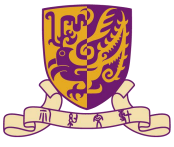
Assignment 1

CSC3150

Student Number: 117010032

Student Name: 陈雅茜



香 港 中 文 大 學（深 圳）

The Chinese University of Hong Kong, Shenzhen

Task 1

Introduction: Environment and Steps to Compile

This program is written to implement the functions which is able to fork a child process to evoke the external program. The parent process is able to catch the signal raised in child process and return the value. The program is implemented on C programming language on ubuntu (ubuntu version 16.04 and kernel version v4.8.0-36-generic). The user should input the test external file and the output will be the signal raised in the file.

This program will only support 15 signals which are:
SIGHUP, SIGINI, SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGFPE, SIGKILL, SIGBUS, SIGSEGV, SIGPIPE, SIGALARM, SIGTERM, SIGSTOP, SIGCHIL

Your command in your terminal should be like:

```
sudo make clean
sudo make
./program1 ./filename
```

(the filename here should be the name of your external test file, the command should be under the program1 folder)

For example, your input file should be:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc,char* argv[]){
	printf("------------CHILD PROCESS START------------\n");
	printf("This is the SIGPIPE program\n\n");
	raise(SIGPIPE);
	sleep(5);
	printf("------------CHILD PROCESS END------------\n");

	return 0;
}
```

For example, the corresponding output file should be:

```
  process start to fork
```

I'm the Parent process, my pid = 20735
I'm the Child process, my pid = 20736
Child process start to execute the program
----------CHILD PROCESS START----------
This is the SIGPIPE program

Parent process receiving the SIGCHILD signal
child process get SIGPIPE signal
child process is broken pipe by broken pipe signal
CHILD EXECUTION FAILED!!

Flow: How Did I Design My Program

1. Create the fork in the user mode to fork a child process

```
pid_t pid;
pid = fork();
int status;
//if the fork failed
if (pid == -1){
    perror("fork");
    exit(1);
}
```

2. The child process will execute the test program while the parent process will wait until the child process return.

```
/* fork a child process */
if (pid == 0){
    char *arg[argc];
    /* execute test program */
    printf("I'm the Child process, my pid = %d\n",getpid());
    for (int i = 0; i < argc - 1; i++){
        arg[i]=argv[i+1];
    }
    arg[argc-1] = NULL;
    printf("Child process start to execute the program\n");
    execve(arg[0],arg,NULL);
    perror("execve");
    //            raise(SIFCHILD);
}
```

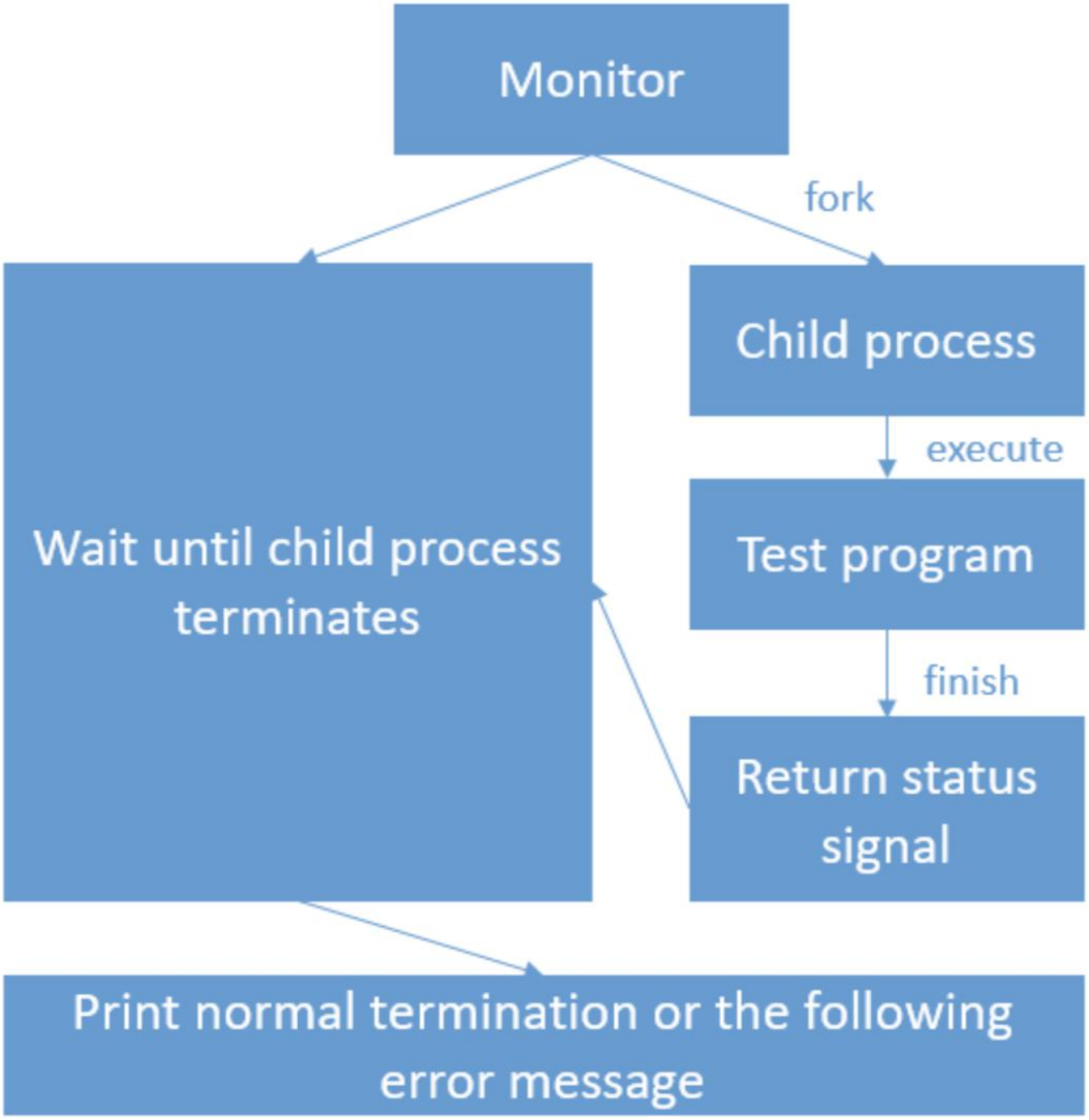
3. The child process will raise the signal and return it to the parent process

4. The parent process will recognize the signal with WIFEXITED(status), WIFSIGNALED(status) and WIFSTOPPED(status) functions.

