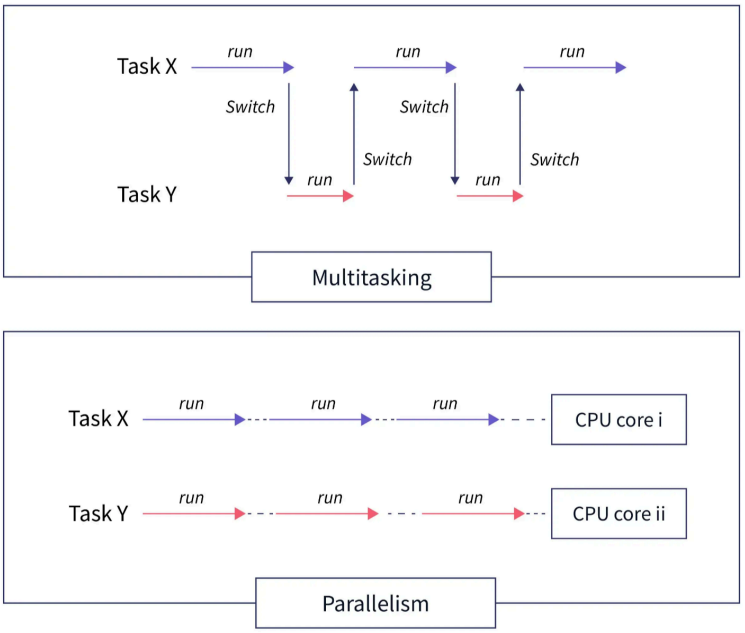
Introduction to Concurrent Systems and Multithreading



Introduction to Concurrency

Concurrency means:

> Two or more tasks happening at the same time or overlapping in progress.

It doesn't always mean they're happening exactly **at the same microsecond**, but they are happening in a way that the brain (or CPU) switches between them smoothly and effectively.

Real-Life Examples

We do **concurrent tasks** every day without thinking!

Examples:

Example	What's Happening?
Listening to music while **solving math**	Your ears and brain handle music, eyes and brain handle numbers — **simultaneous focus**.
Walking and **talking**	Your legs move, and you speak — **two separate actions at once**.
Swimming while **thinking about homework**	**Physical** and **mental** tasks running together.

These examples show that **humans naturally perform multiple activities at once** — this is **natural concurrency**.

Why is this important in Computers?

Computers, like humans, need to handle:

- Multiple **tasks or processes**
- At the same or **interleaved time**
- With good performance and no conflicts

Concurrency in Computer Systems

Just like we juggle tasks in daily life, computers **juggle multiple operations** using **concurrency**.

Examples in Computers:

Task	How Concurrency Helps
Watching a YouTube video while **downloading files**	The system handles video rendering AND data download together
A **web server handling 100 user requests**	Each request handled via **concurrent threads/processes**
Playing a game with music, **graphics**, and **keyboard input**	All handled concurrently using **multithreading**

Concurrency in computer systems involves single and multiple CPUs. Single CPUs switch between applications, causing delays, while multiple CPUs run applications in parallel, reducing switching time. This allows for more efficient processing, though background applications still require some switching.

Threads within processes enable simultaneous tasks, enhancing user experience.

Single CPU

- **Switching Mechanism**: A single CPU handles multiple applications by switching between them. This process involves loading and unloading data for each application, which introduces delays.
- **Performance Drawbacks**: As the number of applications increases, the switching time becomes more noticeable, leading to a lag in performance. For example, if you're running a game and music simultaneously, the CPU may struggle to allocate sufficient time to each, resulting in interruptions (e.g., music playing in chunks).
- **Capacity Limitations**: Older computers with a single CPU can only process one application at a time, even though they give the illusion of parallel processing by switching quickly.

Multiple CPUs

- **Parallel Processing**: With multiple CPUs, applications can run simultaneously on their dedicated CPUs, which eliminates the need for constant switching. Each CPU can handle its workload independently.
- **Efficiency**: This configuration allows for significant savings in switching time, meaning that tasks are completed faster and more efficiently.
- **Background Processes**: Even with multiple CPUs, the operating system still manages several background applications, which can cause some level of switching, but overall performance remains much better compared to a single CPU.

Key Insight:

> “**Concurrency** makes a system more responsive, more efficient, and more real-world capable.”

Concurrency vs Parallelism

Concept	Description

Concurrency	Tasks appear to be running at the same time (interleaved or overlapping)
Parallelism	Tasks truly run at the same time on **multiple cores** or CPUs

Example:

- **Concurrency**: You talk to two people by switching between them quickly (1 brain, fast switching)
- **Parallelism**: You and your friend talk to two people at once (2 brains)

Why do we need concurrency?

Concurrency in computer systems is crucial for performance and separation of concerns. It allows multiple processes to run simultaneously, enhancing system speed and user experience. For instance, in a music player, different threads manage video playback, audio, and GUI interactions, ensuring seamless functionality without delays caused by single-threaded processing.

