Parallel & Distributed Systems

Lecture # 3

By,

Dr. Ali Akbar Sidduque

Introduction to Threads

01

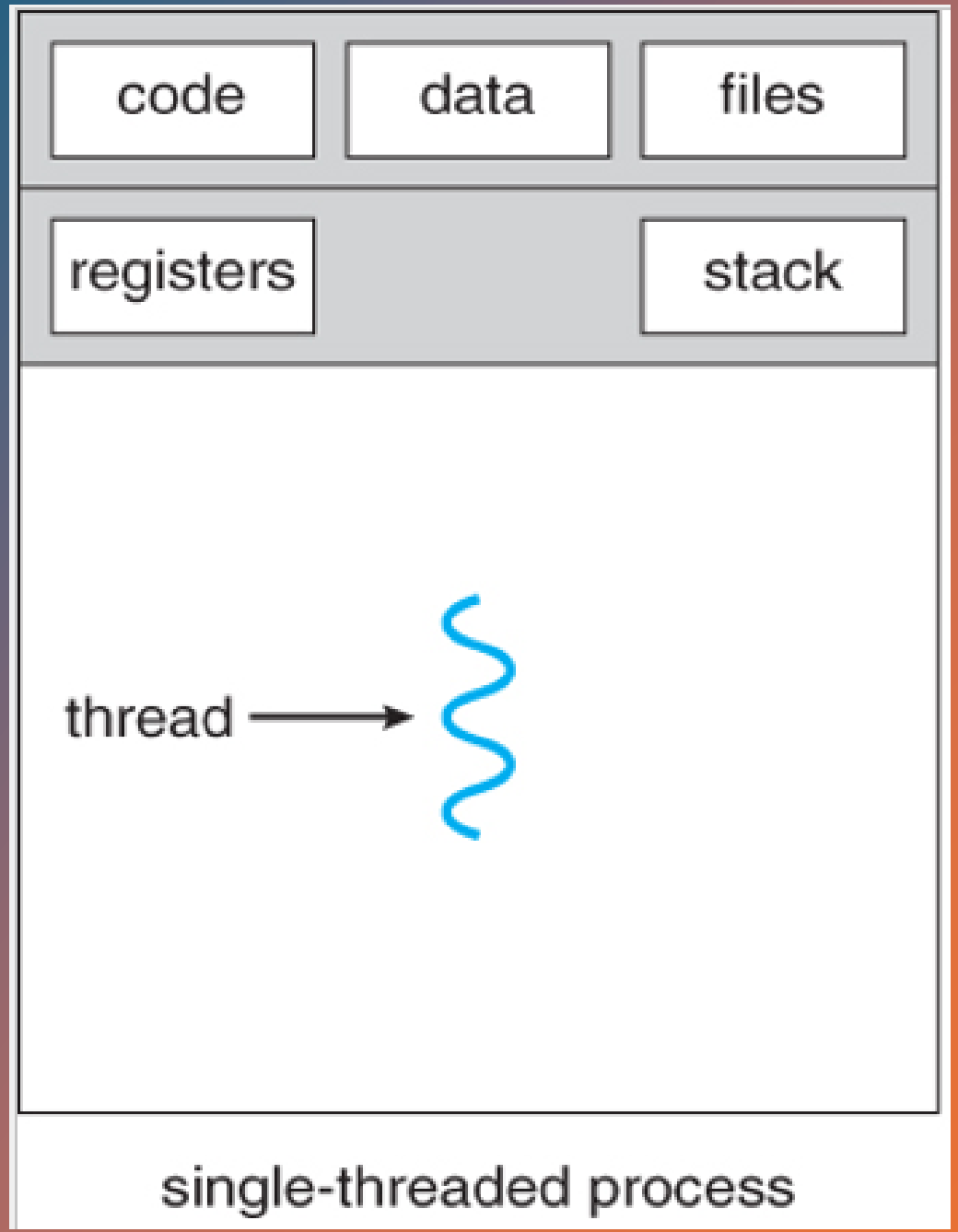
A thread is a lightweight process that runs independently within a program.

02

Threads share the same memory space but can execute different parts of a program simultaneously.