```
            /* wait for child process terminates */
            printf("I'm the Parent process, my pid = %d\n",getpid());
            waitpid(pid,&status,WUNTRACED);
            printf("Parent process receiving the SIGCHILD signal\n");
            /* check child process'  termination status */
// for exited signals
            if (WIFEXITED(status)){
                printf("Normal termination with EXIT STATUS = %d\n",status);
            }
// for signaled signals
            else if (WIFSIGNALED(status)){
                char* buff1, *buff2;
                if (WTERMSIG(status) == 1){
                    buff1 = "SIGHUP";
                    buff2 = "hangup";
                }
                else if (WTERMSIG(status) == 2){
                    buff1 = "SIGINT";
                    buff2 = "interrupt";
                }
```

5. The parent process will use the WTERMSIG(status), WIFSTOPPED(status) and WEXITSTATUS(status) functions to further determine the signals' specific type.
6. Return the type in terminal

(the flow is shown as below)



Functions: How did I Design My Program

pid = fork(); this function is used to fork the child process whose pid is equal to 0; You can check if the current process is parent process or child process by check if pid is zero or positive number.

waitpid(pid,&status,WUNTRACED); this function in the parent process is used to let the parent process wait for the child process to be finished. It will catch the signal in child process when the child process finish.

WIFEXITED(status), WIFSIGNALED(status) and WIFSTOPPED(status); these functions is used to check the state of signal raised in the child process. If the child process exit the WIFEXITED(status) will return non-zero(true), the other two functions will return zero(false). It is the same ideal with the other two functions.

WTERMSIG(status); this function is used when WIFSIGNALED(status) returns true and you want to further check the single type.

Results: Screen Shot of My Output Result

For the test file: normal, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./normal
process start to fork
I'm the Parent process, my pid = 3920
I'm the Child process, my pid = 3921
Child process start to execute the program
------------CHILD PROCESS START------------
This is the normal program

------------CHILD PROCESS END------------
Parent process receiving the SIGCHILD signal
Normal termination with EXIT STATUS = 0
[09/26/19]seed@VM:.../program1$
```

For the test file: abort, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./abort
process start to fork
I'm the Parent process, my pid = 3927
I'm the Child process, my pid = 3928
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGABRT program

Parent process receiving the SIGCHILD signal
child process get SIGABRT signal
child process is abort by abort signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

