OPERATING SYSTEMS

Threads and Concurrency 01

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

OPERATING SYSTEMS

Course Syllabus - Unit 2

UNIT 2: Threads and Concurrency

Introduction to Threads, types of threads, Multicore Programming, Multithreading Models, Thread creation, Thread Scheduling, PThreads and Windows Threads, Mutual Exclusion and Synchronization: software approaches, principles of concurrency, hardware support, Mutex Locks, Semaphores. Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts. Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads. Deadlocks: principles of deadlock, tools for detection and Prevention.

OPERATING SYSTEMS

Course Outline - Unit 2

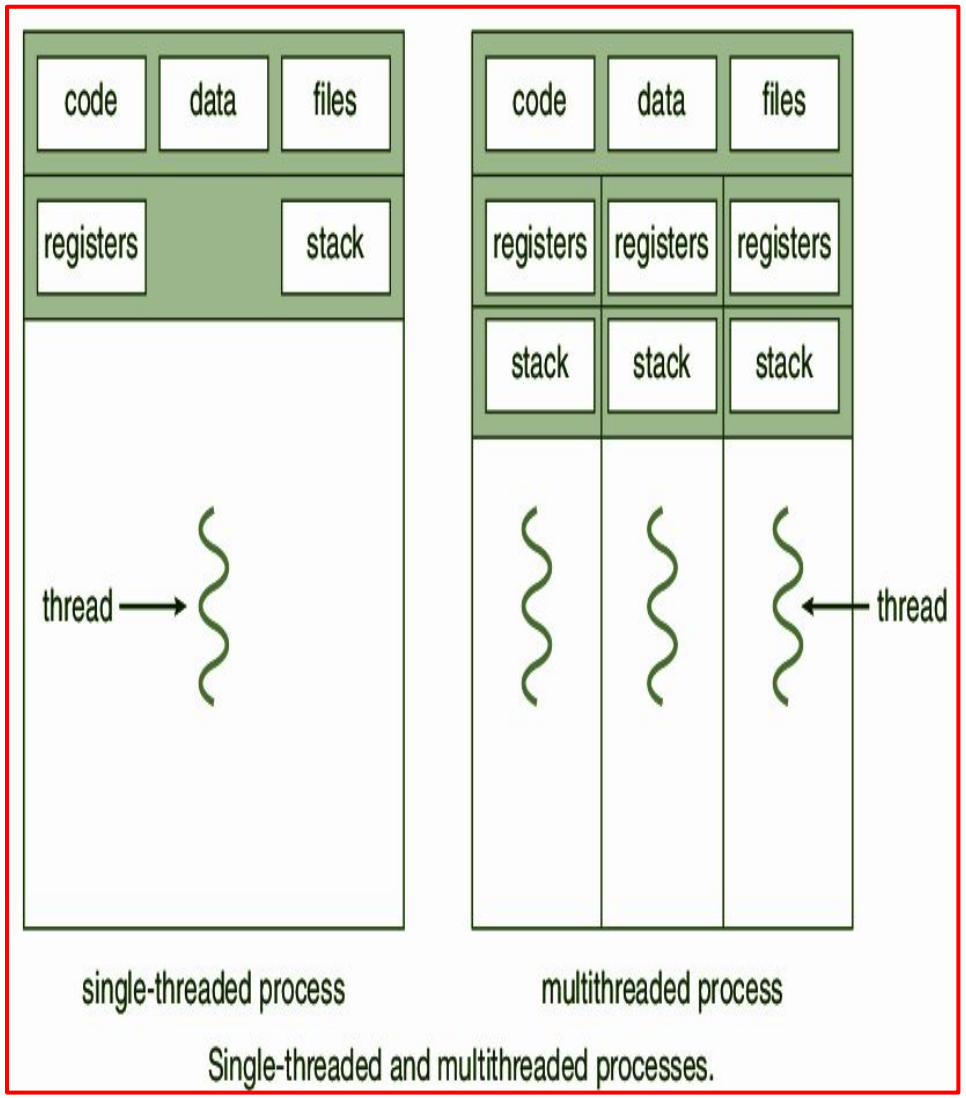
13	4.1,4.2	Introduction to Threads, types of threads, Multicore Programming.	4	42.8
14	4.3,5.4	Multithreading Models, Thread creation, Thread Scheduling	4	
15	4.4	Pthreads and Windows Threads	4	
16	6.1-6.3	Mutual Exclusion and Synchronization: software approaches	6	
17	6.3-6.4	principles of concurrency, hardware support	6	
18	6.5,6.6	Mutex Locks, Semaphores	6	
19	6.7.1-6.7.3	Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts	6	
20	6.9	Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6	
21	Handouts	Demonstration of programming examples on process synchronization		
22	7.1-7.3	Deadlocks: principles of deadlock, Deadlock Characterization.	7	
23	7.4	Deadlock Prevention, Deadlock example	7	
24	7.6	Deadlock Detection	7	

