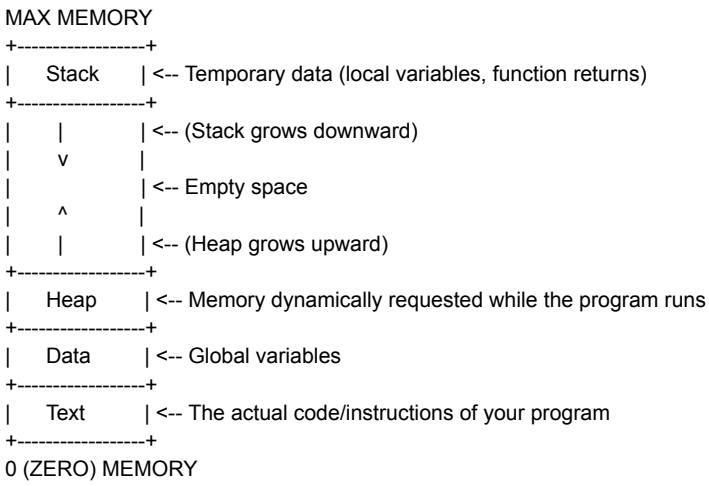
A **program** is just a passive file stored on your disk (like an app you haven't opened yet). A **process** is an active program that is currently running and executing instructions. You can have multiple processes running the exact same program at the same time. When a program becomes a process, the computer loads it into memory. Here is a visualization of how a process is organized in memory:



1. Process States: As a process executes, it changes state:

- **New:** The process is being created.
- **Ready:** The process is waiting to be assigned to a processor.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur (like I/O completion).
- **Terminated:** The process has finished execution.

To keep track of all this, the operating system uses a **Process Control Block (PCB)**. Think of the PCB like a "saved game" file for your process. If the CPU needs to pause one process to work on another, it saves the current state, memory info, and exact location of the code in the PCB. When it's time to run that process again, it simply reloads the PCB and picks up exactly where it left off.

2. Process Control Block (PCB): Also called a task control block, this represents the "context" of a process. It contains:

- **Process state:** running, waiting, etc.
- **Program counter:** The location of the *next* instruction to execute.
- **CPU registers:** The contents of all process-centric registers.
- **CPU scheduling information:** Priorities and scheduling queue pointers.

- **Memory-management information:** The memory allocated to the process.
- **Accounting information:** CPU used, clock time elapsed since start, and time limits.
- **I/O status information:** I/O devices allocated to the process and a list of open files.

3. Threads: Normally, a process only does one thing at a time, using a single "program counter" or a single main thread to know which instruction to run next. A **thread** is simply a way for a process to have *multiple* program counters, allowing it to execute different parts of its code at the exact same time. Because of this, the PCB must have enough space to store the details and program counters for all of those individual threads.

The main goal of process scheduling is to maximize CPU use by keeping it busy at all times and quickly switching between processes.

1. Scheduling Queues (Where processes wait)

- **Job queue:** A set containing all the processes currently in the system.
- **Ready queue:** A set of processes residing in main memory that are ready and waiting to execute on the CPU.
- **Device queues:** A set of processes waiting to use a specific I/O device. Processes constantly migrate between these various queues as they run.

2. The Schedulers (The decision makers)

- **Short-term scheduler (CPU scheduler):** Selects which process from the ready queue gets to use the CPU next. Because it runs very frequently (every few milliseconds), it must be very fast.
- **Long-term scheduler (Job scheduler):** Selects which processes should be brought into the ready queue. It runs infrequently (seconds or minutes) and controls the "degree of multiprogramming," which is how many programs are competing for memory and the CPU.
- **Medium-term scheduler:** If memory gets too crowded, this scheduler can remove a process from memory, store it on the disk, and bring it back later to continue executing. This is called "swapping".

3. Process Types Schedulers look at how a process behaves to make decisions:

- **I/O-bound process:** Spends more time doing I/O operations (like waiting for user input or loading files) and only needs short bursts of the CPU.
- **CPU-bound process:** Spends more time doing heavy computations and uses very long bursts of the CPU.

The operating system uses the **long-term scheduler** (also called the job scheduler) to strike this balance. To maximize system performance, the scheduler aims for an optimal mix of the two process types.

