# Concurrency and Parallelism Programming

## Concurrency Is Not Parallelism

|  |  |
| --- | --- |
| **Concurrency** | **Parallelism** |
| The ability to **run multiple tasks (or threads) simultaneously**, independently but not necessarily in parallel. | The ability to **run multiple tasks (or threads) in parallel, truly simultaneously.** |
| Being achieved through the interleaving operation of processes **on a single CPU or core**.  Depending on the scheduling mechanism by the OS. | Being achieved by **through multiple CPUs or cores**.  Without relying on interleaved execution or time-sharing of the OS. |
| Focusing on increasing the amount of work finished at a time. | Focusing on improving the throughput and computational speed of the system, especially when working with a large data set. |

**Concurrency Use Cases:**

* **User Interface Responsiveness**: Concurrency is often used in GUIs to keep the user interface responsive while performing time-consuming operations in the background. For example, a web browser fetch and render web pages while allowing the user to interact with the browser interface.
* **Networking**: Concurrency is vital in networking applications where multiple clients or servers need to be handled simultaneously. For example, a web server can handle multiple incoming requests concurrently, allowing it to serve multiple clients concurrently without blocking or delaying other requests.
* **Asynchronous Programming**: Concurrency is frequently used to handle I/O operations, like reading from or writing to files, databases, or network sockets. By executing operations concurrently, the program can continue making progress while waiting for I/O to complete.

**Parallelism Use Cases:**

* **Data Processing**: Parallelism is widely used in computationally intensive tasks that can be divided into smaller subtasks that can be processed independently. Examples include video processing, scientific simulations, and data analysis.
* **Machine Learning and AI**: Parallelism is crucial in training and inference of machine learning and AI models. Many algorithms used in these domains can be parallelized, allowing the processing of large datasets or complex computations to be distributed across multiple processors or GPU cores.
* **Mathematical Computations**: Parallelism can be beneficial in mathematical computations, like matrix operations, numerical simulations, or solving complex equations. These computations often involve performing similar operations on large sets of data, making them suitable for parallel execution.

## Process, Thread and Task

### Process

A process is an **instance of a running program**, with its own memory space and resources. Processes don’t share memory or resources unless explicitly communicated. They provide a high level of isolation, as failures in one process do not directly affect others.

Switching between processes are done at the kernel level.

### Thread

A thread is a **unit of execution within a process**.

Multiple threads within a process can execute concurrently on different CPU cores or be scheduled to run on a single core using time-sharing techniques.

Threads share the same memory space and resources, and can communicate with each other more efficiently than separate processes. But each thread has its own stack, state, program counter and registers, which provide the execution context for the instructions it runs. So they can maintain local variables, manage stack frames, and handle interrupts or exceptions within their own execution context.

Switching between thread are done at either the kernel level or the user level.

### Task

A task is a **piece of work** or a specific operation that can be executed concurrently or asynchronously. It represents a logical concept and does not necessarily correspond to a specific implementation detail.

Compared to threads, tasks don't have their own execution context, but rely on the execution context of the thread executing them. In addition, while threads are scheduled by the OS's scheduler, tasks are scheduled by a task scheduler or executor.

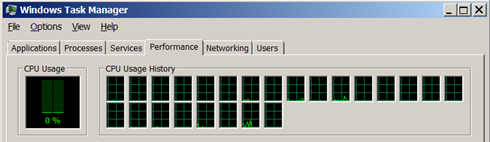
A thread can execute multiple tasks over its lifetime, one after another or concurrently, depending on the scheduling mechanism and the specific design of the application.

## Multi-Processing

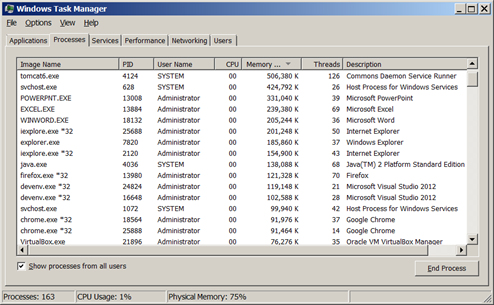
### What Is It?

Modern CPUs provide multiple processing cores (independent processing units) that allow multiple apps to run concurrently.

For example, Windows Task Manager shows 24 CPU cores on a workstation:

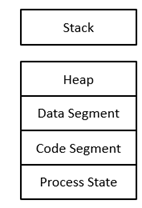


From the same computer, we can see many applications running at the same time:



**Each program runs as a *'process'* and is assigned a Process ID (PID)**.

Below is a basic diagram showing how memory is divided up for each process:



* Firstly, a process needs to keep track of the state of the CPU processor it’s running under (**Process State**). This includes any program counters, register values, stack pointer (see below) and file/resource handles.
* Secondly, the process will include the **Code Segment** which is the actual programming code (machine language) it’s running.
* Thirdly, the **Data Segment** which holds global and static variables.
* Next, the **Heap** is where any dynamically allocated memory is placed.
* Finally, the **Stack** is a memory area that holds local variables and function parameters.

As long as one process is operating on its own set of data (such as single-processed, single-threaded programs), this model works out very well.

As said, modern OSs like Windows and Linux allow us to run multiple programs at the same time. However, the limitation here is that **such programs themselves can only work on one thing at a time**. So, if your program is processing a data file, it basically has to start at the beginning and go through the entire file to the end. While it’s processing the data, the single-threaded program cannot do anything else.

On the other hand, **if we want our program to take advantage of multiple CPUs (processors), we need a way of allowing our program to carry out some work in parallel, which would require the program to share the data**.

### Example – Image Processing

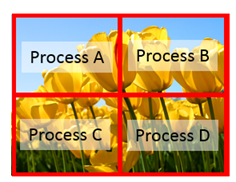
One of the classic examples of this type of parallel work can be found with image processing. For example, applying a filter to an image requires processing (doing some math on) each pixel. With one process, it will take some time to visit each pixel and process it. For example:



In the meantime, the program cannot do anything else until it is all done with running the image filter.

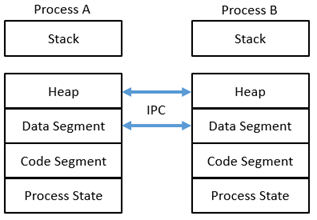
However, images can be segmented into multiple sections and each can be processed independently at the same time. Some ways to describe this type of data processing are parallel processing or multi-threaded processing of data.

For example, we might have 4 processes – each work on a different portion of the image. The processes would need to communicate with one another and pass the image data around to work on it.



