1. **Program 1**
    1. Basic idea

        The program is to implement a function that can fork a child process to excute a test program. The monitor will fork a child process and the the child process will begin the test program. In the meantime, the parent process will be suspended until child process terminates. As soon as the parent receive the signal from the child process, it will determine the kind of signal and print corresponding information.

        Firstly, use fork( ) function to fork the child process. Then, the program will check the pid of the process and do the corresponding operations. As the value of pid is different for child and parent process, I use if to judge which process it is. If pid equal to 0, this process is child process, so I use execve( ) to execute the test program. If pid larger than 0, this process is parent process. So I use waitpid( ) to let the parent process wait the child process terminate. When parent receive the signal, it will use if to check how the child process terminated, and which signal it receive. Then the parent will print corresponding information, and exit normally.

    2. Environment

        version of OS: Ubuntu 16.04.2

        version of kernel: 5.10.5

    3. Sample output

        Terminated normally:

```
Process start to fork
I'm the Parent Process, my pid = 3447
I'm the Child Process, my pid = 3448
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the normal program

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

        Terminated with other signals:

        SIGBUS:

```
Process start to fork
I'm the Parent Process, my pid = 3487
I'm the Child Process, my pid = 3488
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGBUS program

Parent process receives the SIGCHIL signal
child process get SIGBUS signal
child process is bussed
CHILD EXECUTION FAILED
```

        SIGTERM:

```
Process start to fork
I'm the Parent Process, my pid = 3546
I'm the Child Process, my pid = 3547
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGTERM program

Parent process receives the SIGCHIL signal
child process get SIGTERM signal
child process is TERMINATED
CHILD EXECUTION FAILED
```

SIGPIPE:

```
Process start to fork
I'm the Parent Process, my pid = 3608
I'm the Child Process, my pid = 3609
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGPIPE program

Parent process receives the SIGCHIL signal
child process get SIGPIPE signal
child process is piped
CHILD EXECUTION FAILED
```

Process stopped with SIGSTOP:

```
Process start to fork
I'm the Parent Process, my pid = 3572
I'm the Child Process, my pid = 3573
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receives the SIGCHIL signal
child process get SIGSTOP signal
child process stopped
CHILD PROCESS STOPPED
```

2. Program

1. Design Rule

This program is to create a kernel thread. This thread will fork a process to execute the test program, and the parent process will wait until the child process terminates. The parent process should catch the signal raise by child process, and print the information. At first, we use _do_fork() to fork the process. This function can directly let the child process to perform a function, which I set is my_exec(). In the my_exec() function, it will use do_execve to execute the teat program we set before. While the test program terminates, the child process terminates too. I use my_wait() function for the parent process. In the my_wait() function, I use do_wait() to wait the signal from child process. The do_wait() function have a parameter wait_opts, which we should set before we use it. After child process terminate, the signal will be sent

to the wait_opts struct. The parent process can read the signal from it, and use if to decide which kind of signal it is and print corresponding information.

2. Compile kernel

The function _do_fork, do_execve, getname and _do_wait we use in the program2 could not use in program2. So we should export it from the kernel. At first, I use Vscode to open the files where these function in and export them by EXPORT_SYMBOL. As there is a static before do_wait, we should delete it first. After we have modified these files, we can start compile the kernel. We start from make bzImage, and then make modules, make modules_install, make install and reboot. While the machine reboot, we should choose the latest version of kernel. Then we can use extern in program2 to use these functions. And also, we should declare the struct wait_opts we use in do_wait function.

3. Sample out

…/program2/test

```
[ 9523.945633] [program2] : Module_init
[ 9523.946125] [program2] : Module_init create kthread start
[ 9523.946126] [program2] : Module_init Kthread starts
[ 9523.946844] The Child Process has pid = 7319
[ 9523.946845] This is the parent process, pid = 7317
[ 9524.073471] child process
[ 9524.073472] get SIGBUS signal
[ 9524.073473] child process is bussed
[ 9524.073473] The return signal is 7
[ 9525.148699] [program2] : Module_exit./my
```

…/program1/normal

