Unit-II
Lecture: 8
(Process Management)
(Part-V)

Process Hierarchy:

In the operating system, a process can create a child process. Hence, all processes come from a single root in which a creating process is the parent process and the created process is the child of that process. The child process can itself create more processes, thereby forming a process hierarchy.

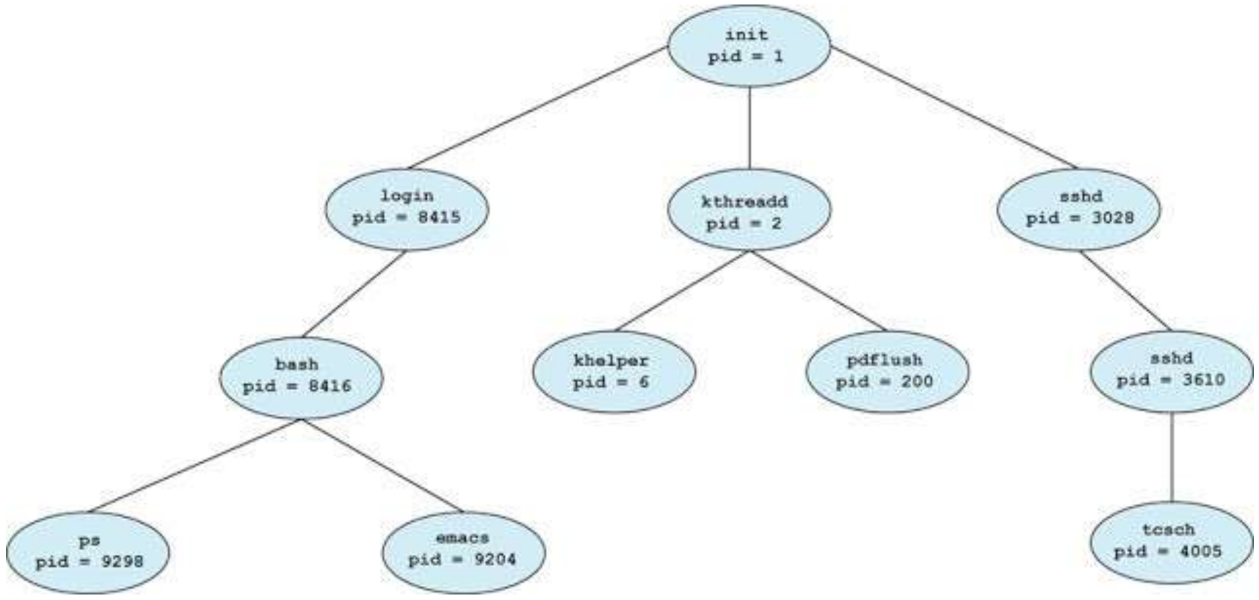

Fig: Process Creation Tree

For example, in UNIX:

Once the initial boot process is done and the operating system is loaded, a special process called **init**, present in the boot image starts running. It reads a file telling how many terminals are there. Then it forks off a new process per terminal. These processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes and so forth. Thus, all the processes in the whole system belong to a single tree, with init at the root.

In UNIX-like environment, we can view the current process tree by:

$ps –e

Then, you can find that we have many processes and the root for all of them is the process with pid=1 and name init (for UNIX-based systems).

To find the process related only to the current terminal use:

$ps –a

Creating new processes:

Use of fork and exec provide a very simple way to create new processes. Both the fork and exec are system calls of the operating system.

- **fork:** will create a copy of the current process as its child process. The values from the PCB of the current process will be copied. At that point, both the parent and the child will be continuing immediately after the fork because both of them have almost the same values in their process control block.
- **exec:** will take the created PCB by fork and rewrite the value of it with some other programs. Thus a new program will be loaded and the child PCB will now be different from the parent PCB. The child process now contains the values from the new programs.

Let's open two terminals and execute the command ps in one of these 2 terminals,

```
$ ps
```

The output should be as follows (note that the PID, TTY, and TIME can be different),

```
PID   TTY          TIME CMD
25867 ttys000    0:00.38 -bash
32580 ttys001    0:00.24 -bash
```

Now, let's run the following command in another terminal (for me, pid = 32580),

```
$ cat
```

And from the first terminal, we now run,

```
$ ps
```

Then we can find a new process of cat ,

```
  PID TTY          TIME CMD
25867 ttys000    0:00.38 -bash
32580 ttys001    0:00.24 -bash
33681 ttys001    0:00.00 cat
```

This process is actually the child process (for me, pid = 33681) of the second process (for me, pid = 32580). Then we can use ^c to kill the process of 33681 . After killing this process, we can find out that there is no cat process.

```
$ ps
```

Thus, what really happens is that:

when we run the command cat , the terminal actually uses both the fork system call and the exec system call. The fork is called (shown in 1.) in the first place to create a copy of the process 32580 . Then the exec is called to replace the PCB of the process 33681 with the context of the command cat . After we killed the process 33681 by ^c, the CPU will restore the state of its parent process 32580. This is shown by the following diagram.

```
+------------+
| pid=32580  |
| bash       |
|            |
+------------+
   |
   | 1. calls fork
   V
+------------+              +-----------+
| pid=32580  |    forks     | pid=33681 |
| bash       | -----------> | bash      |
|            |              |           |
+------------+              +-----------+
   |                           |
   | waits for pid 33681       | 2. calls exec to run cat
   |                           V
   |                        +-----------+
   |                        | pid=33681 |
   |                        | cat       |
   |                        |           |
   V                        +-----------+
+------------+                 |
| pid=32580  |                 | exits
| bash       | <---------------+
|            |
+------------+
   |
   | continues
   V
```

Reference: https://medium.com/adamedelwiess/operating-system-3-process-and-process-management-process-control-block-process-lifecycle-66bbf73ee3f6

So, the mechanism of creating a new program is like calling the fork which creates a child process with exact same PCB as that of the parent and then calling exec which replaces the child's image with the new program's image.

Threads:

Modern operating systems allow a process to be divided into multiple threads of execution. The threads within a process share all process management information except for information directly related to execution.

Threads in a process can execute different parts of the program code at the same time. They can also execute the same parts of the code at the same time, but with different execution state.

- A thread is the unit of execution within a process.
- It is also known as lightweight process.
- Thread uses parallelism which provides a way to improve application performance.
- A thread is a basic unit of CPU utilization, consisting of a program counter, a stack and a set of registers (and a thread ID).
- A thread is an entity within a process that can be scheduled for execution.

Each thread belongs to exactly one process and no thread exists outside a process. Each thread represents a separate flow of control.

Advantages of Thread:

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.