Shubham Sharma
Section L
CPRE 308

Lab 2: Report

In this lab we learn about how processes are created. The relationship of the parent and the child processes. The use case of terminating the process. And understanding the how the kernel scheduler functions in linux.

Experiments:

Experiment 3.1:

Screenshot of "ps -l"

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 354934 6207  6202  0  80   0 - 31790 do_wai pts/0    00:00:00 bash
0 S 354934 6269  6207  0  80   0 -  1053 hrtime pts/0    00:00:00 output1
0 S 354934 6278  6207  0  80   0 -  1053 hrtime pts/0    00:00:00 output2
0 R 354934 6287  6207  0  80   0 - 38300 -      pts/0    00:00:00 ps
0 S 354934 6288  6207  0  80   0 - 27576 pipe_w pts/0    00:00:00 less
(END)
```

Here we can see the following items:

- Output: A table with information of the running processes in the terminal window
- Process Name: This on the column CMD. There output names for the two processes are output1 and output2 as seen in the column
- Process State: As seen from the screenshot below, we can find out the process state codes by looking through the man page. According the the man page, the two created processes are 'S' which means that they are in 'interruptible sleep'

```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process:

               D    uninterruptible sleep (usually IO)
               R    running or runnable (on run queue)
               S    interruptible sleep (waiting for an event to complete)
               T    stopped by job control signal
               t    stopped by debugger during the tracing
               W    paging (not valid since the 2.6.xx kernel)
               X    dead (should never be seen)
               Z    defunct ("zombie") process, terminated but not reaped by its parent

       For BSD formats and when the stat keyword is used, additional characters may be displayed:

               <    high-priority (not nice to other users)
               N    low-priority (nice to other users)
               L    has pages locked into memory (for real-time and custom IO)
               s    is a session leader
               l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
               +    is in the foreground process group
```

- Process ID (PID): The PID of output1 is 6269 and the PID of output2 is 6278 as seen in the PID column
- Parent process ID (PPID): The PPID of output1 and output2 is 6207. They have the same PPID as they are created by the same process.

Screenshot of the repeated experiment:

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 354934 6207  6202  0  80   0 - 31790 do_wai pts/0    00:00:00 bash
0 S 354934 6982  6207  0  80   0 -  1053 hrtime pts/0    00:00:00 output1
0 S 354934 6991  6207  0  80   0 -  1053 hrtime pts/0    00:00:00 output2
0 R 354934 7058  6207  0  80   0 - 38300 -      pts/0    00:00:00 ps
0 S 354934 7059  6207  0  80   0 - 27576 pipe_w pts/0    00:00:00 less
(END)
```

When comparing the repeated experiment we can see that a **2 main** things have changed.
The PID of output1 and the PID of output2. This is because we ran a new process in the repeated experiment which created new processes, due to this they were given a new PID.

Other fields did not change as much as it is running the same instance of the parent process. For example we can see that the PPID of all of the process is the same in the repeated experiment. This is because the parent process was not stopped, hence it still is running the same instance.

Screenshot demonstrating the use of the 'killall' command:

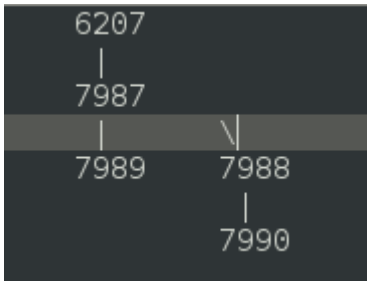
```
[shubham@co2048-13 lab2]$ killall output1 output2
[1]-  Terminated              ./output1
[2]+  Terminated              ./output2
[shubham@co2048-13 lab2]$
```

Here we terminated 2 processes using the 'killall' command. Since I created these processes in the terminal the system gave me permission to kill the processes.