If the scheduler only loaded I/O-bound processes, the CPU would often sit idle because all the processes would be stuck in the waiting queue for their I/O tasks to finish. If it only loaded CPU-bound processes, the system's I/O devices would sit unused while the CPU gets entirely bottlenecked.

4. Context Switch When the CPU switches to another process, the operating system must save the state of the old process into its PCB, and then load the saved state from the new process's PCB. The time it takes to do this is considered "overhead," meaning the system does no useful work while the switch is happening.

1. Process Creation

- A "parent" process creates "child" processes, forming a tree structure.
- Every process is identified and managed using a unique Process Identifier (PID).
- **Resource Sharing:** The parent and child can share all resources, a subset of the parent's resources, or no resources at all.
- **Execution:** The parent and child can run at the same time (concurrently), or the parent can pause and wait for the child to finish.
- **UNIX Commands:** The `fork()` system call is used to create a new child process. The `exec()` system call is used right after `fork()` to replace the child's memory space with a brand new program.

2. Process Termination

- A process normally ends by executing its last statement and using the `exit()` command to ask the operating system to delete it and free up its resources.
- A parent can forcefully end a child process using the `abort()` command. This usually happens if the child uses too many resources, the task is no longer needed, or if the parent itself is exiting (some systems do not allow children to keep running if the parent dies, which is called "cascading termination").

3. Special Process States

- **Zombie:** A process that has completely terminated, but its parent hasn't yet called the `wait()` command to acknowledge it and collect its final status.
- **Orphan:** A child process whose parent has completely terminated without ever calling `wait()`.

Part 4: Interprocess Communication

Processes in a system can either be independent or they can cooperate with each other. Cooperating processes need Interprocess Communication (IPC) to share data and affect one another. The slides list four main reasons why processes cooperate: information sharing, computation speedup, modularity, and convenience. There are two primary models for how IPC works:

1. Shared Memory A region of memory is shared between processes so they can read and write to the same space. A classic example of this is the **Producer-Consumer Problem**, where a "producer" process creates information and a "consumer" process uses it. They share a buffer to hold this data, which can be:

- **Unbounded-buffer:** No practical limit on the size of the buffer.
- **Bounded-buffer:** A fixed buffer size, meaning the producer must wait if the buffer is full. It can only use BUFFER_SIZE-1 elements. By always leaving one slot empty, the system can easily tell the difference between a completely full buffer and a completely empty buffer.

2. Message Passing Processes communicate by exchanging messages without using shared variables. This uses two basic operations: `send(message)` and `receive(message)`. Message passing can be set up in two ways:

- **Direct Communication:** Processes explicitly name each other (e.g., send to Process P), and a link is automatically created between exactly those two processes.
- **Indirect Communication:** Messages are sent to and received from shared "mailboxes" or ports that have unique IDs. For example, Windows uses this method via a Local Procedure Call (LPC) facility where clients and servers communicate through ports.
- **Mailbox Sharing (IPC):** If process A sends a message to an indirect communication mailbox shared by processes B and C, who gets the message? The operating system solves this by either limiting links to two processes, allowing only one process to execute a receive command at a time, or letting the system pick arbitrarily and notifying the sender.

Synchronization When passing messages, the operating system must synchronize the actions. This is done in two ways:

- **Blocking (Synchronous):** The sender is paused (blocked) until the message is successfully received, or the receiver is paused until a message becomes available.
- **Non-blocking (Asynchronous):** The sender sends the message and immediately continues its work, or the receiver checks for a message and grabs a valid message or simply a "null" if nothing is there.

Extra information related to companies:

- **Mobile Multitasking:** Early systems only ran one process at a time. Today, iOS limits background processes heavily (allowing only short tasks or specific things like audio playback), while Android uses "services" to run background tasks with fewer limits.
- **Chrome Browser Architecture:** Older browsers ran as a single process, so one bad website could crash the whole program. Chrome prevents this by using three separate processes: a **Browser process** (for the user interface and I/O), a **Renderer process** (which runs in a secure "sandbox" to handle HTML/Javascript for each individual website), and a **Plug-in process**.