Report.md

# Lab 2 Report

Name: Reid Schneyer

Section: 9

University ID: 944014098

*If there are any issues with viewing this PDF, you should also be able to read it at https://gist.github.com/RSchneyer/a0aa609511229027f4bbe9618bcdf827*

## Summary

*In this lab, I leaned the basics of Unix processes, and different C language functions that interact with them. In part 3.1, I learned how to use the `ps` command to view running processes and their basic information. In 3.2, I learned the how the `fork()` C function worked, and how it created the child processes in a tree format. Section 3.3 showed that the return value of `fork()` could be used to check if a process was the parent process, or if it was a child process. Section 3.4 was probably the most confusing for me, but I learned that even though both a parent and child process can be ran at the same time on separate cores, they still access the same file, which can lead to cutoff outputs.*

*Section 3.5 showed that using `wait()` helped sync the parent and child processes, and Section 3.6 showed that `kill()` could be used to end a process, potentially breaking out of an infinite loop. Finally, 3.7 used the `execl()` function to execute a bash command, and if that command fails, the code after it gets ran. 3.8 was also a little confusing, but I learned how to use the different potential exit values of both a child process, and the `main()` function.*

## Lab Questions

### 3.1 Process Table

**In the report, include the relevant lines from `ps -l` for your programs, and point out the following items:**

- process name
- process state (decode the letter!)
- process ID (PID)
- parent process ID (PPID)

```
reid@REID-DESKTOP:/mnt/d/Files/Documents/School/ISU_Spring_2021/CPRE_308/CPRE_308_S21/Lab_02$ Process ID is: 141
Parent process ID is: 111
ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   111   110  0  80   0 -  4554 -      tty2     00:00:00 bash
0 S  1000   140   111  0  80   0 -  2634 -      tty2     00:00:00 three_one
0 S  1000   141   111  0  80   0 -  2634 -      tty2     00:00:00 three_one
0 R  1000   142   111  0  80   0 -  4642 -      tty2     00:00:00 ps
```

*Name: three_one; State: Interruptable Sleep; PID: 140, 141; PPID: 111 for both*

**Repeat this experiment and observe what changes and doesn't change**

```
ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   111   110  0  80   0 -  4587 -      tty2     00:00:00 bash
0 S  1000   144   111  0  80   0 -  2634 -      tty2     00:00:00 three_one
0 S  1000   145   111  0  80   0 -  2634 -      tty2     00:00:00 three_one
0 R  1000   146   111  0  80   0 -  4642 -      tty2     00:00:00 ps
```

*PID changes. Name, State, and PPID remain the same*

**Find out the name of the process that started your programs (the parent). What is it, and what does it do?**

*I found the parent with* `ps -p <PPID> -o comm=`

```
reid@REID-DESKTOP:/mnt/d/Files/Documents/School/ISU_Spring_2021/CPRE_308/CPRE_308_S21/Lab_02$ ps -p 111 -o comm=
bash
```

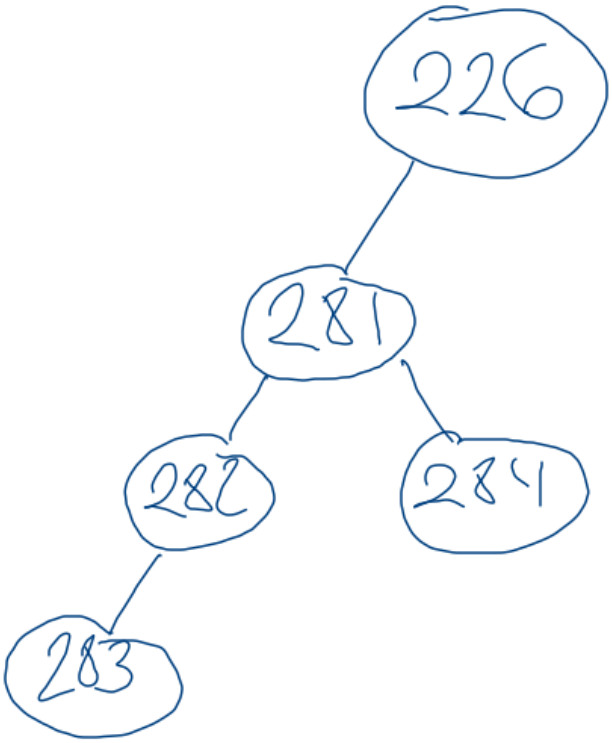
*Bash is the default shell, and it executes commands from the user or from a file.*

