Operations on Processes -

The Processes in most s/y's can execute concurrently, they may be Created & Deleted Dynamically.

Process Creation - Fork()

During Execution, a Process may create several new processes.

Creating process → Parent Process.
New process → child process.
child process may create sub process, forming a tree of process.

OS identify process using unique Process identifier (pid)

2 Execution Possibilities -
1) The parent continues to execute concurrently with its children
2) parent waits until some or all its child get terminated.

2 Address space possibilities
1) child is duplicate of parent process (ie) same program & Data)
2) child has a new program loaded to it.

Process Termination - Exit()

* A process terminates when it finishes Executing its final statement & inform OS to delete it using Exit() system call. The process return status to its parent process. All the resources including memory, files & I/O buffers are deallocated by the OS.

* A process can also be terminated by parent process using terminateprocess() system call.

A parent may terminate its child due to following reasons.

* The child exceeded usage of allocated resources.
* The task assigned to child is no longer required.
* The Parent is exiting & the OS does not allow child to continue if its Parent terminates.

Some s/y do not allow child to exist if its parent has terminated. In such s/y's, if a process terminates (normally or abnormally), then all its children must also be terminated. this phenomenon, referred to as Cascading termination, is normally initiated by the OS.

A Parent process may wait for termination of child process by using the wait() system call. wait() is passed as parameter that allows the parent to obtain the exit status of the child. this also returns the Pid of terminated child, so parent can tell which child has terminated.

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

When a process terminates, its resources are deallocated by OS. But its entry in process table which contains process's exit status, must remain until parent calls wait(). A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process. Once parent calls wait(), the pid & entry in process table of zombie process are released.

If Parent didn't invoke wait() & terminated leaving its child processes as orphans. Linux & Unix address this by assigning init process as new parent to orphan processes. this init periodically invokes wait() allowing collecting exit status of orphan process & release orphan's process identifier & process-table entry.

Process Creation

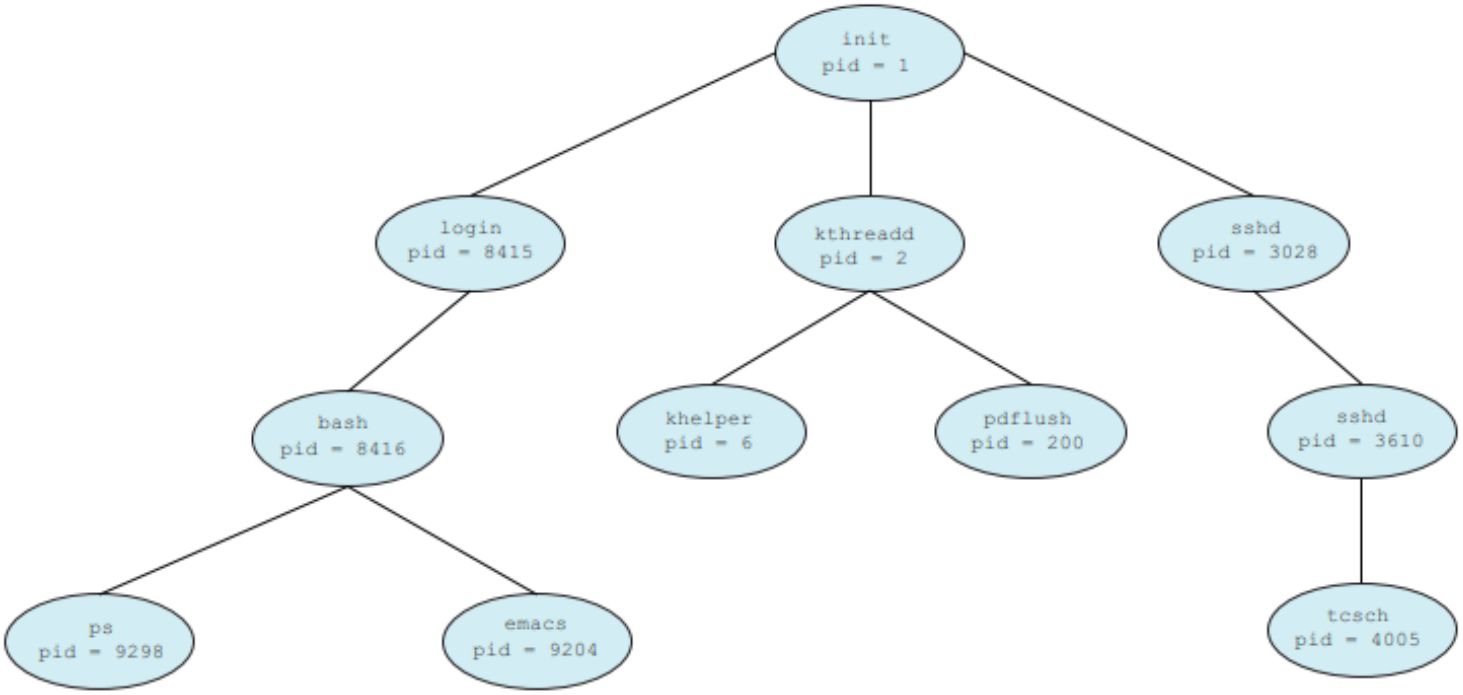

Figure 3.8 A tree of processes on a typical Linux system.

```
#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;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

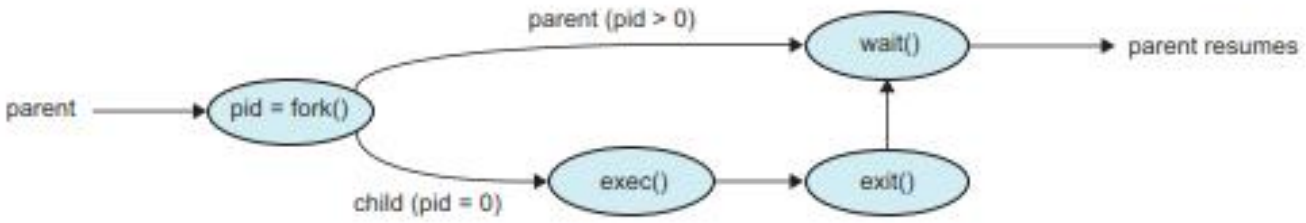

Figure 3.10 Process creation using the `fork()` system call.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figure 3.11 Creating a separate process using the Windows API.