03

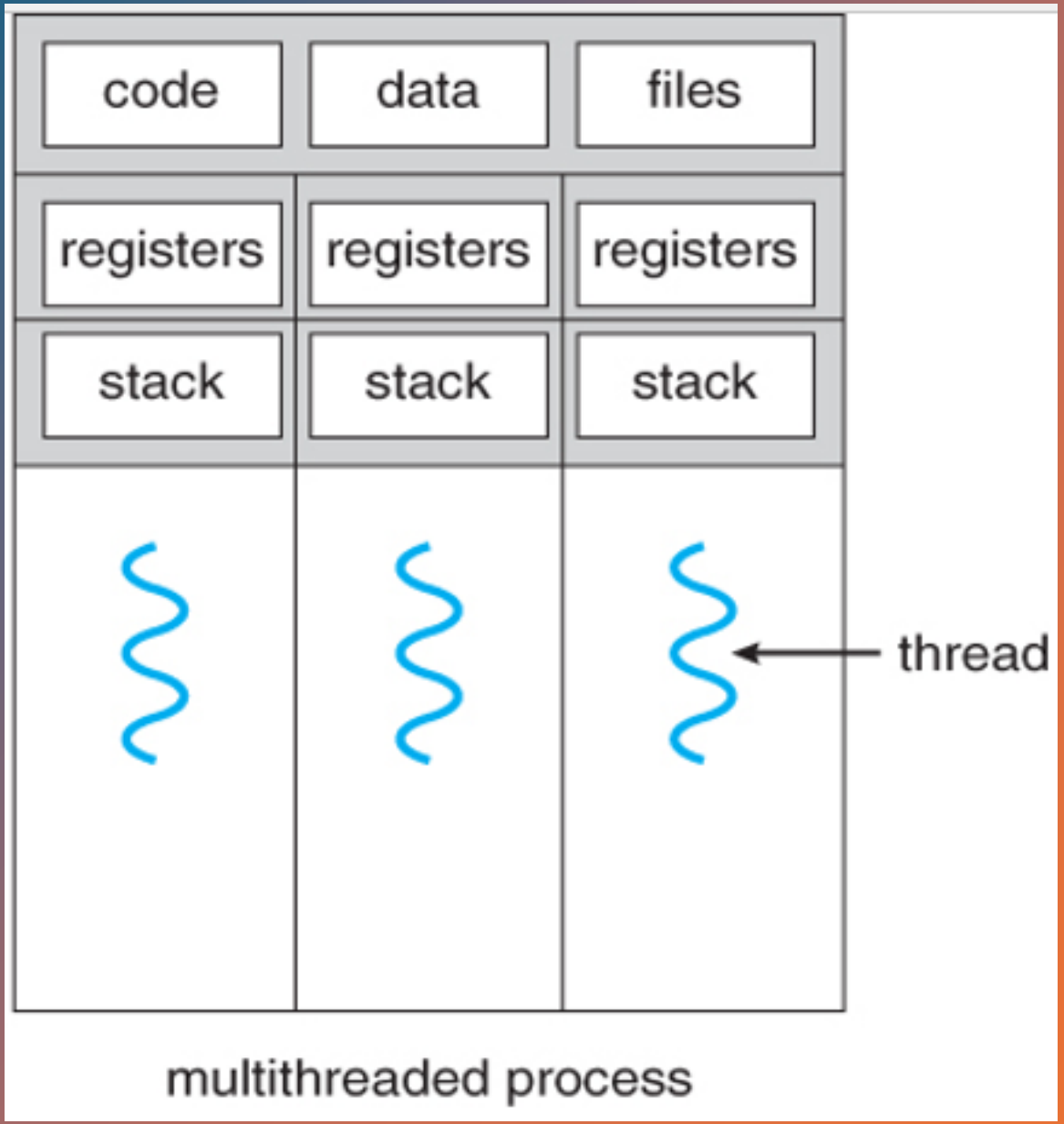
Used in modern computing to improve performance and responsiveness.



Single-threaded

Single-Threaded Process (Left Side):

- A single-threaded process contains only **one thread of execution**.
- It has a **code section, data section, and file section** that represent the program's instructions and resources.
- The **stack and registers** are dedicated to the single thread, meaning only one sequence of instructions is executed at a time.
- Since there is only **one execution thread**, tasks must be completed sequentially.



Multi-threaded processes

- A multi-threaded process consists of multiple threads within the same process.
- The code, data, and file sections are shared among all threads, allowing for efficient resource utilization.
- Each thread has its own stack and register set, enabling concurrent execution of different tasks within the same application.
- Advantage: Multi-threading improves performance by executing multiple operations in parallel, reducing CPU idle time and enhancing responsiveness.

