**Process Management**

A **process** is a program in execution, requiring **CPU, memory, I/O, and files** to complete its task. A **program** is passive (just stored in memory), while a **process** is active (executing with a program counter). Program becomes process when executable file loaded into memory. One program can be several processes.

**Key Concepts**

* **Single-threaded processes** execute one instruction at a time with one program counter.
* **Multi-threaded processes** have multiple threads, each with its own program counter.
* **Concurrency** allows multiple processes to run simultaneously by time-sharing CPUs.

**Process Management Activities**

The OS is responsible for:

1. **Creating & Deleting Processes** – Manages both user and system processes.
2. **Suspending & Resuming Processes** – Supports pausing and resuming execution.
3. **Process Synchronization** – Ensures correct execution when multiple processes interact.
4. **Process Communication** – Enables processes to share data using IPC mechanisms.
5. **Deadlock Handling** – Detects and resolves process deadlocks.

A diagram of a stack

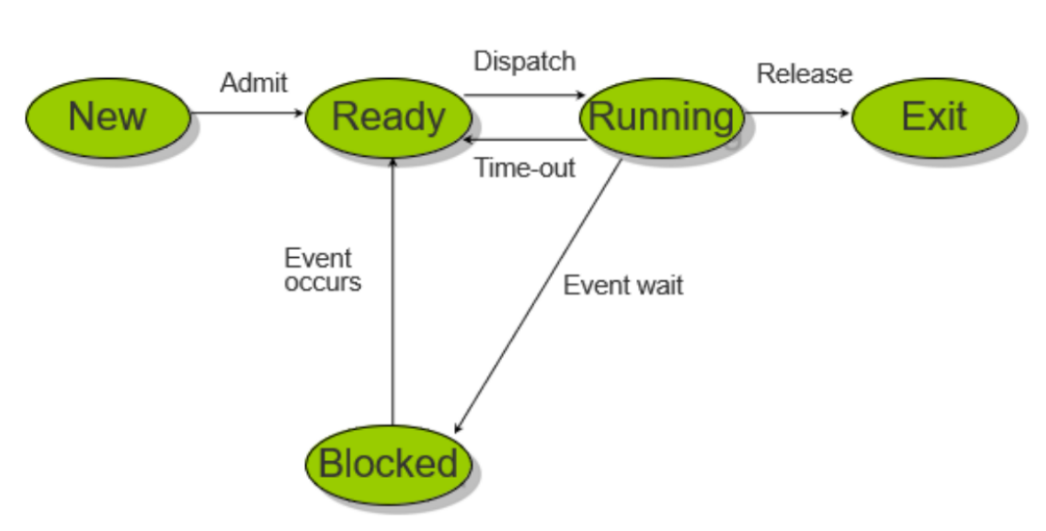
AI-generated content may be incorrect.**Process Concept**

A process is a program in execution. It is the basic unit of work in an operating system.

A process includes:

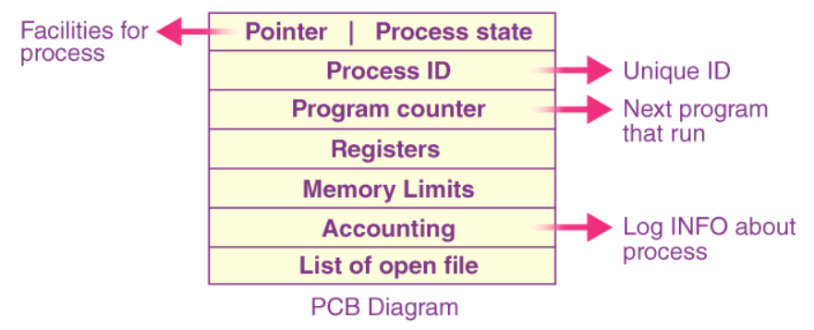
* Program code **(text section**).
* Current activity (program counter, **CPU registers**).
* **Stack** (temporary data like function parameters, return addresses, local variables).
* **Data** section (global variables).
* **Heap** (dynamically allocated memory during runtime).

