

Multithreading Models

- Some operating systems use a **combination of User-Level Threads (ULT) and Kernel-Level Threads (KLT)**.
- **Solaris** is a well-known example of a combined threading model.
- Threads within the same application can run **simultaneously on multiple processors**.
- A **blocking system call** by one thread does not block the entire process.
- The model provides **better concurrency and improved system performance**.
- It supports **parallel execution** in multiprocessor environments.
- The operating system kernel manages kernel threads, while the thread library manages user threads.
- Combined threading offers a balance between **high performance and flexibility**.

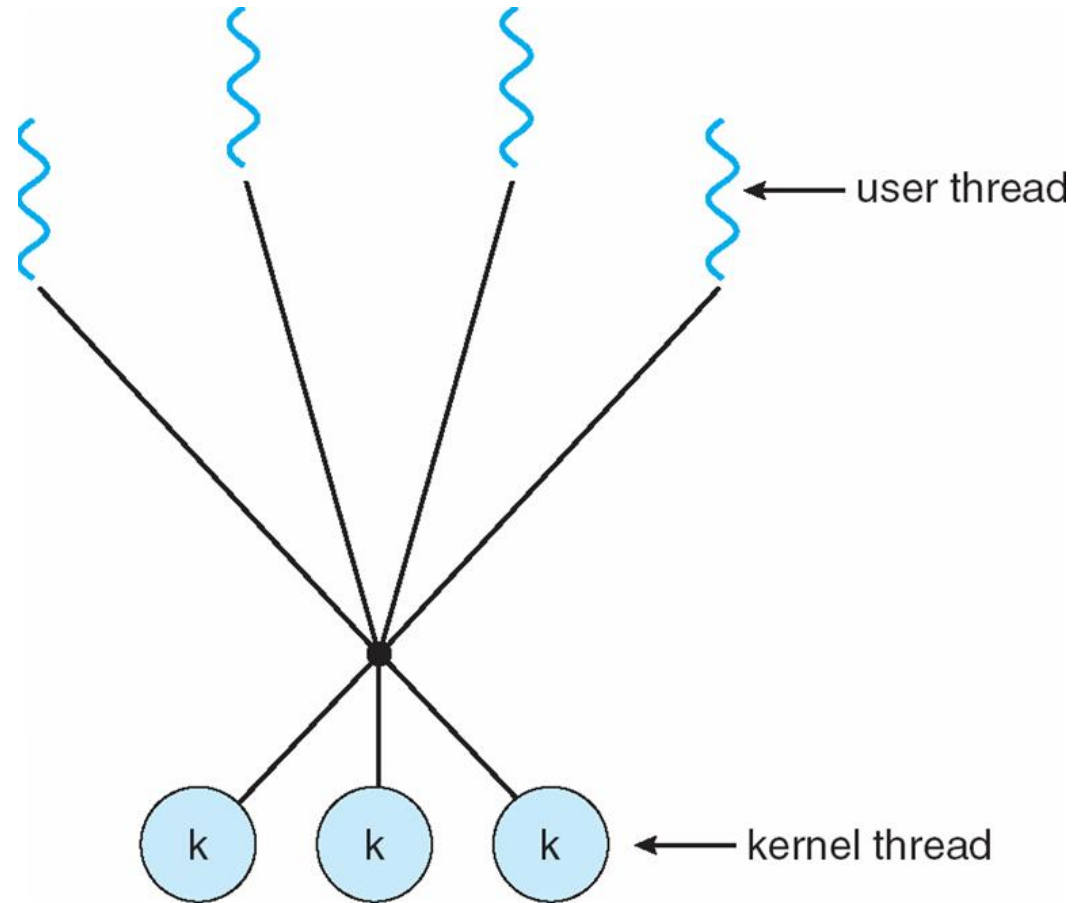
Types of Multithreading Models

- 1. Many-to-One Model**
- 2. One-to-One Model**
- 3. Many-to-Many Model**

Many to Many Model

- The **Many-to-Many (M:M) model** maps multiple user threads to an equal or smaller number of kernel threads.
- It allows developers to create **a large number of user-level threads** as needed.
- Multiple user threads can be **multiplexed over multiple kernel threads**.
- Kernel threads can execute **in parallel on multiprocessor systems**.
- This model provides **high concurrency and better resource utilization**.
- If one thread performs a **blocking system call**, other threads can continue execution.
- The kernel can schedule another thread while the blocked thread waits.
- It combines the advantages of both **User-Level Threads** and **Kernel-Level Threads**.
- The model offers **better scalability and performance** for multithreaded applications.
- It is considered one of the most **flexible and efficient threading models** for parallel processing.

