**What is a Thread?**

A thread is the smallest unit of processing that can be performed in an operating system. Also known as a lightweight process (LWP), a thread exists within a process and shares the process's resources, such as memory and files.

* **Components of a Thread**:
  + **Thread ID**: A unique identifier for the thread.
  + **Program Counter**: Keeps track of the next instruction to be executed.
  + **Register Set**: Contains the CPU registers used by the thread.
  + **Stack**: Used for function calls and local variables.

However, a thread shares with other threads in the same process its **code section**, **data section**, and **other OS resources** (e.g., files and signals).

**Threads vs. Processes**

* **Process**: An independent program in execution with its own memory space.
* **Thread**: A subset of a process, sharing the same memory space and resources.

**Key Difference**: Creating a thread is lighter (less resource-intensive) than creating a process because threads share the same resources within a process.

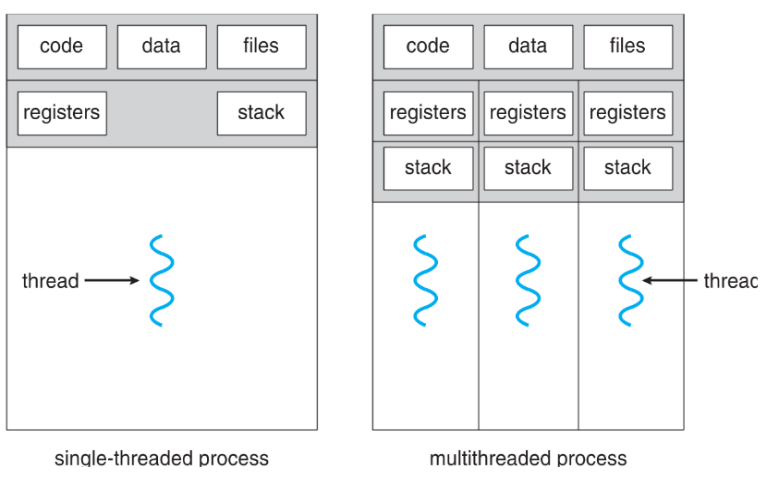
**Why Use Threads?**

1. **Efficiency**: Threads allow multiple tasks to be executed concurrently within the same program. For example, a web browser can load images while fetching data from the network.
2. **Resource Sharing**: Threads within the same process can share data easily, reducing the need for complex communication mechanisms like inter-process communication (IPC).
3. **Responsiveness**: Even if one thread is blocked (e.g., waiting for user input), other threads can continue executing, which is particularly important for user interfaces.

**Examples**

* **Photo Thumbnail Generation**: An application can use separate threads to generate thumbnails for each image in a collection.
* **Web Browser**: One thread displays images or text while another retrieves data from the network.
* **Word Processor**: Separate threads handle displaying graphics, responding to keystrokes, and performing background tasks like spell checking.

**Single vs. Multithreaded Processes**

**Single-Threaded Process**

* Contains one thread of execution.
* All tasks are performed sequentially.

**Multithreaded Process**

* Contains multiple threads, each capable of executing tasks concurrently.
* Shares the same resources (code, data, files) among threads.

**Benefit**: Multithreading allows for better utilization of CPU resources, especially on multi-core systems.

**Why Need Multithreading?**

**Concurrency and Parallelism**

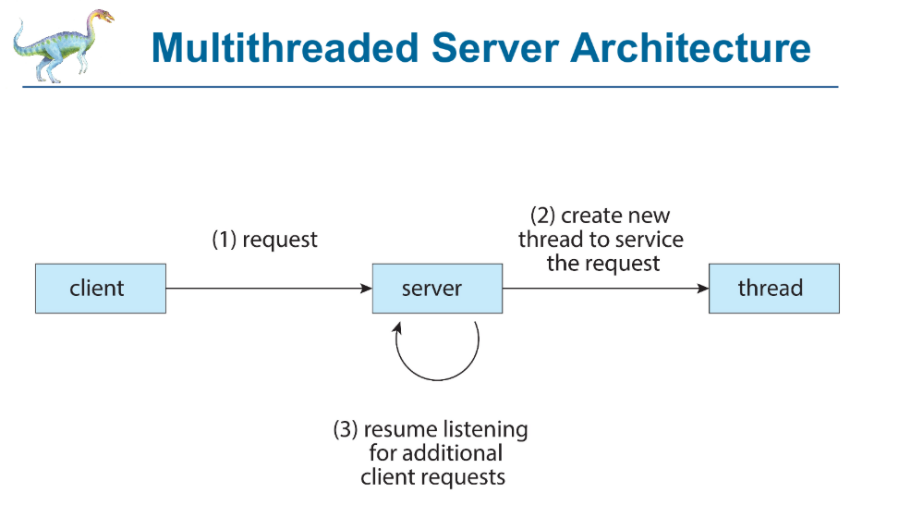
* **Concurrency**: Multiple tasks making progress within overlapping time periods.
* **Parallelism**: Multiple tasks executed simultaneously on multi-core systems.

