Introduction to Multiprocessing (`fork()` `wait()` `waitpid()`)

See *Advanced Programming in the Unix Environment* chapter 8.1, as well as `man 2 fork`, `man 3am fork`, `man 2 wait`, `man 3  wait`.

A process is a running set of instructions that you might commonly call a "program". However, a program (which is loosely defined as a piece of software created to solve a problem) can have many processes, all of which must be uniquely identified. To achieve this, each process (regardless of its position as a standalone process in a program or a process among many in a program) has a procces identifier (PID).

Each process consists of its own memory space, which means it has its own stack, heap, and all other segments of memory. One process does not have access to another process's address space. This is important for what comes later.

When you run a program, the operating system creates a processes, and starts your program with the `main()` function. Now, if the operating system can create new processes, why can't you? Well, it turns out you can! This can be achieved with the system call `fork()`. `fork` takes no parameters, and its functionality is simple: it creates an exact copy of the process from which it was called. This includes the stack, the heap, and any additional associated data. Now, it should be noted that modern implementations of `fork` don't create a full copy, but rather use a technique called *copy-on-write*, but this is beyond the scope of this course. For all intents and purposes, you can consider the new process as having its own complete copy of the address space.

When a process is successfully created, the new process is called the `child` process, and the original process is called the `parent` process.

A call to `int pid = fork();` can end in the following situations:

1. `pid < 0`: An error has occurred, and no process has been created.
2. `pid >= 0`: A process has been created.
    1. if `pid > 0`, then the current process is the parent process, and the PID of the child process is stored in `pid`.
    2. if `pid == 0`, then the current process is the child process.

The following snippet describes the behavior:

```c
int pid = fork();

if (pid < 0) {
    fprintf(stderr, "An error occurred!\n");
} else if (pid > 0) {
    printf("This is the parent process, and the child process's PID is %d.\n", pid);
} else {
    // pid == 0
    printf("This is the child process.\n");
}
```

If the above code is run, either it will print:

```
An error occurred!
```

OR it will print:

```
This is the parent process, and the child process's PID is <child PID>.
This is the child process.
```

OR it will print:

```
This is the child process.
This is the parent process, and the child process's PID is <child PID>.
```

Huh? Why could it be either order? That's because, once the child process is created, there is no guarantee which will run first, the parent or the child. Don't believe me? Try it! Compile it to some program. Say the program is called `fork_test`. Execute the following in Bash:

```
for _ in {1..100}; do
    echo "---------------------------------------------";
    ./fork_test;
done
```

Chances are, you will see each of the two orders at least once, and if nothing goes wrong, you will never see the error message.

Now, there is one thing we have seen that remains shared between the parent and the child, and that is file descriptors. If the parent opens a file descriptor before calling `fork`, after the call to `fork` the child will have access to the opened file. This can lead to some fun synchronicity issues, and must be taken into account. What happens if the parent and child processes both read from the file at the same time? Write yourself a test program to see what happens.