Experiment 3.2:
Output of the program:

```
[shubham@co2048-13 3.2]$ gcc -o output fork.c
[shubham@co2048-13 3.2]$ ./output
Process 7987's parent process ID is 6207
Process 7989's parent process ID is 7987
Process 7988's parent process ID is 7987
Process 7990's parent process ID is 7988
```

Process Tree:



In the tree we can see that the total number of processes are 4. This makes sense as we call fork() only twice. As fork() by definition is creates a new process, which is called child process, which runs concurrently with the parent process. So on running 2 fork() commands will result in $2^2 = 4$ total processes.

PPID for a process is 1:
This mean that the parent process has already completed running and exited by the time the third child checked for the parent's PID. That would have caused the child to be re-parented under init, due to which it has the parent process ID 1.

Experiment 3.3:

Screenshot of the completed program:

```
#include <stdio.h>

int main() {
        int ret;
        ret = fork();
        printf("The ret value: %d\n", ret);
        if (ret == 0) {
        /* this is the child process */
        printf("The child process ID is %d\n", getpid());
        printf("The child's parent process ID is %d\n", getppid());
        } else {
        /* this is the parent process */
        printf("The parent process ID is %d\n", getpid());
        printf("The parent's parent process ID is %d\n", getppid());
        }
        sleep(2);
}
```

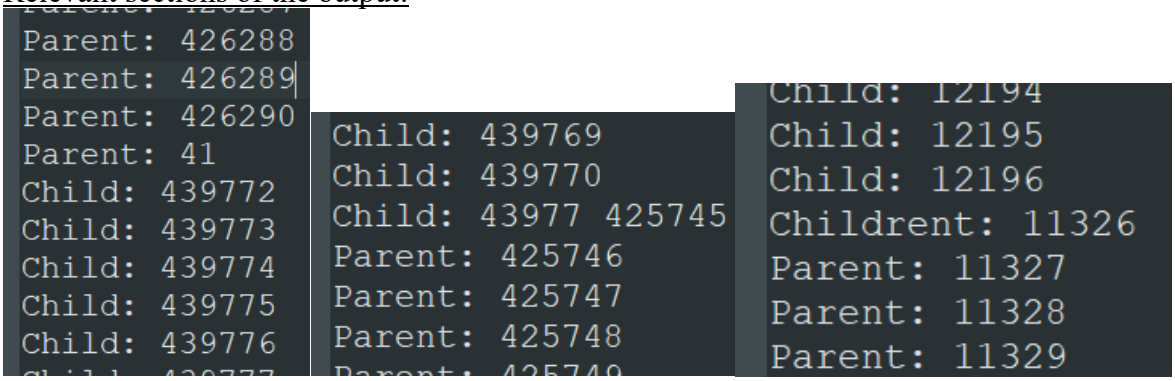
Screenshot of the ouput:

```
[shubham@co2048-13 3.3]$ gcc -o output fork_3.3.c
[shubham@co2048-13 3.3]$ ./output
The parent process ID is 13987
The parent's parent process ID is 6207
The child process ID is 13988
The child's parent process ID is 13987
```

It is useful to have fork return different values for the parent and the child as it helps the programmer identify which is which, since fork() runs the same code in different threads. For example, if we need to run the same website in different servers, the use of fork would be useful. In this case it is important to know which server the user is accessing, hence the value of PID an PPID is useful for keeping track of the child and the parent.

Experiment 3.4:

Relevant sections of the output:



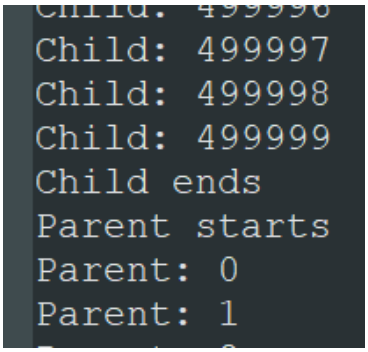
Here we see the effect of time slicing. It is a short interval of time allotted to each program in a multitasking or timesharing system. Time slices are typically in milliseconds. Due this, we can see two different outputs conflicting each other in the output above. For example, we can see the output "Childrent: 11326", which is the effect of the time slicing timing out on one thread and moving from the child process to the parent process. Which explains the odd outputs on the screenshots above. In linux these time slicing is decided by the kernel scheduler which handles CPU allocation for executing purposes and aims to maximize overall CPU utilization.