**Benefits**

* **Simplified Code**: Breaking down tasks into threads can make the code easier to manage.
* **Increased Efficiency**: Tasks can be executed concurrently, reducing idle time.

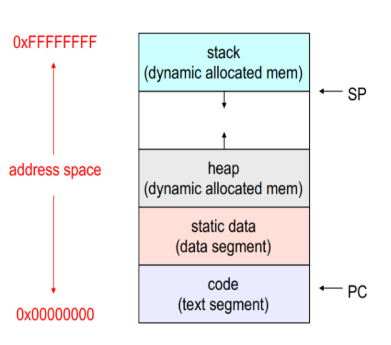
**Example: Web Server**

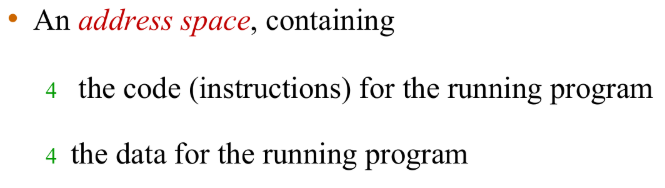
* **Single-Threaded**: Each client request requires creating a new process, which is resource-intensive.
* **Multithreaded**: The server creates a new thread for each client request, allowing for more efficient handling of multiple clients.

**Multithreaded Server Architecture**

1. **Request**: A client sends a request to the server.
2. **Thread Creation**: The server creates a new thread to handle the request.
3. **Resume Listening**: The server continues to listen for additional requests while the thread services the current one.

**What’s In Process?:**

A diagram of a data processing process

AI-generated content may be incorrect.**** **A white background with black text

AI-generated content may be incorrect.**

**A close-up of a sign

AI-generated content may be incorrect.**

**Concurrency vs. Parallelism:**

**Concurrency**

* Achieved on single-core systems where tasks are inserted over time.
* Example: A single-core CPU switches between multiple tasks, giving the illusion of simultaneous execution.

A white and blue square with black letters

AI-generated content may be incorrect.

**Parallelism**

* Achieved on multi-core systems where tasks are executed simultaneously.
* Example: Different cores handle different tasks at the same time.

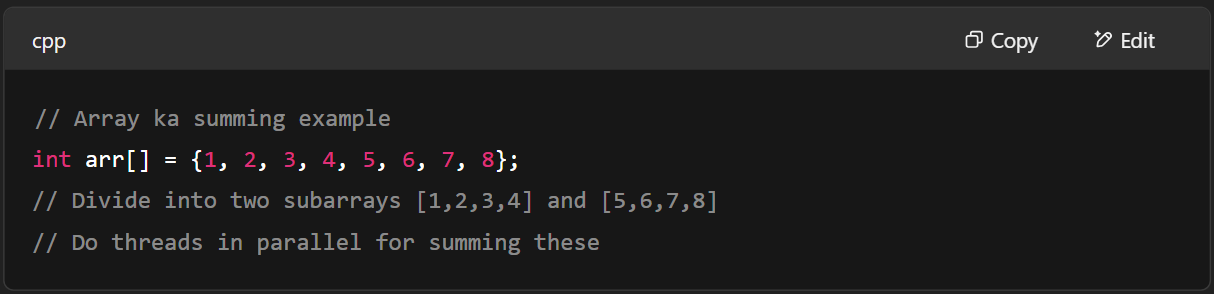
A group of blue squares with black letters

AI-generated content may be incorrect.

**Multicore Programming:**

**Types of Parallelism**

1. **Data Parallelism**:
   * Same operation performed on different data subsets.
   * Example: Summing elements of an array by dividing it into subarrays.



1. **Task Parallelism**:
   * Different operations performed by different threads.
   * Example: One thread performs summation while another performs multiplication.

A black rectangular object with a black stripe

AI-generated content may be incorrect.