For the test file: stop, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./stop
process start to fork
I'm the Parent process, my pid = 3934
I'm the Child process, my pid = 3935
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receiving the SIGCHILD signal
child process get SIGSTOP signal
child process stopped
CHILD PROCESS STOPPED
[09/26/19]seed@VM:.../program1$
```

For the test file: alarm, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./alarm
process start to fork
I'm the Parent process, my pid = 3941
I'm the Child process, my pid = 3942
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGALRM program

Parent process receiving the SIGCHILD signal
child process get SIGALARM signal
child process is alarm by alarm signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

For the test file: bus, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./bus
process start to fork
I'm the Parent process, my pid = 3946
I'm the Child process, my pid = 3947
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGBUS program

Parent process receiving the SIGCHILD signal
child process get SIGBUS signal
child process is bus error by bus error signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

For the test file: floating, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./floating
process start to fork
I'm the Parent process, my pid = 3956
I'm the Child process, my pid = 3957
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGFPE program

Parent process receiving the SIGCHILD signal
child process get SIGFPE signal
child process is floating point exception by floating point excep
tion signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

For the test file: hangup, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./hangup
process start to fork
I'm the Parent process, my pid = 3963
I'm the Child process, my pid = 3964
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGHUP program

Parent process receiving the SIGCHILD signal
child process get SIGHUP signal
child process is hangup by hangup signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

For the test file: illegal_instr, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./illegal_instr
process start to fork
I'm the Parent process, my pid = 3968
I'm the Child process, my pid = 3969
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGILL program

Parent process receiving the SIGCHILD signal
child process get SIGILL signal
child process is illegal instruction by illegal instruction signa
l
CHILD EXECUTION FAILED!!
```

For the test file: interrupt, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./interrupt
process start to fork
I'm the Parent process, my pid = 20464
I'm the Child process, my pid = 20465
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGINT program

Parent process receiving the SIGCHILD signal
child process get SIGINT signal
child process is interrupt by interrupt signal
CHILD EXECUTION FAILED!!
```

For the test file: kill, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./kill
process start to fork
I'm the Parent process, my pid = 20569
I'm the Child process, my pid = 20570
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGKILL program

Parent process receiving the SIGCHILD signal
child process get SIGKILL signal
child process is kill by kill signal
CHILD EXECUTION FAILED!!
```

For the test file: pipe, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./pipe
process start to fork
I'm the Parent process, my pid = 20578
I'm the Child process, my pid = 20579
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGPIPE program

Parent process receiving the SIGCHILD signal
child process get SIGPIPE signal
child process is broken pipe by broken pipe signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

For the test file: quit, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./quit
process start to fork
I'm the Parent process, my pid = 20585
I'm the Child process, my pid = 20586
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGQUIT program

Parent process receiving the SIGCHILD signal
child process get SIGQUIT signal
child process is quit by quit signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

For the test file: segment_fault, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./segment_fault
process start to fork
I'm the Parent process, my pid = 20592
I'm the Child process, my pid = 20593
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGSEGV program

Parent process receiving the SIGCHILD signal
child process get SIGSEGV signal
child process is segment fault by segment fault signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

For the test file: fterminate, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./terminate
process start to fork
I'm the Parent process, my pid = 20607
I'm the Child process, my pid = 20608
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGTERM program

Parent process receiving the SIGCHILD signal
child process get SIGTERM signal
child process is terminate by terminate signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

For the test file: trap, the return result in the terminal is

```
[09/26/19]seed@VM:.../program1$ ./program1 ./trap
process start to fork
I'm the Parent process, my pid = 20612
I'm the Child process, my pid = 20613
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGTRAP program

Parent process receiving the SIGCHILD signal
child process get SIGTRAP signal
child process is trap by trap signal
CHILD EXECUTION FAILED!!
[09/26/19]seed@VM:.../program1$
```

Conclusion: What I have learnt from this program

In this program I learnt 1) how to fork the child process, 2) how to execute the external file in the child process, 3) how to let the parent process wait for the child process and 4) how to determine the caught signal.

1) how to fork the child process

You should use the fork() function in the program to fork a child process. The function will return the parent process pid(process identifier) for the parent process, return 0 for the child process and return -1 if the fork failed. The variable type for the pid is pid_t.

After the child process being forked, the system will return an identical program in child process. Both child process and parent process will continue to execute the line after the fork function.

The getpid() function will return the pid of the current process and the ppid() function will return the pid of the parent of current process.

2) how to execute the external file in the child process

There are multiple functions for executing an external program in a child process, which are following

- Exec function family
  - int **execl** (const char *filename, const char *arg0, ...)
  - int **execve** (const char *filename, char *const argv[], char *const env[])
  - int **execle** (const char *filename, const char *arg0, char *const env[], ...)
  - int **execvp** (const char *filename, char *const argv[])
  - int **execlp** (const char *filename, const char *arg0, ...)

They are of the same function but with different input parameters. After execve() function, the child process will start to execute the external program from the very beginning. Any code in the child process after the execve() function will not be executed.

The function will not return anything if the execution function succeeds. It will return -1 if fails.

3) how to let the parent process wait for the child process

Since the terminated child process without parent's wait will become a zombie and the terminated child process with parent's termination will become an orphan,

it is important to make sure that parent process will wait for the child process to terminate with wait() function or waitpid() function.

The wait function only requires the status of the child process while the waitpid function need three input parameters which are pid, status and option. The waitpid function has three options which are 0, WNOHANG and WUNTRACED.

0 skips the option, WNOHANG requires the signal information immediately and WUNTRACED option will wait for the child process to terminate and return the signal information.