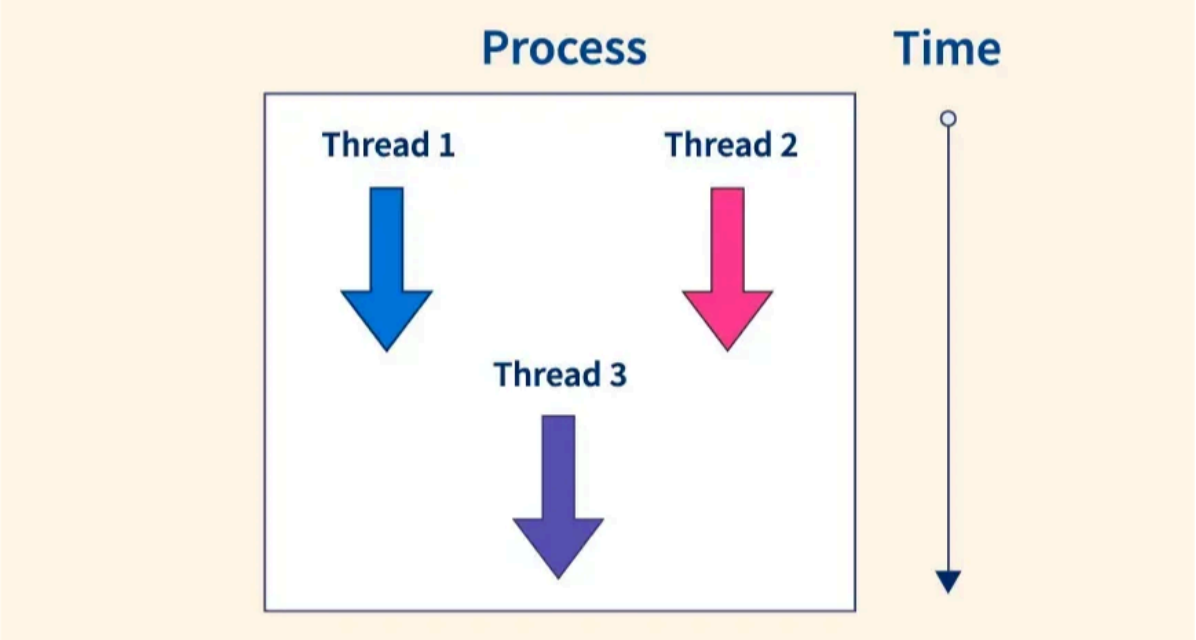
Without concurrency:

- CPU does **one task at a time**
- Wastes time **waiting** (e.g., during file read, download)

With concurrency:

- CPU switches between tasks, or
- If it has multiple cores, it does **actual parallel processing**!

What is Multithreading?



Definition:

> Multithreading is a programming technique that allows multiple threads to run independently and concurrently within the same process/task.

With & Without Multithreading

> **Without Multithreading** (tasks run one after the other)
>
> Example: **Making Maggi and Tea**
>
> Imagine you're alone:
>
> 1. Boil water for Maggi (wait 5 min)
> 2. Cook Maggi (wait 3 min)
> 3. Then boil water for tea (wait 4 min)
> 4. Make tea (wait 2 min)
>
> **Total time: 14 minutes**

> **With Multithreading** (tasks run in parallel)
>
> **You start**:
>
> - Boiling water for both Maggi & Tea at the same time
> - While Maggi is cooking, tea is also getting ready
>
> **Total time: Around 7–8 minutes**

Task: Simulate downloading two files (using `sleep_for`)

Without Multithreading

```
#include <iostream>
#include <chrono>
#include <thread>
using namespace std;

void downloadA() {
    cout << "Downloading File A...\n";
    this_thread::sleep_for(chrono::seconds(3));
    cout << "File A downloaded\n";
}

void downloadB() {
    cout << "Downloading File B...\n";
    this_thread::sleep_for(chrono::seconds(3));
    cout << "File B downloaded\n";
}

int main() {
    auto start = chrono::high_resolution_clock::now();

    downloadA(); // Runs first
    downloadB(); // Runs after A

    auto end = chrono::high_resolution_clock::now();
```

```
    auto duration = chrono::duration_cast<chrono::seconds>(end - start);
    cout << "Time Taken: " << duration.count() << " seconds\n";

    return 0;
}
```

Output:

```
Downloading File A...
(wait 3 sec)
File A downloaded
Downloading File B...
(wait 3 sec)
File B downloaded
Time Taken: 6 seconds
```

With Multithreading

```
#include <iostream>
#include <chrono>
#include <thread>
using namespace std;

void downloadA() {
    cout << "Downloading File A...\n";
    this_thread::sleep_for(chrono::seconds(3));
    cout << "File A downloaded\n";
}

void downloadB() {
    cout << "Downloading File B...\n";
    this_thread::sleep_for(chrono::seconds(3));
    cout << "File B downloaded\n";
}

int main() {
    auto start = chrono::high_resolution_clock::now();
```

```
    thread t1(downloadA); // Both threads run at the same time
    thread t2(downloadB);

    t1.join();
    t2.join();

    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::seconds>(end - start);
    cout << "Time Taken: " << duration.count() << " seconds\n";

    return 0;
}
```

Output:

```
Downloading File A...
Downloading File B...
(wait 3 sec)
File A downloaded
File B downloaded
Time Taken: 3 seconds
```

Summary:

Feature	Without Threads	With Threads
Total Time Taken	6 seconds	3 seconds
Runs in Parallel	No	Yes
CPU Utilization	Low	Better

Why Use Multithreading?

Without Threads	With Threads
Sequential processing	Parallel processing
Wasted CPU time	Full CPU utilization

Slower performance	Faster execution
Unresponsive programs	Smooth and responsive UI

Basic Elements of Multithreaded Programs

1. **Threads**
   - A **thread** is the smallest unit of execution.
   - Multiple threads can run **independently** inside the same program (process).

