Processes

• A process is a program in execution
• Synonyms include job, task, and unit of work
• Not surprisingly, then, the parts of a process are precisely the parts of a running program:
  ◦ Program code, sometimes called the text section
  ◦ Program counter (where we are in the code) and other registers (data that CPU instructions can touch directly)
  ◦ Stack — for subroutines and their accompanying data
  ◦ Data section — for statically-allocated entities
  ◦ Heap — for dynamically-allocated entities

Process States

• Five states in general, with specific operating systems applying their own terminology and some using a finer level of granularity:
  ◦ New — process is being created
  ◦ Running — CPU is executing the process's instructions
  ◦ Waiting — process is, well, waiting for an event, typically I/O or signal
  ◦ Ready — process is waiting for a processor
  ◦ Terminated — process is done running

• See the text for a general state diagram of how a process moves from one state to another
The Process Control Block (PCB)

- Central data structure for representing a process, a.k.a. task control block
- Consists of any information that varies from process to process: process state, program counter, registers, scheduling information, memory management information, accounting information, I/O status
- The operating system maintains collections of PCBs to track current processes (typically as linked lists)
- System state is saved/loaded to/from PCBs as the CPU goes from process to process; this is called…

The Context Switch

- Context switch is the technical term for the act of changing the currently running process — the aforementioned saving/loading of PCB data
- When a process must exit the running state (interrupt, I/O request, time slice expiration, etc.), a save state operation updates its PCB
- A state restore operation reads the PCB of the next running process into the system
- Textbook case of overhead: context switch does take time, but ultimately doesn’t do any “real” work
Scheduling Queues

• Only one running process per CPU — part of an operating system’s core tasks is to decide which process is “the one”…and the next one, and the next

• To assist in making these decisions, multiple scheduling queues exist — linked lists of PCBs — that correspond to the process state (thus, events that trigger state changes have corresponding queue changes)
  ◆ Job queue: all processes in the system
  ◆ Ready queue: processes that are waiting for a CPU
  ◆ Device queues: one per I/O device, containing processes that are waiting for that device

Types of Schedulers

• Batch systems are unable to immediately run every single process submitted to it; these are spooled to secondary storage to await execution — deciding the next job to run from this pool is long-term scheduling

• Deciding among jobs already in memory for processing by the CPU is short-term or CPU scheduling

• Most systems today have a very high degree of multiprogramming, and so have no long-term scheduling at all; time-sharing results when the short-term scheduler enforces rapid switching among processes
Process Creation and Termination

- All processes have a unique identifier — the process identifier or pid for short
- The boot sequence typically leads to process 0, whose name varies according to the operating system; all other processes are created by this one
- Thus, all processes (except process 0 of course) also have a parent process ID (ppid)
- Parents may terminate their children, or processes may end/terminate on their own

APIs for Process Creation and Termination

Programming specifics for process creation and termination vary per OS, but they generally consist of:

- Function to create a new (child) process — this returns information about the child to the parent
- Function to wait for a child to finish or to continue execution concurrently
- Function to load a program (executable) for execution
- Function to end execution (willingly — we will discuss external termination later)
Interprocess Communication

- Processes aren’t isolated from each other — if desired, they can communicate, and facilitating interprocess communication (IPC) is another fundamental operating system service.

- Two overall models:
  - *Shared memory* — processes are allowed to read/write a section of memory.
  - *Message passing* — processes send information blocks (messages) to each other.

IPC Issues

Things to consider when designing or implementing an IPC scheme:

- Buffer sizes (shared memory blocks, message passing queues) — *unbounded* or *bounded*.
- Naming of message passing sources/destinations — *direct* (PID) or *indirect* (intervening abstractions, such as mailboxes or ports).
- The big one: *synchronization* — how to coordinate reads/writes to shared memory; should message passing be *blocking* or *nonblocking*.
IPC Across Machines

Modern operating systems allow IPC across different hosts; because we cross machine boundaries, these methods follow the message passing model

- **Sockets**: communicate via machine address and port numbers; as the Internet evolved, *well-known ports* have been reserved for certain protocols
- **Remote procedure call (RPC)**: instead of raw bytes, communication resembles (duh) a procedure call
- **Remote method invocation (RMI)**: object-oriented RPC — objects are accessible over the network

RPC/RMI Mechanics

- Because we cross machine boundaries, RPC is semantically a *pass-by-value* call — data is necessarily copied over the network
- The translation of RPC arguments into a network message then back into arguments on the remote host has a specific term — *marshalling*
- RMI adds the notion of a *remote object* — the ability to hold a reference to an object on another machine; with remote objects, we are able to do a limited form of *pass-by-reference*, but on other remote objects only
Managing Processes from the Unix Command Line

• *bash* (along with many other shells) has a number of built-in process-related features:

  ◦ Appending commands with an ampersand (\&) runs that command concurrently with the shell (otherwise, the shell waits for the command to finish)

  ◦ Multiple commands can be run in sequence by separating them with semicolons (;)

  ◦ When multiple programs are running within a shell, *fg, bg, and jobs* help to examine and manage them

• Not surprisingly, *man -k process* yields a lot of “hits” on typical Unix distributions; here are some essentials:

  ◦ *ps, or process status,* is the one-stop shot for listing currently-running processes

  ◦ *kill* sends signals to or forcibly terminates processes

  ◦ *top* provides a sorted/updated display of processes

  ◦ *time* runs another program and displays timing statistics for that program

• Piping *ps* through *grep* filters *ps*’s output by process attributes such as user, program name, status, etc. — this combination is virtually all you need for querying or focusing on specific process information