Topics Outline

- Threads
  - Overview
  - Motivation
  - Benefits
- Multicore Programming
  - Programming Challenges
  - Types of Parallelism

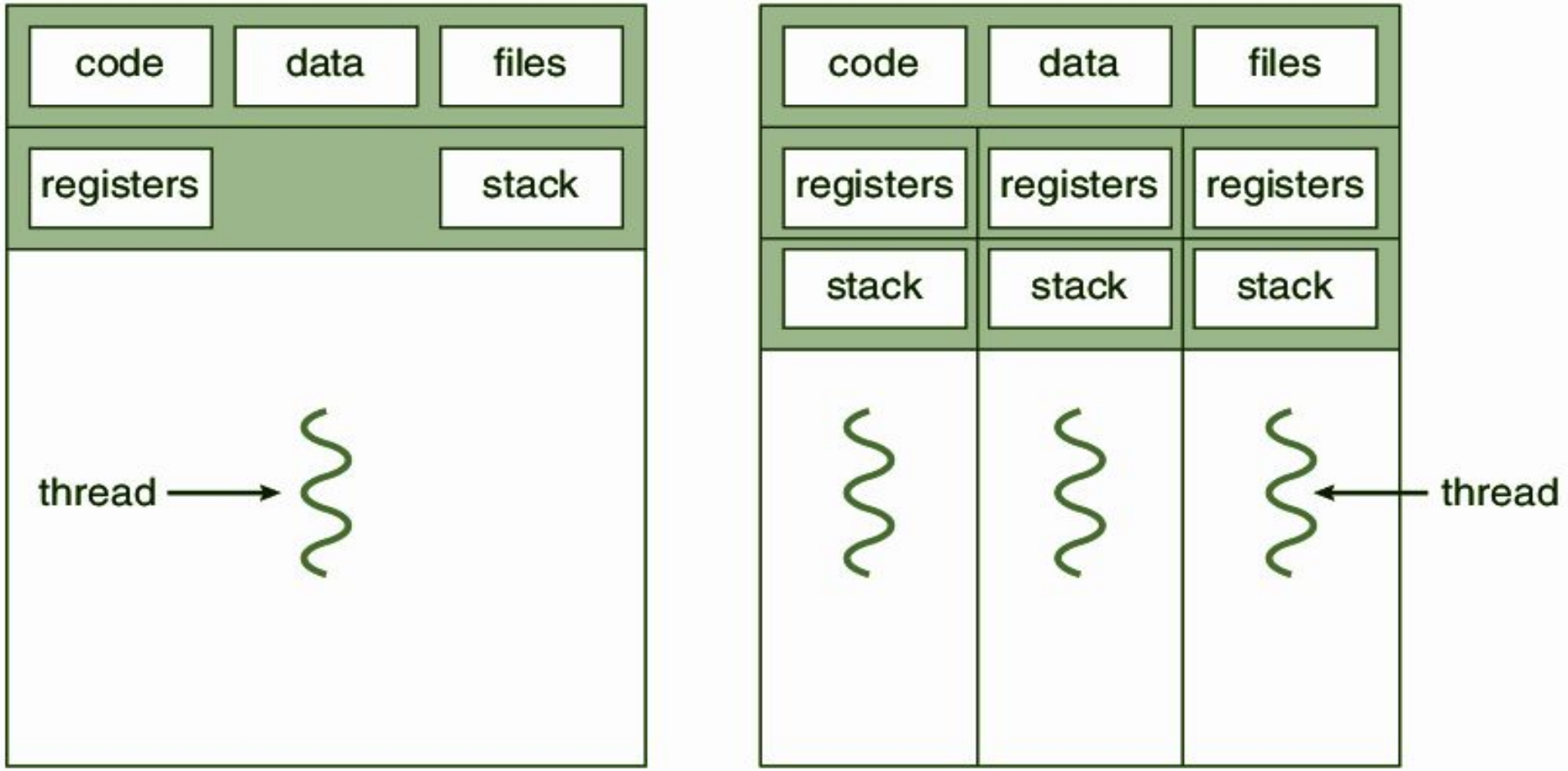
Threads Overview

- The process model introduced so far assumed that a process was an executing program with a single thread of control.
- Virtually all modern operating systems, however, provide features enabling a process to contain multiple threads of control.
- A **thread** is a basic unit of CPU utilization; it comprises a **thread ID** , **a program counter**, a **register set**, and a **stack**.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional (or heavyweight) process has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time.



Single-threaded and multithreaded processes.