Experiment 3.5:

In the code we notice a line called wait(NULL);

The wait(NULL) command will block parent process until any of its children has finished. If child terminates before parent process reaches wait(NULL) then the child process turns to a zombie process until its parent waits on it and its released from memory. [Source]

So due to the wait command the parent process is placed on hold until its child is finished running. This helps solve the issue with time slicing in experiment 3.5. As there is no sign of bad formatted text in the output as seen in experiment 3.4. The output will be the child loop finishing 500000 iterations and ONLY then the parent process resumes. This is validated from the screenshot below.



Experiment 3.6:

The program appears to have an infinite loop only in the child process. The parent process terminates the child, which in turn stops the infinite loop.

In this program each iteration of the loop takes 1/100 of a second (0.01 sec). And the parent is asleep for 10 seconds. So eventually the child program will only be able to execute a total of 0.01 x 10 iterations, which is equal to 1,000 iterations before the child process ends.

The child process is only able to complete a number of iterations close to 1,000 but not equal to a 1,000. This is because the program is also spending time on other commands and task. Such as, printing the output, incrementing the value of i every loop, etc. This takes up time till 10 seconds before the child could finish 1,000 iterations.

```
Child at count 866
Child at count 867
Child at count 868
Parent sleeping
Child has been killed. Waiting for it...
done.
```

Experiment 3.7:

We can define exec() as the family of functions replaces the current process image with a new process image. [From the man page]

execl("/bin/ls", "ls", NULL);

This line of c code is the equivalent of the ‘ls’ command which displays the files in the current working directory. Since execl created a new process, replacing the current one, that in turn terminals the current program. Hence the next print statement is never executed.

```
[shubham@linux-1 3.7]$ ls
execve.c  output
[shubham@linux-1 3.7]$ ./output
execve.c  output
[shubham@linux-1 3.7]$
```

The printf statement will be executed when ecec() throws an error. Like, if the bin/ls file is not found or if the command is not part of linux or the end arg is not a NULL pointer. This would prevent it from starting a new process so it will just continue the current program.

Experiment 3.8:

I don’t know ¯_(ツ)_/¯

Extra Points:

When a process dies on Linux, it isn't all removed from memory immediately — its process descriptor stays in memory. The process's status becomes EXIT_ZOMBIE and the process's parent is notified that its child process has died with the SIGCHLD signal. The parent process is then supposed to execute the wait() system call to read the dead process's exit status and other information. This allows the parent process to get information from the dead process. After wait() is called, the zombie process is completely removed from memory. [Source]

Program that creates a zombie process:

```c
// A C program to demonstrate Zombie Process.
// Child becomes Zombie as parent is sleeping
// when child process exits.
// [SOURCE]:https://www.geeksforgeeks.org/zombie-and-orphan-processes-in-c/

#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
}
```

The command "ps -l" allows us to see the running processes in the current instance.

```
[shubham@linux-1 ExtraPoints]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 354934 1872 14935  0  80   0 -  1052 hrtime pts/0    00:00:00 output
1 Z 354934 1873  1872  0  80   0 -     0 do_exi pts/0    00:00:00 output <defunct>
0 R 354934 1913 14935  0  80   0 - 38302 -      pts/0    00:00:00 ps
0 S 354934 14935 14932  0 80   0 - 31796 do_wai pts/0    00:00:00 bash
```

We can see the two instances of the program running, one is the child one is the parent. In the column 'S' shows the status of all the processes. Here was can see one instance of the output is 'Z' which according to the Linux man page means it is a zombie process. And this is how we can verify that the zombie process is created.