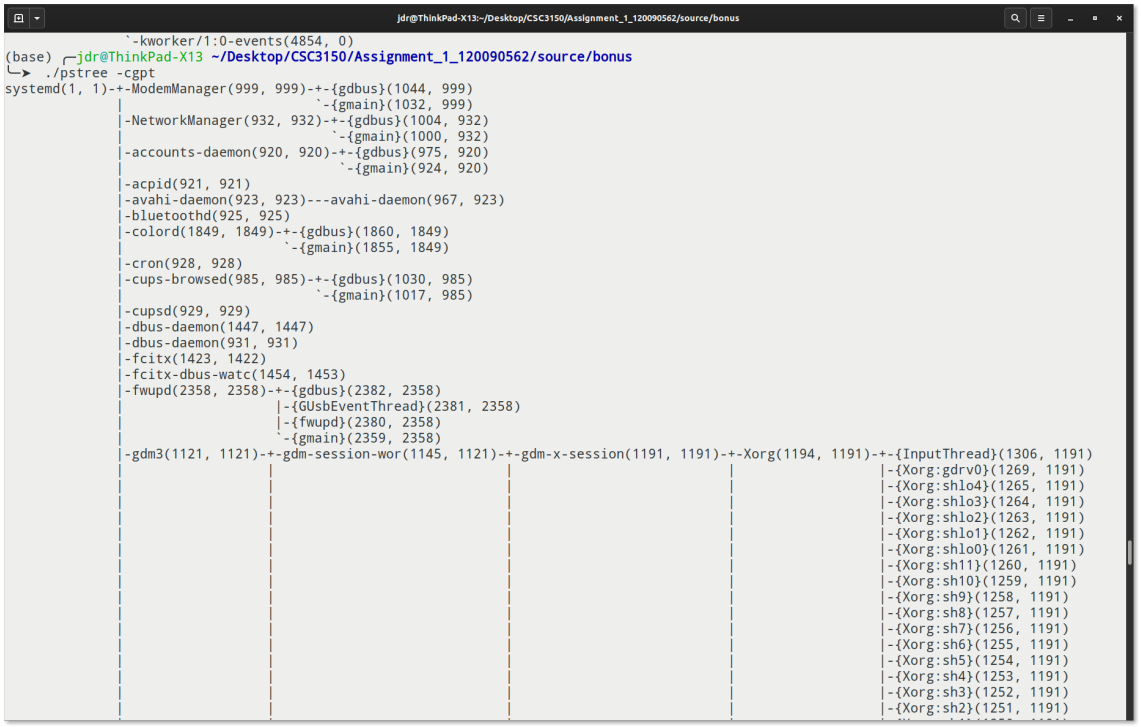
(h) start from a given PID

Figure 9: Example with multiple options

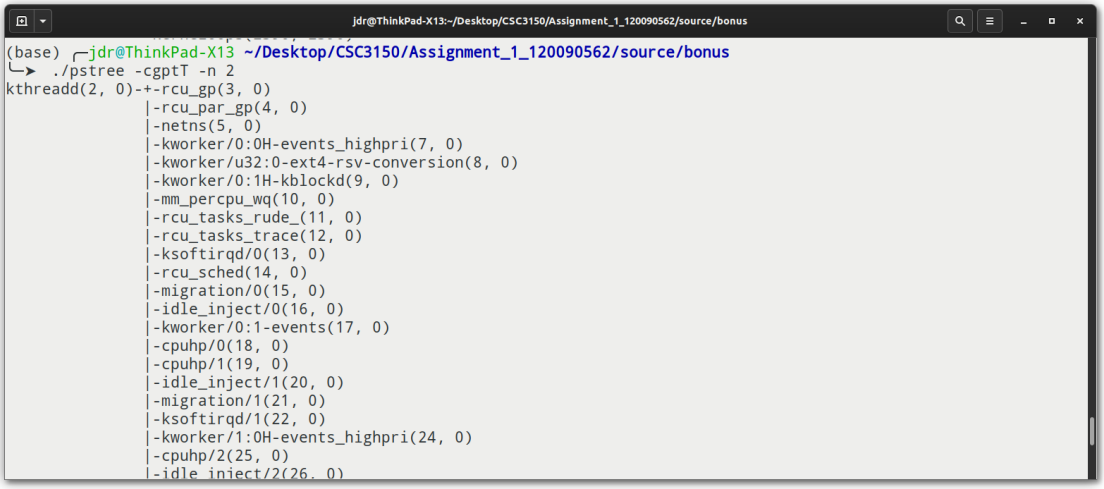


(a) seperate options

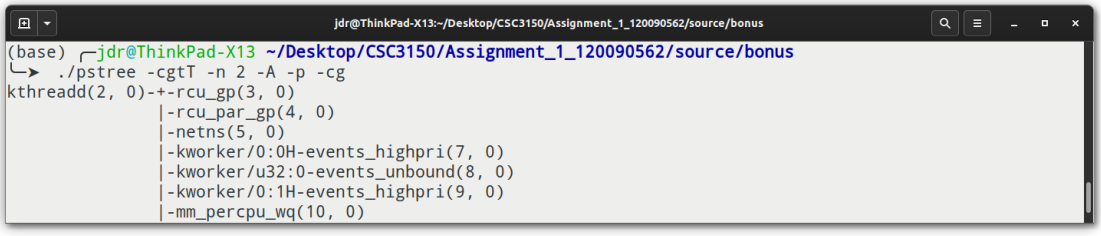
(b) nested options



(c) more options



(d) all options



(e) all options with random order

6 What I have learned from this assignment

In this assignment, I learned to use call system calls and became familiar with the interface between C language and linux OS (task I). I also wrote my first kernel module and became more familiar with the interfaces and structures (e.g., `struct task_struct`)defined in the kernel, and for the first time I came across data structures such as linux kthread_queue and learned some principles and rules of kernel development . In the process of completing the bonus task, I got a deeper understanding of linux process management from reading the manuscript.

This assignment not only brought me knowledge growth, but also greatly consolidated my knowledge of operating system concepts. When I was learning the theory of operating system, I was exposed to the concepts of process, thread, user space and kernel space. However, the textbook did not go into much detail about their implementation. In this assignment, I completed some system calls using C language and learned about the concepts of argument variables and environment before the program runs on hardware (`exec`), which gave me a deeper understanding of the underlying system. Not only that, but in the second task, I also implemented a kernel module using kernel functions for the first time. I had been using linux for a few years before that, but I basically knew nothing about how it worked, and most of the functions were direct calls to compiled binary executables. But to complete task 2, I consulted a lot of documentation and source code, which allowed me to really see the underlying implementation of the OS and gradually understand the underlying code logic. A large part of it is not explained in detail in the official documentation. For example, the waitoption item does the waitoption | `WEXITED` operation in the kernel when calling waitpid to wait for a stopped program, which ensures that the program can be waited for as long as it is terminated.

If the first two tasks enhanced my engineering ability in reading code and official documentation, the bonus task honed my ability to create a complete program from scratch. In writing the pstree code, I went through the complete process of analyzing the problem, deconstructing it, designing the flow, and writing the code. These processes definitely contributed to my global perspective in writing code and increased my experience in designing programs. I must mention that while working on pstree, it also deepened my mastery of C and my understanding of “everything is a file” in Linux.