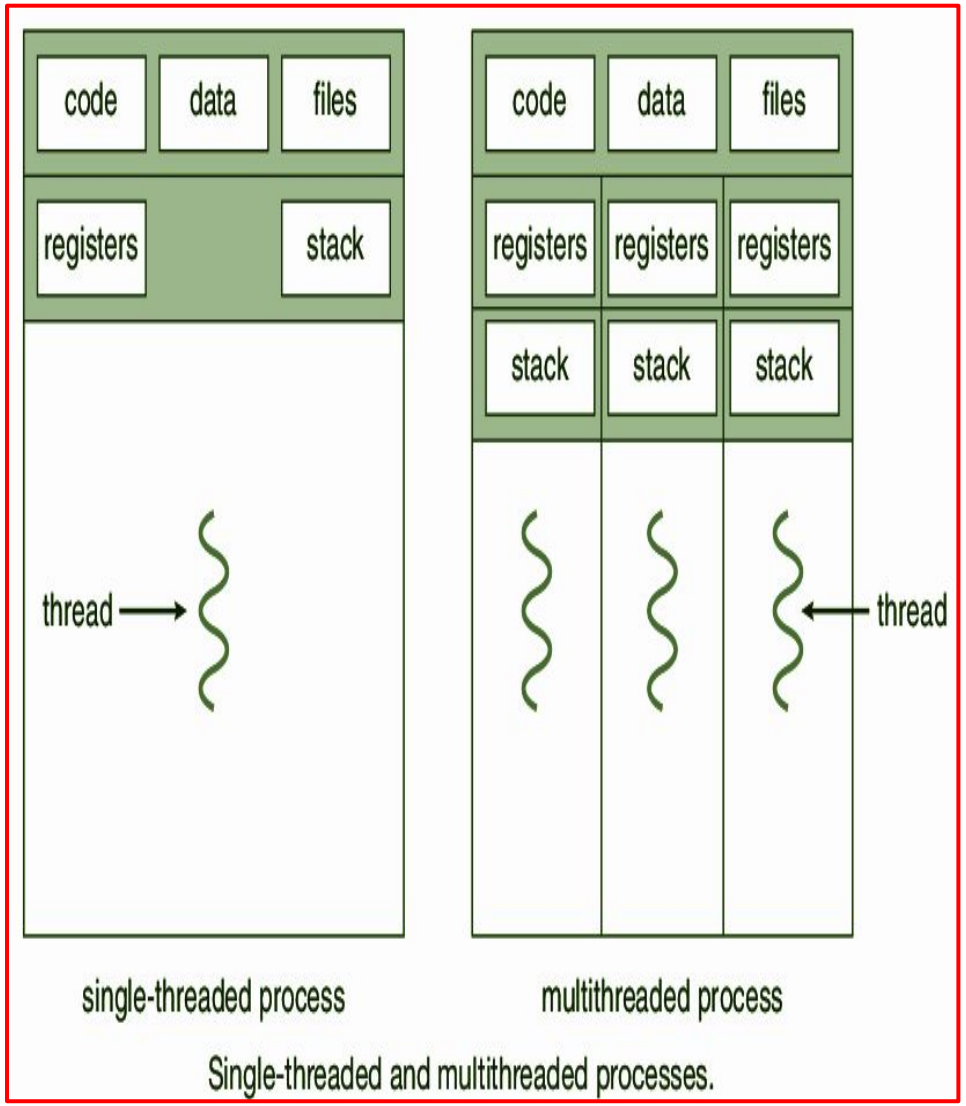
Threads Overview



Single-threaded and multithreaded processes.

Threads Motivation

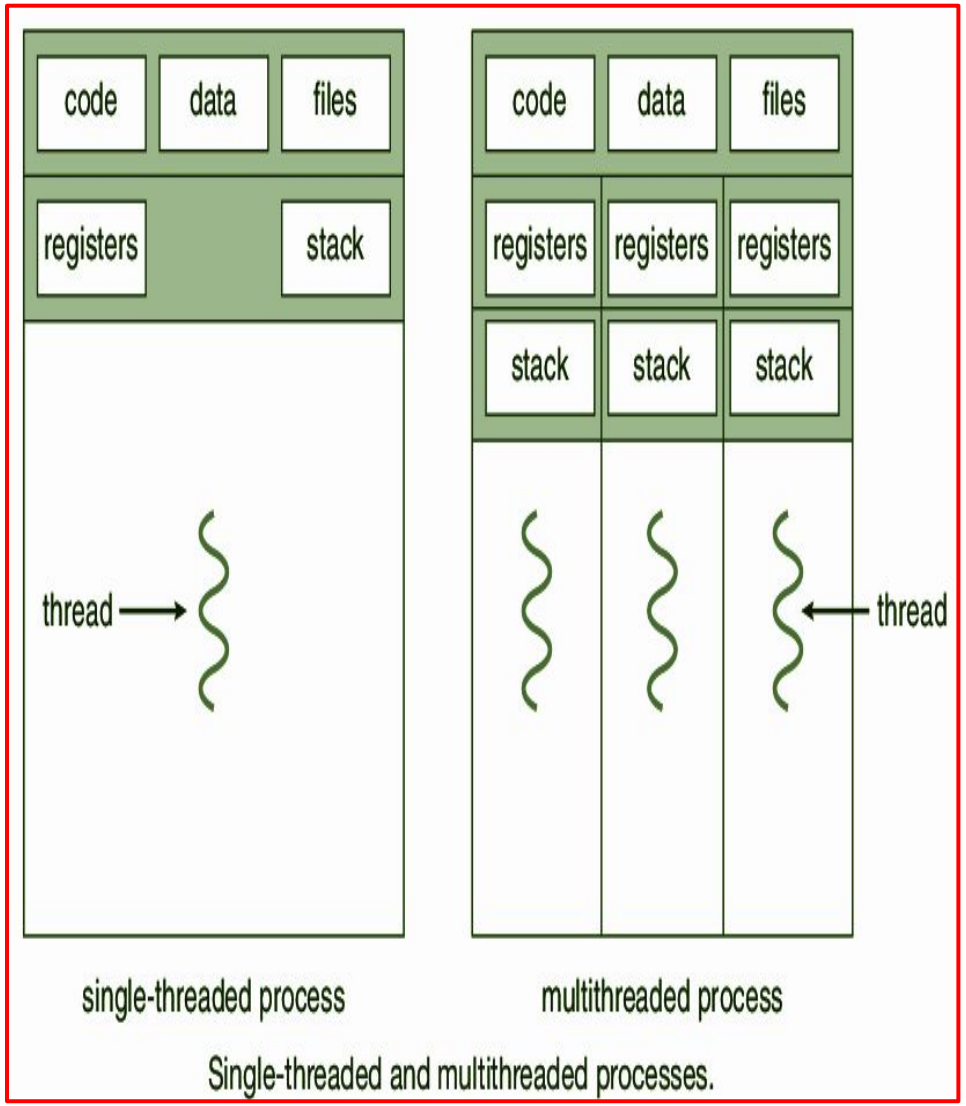
- Most software applications that run on modern computers are **multithreaded**
- An application typically is implemented as a separate process with several threads of control.
- A web browser might have one thread display images or text while another thread retrieves data from the network
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.



