Assignment Report
Name: Meng Shuhan
Student ID: 123090422

Task1:

1. Program Design

In user mode, a child process is created, and fork() is used to separate the parent and child processes. The child process uses execl() to execute the specified test program. The parent process uses waitpid() to wait for the child process to finish. Based on the termination status of the child process, different information is printed:Normal termination: outputs Normal termination along with the exit code. Abnormal termination: prints the corresponding signal type and number.

2. Development Environment Setup

Linux Distribution: Ubuntu 16.04
Linux Kernel Version: 5.15.10
gcc version 5.4.0

3. Program Output Screenshot

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$  ./program1 ./normal
Process start to fork
I'm the Parent Process, my pid = 7765
I'm the Child Process, my pid = 7766
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$  ./program1 ./abort
Process start to fork
I'm the Parent Process, my pid = 7877
I'm the Child Process, my pid = 7878
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./alarm
Process start to fork
I'm the Parent Process, my pid = 7966
I'm the Child Process, my pid = 7967
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./bus
Process start to fork
I'm the Parent Process, my pid = 8010
I'm the Child Process, my pid = 8011
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./floating
Process start to fork
I'm the Parent Process, my pid = 8052
I'm the Child Process, my pid = 8053
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./hangup
Process start to fork
I'm the Parent Process, my pid = 8094
I'm the Child Process, my pid = 8095
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./illegal_instr
Process start to fork
I'm the Parent Process, my pid = 8158
I'm the Child Process, my pid = 8159
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./interrupt
Process start to fork
I'm the Parent Process, my pid = 8209
I'm the Child Process, my pid = 8210
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./kill
Process start to fork
I'm the Parent Process, my pid = 8265
I'm the Child Process, my pid = 8266
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./quit
Process start to fork
I'm the Parent Process, my pid = 8305
I'm the Child Process, my pid = 8306
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./pipe
Process start to fork
I'm the Parent Process, my pid = 8358
I'm the Child Process, my pid = 8359
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./segment_fault
Process start to fork
I'm the Parent Process, my pid = 8410
I'm the Child Process, my pid = 8411
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./stop
Process start to fork
I'm the Parent Process, my pid = 8448
I'm the Child Process, my pid = 8449
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./terminate
Process start to fork
I'm the Parent Process, my pid = 8500
I'm the Child Process, my pid = 8501
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
```

```
vagrant@csc3150:~/csc3150/Assignment_1_123090422/source/program1$ ./program1 ./trap
Process start to fork
I'm the Parent Process, my pid = 8540
I'm the Child Process, my pid = 8541
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
```

Task 2:

1. Program Design:

In Task 2, the main goal is to implement a kernel module (program2.c) that can create a kernel-level child process, execute a test program, and report its termination status, similar to Task 1 but entirely in kernel space. The design can be summarized in the following steps:

Kernel Module Initialization

When the module is loaded (program2_init), it creates a kernel thread using kthread_run().

This thread executes the function my_fork().

Using a kernel thread ensures that the fork-execute-wait sequence can run asynchronously in the kernel without blocking the module initialization.

Creating a Kernel-Level Child Process (my_fork)

Inside my_fork(), the kernel function kernel_clone() is used to create a new kernel-level child process.

The kernel_clone structure (kernel_clone_args) specifies:

The SIGCHLD signal to notify the parent process on termination.

The stack pointer of the child process points to the my_exec() function.

Default signal actions for the current process are initialized to SIG_DFL to ensure predictable signal behavior.

Child Process Execution (my_exec)

The child process executes my_exec(), which internally uses do_execve() to run the test program located at /tmp/test.

getname_kernel() is used to convert the file path into a kernel-usable filename structure.

This allows the kernel child process to run a user-space program.

Parent Process Waiting (my_wait)

The parent process waits for the child process to terminate using do_wait().

The termination status is stored in a wait_opts structure, which includes the child PID, wait flags, and return status.

This mimics the waitpid() behavior in user-space but works entirely in kernel context.

Parsing Termination Status

After the child process terminates, my_wait() parses its exit status using custom helper functions: wifexited(), wexitstatus(), wifsignaled(), wtermsig(), wifstopped(), wstopsig().

Normal termination prints a message including the exit status.

Abnormal termination prints which signal caused the termination (e.g., SIGSEGV, SIGABRT).

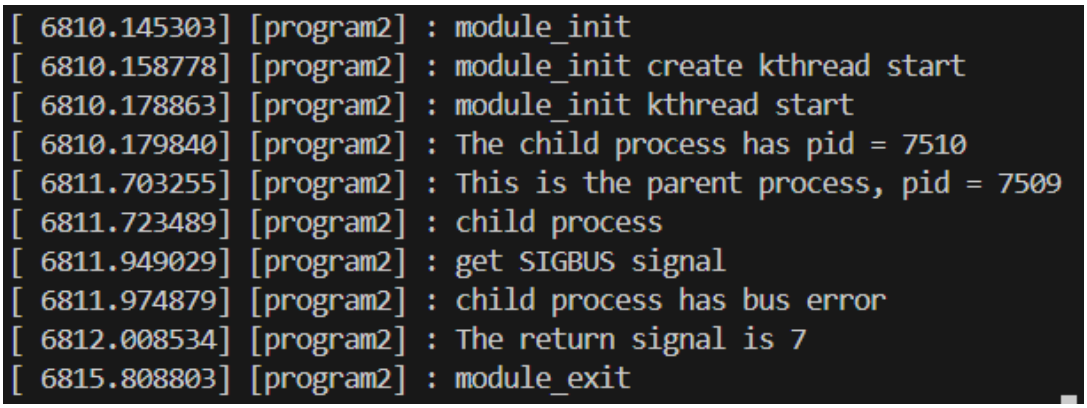
The kernel log (printk) is used to display messages, allowing observation via dmesg.

Parent-Child Process ID Reporting
my_fork() prints both parent and child PIDs using printk() to clearly identify the process relationship.
This helps in debugging and confirming that the kernel fork worked correctly.
Kernel Module Exit
When the module is removed (program2_exit), a simple printk() statement logs the exit.
The kernel thread and child processes terminate automatically when the module is removed.
Summary:
Task 2 effectively reproduces Task 1's fork-execute-wait pattern entirely in the kernel by combining kthread_run(), kernel_clone(), do_execve(), and do_wait(). Careful handling of signals and exit statuses ensures that the parent kernel thread can report the child process termination correctly, providing a full kernel-space demonstration of process control.

2. Development Environment Setup

Based on TASK 1,
modify the kernel functions to add
EXPORT_SYMBOL functionality,
and then recompile the kernel.

3. Program Output Screenshot



4. Lessons Learned

Understood the Linux user-space and kernel-space process creation mechanisms:In user-space, processes are created and executed using fork() and exec.In kernel-space, processes are managed via kernel_clone() and do_execve().
Learned how to use waitpid() and do_wait() to obtain child process termination information.
Mastered creating kernel threads (kthread) within kernel modules and interacting with user-space programs.
Learned how to interpret process termination status, including normal exit and signal-induced abnormal termination.
Improved understanding of Linux kernel debugging, such as using printk() and dmesg to view output.