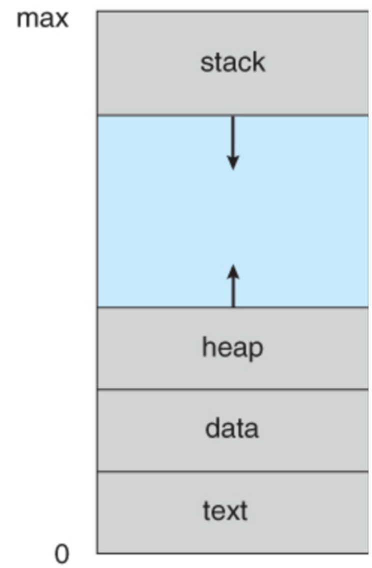
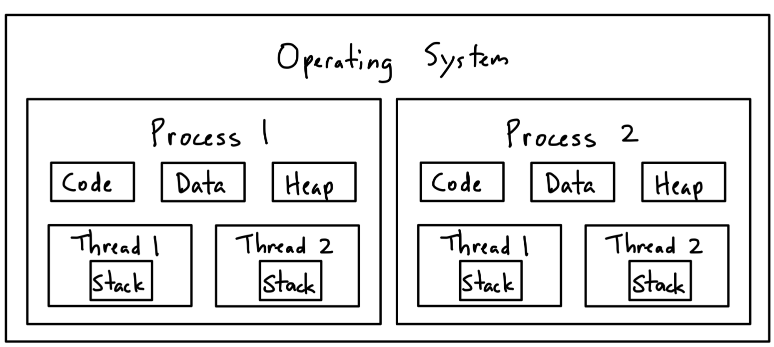
Processes



- A **process** is a program loaded into main memory, ready for execution.
- The OS assigns isolated memory to each process, ensuring no process can access another's memory. The memory is divided into four segments:
    1. **Stack**: Contains local variables, parameters, and return addresses.
    2. **Heap**: Used for dynamic memory allocation.
    3. **Data**: Stores global variables, arrays, and structures. It is further divided into:
        - **BSS (Block Started by Symbol)**: For uninitialized variables.
        - **Data**: Contains initialized variables.
    4. **Text**: The program's compiled machine code.
- **Memory Growth**:
    - The stack and heap grow toward each other.
        - **Stack Overflow**: When the stack crosses into the heap.
        - **Heap Overflow**: When the heap crosses into the stack.

Threads

- A **thread** is the smallest unit of execution within a process, sharing the process's memory space.
- Each thread has its own stack within the allocated memory of the process.
- **Thread Levels**:
    - Threads operate at both user-level and kernel-level.
    - In a multi-threaded environment, thread management can follow these models:
        - Many-to-One
        - One-to-One
        - Many-to-Many

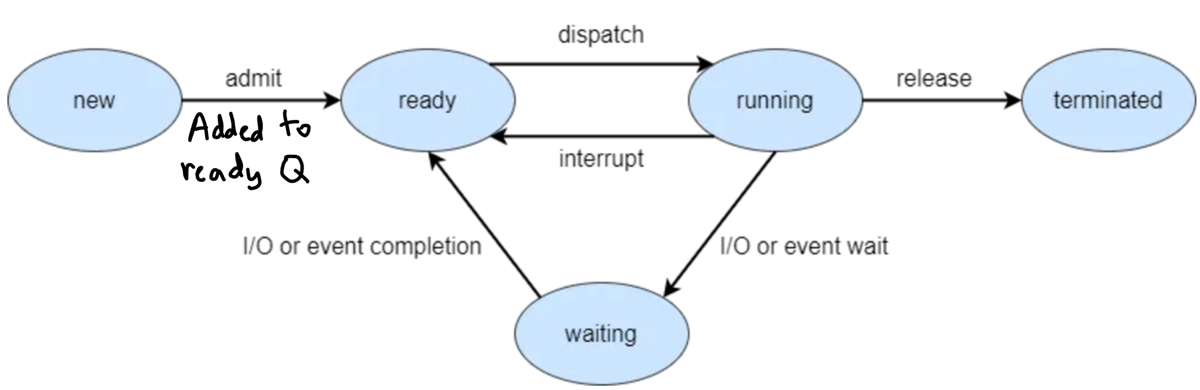


Process Control Block (PCB)

- The **Process Control Block (PCB)** is a data structure used by the OS to manage process metadata.
- Key elements of the PCB include:
  - **PID (Process ID):** A unique identifier for the process.
  - **Parent PID:** The ID of the parent process.
  - **Child PID:** The ID of child processes.
  - **Program Counter (PC):** The location of the next instruction to execute.
  - **Memory Limits:** Defines the boundaries of the process's memory.
  - **Other Metadata:** Includes process state, scheduling information, and I/O status.