Single-threaded and multithreaded processes.

Threads Motivation

- Applications can also be designed to leverage processing capabilities on **multicore** systems.
- Such applications can perform several **CPU -intensive** tasks in **parallel** across the multiple computing **cores**.
- If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.



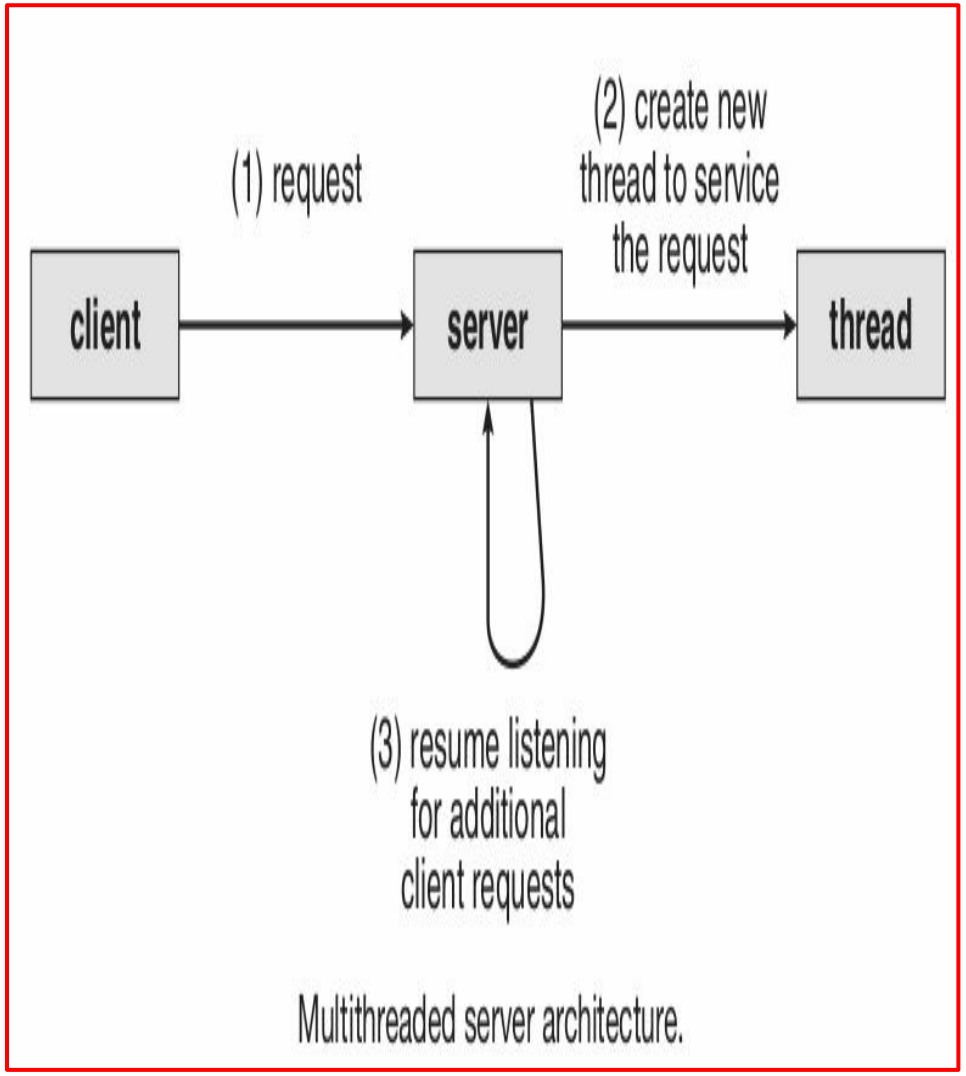
Single-threaded and multithreaded processes.