Key Takeaways

- Single-threaded processes execute tasks one after another, leading to potential inefficiencies.
- Multi-threaded processes allow multiple tasks to run concurrently within the same program, improving performance and responsiveness.
- Multi-threading is commonly used in applications like web servers, parallel computing, and real-time systems to handle multiple operations simultaneously.

Types of Threads

Feature	User Threads	Kernel Threads
Definition	Threads managed by user-level libraries without kernel intervention.	Threads managed directly by the OS kernel.
Management	Managed by the user-space thread library (e.g., POSIX Pthreads, Java threads).	Managed by the operating system.
Speed	Faster as context switching does not involve the kernel.	Slower due to kernel involvement in scheduling.
Dependency	If one thread blocks, all threads in the process block.	If one thread blocks, other threads in the process continue execution.
Portability	More portable across different OS platforms.	Less portable as they are OS-dependent.
System Calls	Cannot take advantage of multi-core processors since the kernel sees only a single-threaded process.	Can run on multiple processors simultaneously, improving performance.
Example	Java Virtual Machine (JVM) threads, Green Threads.	Linux pthreads, Windows threads.

Thread Lifecycle

New: Thread is created but not yet started.

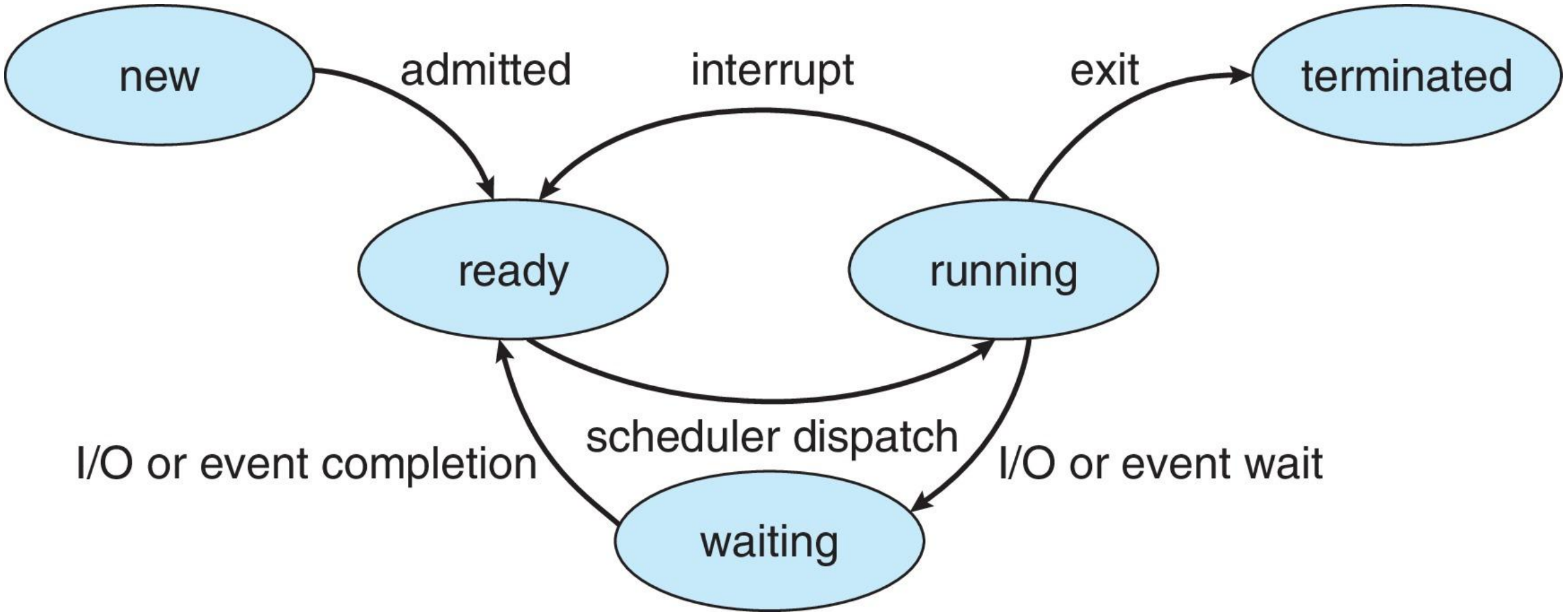
Ready: Thread is waiting for CPU allocation.