4) How to Determine the Caught Signal

There are 64 signals in linux and each of them have a corresponding signal code. The code and the signal can be shown by typing kill -l in the terminal.

```
[09/29/19]seed@VM:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

The signals are divided into three categories which are exited, signaled and stopped. The three categories of signals can be detected by WIFEXITED, WIFSIGNALED, WIFSTOPPED separately. The WIFEXITED signal will return if the process exited smoothly, the system will usually ignore such signals. The WIFSIGNALED signal will return if the process is terminated. The WIFSTOPPED signal will return if the process is stopped.

The function needs the status of the child process as the input and output the integer value. If the signal belongs to the corresponding category, the function will return non-zero, otherwise it will return 0.

In order to find the specific signal code for each signal. We can also use WEXITSTATUS, WTERMSIG, WSTOPSIG for each category.

Task 2

Introduction

This program is to implement a kernel module and insert it. The kernel module will firstly fork a child process and let the child process execute the test program. The program is able to catch the signal and return it in the kernel log. The program is implemented on C programming language on ubuntu (ubuntu version 16.04 and kernel version v4.8.0-36-generic). The kernel folder (version linux-4.10.14) is also required to execute the program and you need to make sure the "_do_fork", "do_execve", "getname" and "do_wait" functions in the kernel file is non-static. The user also needs to use EXPORT_SYMBOL_GLP (function_name) function to help you provide APIs to other modules. The user should input the test external file and the output will be the signal raised in the file.

This program will only support 15 signals which are:
SIGHUP, SIGINI, SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGFPE, SIGKILL, SIGBUS, SIGSEGV, SIGPIPE, SIGALARM, SIGTERM, SIGSTOP, SIGCHIL

You should execute the program following below steps:

1. Type "gcc -o test test.c" command under the program2 folder to compile test file.
2. Type "sudo make" command under the program2 folder to make the program2 file.
3. Type "sudo insmod program2.ko" command under the program2 folder to insert the kernel module into kernel.
4. Type "dmesg" command under the program2 folder to show the message

For example, your input test file should be like

```c
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

int main(int argc,char* argv[]){
    int i=0;

    printf("--------USER PROGRAM--------\n");
//  alarm(2);
    raise(SIGBUS);
    sleep(5);
    printf("user process success!!\n");
    printf("--------USER PROGRAM--------\n");
    return 100;
}
```

For example, the corresponding output file should be like

```
, Flow Control: RX
[ 1613.273401] [program2] : Module_exit
[ 1657.648362] [program2] : module_init
[ 1657.648382] [program2] : module_init create kthread start
[ 1657.648382] [program2] : module_init Kthread starts
[ 1657.654022] [program2] : The child process has pid = 4298
[ 1657.654023] [program2] : This is the parent process,pid = 4296
[ 1657.654932] [program2] : child process
[ 1657.654995] [program2] : child
[ 1657.655227] [program2] : get SIGBUS signal
[ 1657.655228] [program2] : child process has bus error error
[ 1657.655228] [program2] : The return signal is 7
```

Flow

1. Firstly, extern the needed functions in order to use them in your module.

```c
static struct task_struct* task;

extern long _do_fork(
                    unsigned long clone_flags,
                    unsigned long stack_start,
                    unsigned long stack_size,
                    int __user *parent_tidptr,
                    int __user *child_tidptr,
                    unsigned long tls);

extern int do_execve (
                     struct filename *filename,
                     const char __user *const __user *__argv,
                     const char __user *const __user *__envp);

struct wait_opts {
    enum pid_type wo_type; //It is defined in '/include/linux/pid.h'.
    int wo_flags; //Wait options. (0, WNOHANG, WEXITED, etc.)
    struct pid *wo_pid;  //Kernel's internal notion of a process identifier. "Find_get_pid()"
    struct siginfo __user *wo_info; //Singal information.
    int __user *wo_stat; // Child process's termination status
    struct rusage __user *wo_rusage; //Resource usage
    wait_queue_t child_wait; //Task wait queue
    int notask_error;};

extern long  do_wait (struct wait_opts *wo);

extern struct filename *getname(const char __user *filename);
```

2. Then, the program should initialize the kernel module and create a kernel thread using the __init program2_init(void) function. It checks if the initialization success to wake up the process.

```c
static int __init program2_init(void){

    printk("[program2] : module_init\n");

    /* write your code here */
    task = kthread_create(&my_fork,NULL,"Mythread");
    printk("[program2] : module_init create kthread start\n");

    if(!IS_ERR(task)){
        printk("[program2] : module_init Kthread starts\n");
        wake_up_process(task);
    }
    /* create a kernel thread to run my_fork */
    return 0;
}
```

3. After that, the child process will automatically evoke my_fork function whose implementation is shown as below. In the my_fork function, the program will set the default sigaction for the current process, fork a child process using the do_fork function, let the child process execute the my_exec function and let the parent process wait for the child process finish.