1. **Hybrid**:
   * Combination of data and task parallelism.
   * Example: Sorting and summing an array.

A black rectangle with white text

AI-generated content may be incorrect.

**Benefits of Multithreading**

1. **Responsiveness**: Continued execution even if part of the process is blocked.
2. **Resource Sharing**: Easier sharing of resources within a process.
3. **Economy**: Lower overhead compared to process creation.
4. **Scalability**: Utilizes multicore architectures effectively.

**Programming Challenges in Multithreading:**

Multithreading introduces several challenges that programmers must address to ensure efficient and correct execution of multithreaded applications. Here are the key challenges:

**1. Dividing Activities**

* **Challenge**: Efficiently dividing tasks among threads so that they can run concurrently.
* **Example**: Consider a matrix multiplication problem. To divide activities effectively, you need to determine how to split the matrix into parts that can be processed simultaneously by different threads.
  + **Solution**: You might assign each thread to compute a portion of the resulting matrix. For instance, if you have a 4x4 matrix multiplication, you could have four threads, each responsible for computing one row of the result.

**2. Balance**

* **Challenge**: Ensuring that each thread contributes equally to the computation to maximize CPU utilization.
* **Example**: In a sorting algorithm like quicksort, if one thread is responsible for sorting a large partition while another thread sorts a small partition, the thread sorting the smaller partition will finish quickly and remain idle.
  + **Solution**: Implement load balancing techniques to dynamically adjust the workload assigned to each thread, ensuring that all threads are actively contributing to the task.

**3. Data Splitting**

* **Challenge**: Dividing data into sections that can be processed separately by different threads.
* **Example**: In a parallel merge sort, the array to be sorted is split into subarrays. Each thread sorts a subarray independently.
  + **Solution**: Use a divide-and-conquer approach to split the data into roughly equal parts, allowing each thread to work on a separate portion of the data.

**4. Data Dependency**

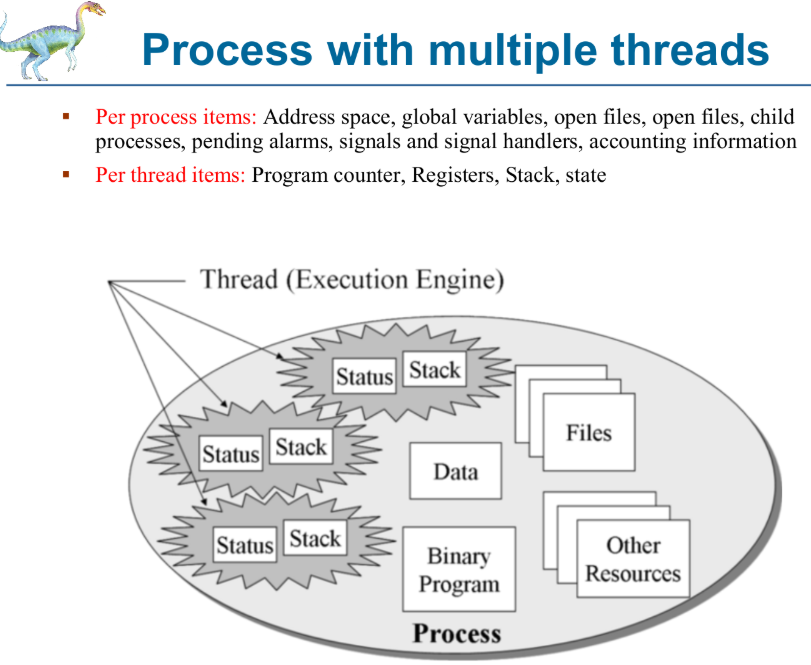
* **Challenge**: Managing shared data access to prevent race conditions and ensure data consistency.
* **Example**: Suppose two threads are incrementing a shared counter variable. If both threads read the counter value simultaneously, they might both increment the same value, leading to incorrect results.
  + **Solution**: Use synchronization mechanisms like mutexes or semaphores to ensure that only one thread can update the shared variable at a time.

**5. Testing and Debugging**

* **Challenge**: The dynamic behavior of multithreaded programs makes testing and debugging complex. Bugs may not appear consistently and can be difficult to reproduce.
* **Example**: A race condition might occur only under specific timing conditions, making it hard to identify and fix.
  + **Solution**: Use tools and techniques such as thread sanitizers, logging, and stress testing to identify potential issues. Implement thorough testing strategies to cover various execution scenarios.

