Process Management

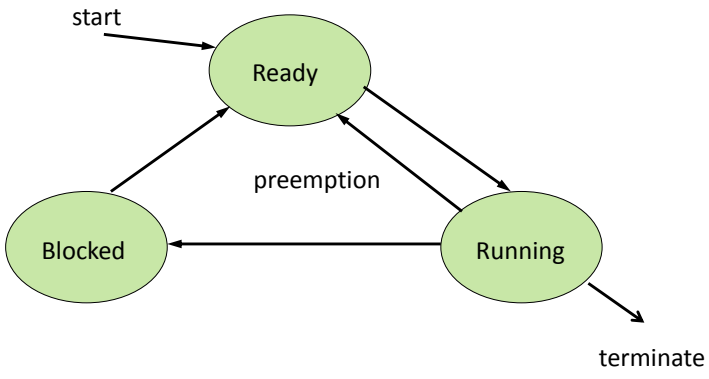
The Process Concept

A process is usually defined as a “program in execution”.

A *program* is a static sequence of instructions stored in memory. The *process* is the dynamic operation of executing a program

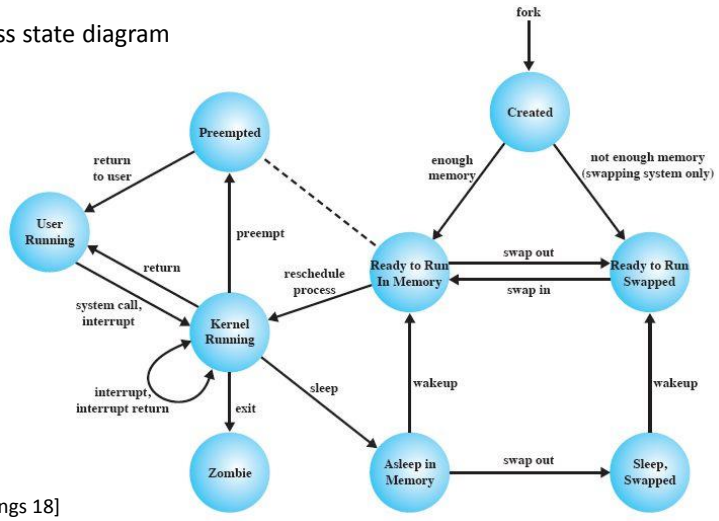
The process is the fundamental unit of computation that the system must manage. It is the unit to which resources are assigned.

Process State Diagram



Process State Diagram

UNIX process state diagram



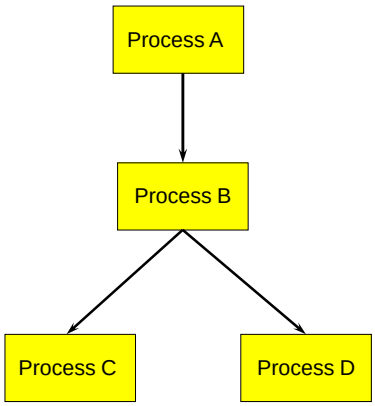
Source:[Stallings 18]

Process Tree

A new process is generated using a system call, e.g. fork in UNIX or CreateProcess in Windows.

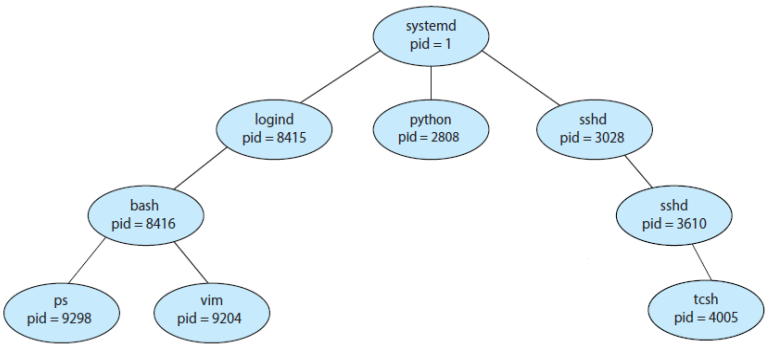
Parent process can spawn one or more child processes.