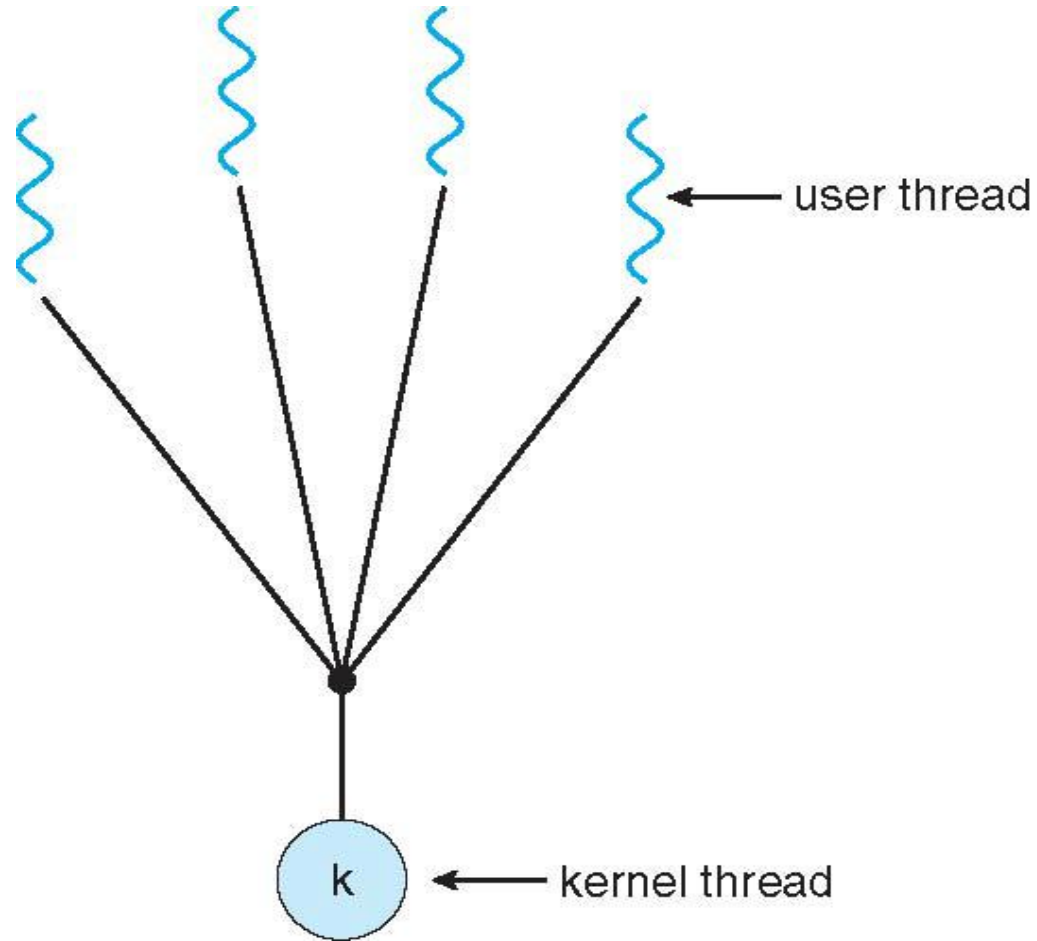
Many to Many Model



Many-to-One (M:1) model

- The **Many-to-One (M:1) model** maps multiple user-level threads to a **single kernel thread**.
- Thread management is performed in **user space** by a thread library.
- The operating system kernel is **not aware of individual user threads**.
- Multiple user threads share **one kernel thread** for execution.
- If one thread makes a **blocking system call**, the entire process is blocked.
- Only **one thread can access the kernel at a time**.