**State Process Model**



* A diagram of a running process

  AI-generated content may be incorrect.**New:** A process has been created but has not yet been admitted to the pool of executable processes.
* **Ready:** Processes that are prepared to run if given an opportunity. That is, they are not waiting on anything except the CPU availability.
* **Running:** The process that is currently being executed. (Assume single processor for simplicity.)
* **Blocked:** A process that cannot execute until a specified event such as an IO completion occurs.
* **Exit:** A process that has been released by OS either after normal termination or after abnormal termination (error).

**Process Control Block (PCB)**

The Process Control Block (PCB) is a fundamental data structure in operating systems.

The PCB stores all the necessary information about a process, allowing the OS to track its state, allocate resources, and manage its execution.

**How the PCB is Used:**

**Process Creation:**

* When a new process is created, the OS allocates a PCB for it and initializes the fields.

**Context Switching:**

* When the OS needs to switch from one process to another, it saves the current state of the running process in its PCB.
* It then loads the state of the next process from its PCB, restoring the CPU registers and memory management information.

**Scheduling:**

* The OS uses the scheduling information in the PCBs to determine which process to run next.

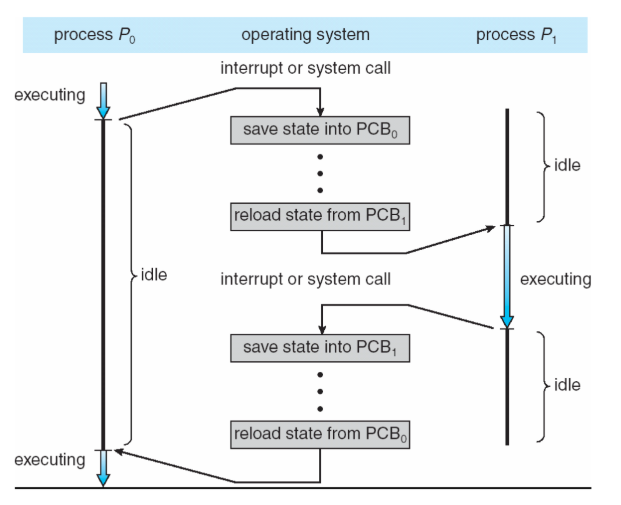
**Resource Management:**

* The OS uses the memory and I/O information to allocate and deallocate resources to processes.

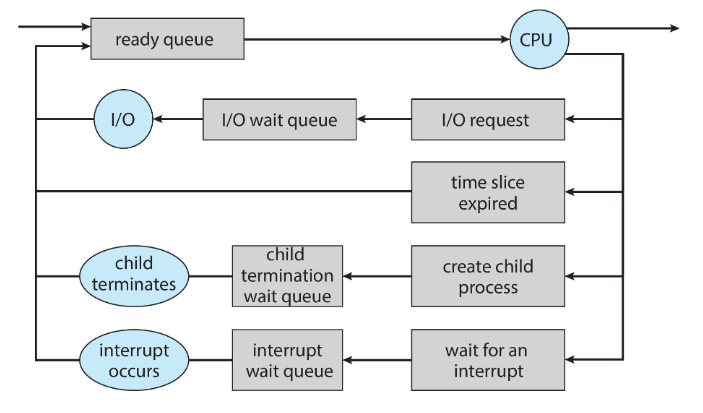
**Process Termination:**

* When a process terminates, the OS releases its PCB and any other resources allocated to it.

**Context Switching Between Process to Process**

* **Context Switching:**
  + When the CPU switches from one process to another, the state of the current process is saved in its PCB, and the state of the new process is loaded from it PCB.
  + This is called a context switch.
* **Overhead:**
  + Context switching takes time and is considered overhead since no useful work is done during the switch.

**When does context switching happen?**

* **Multitasking:** Switching between processes in a time-shared system.
* **Interrupt Handling:** Switching from user mode to kernel mode to handle system calls or interrupts.
* **Process Priority Changes:** Switching to a higher-priority process.
* **I/O Blocking:** When a process is waiting for I/O, the CPU switches to another ready process.