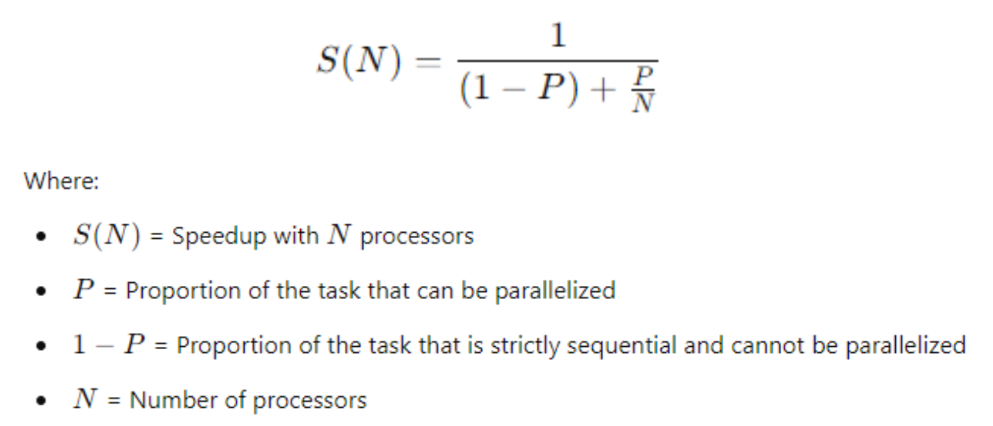
**Threads and Processes in Modern OS:**

* **Process**: Defines the address space and general attributes.
* **Thread**: Defines a sequential execution stream within a process.
* **Scheduling**: Threads are the unit of scheduling; processes are containers for threads.



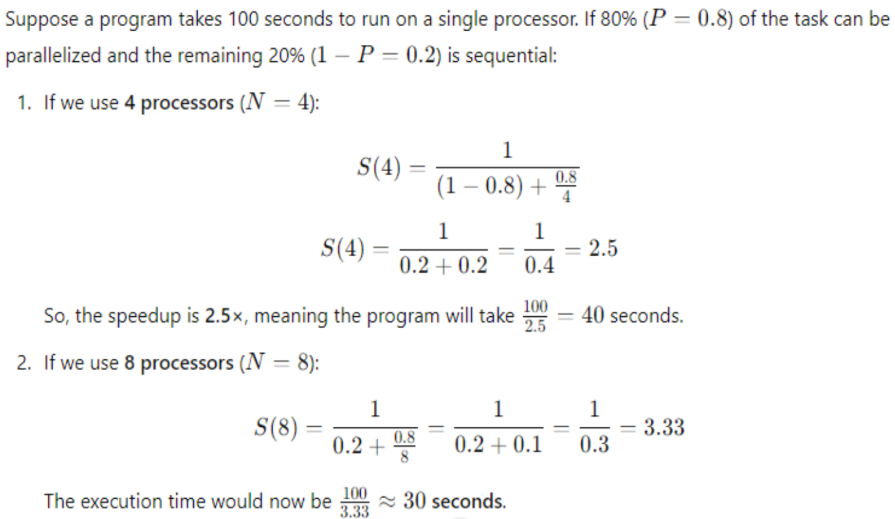
A diagram of a process

AI-generated content may be incorrect.

**Amdahl’s Law:**

**Definition**

Amdahl's Law is a principle used to determine the maximum improvement in performance that can be achieved by adding additional processors to a system. It is particularly relevant in the context of parallel computing, where tasks are divided between serial (non-parallelizable) and parallel (parallelizable) components.

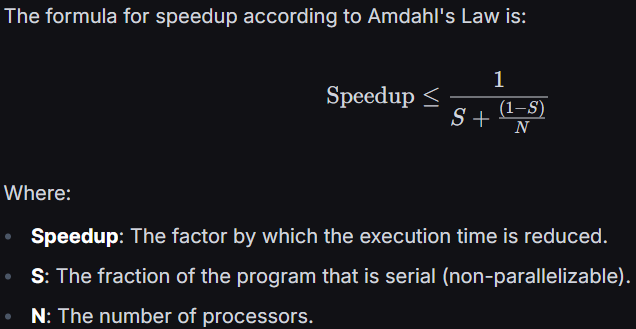


**Implications**

* **Diminishing Returns**: Beyond a certain point, adding more processors has minimal effect on performance.
* **Serial Portion**: The part of the task that cannot be parallelized limits the overall speedup.