Running: Thread is executing instructions.

Blocked: Waiting for a resource or input.

Terminated: Execution completed or manually stopped.

Thread Lifecycle



Thread Life Cycle

1. **New State**:
   - The process is created but not yet ready for execution.
   - It waits for admission into the ready queue by the operating system.
2. **Ready State:**
   - The process is loaded into memory and waiting for CPU time.
   - It is ready to be assigned to a processor by the CPU scheduler.
3. **Running State**:
   - The process is currently executing on the CPU.
   - If it completes execution, it moves to the **terminated** state.
   - If an **interrupt** occurs, it moves back to the **ready** state.
4. **Waiting State**:
   - The process is waiting for an I/O operation or an event to occur.
   - It cannot execute until the required operation is completed.
5. **Terminated State**:
   - The process has completed execution or has been forcefully stopped.
   - It is removed from memory.
6. **Transitions Between States**:
   - **Admitted**: A process moves from **new** to **ready** state when it is scheduled.
   - **Scheduler Dispatch**: Moves a process from **ready** to **running** when the CPU is allocated.
   - **Interrupt**: A process in **running** can be preempted and moved back to **ready**.
   - **I/O or Event Wait**: A process moves from **running** to **waiting** if it requires an I/O operation.
   - **I/O or Event Completion**: The process returns from **waiting** to **ready** when the I/O task is completed.
   - **Exit**: A process moves from **running** to **terminated** once execution is complete.

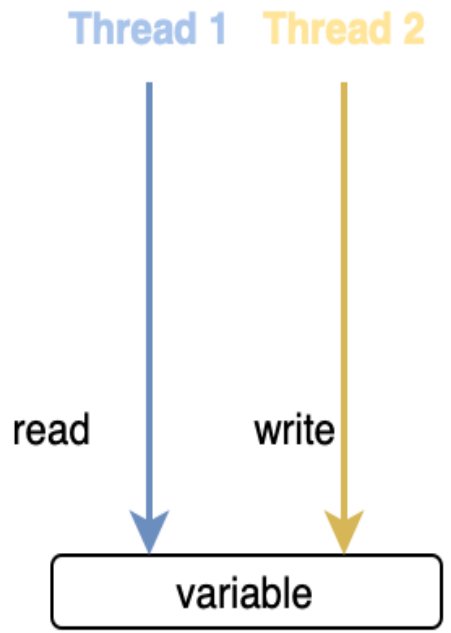
Why Use Threads?

- Improved **performance** via parallel execution.
- **Efficient resource utilization** (same memory space shared).
- **Better responsiveness** in applications like UI-based systems.
- **Simplifies complex tasks** by breaking them into smaller units.

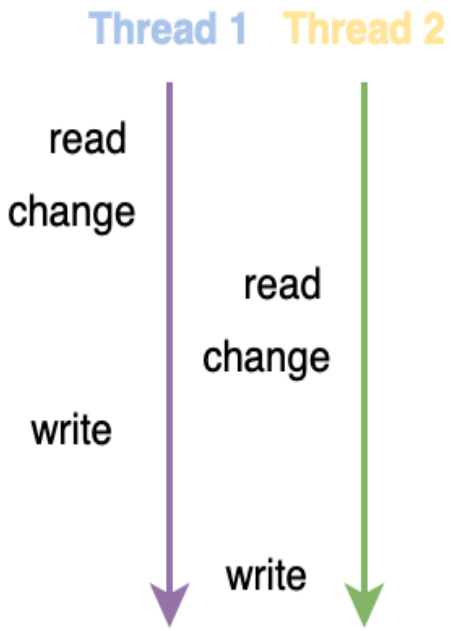
What is Synchronization?

- Synchronization ensures orderly access to shared resources.
- Prevents **race conditions**, **deadlocks**, and **inconsistent data states**.
- Common in **multithreaded applications** and **parallel computing**.