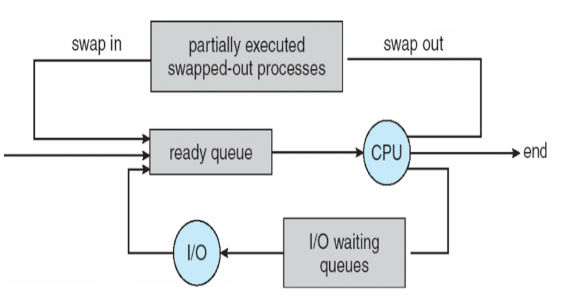
**Process Scheduling**

Image 1: **Queueing Diagram** represents queues, resources, flows. It shows how a process can leave CPU in 5 different ways.

Image 2: **Ready Queue & Device Queue** representation.

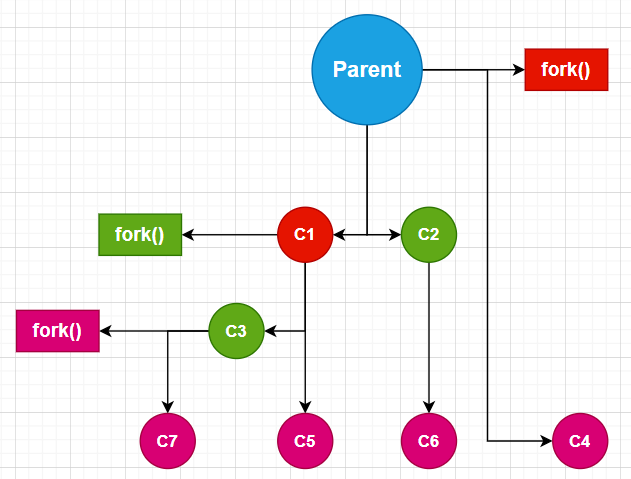
Image 3: **Medium-term scheduler** represents how a process is swapped in ready queue and swapped out of CPU.

* **Purpose**:
  + The **process scheduler** selects which process should execute next on the CPU.
* A diagram of a computer process

  AI-generated content may be incorrect.**Scheduling Queues**:
  + **Job Queue**: All processes in the system.
  + **Ready Queue**: All processes residing in main memory, ready and waiting to execute.
  + **Device/Wait Queue**: Processes waiting for I/O devices.
* **Schedulers**:
  + **Short-term scheduler (CPU scheduler)**: Selects the next process from ready queue to execute on the CPU (invoked frequently).
  + **Long-term scheduler (job scheduler)**: Selects which processes should be brought into the ready queue from job queue (invoked infrequently). It controls the degree of multiprogramming.
  + **Medium-term scheduler**: Handles **process** **swapping** (moving processes between memory and disk). Remove process from memory, store on disk, bring back in from disk to continue execution. It can be added if the degree of multiple programming needs to decrease. It oversees **multiprogramming**. This swapping occurs when a process is not using CPU, it is swapped out of RAM and moved into the disk, which frees up RAM.

**Operations on Processes**

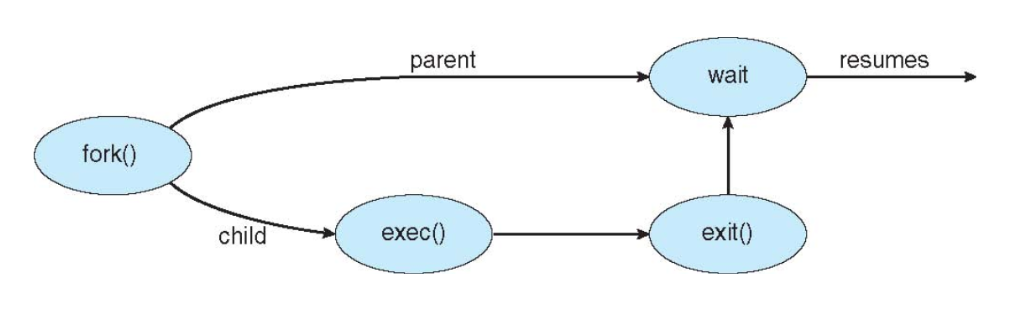
* The OS provides mechanisms for:
  + **Process creation**.
  + **Process termination**.
  + **Inter Process communication (IPC)**. (Chapter 3 Part 2)

**Process Creation**

Fork(); duplicates the code written below it and then runs it. The duplicated code is the child process.