Parent either wait for the child or run concurrently with it.



Process Tree

Example: Typical LINUX process tree



Source: [Silberschatz 18]

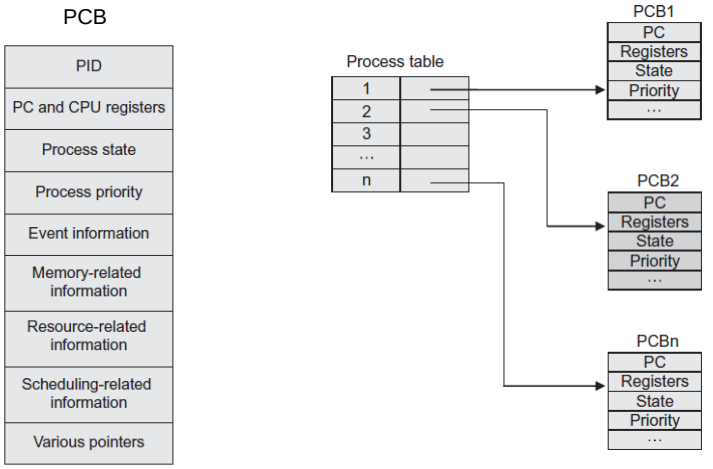
Process Control Block

Information needed by the system to control a process is stored in an appropriate data structure.

Stored information include:

- Process id, parent id, owner id.
- Process context.
- Process state.
- Memory and I/O resources assigned to the process.

Process Control Block



Source: [Chauhan 14]

Threads

So far, we assumed that a process has a single *thread* of execution: i.e. a single sequence of executed instructions.

In many applications, however, several parts of the same process can run in parallel. These parts share memory address space and other resources.

In a *multithreading* system, a process may be composed of several threads that run in parallel.

Memory space, I/O devices, ..etc. are assigned to a process. However, CPU time is assigned to a single thread. Each thread has independent state and context.

Threads

Advantages of Multithreading

- Process can remain responsive if a part of it is blocked or performs a long operation.
- Creating a new thread of an existing process requires less time and resources than spawning a new process. Also context switching between threads of the same process is faster than switching to another process.
- In a mutliprocessor system, threads can be assigned to different processors (or cores), thus speeding up the execution of a given process.

CPU Scheduling

The problem of CPU scheduling is to decide which ready process (thread) to run next on the CPU and for how long.

In multitasking (non real-time) systems, a scheduling algorithm is selected trying to:

- increase the processor throughput.
- decrease the process waiting time.
- preserve fairness among users or tasks.

No single algorithm can achieve all these requirements under all conditions.

First-Come First-Served (FCFS) Scheduling

Whenever a running process is terminated or blocked, select for running the oldest process in the ready queue, and run it until it is terminated or blocked.

This algorithm is a non-preemptive scheduling algorithm: once a process runs, it will not be interrupted by the OS until it is terminated or blocked.

This algorithm is simple to implement, however:

- It can be unfair for short processes.
- It results in poor system reliability in case of errors.

CPU Scheduling

Let a_i = the arrival time of process i

t_i = its termination time

p_i = its process time (execution time)

Then for each process $(t_i - a_i) \geq p_i$

The process *waiting time* is given by:

$$w_i = (t_i - a_i) - p_i$$

We are interested in the average waiting time w_{av} or the average value of w_i / p_i.

CPU Scheduling

Example 1:

Process	a_i	p_i
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

Example 2:

Process	a_i	p_i
A	0	100
B	1	5
C	2	5

Find the average waiting time in the above two examples using different scheduling algorithms.

Example 2 illustrates that FCFS may be unfair to short processes.