Data races



Race condition



Data Races vs. Race Condition

Data Races:

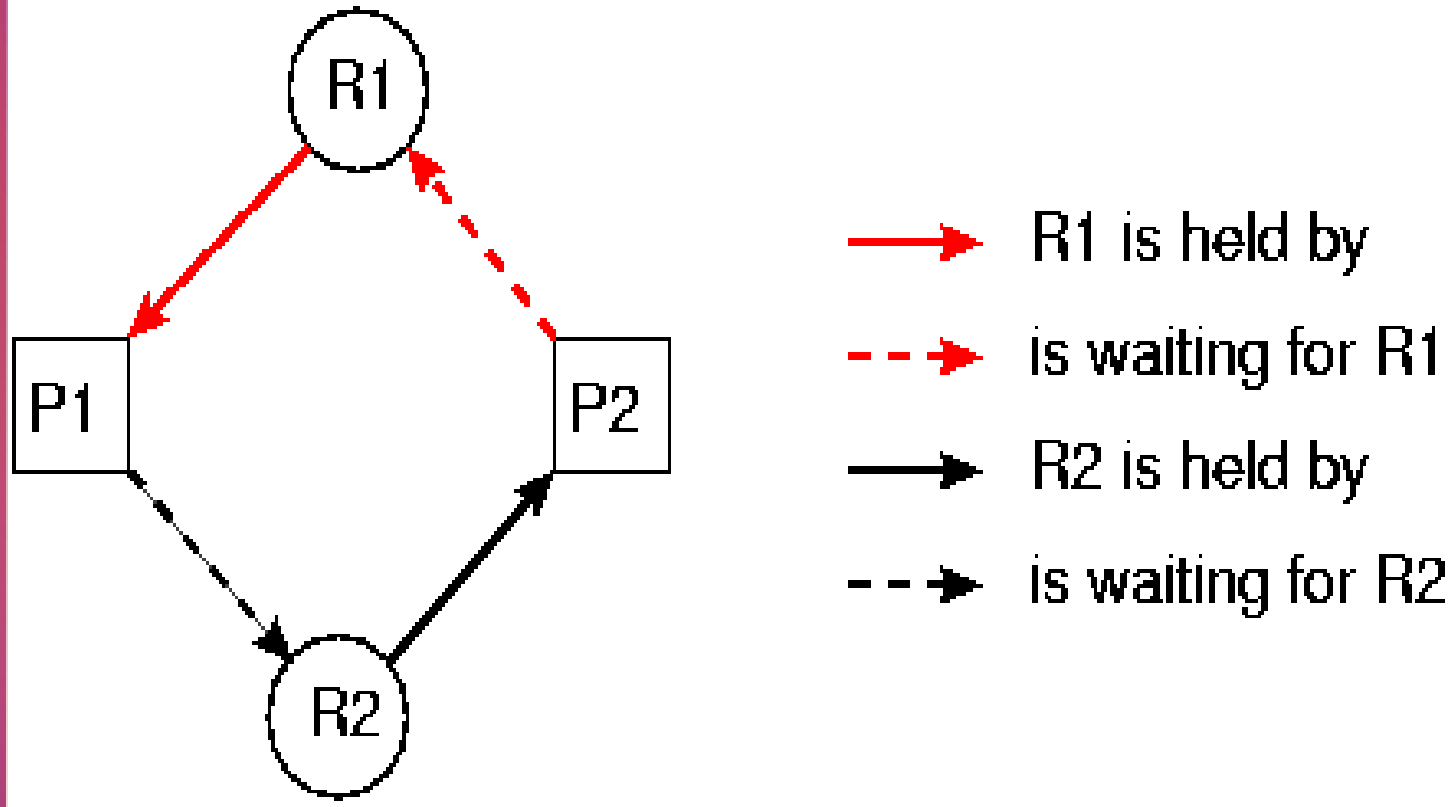
- Occurs when two threads access a shared variable concurrently.
- One thread reads while another writes at the same time.
- Leads to unpredictable or inconsistent values in the variable.

Race Condition:

- Happens when the outcome depends on the sequence/timing of threads.
- Both threads read, modify, and write the variable without proper synchronization.
- Can cause unexpected behaviors or incorrect results in a program.

Race Conditions & Deadlocks

- **Race Condition:** Two or more threads access a shared variable simultaneously, leading to unpredictable results.
- **Deadlock:** Two or more threads wait indefinitely for resources locked by the other.

