CSC3150 Projecet1 Report

Name: Yiyan Hu

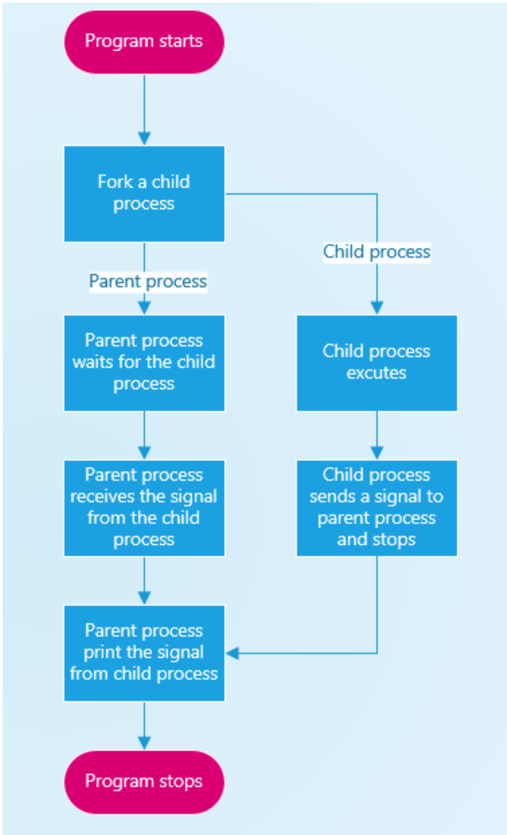
Student ID: 119010104

Design

Design 1: Program1

1. Overall design

   In the program 1, we are required to implement a program to (1) fork a child process, (2) make child process execute and send signal, (3) parent process wait for the child process, (4) parent process receives the signal and print the received signal. The following flow chart demonstrates the overall design of program 1.



2. Detailed design

   In this section, I will illustrate the detailed design of the program1.

   - Fork a child process: **fork()**

   > To fork a child process, we need to call the fork() function, the fork() function will create a child process and return a pid (process identification number), we can use the pid to check whether we are in the parent process or child process.

   - Child process executes: **execve()**

> To execute the child process, we can use the exec functions. One of the exec functions is the execve(), we can pass executable file name and arguments to the execve() and execute. The filename and arguments are passed to the main function of program1, which can be used to excute the child process.
>
> ```
> int execve (const char *filename, char *const argv[], char *const
> env[])
> ```

- Parent process wait for the child process: **waitpid()**

> To make the parent process wait for the child process, the parent process should call the wait() function.
>
> However, if the child process failed, the wait() function will not report the stopped child process. In this case, we should use the set the wait flag in the wait function to be "WUNTRACED" (WUNTRACED reports on stopped child process as well as terminated ones).
>
> To use the WUNTRACED wait flag, we need to use the waitpid() function with the WUTRACED flag setted.
>
> ```
> waitpid(pid, &status, WUNTRACED);
> ```

- Parent process receives and prints the signal

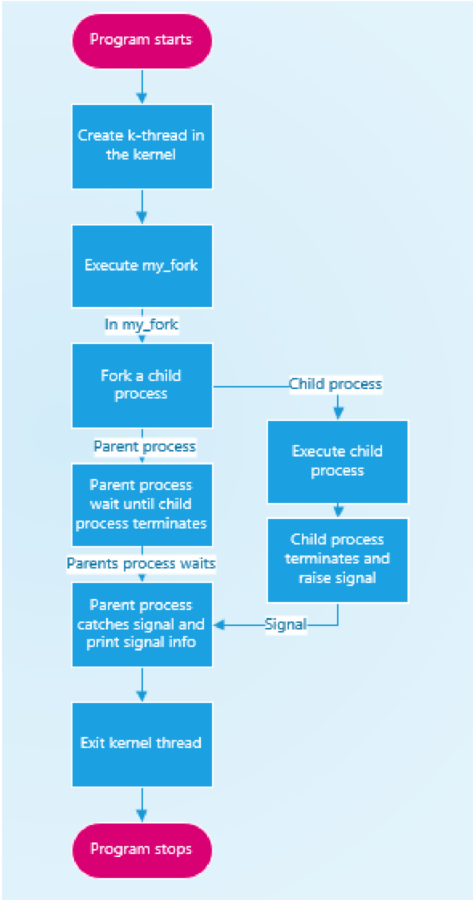
> The parent process will receive the signal from child process, and the parent process will print the information of the signal. To print the signal information, the parent process can read the process status passed by the child process.
>
> ```
> waitpid(pid, &status, WUNTRACED); //get status from child process
> if(WIFEXITED(status)){...} // get the signal info based on the status
> else if(WIFSIGNALED(status)) {...}
> else if(WIFSTOPPED(status)) {...}
> ```

Design 2: Program2

1. Overall design

   In program 2, we should create a kernel thread and fork a child process.

   We need: (1) create a kernel thread and run my_fork function, (2) fork a process in my_fork and execute the test programs, (3) in the kernel thread, parent process wait until the child process terminates, (4) the child process raise signal, and the parent process prints the signal. The following flow chart shows the overall process of program 2.



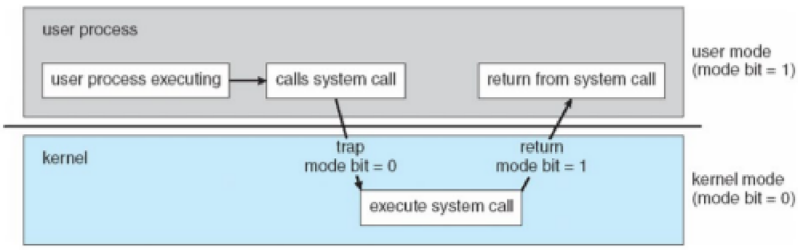
2. Detail design
    - Kernel compile: **EXPORT_SYMBOL()**

    > To make the program works properly, we need to use EXPORT_SYMBOL() and compile the kernel. The EXPORT_SYMBOL() provides API to be used in other module. We need implement the EXPORT_SYMBOL() in the kernel source code and exports the _do_wait(), do_fork(), getname() and do_execve().
    >
    > note that the we need to remove the static declaration when exporting the symbol (since the static indicates that the function only works within the file).

    ```
    EXPORT_SYMBOL(do_wait); // we need to add this line after the function in the source code
    ```

    - Create and exit the kernel thread: **k_thread_create()**

    > One important part of program 2 is controlling the kernel thread. The way for users to control the kernel is shown in the figure below.



> In program 2, we need to initialize and exit a kernel thread. To implement these functions, we need to create 2 methods: program2_init() and program2_exit(). In the program2_init method, we should create the kernel thread by calling the kthread_create().
>
> ```
> task=kthread_create(&my_fork,NULL,"MyThread"); // the first argument is the fcuntion to be executed in the thread
> if (!IS_ERR(task)) wake_up_process(task); // wake up the process after creating the thread
> ```

- Fork a child process and execute child process: **do_execve()** & **_do_fork()**

> To fork a child process, we can call the _do_fork() function. _do_fork() will create a child process, and _do_fork will return the pid of child process if the fork is successfully executed.
>
> The program can call the do_execve() to execute child process. In program 2, I encapsulate the do_execve() function in the my_exec() function. We can pass the pointer to my_exec() function as an argument to the _do_fork function to make the child process execute specific functions.
>
> ```
> pid = _do_fork(SIGCHLD, (unsigned long)&my_exec, 0, NULL, NULL, 0);
> // specify my_exec()'s address as child process stack pointer
> ```

- Parent process wait until the child process terminates: **do_wait()**

> do_wait() will make the parent process wait until child process terminates. In the kernel mod, when the system call do_wait() is executed, exit.ko module will be loaded. In program 2, I encapsulate do_wait in the my_wait function.