Threads Motivation

- One solution is to have the server run as a single process that accepts requests.
- When the server receives a request, it creates a separate process to service that request.
- In fact, this process-creation method was in common use before threads became popular.
- Process creation is time consuming and resource intensive

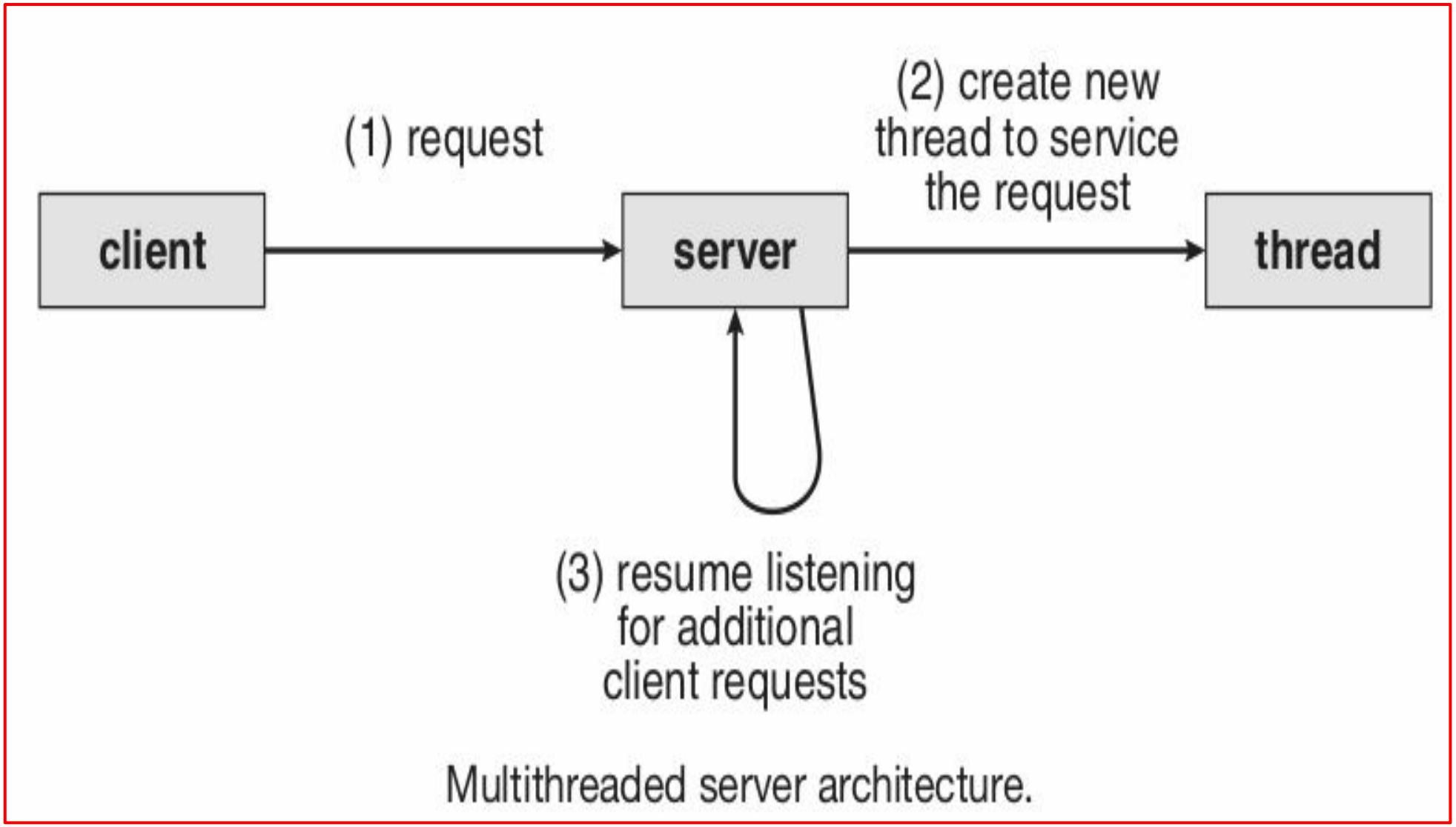
Threads Motivation

- It is generally more efficient to use **one process** that contains **multiple threads**.
- If the web-server process is multithreaded, the server will create a separate thread that listens for client requests.
- When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.



Multithreaded server architecture.

