Processes:

- The definition of a process, informally, is quite simple: it is a running program [V+65,BH70].
- The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action.
    - It is the operating system that takes these bytes and gets them running, transforming the program into something useful
- The OS creates an illusion by virtualizing the CPU. By running one process, then stopping it and running another, and so on, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few).
    - This basic technique, known as **time sharing** of the CPU, allows users to run as many concurrent processes as they would like; the potential cost is performance, as each will run more slowly if the CPU(s) must be shared.
    - Time sharing is a basic technique used by an OS to share a resource.
        - By allowing the resource to be used for a little while by one entity, and then a little while by another, and so forth, the resource in question (e.g., the CPU, or a network link) can be shared by many.
        - The counterpart of time sharing is space sharing, where a resource is divided (in space) among those who wish to use it. For example, disk space is naturally a space shared resource; once a block is assigned to a file, it is normally not assigned to another file until the user deletes the original file.
- Policies are algorithms for making some kind of decision within the OS. For example, given a number of possible programs to run on a CPU, which program should the OS run?
    - A scheduling policy in the OS will make this decision, using historical information (e.g., which program has run more over the last minute?), workload knowledge (e.g., what types of programs are run), and performance metrics (e.g., is the system optimizing for interactive performance, or throughput?) to make its decision.
- To understand what constitutes a process, we thus have to understand its machine state: what a program can read or update when it is running.
    - One component of the machine state that comprises a process is its memory. Instructions lie in memory; the data that the running program reads and writes sits in memory as well. Thus the memory that the process can address (called its address space) is part of the process
    - Also part of the process's machine state are registers; many instructions explicitly read or update registers and thus clearly they are important to the execution of the process.

Process API:

- The process API consists of calls programs can make related to processes. Typically, this includes creation, destruction, and other useful calls.

Process Creation

- The first thing that the OS must do to run a program is to load its code and any static data (e.g., initialized variables) into memory, into the address space of the process.
    - Programs initially reside on disk (or, in some modern systems, flash-based SSDs) in some kind of executable format; thus, the process of loading a program and static data into memory requires the OS to read those bytes from disk and place them in memory somewhere
    - modern OSes perform the process lazily, i.e., by loading pieces of code or data only as they are needed during program execution
    - Once the code and static data are loaded into memory, some memory must be allocated for the program's run-time stack (or just stack).
        - C programs use the stack for local variables, function parameters, and return addresses;
        - the OS allocates this memory and gives it to the process. The OS will also likely initialize the stack with arguments; specifically, it will fill in the parameters to the main() function, i.e., argc and the argv array
    - The OS may also allocate some memory for the program's heap. In C programs, the heap is used for explicitly requested dynamically-allocated data; programs request such space by calling malloc() and free it explicitly by calling free().
        - The heap is needed for data structures such as linked lists, hash tables, trees, and other interesting data structures.
        - The heap will be small at first; as the program runs, and requests more memory via the malloc() library API, the OS may get involved and allocate more memory to the process to help satisfy such calls.
    - The OS will also do some other initialization tasks, particularly as related to input/output (I/O). For example, in UNIX systems, each process by default has three open file descriptors, for standard input, output, and error; these descriptors let programs easily read input from the terminal and print output to the screen.
    - After all this, the OS has one last task: to start the program running at the entry point, namely main(). By jumping to the main() routine the OS transfers control of the CPU to the newly-created process, and thus the program begins its execution.

States:

- Running: In the running state, a process is running on a processor. This means it is executing instructions.
- Ready: In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- Blocked: In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.
- Being moved from ready to running means the process has been scheduled; being moved from running to ready means the process has been descheduled.

Data Structures:

- To track the state of each process, for example, the OS likely will keep some kind of process list for all processes that are ready and some additional information to track which process is currently running. A process list contains information about all processes in the system. Each entry is found in what is sometimes called a process control block (PCB), which is really just a structure that contains information about a specific process.
- Must also track blocked processes
- Sometimes a system will have an initial state that the process is in when it is being created. Also, a process could be placed in a final state where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called the zombie state).
    - This final state can be useful as it allows other processes (usually the parent that created the process) to examine the return code of the process and see if the just-finished process executed successfully
    - When finished, the parent will make one final call (e.g., wait()) to wait for the completion of the child, and to also indicate to the OS that it can clean up any relevant data structures that refer to the now-extinct process.