   ➔ Example: Two threads printing messages at the same time.

2. **Shared Resources**
   - Threads can **share** memory (variables, arrays, files, etc.).
   - This makes communication between threads easy but **can create problems** if two threads modify the same data at the same time.

   ➔ Example: Two threads adding to the same counter.

3. **Synchronization**
   - To avoid **conflicts** when multiple threads access shared resources, we use **synchronization tools** like:
     - **Mutex (mutual exclusion lock)**
     - **Semaphores**
     - **Locks**
   - These tools **allow only one thread** to access critical sections at a time.

   ➔ Example: A mutex is locked when one thread is incrementing the counter and unlocked after.

4. **Thread Creation**
   - We can **create threads** using various ways depending on language (in C++: `std::thread`).

   ➔ Example in C++:

```
thread t1(functionName);
```

5. **Thread Termination**
   - After a thread finishes its task, it **ends**.
   - **Joining threads** means the main program **waits** for all threads to complete.

   → Example in C++:

   ```
   t1.join();  // Main thread waits for t1 to finish
   ```

6. **Communication between Threads**
   - Sometimes threads need to **send information** to each other.
   - Methods: Shared variables + synchronization, message passing, condition variables, etc.

7. **Concurrency vs Parallelism**
   - **Concurrency:**

     Threads are making progress **at the same time** (not necessarily running exactly together).
   - **Parallelism:**

     Threads are actually **running at the same instant** on different CPU cores.

Introduction to POSIX Threads (`pthread.h`)

What is `pthread.h`?

- `pthread.h` stands for **POSIX Threads** (Portable Operating System Interface Threads).
- It is a **C library** for creating and managing **multiple threads** on **Unix/Linux systems**.

- It allows **multithreading**: where a program can **perform multiple tasks** at the same time.

Why use `pthread.h`?

- To **speed up** a program by doing multiple tasks simultaneously.
- To better **utilize CPU cores**.
- To handle **large, independent tasks** like downloading multiple files, handling web server requests, etc.
- To **learn** about thread synchronization (mutex, condition variables, etc.).

How to use `pthread.h`?

You need to:

- **Include** the header:

```
#include <pthread.h>
```

- **Compile** using `pthread` option:

```
gcc program.c -o program -pthread
```

Key Functions in `pthread.h`

Function	Description
`pthread_create()`	Create a new thread
`pthread_exit()`	Exit a thread
`pthread_join()`	Wait for a thread to finish
`pthread_mutex_init()`	Initialize a mutex
`pthread_mutex_lock()`	Lock a mutex
`pthread_mutex_unlock()`	Unlock a mutex
`pthread_mutex_destroy()`	Destroy a mutex

Simple Example Program

```
#include <stdio.h>
#include <pthread.h>

void* myThread(void* arg) {
    printf("Hello from the thread!\n");
    return NULL;
}

int main() {
    pthread_t tid;

    // Create a thread
    pthread_create(&tid, NULL, myThread, NULL);

    // Wait for thread to finish
    pthread_join(tid, NULL);

    printf("Thread finished.\n");

    return 0;
}
```

Output:

```
Hello from the thread!
Thread finished.
```

Important Concepts in `pthread.h`

Threads

- Threads are **lighter** than processes.
- They **share the same memory space**.

Mutex

- A **mutex (mutual exclusion)** prevents multiple threads from accessing a shared resource at the same time.
- Example:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
// critical section
pthread_mutex_unlock(&lock);
```

Thread Synchronization

- Used to **coordinate** threads safely (prevent race conditions).
- Methods:
  - Mutex
  - Condition Variables
  - Semaphores (in `semaphore.h`)

Race Condition

- Happens when **two threads access a shared variable** at the same time without synchronization.
- Example Problem: Both threads try to increment a shared counter at the same time, leading to wrong results.

---

pthread_create() Function Details

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

Parameter	Meaning
`thread`	Pointer to `pthread_t` to hold thread ID
`attr`	Thread attributes (NULL for default)

`start_routine`	Function to be executed by thread
`arg`	Arguments to pass to function (can be NULL)

pthread_join() Function Details

```
int pthread_join(pthread_t thread, void **retval);
```

- **Waits** for the specified thread to **finish**.
- It **blocks** the calling thread until the specified thread exits.

Real Life Analogy

Imagine a **restaurant**:

- **Chef 1** making pizza 🍕
- **Chef 2** making pasta 🍝
- Both chefs are threads!
- They share **kitchen space (memory)**.
- They need to **lock the oven (mutex)** before baking to avoid conflicts!

Student Task 1:

Create Two Threads that Print Messages

Problem Statement:

Write a program to create **two threads**.

Each thread should print a **different message**:

- Thread 1 should print `"Hello from Thread 1"`
- Thread 2 should print `"Hello from Thread 2"`

Use `pthread_create` and `pthread_join`.

Input Format:

No input from user.

Output Format:

```
Hello from Thread 1
Hello from Thread 2
```

(Order may vary because threads run independently.)

▼ **Solution**

```
#include <stdio.h>
#include <pthread.h>

void* printMessage1(void* arg) {
    printf("Hello from Thread 1\n");
    return NULL;
}

void* printMessage2(void* arg) {
    printf("Hello from Thread 2\n");
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, printMessage1, NULL);
    pthread_create(&t2, NULL, printMessage2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

Student Task 2:

Sum of Two Numbers Using Threads

Problem Statement:

Create a program where:

- Thread 1 adds two numbers and prints the sum.
- Thread 2 multiplies two numbers and prints the product.