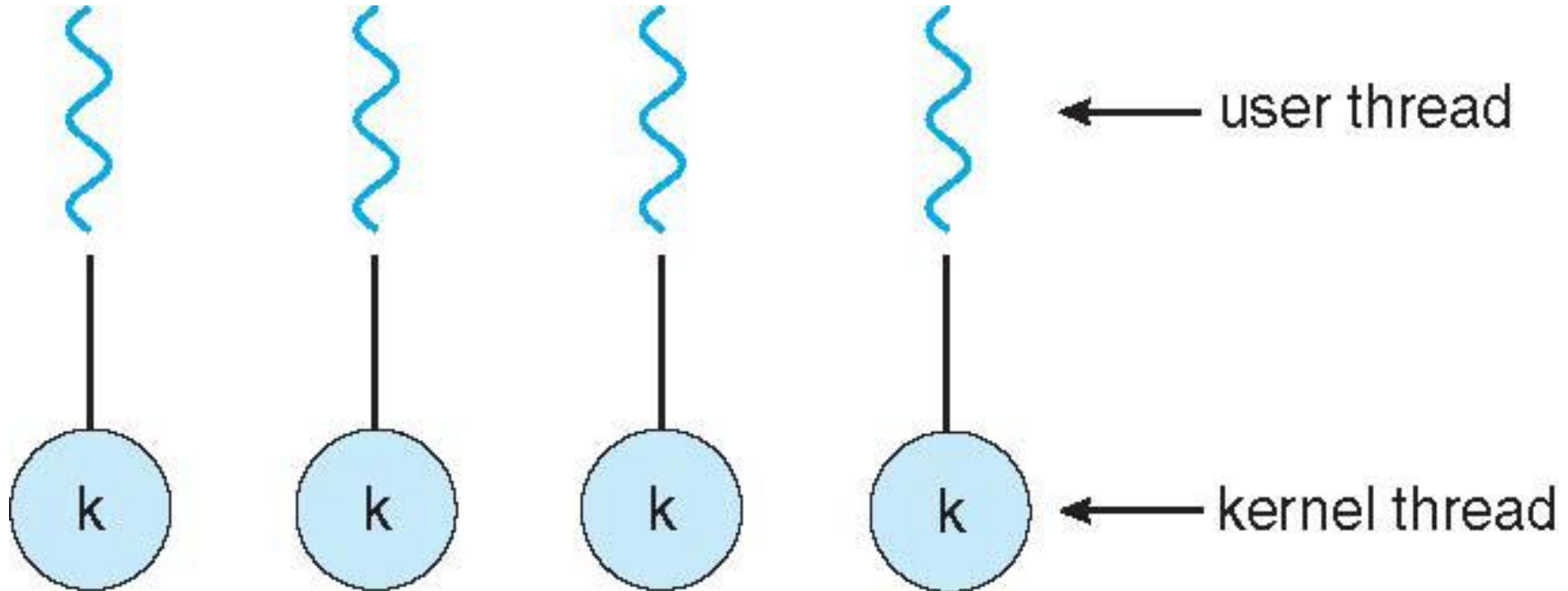
Many-to-One (M:1) model



One-to-One (1:1) model

- The **One-to-One (1:1) model** maps each user-level thread to a separate kernel-level thread.
- Every user thread has a **corresponding kernel thread**.
- This model provides **greater concurrency** than the Many-to-One model.
- If one thread performs a **blocking system call**, other threads can continue executing.
- Multiple threads can run **simultaneously on multiprocessor or multicore systems**.
- The operating system kernel manages and schedules each thread independently.
- Large numbers of threads may consume more system resources.
- Used in operating systems such as **Windows NT, Windows 2000**.