- Parent process print the signal form child process: **my_wait()**

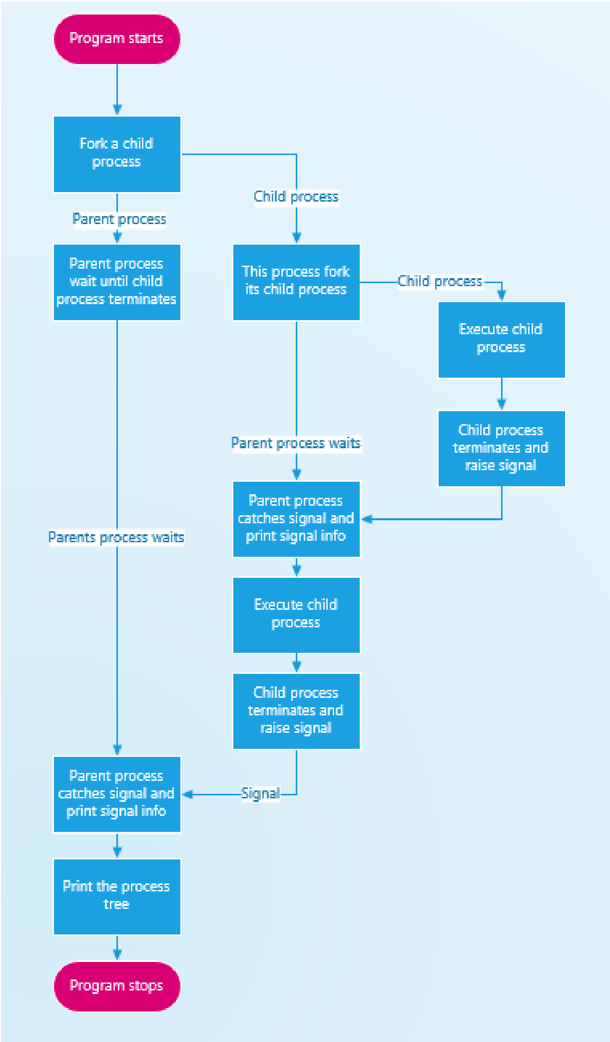
> ```
> struct wait_opts wo;
> ... // set the wait options
> int a=do_wait(&wo);
> ```
>
> The do_wait() function will change the information in the wait_opts wo, we can read the wo_stat (wait option status) in the wo to retrieve the signal from the child process.
>
> Note that we need to set the wo_flag of the wait options to make the parent process report on both stopped child process as well as terminated child process.
>
> ```
> wo.wo_flags=WEXITED|WSTOPPED;
> // WEXITED: wait for children terminate
> // WSTOPPED: wait for children that have been stopped by delivery of a signal
> ```

Design 3: Bonus Task

1. Overall design

In the bonus task, we need to implement a program myfork which can: (1) add multiple executable files as the argument of myfork , (2) fork child processes based on the argument (the preceeding argument is the parent of the tailing one), (3) child processes executes and parent processes catch the termination signals, (4) print out the process tree.

In this task, we need fork child process of a child process, the current child process can be the parent process if it fork another child process. Suppose we have 2 executable files as the argument to myfork, the flow of the program should look like the flowchart below.



2. Detailed design

- Read multiple executable file arguments: Use **argc** and **argv[]**

> The executable file arguments are passed to the main() function of the program, the argc equals to the number of the arguments, and char* argv[] contains the arguments passed to main(). We can use a char* arg[] array to store all the executable file arguments (except myfork).

- Build child-parent process tree using fork(): **Recursion**

> In the bonus task, the program should read multiple executable file arguments, and the proceeding argument should be the parent of the tailing one.

Suppose the argument structure is as follows, and each executable corresponds to a process.

```
test1(process1) -> test2(process2) -> test3(process3)
```

then

```
program starts
=> process1 is the parent of process2, process1 wait for process2
=> process2 is the parent of process3, process2 wait for process3
=> process3 execute and returns signal to process2
=> process2 gets signal and executes, process2 return signal to
process1
=> process1 gets signal
=> program finishes
```

If we replace the "return signal" words with "return value" in the above lines, we can find that the process of the bonus program has the same structure as recursion. In the recursion, the process (1) fork its child process, (2) wait for the child process, (3) execute if the child process terminates. Below is the recursive function structure.

```
void fork_child_process(/*arguments*/){
    if (iter==0) return; // base case
    /*fork a child*/
    fork();
    if (/*child process*/) {
        /*next recursion*/
        fork_child_process(/*arguments*/);
        /*execute this process*/
        execve(/*arguments*/);
    }
    else if (/*parent process*/) {
        /*wait for child process*/
        waitpid(/*arguments*/);
    }
}
```

- Collect information from parent processes and child processes: **Shared Memory**

We need to get information of different processes to print the child-parent process tree, which means that we should access memory across the processes.

Processes use different memory spaces, the parent process and child process both have their own copy of variables. The memory addresses with the same value in different processes are mapped into different physical addresses. In this case, we cannot pass an array (or pointer) to collect information across the processes.

One method to access memory across processes is to create a memory mapping which is shared by all the processes. The function can be realized by the mmap() function (which is included in the <sys/mman.h>).