```c
//implement fork function
int my_fork(void *argc){
    //set default sigaction for current process
    int i;
    pid_t pid;
    struct k_sigaction *k_action = &current->sighand->action[0];
    for(i=0;i<_NSIG;i++){
        k_action->sa.sa_handler = SIG_DFL;
        k_action->sa.sa_flags = 0;
        k_action->sa.sa_restorer = NULL;
        sigemptyset(&k_action->sa.sa_mask);
        k_action++;
    }
    /* fork a process using do_fork */
    pid = _do_fork(SIGCHLD,(unsigned long)&my_exec,0,NULL,NULL,0);
    /* execute a test program in child process */
    printk("[program2] : The child process has pid = %d\n",pid);
    printk("[program2] : This is the parent process,pid = %d\n",(int)current->pid);
    /* wait until child process terminates */
    my_wait(pid);
    return 0;
}
```

4. The child process will then execute the my_exec function which will let the child process to execute the external test file. When the test file returns the valid result, the child process will exit with the do_exit(result)

function.

```c
int my_exec(void){
    printk("[program2] : child process");
    int result;
    const char path[] = "home/seed/work/test";
    const char *const argv[] = {path,NULL,NULL};
    const char *const envp[] = {"HOME=/","PATH=/sbin:/user/sbin:/bin:/usr/bin",NULL};

    struct filename * my_filename = getname(path);

    result = do_execve(my_filename,argv,envp);

    if(!result){
        printk("[program2] : child");
        return 0;
    }

    do_exit(result);
}
```

5. The parent process will wait for the child process to finish with the my_wait function. It will also check the return signal type with the *wo.wo_stat.

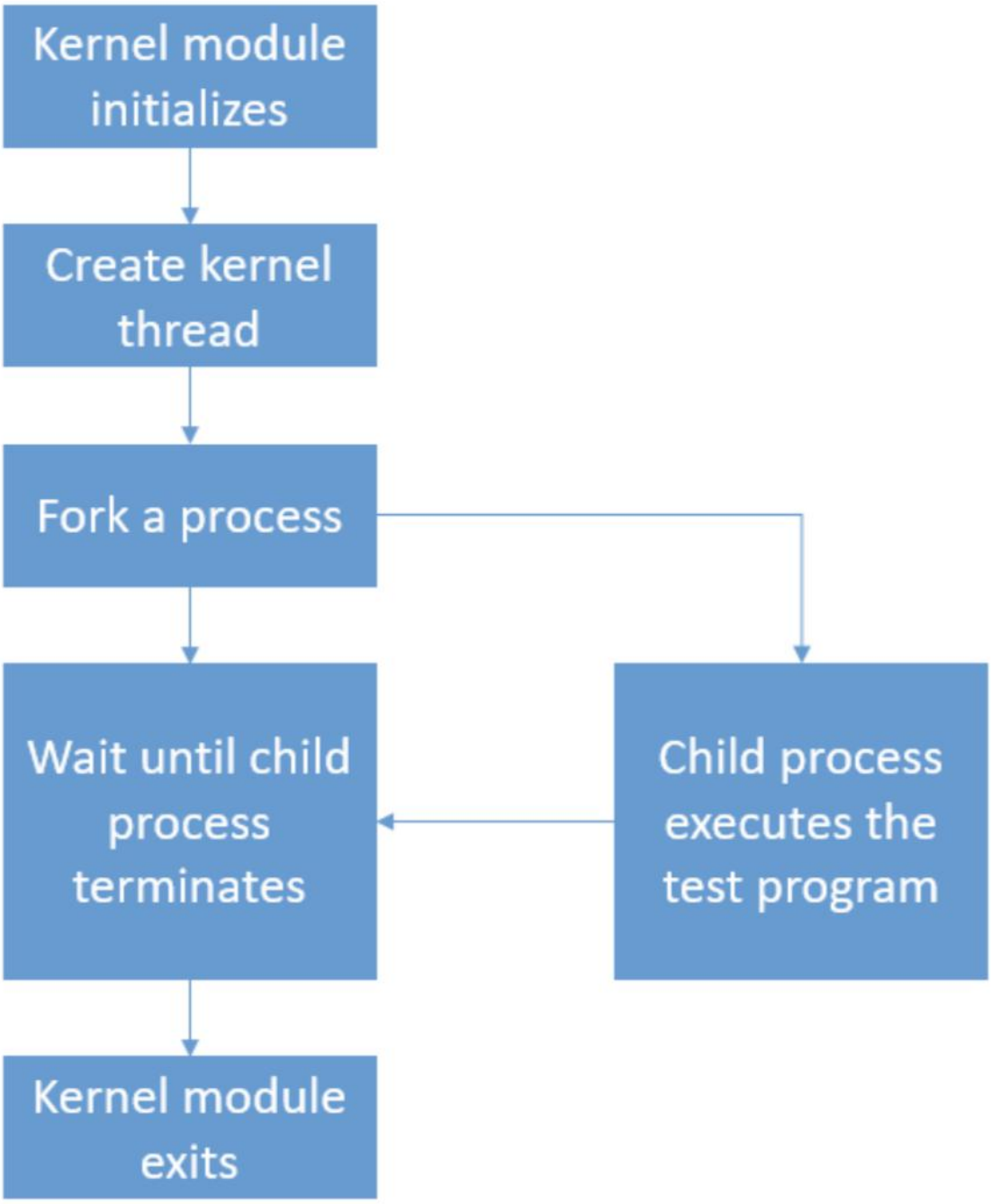
```
void my_wait(pid_t pid){
    int status;
    struct wait_opts wo;
    struct pid *wo_pid = NULL;
    enum pid_type type;
    type = PIDTYPE_PID;
    wo_pid = find_get_pid(pid);

