System Level Operations with C

When a program executes, it is known as a "process". Each process is assigned a process id (PID) when its first created, and that PID is then used to manage the process throughout its execution. Processes can be in *user mode* and in *kernel mode*. *User mode* for a process is any user written code that is executing, like an application running logic, whereas *kernel mode* is when a process is running low level operations that interact with the kernel of the operating system.

We've already done a ton of system level programming, however, what we've done was behind the scenes! A great example of how these calls work is through system input/output! When we use commands such as `putc` and `fopen`, we're using commands provided by the language libraries, and these library functions then make system level calls! When a process is running in *user mode* and makes a call with an I/O function, the process is then interrupted and switched from *user mode* to *kernel mode*. From there, once the required data have been input or output, the process is then switched back from *kernel mode* to *user mode* to then make use of the data retrieved from (or the data sent to) I/O.

System programming isn't just using calls to the *kernel mode* for I/O processes! It also lets us execute shell commands from within a c program, access the process control table, create brand new processes and let them run off and do their own work, and so much more!

Executing Shell Commands

In a c program, you can execute shell commands if necessary with the `system` command:

`call.c`

```c
#include <stdlib.h>

int main(int argc, char** argv){
  system("touch file_{1,2,3}");
}
```

By making the call to the system command, we're reaching out and running a shell command from within the same working directory as our executing program!

```
gcc call.c
./a.out
ls
```

> a.out call.c file_1 file_2 file_3

Curiously, by making these calls, we can also use the power of the shell by using any expansions that your shell can! A downside, though, is that your shell's execution speed will also slow down your program:

`sleep.c`

```c
#include <stdlib.h>

int main(int argc, char** argv){
  system("sleep 20");
}
```

The `system` command does not spin off an entirely new process to run in the background. The `system` command instead waits to return until the command is finished. There are ways to run processes in the background, which well get to later! Additionally, you cannot retrieve data from what the shell command returned with the command.

Process Control

When processes are created, they are given a virtual address space. The *user space* contains the regions:

- **Stack** - the stack, where our functions and the references to the variables/parameters/return values/etc.
- **Data** - the actual region in which the variables are all stored (i.e. where the addresses actually are). This region contains fixed memory locations for things that are allocated specifically at run time, as well as dynamically allocated memory.
- **Text** - Machine instructions for the process's functions.
- **Shared** - Shared code with other libraries.

Processes are additionally granted additional space for the kernel to interact with the process.

Life Cycle

Because of the interactive (with things like I/O) and the multiprogramming nature of linux, each process is either in a state of **Running**, **Blocked**, **Ready** or it is a **Zombie**.

Running

When a process is "running", that means that it is in the process of being executed.

Blocked

Also known as *waiting*, this is when the processes is waiting for something, such as a response from user input, a call from another program, a system call, etc. Once a program is unblocked, it can then return to the *ready* state.

Ready

A process is ready when it is not blocked, it can be scheduled to return to *running* immediately.

Zombie

When a process finishes, it becomes a zombie. The process itself dies, but it leaves behind data concerning its exit status and other statistics.

Process Table

All processes are managed by the *kernel* through the **process table**. Each process is has its own entry, distinguished by the process's id (PID), its status, and other data. To see data concerning a process, use the `ps` command in your command line:

```
ps
```

> PID TTY TIME CMD
>
> 2837 pts/0 00:00:00 bash
>
> 6071 pts/0 00:00:00 ps

Curiously, we see `bash` and `ps`. This is because we're currently interacting with the computer through the `bash` command line, and we just happened to execute the `ps` command through bash, so we not only see the program we're interacting with (i.e. bash), but also the command that lets us view the processes to begin with (i.e. ps).

Let's flesh this out a little more:

```
sleep 10 &
ps
```

> [1] 6658
>
> PID TTY TIME CMD
>
> 2837 pts/0 00:00:00 bash
>
> 6658 pts/0 00:00:00 sleep
>
> 6686 pts/0 00:00:00 ps

So not only do we see our original `bash` (note, with the same `PID` as above, since the shell never stopped), we also now see `sleep` with the `PID` that was output when we placed it in the background and `ps` which we called to see the processes in the first place!

The `ps` comand can also show us a full format listing:

```
sleep 10 &
ps -f
```

> UID PID PPID C STIME TTY TIME CMD
>
> mjlny2 2837 2836 0 21:07 pts/0 00:00:00 -bash
>
> mjlny2 7212 2837 0 21:47 pts/0 00:00:00 sleep 10
>
> mjlny2 7236 2837 0 21:47 pts/0 00:00:00 ps -f

Above, we see UID, PID, PPID, C, STIME, TTY, TIME, and CMD. What are these?

- **UID** - The id of the user who started the process
- **PID** - The process ID
- **PPID** - The process ID of the process's parent. Notice that the PPID of both `sleep` and `ps -f` is 2837, the same PID as `bash`?
- **C** - Cpu usage and scheduling information
- **STIME** - The start time of the process.
- **TTY** - The terminal associated with the process.
- **CMD** - The name of the process's command.

All processes are started from other processes, all the way back up to the root process. How do we do with with C programs?