Week 3 - Monday

COMP 3400

Last time

- What did we talk about last time?
- Kernel
- System calls
- Process lifecycle

Questions?

Assignment 2

Project 1

Process Life Cycle

Creating processes in code

- Processes are, of course, created when you run a program from the command line
- However, you can also create processes from within a program, using calls to special functions
- The **`fork()`** function creates a new processes that's exactly the same as the current process
- The **`exec()`** function allows you to replace the current process with another program
- Each process has a unique ID, its process ID or PID
  - **`getpid()`** returns the PID of the current process
  - **`getppid()`** returns the PID of the current process's parent process

Using `fork()`

- The **`fork()`** function is pretty crazy!
  - When you call it, the process you're inside of keeps running
  - And another process spawns at exactly the same point in code
  - Both processes have *exactly* the same memory layout
  - The only difference is that **`fork()`** returns the child PID for the original process and o if you're the process that just got forked

```
pid_t child_pid = fork ();

if (child_pid < 0)
  printf ("ERROR: No child process created\n");
else if (child_pid == 0)
  printf ("Hi, I'm the child!\n");
else
  printf ("Parent just gave birth to child %d\n", child_pid);
```

Fork bombing

- If you call **`fork()`** in a loop, you will quickly create too many processes and slow/crash your computer
- Each **`fork()`** creates a new process, but the old process keeps running
- The following code will have four prints:

```
pid_t first_fork = fork ();

// Original parent and child create new processes
pid_t second_fork = fork ();

// This line prints four times
printf ("Hello from %d!\n", getpid ());
```

Running another program

- Sometimes it's useful to fork a clone of yourself
- Other times, you want to run another program
- In those situations, you first fork yourself and then have your child call something from the **`exec()`** family of functions:

Function	Description
`execl(char *path, char *arg0, ..., NULL)`	Executes the program with the given path
`execle(char *path, char *arg0, ..., NULL, char* envp[])`	Executes the program with the given path and environment variables
`execlp(char *file, char *arg0, ..., NULL)`	Executes the program by looking it up in the current **`PATH`**
`execv(char *path, char *argv[])`	Like **`execl()`** but command-line arguments are in an array
`execve(char *path, char *argv[], char *envp[])`	Like **`execle()`** but command-line arguments are in an array
`execvp(char *file, char *argv[])`	Like **`execlp()`** but command-line arguments are in an array
`fexecve(int fd, char *argv[], char *envp[])`	Executes the program stored in the file descriptor **`fd`**

Example with exec ()

- The following programs runs **ls**, listing the contents of the current directory:

```
pid_t child_pid = fork ();
if (child_pid < 0)
   exit (1); // exit if fork() failed

if (child_pid == 0) // child process
   {
      int rc = execlp ("ls", "ls", "-l", NULL);
      exit (1); // only reached if exec() failed
   }
```

What if you don't want to `fork()` and `exec()`?

- Forking the current process and then executing a new process is the traditional approach
- But there are annoyances:
  - It's two different calls
  - The original process memory is copied over, even though you're just going to throw it out immediately with an **`exec()`**
  - It's not always clear what happens to the threads associated with the original process
- To simplify matters, there are **`posix_spawn()`** and **`posix_spawnp()`** (the same, but with filename lookup in PATH) functions that execute a new process all in one go
  - Of course, it has more complicated arguments to make up for being simpler

```
pid_t child = -1;
char *path = "ls";
char *argv[] = { "ls", "-l", NULL };
posix_spawnp (&child, path, NULL, NULL, argv, NULL);
```

Waiting for a child to finish

- Once you've forked or spawned a process, it will be scheduled to run
- There are no guarantees about when a parent or a child will be scheduled relative to each other
- It can be useful for a parent process to wait until its child processes have terminated
- There are two functions for this:
  - **`wait(int *stat_loc)`**
    - Waits for all children
  - **`waitpid(pid_t pid, int *stat_loc, int options)`**
    - Waits only on child process with PID

Example with `wait()`

- Here's the **`ls`** example from earlier, except that the parent process waits for **`ls`** to finish
- More code isn't shown, but the parent could continue doing other things

```
pid_t child_pid = fork ();
if (child_pid < 0)
  exit (1); // exit if fork() failed

if (child_pid == 0) // child process
  {
    int rc = execlp ("ls", "ls", "-l", NULL);
    exit (1); // only reached if exec() failed
  }

wait (NULL); // waits for ls to finish
```

Write a shell

- Read in tokens
  - The first token is the program you want to run
  - Each token after a space is an argument
  - When you reach a newline, that's the whole command
- To keep it simple, we'll support only commands with up to 10 tokens, each of which is 10 or fewer characters long
- After reading in the tokens, we wait for the child process to finish
- Repeat until the user types in **`exit`**

Files

Sharing resources

- Although physical memory is shared between processes, the virtual memory system means that processes don't share memory directly
- Other things must be shared by processes:
  - Network cards
  - Hard drives and SSDs
  - User input and output devices
- A uniform way to work with most shared resources is to *treat them all like files*
- This file abstraction makes many libraries similar and simpler

UNIX file abstraction

- The UNIX file abstraction uses two key ideas:
  - A file is a sequence of bytes
  - Everything is a file
- This abstraction is different from the traditional idea of files in a few ways:
  - Moving backwards and forwards within a file isn't always possible
  - Files don't always have names or live in a particular place
  - Files don't always have a set structure
- Even so, creating, deleting, opening, closing, reading, and writing can be treated the same

Upcoming

Next time…

- Finish files
- Events and signals

Reminders

- Work on Assignment 2
  - Due Friday by midnight
- Start working on Project 1
- Read sections 2.6 and 2.7