**Speed Up Calculation using Amdahl’s Law:**

A black background with white text

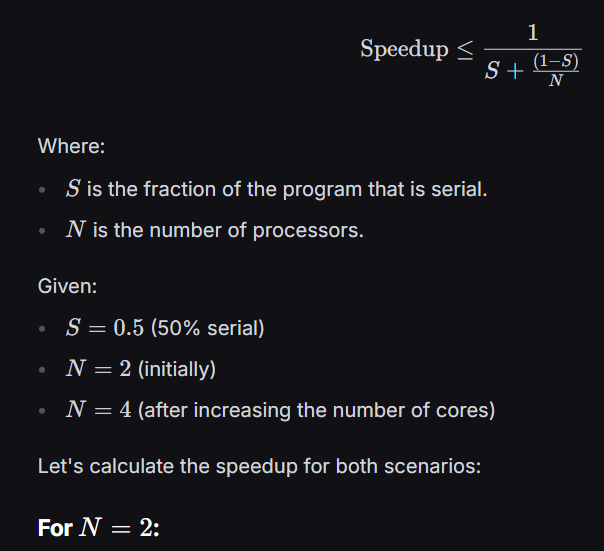
AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Q: Consider a program that is 50% Serial and 50% parallel. The number of cores are increased from N = 2 to**

**N = 4. Calculate the speedup according to Amdahl’s law.**

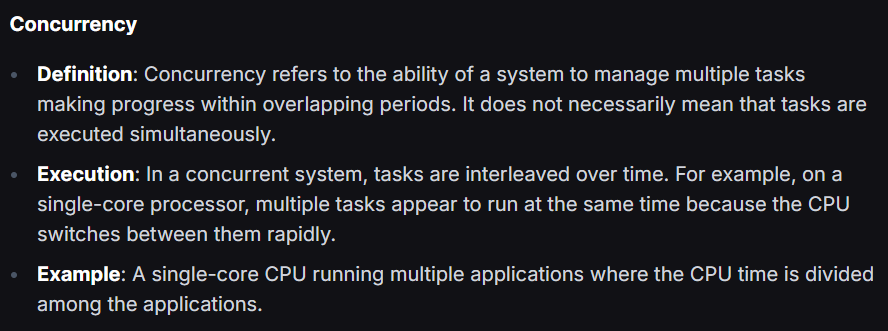
****

**A screenshot of a computer

AI-generated content may be incorrect.**

**Q2: how would you differentiate between concurrent and parallelism**

**A black background with white text

AI-generated content may be incorrect.**

**User Threads and Kernel Threads**

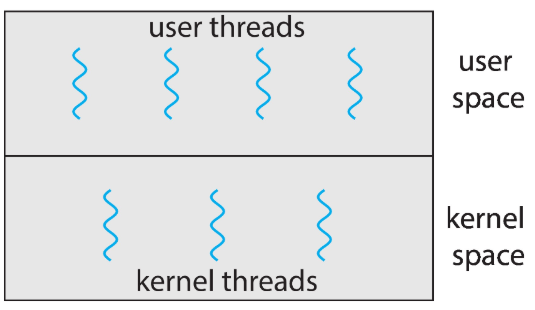
**User Threads**

1. Managed by user-level libraries (e.g., **Pthreads, Windows threads, Java threads).**
2. **Advantages**: Cheap context switch costs, user-programmable scheduling.
3. **Disadvantages**: Handling blocking system calls can be challenging.

**Kernel Threads**

* Managed directly by the **OS** (e.g., **Windows, Linux**).
* **Advantages**: Full knowledge of all threads, better for applications that frequently block.
* A close-up of a few informational messages

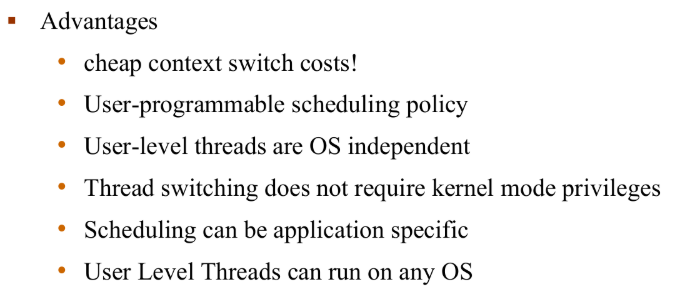
  AI-generated content may be incorrect.**Disadvantages**: Higher overhead and complexity.