> ```
> // create shared memory mapping
>     int * sharedMem = mmap(NULL, (2*argc-1)*sizeof(int),
>                            PROT_READ|PROT_WRITE,
>                            MAP_SHARED|MAP_ANONYMOUS, -1, 0);
> // mmap() creates a new mapping in the virtual address space of the
> calling process
> // the flag MAP_SHARED indicates that processes share this mapping
> ```

- Print the child-parent process tree and signal information

> After we collect the information of different processes, this step is fairly easy. The implementation of the printing part of bonus program is similar to that of the program2.

Environment to run the program

Linux, Linux Kernel, GCC Version

```
hyy@ubuntu:~$ cat /etc/issue
Ubuntu 16.04.5 LTS \n \l
hyy@ubuntu:~$ uname -r
4.10.14
hyy@ubuntu:~$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
```

Note: in the program2, we need to re-compile the kernel.

Program execution

Program1

1. Type **make** to make the files
2. Type **./program1** + executable file name (eg. normal)

   ```
   hyy@ubuntu:~/projects/ASS1/source/program1$ make
   ...
   hyy@ubuntu:~/projects/ASS1/source/program1$ ./program1 normal
   ...
   ```

Program2

1. Remember to use the re-compiled kernel
2. Type **Make** to make files
3. Sign in the root account
4. insert the module **insmod**, remove the module **rmmod**, check the log by **dmesg**

```
hyy@ubuntu:~/projects/ASS1/source/program2$ make
...
hyy@ubuntu:~/projects/ASS1/source/program2$ sudo su
[sudo] password for hyy:
root@ubuntu:/home/hyy/projects/ASS1/source/program2# insmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# rmmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# dmesg | tail -n 10
...
```

Bonus task

1. Make the files: **make**
2. Type **./myfork** + executable file names (eg. normal1 interrupt normal2)

```
hyy@ubuntu:~/projects/ASS1/source/bonus$ make
...
hyy@ubuntu:~/projects/ASS1/source/bonus$ ./myfork abort normal1 alarm
interrupt normal2 kill trap
...
```

Program execution output

Note: In this section, I will show some outputs of the programs. I only show several examples of each program in the report.

Program1

1. Normal termination:

```
hyy@ubuntu:~/projects/ASS1/source/program1$ ./program1 normal
Process start to fork
I'm the Parent Process, my pid = 31710
I'm the Child Process, my pid = 31711
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the normal program

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

2. Terminated by signals

Abort:

```
hyy@ubuntu:~/projects/ASS1/source/program1$ ./program1 abort
Process start to fork
I'm the Parent Process, my pid = 31738
I'm the Child Process, my pid = 31739
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGABRT program

Parent process receiving the SIGCHLD signal
child process get SIGABRT signal
child process encounters abnormal termination
CHILD EXECUTION FAILED
```

Pipe:

```
hyy@ubuntu:~/projects/ASS1/source/program1$ ./program1 pipe
Process start to fork
I'm the Parent Process, my pid = 31745
I'm the Child Process, my pid = 31746
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGPIPE program

Parent process receiving the SIGCHLD signal
child process get SIGPIPE signal
child process writes to pipe with no reader
CHILD EXECUTION FAILED
```

3. Child process stops

```
hyy@ubuntu:~/projects/ASS1/source/program1$ ./program1 stop
Process start to fork
I'm the Parent Process, my pid = 31916
I'm the Child Process, my pid = 31917
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receiving the SIGCHLD signal
child process get SIGSTOP signal
child process stopped
CHILD PROCESS STOPPED
```

Program2

1. Normal termination

```
root@ubuntu:/home/hyy/projects/ASS1/source/program2# insmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# rmmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# dmesg | tail -n 9
[82215.380560] [program2] : module_init
[82215.380562] [program2] : module_init create kthread start
[82215.381056] [program2] : module_init kthread start
[82215.384085] [program2] : The child process has pid = 34507
[82215.384087] [program2] : This is the parent process, pid = 34505
[82215.384089] [program2] : child process
[82215.384754] [program2] : child process runs normally
[82215.384756] [program2] : The return signal is 0
[82217.282805] [program2] : Module_exit./my
```

2. Terminated by signals

   Note: if the signal in the child process is sent with some time delay, we cannot remove the kernel module until the signal is sent.

   Hangup:

