Scribe Notes for Lecture 14 - 2/15/16

Prepared by Mike Rabbitz, Parth Patel, Alexander Patin, and Shawn Philip

Overview

This lecture is a continuation of lecture 13, where we were talking about the Process Environment and the difference between a program and a process. In lecture 14, we covered topics relating to Process Management. During this lecture, we discussed how processes work and how the OS distinguishes the difference between each of these processes.

The following topics were covered in class:

- Process Identity
- Processes in Unix
- Process Trees in Unix
- Process Lifecycle
- Fork()

I. Process Identity

Every process has its own unique ID, also known as the **pid**. This way the OS can keep track of each process. The number of processes assigned is finite. If we want to find the ID of the current process, we need to use:

```
pid_t getpid(); //Returns the process ID for the current process
```

The pid_t data type is a signed integer type which is capable of representing a process ID. The **getpid()** function returns the process ID of the current process.

The functions that are covered in this lecture can be found in these header files' libraries:

```
#include <sys/types.h>
#include <unistd.h>
```

Another way of identifying a process is through the Task Manager. The applications you may see in the Task Manager are actually made up of multiple processes. These processes operate by their communication with each other. On Windows, you can view processes by running "Task Manager". In Linux/Unix, the **$top** command displays and updates information about the top cpu processes. It provides a dynamic real-time view of a running system. It can display system summary information, as well as a list of processes or threads currently being managed by the kernel. The types of system summary information shown, and the types, order, and size of information displayed for tasks, are all user-configurable. Raw cpu percentage is used to rank the processes. If *number* is given, then the top *number* processes will be displayed instead of the default. The first process to run is called "**init**", which has a pid value of 1.

It is important to note that there are a **finite** number of processes that can be concurring. It is a very large number, but poor handling of processes can cause buildup on the CPU. However, when processes are terminated, their process id can be recycled.

Here's an example of using the *top* utility:

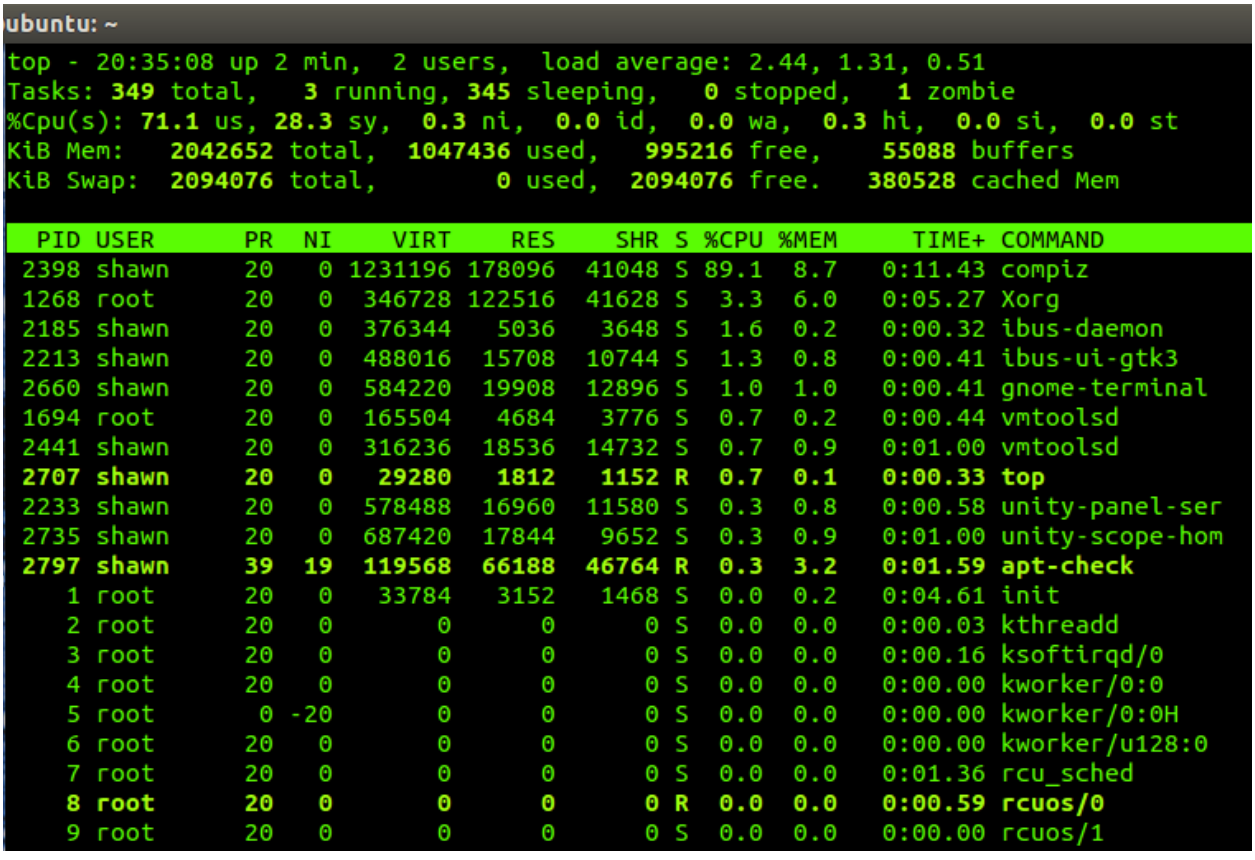

Figure 1. The top utility helps identify running processes.

