

# CSC3150 A1 Report

---

## Part1: Problem Brief

---

The project is divided into two parts.

The tasks in part1 includes:

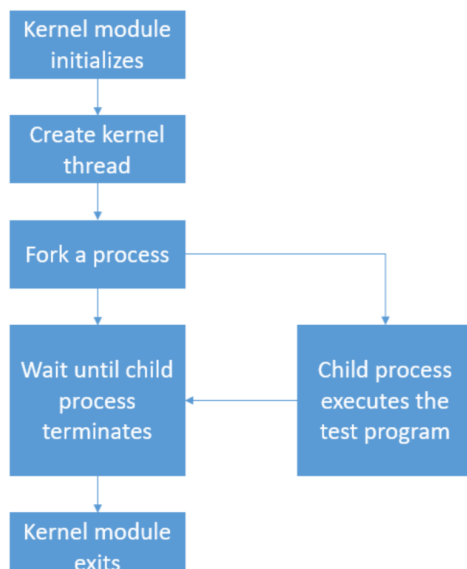
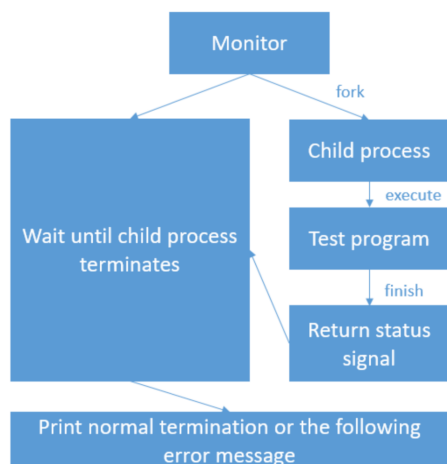
1. Fork a child process to execute test programs (15 of them)
2. Use wait() to let the parent process receives the SIGCHLD signal
3. Print out the termination information of child process (normal or abnormal)

The tasks in part2 includes:

1. Create a kernel thread and run my\_fork function
2. Fork a process to execute test.
3. Use do\_wait() to let the parent process wait for the child process
4. Print out pid of both parent and child processes
5. Catch the signal raised by the child process and print out related log
6. Recompile the Linux kernel source code to use its functions

## Part2: Overall Project Structure

---



## Part3: Program Design

### Program1:

Fork the child process

```
pid_t pid;
printf("Process start to fork\n");
pid = fork();
```

wait for SIGCHLD signal

```
/* wait for child process terminates */
waitpid(pid, &status, WUNTRACED);
printf("Parent process receives the SIGCHLD signal\n");
```

Parent process analyses exit status and print out info

```
if (WIFEXITED(status)) {
    printf("Normal termination with EXIT STATUS = %d\n", status);
} else if (WIFSIGNALED(status)) {
    printf("CHILD EXECUTION FAILED: %d\n", WTERMSIG(status));
} else if (WIFSTOPPED(status)) {
    printf("CHILD PROCESS STOPPED: %d\n", WSTOPSIG(status));
} else {
    exit(0);
}
```

Child process execute the test program

```
printf("I am the child ");
printf("my pid: %d\n", getpid());
printf("Child process start to execute test program:\n");

/* execute test program */
execve(arg[0], arg, NULL);

printf("Continue to run original child process!\n");
perror("execve");
exit(SIGCHLD);
```

### Program2

Create a kernel thread and run my\_fork

```
task = kthread_create(&my_fork, NULL, "MyThread");

if (!IS_ERR(task)) {
    printk("kthread starts\n");
    wake_up_process(task);
}
```

Fork a process and print out pid

```
pid_t pid;
pid = kernel_clone(&args);

if (pid == -1) {
    printk("fork failed");
    return -1;
}

printk("[program2] : The child process has pid = %d\n", pid);
printk("[program2] : This is the parent process, pid = %d\n", (int)
current->pid);
```

Execute the test program

```
int my_exec(void) {
    int result;
    const char __user path[] = "/tmp/test";

    /* execute a test program in child process */
    printk("[program2] : child process");

    result = do_execve(getname_kernel(path), NULL, NULL);

    if (!result) {
        return 0;
    } else {
        do_exit(result);
    }
}
```

Wait for child process termination

```
void my_wait(pid_t pid) {
    int a;
    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 | WUNTRACED;
    wo.wo_info = NULL;
    wo.wo_rusage = NULL;

    a = do_wait(&wo);

    my_output(wo.wo_stat);
    put_pid(wo_pid);

    return;
```

```
}
```

Catch the signal and printed out message

```
void my_output(int signal) {
    switch (signal) {
        case 1:
            printk("[program2] : get SIGHUP signal\n");
            printk("[program2] : child process is hung up\n");
            printk("[program2] : The return signal is 1\n");
            break;
        case 2:
            printk("[program2] : get SIGINT signal\n");
            printk("[program2] : terminal interrupt\n");
            printk("[program2] : The return signal is 2\n");
            break;
        case 131:
            printk("[program2] : get SIGQUIT signal\n");
            printk("[program2] : terminal quit\n");
            printk("[program2] : The return signal is 3\n");
            break;
        case 132:
            printk("[program2] : get SIGILL signal\n");
            printk("[program2] : child process has illegal instruction
error\n");
            printk("[program2] : The return signal is 4\n");
            break;
        .....

        default:
            printk("[program2] : a signal not contained in the signal
list");
            printk("[program2] : The return signal is %d\n", signal);
            break;
    }
    return;
}
```

Recompile the kernel in order to use kernel function

```
extern struct filename *getname_kernel(const char *filename);
extern pid_t kernel_clone(struct kernel_clone_args *args);
extern int do_execve(struct filename *filename,
                    const char __user *const __user *__argv,
                    const char __user *const __user *__envp);
extern long do_wait(struct wait_opts *wo);
```

## Part4: Program Environment

Virtual machine application: virtual box 6.1.32

The program is run on a Ubuntu 16.04 LTS operation system, with kernel version 5.10.146.

Compiler: gcc version 5.4.0

Before run the program2, rewrite the kernel source code.

Find these four functions in the kernel source code:

1. `getname_kernel` in `./fs/namie.c`
2. `do_execve` in `./fs/exec.c`
3. `do_wait` in `./kernel/exit.c`
4. `do_fork` in `./kernel/fork.c`

Add `EXPORT_SYMBOL(*Function_name*);` below the corresponding function implementations.

Then recompile the kernel.

## Part5: How to run my program

---

### Program1:

```
cd ./program1
make
./program1 ./filename
```

### Program2:

```
cd ./program2
gcc test.c -o test
sudo make
insmod program2.ko
rmmod program2
dmesg | tail -n 10
```

(You might need to export functions in linux kernel and recompile first)