The **total process** would be 2n and **total child** process would be 2n-1.

**Examples:** Total process are 4 out of which 3 are child processes. So, printf would run 4 times.



A screenshot of a computer

AI-generated content may be incorrect.**A screen shot of a computer code

AI-generated content may be incorrect.**

**Process Termination**

Process executes last statement and then asks the operating system to delete it using the **exit()** system call.

* Returns status data from child to parent (via **wait()**)
* Process’ resources are deallocated by operating system

Parent may terminate the execution of child processes using the **abort()** system call. Some reasons for doing so:

* Child has exceeded allocated resources
* Task assigned to child process is no longer required
* The parent is exiting and the operating system does not allow a child to continue if its parent terminates

**Cascading termination:** All children, grandchildren, etc. are terminated.

The termination is initiated by the operating system.

The parent process may wait for termination of a child process by using the **wait()** system call. The call returns status information and the **pid** of the terminated process:

**pid = wait(&status);**

* If no parent waiting (did not invoke **wait()**), the process is a **zombie**
* If a parent terminated without invoking **wait()**, the process is an **orphan**

**Zombie Process**

A **zombie process** is a process that has completed execution (terminated) but still has an entry in the process table.

Zombie processes are essentially "dead" processes that don't consume any resources except for the process table entry.

When a child process terminates, it **does not immediately disappear**. Instead, it enters the **zombie state** until the parent process collects its exit status using wait().

If the parent process does not call wait(), the child process remains in the **process table** as a zombie.

This happens because the kernel **keeps the process metadata (PID, exit status, etc.)** to allow the parent process to retrieve it.

If the parent process **itself terminates**, the zombie process is adopted by **init (PID 1)**, which automatically cleans up the zombie.

**Orphan Process:**  
An **orphan process** is a process whose **parent process has terminated** (either normally or abnormally) without waiting for the child process to complete. In most operating systems, when a parent process terminates, its child processes are either terminated as well or become orphaned. Orphan processes are typically **adopted by a special system process** (often the **init process**, with PID 1 in Unix-like systems), which becomes their new parent.

**Key Characteristics of Orphan Processes**

1. **Parent Termination**:
   * The parent process terminates before the child process completes its execution.
   * The child process is left without a parent.
2. **Adoption by Init**:
   * In Unix-like systems, orphan processes are adopted by the **init process** (or its modern equivalents like **systemd**).
   * The init process periodically calls **wait()** to collect the exit status of orphaned processes, preventing them from becoming zombies.
3. **No Direct Relationship**:
   * Once orphaned, the child process no longer has a direct relationship with its original parent.
4. **Resource Usage**:
   * Orphan processes continue to consume system resources (e.g., memory, CPU) until they terminate.



**Parent Terminates Without wait()**:

* If a parent process terminates without calling **wait()** or **waitpid()**, its child processes become orphans.

**what happens when the parent uses wait and then child uses sleep?**

**Parent Process Calls** wait():

* The parent process calls the wait() function to wait for any of its child processes to terminate.
* The wait() function makes the parent process pause its execution until one of its child processes finishes executing and terminates.
* The parent process remains in a "waiting" state during this time.

**Child Process Calls** sleep():

* The child process calls the sleep() function to pause its execution for a specified number of seconds.
* During the sleep period, the child process remains inactive but still exists in the system.
* The child process will wake up and resume execution after the sleep period ends.

**Sequence of Events**

what happens when the parent uses wait() and then child uses sleep()?