### How Does Data Communication Between Processes Work?

In this section, I will introduce two ways in which we can process data in parallel.

One way to work on data in parallel is to run multiple processes and then use an Inter-Process Communications (**IPC**) method to share the data. Some IPC methods include:

* Sockets
* Pipes
* Remote Procedure Calls
* Shared File
* Shared Memory
* Signals
* Message Passing Interfaces (MPI)
* ...

However, **creating a completely new process each time we want to share the data will have a lot of overhead** since we need to make a completely new Process State, Code Segment, Data Segment, Stack and Heap. Not to mention the complexity of transmitting data between processes using IPCs.

In a nutshell, using multiple processes to work on the same set of data is going to be inefficient. So, we’ll learn a lot to apply the technique effectively.

## Multi-Threading

**Great short video series about programming with threads in C:**

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFZxuqyJ9JqVYmuFZHr7CFM>

**Great short video series about programming with processes in C:**

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFNNfpY6fHjVzAwtgRYjhPw>

### What Is It?

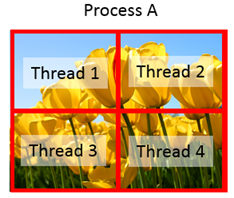
Each program typically consists of at least one thread called the *main* thread, which is created when the program starts executing. If that program adds one or more threads which run simultaneously with the main thread or with other threads, it would be described as a multi-threaded program.

### How Does It Work?

One process with two threads will share the Process State, Code Segment, Data Segment and Heap. However, each thread needs its own Stack space to keep track of the function and method calls. Also, each thread will need some Thread-Specific State information held aside.

Because there is less work to be done to create a new thread (as compared to process), we can say typically **using multiple threads will be less overhead and more efficient processing than using multiple processes.**

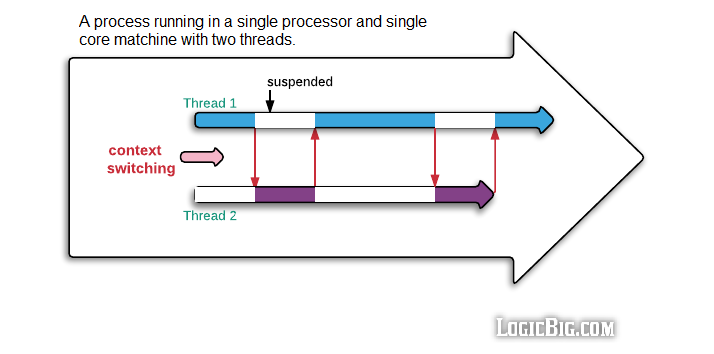
Going back to our image-processing example, our program might launch 4 threads to process different regions of the image. Because the image data is shared on the Heap, the threads do not need to pass anything between them using IPC.



# Multi-Threading

## Context Switching

*Context Switching* refers to the process of saving the current state of a thread (its register values, program counter, stack pointer, etc.) and restoring the saved state of another thread for execution. In a multithreading environment, context switching allows the OS to efficiently share the CPU among multiple threads.



## Race Conditions

In multi-threading, a race condition occurs **when two or more threads access shared data concurrently**. It can lead to unpredictable and incorrect results, as the threads may interfere with each other's execution and manipulate shared data in an unintended way. This is one of the most common issues we will come up against when running multiple threads. The corrupted data in this case is called a *data race*.

If all of our threads are only reading the data, then we generally do not have much to worry about. However, if one or more threads are writing some data (such as modifying the value of a variable) while other threads may be reading/writing the data, then we face a potential data corruption problem. We can extend the same logic to any resource including memory, files and any other system devices.

One common example can be found in the console where std::cout is directed. Having 2+ threads printing a text repetively to the console, you’ve probably seen that the output lines are mixed up sometimes. That’s due to multiple threads printing at the same time.

## Synchronization Mechanisms

### Mutex

A mutex (Mutual Exclusion) **acts as a barrier, which takes on one of two states: *Locked* and *Unlocked***.

The working principle is as follows:

* It works upon locking mechanism. Once the lock has been acquired and owned by a thread, any other thread attempting to access the shared resource will be blocking – held in a **wait state until the lock is freed up**.
* It uses two atomic operations for process synchronization:
* ***Lock***: When a worker thread reaches a critical section where it’ll work with some shared resource.
  + ***Unlock***: Once the thread is done working on the critical section.

You might not know!

In some docs, people specific mutex as a *binary semaphore*.

The C++11 Standard Library includes std::mutex in the <mutex> header. But note that it’s **designed for thread synchronization within a single process**. It does not provide inter-process synchronization. For example, check [this session](#_Example:_Mutex).

### Semaphore

A semaphore allows multiple threads to access a shared resource simultaneously, **up to a certain limit**. The working principle is as follows:

* It has an non-negative integer value (typically initialized to 0) that can be increased or decreased.
* It works upon heavily signaling mechanism, in this a thread can be signaled by another thread. It uses 2 atomic operations for process synchronization:
  + ***Wait (P)***: Decreases the semaphore count. If the count is 0, it blocks the calling thread. If the count is greater than 0, simply decreases the count but still allows the thread to proceed.
  + ***Signal (V)***: Increases the semaphore count. It signifies that the shared resource is now available for other threads to acquire. So if there are any threads waiting on the semaphore, one of them will be unblocked and allowed to proceed.

You might not know!

In some docs, people specific such a semaphore as a *counting semaphore*.

C++11 Standard Library does not provide a built-in semaphore implementation. However, C++20 Standard Library includes a std::counting\_semaphore class, which provides a counting semaphore implementation.

In C++11, you can use the <semaphore.h> header providing semaphores in C. For example, check [this session](#_Example:_Samaphore).

### Condition Variable

A condition variable provides a way for a thread to **suspend its execution until another thread signals that a particular condition has been met**. This allows threads to synchronize their activities and avoid busy waiting, where a thread repeatedly checks for a condition in a tight loop.

In C++11 Standard Library, std::condition\_variable is supported and defined in the <condition\_variable> header, with following operations:

* ***Wait***: A thread can call the wait() while holding a lock (typically a mutex). This operation will internally:
  + Releases the associated lock (mutex). This is equivalent to calling unlock() on the mutex.
  + Puts the thread in a blocked state, waiting for a notification.

That wait is ended until another thread notifies the condition variable. The thread will remain asleep until it is awakened by a call to notify\_one() or notify\_all().