    wo.wo_type = type;
    wo.wo_pid = wo_pid;
    wo.wo_flags = WEXITED;
    wo.wo_info = NULL;
    wo.wo_stat = (int __user*)&status;
    wo.wo_rusage = NULL;

    do_wait(&wo);

    if (*wo.wo_stat == 17){
        printk("[program2] : Normal termination with EXIT STATUS = %d\n",*wo.wo_stat);
    }
    // for stopped
    else if (*wo.wo_stat == 19){
        char *buff1;
        buff1 = "SIGSTOP";
        printk("[program2] : get %s signal\n", buff1);
        printk("[program2] : CHILD PROCESS STOPPED\n");
    }
    // for signaled
    else {
        char* buff1, *buff2;
        if (*wo.wo_stat == 1){
            buff1 = "SIGHUP";
            buff2 = "hangup";
        }
        else if (*wo.wo_stat == 2){
            buff1 = "SIGINT";
            buff2 = "interrupt";
        }
```

(The flowchart for program2 is shown below)



Functions

`void my_wait(pid_t pid)` function is used to let the parent wait for the child process while the child process is executing the test file. It is also able to catch and check the signal returned by the child process with *wo.wo_stat and print the signal in the kernel log.

1) It firstly, constructed a struct wo, which contains the pid type, wait options, pid, signal, termination status, information, resources usage and task wait queue from the child process.
2) Then, it constructed a struct pointer which is wo_pid.
3) It also let the parent process to wait for the child process to finish by using the do_wait() function from the kernel file.
4) It is also able to check the returned signal type by *wo.wo_stat value.
5) Finally, it frees the memory using put_pid() function.

`int my_fork(void *argc)` function is used to fork the child process and let the child process automatically execute the my_exec() function.

1) It will firstly set the default sigaction for the current process.
2) It will fork the child process and let the child process execute the my_exec() function by using the _do_fork function from the kernel file.
3) It will call my_wait function to let the parent process wait.

`static int __init program2_init(void)` function is used initialize the kernel module and create a kernel thread with the the __init program2_init(void) function.

1) It will firstly create the kernel thread with the `kthread_create()` function, it will automatically call the my_fork function function by sending the pointer of my_fork into the function as a parameter.
2) It will check if the kernel thread is started or not by using the `IS_ERR(task)` function.
3) If the kernel thread is created successfully, the function will wake up the process by `wake_up_process(task)` function.

`static void __exit program2_exit(void)` this function is used to exit the program2 when the process is finished.

`int my_exec(void)` this function is used to execute the external test file.

1) It will firstly get the path of test file and the system environment set.

2) It will get the filename using the `getname()` function from the kernel file and put it into the struct.
3) It will execute the test file using the `do_execve()` function and put the return value into the result variable.

Results

The result for program 2 is

