2.3 OPERATIONS ON PROCESSES

The operating system must provide a mechanism for process creation and termination. The process can be created and deleted dynamically by the operating system. The Operations on the process includes

- Process creation
- Process Termination

2.3.1 Process Creation

During Execution a process may create several new processes.

- The creating process is called as the **parent process** and the newly created process is called as the **child process**.
  - Processes may create other processes through appropriate system calls, such as **fork** or **spawn**.
  - The operating systems identify the processes according to their unique process identifier.

:

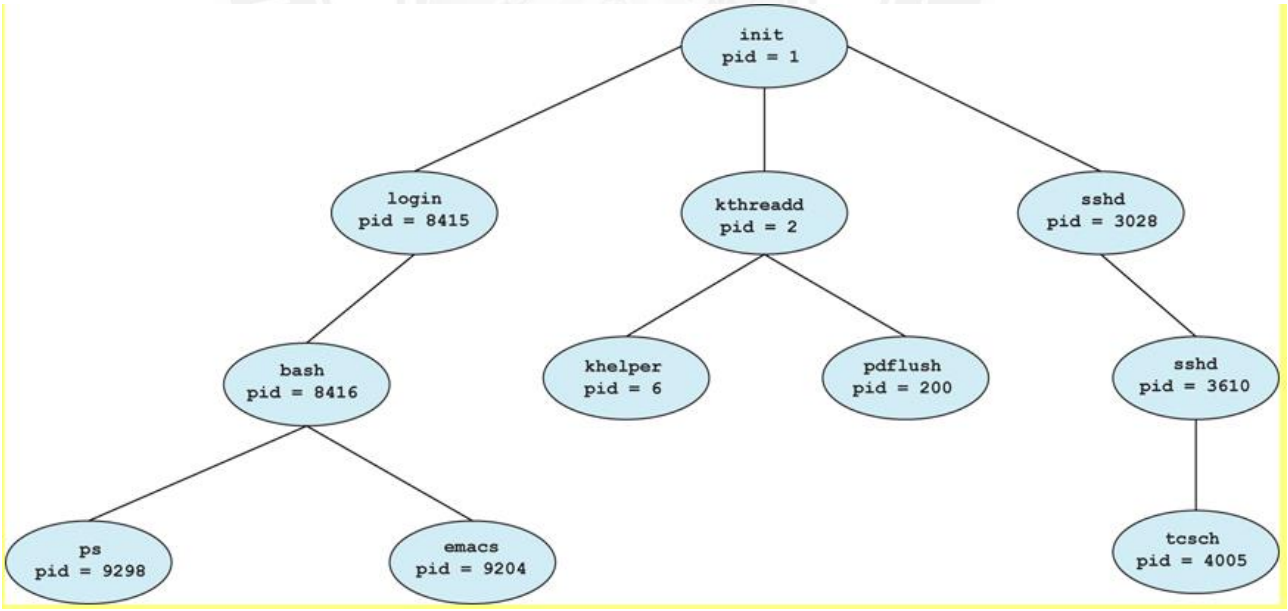


Fig: A tree of processes on a typical Linux system

- The init process serves as the root parent process for all the user process.
- Once the system has booted, the init process can also create various user processes, such as a web or printserver, an ssh server.
- The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel

- The sshd process is responsible for managing clients that connect to the system by using ssh(Secure shell)
- The login process is responsible for managing clients that directly log onto the system
- The command ps –el will list complete information for all processes currently active in the system.

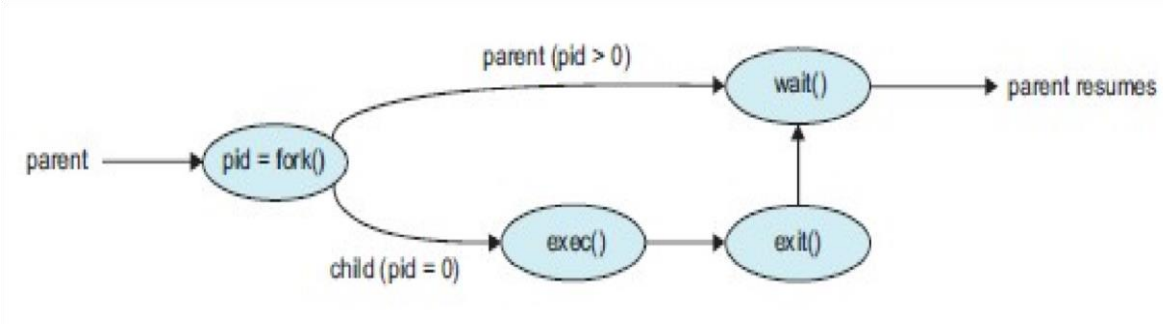
When a process creates a new process, two possibilities for execution exist:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated

There are also two address-space possibilities for the new process:

- The child process is a duplicate of the parent process (it has the same program as the parent).
- The child process has a new program loaded into it.
- The return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
    - After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program.
    - A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

Depending on system implementation, a child process may receive some amount of shared resources with its parent. Child processes may or may not be limited to a subset of the resources originally allocated to the parent, preventing runaway children from consuming all of a certain system resource.




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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Creating a separate process using the UNIX fork( )system call.

Process creation using the fork( ) system call

2.3.2 Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- At that point, the process may return a status value (typically an integer) to its parent process.
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system

- **A parent may terminate the execution of one of its children for a variety of reasons, such as**
- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.
- Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon is referred to as **cascading termination.**
- A parent process may wait for the termination of a child process by using the wait() system call
- This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid t pid;
int status;
pid = wait(&status);
```

  - A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process.