* ***Notify***: A thread can call the notify\_one() to wake up the other thread waiting on the condition variable. Alternatively, notify\_all() can be used to wake up all threads. When a thread is awakened, it re-acquires the lock it held before calling wait() and continues its execution.

Condition variables are **typically used in conjunction with mutexes** to coordinate the execution of multiple threads.

But note that like std::mutex, std::condition\_variable it’s **designed for thread synchronization within a single process**. It does not provide inter-process synchronization.

For example, check [this session](#_Example:_Thread_Pool).

### Read-Write Lock

A read-write lock, also known as a *shared-exclusive lock*, provides synchronization between threads that need to either read or write a shared resource. **Multiple threads can acquire the lock for reading simultaneously, but only one thread can acquire it for writing**, ensuring exclusive access.

This kind of locks is suitable when there are many read operations and fewer write operations on a shared resource, such as in read-heavy scenarios or data structures that allow concurrent reading.

### Spinlock

A spinlock allows multiple threads to access a shared resource in a mutually exclusive manner. **Instead of blocking the thread, it repeatedly checks if it can acquire the lock, spinning in a loop until it becomes available**.

The working principle is as follows:

* The first process acquires the lock on the shared resource and continues on its way, doing what it needed to with the resource.
* Any other processes that try to acquire the lock then "spin in place" waiting on the lock to be released by the first process.

For example, check [this session](#_Example:_Spinlock).

### Read-Write Spinlock

A read-write spinlock is **similar to a read-write lock, but uses spinning instead of blocking**. It allows concurrent read access but exclusive write access to a shared resource.

### Barrier

A barrier is used to synchronize a **group of threads, ensuring that each has to wait for each other at a particular point in the code**. Once all threads have reached the barrier, they are released simultaneously, allowing them to proceed.

The working principle is as follows:

* The barrier works by using a mutex (like std::mutex), a condition variable (like std::condition\_variable) and an integer counter.
* When a thread reaches the barrier and calls the wait() method, it acquires the lock of the associated mutex. The other threads will wait until the lock is released.
* Each thread then decrements the count of threads that have reached the barrier.
* If count = 0, the current thread is the last one to arrive at the barrier. Then, the thread calls notify\_all() on the condition variable, which wakes up all the waiting threads.
* If count != 0, there are still threads that have not arrived at the barrier. Then, the thread releases the lock and waits on the condition variable, using a lambda function which checks if the count is 0, indicating that all threads have arrived. If the count is not 0, the thread remains blocked until the condition variable is notified and the lambda function evaluates to true.
* Once all threads have been unblocked and are executing again, the barrier is reset to its initial state by calling the reset() method. This allows the barrier to be used again for subsequent synchronization.

For example, check [this session](#_Example:_Barrier).

### Future/Promise

### Which One To Choose?

**Each has different pros and cons:**

|  |  |  |
| --- | --- | --- |
|  | **Mutex** | **Semaphore** |
| Advantages | * **No race condition occurs**, as only one thread is in the critical section at a time. So, data remains consistent. * **Simple** to use. | * **Cannot prevent race condition**, as multiple threads can access the critical section at the same time. * **Machine-independent**, so should be run over microkernel. * More **versatile than mutexes** but can be more **complex** to use correctly. |
| Disadvantages | * The waiting mechanism can cause **wastage of the CPU cycle**, or even cause degrade where hundreds of threads share the same resource. * Can potentially lead to **deadlocks** if not used properly. * Can cause **priority inversion** issues. | * Can potentially lead to **deadlocks** if not used properly. * Can cause **priority inversion** issues. * Prone to error, as any thread can signal or wait on a semaphore. |

**Each has different use cases:**

**Mutex:**

* **Shared Data Protection**: Avoids data corruption before accessing or modifying the data (in global variables, data structures, buffers, etc.).
* **File Access Protection**: Prevents multiple threads from simultaneously reading from or writing to the same file.
* **Resource Pooling Protection**: Prevent multiple threads from simultaneously access a shared pool, such as database connections, network sockets, or thread pools.

**Semaphore:**

* **Resource Limits**: Manages traffic flow in systems that involve concurrent access to shared resources, such as limiting the number of concurrent connections to a server, restricting the number of active threads, or managing access to a pool of system resources like memory or file descriptors.

Example: Let's say we have a limited pool of database connections, and multiple threads need to acquire and release connections to perform DB operations. In this case, a semaphore can be used to control access to the pool of connections. The semaphore count represents the number of available connections, and each thread needs to acquire a connection from the semaphore before performing its operations. If all connections are currently in use, the semaphore will block the threads until a connection becomes available.

* **Synchronization Between Threads**: Synchronizes activities between threads, allowing them to coordinate their execution. For example, ensure that a set of threads waits until a certain condition is met before proceeding.
* **Multithreaded Task Scheduling**: Used in scenarios where multiple threads need to perform tasks concurrently but with a defined level of parallelism.

**Condition Variable:**

* **Event Notification**: Threads can wait for desired events, state changes or signals to occur using condition variables. For example, a thread waits until a file is available for reading or until a network connection is established.
* **Synchronization with Timeouts**: Threads can wait on a condition variable for a specified period, and if the condition is not met within that time, they can take alternative actions or handle the timeout scenario.

Helpful tip:

A barrier is like the starting line at a horse race. You wait for all the horses to get in their gates, and then they all start off (i.e., resume down the track) at the same time. If you leave the starting gate on the track, the horses will all line up again after they eventually complete one lap in staggered order, and then once they all arrive, they all start again at the same time.

A condition variable is like the kitchen service bell at diner. The order-taker puts the order on the counter, rings the bell, and runs off. The cook wakes up if they were asleep (or wanders over eventually if they were busy) and picks up the order. There might be multiple order-takers and/or cooks using the bell. How efficient the kitchen is depends on how well the order-takers and cooks work together, and even how many bells are used. The bell itself can only ring.