## 3.2

**Include the output from the program.**

```
/mnt/.../CPRE_308_S21/Lab_02$ ./three_two
Process 281's parent process ID is 226
Process 282's parent process ID is 281
Process 283's parent process ID is 282
Process 284's parent process ID is 281
/mnt/.../CPRE_308_S21/Lab_02$
```

**"Draw" the process tree (the nodes should be PIDs)**




**Explain how the tree was built in terms of the program code**

*Node 226 is the bash process, and running the program creates its child, proc 281. Process 281 then calls the first* `fork()`*, which creates its child, proc 282. The process 282 calles the next* `fork()`*, which creates proc 283 as a child. Proc 283 then returns, and proc 281 calls the second* `fork()`*, which creates proc 284 as a child of 281. Proc 284 returnsm and then the program calls the* `usleep()`*,* `printf()`*, and* `sleep()` *functions, which create no further processes.*

**Try the program again without `sleep(2)`. Explain what happens when the sleep statement is removed. You should see processes reporting a parent PID of 1. Redirecting output to a file may interferewith this, and you may need to run the program multiple times to see it happen.**

```
/mnt/.../CPRE_308_S21/Lab_02$ ./three_two
Process 324's parent process ID is 226
Process 327's parent process ID is 325
Process 326's parent process ID is 1
Process 325's parent process ID is 1
```

*When the* `sleep()` *statement is removed, the parent process doesn't wait for the child process to complete, and exits before the child. Since the child is now an orphan, it gets adopted by the* `init` *process (PID=1)*

## 3.3) The `fork()` syscall, continued

**Include the (completed) program and its output.**

```c
#include <stdio.h>
int main(){
    int ret;
    ret = fork();
    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);
    return 0;
}
```

```
/mnt/.../CPRE_308_S21/Lab_02$ ./three_three
The parent process ID is 380
The child process ID is 381
The parent's parent process ID is 226
The child's parent process ID is 380
```

**Speculate why it might be useful to have fork return different values to the parent and child. What advantage does returning the child's PID have? Keep in mind that often the parent and child processesneed to know each other's PID.**

`fork()` *returns the child's PID to the parent, since the child can already access its own PID and PPID with* `getpid()` *and* `getppid()`.

### 3.4 Time Slicing

**Include small (but relevant) sections of the output**

```
349    Parent: 348
350    Parent: 349
351    ParentChild: 0
352    Child: 1
353    Child: 2
```

```
8710    Child: 4528
8711    Child: 452t: 4181
8712    Parent: 4182
```

**Make some observations about time slicing. Can you find any output that appears to have been cut off? Are there any missing parts? What's going on(mention the kernel scheduler)?**

*There are missing parts in the output because the processes aren't waiting, and can cut eachother off as a result*

## 3.5 Process synchronization using `wait()`

**Explain the major difference between this experiment and experiment 4. Be sure to look up whatwait does.**

*In this experiment, the child process loop all the way from 0 to 499999 without any missing parts or interruptions, then the parent process does the same, again without any missing parts or interruptions*

## Signals using `kill()`

**The program appears to have an infinite loop. Why does it stop?**

*The program eventually stops when the parent is done sleeping. Once the parent has finished sleeping, it uses `kill()` to kill the child process, ending the "infinite" loop*

**From the definitions of sleep and usleep, what do you expect the child's count to be just before it ends?**

*I would expect the child's count to be 1000 just before it ends, since it waits for 0.01s each loop, and the parent is asleep for 10s.*

**Why doesn't the child reach this count before it terminates?**

*The child does not reach 1000 before it terminates because it takes time to make the fork and run the other lines in the loop, which cuts down on the amount of iterations the child can make.*

## 3.7 The `execve()` family of functions

**Run the following program and explain the results.**

*When the program is ran, it lists out the contents of the directory that it gets called from using the `ls` command.*

**Under what conditions is the printf statement executed? Why isn't it always executed?**

*The printf statement is executed if the command in `execl()` is unsuccessful. For example, changing the line to*

```
execl("/bin/cd", "cd dne", (char *)NULL);
```

*(where dne is a sub-directory that does not exist in the directory that the program is called from) will instead execute the printf statement.*

## 3.8 The return value of `main()`

**What is the range of possible exit status values printed for the child process?**

*The range of possible exit values for the child process is from 0-255*

**What is the signal value the child process uses to terminate itself, as determined and printed by the parent process?**

*The signal value the child process uses to terminate itself is 15, which is what the macro `SIGTERM` evaluates to.*

**When do you think the return value of `main()` would be useful? Hint: look at the commands `true` and `false`.**

*The return value of `main()` would be useful in telling whether or not the program ran/worked. `true` and `false` could be used with the return value of main to check if the C program worked*