```
[ 9805.933169] [program2] : Module_init
[ 9805.933428] [program2] : Module_init create kthread start
[ 9805.933429] [program2] : Module_init Kthread starts
[ 9805.934477] The Child Process has pid = 9646
[ 9805.934479] This is the parent process, pid = 9644
[ 9805.935164] child process
[ 9805.935165] get SIGCHID signal
[ 9805.935167] child process is normal
[ 9805.935167] The return signal is 0
[ 9807.104540] [program2] : Module_exit./my
```

…/program1/hangup

```
[10477.405480] [program2] : Module_init
[10477.406546] [program2] : Module_init create kthread start
[10477.406548] [program2] : Module_init Kthread starts
[10477.406722] The Child Process has pid = 11088
[10477.406723] This is the parent process, pid = 11086
[10477.417292] child process
[10477.417293] get SIGHUP signal
[10477.417295] child process is hung up
[10477.417296] The return signal is 1
[10478.528093] [program2] : Module_exit./my
```

…/program1/abort

```
[ 9678.801185] [program2] : Module_init
[ 9678.801280] [program2] : Module_init create kthread start
[ 9678.801280] [program2] : Module_init Kthread starts
[ 9678.801569] The Child Process has pid = 8255
[ 9678.801570] This is the parent process, pid = 8253
[ 9678.909733] child process
[ 9678.909734] get SIGABRT signal
[ 9678.909736] child process is aborted
[ 9678.909737] The return signal is 6
[ 9680.029121] [program2] : Module_exit./my
```

…/program1/interrupt

```
[ 9852.543948] [program2] : Module_init
[ 9852.544207] [program2] : Module_init create kthread start
[ 9852.544209] [program2] : Module_init Kthread starts
[ 9852.544751] The Child Process has pid = 10109
[ 9852.544753] This is the parent process, pid = 10107
[ 9852.555580] child process
[ 9852.555581] get SIGINT signal
[ 9852.555583] child process is interrupted
[ 9852.555584] The return signal is 2
[ 9854.877262] [program2] : Module_exit./my
```

…/program1/stop