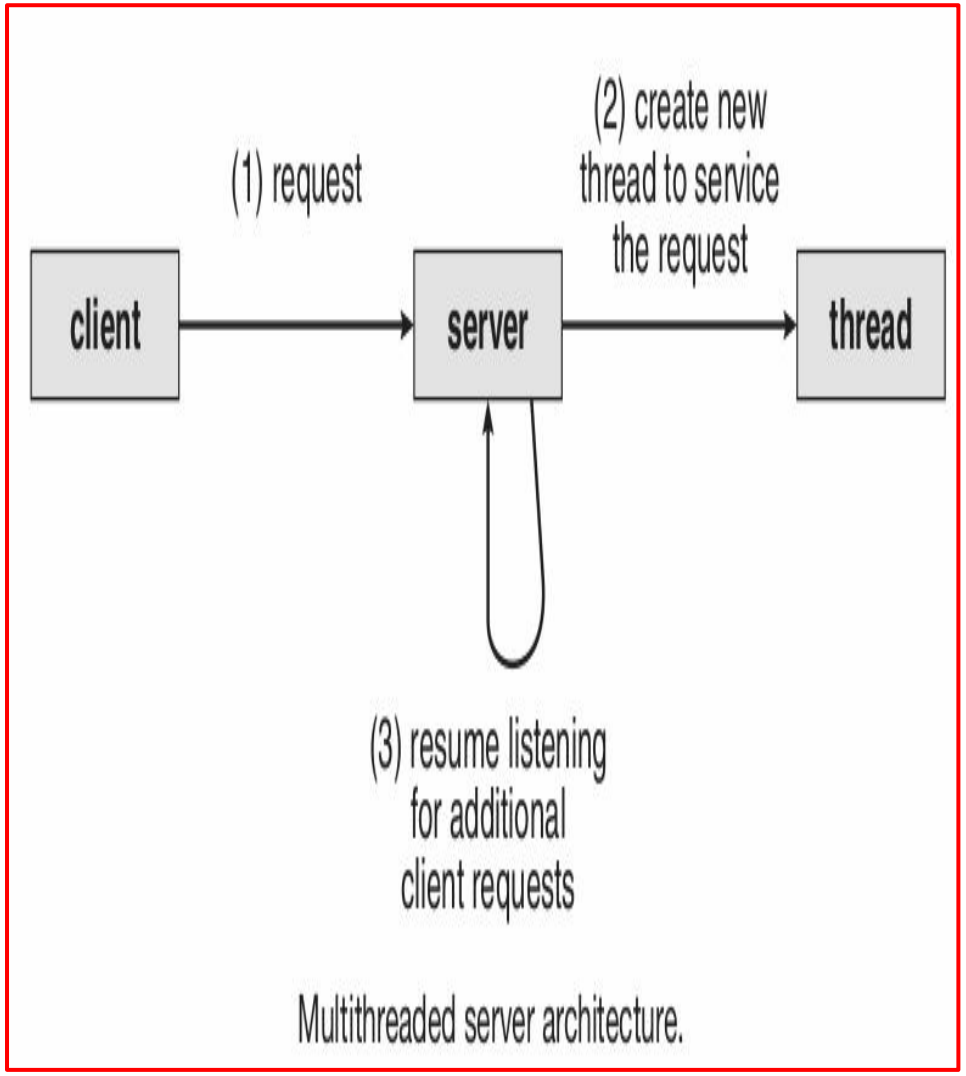
Threads Motivation



Multithreaded server architecture.

Threads Motivation

- Threads also play a vital role in remote procedure call (RPC) systems.
- Typically, RPC servers are multithreaded
- When a server receives a message, it services the message using a separate thread.
- This allows the server to service several concurrent requests.



Multithreaded server architecture.

Threads Benefits

- The benefits of multithreaded programming can be broken down into **four** major categories

1. Responsiveness

- Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- This quality is especially useful in designing user interfaces.
- If the time-consuming operation is performed in a **separate thread**, the application remains **responsive** to the user.

Threads Benefits

- The benefits of multithreaded programming can be broken down into **four** major categories

2. Resource sharing

- Processes can only share resources through techniques such as shared memory and message passing.
- **Threads share** the memory and the resources of the process to which they belong by **default**.
- The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Threads Benefits

- The benefits of multithreaded programming can be broken down into **four** major categories

3. Economy

- Allocating memory and resources for process creation is costly.
- Since threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Empirically** gauging the difference in overhead can be difficult, but in general it is significantly **more time consuming** to create and manage **processes than threads**.