Use `pthread_create` and `pthread_join`.

Input Format:

Hardcoded inside the program: 5 and 3.

Output Format:

```
Sum = 8
Product = 15
```

(Order may vary.)

▼ **Solution**

```
#include <stdio.h>
#include <pthread.h>

void* addNumbers(void* arg) {
    int a = 5, b = 3;
    printf("Sum = %d\n", a + b);
    return NULL;
}

void* multiplyNumbers(void* arg) {
    int a = 5, b = 3;
    printf("Product = %d\n", a * b);
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, addNumbers, NULL);
```

```
    pthread_create(&t2, NULL, multiplyNumbers, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

Student Task 3:

Divide an Array Sum into Two Parts Using Threads

Problem Statement:

Given an array of 6 numbers: `{2, 4, 6, 8, 10, 12}`,

- Thread 1 should calculate the **sum of first 3 numbers**.
- Thread 2 should calculate the **sum of last 3 numbers**.
- Finally, print the total sum.

Input Format:

Array: `{2, 4, 6, 8, 10, 12}` (Hardcoded)

Output Format:

```
Sum of first half = 12
Sum of second half = 30
Total Sum = 42
```

▼ **Solution**

```
#include <stdio.h>
#include <pthread.h>

int arr[] = {2, 4, 6, 8, 10, 12};
int sum1 = 0, sum2 = 0;
```

```
void* sumFirstHalf(void* arg) {
    for (int i = 0; i < 3; i++) {
        sum1 += arr[i];
    }
    return NULL;
}

void* sumSecondHalf(void* arg) {
    for (int i = 3; i < 6; i++) {
        sum2 += arr[i];
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, sumFirstHalf, NULL);
    pthread_create(&t2, NULL, sumSecondHalf, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Sum of first half = %d\n", sum1);
    printf("Sum of second half = %d\n", sum2);
    printf("Total Sum = %d\n", sum1 + sum2);

    return 0;
}
```

Modern C++ Threads (C++11 Onwards)

What is a Thread?

- A **thread** is an independent unit of execution that runs inside a program (**process**).

- A **process** can have **multiple threads** running **simultaneously**, sharing the same memory.
- Threads help us perform **multiple tasks at once** — this is called **multithreading**.

Why Use Threads in C++?

Without Threads	With Threads
One task at a time.	Multiple tasks at once.
Slower if many tasks.	Faster and responsive.
Poor CPU utilization.	Better CPU utilization.

Multithreading improves **speed**, **efficiency**, and **responsiveness**.

How to Work with Threads in C++?

C++11 introduced a **thread library** `<thread>` that makes it super easy to create and manage threads.

Header file:

```
#include <thread>
```

Basic way to create a thread:

```
void myFunction() {
    // Task
}

int main() {
    std::thread t1(myFunction); // Create thread
    t1.join(); // Wait for thread to finish
}
```

Important Thread Functions

Function	Purpose

`std::thread t(func)`	Creates a new thread.
`t.join()`	Main thread waits for t to finish.
`t.detach()`	Separate thread execution, cannot join later.
`t.joinable()`	Checks if a thread can be joined.
`std::this_thread::sleep_for`	Pause thread for some time.
`std::this_thread::get_id()`	Get unique ID of the thread.

Example 1: Basic Thread

```
#include <iostream>
#include <thread>
using namespace std;

void sayHello() {
    cout << "Hello from Thread!" << endl;
}

int main() {
    thread t1(sayHello); // Create a thread
    t1.join();           // Wait for t1 to complete
    cout << "Main thread ends." << endl;
    return 0;
}
```

Example 2: Thread with Arguments

```
#include <iostream>
#include <thread>
using namespace std;

void greet(string name) {
    cout << "Hello, " << name << endl;
}

int main() {
```

```
    thread t1(greet, "Anand");  // Pass argument to thread
    t1.join();
    return 0;
}
```

Important Concepts in Multithreading

1. Race Condition

- Happens when two or more threads try to modify shared data at the same time.
- Results are **unpredictable**.

Example:

```
// Two threads increment counter → Race condition
counter++;
```

---

2. Mutex (Mutual Exclusion)

- Mutex is used to **lock** the critical section so that **only one thread** can access it at a time.

Example:

```
#include <mutex>
mutex m;

void increment() {
    m.lock();
    counter++;
    m.unlock();
}
```

Or better (safe) way:

```
void increment() {
    lock_guard<mutex> guard(m);
```

```
    counter++;
}
```

(lock_guard automatically unlocks when going out of scope)

3. Joining vs Detaching Threads

`join()`	`detach()`
Main thread waits for this thread to finish.	Main thread does NOT wait.
Safer to use.	Risky if thread accesses deleted resources.
Example: `t1.join();`	Example: `t1.detach();`

Things to Remember

- Always call `join()` or `detach()` on a thread, otherwise your program will crash!
- Use `mutex` to avoid **data corruption**.
- Prefer using **lock_guard** instead of manual `lock` and `unlock`.

C++ Thread vs Process

Feature	Process	Thread
Memory	Separate memory space.	Shared memory.
Creation	Heavyweight.	Lightweight.
Communication	Need IPC (Inter-process communication).	Easy (shared variables).
Example	Chrome tabs as processes.	Different tasks like loading images and playing video inside a tab.

Quick Example: Sum of Array using Threads

```
#include <iostream>
#include <thread>
#include <vector>
using namespace std;
```

