EECS 338 Homework 3: Concurrency using fork()

General requirements:

- See posted due date.
- Create a typed report for discussion questions and program output.
- Upload a single, compressed file (e.g. zip) to Canvas that contains all required files (programs + report).
- All work should be your own, as explained in the Academic Integrity policy from the syllabus.
- Include either a single makefile that compiles all programs or a separate makefile for each.

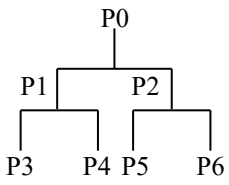
1. Write a custom shell program as described in the textbook: Programming Project 1, Part 1 (page 157). Do not do Part 2 (history feature) unless you wish to do so for fun. You are not required to detect "&" for background processes unless you wish to do so for fun. The minimum requirements are:

   - Use the basic design provided in simple-shell.c that was provided in class and appears in Fig. 3.36.
   - Parse the user input appropriately. You may use the approach provided in execvp_demo.c that was provided in class, or you may use a different approach that is more robust.
   - Fork a new process for every command (except "exit") in order to provide separate process control.
   - Use wait() for the parent process.

   Below is an example of how your output might look (user input is in **bold**):

```
EECS338> echo This is my own shell!
This is my own shell!
EECS338> ls shell.c -all
-rwxr-xr-x 1 chrisf None 1146 Sep 14 22:25 shell.c
EECS338> ./sleep
Sleeping
EECS338> exit
```

2. Write separate programs, as described below, that create 7 processes according to the parent/child hierarchy shown on the right, where P0 is a parent to P1 and P2, etc. *Before each parent terminates, it must wait until all of its children have terminated.* When created, every process should print its index (0 – 6), its PID, and its parent's PID. Each process does no work and prints a message indicating when it is done. **In your report, include your program output and a *brief description of how you designed the program.***



   a. Use a full, nested if-else structure after every fork() instruction such that one branch handles the parent and one branch handles the child.

   b. Use if statements without "else" and no more than 2 if-levels deep (maximum of one if inside another if). Note that P3 – P6 can also call "wait" even though they do not have children (has no effect).

   The only requirement for output ordering is that it should demonstrate the necessary creation and termination of parents relative to children. For example, P1 should clearly be created some time before P3 and P4, and it should clearly terminate some time after they do. However, P1 output does not need a particular order relative to P2 or its children. OPTIONAL: If you wish, you may create a helper function to perform tedious tasks, such as printing (or more!). Note that defining a C function after main() requires a prototype (https://computer.howstuffworks.com/c13.htm).

   Below are two examples of possible output. Theoretically, both are possible for both programs. You may

get slightly different ordering every time you run your program.

Example #1:	*Example #2:*
`Created P0 (PID 207). Parent is PID 4.`	`Created P0 (PID 382). Parent is PID 4.`
`Created P1 (PID 208). Parent is PID 207.`	`Created P1 (PID 383). Parent is PID 382.`
`Created P2 (PID 209). Parent is PID 207.`	`Created P3 (PID 384). Parent is PID 383.`
`Created P3 (PID 210). Parent is PID 208.`	`Created P2 (PID 385). Parent is PID 382.`
`P3 is done.`	`Created P4 (PID 386). Parent is PID 383.`
`Created P4 (PID 211). Parent is PID 208.`	`P3 is done.`
`P4 is done.`	`P4 is done.`
`P1 is done.`	`P1 is done.`
`Created P5 (PID 212). Parent is PID 209.`	`Created P5 (PID 387). Parent is PID 385.`
`Created P6 (PID 213). Parent is PID 209.`	`Created P6 (PID 388). Parent is PID 385.`
`P5 is done.`	`P5 is done.`
`P6 is done.`	`P6 is done.`
`P2 is done.`	`P2 is done.`
`P0 is done.`	`P0 is done.`

Tip (output order): Don't forget to use fflush(stdout) to get reasonable output ordering. This was demonstrated in sleep.c in HW #1 and discussed at the end of a lecture exercise on using fork().

Tip (self identity): In part (b), processes can decide what to do based on their own identities. You can use several if statements, where each if statement checks its process's identity. Rather than using the PID assigned by the OS, an easier way for a process to track its own identity can be to use a variable that stores a value from 0 to 6. For example, if process #0 makes child process #2, that child process can immediately set its identity to be 2. Then any actions that P2 needs to perform can be done after checking for this custom identity value of 2. In this way, you don't need to worry about keeping track of the PID assigned by the OS.

3. Write separate programs, as described below, where the user enters the number of processes to be created. You must use a "for" loop to create the processes. *Before each parent terminates, it must wait until all of its children have terminated.* When created, every process should print its index (0, 1, ...), its PID, and its parent's PID. Each process does no work and prints a message indicating when it is done. **In your report, include your program output.**

   a. Create the processes as a series of *parent/child* pairs such that each new process is a child of the previous process. Below is an example of possible output (user input in **bold**):

```
How many processes? 4
Created P0 (PID 207). Parent is PID 9.
Created P1 (PID 208). Parent is PID 207.
Created P2 (PID 209). Parent is PID 208.
Created P3 (PID 210). Parent is PID 209.
P3 is done.
P2 is done.
P1 is done.
P0 is done.
```

   b. The first process is the parent, and all of the remaining processes are children of that first process. The only requirement for the output order is that the parent is clearly created first and terminates last. Below is an example of possible output (user input in **bold**). Note that child terminations do not need to be synchronized.

```
How many processes? 4
Created P0 (PID 207). Parent is PID 9.
Created P1 (PID 208). Parent is PID 207.
Created P2 (PID 209). Parent is PID 207.
P2 is done.
Created P3 (PID 210). Parent is PID 207.
P1 is done.
P3 is done.
P0 is done.
```

Tip (where to start?): You might try just using fork() in a for loop and printing the PIDs to see what happens (and think about why). This would be similar to the lecture exercise we did in class where every process kept calling fork() again and again. Then think about what each parent or child should do inside the loop.

Tip (debugging): Since every process will run the loop separately, it may be challenging to understand what each process is doing. To help make sense of it, you might draw a table by hand with one column for each process, and track the value of all the variables in each process, including the for-loop iterator variable.