1. The parent process calls wait() and enters the waiting state.
2. The child process calls sleep() and pauses its execution for the specified number of seconds.
3. The parent process remains in the waiting state during the entire sleep period of the child process.
4. After the sleep period ends, the child process wakes up and continues its execution.
5. Once the child process completes its execution and terminates (e.g., using exit()), the wait() function in the parent process returns.
6. The parent process resumes its execution after the wait() function returns, and it can then proceed to handle the termination status of the child process.

**What if the parent uses wait and then the first child uses sleep, and there exists a second child, what happens in this scenario? what will be the execution order?**

**Parent Process Calls** wait():

* The parent process calls the wait() function to wait for any of its child processes to terminate.
* The wait() function makes the parent process pause its execution until one of its child processes finishes executing and terminates.
* The parent process remains in a "waiting" state during this time.

**First Child Process Calls** sleep():

* The child process calls the sleep() function to pause its execution for a specified number of seconds.
* During the sleep period, the child process remains inactive but still exists in the system.
* The child process will wake up and resume execution after the sleep period ends.
* Until then, the **second child** starts executing.
* Once the **second child** **process** completes its execution and terminates (e.g., using exit()), the **wait() function** in the parent process **returns**.
* If the **second child** had also called the sleep function, the execution would have switched back to the **first child**.
* After the sleep period ends, the **first child process** wakes up and continues its execution after the parent process has terminated, as there was **only one** wait(), and it collected the **second child’s exit status** and **returned**.

**Does the sleep timer only decrements when we are in the child process or it decrements independently of the current process in execution, meaning even if we are in the parent process?**

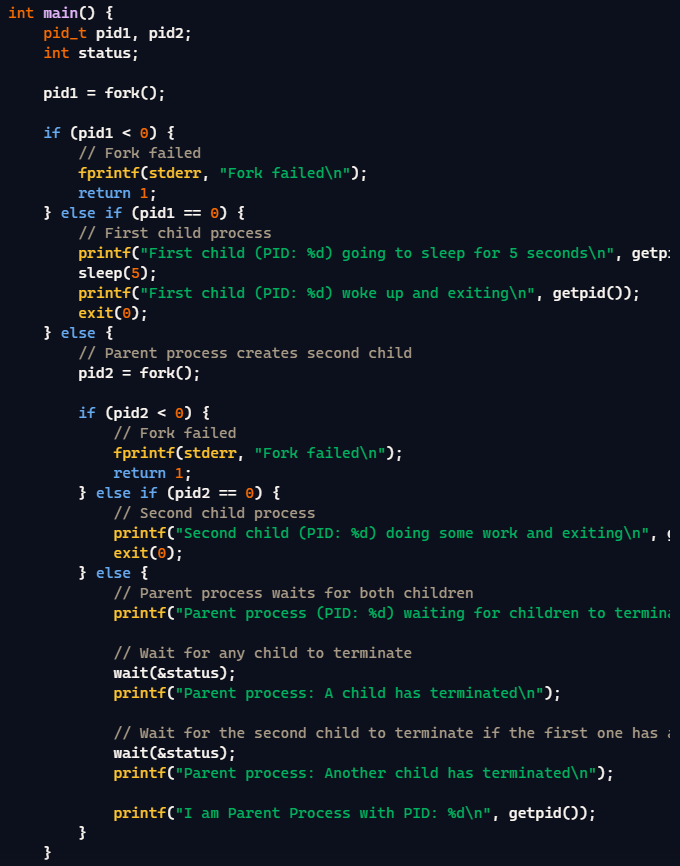
The **sleep timer decrements independently** of which process is currently executing. It is managed by the **operating system’s scheduler and timer interrupts**, not by the specific process that called sleep().

**How Sleep Works Internally**

1. **Process Calls sleep(n)**
   * The process moves to a **blocked state**.
   * The OS places it in the **sleep queue** and sets a countdown timer.
2. **Timer Interrupts Handle Decrementing**
   * The OS has a **system timer** (usually ticking every 1ms).
   * On each tick, the OS **decrements sleep counters** for all sleeping processes **regardless of which process is currently running**.
3. **Process Wakes Up**
   * Once the sleep counter reaches **zero**, the process is moved from the **blocked queue** to the **ready queue**.
   * The scheduler then decides **when** to run it.