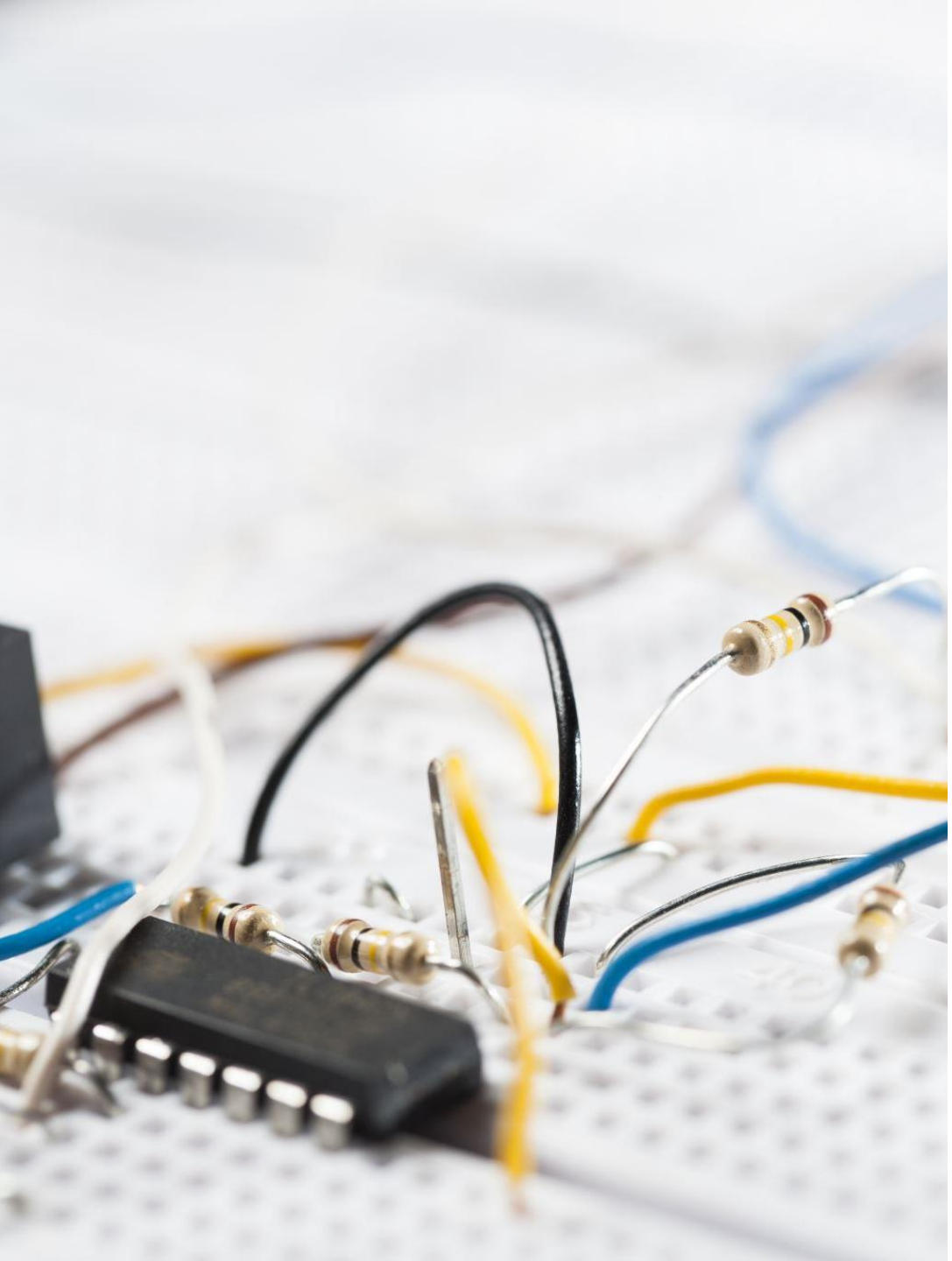


Synchronization Mechanisms

1. **Mutexes (Mutual Exclusion):** Locking mechanism to ensure only one thread accesses a resource.
2. **Semaphores:** Signaling mechanism for limited resource access.
3. **Monitors:** High-level synchronization that encapsulates shared resources.
4. **Atomic Variables:** Prevents partial updates to shared data.