One-to-One (1:1) model

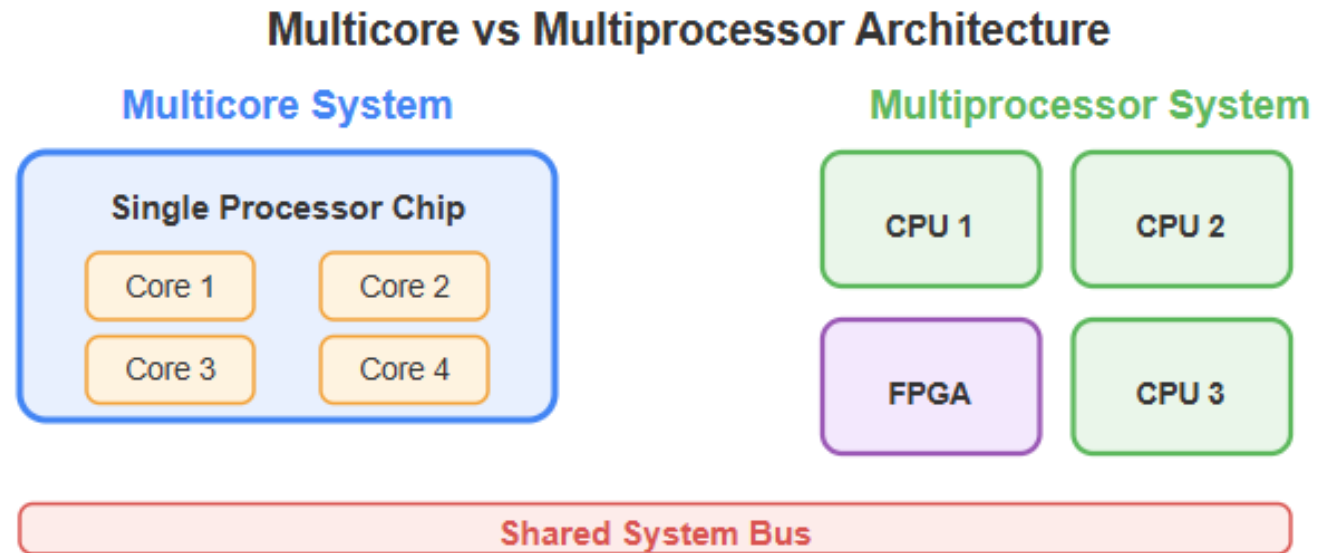


Multicore Programming

- Multicore programming is a technique for developing software that runs on **multiple CPU cores simultaneously**.
- It enables **concurrent execution** of multiple tasks or threads.
- Multiple cores can work in parallel, resulting in **faster program execution**.
- It improves **system performance and efficiency**.
- Applications become more **responsive**, especially when handling multiple tasks.
- It makes better use of modern **multicore and multiprocessor systems**.
- Workloads can be divided among cores to achieve **parallel processing**.
- Commonly used in operating systems, web servers, databases, and high-performance applications.

Multicore vs Multiprocessor Systems

- A **multicore processor system** contains a single processor chip with multiple execution cores, while a **multiprocessor system** has multiple separate processors on the motherboard or chip.



Concurrent Execution and Multitasking

- Concurrent systems created using multicore programming have multiple tasks executing in parallel, known as **concurrent execution**.
- When a processor executes multiple parallel tasks, this is called **multitasking**.
- The **CPU scheduler** manages these parallel tasks.
- Tasks can execute independently while maintaining data transfer capabilities between them.

Implicit Threading

- Implicit threading is a technique where the **creation and management of threads are handled automatically** by the compiler, runtime library, or operating system.
- Programmers focus on **defining tasks**, rather than managing threads directly.
- It reduces the complexity of **multithreaded programming**.
- Helps minimize common threading errors such as **race conditions and deadlocks**.
- Improves utilization of **multicore processors** by automatically distributing work among available cores.
- Makes parallel programming easier, more efficient, and more portable across different hardware platforms.

Thread Issues

- Thread issues refer to the **challenges and problems** encountered in multithreaded programming.
- **Fork() and Exec() system calls** affect the creation and execution of threads in a process.
- **Thread cancellation** allows one thread to terminate another thread before it completes.
- **Thread pools** improve performance by reusing existing threads instead of creating new ones.

Thank You
Any Questions?



بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ، وَ عَلَى رَسُولِهِ الْكَرِيمِ، وَ عَلَى آلِهِ وَصَلِّهِ الْجَمْعَيْنِ، أَمَّا بَعْدُ:

Operating System Concept

Process Synchronization

Week 05

Week 5 Course Outline

- Process Synchronization
- The Critical Section Problem
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic problem of Synchronization



Learning Objectives

Process synchronization

- Process synchronization is used to **coordinate multiple processes or threads**.
- It ensures that **shared data is accessed safely**.
- It prevents processes from **interfering with each other**.
- It helps avoid **data inconsistency and errors**.
- Synchronization is needed when processes **share resources** such as files, memory, or variables.
- It controls the order in which processes execute **critical operations**.
- Common synchronization tools include **mutexes, semaphores, and monitors**.
- It improves the **correctness and reliability** of a system.