II. Processes in Unix

All of the processes in Unix are related with the concept of a parent-child relationship. By associativity, each process is like a grandfather to all of the other ones The first process started during booting of the computer system is **init** (short for initialization.) It is a special daemon process that is unique and is also known as the ancestor of all processes, whether direct or indirect. Even though it doesn't really do much, it initially gets everything running, and always stays running. It is also good to note down that every process has a parent. In Figure 1, you can see the list of processes and the PIDs associated with each.

III. Process Trees in Unix

Figure 2 is an example of parent-child processes. Not all processes have to be the child of the previous process made. This is because processes are given identity files based on the order in which they are created; hence why in Figure 2, Parent 1 has Children labeled 2, 5, and 6, and not simply 2, 3, and 4. Child 2 went onto be the Parent of Child 3 and Child 4 before the Child 5 of Parent 1 was created. This concludes the fact that other processes can continue to be created from an ancestor process.

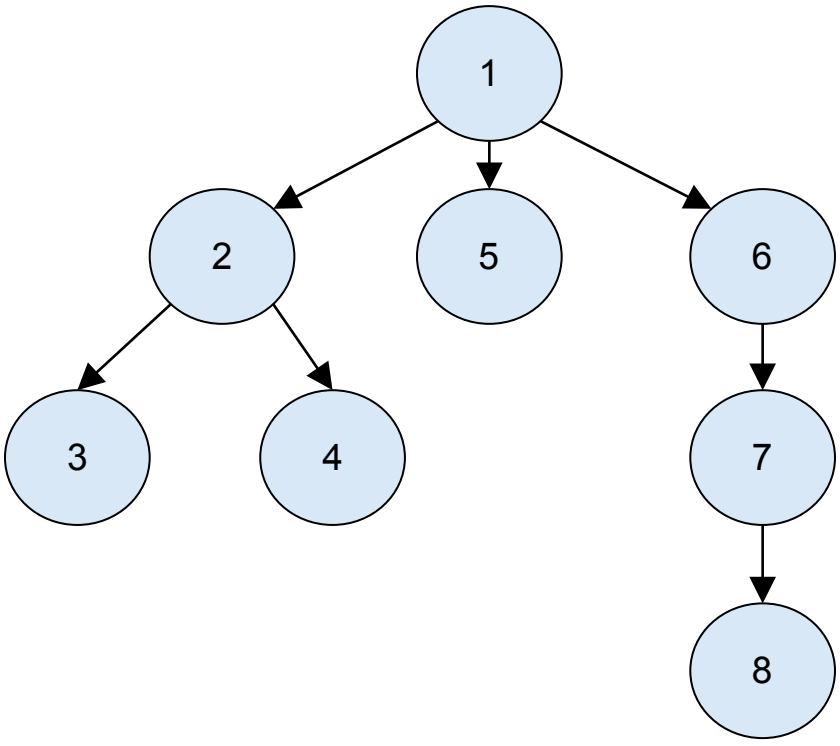

Figure 2. Parent-child relationship.

IV. Process Lifecycle

1. First, a process is created.
    a. The parent process "clones" itself when a new process is created.
2. Next, the process executes a program.
    a. It loads the program from a file.
    b. Executes the code afterwards
3. The program exits.
    a. A parent is occasionally dependant on the child finishing first. In this case, the parent will pause execution until the child has finished.

Example: Suppose a shell is opened, which in itself is a process. In the occurrence of command **top** is the creation of a new **top** process, which is the parent of the shell. The parent then waits for the child to finish, and there is nothing you can do in the shell until the **top** process exits. This is a good example of why a parent would want to wait for the child to finish first.

V. Fork()

Process Creation: Fork

1. The system call for fork is **pid_t fork();** and it is used to create processes. It takes no arguments and the significance of the return is the ID of the **new process**. The purpose of **fork()** is to create a **new** process, which becomes the **child** process of the caller. After a new child process is created, the result will be **2 processes** running (possibly,) each with its own code. **Both processes** will then execute the next instruction following the **fork()** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the value returned by **fork()**, which is **different for each process**.
   Note: fork() returns twice, once in the parent and once in the child.

There are three cases for the returned value by **fork() :**

- If **fork()** returns a negative value to the parent, the creation of a child process was unsuccessful (**an error)**
- **fork()** returns a zero to the newly created child process
- **fork()** returns a positive value, the **process ID** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an **integer**. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

*The child's program counter is initialized to that of the parent. However, because each has its own unique pid, a system call to **fork()** can tell which process is the child. After this, both processes resume from the **same point after fork()**, but with different return values

At the end of the lecture, we were going to learn about **Process Execution**. The next lecture should introduce more of this topic in Process Management.