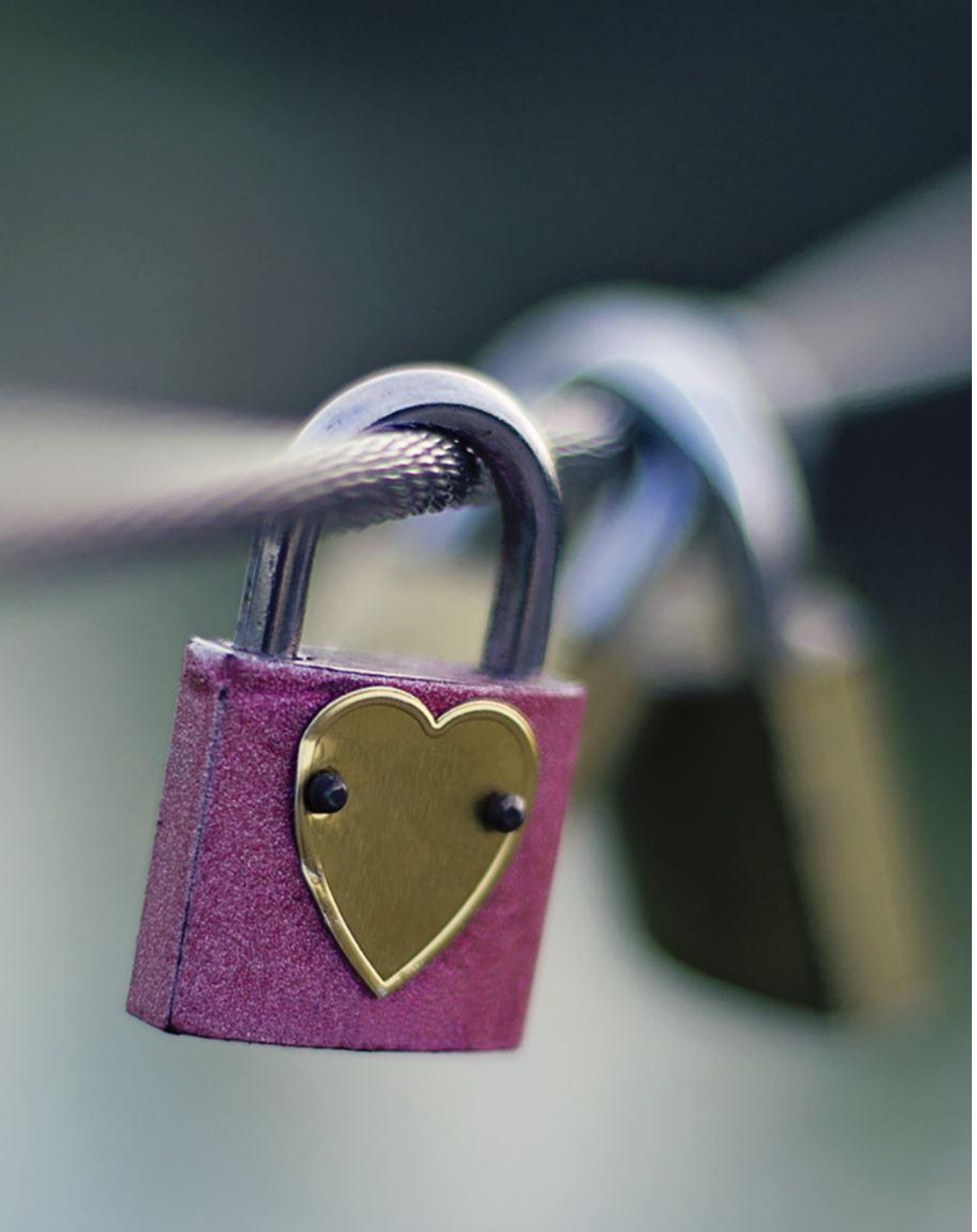
Implementing Synchronization in Python

```
import threading
lock = threading.Lock()
def critical_section():
    with lock:
        # Critical section code
        print("Thread is running safely")
thread =
threading.Thread(target=critical_section)
thread.start()
thread.join()
```