Threads Benefits

- The benefits of multithreaded programming can be broken down into **four** major categories

4. Scalability

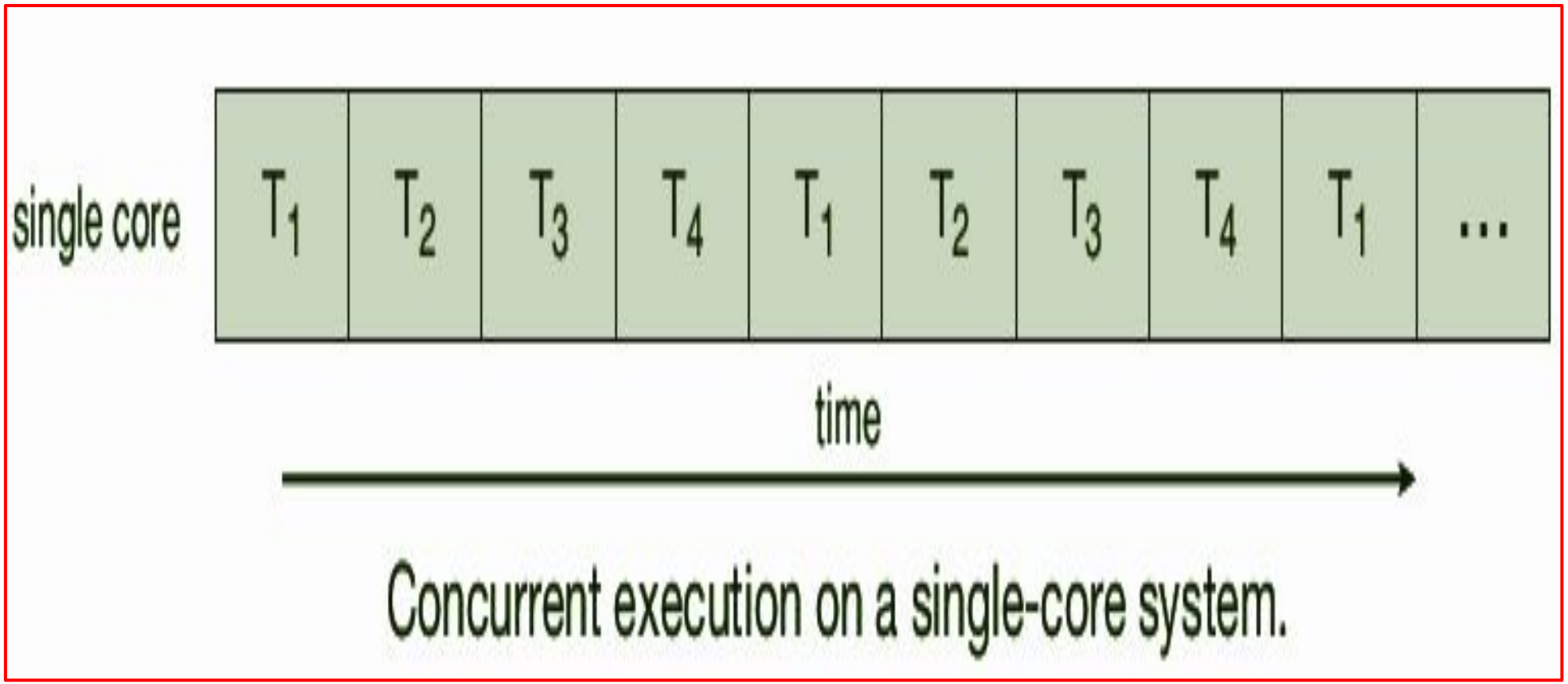
- The **benefits** of **multithreading** can be even greater in a **multiprocessor** architecture, where **threads** may be running in parallel on different processing cores.

- A **single-threaded** process can run on only **one** processor, **regardless** how many are **available**.

Multicore Programming

- A more recent, similar trend in system design is to **place** multiple computing **cores** on a **single chip**.
- Each **core appears** as a separate **processor** to the operating system
- Regardless of the cores appear across CPU chips or within CPU chips, we call these systems **multicore** or **multiprocessor** systems.
- On a system with a **single** computing **core**, concurrency merely means that the execution of the **threads** will be **interleaved** over time , because the **processing** core is capable of **executing** only **one** thread at a time.