```
int partial_sum = 0;
mutex m;

void sumPart(vector<int>& arr, int start, int end) {
    int temp = 0;
    for (int i = start; i < end; i++) {
        temp += arr[i];
    }
    lock_guard<mutex> guard(m);
    partial_sum += temp;
}

int main() {
    vector<int> arr = {1,2,3,4,5,6,7,8,9,10};
    thread t1(sumPart, ref(arr), 0, 5);
    thread t2(sumPart, ref(arr), 5, 10);

    t1.join();
    t2.join();

    cout << "Total Sum: " << partial_sum << endl;
    return 0;
}
```

Summary

Thread: smallest unit of execution inside a program.

Create Thread: `std::thread t(func);`

Synchronize: Use `mutex` and `lock_guard` to protect shared data.

Join or Detach threads properly.

Multithreading = Better performance + faster programs!

Would you also like me to create a **visual diagram** for

> "How Threads are created, executed, and managed in C++"?
>
> It'll make it even easier to remember!

> **Shall I create that too?**

Life Cycle of a Thread

What is the Life Cycle of a Thread?

- Just like a human has different stages of life — a **thread** also passes through **different stages** from its creation to completion.
- Understanding the **thread life cycle** helps us to manage threads **properly** and **avoid errors**.

Stages in the Life Cycle of a Thread

Stage	Meaning
1. New (Created)	Thread is created but not running yet.
2. Runnable	Thread is ready to run and waiting for CPU.
3. Running	Thread is executing its task.
4. Blocked/Waiting	Thread is paused, waiting for some resource or condition.
5. Terminated (Dead)	Thread has completed its task or forcefully stopped.

Detailed Explanation of Each Stage

1. New (Created) State

- Thread object is **created** but **not started yet**.
- No execution of code.

```
thread t1(myFunction);  // Created
// Thread created but not yet running (in C++, constructor starts immediatel
y unless you control)
```

2. Runnable (Ready) State

- Thread is **ready to run**.
- Waiting for CPU time to be scheduled.
- In C++, once the thread is created, it's almost immediately **runnable**.

> Think of a student raising their hand, waiting for the teacher to call them.

---

3. Running State

- CPU **allocates time** to the thread.
- The thread is **executing** its function now.

```
void task() {
    cout << "Task is running!" << endl;
}
```

- After getting CPU time, thread **actually works**.

---

4. Blocked or Waiting State

- Thread **pauses** because:
  - Waiting for **I/O** (input/output),
  - Waiting for **lock (mutex)**,
  - Waiting for **some event** to happen.

Example:

```
mtx.lock();
// Thread might wait here if mutex is already locked by another thread
counter++;
mtx.unlock();
```

- Blocked threads **cannot continue** until they get the resource.

---

5. Terminated (Dead) State

- Thread finishes its task successfully or forcibly ends.
- Cannot restart a terminated thread.

```
t1.join();  // Main thread waits for t1 to complete
// After join, thread ends and goes to terminated state
```

Visual Diagram

```
[ New (Created) ]
       ↓ (start)
[ Runnable (Ready) ]
       ↓ (CPU gives time)
[ Running ]
   ↓          ↓
(Blocked/Waiting)  (Completes normally)
       ↓
   (Resumes after resource)
       ↓
[ Running again ]
       ↓
[ Terminated (Dead) ]
```

Real-Life Example Analogy

Imagine a **student giving a presentation**:

Thread State	Real-Life Example
New (Created)	Student is assigned a topic but hasn't started yet.
Runnable (Ready)	Student stands in line to present.
Running	Student is presenting.
Blocked/Waiting	Student pauses because projector is not working.
Terminated (Dead)	Student finishes the presentation.

Important Points to Remember

In C++, `std::thread` immediately starts after creation unless you delay it manually.

Always call `join()` or `detach()` to **safely clean up** the thread.

Threads **share memory** in C++, so blocked/waiting often happens during **mutex** or **resource access**.

Quick Code Example: Thread Life Cycle

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
using namespace std;

mutex m;

void printTask() {
    m.lock();
    this_thread::sleep_for(chrono::seconds(2)); // Blocked for 2 seconds
    cout << "Thread is running..." << endl;
    m.unlock();
}

int main() {
    thread t1(printTask); // New → Runnable → Running
    t1.join(); // Main waits
    cout << "Thread finished." << endl; // Terminated
    return 0;
}
```

Summary

New (Created) → Thread is created.

Runnable → Thread is ready and waiting for CPU.

Running → Thread is executing.

Blocked/Waiting → Thread is paused due to unavailable resources.

Terminated → Thread has completed or exited.

Writing Parallel Algorithms for

Parallel Vector Addition and Matrix Multiplication using Multithreading

Important Topics You Should Know Before These Programs:

Before you can write parallel algorithms like **Vector Addition** or **Matrix Multiplication**, you should understand:

Topic	Why It's Needed
Threads	So you can divide work among multiple execution units.
Mutex (optional)	If you share data between threads, mutex is needed for safety. (In our case, mostly not needed.)
Loops and Arrays	Basic knowledge of `for` loops and 1D/2D arrays.
Thread Synchronization	Understanding `join()` to wait for all threads to complete.

Quick Recap of Threads

(You must know this before proceeding.)