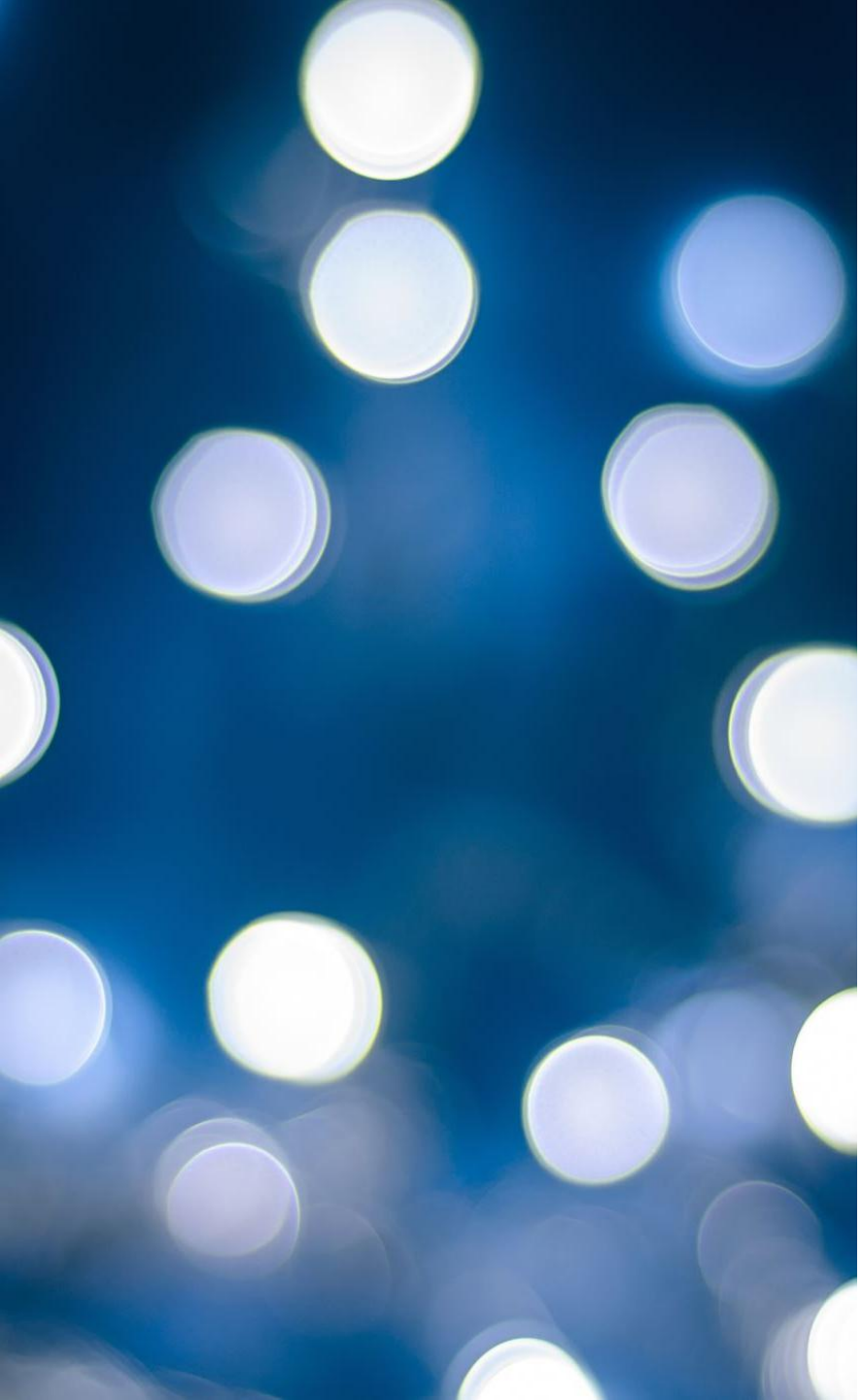
1. Importing the Threading Module

- The `threading` module provides tools for managing threads in Python.
- Used to implement synchronization mechanisms like Locks and Semaphores.



2. Creating a Lock

- `lock = threading.Lock()` creates a Lock object.
- Prevents multiple threads from accessing shared resources simultaneously.



3. Defining the Critical Section

- `with lock:` ensures only one thread enters the critical section at a time.

- Automatically releases the lock after execution to prevent deadlocks.

4. Creating and Starting a Thread

- `thread = threading.Thread(target=critical_section)` creates a new thread.

- `thread.start()` starts execution in a separate thread.

5. Waiting for the Thread to Finish

- `thread.join()` blocks the main program until the thread completes execution.

- Ensures the program does not exit prematurely.

Key Takeaways

- - Locks prevent race conditions by allowing only one thread to execute at a time.
- - `with lock:` ensures safe access to shared resources.
- - `thread.start()` runs the thread, and `thread.join()` waits for it to finish.