**Implementation of Thread in User Space:**

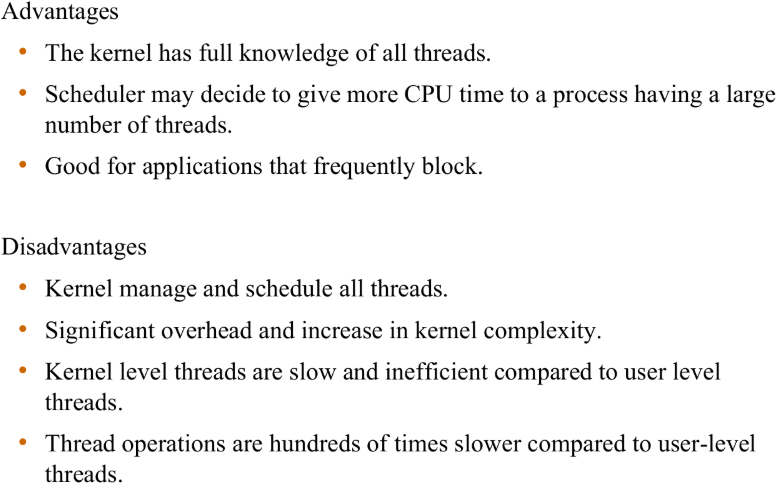
1. **Management by Thread Library:**
   * User-level threads are managed by a thread library that runs in user space.
   * The kernel is unaware of these threads and only sees the process as a whole.
   * Examples of thread libraries include POSIX Pthreads and GNU Portable Threads.
2. **Thread Control Block (TCB):**
   * The thread library maintains a Thread Control Block for each thread, which includes information such as the thread ID, program counter, register set, stack pointer, and scheduling parameters.
   * The TCB is used to manage the state and execution of each thread.
3. **Context Switching:**
   * Context switching between user-level threads is handled entirely by the thread library, without kernel intervention.
   * This makes context switching faster and more efficient compared to kernel-level threads, as it doesn't require a switch to kernel mode.
4. **Scheduling:**
   * The thread library implements its own scheduling algorithm to decide which thread to run at any given time.

**A diagram of a process

AI-generated content may be incorrect.**

**Implementation of Thread in Kernal Space:**

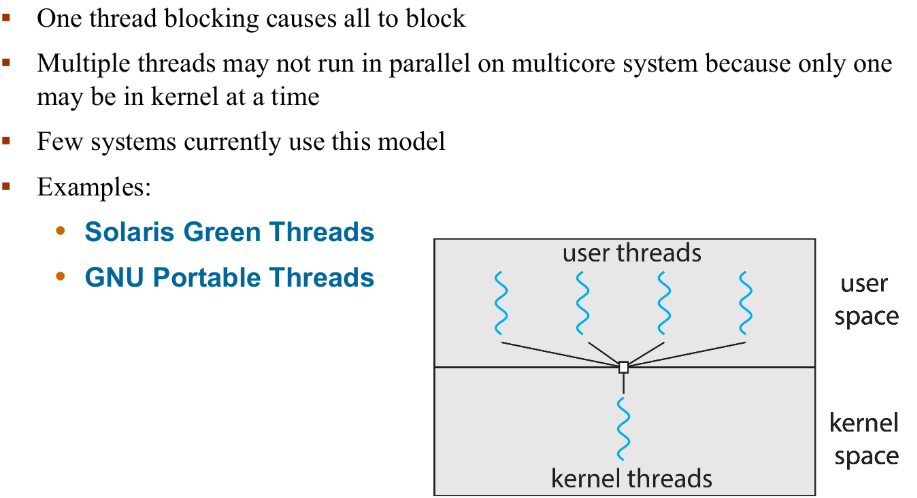
1. **Kernel Management**:
   * Kernel-level threads are created and managed by the operating system's kernel.
   * The kernel maintains a Thread Control Block (TCB) for each thread, which includes information such as the thread ID, program counter, register set, stack pointer, and scheduling parameters.
2. **Visibility to the OS**:
   * The operating system is aware of each kernel-level thread and can schedule them independently.
   * This allows for better utilization of CPU resources and more efficient handling of blocking system calls.
3. **Context Switching**:
   * Context switching between kernel-level threads involves the kernel, which can be more resource-intensive compared to user-level threads.
   * The kernel handles the switching, ensuring that each thread gets its fair share of CPU time.
4. **Scheduling**:
   * The kernel's scheduler decides which thread to run based on its scheduling policies.
   * This can lead to more efficient use of CPU resources, especially in multi-core systems.



