Lecture 10: Processes

Processes vs Programs

The word Process and Program can sometimes be used interchangeably in common language. Phrases like ‘close the program’, ‘is your program running?’ and so forth hint at this blurriness. However, in computer-science land, the two have very specific definitions which make them related but clearly not the same. So, what is the distinction?

Process

A process is a thread of execution currently running on an operating system. Each clock cycle of the CPU, the operating system will have a think about all the processes running on it.

Program

A program is a series of instructions which (hopefully) do something useful when followed one after the other. A computer can read a program line by line following each of its instructions in turn. The reason I underline ‘can’ is that a program does not need a computer to be a program. A program can be written on a piece of paper many old programs were written on ‘punch-cards’. When you are writing pseudocode in a notebook, you are writing a program.

How do the two relate?

Once a computer has decided to begin execution of a program (i.e. to read and follow its instructions) it becomes a process. The process is the act of reading the program. Now, it is possible to pause a process. Is it now a process if it is not running? Sort of. The operating system will still have a process ID associated with the process and will be capable of restarting it.

Two processes can be running the same program. In fact, there is not inherent limit to this relationship. If you have ever accidentally opened the same program twice you had two processes but only one program. What about the other way around? Can you have two programs in one process? Not really. You could have one program which calls a second program, but it is hard for them to be simultaneously active within the same process. Even if they were, the definition would tend to imply that the two were actually the same program… some kind of meta program.

A process can be halfway through its program. Can you be halfway through a process? Not really. You can be halfway through one step of a process, but not halfway through a process.

A final useful analogy is that a program is like a recipe in a kitchen. A head chef who has access to many kitchen hands could set each to make the same dish. If they all follow the same recipe, the recipe is the program, and the kitchen hands are the processes being managed by the operating system (head chef). The process is the ‘making of the recipe’.

Processes in Unix

The first step to engaging with processes in Unix is to be able to view them. The command is pretty simple:

```
>> ps
```

This is not the most useful command as it only shows processes currently running in the open terminal. This is thus usually a very short list. The more useful command is:

```
>> ps -a
```

This shows all the processes running on the machine. When you call this command, you will see the name of the process and its corresponding ID. To manage processes, the operating system gives each process a unique ID which can be used to explicitly interact with the process.

Another more detailed command is:

```
>> top
```

This is more akin to task manager in Windows. (Hint 'q' is quit).

In addition to the PID (process ID), each process also has a PPID (parent process ID). The idea here is that some processes are actually parts of other processes. In task manager and other similar tool, some programs like Chrome have many processes associated with them. The idea behind the PPID is that all of these 'little processes' will link back to a parent process. Thus, if the operating system can establish rules about what happens when a parent dies (for example children could also die).

At the top of this hierarchy is the **init** process which has a PID of 1. This process is what happens when the operating system starts. It is also responsible for collecting **orphan processes** (those children who have lost their parents).

Executing Programs in Unix

The normal way to execute a program is to simply run it like a Bash command.

```
./my_program
```

OR

```
ls, cd, grep
```

Another alternative is **system**.

System Commands

The following is an example of a system command. Basically, it is a way of calling 'ls -l' from C.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main () {
   char command[50];
   strcpy( command, "ls -l" );
   system(command);
   return(0);
}
```

The **system** command effectively just calls the Unix command in sequence. The problem with this is that firstly, the C program is just pausing waiting for the **system** command to finish. This means that firstly the C program must **wait** (i.e. do nothing useful) but also must **wait** (i.e. not free up resources).

Exec Commands

What if you wanted to run a program from within a C program but not wait? C has a series of specialised functions which are designed specifically for this purpose. There are a few but we will start with **execvp**.

```
int execvp(const char * path,  char * const argv[]);
```

Unlike **system**, **execvp** starts a new process, but simultaneously ends the current process. It is a bit like jumping from one program to another whereas **system** is more like calling a program like a function.

execvp is a bit of a pain to use. The **path** argument is fairly straightforward (it is a path), but **argv** must contain the name of the executable as the first argument. Furthermore **argv** requires two arguments at a minimum, even when calling a program that takes no arguments. In this case, the second argument must be **NULL**.

Forking

There is a clear limitation to both **system** and **execvp** in that both basically hijack the current process to do something else. In the **system** case the current execution just pauses, where as in **execvp** the program simply dies. To make full use of this functionality, we need to understand something about forks.

Forking, is the process of turning one process into two. Interestingly, the second process created is of exactly the same program as the first program. Now, why would you ever want to have two instances of the same program running? It seems pretty limited.

Well, the new process is not exactly the same as the old process. The key difference is that the **fork** function (which is called to do the fork) returns different values to the parent and child. This allows each instance of the program to distinguish which kind of program it is. Is it a child? If so, **fork()** will return zero. Is it a parent? If so, **fork()** will return a non-zero number. Think of it like branding a clone with a big fat zero on its head. That way when you are trying to work out which is the original clone, you can just look for one without a zero.

This gives rise to the following program pattern:

- Execute
- Fork
- Replace Child with another Program (via **execvp**)
- Use parent to wait for Child to finish

```
while (TRUE)
{
      read_command(command, params);
      pid= fork();
      // Imagine two different programs executing from here
      if (pid < 0) // fork failed
      {
            handle_error();
      }
      else if (pid> 0) // parent process, waits for child to finish
      {
            wait(NULL);
      }
      else // child process, executes command and finishes
      {
            execvp(command, params);
      }
}
```

The **wait(NULL)** line commands the parent to wait for the child. This is important as under normal circumstances the parent's death will cause the child's (a bit like Game of Thrones zombies).

Fork Bombs

One nefarious use of **fork** is to create a so-called 'fork bomb'. If a process spawns another process using a while loop, it can continue to make more and more processes. These processes can create more processes again thus creating a kind of process zombie apocalypse. The worst part about fork bombs is that they work by infinitely consuming the computer's resources and thus are often difficult to get rid of as a user is incapable of killing the processes faster than they appear. The reboot button is a good option at this point.

While the default is for child processes to die with their parents, this relationship can be decoupled. As mentioned above the **init** process then has the job of collecting the **orphan child processes**.

Fork bombs are not always the result of malicious code. Sometimes a poorly placed while loop and an inbuilt function (such as play a sound) can be enough to crash a computer.