**1. Parallel Execution**

**Definition**

Parallel execution means multiple tasks are running **at the same time** on multiple processors or CPU cores.

**Characteristics**

* **True simultaneous execution** of multiple tasks.
* Requires **multiple CPU cores** or processors.
* Best suited for tasks that can be **divided into independent subtasks**.
* Improves performance by dividing workloads across cores.

**Example**

* A **video processing application** using multiple cores to encode different parts of a video at the same time.
* **GPU computing**, where multiple pixels are rendered simultaneously.

**2. Concurrent Execution**

**Definition**

Concurrency means multiple tasks are **in progress at the same time**, but they are not executing simultaneously. The system switches between tasks rapidly.

**Characteristics**

* **Task switching** happens on a single processor.
* Tasks appear to run at the same time but are actually **interleaved/switched**.
* Useful for handling **multiple tasks efficiently without true parallelism**.
* Often used in **multi-threading and event-driven programming**.

**Example**

* A **web server** handling multiple client requests by switching between them quickly.
* A **multitasking operating system**, where multiple applications (browser, music player, text editor) seem to run at the same time but actually take turns using the CPU.

**Key Differences**

| **Feature** | **Parallel Execution** | **Concurrent Execution** |
| --- | --- | --- |
| **Execution** | Tasks run **simultaneously** | Tasks are **interleaved/switched** |
| **CPU Requirement** | Requires multiple cores | Can run on a single core |
| **Efficiency** | Best for computational tasks | Best for I/O-bound tasks |
| **Example** | Video processing on multiple cores | Handling multiple user requests on a web server |

**Why Use Concurrency with Threads Instead of Parallelism?**

Parallelism is great when you have **multiple CPU cores**, but concurrency is often used instead of parallelism in **multi-threading** due to several reasons:

**1. Hardware Limitations (Not Enough Cores)**

* **Concurrency** works on **single-core and multi-core CPUs**, while **parallelism** requires multiple cores.
* If a system has **only one CPU core**, **true parallel execution isn’t possible**, but concurrency allows multiple tasks to make progress by switching between them.

✅ **Concurrency**: Works everywhere (single-core & multi-core).  
❌ **Parallelism**: Needs multiple cores to be effective.

**2. Efficient Resource Utilization**

* Many applications involve **I/O-bound tasks** (e.g., network requests, file operations).
* Instead of waiting for I/O operations to complete, concurrency allows another thread to use the CPU.
* **Parallelism wastes CPU time** if the workload is mostly I/O-based.

✅ **Concurrency**: Keeps the CPU busy while waiting for I/O.  
❌ **Parallelism**: Not useful when tasks spend most time waiting.

**3. Shared Memory & Synchronization Challenges**

* Parallel threads running at the **same time** can cause **race conditions**, where two threads modify shared data **simultaneously**, leading to **bugs**.
* **Concurrency allows controlled execution** (e.g., using locks, semaphores) to avoid such issues.
* **Parallelism requires careful synchronization**, which can slow down performance.

✅ **Concurrency**: Easier to manage shared resources.  
❌ **Parallelism**: Needs complex synchronization.

**4. Scheduling by OS**

* **The OS scheduler handles concurrency well** by switching between threads efficiently.
* Parallelism requires **manual workload division**, which may not always be efficient.

✅ **Concurrency**: OS manages it automatically.  
❌ **Parallelism**: Requires workload balancing.

**5. Most Applications Don’t Need True Parallelism**

* Many real-world applications (e.g., **web servers, GUIs, real-time systems**) do **not** need every thread to run at the exact same time.
* A **web server** handles multiple client requests using **concurrent threads**, but each request doesn’t require **full CPU power**.

✅ **Concurrency**: Ideal for responsiveness and real-time applications.  
❌ **Parallelism**: Overkill for many real-world scenarios.

**Example: Why Concurrency is Better in Some Cases**

Imagine a **restaurant**:

* **Concurrency:** One waiter handles multiple tables by switching between customers.
* **Parallelism:** Multiple waiters serve multiple tables at the same time.

If you only have **one waiter (one CPU core)**, **concurrency** is the only option. Even if you add more waiters, you still need concurrency to manage interactions efficiently.

**Conclusion**

* **Concurrency is more flexible** (works even on single-core CPUs).
* **Parallelism is better for CPU-intensive tasks**, but concurrency is better for **I/O-heavy tasks**.
* **Synchronization issues are easier to handle with concurrency** than with full parallel execution.

**1. Process Scheduling**

**Definition**

Process scheduling is the mechanism used by the operating system to select which process (a program in execution) gets access to the CPU at any given time.

**Characteristics**

* Deals with **independent programs**.
* Each process has its **own memory space** and system resources.
* Process switching requires a **context switch**, which has high overhead.
* The CPU scheduler selects processes from the **ready queue**.

