Lecture 3: Processes & Threads (Part 1 of 3)

Defintion of Process

We're starting with CPU as a resource, so we need an abstraction of CPU ues. We define a *process* as the OS's representation of a program in execution so that we can allocate CPU time to it. (Other defs: "the thing pointed to by a PCB" to "the animated spirit of a procedure"). Note the difference between a program and a process. The `ls` program on disk is a program; the `ls` instance running on a computer is a process. Multiple processes can be running the same program.

The process abstraction turns out to be convenient to hang other resource allocations on. File system usage (disk/tape), memory usage, network usage, and number of subprocesses (i.e. further CPU utilization) are all assigned on a per-process basis.

Processes are the unit of isolation. Two processes cannot directly affect each other's behavior without making explicit arrangements to. This isolation extends itself to more general security concerns. Processes are tied to users, and the credentials from running on a user's behalf determine what resources the process can use.

Process Operatons and Attributes

Processes can be created and destroyed. Becuase the're created or destroyed by the CPU, they are created or destroyed by other processes. Because of this OS designers speak of *parent* and *child* processes (the parent being the creator and the child being the creation).[1]

Process lineage is often more than an accounting convention. Some attributes of processes are inherited (user, open files, current directory), and parents and children often have a communication channel.

> UNIX®processes inherit open file descriptors, current and root directory, user credentials and other attributes. A child can return an integer to its parent on completion and all children must have their completion checked by their parent. Children whose parents exit before them become children of the system `init` process.

Processes can form other associations for the purpose of communication. These associations can be formed after the processes have been created and can change dynamically. In UNIX the ordering of processes imposed by their creation order is called the *process heirarchy* and the dynamic process associations are called *process groups*. Some systems have both or neither.

Signals

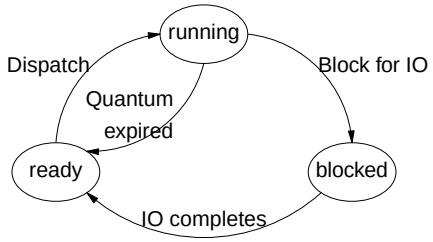
Signals are OS abstractions of interrupts. They are indications to a process that some asynchronous event has occurred. They may be generated by external events (some other process writes to a file or socket) or internal events (some period of time has passed). Processes can generally define handler procedures that are invoked when a signalis delivered. These *signal handlers* are analogous to interrupt service routines (ISRs) in the OS.

Processes in Action

At any given moment a process is in one of several states:

---

[1] These gender-neutral terms are something of an innovation. In the early days of computer science, talk of father and son processes was more common. This tradition worked in reverse at IBM, where processes were female. Because male mammals don't bear young, this is one of the few times where IBM nomenclature is more sensible.



The functions of the states are:

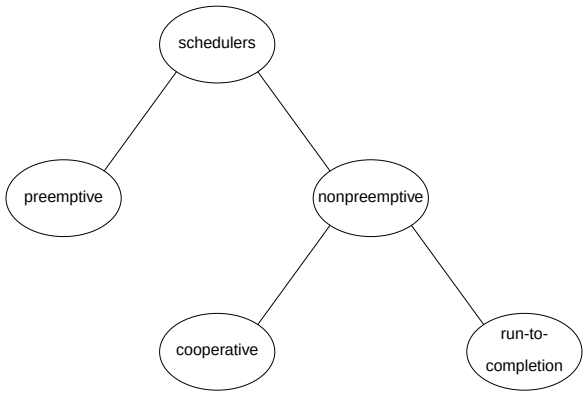
Running	The process is executing on the processor. Only one process is in this state on a given processor.
Blocked	The process is waiting for some external event, for example disk I/O
Ready	The process is ready to run

These states may be defined implicitly. A process is in the ready state if its on the ready queue, or blocked if it's on the blocked queue. Frequently there is more than one ready or blocked queue. There may be multiple ready queues to reflect jobs priorities and multiple blocked queues to represent the events for which the process is waiting.

The act of removing one process from the running state and putting another there is called a *context switch* because the context (that is the running environment: all the user credentials, open files, etc.) of one process is changed for another. We'll talk more about the details of this next lecture, but you should think about what constitutes a process context.

Good questions to ask are "why does a process leave the running state?" and "how does the OS pick the process to run?" The answers to those questions make up the subtopic of process scheduling.

There are several kinds of schedulers:



Run-to-completion schedlers are the easiest to understand. The process leaves the running state exactly once, when it exits. A process never enters the blocked state. Examples are batch systems. Some web servers are conceptually run to completion, but because they are usually implemented on systems with a more complex scheduler, their behavior is more complex.

Processes in a coopertive multitasking environment tell the OS when to switch them. They explicitly block for I/O or they specifically give up the CPU to other processes. An example is Apple's original multitasking system and some Java systems.

A preemptive multiprocessing system interrupts (preempts) a running process if it has had the CPU too long and forces a context switch. UNIX is a preemptive multitasking system.

The time a process can keep the CPU is called the system's *time quantum.* The choice of time quantum can have a profound effect on system performance. Small time quanta give good interactive performance to short interactive jobs (which are likely to block for I/O). Larger quanta are better for long-running CPU bound jobs because they do not make as many context switches (which don't move their computations forward). If the time quantum is so small that the system spends more time switching processes than doing useful work, the system is said to be *thrashing*. Thrashing is a condition we shall see in other subsystems as well. The general definition is when a system is spending more time on overhead than on useful work.