```
[  335.480396] [program2] : module_init
[  335.482780] [program2] : module_init create kthread start
[  335.482780] [program2] : module_init Kthread starts
[  335.486748] [program2] : The child process has pid = 3262
[  335.486750] [program2] : This is the parent process,pid = 3260
[  335.491730] [program2] : child process
[  335.492026] [program2] : get SIGBUS signal
[  335.492026] [program2] : child process has bus error error
[  335.492027] [program2] : The return signal is 7
[  345.280112] [program2] : module_exit
[09/29/19]seed@VM:~/.../program2$
```

Conclusion: What I have learnt from this program

In this program I have learnt how to (1) fork a child process in kernel mode, (2) how to let the child process execute the external program in kernel mode, (3) how to let the parent process wait for the child process to terminate in the kernel mode and (4) how to handle the signal in kernel model.

(1) How to Fork a Child Process in Kernel Mode

We use do_fork function load a child process in the kernel module. After the function executed, it is loading the kernel fork.ko module.

The function will return the pid of child process and we can also use the current -> pid to get the pid of the current process.

(2) How to Let the Child Process Execute the External Program in Kernel Mode

We use the do_execute or _do_execute function to let the child process execute the external program in kernel mode. It will return the non-zero value if the execution succeeds otherwise it will return 0. The child process will not execute any lines in the original file after the execution function.

(3) How to Let the Parent Process Wait for the Child Process to Terminate in the Kernel Mode

It will need the struct wait_opts and do_wait function to let the parent process wait. Wait_opts is a struct which contains multiple of process information. It also needs to decrease the count and free the memory after do_wait function.

(4) How to Handle the Signal in Kernel Model

We use the *wo.wo_stat or the status form the wait_opts to determine the return signal

(5) Additional

In order to execute the exit function, we need to firstly remove the kernel module from the kernel.

Bonus Task

Introduction: Environment and Steps to Compile

This program is written to implement the functions which is able read multiple test programs and recursively fork the child process (the proceeding one is the parent of the process of the later input file). The program is able to return the pid of each process including the main process of "myfork" file and is also able to return the signal raised in each input file. The program is implemented on C programming language on ubuntu (ubuntu version 16.04 and kernel version v4.8.0-36-generic). The user should input multiple test external file (no less than one file and no more than 99 files) and the output will be the pid of each process and the signals raised in the files.

This program will only support 15 signals which are:
SIGHUP, SIGINI, SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGFPE, SIGKILL, SIGBUS, SIGSEGV, SIGPIPE, SIGALARM, SIGTERM, SIGSTOP, SIGCHIL

Your command in your terminal should be like:

```
sudo make clean
sudo make
./myfork filename1 filename2 filename3
```

(the filename here should be the name of your external test file, the command should be under the program1 folder)

For example, your input file should be:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
```

```
int main(int argc,char* argv[]){
    printf("------------CHILD PROCESS START------------\n");
    printf("This is the SIGKILL program\n\n");
    raise(SIGKILL);
    sleep(5);
    printf("------------CHILD PROCESS END------------\n");

    return 0;
}
```

For example, the corresponding output file should be:

```
[09/29/19]seed@VM:~/.../bonus$ ./myfork hangup normal8 trap
------------CHILD PROCESS START------------
This is the SIGTRAP program

This is normal8 program
------------CHILD PROCESS START------------
This is the SIGHUP program

The process tree: 25913->25914->25915->25916
The child process (pid=25916) of parent process (pid=25915) is stopped by signal
Its signal number is 5
Child process got SIGTRAP signal
Child was terminated by trap signal

The child process (pid=25915) of parent process (pid=25914) has normal execution
Its exit status = 0

The child process (pid=25914) of parent process (pid=25913) is stopped by signal
Its signal number is 1
Child process got SIGHUP signal
Child was terminated by hang up signal

Myfork process(pid=25913) execute normally
[09/29/19]seed@VM:~/.../bonus$
```

Flow: How Did I Design My Program

We implement the program with the idea of recursion and store every signal and pid in two arrays which are signal_array and pid_array with the size of 100.

1. We first read every input file and put the test files into a character array (exclude the myfork file). The integer variable argc is the number of the input file (include the myfork file).
2. If the input file is no more than 1, the program will return the “Invalid input” error and exit with error.

```c
int main(int argc,char *argv[])
{
    char *arg[argc];
    for (int i = 0; i < argc - 1; i++){
        arg[i]=argv[i+1];
    }
    arg[argc-1] = NULL;

    //check the invalid input
    if (argc <= 1){
        printf("Invalid input\n");
    exit(1);
    }

    //do the recursion
    int initial = 1;
    execution_recursion(initial, argc - 1, arg);

    //check the array (for debug)
    //display();

    //process tree
```

3. Then the main function will call the execution_recursion function with the initial index of 1 and totally input number of argc -1 and the file list arg.
4. We will do the recursion in the execution recursion function. Whenever the function is called it will fork a child process with the vfork() function since we need the parent and the child process to share the variables.
5. As long as the process is not for the last input file, the child process will continuously call the recursion execution function.
6. The parent process will catch the signal and pid of their child process and put the signal and pid value into the signal_array and pid_array respectively. Parent process also need to execute the input files in order to pass signal and pid information to their parent unless it does not have the parent process (index != 1).
7. When we reached the process for the last input file. We will execute the last file in the input file.

```c
void execution_recursion (int index, int numInput, char* arg[])
{
    pid_t pid;
    pid = vfork();
    int status;
    //if the fork failed
    if (pid == -1){
        perror("fork");
        exit(1);
    }
    else{
        if (pid == 0){
            //child process
            if (index == numInput){
                //for debug
                //printf("index for the last = %d\n",index);
                execve(arg[numInput - 1],arg,NULL);
            }
            else{
                //printf("for other children\n");
                execution_recursion(++index,numInput,arg);
                //execve(arg[index - 1],arg,NULL);
            }
        }
        else {
            waitpid(pid,&status,WUNTRACED);
            //for debug
        if (count != 0){
        index--;
}
        count++;
        //printf("count = %d",count);
            //printf("index for the parent = %d\n",index);
           // printf("status = %d\n",status);
            pid_array[index - 1] = pid;
            signal_array[index - 1] = status;

            if (index != 1){
                execve(arg[index - 2],arg,NULL);
            }
        }
    }
}
```