```
[ 9727.865158] [program2] : Module_init
[ 9727.865344] [program2] : Module_init create kthread start
[ 9727.865345] [program2] : Module_init Kthread starts
[ 9727.865437] The Child Process has pid = 8719
[ 9727.865438] This is the parent process, pid = 8717
[ 9727.866590] child process
[ 9727.866590] get SIGSTOP signal
[ 9727.866593] child process stopped
[ 9727.866593] The return signal is 19
[ 9729.097550] [program2] : Module_exit./my
```

3.Bonus

1. Design idea

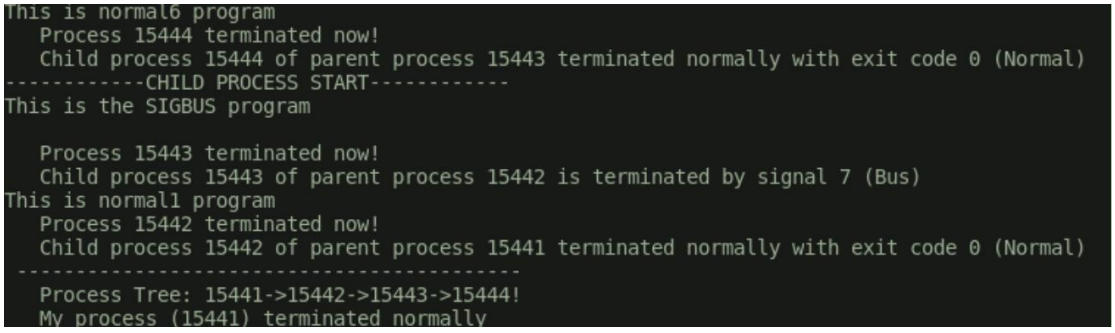
This program wants us to fork several processes and let the first one be second one's parent, second be third's parent, and so on. We use the same function fork() to fork the child process. At first, I want to use for loop to fork the child process. In every loop, the program fork at first, the use if to check what the program is. If it is child, execve() and continue. If it is parent, wait the signal and print information. But I find that I don't know how to do that, so I give up this way and try to use recursion function. Second, I use a function call forkmanytime(). This function will at first check whether it should fork again. If it needs to fork, it will fork the child process. I also use if to check which process it is. If it is child process, call forkmanytime() again, and execve() the corresponding program. If it is parent process, wait the signal. However, after I make it to run, I find that it will only execute the last program, because after the first execve() is called, all the child process will terminate, while the other execve() will not be called. So I think that I can put it in the part of parent, and this takes effect. Finally I design the function successfully. Forkmanytime() has 4 input: time, max, arg and first. Time is used to check the times of this function is called, while max control the max time of recursion. arg is the array of test program, and the first is used to

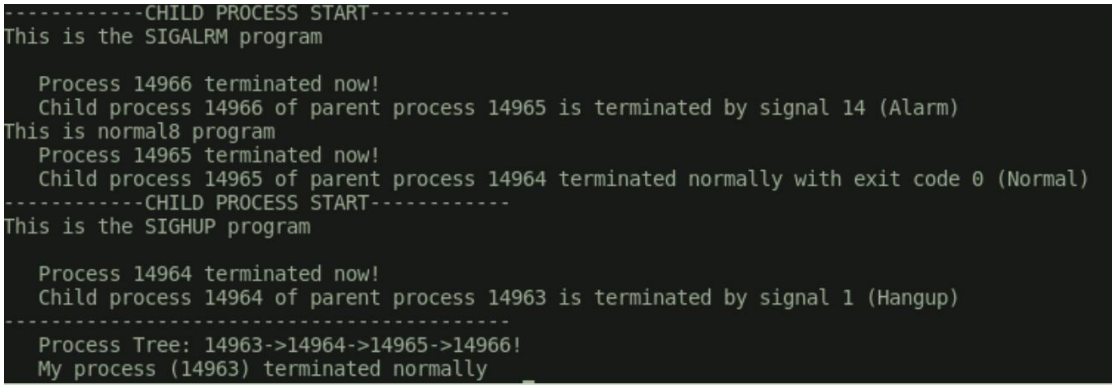
check whether the parent need to execve(). I will use a flow chart to make an introduction.

After we call forkmanytime(), at first, the pid is 100. Then it will fork a child process A1 with pid 101, while the parent process is B1. B1 wait the child process A1 terminate. Then the A1 will call forkmanytime(), fork A2 with pid 102. The parent process of A2 is B2, whose pid is 101. Then the A2 will call forkmanytime(), fork child process A3 with pid 103. The parent process of A3 is B3, whose pid is 102. A3 will call forkmanytime(), but this will not fork again because it achieve the max. So it executes alarm, then A3 terminate. The B3 receive the signal from A3, print the message and then execute normal8. After execute, process with pid 102 terminate, then B2 can receive the signal and execute hangup. And B1 will receive the signal from its child process B2. With the variable 'first' set by main, B1 will not execute. Then it will print and then terminate normally.

The forkmanytime( ) has been called four times. The first time is called by main function, while the 'time' set by 1, max set by number of programs without myfork, and the 'first' set by 0. This 0 will let parent process B1 not to execve(). Another three times called by the function itself. The 'time' will plus 1 every time, and the 'first' will set to 1 to let the B1 and B2 execve( ). The last call will not happen anything.

2.Sample out





4.Test

1. Program1: In the terminal of program1, first input 'make' and press Enter. Then input'. /program test' and press Enter. 'test' can be any name of file we want to test.

2. Program2: In the terminal of program2, first input 'make' and press Enter. Then input 'gcc - o test test.c' to make the test program. Then input 'sudo insmod program. ko' and

press Enter. Then input ‘sudo rmmod program. o’ and press Enter. Then input ‘dmesg | tail -n 10’ and press Enter. Then we can see the message in the terminal.

3. Bonus: In the terminal of bonus, first input ‘make’ and press Enter. Then input ‘. /myfork test1 test2 …’ and press Enter. Test can be any programs we want to test, and can have more than onr test.