Create Thread:

```
#include <thread>
thread t1(function_name);
```

Wait for Thread to Finish:

```
t1.join();
```

Multiple Threads Example:

```
thread t1(func1);
thread t2(func2);
```

```
t1.join();
t2.join();
```

---

1. Parallel Vector Addition using Multithreading

Problem Statement

> Write a program to add two vectors using multiple threads.
>
> Each thread will add a **portion** of the arrays.

Input

- Two arrays of integers (same size).
- Number of threads.

Output

- One resultant array containing the sum of elements.

Parallel Approach

1. Divide the array into **equal parts** for each thread.
2. Each thread **adds** its assigned portion independently.
3. Combine results.

Code: Parallel Vector Addition

```
#include <iostream>
#include <vector>
#include <thread>
using namespace std;

void addVectors(vector<int>& A, vector<int>& B, vector<int>& Result, int st
art, int end) {
```

```
    for (int i = start; i < end; ++i) {
        Result[i] = A[i] + B[i];
    }
}

int main() {
    int n = 8;
    vector<int> A = {1,2,3,4,5,6,7,8};
    vector<int> B = {8,7,6,5,4,3,2,1};
    vector<int> Result(n, 0);

    int num_threads = 4;
    vector<thread> threads;

    int chunk_size = n / num_threads;

    for (int i = 0; i < num_threads; i++) {
        int start = i * chunk_size;
        int end = (i == num_threads - 1) ? n : start + chunk_size;
        threads.push_back(thread(addVectors, ref(A), ref(B), ref(Result), start,
end));
    }

    for (auto& t : threads) {
        t.join();
    }

    cout << "Result Vector: ";
    for (auto val : Result) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

Key Points

- `ref()` is used to pass vectors **by reference** to avoid copying.
- Each thread only works on its **own part**, so **no mutex** is needed.
- Properly `join()` all threads.

Output

```
Result Vector: 9 9 9 9 9 9 9 9
```

---

2. Parallel Matrix Multiplication using Multithreading

Problem Statement

> Write a program to multiply two matrices using multiple threads.
>
> Each thread will calculate a **portion** (for example, some rows) of the resultant matrix.

Input

- Two 2D arrays (matrices).
- Number of threads.

Output

- Product of two matrices.

Parallel Approach

1. Divide **rows** of the first matrix among threads.
2. Each thread computes the product for its assigned rows.
3. Combine the results into the final matrix.

Code: Parallel Matrix Multiplication

```
#include <iostream>
#include <vector>
#include <thread>
using namespace std;

void multiplyMatrices(vector<vector<int>>& A, vector<vector<int>>& B, ve
ctor<vector<int>>& Result, int start_row, int end_row) {
    int n = B.size();         // Number of rows in B
    int m = B[0].size();      // Number of columns in B

    for (int i = start_row; i < end_row; ++i) {
        for (int j = 0; j < m; ++j) {
            Result[i][j] = 0;
            for (int k = 0; k < n; ++k) {
                Result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    vector<vector<int>> A = {{1, 2}, {3, 4}, {5, 6}};
    vector<vector<int>> B = {{7, 8}, {9, 10}};
    int rows = A.size();
    int cols = B[0].size();

    vector<vector<int>> Result(rows, vector<int>(cols, 0));

    int num_threads = 2;
    vector<thread> threads;

    int chunk_size = rows / num_threads;

    for (int i = 0; i < num_threads; i++) {
        int start_row = i * chunk_size;
        int end_row = (i == num_threads - 1) ? rows : start_row + chunk_size;
```

```
        threads.push_back(thread(multiplyMatrices, ref(A), ref(B), ref(Result),
start_row, end_row));
    }

    for (auto& t : threads) {
        t.join();
    }

    cout << "Result Matrix:" << endl;
    for (auto& row : Result) {
        for (auto& val : row) {
            cout << val << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Key Points

- Rows are divided among threads.
- Each thread works on different rows → **no need for locking**.
- Always `join()` all threads to complete before printing.

Output

```
Result Matrix:
25 28
57 64
89 100
```

Understanding Shared Memory Model and Race Condition Challenges

What is Shared Memory Model?

In **multithreaded programming**, when multiple threads run **inside the same process**,

they **share**:

- the **same memory space** (like global variables, heap memory, static variables).
- the **same file descriptors** (files opened), and
- the **same code section**.

👉 **Shared Memory Model** means:

Threads share data and resources directly **without sending messages**.

Example

```
#include <iostream>
#include <thread>
using namespace std;

int counter = 0; // Shared memory: counter

void increment() {
    for (int i = 0; i < 100000; ++i) {
        counter++;
    }
}

int main() {
    thread t1(increment);
    thread t2(increment);

    t1.join();
    t2.join();

    cout << "Counter: " << counter << endl;
```

```
    return 0;
}
```

Expected output:

`Counter: 200000`

Actual output:

Some random value like `137892`, `190003` ❌

because of **race condition**.

What is Race Condition?

Race Condition happens when:

- **Two or more threads access the same shared resource** (like a variable),
- **At the same time**,
- And at least **one thread modifies it**,
- Without proper **synchronization**.

> ✨ In simple words:
>
> "Threads are racing to update the same memory at the same time, causing wrong/unexpected results."

Why Race Condition Happens?

When two threads **read**, **modify**, and **write** at the **same time**,

they **overwrite** each other's work.

Steps Where Problem Happens:

For `counter++`, this actually happens internally:

1. **Read** counter from memory.
2. **Increment** it by 1.
3. **Write** it back to memory.

👉 These steps are **NOT atomic** (not single step).

One thread might read old value before another thread writes new value → problem!