```
root@ubuntu:/home/hyy/projects/ASS1/source/program2# insmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# rmmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# dmesg | tail -n 10
[82322.041871] [program2] : module_init
[82322.041873] [program2] : module_init create kthread start
[82322.041925] [program2] : module_init kthread start
[82322.042084] [program2] : The child process has pid = 34977
[82322.042085] [program2] : This is the parent process, pid = 34976
[82322.042104] [program2] : child process
[82322.042578] [program2] : get SIGHUP signal
[82322.042579] [program2] : child process is hung up
[82322.042580] [program2] : The return signal is 1
[82323.723897] [program2] : Module_exit./my
```

   Alarm

```
root@ubuntu:/home/hyy/projects/ASS1/source/program2# insmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# rmmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# dmesg | tail -n 10
[82478.439412] [program2] : module_init
[82478.439419] [program2] : module_init create kthread start
[82478.440143] [program2] : module_init kthread start
[82478.440354] [program2] : The child process has pid = 35412
[82478.440357] [program2] : This is the parent process, pid = 35410
[82478.440360] [program2] : child process
[82480.441045] [program2] : get SIGALARM signal
[82480.441051] [program2] : child process is alarmed by real-timerclock
[82480.441053] [program2] : The return signal is 14
[82487.842800] [program2] : Module_exit./my
```

   Terminate

```
root@ubuntu:/home/hyy/projects/ASS1/source/program2# insmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# rmmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# dmesg | tail -n 10
[82649.445678] [program2] : module_init
[82649.445683] [program2] : module_init create kthread start
[82649.449992] [program2] : module_init kthread start
[82649.450094] [program2] : The child process has pid = 35840
[82649.450098] [program2] : This is the parent process, pid = 35839
[82649.450103] [program2] : child process
[82649.450658] [program2] : get SIGTERM signal
[82649.450661] [program2] : child process terminates
[82649.450663] [program2] : The return signal is 15
[82651.194578] [program2] : Module_exit./my
```

3. Child process stops

```
root@ubuntu:/home/hyy/projects/ASS1/source/program2# insmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# rmmod program2.ko
root@ubuntu:/home/hyy/projects/ASS1/source/program2# dmesg | tail -n 10
[82745.759295] [program2] : module_init
[82745.759300] [program2] : module_init create kthread start
[82745.760014] [program2] : module_init kthread start
[82745.761967] [program2] : The child process has pid = 36270
[82745.761972] [program2] : This is the parent process, pid = 36269
[82745.761976] [program2] : child process
[82745.763041] [program2] : get SIGSTOP signal
[82745.763045] [program2] : child process stops
[82745.763047] [program2] : The return signal is 19
[82747.499167] [program2] : Module_exit./my
```

Bonus Task

One demo ouput:

```
hyy@ubuntu:~/projects/ASS1/source/bonus$ ./myfork abort normal1 alarm interrupt
normal2 kill trap
------------CHILD PROCESS START------------
This is the SIGTRAP program

------------CHILD PROCESS START------------
This is the SIGKILL program

This is normal2 program
------------CHILD PROCESS START------------
This is the SIGINT program

------------CHILD PROCESS START------------
This is the SIGALRM program

This is normal1 program
------------CHILD PROCESS START------------
This is the SIGABRT program
```

```
---
process tree: 40362->40363->40364->40365->40366->40367->40368->40369
Child process 40369 of parent process 40368 is terminated by signal 5(Trap)
Child process 40368 of parent process 40367 is terminated by signal 9(Kill)
Child process 40367 of parent process 40366 terminated normally with exit code 0
Child process 40366 of parent process 40365 is terminated by signal 2(Interrupt)
Child process 40365 of parent process 40364 is terminated by signal 14(Alarm)
Child process 40364 of parent process 40363 terminated normally with exit code 0
Child process 40363 of parent process 40362 is terminated by signal 6(Abort)
Myfork process (40362) terminated normally
```

Some thoughts

In the project 1, I:

1. Work on parent processes and child processes => get a better understanding of the fork() and the parent-child process relationship.
2. Execute child process and make parent process wait for child process => learn system call
3. Create kthread and build kernel module => have a better insight of kernel and thread
4. Work on the bonus task => get knowledge of memory allocation of processes, learn to pass data across processes

The project 1 make me have hand-on practice regarding the process and kernel, which add on my coding experience. By the way, I feel my ability to search google is improved .

Thanks for being patient to grade my code and read this long report!