Section 8: Processes

- ~~What is a process~~
- Creating processes
- Fork-Exec

Creating New Processes & Programs

- **fork-exec model:**
  - **`fork()`** creates a copy of the current process
  - **`execve()`** replaces the current process' code & address space with the code for a different program

- **`fork()` and `execve()` are *system calls***
  - Note: process creation in Windows is slightly different from Linux's fork-exec model

- **Other system calls for process management:**
  - **`getpid()`**
  - **`exit()`**
  - **`wait()` / `waitpid()`**

`fork`: Creating New Processes

- **`pid_t fork(void)`**
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's process ID (**`pid`**) to the parent process

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

- **`fork` is unique (and often confusing) because it is called *once* but returns *twice***

Understanding fork

Process n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Child Process m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` *Which one is first?* `hello from child`

Fork Example

- **Parent and child both run the same code**
  - Distinguish parent from child by return value from **`fork()`**
  - Which runs first after the **`fork()`** is undefined
- **Start with same state, but each has a *private* copy**
  - Same variables, same call stack, same file descriptors…

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```