**Multithreading Models**

**1. Many-to-One (M:1)**

* Description: In this model, multiple user-level threads are mapped onto a single kernel-level thread.

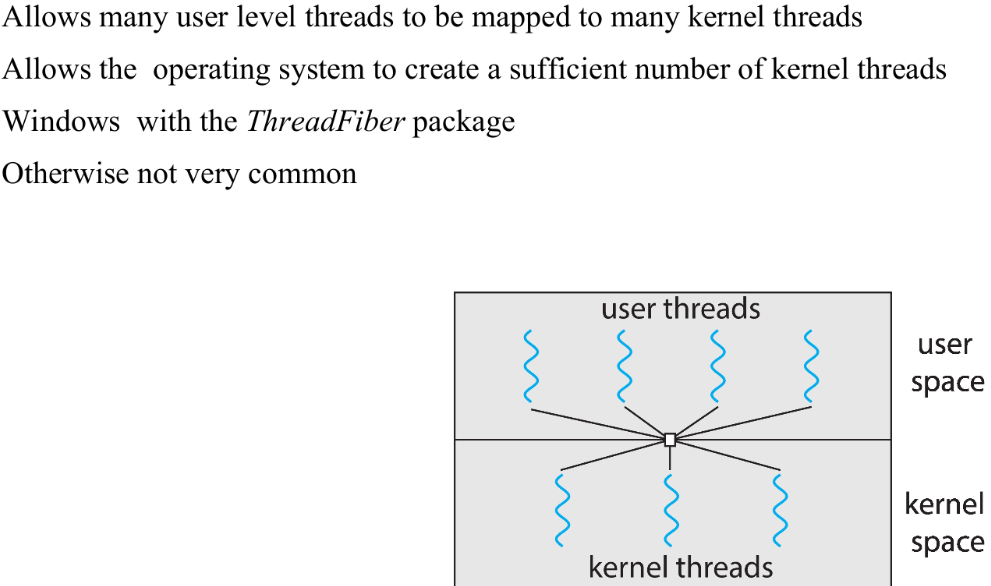


**2. One-to-One (1:1)**

**A diagram of a machine

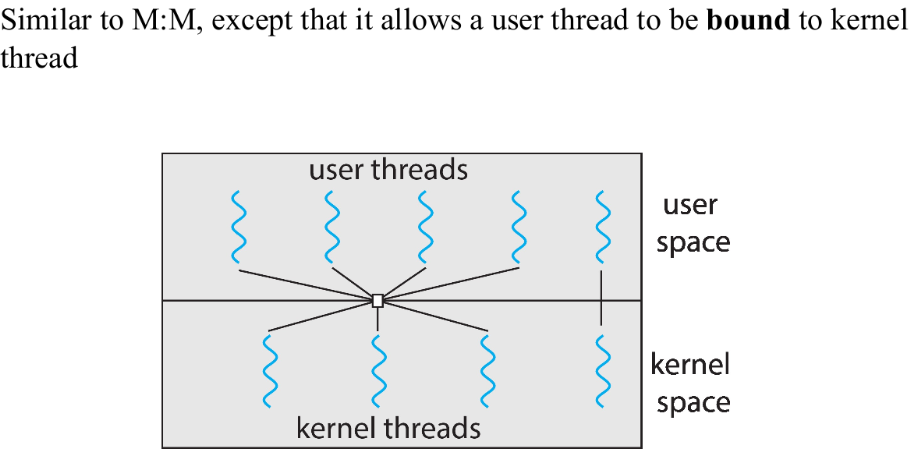
AI-generated content may be incorrect.**

**3. Many-to-Many (M:N)**

****

**4. Two Level Model:**

* Description: A variation of the M:N model where user threads can be bound to specific kernel threads.