**Key Point**

* **Sleep time decrements independently of execution**.
* Even if the **parent process is running**, the child's sleep timer continues to count down.
* If multiple processes call sleep(), the OS tracks them **individually** and wakes them up when their time expires.

**Output:**

Parent process (PID: %d) waiting for children to terminate.

First child (PID: %d) going to sleep for 5 seconds.

Second child (PID: %d) doing some work and exiting.

Parent process: A child has terminated.

First Child (PID: %d) woke up and exiting.

Parent process: Another child has terminated.

I am Parent Process with PID: %d

**Parallel Execution vs Sequential Execution**

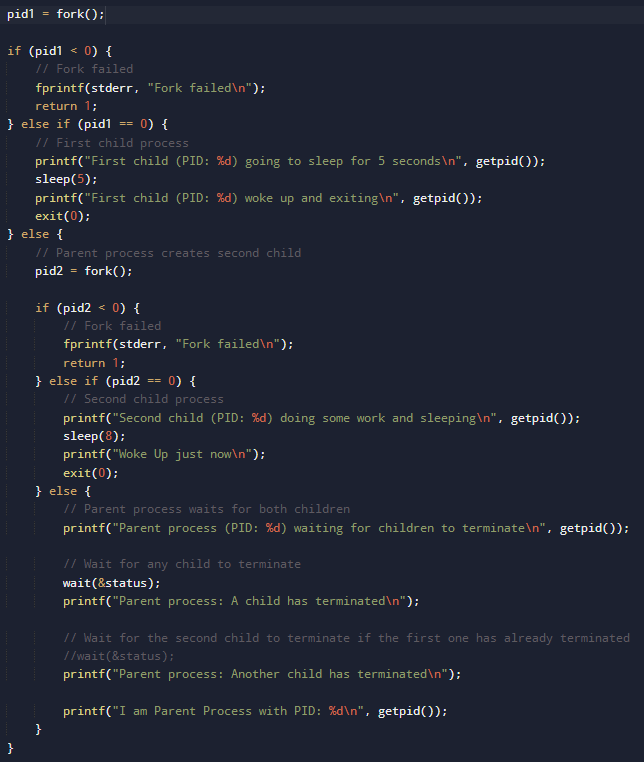
The **child and parent processes can execute in parallel** after a fork() system call, but their execution order depends on the **scheduler** of the operating system.

**How fork() Works**

When a process calls fork(), it creates a **child process** that is an exact copy of the **parent process**. Both processes start executing **from the next instruction after fork()**.

**Parallel Execution vs Sequential Execution**

* If the system has **multiple CPU cores**, the **parent and child can run truly in parallel**.
* If there is only **one CPU core**, the OS **context switches** between them, making it appear that they are running in parallel.

First child (PID: 31778) going to sleep for 5 seconds

Parent process (PID: 31777) waiting for children to terminate

Second child (PID: 31779) doing some work and sleeping

First child (PID: 31778) woke up and exiting

Parent process: A child has terminated

Parent process: Another child has terminated

I am Parent Process with PID: 31777

Woke Up just now

A close-up of a document

AI-generated content may be incorrect.

A screenshot of a checklist

AI-generated content may be incorrect.

**How scanf() Works in the Parent Process**

* scanf() **puts** the parent process to sleep when it is waiting for a user input.
* scanf("%d", &buffer[i]) waits for the user to enter an integer and press **Enter**.
* During this time, the parent process is **blocked** (put to sleep) by the operating system.
* It **remains in this blocked state** until input is provided. Meanwhile, the child process **immediately starts** execution when the parent is still sleeping.

**Does This Affect Process Execution?**

* **If the user delays input** → The parent stays in a **blocked state** (sleeping) and does not execute further instructions. It switches to the child process.
* Use scanf() before fork(), **this does NOT affect the child process**, since the child is only created **after the parent executes fork()**.