8. After we get the full pid_array and signal array, we print the process tree. The first pid value in the process tree should be the value of current process(myfork). After that we will print the element from the pid_array from the first element to the last.

```
//process tree
printf("The process tree: %d", getpid());
for (int i = 0; i < argc - 1; i++){
    printf("->%d",pid_array[i]);
}
printf("\n");
```

9. We continuously print the information of each process by fetching the value from the pid_array and signal_array

```
printf("The process tree: %d", getpid());
for (int i = 0; i < argc - 1; i++){
    printf("->%d",pid_array[i]);
}
printf("\n");

//show the child and parent process info
int num_previous, num_after;

for (int i = 0;i <= argc - 2; i++){
    if (i == argc - 2){
        num_previous = argc - 2 - i;
        num_after = getpid();
    }
    else{
        num_previous = argc - 2 - i;
        num_after = pid_array[argc - 3 - i];
    }

    // normal
    if (signal_array[num_previous] == 0){
        printf("The child process (pid=%d) of parent process (pid=%d) has normal execution\n",pid_array[num_previous],num_after);
        printf("Its exit status = 0\n\n");
    }
    else if (signal_array[num_previous] == 19){
        printf("The child process (pid=%d) of parent process (pid=%d) is stopped by signal\n",pid_array[num_previous],num_after);
        printf("Its signal number is %d\n",num_previous);
        printf("Child process got SIGSTOP signal\n");
        printf("Child process stopped\n\n");
    }
    else{
        char *buff1, *buff2;
        // SIGHUP
        if (signal_array[num_previous] == 1){
            buff1 = "SIGHUP";
            buff2 = "hang up";
        }
        // SIGINT
        else if (signal_array[num_previous] == 2){
            buff1 = "SIGINT";
            buff2 = "interrupt";
        }
        /* SIGQUIT */
        else if (signal_array[num_previous]== 3){
            buff1 = "SIGQUIT";
            buff2 = "quit";
        }
        /* SIGILL */
```

Results: Screen Shot of My Output Result

The result for not enough input

```
[09/29/19]seed@VM:~/.../bonus$ ./myfork
Invalid input
[09/29/19]seed@VM:~/.../bonus$
```

The result for single input hangup test file

```
[09/29/19]seed@VM:~/.../bonus$ ./myfork hangup
------------CHILD PROCESS START------------
This is the SIGHUP program

The process tree: 25530->25531
The child process (pid=25531) of parent process (pid=25530) is stopped by si
gnal
Its signal number is 1
Child process got SIGHUP signal
Child was terminated by hang up signal
Myfork process(pid=25530) execute normally
[09/29/19]seed@VM:~/.../bonus$
```

The result for two input files: hangup and normal8

```
[09/29/19]seed@VM:~/.../bonus$ ./myfork hangup normal8
This is normal8 program
------------CHILD PROCESS START------------
This is the SIGHUP program

The process tree: 25535->25536->25537
The child process (pid=25537) of parent process (pid=25536) has normal execu
tion
Its exit status = 0
The child process (pid=25536) of parent process (pid=25535) is stopped by si
gnal
Its signal number is 1
Child process got SIGHUP signal
Child was terminated by hang up signal
Myfork process(pid=25535) execute normally
[09/29/19]seed@VM:~/.../bonus$
```

The result for three input files: hangup normal8 and trap

```
[09/29/19]seed@VM:~/.../bonus$ ./myfork hangup normal8 trap
------------CHILD PROCESS START------------
This is the SIGTRAP program

This is normal8 program
------------CHILD PROCESS START------------
This is the SIGHUP program

The process tree: 25544->25545->25546->25547
The child process (pid=25547) of parent process (pid=25546) is stopped by si
gnal
Its signal number is 5
Child process got SIGTRAP signal
Child was terminated by trap signal
The child process (pid=25546) of parent process (pid=25545) has normal execu
tion
Its exit status = 0
The child process (pid=25545) of parent process (pid=25544) is stopped by si
gnal
Its signal number is 1
Child process got SIGHUP signal
Child was terminated by hang up signal
Myfork process(pid=25544) execute normally
```

Conclusion: What I have learnt from this program

In this program I learnt that

1. If we want to let the processes share the variables we can use vfork() function to fork a child process.
2. How to use the recursion to continuously fork the child process and execute the external file.
3. How to manipulate multiple processes.
4. With the vfork function the program will always execute the child process before the parent process.