Process Creation

- A new process (**child**) is created by forking (using the `fork()` system call) from an existing process (**parent**).
  - **fork() Return Values**:
    - The parent process receives the child's PID.
    - The child process receives `0`.
- Processes can exist in the following states:
  1. **New:** Just created, waiting to be admitted.
  2. **Ready:** Prepared to run when the CPU is available.
  3. **Running:** Actively being executed by the CPU.
  4. **Waiting:** Paused, waiting for an event (e.g., I/O operation).
  5. **Terminated:** Finished execution.
- **State Transitions**:
  - A process in the ready state moves to running if it has the highest priority.
  - If it times out, it returns to the ready state.



Multiprocessing and Multithreading

Concurrency vs. Parallelism

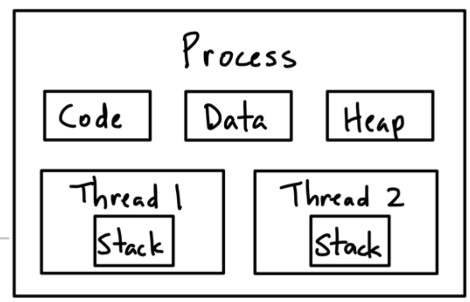
- **Concurrency**: A single core CPU switches between tasks (context switching).
- **Parallelism**: A multicore CPU executes multiple tasks simultaneously.

Multiprocessing

- Multiprocessing involves splitting a task into multiple processes.
  - **Pros**: Stable, as processes operate independently.
  - **Cons**: Overhead due to context switching, which costs time and memory.
- Processes are allocated separate memory spaces and communicate via **IPC (Inter-Process Communication)**.

Multithreading

- Multiple threads exist within a process, each performing different tasks.
- Threads share the **heap, data, and text** segments but have separate stacks.
- **Pros**:
  - Less overhead in context switching.
  - No need for IPC.
- **Cons**:
  - Shared stack requires synchronization.
  - A problem in one thread may affect others.



Context Switching

- **Interrupts**: Requests made to the CPU for attention due to events like:
  - I/O operations.
  - CPU usage time expiration.
  - Creation of child processes.
- **Mechanism**:
  - The CPU can only process one task at a time.
  - An interrupt, often caused by the CPU scheduler, switches the CPU to another process.
  - Overhead occurs during context switching because the CPU idles while loading a process's state into the registers.
- **Program Counter (PC) and Stack Pointer**:
  - **PC:** Holds the address of the next instruction to execute.
  - **Stack Pointer:** Points to the largest address in the stack.

Process Synchronization

Race Condition

- Multiple processes (or threads) accessing a shared resource at the same time, leading to unpredictable outcomes.
- Example: Too much milk problem

Critical Section

A **critical section** is a part of a program where shared resources are accessed. To prevent issues like race conditions, certain conditions must be met:

1. **Mutual Exclusion**: Ensures only one thread or process can access the critical section at a time.
    - **Semaphore**: Uses a wait-and-signal mechanism to manage access among multiple threads.
    - **Mutex**: Allows only one thread access; other threads are blocked (busy waiting).
2. **Progress**: If no process is in the critical section, the decision of which process will enter next cannot be postponed indefinitely.
3. **Bounded Waiting**: Guarantees that every process will eventually get a turn to access the critical section after a limited number of turns by others.

Critical Section Problems

1. **Deadlock**:
    - Occurs when two or more processes are waiting indefinitely for each other to release resources, creating a standstill.
2. **Starvation**:
    - Happens when a process is perpetually denied access to the critical section because other processes monopolize the resource.
3. **Preemptive vs. Non-Preemptive**:
    - **Preemptive**: A thread may be forcibly removed from the critical section before it finishes.
    - **Non-Preemptive**: A thread runs to completion before another is allowed access.

Synchronization Concepts

1. **Synchronization**: Ensures the execution order of processes or threads, maintaining a predictable sequence of operations.
2. **Asynchronization**: Does not guarantee the order of execution.
3. **Blocking**: A process or thread waits (is blocked) until a condition is met, resulting in some delay.

4. **Non-Blocking**: The process or thread continues without waiting for conditions to be satisfied.