[c++ - Comparing synchronization primitives (future, barrier, conditional var) - which one fits the best? - Software Engineering Stack Exchange](https://softwareengineering.stackexchange.com/questions/447384/comparing-synchronization-primitives-future-barrier-conditional-var-which)

[Thread Synchronization with Condition Variables or Tasks – MC++ BLOG (modernescpp.com)](https://www.modernescpp.com/index.php/thread-synchronization-with-condition-variables-or-tasks/)

[C++ Core Guidelines: Be Aware of the Traps of Condition Variables – MC++ BLOG (modernescpp.com)](https://www.modernescpp.com/index.php/c-core-guidelines-be-aware-of-the-traps-of-condition-variables/)

## Potential Issues

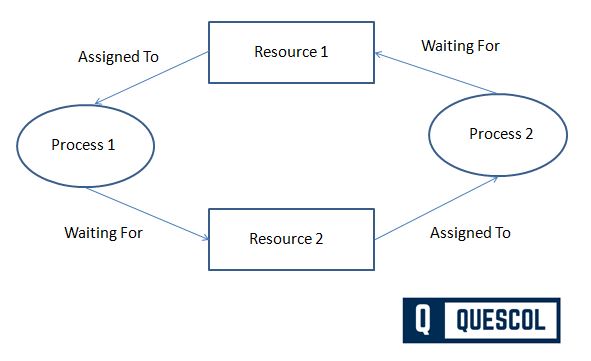
### Deadlock

Deadlock is a situation that occurs in concurrent systems when **two or more threads/processes are unable to proceed because each is waiting for a resource held by another**. In other words, deadlock is a state where multiple threads or processes are stuck, and none can make progress.

Deadlocks is problematic because they can lead to a system freeze or a complete breakdown of functionality

Causes of deadlocks:

* **Mutual Exclusion**: A mutex lock the resource but doesn’t release it for some reasons (forget to release, or error).
* **No Preemption**: Resources cannot be forcibly taken away from a thread/process. Only the holding one can voluntarily release the resource.
* **Circular Wait**: There is a circular chain of two or more threads/processes, where each holds a resource that is requested by another in the chain.



Example:

#include <iostream>

#include <thread>

#include <mutex>

std::mutex mutex1;

std::mutex mutex2;

void threadA() {

    std::unique\_lock<std::mutex> lock1(mutex1);

    std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

    std::unique\_lock<std::mutex> lock2(mutex2);

    // Critical section for thread A

    std::cout << "Thread A acquired both mutexes and is inside the critical section.\n";

    // Release the mutexes

    lock2.unlock();

    lock1.unlock();

}

void threadB() {

    std::unique\_lock<std::mutex> lock2(mutex2);

    std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

    std::unique\_lock<std::mutex> lock1(mutex1);

    // Critical section for thread B

    std::cout << "Thread B acquired both mutexes and is inside the critical section.\n";

    // Release the mutexes

    lock1.unlock();

    lock2.unlock();

}

int main() {

    std::thread t1(threadA);

    std::thread t2(threadB);

    t1.join();

    t2.join();

    return 0;

}

Output: Nothing

Explain: threadA and threadB are trying to acquire two mutexes, mutex1 and mutex2, respectively. However, threadA acquires mutex1 first and then waits for mutex2, while threadB acquires mutex2 first and then waits for mutex1. As a result, both threads are stuck waiting for a mutex that will never be released, leading to a deadlock.

### Livelock

Livelock is similar to deadlock about **preventing threads/processes from being proceed**. But in livelock, they’re not blocked or waiting for resources like in deadlock. Intead, they’re constantly **responding to each other's actions but are unable to achieve their intended goals**.

Example:

#include <iostream>

#include <thread>

#include <atomic>

std::atomic<bool> flag1{false};

std::atomic<bool> flag2{false};

void threadA() {

    while (!flag2) {

        // Wait until thread B sets flag2

    }

    // Perform some work

    std::cout << "Thread A: Executing work.\n";

    flag1 = true;  // Set flag1

    // Continue executing work

    std::cout << "Thread A: Continuing work.\n";

}

void threadB() {

    while (!flag1) {

        // Wait until thread A sets flag1

    }

    // Perform some work

    std::cout << "Thread B: Executing work.\n";

    flag2 = true;  // Set flag2

    // Continue executing work

    std::cout << "Thread B: Continuing work.\n";

}

int main() {

    std::thread t1(threadA);

    std::thread t2(threadB);

    t1.join();

    t2.join();

    return 0;

}

Output: Nothing

Explain: threadA and threadB are waiting for each other to set their flags. However, due to the timing and synchronization, they end up constantly checking the flags without making progress. As a result, both threads are continuously executing, leading to a livelock situation.

### Starvation

Starvation is a situation where a **thread is unable to complete its task due to insufficient access to resources or scheduling issues**. It occurs when one or more threads are consistently deprived of the resources they need to execute, leading to a **delay** or inability to access the required resources. But it doesn’t cause threads/processes permanently blocked like deadlock or livelock.

Causes of starvation:

* **Resource Allocation**: If a scheduling algorithm or resource allocation strategy is unfair, it may prioritize certain threads over others, causing some threads to be starved of resources.
* **Priority Inversion**: If the low-priority thread continues to hold the resource for an extended period that a high-priority thread requires, the high-priority thread may be starved and unable to proceed.
* **Synchronization Issues**: Improper use of synchronization mechanisms, such as locks or semaphores, can result in threads being blocked for an extended period, leading to starvation.

For example:

#include <iostream>

#include <thread>

#include <mutex>

std::mutex mutex;

void lowPriorityThread()

{

    while (true) {

        std::unique\_lock<std::mutex> lock(mutex);

        // Perform low-priority work

        std::cout << "Low-priority thread executing.\n";

        // Release the lock

        lock.unlock();

        // Sleep for a short period to yield the CPU

        std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

    }

}

void highPriorityThread()

{

    while (true) {

        std::unique\_lock<std::mutex> lock(mutex);

        // Perform high-priority work

        std::cout << "High-priority thread executing.\n";

        // Release the lock

        lock.unlock();

        // Sleep for a short period to yield the CPU

        std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

    }

}

int main()

{

    std::thread t1(lowPriorityThread);

    std::thread t2(highPriorityThread);

    t1.join();

    t2.join();

    return 0;

}

Output:

Low-priority thread executing.

High-priority thread executing.

Low-priority thread executing.

High-priority thread executing.

Low-priority thread executing.

High-priority thread executing.

Low-priority thread executing.

High-priority thread executing.

Low-priority thread executing.

…

We have two threads: lowPriorityThread and highPriorityThread. Both acquire and release a mutex before performing their work. However, the lowPriorityThread does not yield the CPU frequently, meaning it holds the lock for longer periods. Thus, the highPriorityThread is starved and unable to execute its work consistently, even though it has a higher priority.

### Priority Inversion

Priority inversion occurs when a **low-priority thread holds a mutex that a high-priority thread needs to acquire**. As a result, the high-priority thread is blocked, and the overall system priority is effectively lowered. This can lead to undesired behavior and performance degradation.