Race Condition Visualization:

Step	Thread 1	Thread 2
1	Reads counter = 5	
2		Reads counter = 5
3	Increments to 6	Increments to 6
4	Writes counter = 6	Writes counter = 6

Result should be 7, but both wrote 6!

One increment is **lost**.

Problems Caused by Race Conditions:

- Wrong calculations.
- Data corruption.
- Crashes.
- Hard-to-find bugs (timing issues).

How to Solve Race Conditions?

Use Synchronization Tools like:

Technique	Use
Mutex (Mutual Exclusion)	Lock the resource when using it.
Semaphores	Controlling access to multiple resources.
Atomic operations	Some updates like increment can be made atomic.
Condition Variables	For thread communication.

Mutex Example to Solve Above Problem

```
#include <iostream>
#include <thread>
```

```
#include <mutex>
using namespace std;

int counter = 0;
mutex mtx; // Create mutex

void increment() {
    for (int i = 0; i < 100000; ++i) {
        mtx.lock();    // Lock before access
        counter++;
        mtx.unlock();  // Unlock after done
    }
}

int main() {
    thread t1(increment);
    thread t2(increment);

    t1.join();
    t2.join();

    cout << "Counter: " << counter << endl;
    return 0;
}
```

Now **Correct Output** will be `200000` every time.

Important Terms to Know

Term	Meaning
Critical Section	Part of the code where shared resource is accessed.
Deadlock	Two or more threads are waiting for each other forever.
Atomic Operation	Operation which completes fully without interruption.

Real World Examples of Race Conditions

- **Banking System**: Two ATMs updating the same account balance at the same time.
- **Online Shopping**: Two people trying to buy the last item in stock.
- **Gaming**: Two players modifying shared score simultaneously.

In Short:

> "Shared memory is powerful but dangerous.
>
> Proper locking and synchronization are must to avoid Race Conditions."

Additional Important Topics

Deadlock in C++

What is Deadlock?

In **multithreaded programming**, a **deadlock** occurs when:

- **Two or more threads are blocked forever**,
- Each waiting for the other to release a resource.

> **In simple words:**
>
> "**Thread A** waits for **Thread B**'s resource and **Thread B** waits for **Thread A**'s resource →
>
> Both keep waiting forever."

How Deadlock Happens

Deadlocks usually happen when:

- **Two or more mutexes** are **acquired in different orders** by different threads.

- Threads **lock one mutex** and **try to lock another**, but the second mutex is already locked by another thread.

Basic Deadlock Example

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

mutex mtx1;
mutex mtx2;

void taskA() {
    mtx1.lock();
    cout << "Thread A acquired mtx1\n";
    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate work

    mtx2.lock(); // Wait for mtx2
    cout << "Thread A acquired mtx2\n";

    // Critical section

    mtx2.unlock();
    mtx1.unlock();
}

void taskB() {
    mtx2.lock();
    cout << "Thread B acquired mtx2\n";
    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate work

    mtx1.lock(); // Wait for mtx1
    cout << "Thread B acquired mtx1\n";

    // Critical section

    mtx1.unlock();
```

```
    mtx2.unlock();
}

int main() {
    thread t1(taskA);
    thread t2(taskB);

    t1.join();
    t2.join();

    cout << "Finished\n";
}
```

Output:

```
Thread A acquired mtx1
Thread B acquired mtx2
```

Then program freezes (deadlock!)

Because:

- `Thread A` holds `mtx1`, waits for `mtx2`.
- `Thread B` holds `mtx2`, waits for `mtx1`.

Neither can proceed — **Deadlock**.

Conditions Required for Deadlock

For a deadlock to occur, **all 4 conditions must be true**:

Condition	Meaning
Mutual Exclusion	Resources cannot be shared. Only one thread can use a resource at a time.
Hold and Wait	Thread holds one resource and waits for another.
No Preemption	Resources cannot be forcibly taken away.
Circular Wait	A circular chain of threads exists, each waiting for a resource held by the next thread.

How to Prevent Deadlock?

Several strategies:

Strategy	How
Lock ordering	Always acquire mutexes in the same order in all threads.
Try-lock approach	Use `try_lock()` instead of `lock()` to avoid waiting forever.
Timeouts	Use timed locks (`std::timed_mutex`) — abort if cannot lock within time.
Deadlock detection	Monitor program for deadlocks (advanced).

Example: Correct Lock Ordering to Prevent Deadlock

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

mutex mtx1;
mutex mtx2;

void taskA() {
    lock(mtx1, mtx2); // Lock both at once in same order
    lock_guard<mutex> lk1(mtx1, adopt_lock);
    lock_guard<mutex> lk2(mtx2, adopt_lock);

    cout << "Thread A acquired mtx1 and mtx2\n";

    // Critical section
}

void taskB() {
    lock(mtx1, mtx2); // Lock both at once in same order
    lock_guard<mutex> lk1(mtx1, adopt_lock);
    lock_guard<mutex> lk2(mtx2, adopt_lock);

    cout << "Thread B acquired mtx1 and mtx2\n";
```

```
    // Critical section
}

int main() {
    thread t1(taskA);
    thread t2(taskB);

    t1.join();
    t2.join();

    cout << "Finished\n";
}
```

No Deadlock because **both threads lock mutexes in same order** using `lock()`.