Multicore Programming

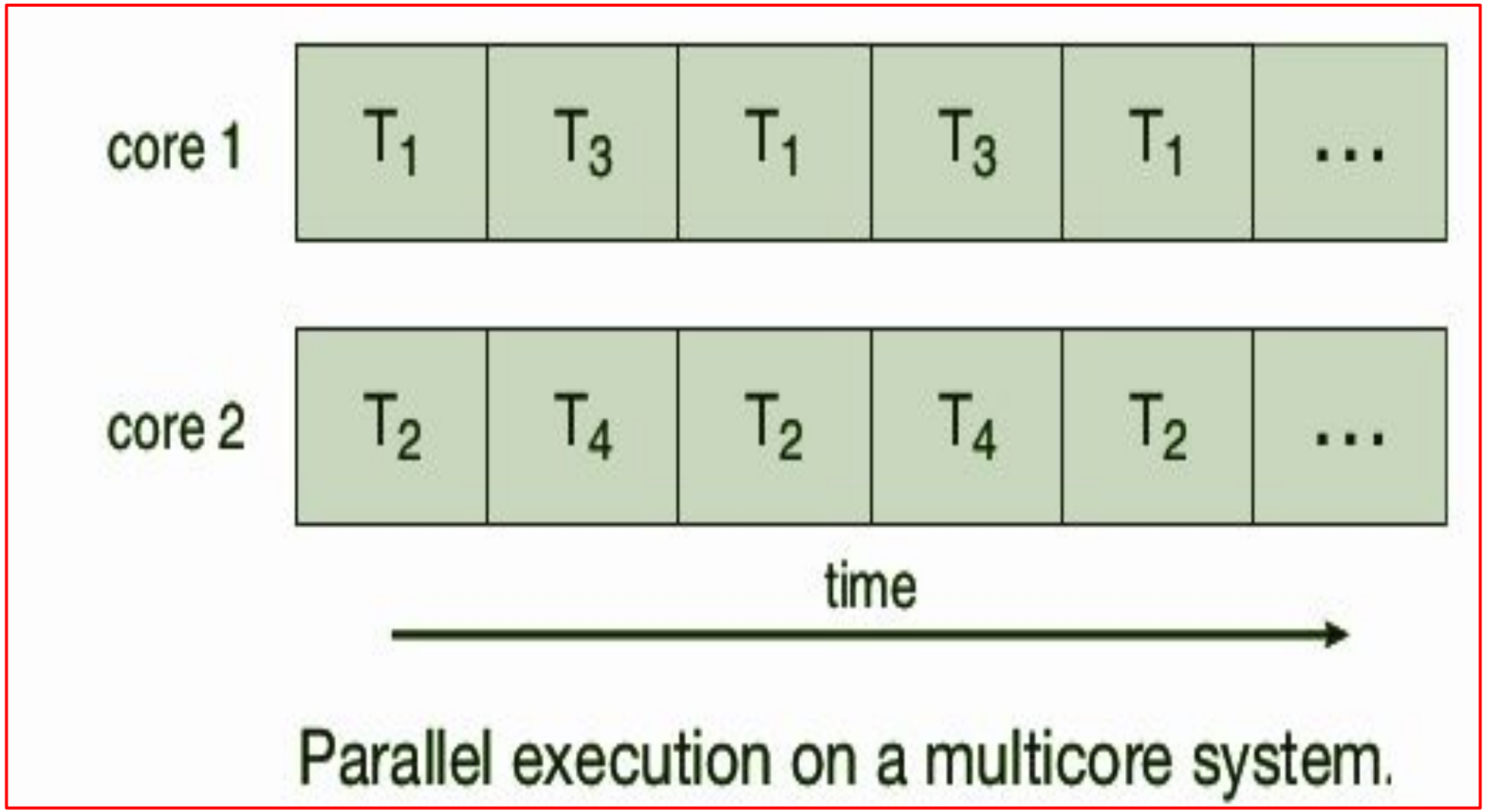


Concurrent execution on a single-core system.

Multicore Programming

- A **system** is **parallel** if it can perform **more** than **one task simultaneously**.
- A **concurrent** system **supports** more than **one** task by allowing all the tasks to make **progress**.
- It is **possible** to have **concurrency** without **parallelism**.
- On a single processor system, CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes in the system, thereby allowing each process to make progress.
- Such **processes** were running **concurrently**, but not in **parallel**

Multicore Programming



Parallel execution on a multicore system.

Multicore Programming: Programming Challenges

- The trend towards multicore systems continues to place pressure on system designers and application programmers to make better use of the multiple computing cores.

- **Designers** of **operating systems must** write scheduling algorithms that use **multiple** processing **cores** to allow the **parallel** execution

- For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded.

Multicore Programming: Programming Challenges

- In general, five areas present challenges in programming for multicore systems

1. **Identifying tasks**. This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.
2. **Balance**. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.
3. **Data splitting**. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
4. **Data dependency**. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
5. **Testing and debugging**. When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Multicore Programming: Types of Parallelism

- In general, there are **two** types of parallelism: **data** parallelism and **task** parallelism.
- **Data parallelism** focuses on distributing **subsets** of the **same data** across **multiple** computing **cores** and performing the **same** operation on each **core**.
- **Task parallelism** involves distributing not data but **tasks** (threads) across **multiple** computing **cores**.
    - Each **thread** is performing a **unique** operation.
    - **Different threads** may be operating on the **same data**, or they may be operating on **different** data.
    - Fundamentally, then, **data** parallelism involves the **distribution** of **data** across **multiple cores** and **task** parallelism on the distribution of **tasks** across **multiple cores**.
- In practice, however, few applications strictly follow either data or task parallelism.
- In most instances, applications use a hybrid of these two strategies.

Multicore Programming: Types of Parallelism - Examples

- Sorting N Integers
  - Sequentially ? N is million numbers
  - Concurrently - Merge Sort - Data Parallelism
- Spellcheck
  - Concurrently - Task Parallelism
- Processing a Table with n columns
  - Concurrently - Task Parallelism
- Addition of Two Numbers - Two numbers that are 5 digit integers
  - Need not require concurrent thinking

Topics Uncovered in this Session

- Threads
  - Overview
  - Motivation
  - Benefits
- Multicore Programming
  - Programming Challenges
  - Types of Parallelism

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com