## Thread Pool

A thread pool is a technique to manage a **pool of worker threads** that can be utilized to perform concurrent tasks. Instead of creating threads on demand for each task, a thread pool maintains **a set of pre-created threads** that are ready to execute tasks as soon as they become available.

Advantages:

* **Improved performance**: By reusing threads, it **avoids the overhead for creating and destroying threads for each task**, resulting in improved performance and reduced latency.
* **Resource management**: It allows you to **control the number of threads created** and limits the concurrent execution of tasks, preventing resource exhaustion and providing better control over resource utilization.
* **Load balancing**: It can employ **scheduling algorithms** that distribute tasks evenly among the available threads, ensuring efficient utilization of system resources and preventing thread starvation.

For example, check [this session](#_Example:_Thread_Pool).

## Libraries

Most modern OS support threads at the OS level. Within C and C++, threading libraries are various:

* C++11 Standard Library Threads (C++11, Unix and Windows)
* POSIX Threads, also called pthreads (C and C++, Unix only)
* Win32 Library Threads (C and C++, Windows only)
* Boost Threads Library (C++, Unix and Windows)

### C++11 Standard Thread Library

#### Example: Worker Function Without Parameters

#include <iostream>

#include <thread>

#include <chrono>

void worker\_functionA(void)

{

int loop = 0;

// Loop 10 times and print to the screen from 1 to 9

while (loop < 10) {

// Sleep for 1.33 seconds

std::this\_thread::sleep\_for(std::chrono::milliseconds(1333));

std::cout << "Thread A Reporting: " << loop << std::endl;

loop++;

}

}

void worker\_functionB(void)

{

int loop = 0;

// Loop 10 times and print to the screen from 1 to 9

while (loop < 10) {

// Sleep for 2.22 seconds

std::this\_thread::sleep\_for(std::chrono::milliseconds(2222));

std::cout << "Thread B Reporting: " << loop << std::endl;

loop++;

}

}

int main()

{

char result;

// Launch two new threads. They will start executing immediately

std::thread worker\_threadA(worker\_functionA);

std::thread worker\_threadB(worker\_functionB);

// Pause the main thread

std::cout << "Press a key to finish" << std::endl;

std::cin >> result;

// Join up the two worker threads back to the main thread

worker\_threadA.join();

worker\_threadB.join();

return 1;

}

Output:

Press a key to finish

Thread A Reporting: 0

Thread B Reporting: 0

Thread A Reporting: 1

Thread A Reporting: 2

Thread B Reporting: 1

Thread A Reporting: 3

Thread A Reporting: 4

Thread B Reporting: 2

Thread A Reporting: 5

Thread B Reporting: 3

Thread A Reporting: 6

Thread A Reporting: 7

Thread B Reporting: 4

Thread A Reporting: 8

Thread A Reporting: 9

Thread B Reporting: 5

#### Example: Worker Function With Parameters

One common feature we may want to take advantage of is to generalize the worker functions especially in cases where they are doing very **similar types of processing just on different subsets of the data or with slightly different operating parameters**. To accomplish this, we can pass parameters to the thread when it is invoked.

In the C++11 standard, this is actually quite easy to do. When we call the constructor for std::thread we can easily pass along the necessary parameters.

For example:

#include <iostream>

#include <thread>

#include <chrono>

void worker\_function(int thread\_number, int iterations, long delay)

{

    int loop = 0;

    // Loop some 'iterations' number of times

    while (loop < iterations)

    {

        // Sleep for some time

        std::this\_thread::sleep\_for(std::chrono::milliseconds(delay));

        std::cout << "Thread " << thread\_number << " Reporting: "

                  << loop << " with delay " << delay << std::endl;

        loop++;

    }

}

int main()

{

    char result;

    // Launch two new threads. They will start executing immediately

    // worker\_function is a generic worker that will take in two parameters:

    // number of iterations and sleep time

    std::thread worker\_thread1(worker\_function, 1, 10, 1555);

    std::thread worker\_thread2(worker\_function, 2, 10, 2222);

    // Pause the main thread

    std::cout << "Press a key to finish" << std::endl;

    std::cin >> result;

    // Join up the two worker threads to the main thread

    worker\_thread1.join();

    worker\_thread2.join();

    // Return success

    return 1;

}

Output:

Press a key to finish

Thread 1 Reporting: 0 with delay 1555

Thread 2 Reporting: 0 with delay 2222

Thread 1 Reporting: 1 with delay 1555

Thread 2 Reporting: 1 with delay 2222

Thread 1 Reporting: 2 with delay 1555

Thread 1 Reporting: 3 with delay 1555

Thread 2 Reporting: 2 with delay 2222

Thread 1 Reporting: 4 with delay 1555

Thread 2 Reporting: 3 with delay 2222

Thread 1 Reporting: 5 with delay 1555

Thread 1 Reporting: 6 with delay 1555

Thread 2 Reporting: 4 with delay 2222

Thread 1 Reporting: 7 with delay 1555

Thread 2 Reporting: 5 with delay 2222

Thread 1 Reporting: 8 with delay 1555

Thread 2 Reporting: 6 with delay 2222

Thread 1 Reporting: 9 with delay 1555

Thread 2 Reporting: 7 with delay 2222

Thread 2 Reporting: 8 with delay 2222

Thread 2 Reporting: 9 with delay 2222

#### Example: Mutex

##### Single Mutex

#include <iostream>

#include <thread>

#include <chrono>

#include <mutex>

// Mutex to protect the console output (std::cout)

std::mutex mtConsole;

void worker\_function(int thread\_number, int iterations, long delay)

{

int loop = 0;

// Loop some 'iterations' number of times

while (loop < iterations)

{

// Sleep for some time

std::this\_thread::sleep\_for(std::chrono::milliseconds(delay));

// Get the lock on the console

mtConsole.lock();

std::cout << "Thread " << thread\_number << " Reporting: "

<< loop << " with delay " << delay << std::endl;

// Remove the lock on the console

mtConsole.unlock();

loop++;

}

}

int main()

{

char result;

// Launch two new threads. They will start executing immediately

std::thread worker\_thread1(worker\_function, 1, 10, 2000);

std::thread worker\_thread2(worker\_function, 2, 10, 2000);

// Pause the main thread

std::cout << "Press a key to finish" << std::endl;

std::cin >> result;

// Join up the two worker threads to the main thread

worker\_thread1.join();

worker\_thread2.join();