****

**Thread Cancellation:**

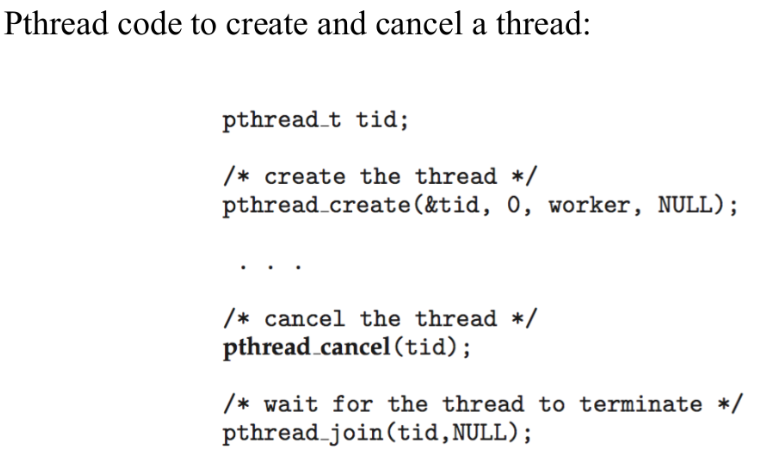
Thread cancellation is the process of terminating a thread before it has completed its execution. This can be useful in scenarios where a thread is performing a long-running operation that is no longer needed, or when an error occurs and the thread needs to be stopped to prevent further issues. However, thread cancellation must be handled carefully to avoid resource leaks and ensure that the program remains in a consistent state.

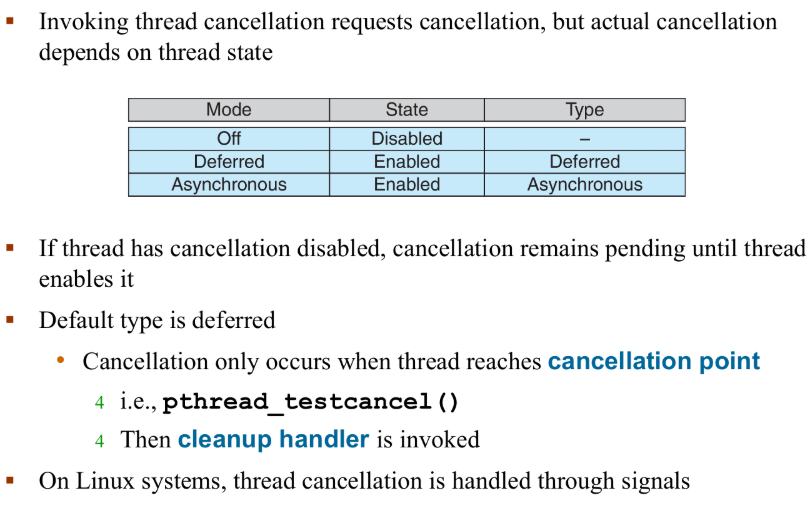
**a) Asynchronous Cancellation**:

* **Description**: The thread is terminated immediately, regardless of what it is doing

**b) Deferred Cancellation**:

* **Description**: The thread is marked for cancellation but is allowed to run until it reaches a safe point where it can be terminated.





**Implicit Threading:**

**Implicit Threading** is a method of parallel programming where the **compiler or runtime system handles the creation and management of threads** instead of the programmer explicitly creating and managing them.

***Methods of Implicit Threading:***

**1. Thread Pools**

A **pool of threads** is created in advance and reused for multiple tasks. You don’t create threads manually; you submit tasks, and the system picks a free thread to run it.

**Example:**

* **Java's ExecutorService**
* **.NET Task Parallel Library**
* **OpenMP (C/C++)**

**Why it's useful:**

* Prevents overhead of creating threads repeatedly
* Better resource management

**2. Fork–Join Framework**

Used to split a task into subtasks (fork), execute them in parallel, and then join results.

Fork/join parallelism is a style of parallel programming useful for exploiting the parallelism inherent in divide and conquering algorithms on shared memory multiprocessors.

The idea is quite simple: a larger task can be divided into smaller tasks whose solutions can then be combined

**Example:**

* Java’s ForkJoinPool
* Intel TBB (Threading Building Blocks)

**A diagram of a task

AI-generated content may be incorrect.A white screen with black text

AI-generated content may be incorrect.Fork Join Parallelism 👇**

A diagram of a task

AI-generated content may be incorrect.

**3. OpenMP**

**OpenMP** stands for **Open Multi-Processing**. It is an **API (Application Programming Interface)** that supports **multi-platform shared memory multiprocessing programming** in C, C++, and Fortran.

**Key Features:**

* Easy to write parallel programs using simple compiler directives.
* Mainly used for **loop-level parallelism** (e.g., for loops).
* Works on **shared memory systems** (i.e., all threads share the same address space).
* Uses **threads** to execute different parts of code simultaneously.

**How OpenMP Works**

OpenMP uses:

* **Compiler Directives** (e.g., #pragma omp parallel)
* **Runtime Library Routines**
* **Environment Variables**

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.