**Types of Process Scheduling**

1. **Long-Term Scheduling**
   * Determines which processes are admitted into the system for execution.
   * Controls the degree of multiprogramming (number of processes in memory).
   * Example: When a user launches an application, the OS decides whether it should be loaded into memory.
2. **Short-Term Scheduling (CPU Scheduling)**
   * Determines which process gets CPU time.
   * Involves selecting a process from the ready queue for execution.
   * Example: Round Robin, First-Come-First-Serve (FCFS), Priority Scheduling.
3. **Medium-Term Scheduling**
   * Temporarily suspends processes (swapping) to optimize CPU and memory usage.
   * Example: When a background process is swapped out to free memory.

**2. Thread Scheduling**

**Definition**

Thread scheduling is the mechanism used by the operating system to determine which thread within a process gets CPU time.

**Characteristics**

* Deals with **multiple threads within a single process**.
* Threads share the **same memory space** and system resources.
* Thread switching has **lower overhead** compared to process switching.
* Threads within a process are scheduled based on priority and availability.

**Types of Thread Scheduling**

1. **User-Level Thread Scheduling**
   * Managed by a user-level thread library (e.g., POSIX threads in C).
   * The OS is unaware of individual threads.
   * Faster but lacks **true parallelism**.
2. **Kernel-Level Thread Scheduling**
   * Managed directly by the operating system.
   * OS schedules individual threads across different CPU cores.
   * True **parallel execution** is possible.

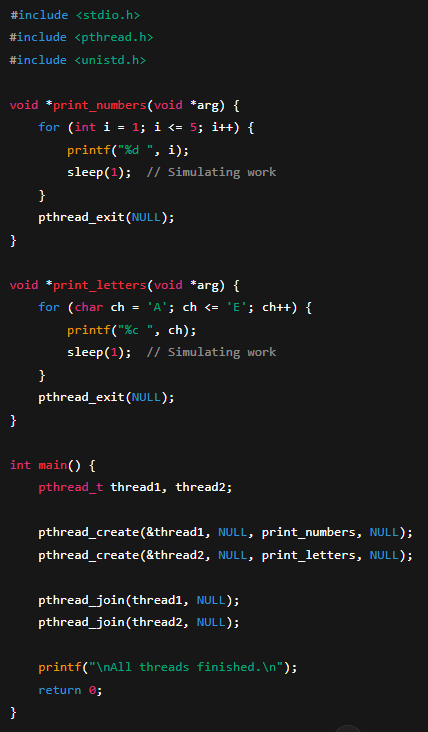
**Key Differences**

| **Feature** | **Process Scheduling** | **Thread Scheduling** |
| --- | --- | --- |
| **Scope** | Manages entire processes | Manages threads within a process |
| **Memory** | Each process has its own memory | Threads share the same memory |
| **Switching Overhead** | High (requires context switch) | Low (less context switching) |
| **Execution Model** | Parallel execution of processes | Concurrent execution within a process |
| **Control** | OS controls process execution | OS or user-level libraries control thread execution |

**Example**

Imagine you are running a **web browser**:

* **Process Scheduling** decides when the browser process gets CPU time.
* **Thread Scheduling** decides when different threads (e.g., UI thread, network thread, JavaScript execution thread) get CPU time within the browser.



A black background with pink and yellow letters

AI-generated content may be incorrect.

**Why Do Processes Use Parallel Execution While Threads Do Not?**

Processes can execute in **true parallelism**, while threads usually rely on **concurrency** instead. The key differences lie in **memory separation, CPU scheduling, and execution models**.

**🔹 1. Processes Have Independent Memory, Threads Share Same Memory**

* **Processes:** Each process has its **own memory space** (allocated by the OS). This allows them to run **truly in parallel** on multiple CPU cores without interfering with each other.
* **Threads:** Threads **share the same memory** within a process. If multiple threads modify shared data at the same time, **race conditions** can occur, leading to unpredictable behavior.

✅ **Processes**: No memory conflicts, easier to run in parallel.  
❌ **Threads**: Shared memory makes true parallelism risky.

**🔹 2. Processes Run on Separate CPU Cores, Threads Often Do Not**

* **Processes:** The OS scheduler can assign **different processes to different CPU cores**, achieving true parallel execution.
* **Threads:** The OS may keep threads within the same process running on **one core at a time** due to shared memory and synchronization constraints.

✅ **Processes**: Fully utilize multi-core CPUs.  
❌ **Threads**: Often scheduled on the same core.

**🔹 3. Threads Need Synchronization, Processes Do Not**

* **Processes:** Since they have **separate memory**, they don't need to worry about synchronization when running in parallel.
* **Threads:** Since they share memory, **synchronization tools** (locks, semaphores, mutexes) are required to prevent **race conditions**.

✅ **Processes**: No need for complex synchronization.  
❌ **Threads**: Synchronization slows down execution.

**🔹 4. OS Scheduling Differences**

* **Processes:** The OS treats **each process as an independent unit** and can schedule them across different cores for true parallel execution.
* **Threads:** The OS sees **threads as part of the same process**, so it often schedules them to **run sequentially** or switches between them using **concurrency** instead of parallelism.

✅ **Processes**: The OS can freely distribute them across cores.  
❌ **Threads**: The OS tries to keep them in one core due to shared resources.