Important C++ Functions Related to Deadlock Prevention

Function	Purpose
`std::lock(mtx1, mtx2)`	Locks multiple mutexes safely to avoid deadlock.
`std::try_lock(mtx1, mtx2)`	Tries to lock multiple mutexes, if cannot, returns immediately.
`std::timed_mutex`	Mutex that allows locking with timeout.

In Real World

Deadlocks are very serious because:

- Programs may **freeze** or **crash**.
- They are very **hard to debug** (no error shown).
- They happen randomly based on **timing**.

Deadlock Visualization (Simple Diagram)

```
Thread A ---- locks -⟶ Mutex 1 ---- waits for -⟶ Mutex 2 (locked b
y Thread B)
```

```
Thread B ---- locks -⟶ Mutex 2 ---- waits for -⟶ Mutex 1 (locked b
y Thread A)
```

Both keep waiting forever!

In Short:

> "Deadlock is a situation where two or more threads are waiting for each other's resources, causing infinite blocking.
>
> **Prevention** can be done by proper lock ordering, using `lock()`, or using `try_lock()`."

Thread Pool

What is a Thread Pool?

A **Thread Pool** is:

> A collection of pre-created, reusable threads that are kept ready to perform tasks whenever needed.

- Threads are **created once** and **reused** to execute multiple tasks.
- It **avoids the overhead** of creating and destroying a thread every time a task needs to run.

Why Use a Thread Pool?

✅ **Benefits**:

- **Faster execution** (threads already exist).
- **Better resource management** (control the number of threads).
- **No thread creation overhead** for every task.
- **Efficient for server-based or high-performance applications**.

Simple Real-World Analogy

Imagine a restaurant:

- You hire **5 chefs** at the start of the day (thread pool).
- Whenever a food order (task) comes, **assign** it to a free chef.
- You **don't hire a new chef** for every new order (avoiding overhead).
- After completing a dish, **chefs stay idle but ready** for next orders.

> Chefs = Threads
>
> **Orders = Tasks**

Key Components of Thread Pool

Component	Description
Worker Threads	A fixed number of threads created once and used to perform tasks.
Task Queue	Tasks are kept in a queue if all threads are busy.
Task Dispatcher	Assigns pending tasks to available threads.
Synchronization	Needed to manage task queue safely (mutex, condition variables).

Very Simple Code Outline of Thread Pool (Idea)

```
- Create a fixed number of threads
- Each thread:
    - Waits for a task from the task queue
    - Executes the task
    - Goes back to waiting
- When main program submits a new task:
    - Put it into task queue
    - Notify one waiting thread
```

Basic Example of Thread Pool

Here’s a basic thread pool concept:

```
#include <iostream>
#include <vector>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <functional>
using namespace std;

class ThreadPool {
private:
    vector<thread> workers;
    queue<function<void()>> tasks;

    mutex queue_mutex;
    condition_variable condition;
    bool stop;

public:
    ThreadPool(size_t threads);
    ~ThreadPool();

    void enqueue(function<void()> task);
};

ThreadPool::ThreadPool(size_t threads) : stop(false) {
    for (size_t i = 0; i < threads; i++) {
        workers.emplace_back([this] {
            while (true) {
                function<void()> task;

                {
                    unique_lock<mutex> lock(this->queue_mutex);
                    this->condition.wait(lock, [this] { return this->stop || !this->task
s.empty(); });
                    if (this->stop && this->tasks.empty())
                        return;
                    task = move(this->tasks.front());
```

```
                this->tasks.pop();
            }

            task();
        }
    });
  }
}

void ThreadPool::enqueue(function<void()> task) {
  {
    unique_lock<mutex> lock(queue_mutex);
    tasks.push(task);
  }
  condition.notify_one();
}

ThreadPool::~ThreadPool() {
  {
    unique_lock<mutex> lock(queue_mutex);
    stop = true;
  }
  condition.notify_all();
  for (thread &worker : workers)
    worker.join();
}

int main() {
  ThreadPool pool(4); // 4 worker threads

  for (int i = 0; i < 8; i++) {
    pool.enqueue([i] {
      cout << "Task " << i << " is executing by thread " << this_thread::ge
t_id() << endl;
      this_thread::sleep_for(chrono::milliseconds(500));
    });
  }
```

```
    this_thread::sleep_for(chrono::seconds(3));
    return 0;
}
```

Output:

```
Task 0 is executing by thread 140212545488576
Task 1 is executing by thread 140212537095872
Task 2 is executing by thread 140212528703168
Task 3 is executing by thread 140212520310464
Task 4 is executing by thread 140212545488576
Task 5 is executing by thread 140212537095872
Task 6 is executing by thread 140212528703168
Task 7 is executing by thread 140212520310464
```

Threads are reused automatically!

Advanced Features That Real Thread Pools Might Have:

- Priority-based tasks
- Dynamic resizing of thread pool (adding/removing threads)
- Handling exceptions inside tasks
- Futures and Promises for returning task results
- Load balancing between threads

Key Points To Remember:

Point	Details
Fixed number of threads	Usually created once during pool initialization.
Task queue	Tasks are submitted and workers pick them up.
Synchronization	Needed to safely push/pop tasks from queue (mutex + condition_variable).

Graceful shutdown	Must ensure all threads are joined before program ends.

When to use Thread Pool?

- Web Servers (handling multiple client requests)
- File Processing in background
- Image processing tasks
- Video encoding
- Large scientific calculations (parallelism)

In Short:

> "Thread Pool is a programming pattern where a fixed number of threads are kept ready to perform tasks, avoiding the overhead of thread creation and improving performance."

Happy Coding!