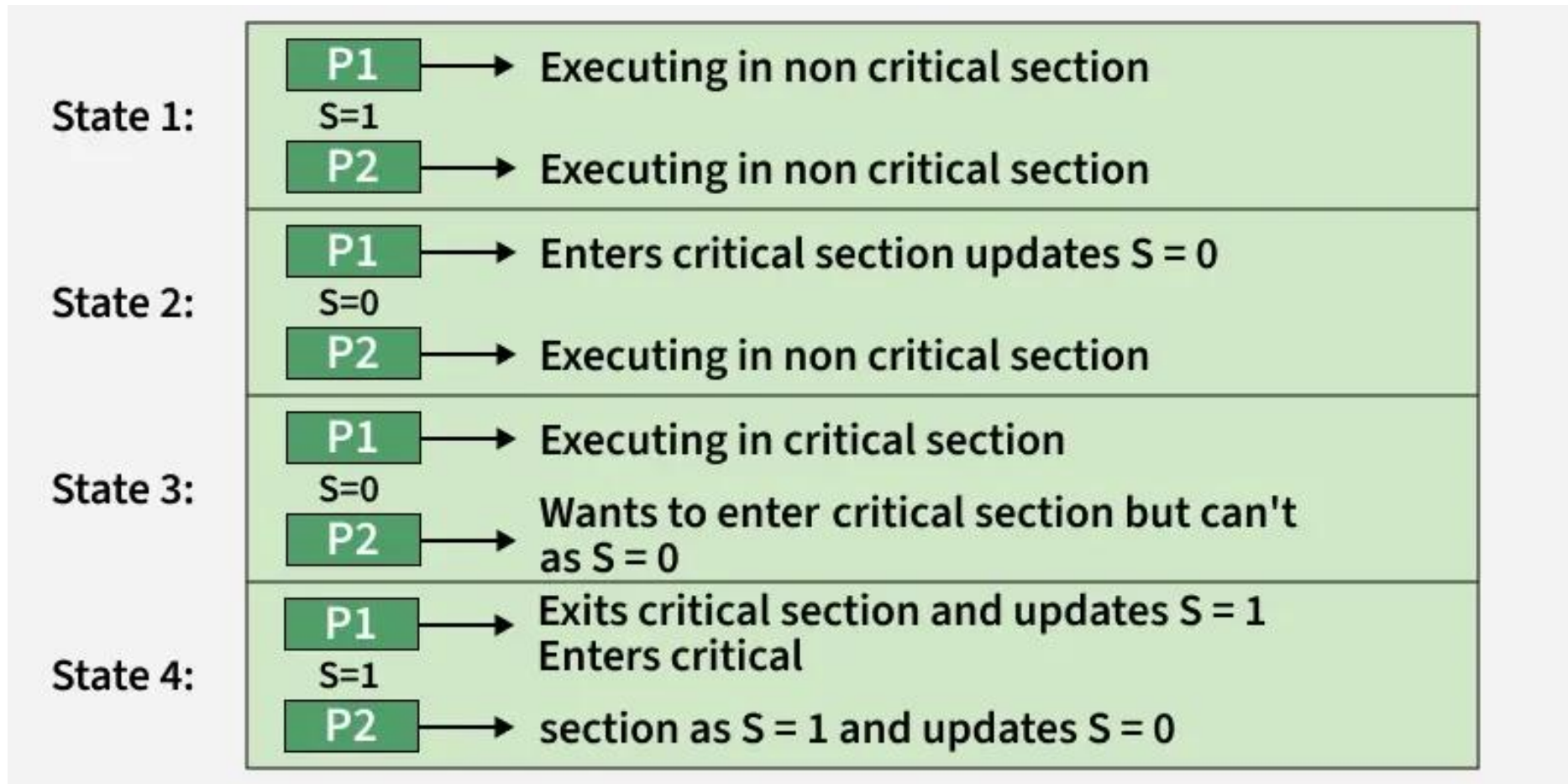
The Critical Section Problem

What is it?

The Critical Section Problem is about making sure that when multiple processes or threads share data, only one can access the shared data at a time.

- A critical section is a part of a program that accesses shared data.
- Only one process or thread should enter the critical section at a time.
- It prevents data conflicts and errors.
- It ensures data remains correct and consistent.
- It is a key problem in process synchronization.

The Critical Section Problem



Synchronization Hardware

What is it?

Synchronization hardware provides special CPU instructions that help processes and threads access shared resources safely.

- It supports process synchronization at the hardware level.
- It prevents multiple processes from modifying shared data simultaneously.
- It helps avoid race conditions.
- It improves system efficiency and performance.

Semaphores

What is it?

A Semaphore is a synchronization tool used to control access to shared resources by multiple processes or threads.

- It uses a counter to manage resource access.
- It helps coordinate multiple processes.
- It prevents race conditions.
- It uses two operations: Wait() and Signal().

Mutex Locks

What is it?

A Mutex Lock is a locking mechanism that allows only one process or thread to use a shared resource at a time.

- Mutex stands for Mutual Exclusion.
- Only one thread can hold a mutex lock at a time.
- A thread must lock before entering the critical section.
- The lock is released after the task is completed.
- It prevents data inconsistency and conflicts.