// Return success

return 1;

}

Output:

Press a key to finish

Thread 2 Reporting: 0 with delay 1000

Thread 1 Reporting: 0 with delay 1000

Thread 2 Reporting: 1 with delay 1000

Thread 1 Reporting: 1 with delay 1000

Thread 2 Reporting: 2 with delay 1000

Thread 1 Reporting: 2 with delay 1000

Thread 2 Reporting: 3 with delay 1000

Thread 1 Reporting: 3 with delay 1000

Thread 2 Reporting: 4 with delay 1000

Thread 1 Reporting: 4 with delay 1000

Thread 2 Reporting: 5 with delay 1000

Thread 1 Reporting: 5 with delay 1000

Thread 2 Reporting: 6 with delay 1000

Thread 1 Reporting: 6 with delay 1000

Thread 2 Reporting: 7 with delay 1000

Thread 1 Reporting: 7 with delay 1000

Thread 2 Reporting: 8 with delay 1000

Thread 1 Reporting: 8 with delay 1000

Thread 2 Reporting: 9 with delay 1000

Thread 1 Reporting: 9 with delay 1000

**Tips:**

* In the above example, we used std::mutex::lock() and std::mutex::unlock(). This is okay, but **can we forget to unlock the mutex manually**?

Yes! A better approach is to use the std::lock\_guard instead. It locks the mutex, and once it goes out of scope, it automatically unlocks the mutex.

We can modify the above example like that:

while (loop < iterations)

{

…

// Get the lock on the console.

// The lock will be automatically unlocked when out of the loop.

std::lock\_guard<std::mutex> lock(mtConsole);

std::cout << "Thread " << thread\_number << " Reporting: "

<< loop << " with delay " << delay << std::endl;

…

}

##### Mutex Pool

#include <iostream>

#include <thread>

#include <mutex>

#include <unordered\_map>

class MyClass {

public:

    void sharedFunction1() {

        std::lock\_guard<std::mutex> lock(getMutex(sharedFunction1));

        // Code for sharedFunction1

    }

    void sharedFunction2() {

        std::lock\_guard<std::mutex> lock(getMutex(sharedFunction2));

        // Code for sharedFunction2

    }

private:

    std::mutex& getMutex(const std::function<void()>& function) {

        std::lock\_guard<std::mutex> lock(mutexMapMutex);

        return mutexMap[function];

    }

    std::unordered\_map<std::function<void()>, std::mutex> mutexMap;

    std::mutex mutexMapMutex;

};

int main() {

    MyClass myObject;

    // Create multiple threads accessing the shared functions

    std::thread thread1(&MyClass::sharedFunction1, &myObject);

    std::thread thread2(&MyClass::sharedFunction1, &myObject);

    std::thread thread3(&MyClass::sharedFunction2, &myObject);

    std::thread thread4(&MyClass::sharedFunction2, &myObject);

    // Wait for all threads to finish

    thread1.join();

    thread2.join();

    thread3.join();

    thread4.join();

    return 0;

}

#### Example: Samaphore

**Without a C++ wrapper:**

#include <iostream>

#include <thread>

#include <semaphore.h>

const int MAX\_CONNECTIONS = 5;

// Semaphore to control access to database connections

sem\_t semaphore;

void threadFunction(int threadId) {

    std::cout << "Thread " << threadId << " is acquiring a DB connection." << std::endl;

    // Wait until a database connection is available

    sem\_wait(&semaphore);

    std::cout << "Thread " << threadId << " has acquired the DB connection." << std::endl;

    // Simulate some work on the connection

    std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

    // Release the database connection

    sem\_post(&semaphore);

    std::cout << "Thread " << threadId << " has released the DB connection." << std::endl;

}

int main() {

    const int numThreads = 4;

    std::thread threads[numThreads];

    // Initialize the semaphore with the maximum number of connections

    sem\_init(&semaphore, 0, MAX\_CONNECTIONS);

    for (int i = 0; i < numThreads; ++i) {

        threads[i] = std::thread(threadFunction, i);

    }

    for (int i = 0; i < numThreads; ++i) {

        threads[i].join();

    }

    // Destroy the semaphore

    sem\_destroy(&semaphore);

    return 0;

}

Output:

Thread 0 is acquiring a DB connection.

Thread 0 has acquired the DB connection.

Thread 2 is acquiring a DB connection.

Thread 2 has acquired the DB connection.

Thread 3 is acquiring a DB connection.

Thread 3 has acquired the DB connection.

Thread 1 is acquiring a DB connection.

Thread 1 has acquired the DB connection.

Thread 0 has released the DB connection.

Thread 2 has released the DB connection.

Thread 3 has released the DB connection.

Thread 1 has released the DB connection.

**With a C++ wrapper:**

#include <iostream>

#include <thread>

#include <semaphore.h>

const int MAX\_CONNECTIONS = 5;

class Semaphore {

public:

    Semaphore(int initValue) {

        sem\_init(&m\_sem, 0, initValue);

    }

    ~Semaphore() {

        sem\_destroy(&m\_sem);

    }

    void acquire() {

        sem\_wait(&m\_sem);

    }

    void release() {

        sem\_post(&m\_sem);

    }

private:

    sem\_t m\_sem;

};

void threadFunction(int threadId, Semaphore& semaphore) {

    std::cout << "Thread " << threadId << " is acquiring a DB connection." << std::endl;

    // Wait until a database connection is available

    semaphore.acquire();

    std::cout << "Thread " << threadId << " has acquired the DB connection." << std::endl;

    // Simulate some work on the connection

    std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

    // Release the database connection

    semaphore.release();

    std::cout << "Thread " << threadId << " has released the DB connection." << std::endl;

}

int main() {

    const int numThreads = 4;

    std::thread threads[numThreads];

    // Semaphore to control access to database connections

    // Initialize semaphore with an initial count of 5

    Semaphore semaphore(MAX\_CONNECTIONS);

    for (int i = 0; i < numThreads; ++i) {

        threads[i] = std::thread(threadFunction, i, std::ref(semaphore));

    }

    for (int i = 0; i < numThreads; ++i) {

        threads[i].join();

    }

    return 0;

}

Output:

Thread 0 is acquiring a DB connection.

Thread 0 has acquired the DB connection.

Thread 1 is acquiring a DB connection.

Thread 1 has acquired the DB connection.

Thread 2 is acquiring a DB connection.

Thread 2 has acquired the DB connection.

Thread 3 is acquiring a DB connection.

Thread 3 has acquired the DB connection.

Thread 0 has released the DB connection.

Thread 1 has released the DB connection.

Thread 2 has released the DB connection.

Thread 3 has released the DB connection.

#### Example: Reader/Writer Thread Pair

There are a number of application scenarios where one thread will populate objects with data while a second (or multiple other) thread reads this data. Such design patterns are called "Produce/Consumer" or "Publish/Subscribe" models.

For this example, we are attempting to simulate the dynamic arrival of market data such as stock prices from an exchange. Such data arrives randomly from a market data provider. There may be period with no updates and periods where there is a high frequency of updates. We simulate this in the data\_writer thread by imposing a random delay in between data "arrivals" which are then loaded into a vector (dynamic array).

We will have one thread that "writes" data into a dynamic array. A second thread will read data from this same dynamic array. To protect the dynamic array from inconsistent reads, we will use a Mutex with .lock and .unlock methods around each use of the array to avoid data corruption.

The data\_reader thread will then iterate over the vector and calculate an average price based on all of the prices that have been received so far. This activity will happen on a fixed time interval.

#include <iostream>

#include <thread>

#include <chrono>

#include <mutex>

#include <vector>

// Mutex to protect our price array

std::mutex mtPriceArray;

// Create an array to hold our prices

std::vector<double> dblPriceArray;

void data\_writer()

{

    // Write our market data into dblPriceArray

    while (1) {

        // random delay

        // Sleep for some random time up to 5 seconds

        std::this\_thread::sleep\_for(std::chrono::milliseconds(rand() % 5000));

        // Lock the price array

        mtPriceArray.lock();

        // Create a price and load it into the array

        // In a real application we would be pulling this price from our market data source

        dblPriceArray.push\_back(20.0 \* (1 + (rand() % 100) / 1000.0));

        // Unlock the price array

        mtPriceArray.unlock();

    }

}

void data\_reader(int delay)

{

    unsigned int loop = 0;

    double dblSum = 0;

    double dblAverage = 0;

    while (1) {

        // Get the lock on the price array

        mtPriceArray.lock();

        if (dblPriceArray.size() > 0) {

            // Calculate the average

            dblSum = 0.0;

            for (loop = 0; loop < dblPriceArray.size(); loop++) {

                dblSum += dblPriceArray[loop];

            }

            dblAverage = dblSum / dblPriceArray.size();

            std::cout << "Latest Price: " << dblPriceArray[dblPriceArray.size() - 1];

            std::cout << " Average: " << dblAverage << std::endl;

        }

        // Unlock the price array vector

        mtPriceArray.unlock();

        // Sleep for some time

        std::this\_thread::sleep\_for(std::chrono::milliseconds(delay));

    } // end while

}

int main()

{

    char result;

    // Launch two new threads. They will start executing immediately

    std::thread writer\_thread(data\_writer);

    std::thread reader\_thread(data\_reader, 2000);

    // Pause the main thread

    std::cout << "Press a key to finish" << std::endl;

    std::cin >> result;

    // Join up the two worker threads to the main thread

    writer\_thread.join();

    reader\_thread.join();

    // Return success

    return 1;

}

Output:

Press a key to finish

Latest Price: 21.34 Average: 21.34

Latest Price: 20 Average: 20.67

Latest Price: 20.48 Average: 20.6067

Latest Price: 21.16 Average: 20.745

Latest Price: 20.9 Average: 20.86

Latest Price: 20.9 Average: 20.86

Latest Price: 20.54 Average: 20.8143

Latest Price: 20.54 Average: 20.8143

Latest Price: 21.82 Average: 20.94

Latest Price: 21.82 Average: 20.94

#### Example: Thread Pool

#include <iostream>

#include <thread>

#include <vector>

#include <queue>

#include <functional>

#include <mutex>

#include <condition\_variable>

class ThreadPool

{

private:

    std::vector<std::thread> m\_threads;         // worker threads in the thread pool

    std::queue<std::function<void()>> m\_tasks;  // tasks to be executed by the worker threads

    std::mutex m\_queueMutex;                    // mutex to synchronize access to the task queue

    std::condition\_variable m\_condition;        // object to notify worker threads when new tasks are added to the queue

    bool m\_stop = false;                        // flag to indicate whether the thread pool should stop processing tasks

public:

    // Create the thread pool based on input number

    ThreadPool(size\_t numThreads)

    {

        for (size\_t i = 0; i < numThreads; ++i) {

            m\_threads.push\_back(std::thread(&ThreadPool::workerThread, this));

        }

    }

    ~ThreadPool()

    {

        std::unique\_lock<std::mutex> lock(m\_queueMutex);

        m\_stop = true;  // signal the worker threads to stop processing tasks

        m\_condition.notify\_all();   // notifies all the worker threads

        // joins all the worker threads to wait for them to complete

        for (std::thread& thread : m\_threads) {

            thread.join();

        }

    }

    // enqueue tasks for execution in the thread pool

    template<typename Func>

    void enqueue(Func func)

    {

        {

            std::unique\_lock<std::mutex> lock(m\_queueMutex);

            m\_tasks.push(std::function<void()>(func));

        }

        m\_condition.notify\_one();   // notify a worker thread that a new task is available

    }

private:

    void workerThread()

    {

        while (true) {

            std::function<void()> task;

            {

                std::unique\_lock<std::mutex> lock(m\_queueMutex);

                // wait for a notification,

                // either until a new task is available or until the m\_stop flag is set to true.

                m\_condition.wait(lock, [this] {

                    return m\_stop || !m\_tasks.empty();

                });

                // If the m\_stop flag is true and the task queue is empty,

                // the worker thread exits the loop and terminates.

                if (m\_stop && m\_tasks.empty()) {

                    return;

                }

                // Otherwise, it retrieves the front task from the queue,

                // removes it from the queue, and unlocks the mutex.

                task = m\_tasks.front();

                m\_tasks.pop();

            }

            // execute the task

            task();

        }

    }

};

// Example task

void printNumber(int number)

{

    std::cout << "Thread ID: " << std::this\_thread::get\_id()

              << ", Number: " << number << std::endl;

}

int main()

{

    // Create a thread pool with 4 threads

    ThreadPool threadPool(4);

    // Enqueue 10 tasks

    for (int i = 0; i < 10; ++i) {

        threadPool.enqueue(std::bind(printNumber, i));

    }

    // Wait for tasks to complete

    std::this\_thread::sleep\_for(std::chrono::seconds(1));

    return 0;

}

Output:

Thread ID: 139760518772288, Number: 0

Thread ID: 139760527164992, Number: 1

Thread ID: 139760535557696, Number: 2

Thread ID: 139760527164992, Number: 3

Thread ID: 139760527164992, Number: 4

Thread ID: 139760527164992, Number: 5

Thread ID: 139760527164992, Number: 7

Thread ID: 139760527164992, Number: 8

Thread ID: 139760527164992, Number: 9

Thread ID: 139760518772288, Number: 6

#### Example: Spinlock

#include <iostream>

#include <thread>

#include <atomic>

#include <vector>

class SpinLock {

    std::atomic\_flag lock\_flag = ATOMIC\_FLAG\_INIT;

public:

    void lock() {

        // Atomically set the flag to true and return the previous value

        while (lock\_flag.test\_and\_set(std::memory\_order\_acquire)) {

            // Spin until the lock is acquired

        }

    }

    void unlock() {

        lock\_flag.clear(std::memory\_order\_release); // Clear the atomic flag

    }

};

SpinLock spinLock;

int sharedVariable = 0;

// Increment the shared variable

void incrementSharedVariable(int loopCount)

{

    for (int i = 0; i < loopCount; ++i) {

        spinLock.lock();    // Acquire the lock

        ++sharedVariable;

        spinLock.unlock();  // Clear the lock

    }

}

int main()

{

    int threadNum = 4;

    int loopCount = 1000;

    std::thread threads[threadNum];

    // Create threads

    for (int i = 0; i < threadNum; ++i) {

        threads[i] = std::thread(incrementSharedVariable, loopCount);

    }

    // Wait for all threads to finish

    for (int i = 0; i < threadNum; ++i) {

        threads[i].join();

    }

    // Print the final value of the shared variable

    std::cout << "Shared variable value: " << sharedVariable << std::endl;

    return 0;

}

Output:

Shared variable value: 4000

#### Example: Barrier

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

class Barrier {

    std::mutex \_mutex;

    std::condition\_variable \_condVar;

    int \_count;

    int \_initial\_count;

public:

    explicit Barrier(int count) :

        \_count(count),

        \_initial\_count(count) {}

    void wait()

    {

        std::unique\_lock<std::mutex> lock(\_mutex);

        if (--\_count == 0) {

            \_condVar.notify\_all();

        } else {

            \_condVar.wait(lock, [this]() { return \_count == 0; });

        }

    }

    void reset()

    {

        std::unique\_lock<std::mutex> lock(\_mutex);

        \_count = \_initial\_count;

    }

};

int threadNum = 4;

Barrier barrier(threadNum);

void workerThread(int id)

{

    std::cout << "Thread " << id << " is waiting at the barrier." << std::endl;

    barrier.wait();

    std::cout << "Thread " << id << " has passed the barrier." << std::endl;

}

int main() {

    std::thread threads[threadNum];

    // Create threads

    for (int i = 0; i < threadNum; ++i) {

        threads[i] = std::thread(workerThread, i);

    }

    // Wait for all threads to finish

    for (int i = 0; i < threadNum; ++i) {

        threads[i].join();

    }

    return 0;

}

Output:

Thread 2 is waiting at the barrier.

Thread 0 is waiting at the barrier.

Thread 3 is waiting at the barrier.

Thread 1 is waiting at the barrier.

Thread 1 has passed the barrier.

Thread 2 has passed the barrier.

Thread 3 has passed the barrier.

Thread 0 has passed the barrier.

Each thread will call wait() and block until the last thread arrives at the barrier. Once the last thread arrives, all threads are unblocked simultaneously. The reset()allows you to reset the barrier to its initial state, enabling it to be used again.

### POSIX Threads

**APIs**: <https://support.sas.com/documentation/onlinedoc/ccompiler/doc750/html/lr2/zid-6908.htm>

**Experience:**

* Create N processes and run them: <https://stackoverflow.com/a/10911719>

### Win32 Thread Libraries

<https://hackernoon.com/need-faster-code-try-multithreading-5dc30c83837c>

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexa>

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openmutexw>

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject>

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-releasemutex>

<https://www.tenouk.com/ModuleAA.html>

JetDrive example: ModeChange\Ini\_Template.h (and .cpp)

## Lock-Free Programming

<https://preshing.com/20120612/an-introduction-to-lock-free-programming/>

## QAs

### Maximum Thread Number?

Suppose my CPU has 4 cores, what is the maximum thread number can run at the same time?

It depends on many factors:

* **CPU Cores**: With 4 cores, you can run 1 thread/core concurrently. That means you could have up to 4 threads running simultaneously without context switching between them.
* **Hyper-Threading**: Some CPUs support Hyper-Threading, which allows each physical core to handle multiple threads simultaneously. With it enabled, each physical core can handle 2 threads. Thus, on a CPU with 4 cores, you could run up to 8 threads concurrently.
* **OS and Scheduler**: The OS's thread scheduler plays a crucial role in managing thread execution and resource allocation. The scheduler determines how threads are allocated to CPU cores and how they are scheduled for execution. The number of threads the scheduler can efficiently manage depends on its algorithms, policies, and the workload characteristics. The scheduler may dynamically adjust the number of active threads based on factors like thread priority, CPU utilization, and available resources.

### Thread-Specific Data (TSD)?

Thread-Specific Data (TSD) or Thread-Local Storage (TLS) is a technique that allows each thread to have its own private data. It prevent data races of global or static variables that are shared among all threads.

Example:

#include <iostream>

#include <thread>

// TSD structure

struct ThreadData {

    std::thread::id threadId;

};

// TSD variable

thread\_local ThreadData threadData;

// Function executed by each thread

void ThreadFunction() {

    // Accessing and modifying thread-specific data

    threadData.threadId = std::this\_thread::get\_id();

    std::cout << "Thread ID: " << threadData.threadId << std::endl;

}

int main() {

    // Create multiple threads

    std::thread t1(ThreadFunction);

    std::thread t2(ThreadFunction);

    std::thread t3(ThreadFunction);

    // Wait for all threads to finish

    t1.join();

    t2.join();

    t3.join();

    return 0;

}

Output:

Thread ID: 140685678634560

Thread ID: 140685695419968

Thread ID: 140685687027264

When you run this code, you should see each thread printing its unique thread ID, indicating that each thread has its own separate copy of the thread-specific data.

# Multi-Processing