**OPERATIONS ON PROCESSES**

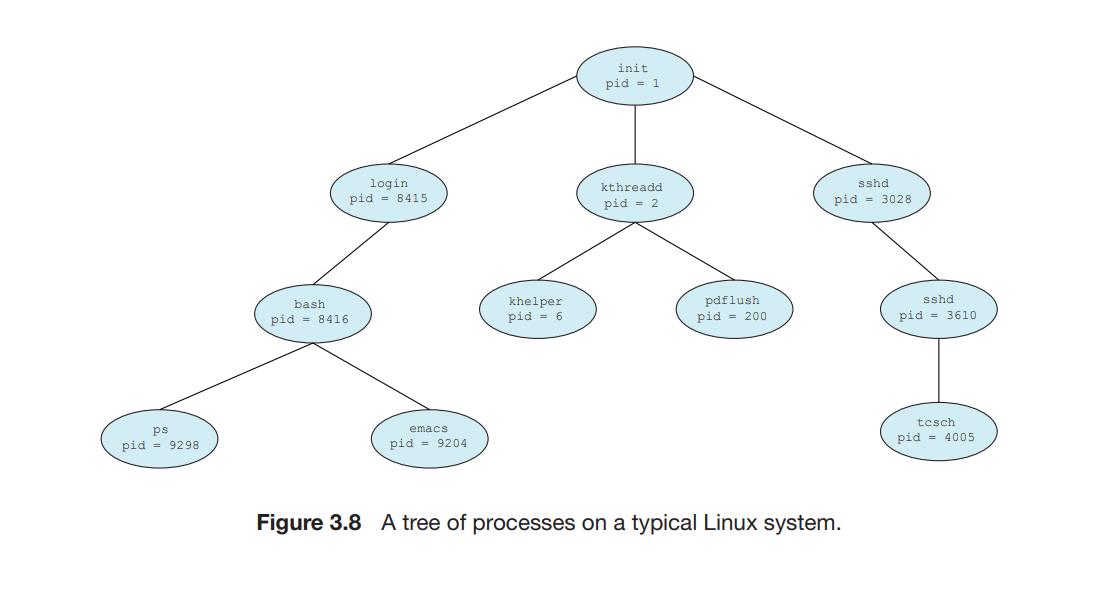
The operations of process carried out by an operating system are primarily of two types:

1. Process creation
2. Process termination

**1)Process creation:**

During the course of execution, a process may create several new processes. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique process identifier (or pid), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.



Above Figure illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. The init process (which always has a pid of 1) serves as the root parent process for all user processes. Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like. In above Figure, we see two children of init—kthreadd and sshd. The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush). The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for secure shell). The login process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416. Using the bash command-line interface, this user has created the process ps as well as the emacs editor.

On UNIX and Linux systems, we can obtain a listing of processes by using the ps command.

**Ex: ps -el** will list complete information for all processes currently active in the system.

When a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.

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

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

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

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).

2. The child process has a new program loaded into it.

Ex: In UNIX, C program for process creation

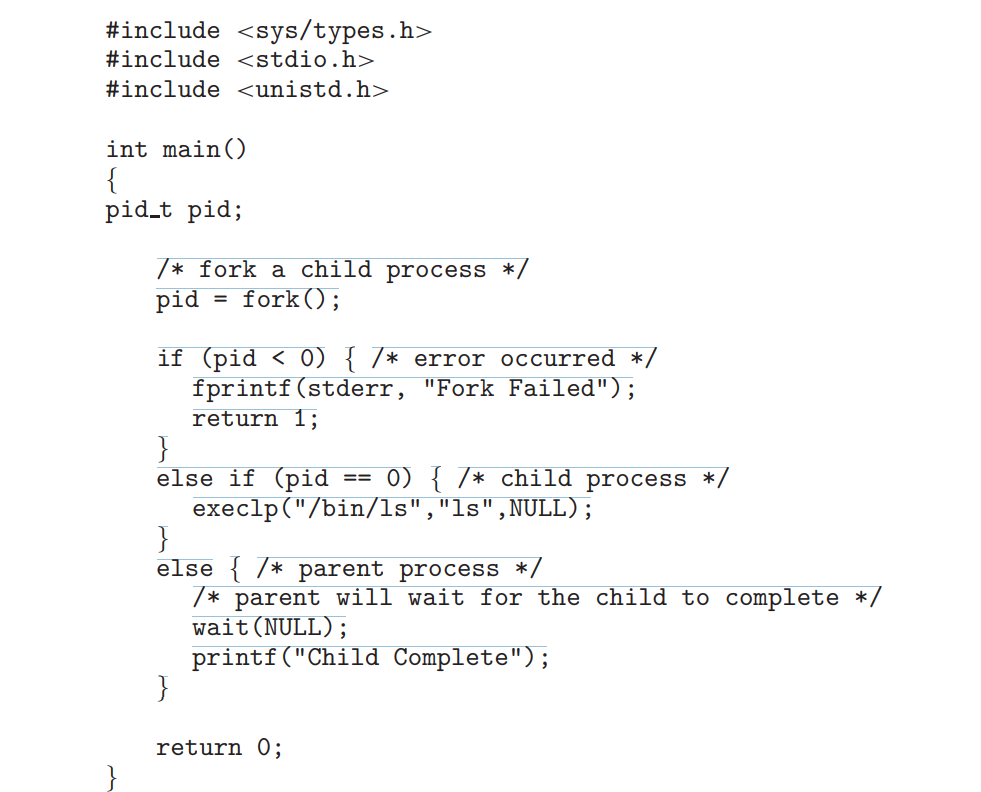
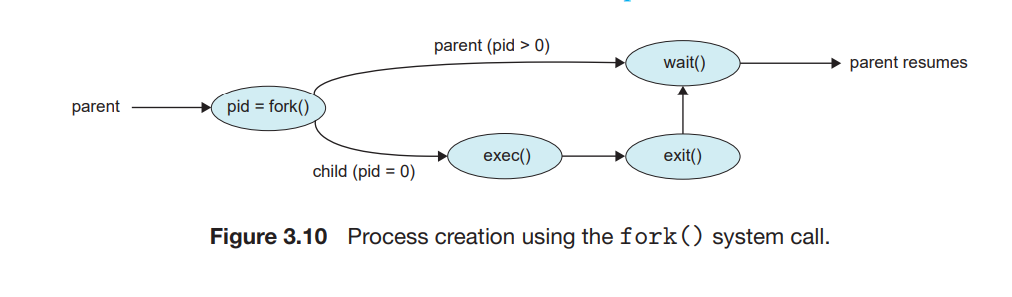


Figure 3.9 Creating a separate process using the UNIX fork() system call.

A diagram that demonstrates process creation using fork() is as follows –

****

**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 (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess() in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other’s jobs.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

• The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)

• 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, referred to as cascading termination, is normally initiated by the operating system.

In UNIX, we can terminate a process by using the exit() system call, providing an exit status as a parameter:

/\* exit with status 1 \*/

exit(1);

A parent process may wait for the termination of a child process by using the wait() system call. The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child. 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. if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans.