Parallel and Concurrent Programming with C++

Index

1. Threads and Processes

a. Introduction

- When a computer runs an application, that instance of the program executing is referred to as a **process**.
- A process consists of the program's code, its data, and information about its state.
- Each process is independent.
- Each process has its own separate address space and memory.
- Within every process, there are one or more smaller sub-elements called `threads`.
- Each of those `threads` is an independent path of execution through the program, a different sequence of instructions.
- `Thread`s can only exist as part of a process.
- `Thread`s are the basic units that the operating system manages, and it allocates time on the processor to actually execute them.
- Threads that belong to the same process share the process's address space, which gives them access to the same resources and memory, including the program's executable code and data.

```cpp
#include <iostream>
#include <thread>       // Needed for thread
#include <chrono>       // Needed for chrono::seconds
#include <process.h>    // Needed for _getpid()

/*
    This program prints the following... It has 3 threads in it.

Main Process ID: 20420
Main THREAD ID: 21732
CPU Waster Process ID: 20420
CPU Waster THREAD ID: 5304
CPU Waster Process ID: 20420
CPU Waster THREAD ID: 9328
*/

using namespace std;

void cpu_waster()
{
    cout << "CPU Waster Process ID: " <<  _getpid() << endl;
    cout << "CPU Waster THREAD ID: " << this_thread::get_id() << endl;

    while (true)
        continue;
}

int main()
{
    cout << "Main Process ID: " << _getpid() << endl;
    cout << "Main THREAD ID: " << this_thread::get_id() << endl;

    thread thread1(cpu_waster);
    thread thread2(cpu_waster);

    while (true)
        this_thread::sleep_for(chrono::seconds(1));

    return 0;
}
```

b. Concurrent Vs Parallel Execution

Concurrency: Ability of a program to be broken into parts that can run independently of each other.
Example:

```
Salad Recipe

1. Chop Lettuce.
2. Chop Cucumbers.
3. Chop Tomatoes.
4. Chop Onions.
5. Mix Chopped Vegetables.
6. Add Dressing.
```

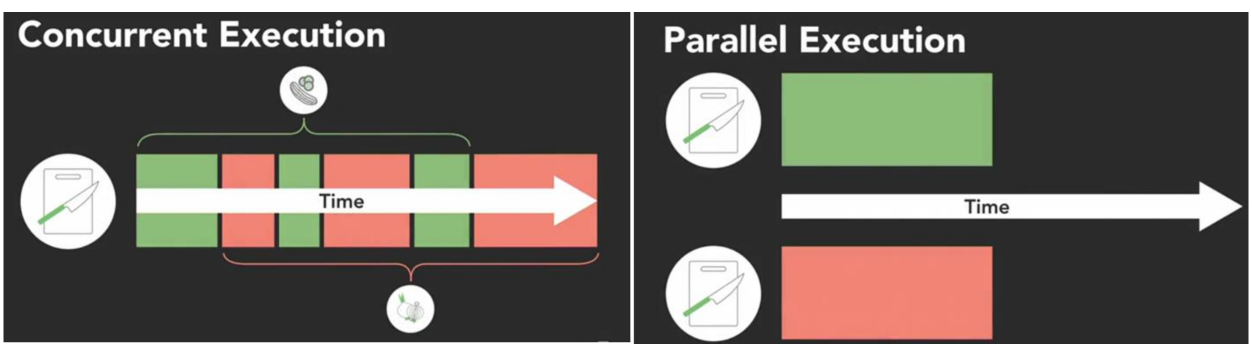
Concurrent Tasks

Chop Lettuce	Chop Tomatoes
Chop Cucumbers	Chop Onions

Order independent

Parallel Execution: Simultaneous Execution, this requires parallel hardware.

Concurrency	Parallelism
Program Structure	Simultaneous Execution
Dealing with multiple things at once	Doing multiple thigs at Once



Concurrency

Definition: Concurrency is the ability to manage multiple tasks at the same time. It involves structuring a program to allow several tasks to make progress without necessarily executing them simultaneously.

Execution: Tasks are not necessarily executed at the same time; they can be interleaved on a single processing unit. This means that the CPU switches between tasks, giving the illusion that they are running concurrently.

Use Case: Concurrency is useful in scenarios where tasks spend a lot of time waiting (e.g., I/O operations), allowing other tasks to run while waiting.

Example: In a web server, multiple requests can be handled concurrently. While one request is waiting for data from a database, the server can process another request.

Parallelism

Definition: Parallelism is the simultaneous execution of multiple tasks to complete them faster. It takes advantage of multiple processing units (cores or processors) to execute tasks at the same time.

Execution: Tasks are executed simultaneously on separate processors or cores, achieving actual simultaneous execution.

Use Case: Parallelism is beneficial for CPU-bound tasks where computation-intensive operations can be divided into smaller tasks and executed at the same time.

Example: A scientific computation that can be split into smaller calculations, such as calculating elements of a large matrix, can run on multiple CPU cores simultaneously.

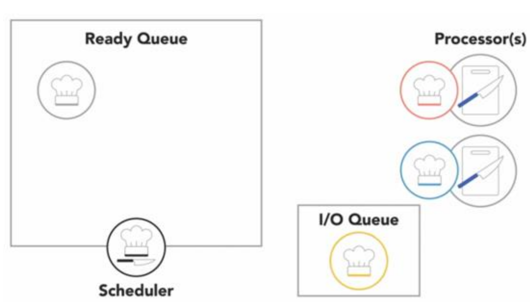
Key Differences

Aspect	Concurrency	Parallelism
Nature	Managing multiple tasks (interleaved execution)	Simultaneous execution of tasks
Execution	Can run on a single core (time-sliced)	Requires multiple cores or processors
Focus	Structure and organization of tasks	Performance and speed
Task Types	I/O-bound tasks (waiting for external resources)	CPU-bound tasks (intensive computations)
Example	Handling multiple client connections in a server	Performing computations on a large dataset

c. Execution scheduling

- Scheduling is operating system's job.
- The OS includes a scheduler that controls when different threads and processes get their turn to execute on the CPU.
- The scheduler makes it possible for multiple programs to run concurrently on a single processor.

- When a process is created and ready to run it gets loaded into memory and placed in the ***ready queue***.
- The ***scheduler*** is like the head chef that tells the other cooks when they get to use the cutting board (***Processor***).
- It cycles through the ready processes so they get a chance to execute on the processor.
- If there are multiple processors, then the OS will schedule processes to run on each of them to make the most use of the additional resources.
- A process will run until it finishes and then the scheduler will assign another process to execute on that processor.



- Or, a process might get blocked and have to wait for an I/O event in which case, it'll go into a separate ***I/O waiting queue*** so another process can run.
- Or, the scheduler might determine that a process has spent its fair share of time on the processor and swap it out for another process from the ready queue. When that occurs, it's called a ***context switch***.
- The operating system must save the state or context of the process that was running so it can be resumed later and it must load the context of the new process that's about to run.
- There's a wide variety of ***algorithms*** that different operating system ***schedulers implement***.
- Some of these algorithms are ***preemptive*** which means *they may pause or preempt a running low-priority task when a higher priority task enters the ready state*.
- In non-preemptive algorithms, once a process enters the running state it'll be allowed to run for its allotted time.
- Some of the "Scheduling Algorithms" are…
    - First come, first served.
    - Shortest job next.
    - Priority.
    - Shortest remaining time.
    - Round-robin.
    - Multiple level queues.

- Scheduling demo code…

```
#include <iostream>
#include <thread>       // Needed for thread
#include <chrono>       // Needed for chrono::seconds
#include <process.h>    // Needed for _getpid()
using namespace std;

bool chopping = true;
/* Output of the Program:
                        Run - 1
-------------------------------------------------------
Chef - I and Chef - II are chopping vegetables...
Chef - II chopped 735150601 vegetables.
Chef - I chopped 735039786 vegetables.
-------------------------------------------------------
                        Run - 2
-------------------------------------------------------
Chef-I and Chef-II are chopping vegetables...
Chef-I chopped 735097496 vegetables.
Chef-II chopped 735636918 vegetables.
-------------------------------------------------------
*/
void vegetable_chopper(const char* name)
{
    unsigned int vegetable_count = 0;
    while (chopping)
    {
        vegetable_count++;
    }
    printf("%s chopped %u vegetables.\n", name, vegetable_count);
}

int main()
{
    thread chef1(vegetable_chopper, "Chef-I");
    thread chef2(vegetable_chopper, "Chef-II");

    //cout << "Chef-I and Chef-II are chopping vegetables..." << endl;
    printf("Chef-I and Chef-II are chopping vegetables...\n");
    this_thread::sleep_for(chrono::seconds(1));
    chopping = false;
    chef1.join();
    chef2.join();

    return 0;
}
```

- Even though these two threads started and stopped at roughly the same time, they chopped very different number of vegetables.
- During the 2nd run now Chef-I and Chef-II both end with different amounts than before.
- Scheduling is not consistent from run to run, so your programs should not rely on execution scheduling for correctness.

d. The std::thread class

- The thread class is implemented using the C++ idiom for classes which own resources, RAII. "Resource Acquisition Is Initialization".
- The class acquires ownership of the resource in its constructor, and releases it in the destructor.
- This is how unique_ptr has ownership of allocated memory, fstream has ownership of a file handle, and so on. The thread object has ownership of an execution thread.
- For any system execution thread, only one object can be bound to it at any one time. So that means that we cannot copy thread objects.
- This means that the thread class is move-only. We can move objects, but not copy them. When we perform a move operation, this transfers the ownership of the execution thread.
- If we are passing a thread object as a function argument, we have to use pass by move. If we are passing a named object, an lvalue, we have to put that inside a call to std::move(), to cast it to an rvalue. Or alternatively, we can just pass a temporary thread object.
- If you are writing a function which takes a thread argument, then it is probably a good idea to put in the double ampersands (&&). When we have double ampersands and it is not a template parameter, that means it must be an rvalue.

```
using namespace std::literals;
void Hello() {
      printf("Hello from Thread!\n");
      std::this_thread::sleep_for(2s);
}
void ExecuteTask(std::thread&& t) {
      printf("Received Thread ID: %d\n", t.get_id());
      t.join();
}
int main() {
      std::thread thr1(Hello);
      ExecuteTask(std::move(thr1));
      return 0;
}
```

- If we are returning a thread object, that is much easier. The compiler will automatically move it for us, regardless of whether we have a named variable, an lvalue, or a temporary object (an rvalue.)
- It is a bad idea to put a call to std::move() while returning, because that may confuse the compiler's optimizer. With exceptions, the basic process is the same as in single-threaded code.

Handling Exception

- When an exception is thrown, the destructors are called for all the objects in the scope where the exception is thrown.
- The program will try to find a suitable handler. It will move up the stack and destroy all the objects there. And it will carry on until, it reaches the top of the stack. And if it is not found a suitable handler, then it will terminate the program.
- The difference is, with threads, that each thread has its own execution stack. So, if the program gets to the top of the thread's execution stack without finding a handler, then it will terminate the entire program.
- With the std::thread class, there is no way for other threads in the program to catch the exception. With the std::thread class, it has to be handled in the thread where the exception is thrown. To deal with an exception, you just use a try/catch block in the normal way, in that thread.

```
void Hello() {
	try {
		throw std::exception();
	}
	catch (std::exception& e) {
		std::cout << "Caught: " << e.what() << "\n";
	}
	printf("Hello from Thread!\n");
}
int main() {
	std::thread t(Hello);
	t.join();
	return 0;
}
```

Exception In Parent thread

- The destructors will be called, for every object which is in scope. And that will include the `thread class`. The destructor will check whether `join()` or `detach()` have been called.
- If neither of these have been called, and there is an active `thread` of execution, then this will call `terminate()`.

```
/*This program crashes*/
void hello() {
	printf("Hello, Thread!\n");
}

int main() {
	try {
		std::thread t(hello);
		throw std::exception();
		t.join();
	}
	catch (std::exception& e) {
		printf("Exception Caught: %s\n", e.what());
	}
	return 0;
}
```

- We must make sure that either `join()` or `detach()` are called, in every path through the code.
- Fix is to move the `thread` object outside `try` block and call `join()` in `catch` block as well.

```
int main()
{
	std::thread t(hello);
	try {
		throw std::exception();
		t.join();
	}
	catch (std::exception& e) {
		printf("Exception Caught: %s\n", e.what());
		t.join();
	}
	return 0;
}
```

- A more C++ solution is to use the RAII idiom. We can write a wrapper `class` which has a `thread` object as a member, and the destructor of this `class` will call `join()` on the `thread` object.
- There is a `joinable()` member function, which will tell us if we need to call `join()`.

```
class ThreadGuard
{
	std::thread t_;

public:
	ThreadGuard(std::thread&& t) : t_(std::move(t)) { }
	~ThreadGuard() {
		if (t_.joinable()) {
			t_.join();
		}
	}

	ThreadGuard(const ThreadGuard&) = delete;
	ThreadGuard& operator=(const ThreadGuard&) = delete;
};
```

- If `join()` or `detach()` have already been called, or if there is no active execution `thread`. Then in that case `joinable()` will return `false`, and we do not need to call `join()`. Otherwise it will return `true` and we do need to call `join()`.
- In C++20, we have the `jthread` `class`, which solves most of these problems. So there is no need to call `join()`. And it also allows co-operative interruption. We can ask a `jthread` to suspend itself.

```
int main() {
	try {
		std::jthread t(hello);
		throw std::exception();
	}
	catch (std::exception& e) {
		printf("Exception Caught: %s\n", e.what());
	}
	return 0;
}
```

e. Ways to Launch Threads

- Using a Function Pointer

```cpp
void my_function(int x) {
    // ...
}

int main() {
    std::thread t(my_function, 42);
    // ...
}
```

- Using a Lambda Expression

```cpp
int main() {
      std::thread t(
            []() {
                  printf("Hello World\n");
            }
      );

      t.join();
}
```

- Using a Function Object (Functor)

```cpp
class MyFunctor {
public:
    void operator()(int x) {
        printf("Value: %d\n", x);
    }
};

int main() {
    MyFunctor functor;
    thread t(functor, 99);
    t.join();
    return 0;
}
```

- Using `std::bind`

```
void my_function(int x, std::string y) {
    printf("Int: %d, String: %s\n", x, y.c_str());
}

int main() {
    auto bound_function = std::bind(my_function,
        42,
        std::placeholders::_1);

    std::string s = "Hello";
    std::thread t(bound_function, s);
    t.join();
    return 0;
}
```

  - The first argument of `my_function` is bound to `42`.
  - The second argument is a placeholder that will be provided when `bound_function` is called with `s`.

- Using `std::async`

```
#include <future>
int doubleIt(int x) {
    return x * 2;
}

int main() {
    std::future<int> future = std::async(std::launch::async,
        doubleIt,
        42);

    // Wait for the result
    int result = future.get();
    printf("Returned Value: %d", result);
    return 0;
}
```

- The `<future>` header, which provides facilities for asynchronous operations such as `std::async` and `std::future`.
- `std::async` is used to run the `doubleIt` function asynchronously.
- `std::launch::async` specifies that the function should be run in a separate `thread`.
- When you use `std::launch::`deferred, it means that the task won't start running immediately. Instead, the execution of the task is deferred until you actually request the result. This is different from the other policy, `std::launch::async`, where the task starts running in a separate `thread` right away.
- `doubleIt` is the function to be called.
- `42` is the argument passed to `doubleIt`.
- `std::future<int> future;` is a `future` object that will hold the result of the asynchronous operation.
- `future.get()` is called to retrieve the result of the asynchronous operation.
- If the result is not ready yet, `future.get()` will block (`wait`) until the calculation is complete.
- Once the calculation is done, `future.get()` will return the result, which is stored in the `result` variable.

f. Thread life cycle

- When a new process or program begins running it will start with just one `thread`, which is called the main `thread` because it's the main one that runs when the program begins.
- That main `thread` can then start or spawn additional `thread`s to help out, referred to as its child `thread`s, which are part of the same process but execute independently to do other tasks. Those `thread`s can spawn their own children if needed.
- As each of those `thread`s finish executing, they'll notify their parent and terminate with the main `thread` usually being the last to finish execution.
- Over the lifecycle of a `thread` from creation through execution and finally termination, `thread`s will usually be in one of four states.

New State

- If main `thread` spawns or create another `thread`, that child `thread` will begin in the **new state**.
- This `thread` isn't actually running yet so it doesn't take any CPU resources.
- Part of creating a new `thread` is assigning it a function, the code it's going to execute.
- Some programming languages require you to explicitly start a `thread` after creating it.

Runnable State

- If `thread` starts running, then it is in **runnable state**, which means the operating system can schedule this `thread` to execute.
- Through contact switches, this `thread` will get swapped out with other `thread`s to run on one of the available processors.

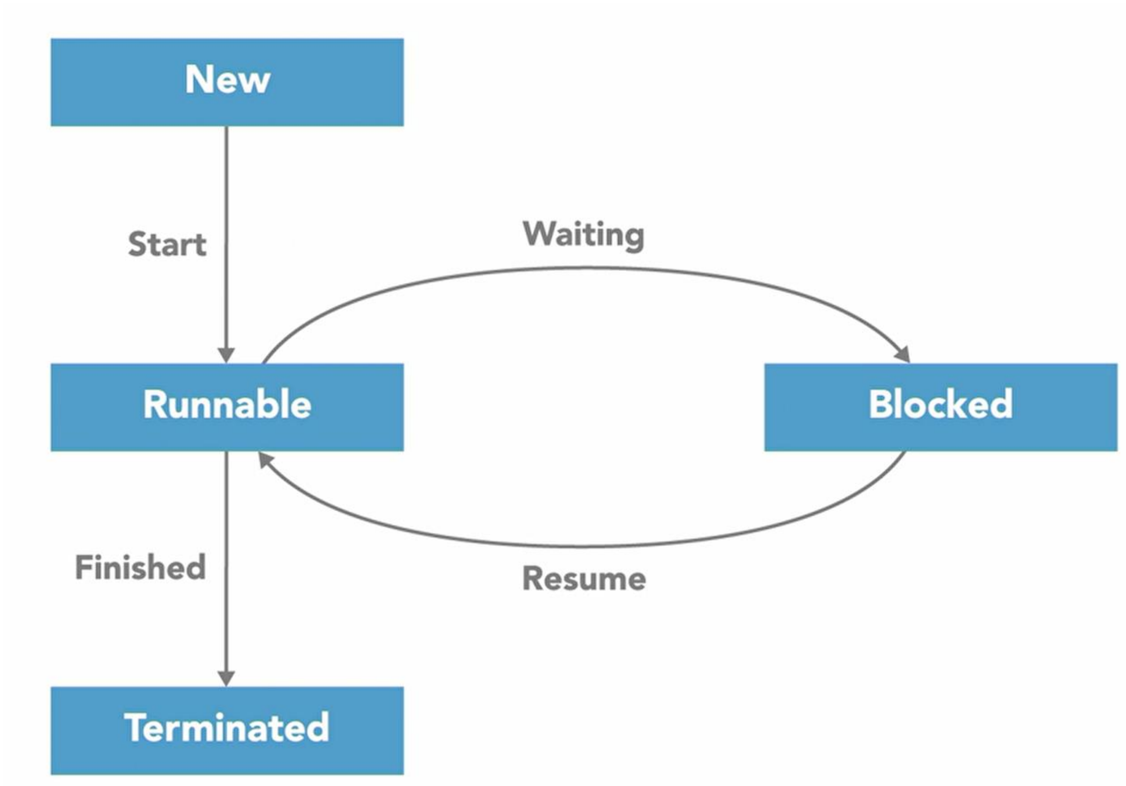
Blocked State

- When a `thread` needs to wait for an event to occur, like an external input or a timer, it goes into a **blocked state** while it waits.
- The good thing is that while a `thread` is blocked, it is not using any CPU resources.

- Main `thread` may eventually reach a point where it needs to wait until one of its children `thread`s has finished for it to continue on.
- Maybe the main `thread` has finished everything else. It can wait for child `thread` to complete its execution by calling the `join` method.
- When the main `thread` calls `join`, main `thread` will enter a **blocked state** waiting until its child `thread` is done.

Terminated State

- Once the child `thread` finished executing, it will notify its parent `thread` that it is done.
- After notifying its parent `thread` it will enter the final **terminated state**.



- A `thread` enters the **terminated state** when it either completes its execution or is abnormally aborted.
- Once the child `thread` notifies the main `thread` that it is done, the main `thread` will return to the runnable state.

- Different programming languages may use different names for their states and have a few additional ones.
- In general, **new**, **runnable**, **blocked**, and **terminated** are the four phases of the lifecycle of a `thread`.

```
#include <iostream>
#include <thread>       // Needed for thread
#include <chrono>       // Needed for chrono::seconds
#include <process.h>    // Needed for _getpid()
using namespace std;

/*
Main thread requests Child's help.
Main thread continues its work.
Child thread started & waiting for IO Operation...
Main thread patiently waits for Child to finish and join...
Child is done executing.
Main thread and Child are both done!
*/
void child_thread()
{
    printf("Child thread started & waiting for IO Operation...\n");
    std::this_thread::sleep_for(std::chrono::seconds(3));
    printf("Child is done executing.\n");
}

int main()
{
    printf("Main thread requests Child's help.\n");
    std::thread theChild(child_thread);

    printf("Main thread continues its work.\n");
    std::this_thread::sleep_for(std::chrono::seconds(1));

    printf("Main thread patiently waits for Child to finish and join...\n");
    theChild.join();

    printf("Main thread and Child are both done!\n");
    return 0;
}
```

g. Detached thread

- We often create `thread`s to provide some sort of service or perform a periodic task in support of the main program.

- A common example of that is garbage collection. A garbage collector is a form of automatic memory management that runs in the background, and attempts to reclaim garbage, or memory that's no longer being used by the program.
- Imagin that a main `thread` spawns a child `thread` named `garbage_collect` to collect the garbage.
- When main `thread` is done executing and it is ready to exit the program, but it can't, because child `thread` is still running.
- Since `main` `thread` spawned `garbage_collect` `thread` as normal child `thread`, `main` `thread` won't be able to execute until `garbage_collect` `thread` terminated.
- Since `garbage_collect` `thread` is designed to collect garbage in a continuous loop, it never exit.
- The `main` `thread` will be stuck and is waiting forever, and this process will never terminate.
- `Thread`s that are performing background tasks like garbage collection can be detached from the `main` program by making them what's called a daemon `thread`.
- The Daemon `thread` is a `thread` that will not prevent the program from exiting if it's still running.
- By default, new `thread`s are usually spawned as non-daemon, or normal `thread`s, and you must explicitly turn a `thread` into a daemon or background `thread`.
- When `main` `thread` is finished executing and there isn't any non-daemon `thread`s left running, this process can terminate and the `garbage_collect` daemon `thread` will terminate with it.
- Since `garbage_collect` `thread` was terminated abruptly with the process, and it didn't have a chance to gracefully shutdown and stop what it was doing. That's fine, in the case of a garbage collection routine because all of the memory this process was using will get cleared as part of terminating it.
- But if a `thread` was doing some sort of io operation like writing to a file, then terminating in the middle of that operation could end up corrupting data.

- If you detach a `thread` to make it a background task, make sure it won't have any negative side effects if it prematurely exits.

Example: Before Detaching a Thread

```
#include <iostream>
#include <thread>       // Needed for thread
#include <chrono>       // Needed for chrono::seconds
using namespace std;

/*
Main thread is doing its work...
garbage_collector reclaimed some memory.
Main thread is doing its work...
garbage_collector reclaimed some memory.
Main thread is doing its work...
Main thread is done!
garbage_collector reclaimed some memory.
garbage_collector reclaimed some memory.
garbage_collector reclaimed some memory.
...
*/

void garbage_collector()
{
    while (true)
    {
        printf("garbage_collector reclaimed some memory.\n");
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

int main()
{
    std::thread gc(garbage_collector);
    for (int i = 0; i < 3; i++)
    {
        printf("Main thread is doing its work...\n");
        std::this_thread::sleep_for(std::chrono::milliseconds(600));
    }
    printf("Main thread is done!\n");
    gc.join();

    return 0;
}
```

Example: After Detaching a Thread

```
#include <iostream>
#include <thread>       // Needed for thread
#include <chrono>       // Needed for chrono::seconds

using namespace std;

/*
Main thread is doing its work...
garbage_collector reclaimed some memory.
Main thread is doing its work...
garbage_collector reclaimed some memory.
Main thread is doing its work...
Main thread is done!
*/

void garbage_collector()
{
    while (true)
    {
        printf("garbage_collector reclaimed some memory.\n");
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

int main()
{
    std::thread gc(garbage_collector);
    gc.detach();
    for (int i = 0; i < 3; i++)
    {
        printf("Main thread is doing its work...\n");
        std::this_thread::sleep_for(std::chrono::milliseconds(600));
    }
    printf("Main thread is done!\n");
    //gc.join();

    return 0;
}
```

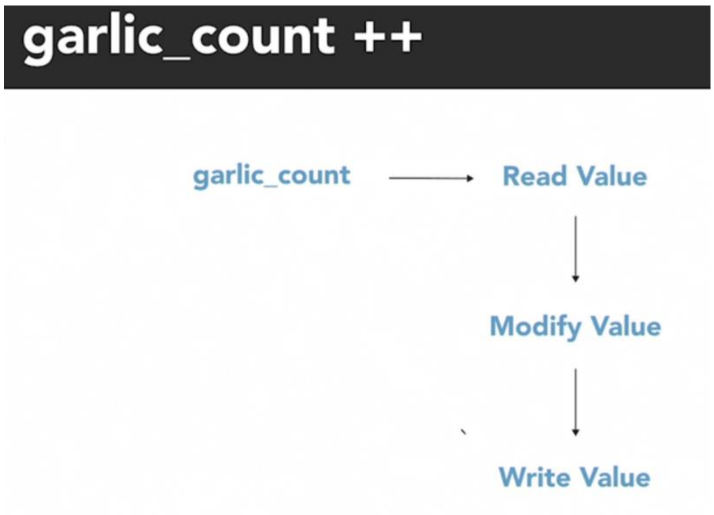
2. Mutual Exclusion

a. Data race

- Data races are a common problem that can occur when two or more `thread`s are concurrently accessing the same location in memory, and at least one of those `thread`s is writing to that location to modify its value.
- Fortunately, you can protect your program against data races by using synchronization techniques.
- First need to know how to recognize the data race. As we can see in the below demo, 2 `thread`s are planning to buy a total of `200000` garlic but end up in buying only `159951`.

```
#include <iostream>
#include <thread>        // Needed for thread
using namespace std;

/*
We should buy 159951 garlic.
*/

unsigned int garlic_count = 0;

void shopper()
{
    for (int i = 0; i < 100000; i++)
    {
        garlic_count++;
    }
}

int main() {
    std::thread me(shopper);
    std::thread myWife(shopper);
    me.join();
    myWife.join();
    printf("We should buy %u garlic.\n", garlic_count);
    return 0;
}
```

- Even though the simple `garlic_count++` operation is only a single line of code, in the background the computer is actually performing a three-step read, modify, write process.



- My two concurrent `thread`s end up stepping on each other's toes and unintentionally overwriting each other's changes to produce an incorrect result.

b. Mutual exclusion

- Anytime multiple `thread`s are concurrently reading and writing a shared resource, it creates the potential for incorrect behavior, like a data race. But we can defend against that by identifying and protecting ***critical sections*** of code.
- A ***critical section***, or ***critical region***, is part of a program that accesses a shared resource, such as a data structured memory, or an external device, and it may not operate correctly if multiple `thread`s concurrently access it.
- The ***critical section*** needs to be protected so that it only allows one `thread` or process to execute in it at a time.
- The above program experienced a data race as the 2 `thread`s added garlic to the shared variable `garlic_count`, because incrementing a value is actually a three-step process.
   i. Read the current value.
   ii. Modify it.

iii. Write back the result.

- Those three steps are a critical section, and they need to execute as an uninterrupted action, so we don't accidentally overwrite each other.
- `Mutex`, short for ***mutual exclusion***, which you'll also hear referred to as a ***lock***.
- A `mutex` has only two states.
    - Locked
    - Unlocked
- Only one `thread` or process can have possession of a ***lock*** at a time so it can be used to prevent multiple `threads` from simultaneously accessing a shared resource, forcing them to take turns.
- The operation to acquire the ***lock*** is an atomic operation, which means it's always executed as a single indivisible action.
- To the rest of the system, an atomic operation appears to happen instantaneously, even if under the hood, it really takes multiple steps. The key here is that the atomic operation is uninterruptible.
- Acquiring the `mutex` is an atomic action that no other `thread` can interfere with halfway through. Either you have the `mutex`, or you don't.
- `Threads` that try to acquire a ***lock*** that's currently possessed by another `thread` can pause and wait 'til it's available.
- We should not forget to release the `mutex` when you're done.
- Since `threads` can get blocked and stuck waiting for a `thread` in the critical section to finish executing, it's important to keep the section of code protected with the `mutex` as short as possible.
- There are just three member functions that we need to know about. Concerned with locking and unlocking the `mutex`.
    - `lock()`
    - `try_lock()`
    - `unlock()`

```cpp
#include <iostream>
#include <thread>       // Needed for thread
#include <mutex>        // Needed for mutex

using namespace std;

unsigned int garlic_count = 0;
std::mutex pencil;

void shopper()
{
    for (int i = 0; i < 5; i++)
    {
        printf("Shopper %d is thinking...\n", std::this_thread::get_id());
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
        pencil.lock();
        garlic_count++;
        pencil.unlock();
    }
}

int main() {
    std::thread me(shopper);
    std::thread myWife(shopper);
    me.join();
    myWife.join();
    printf("We should buy %u garlic.\n", garlic_count);
    return 0;
}
```

c. Internally synchronized classes

- The containers in the C++ standard library need to be synchronized. Before we call a member function, we could, for example, lock a `mutex`. And that would prevent a data race and we need to do that, every time we call any of the member functions on a shared C++ library container. So, that would be external synchronization.
- We can also write `class`es which provide their own synchronization.
- The `class` takes the responsibility for preventing the data race, as opposed to the person who calls its member functions.
- One way to do that is to have a `mutex` as a data member, an additional data member and then the member functions of this `class` will lock the `mutex`, before they access any of the `class`'s internal data and then they will unlock it afterwards.

- So that will prevent a data race, without the person who calls the member functions having to do anything.

```
class Vector {
    std::mutex mut_;
    std::vector<int> vec_;
public:
    void push_back(const int& i) {
        mut_.lock();
        // Start of critical section
        vec_.push_back(i);
        // End of critical section
        mut_.unlock();
    }

    void print() {
        mut_.lock();
        // Start of critical section
        for (const auto i : vec_) {
            std::cout << i << ", ";
        }
        // End of critical section
        mut_.unlock();
    }
};
```

d. The `std::atomic` objects

- Using a lock to protect a shared variable with mutual exclusion works.
- But if you're only doing simple operations, like incrementing a variable's value, then the simpler solution is to use C++ `atomic` types which encapsulate a value and synchronize access to it to prevent a data race.

```
#include <iostream>
#include <thread>        // Needed for thread
#include <atomic>        // Needed for atomic objects

std::atomic<unsigned int> garlic_count(0);

void shopper() {
   for (int i = 0; i < 10000000; i++)
      garlic_count++;
}

int main() {
   std::thread me(shopper);
   std::thread myWife(shopper);
   me.join();
   myWife.join();
   printf("We should buy %u garlic.\n", garlic_count.load());
   return 0;
}
```

- When we make a variable an `atomic` type, all the operations on it will be `atomic`. Other `thread`s will not be able to interleave during that operation and interfere with the operation on the variable.
- We must initialize `atomic` variables.
- The type parameter must be copyable, which basically means a built-in type or a compound object, where all the members are built-in types.
- We have `store()` function that will atomically replace the value of the object with the argument to the call.
- We have the `load()` function will atomically return the object's value.
- Both `store()` and `load()` are similar to assignment and return value.
- There is also an `exchange()` member function. This will also replace the object's value, like the store and assignment, but it returns the previous value.

```cpp
#include <atomic>
int main()
{
   std::atomic<int> x = 0;
   std::cout << "After initialization: x = " << x << '\n';

   // Atomic assignment to x
   x = 2;

   // Atomic assignment from x. y can be non-atomic
   int y = x;

   std::cout << "After assignment: x = " << x << ", y = " << y << '\n';

   x.store(100); // Similar to assignment operator.
   // load() is similar to return value.
   std::cout << "After store(100): x = " << x.load() << '\n';

   // x.exchange(y) will store the value present in y into x and
   // returns the old vlaue of x which is 100.
   std::cout << "Exchange returns " << x.exchange(y) << '\n';
   std::cout << "After exchange: x = " << x << ", y = " << y << '\n';
}
/*
After initialization: x = 0
After assignment: x = 2, y = 2
After store(100): x = 100
Exchange returns 100
After exchange: x = 2, y = 2
*/
```

e. The `std::atomic_flag`

- The `std::atomic_flag` is a simple atomic type designed for efficient synchronization between `thread`s.
- It provides only basic operations for flag manipulation `test_and_set()` and `clear()` operations.
  - The `test_and_set()` function atomically sets the flag to `true` and `return`s its previous value.
  - The `clear()` function resets the flag to `false`.
- It must be explicitly initialized, usually to `false` using `ATOMIC_FLAG_INIT`.
- This `std::atomic_flag` is used in implementing Spinlock.

f. Thread-Local Storage (TLS)

- The `thread_local` storage specifier, which is used to declare variables that should be unique to each `thread`.
- When a variable is declared with `thread_local`, each `thread` gets its own independent copy of the variable. Any modification of this variable by one `thread` does not affect the value seen by other `thread`s.

```
thread_local int counter = 0;
void incrementCounter() {
    for (int i = 0; i < 5; ++i) {
        ++counter;
        printf("Thread %d and counter: %d\n",
            std::this_thread::get_id(), counter);
    }
}
```

- The `thread_local` variables can be…
  - Global variables, or they could be at `namespace` scope.
  - Data members of a `class`.
  - Local variables, in a function or a scope.

Use case

- **Per-Thread Logging**
  - In a multithreaded application, logging is often required for debugging or monitoring. A typical logging system might use a shared log file or output stream. However, managing simultaneous access to the shared log by multiple `thread`s can lead to performance issues due to locking.
  - Using TLS, each `thread` can maintain its own log buffer, accumulating log messages locally. Periodically or upon reaching a certain threshold, each `thread` flushes its buffer to the shared log in a single, efficient operation. This approach minimizes contention on the shared log resource.

```
thread_local std::stringstream logBuffer;
void logMessage(const std::string& message) {
    logBuffer << "Thread " << std::this_thread::get_id()
        << ": " << message << std::endl;
}
void flushLog() {
    std::cout << logBuffer.str();
    logBuffer.str(""); // Clear the buffer
}
void worker() {
    logMessage("Starting work");
    logMessage("Processing data...");
    logMessage("Finishing work");
    flushLog();
}
```

- **Caching Thread-Specific Data**
  - Sometimes, `thread`s need to repeatedly access certain data. Storing this data as a global variable requires synchronization, which can slow down the program. Instead, using TLS allows each `thread` to cache its own copy, avoiding the need for locks and improving performance.

3. Locks

- Let's look at the problem which with the mutex. Let's suppose we have a task function, which might throw an exception in its critical section.

```
std::mutex task_mutex;
void task(const std::string& str)
{
	try {
		// Lock the mutex before the critical section
		task_mutex.lock();

		// Start of critical section
		// Critical section throws an exception
		throw std::exception();
		// End of critical section

		// Never gets called
		task_mutex.unlock();
	}
	catch (std::exception& e) {
		std::cout << "Exception caught: " << e.what() << '\n';
	}
}
```

- We lock a mutex before entering the critical section, and we unlock the mutex afterwards. When the exception is thrown, somewhere in this code, the thread will jump out of the try block, into the catch block and the code which follows is not executed.
- That means that the unlock function is never called. So, the mutex is left in a locked state. The program has just completely ground to a halt.
- So, when the exception is thrown, we have the stack unwinding process. The destructors are called, for all objects in scope.
- The program flow jumps into the catch handler. The `unlock()` call is never executed, and the mutex remains in a locked state.
- So, any other thread which wants to lock that mutex is going to wait, and it is going to wait forever.

- The threads are blocked and if any code has called join() on those blocked threads. Perhaps the main() function, then that code is going to be blocked as well. So that is why the entire program was blocked.
- So, this is one of the drawbacks of using the mutex class. If you call lock(), then there must be a call to unlock() in every path through the code. Including when exceptions are thrown. If you do not do that, if you miss a call somewhere, then the mutex is going to remain locked.
- So that is why we do not normally use the mutex class directly.
- The C++ library provides some wrapper classes for mutexes.
- These have a mutex object as a private member.
- They are defined in the same header as the mutex class and these use the RAII idiom, for managing resources.
- In this case, the resource is a mutex which is locked. The constructor will acquire this resource by locking the mutex and the destructor releases the resource, by unlocking the mutex.
- We create objects of this class on the stack. Then when the object goes out of scope, the destructor is called and the mutex is unlocked. This will always happen automatically, even if an exception is thrown.

a. The std::lock_guard class

- This is a very basic class. It just has a constructor and destructor.
- The constructor takes a mutex object as argument. It moves it into its member, and then locks it.
- The destructor will just unlock the mutex member.
- The lock_guard is a template class.
- We have to give the type of the mutex as a type parameter, and that is because C++ has different types of mutex.
- So, to create a lock_guard object, we need to give the mutex type as the parameter. And then the mutex object, as the argument to the constructor.
- With the help of std::lock_guard we can rewrite the code as shown below...

```
std::mutex task_mutex;
void task(const std::string& str) {
      try {
            // Create an std::lock_guard object
            // This calls task_mutex.lock()
            std::lock_guard<std::mutex> l_g(task_mutex);

            // Start of critical section
            // Critical section throws an exception
            throw std::exception();
            // End of critical section

            // Code after critical section.
            std::this_thread::sleep_for(50ms);
      }     // Calls ~std::lock_guard
      catch (std::exception& e) {
            std::cout << "Exception caught: " << e.what() << '\n';
      }
}
```

b. The `std::unique_lock` class

- We saw in the `std::lock_guard` section that if there is any code which comes between the critical section and the destructor call, then that code will be executed while the `mutex` is still locked. This is because, the `std::lock_guard` goes releases the lock at the closing braces. So, this will prevent other `thread`s from locking the `mutex`, and entering their critical section.
- So, this could affect the performance of the program.
- So, this is where the `unique_lock` comes in.
- It has the same basic features: A data member which is a `mutex`.
- The constructor locks it and the destructor unlocks it, but it also has a member function for unlocking the `mutex`. And we can call that, immediately after the critical section.
- That will allow other `thread`s to lock the `mutex` and perform their critical sections while we continue and do something, which is not critical.
- The `std::unique_lock` object will remember that the `mutex` has been unlocked. So, it will not try to unlock it again when the object is destroyed.

- If we do not call `unlock()`, then eventually the destructor will unlock the `mutex`. So, the lock is always released eventually.

```
using namespace std::literals;
std::mutex task_mutex;
void task(const std::string& str) {
      try {
            // Create an std::unique_lock object
            // This calls task_mutex.lock()
            std::unique_lock<std::mutex> u_l(task_mutex);

            // Start of critical section
            // Critical section throws an exception
            throw std::exception();
            // End of critical section

            u_l.unlock();
            // using namespace std::literals; for 'ms'
            std::this_thread::sleep_for(50ms);
      }     // Calls ~std::unique_lock
      catch (std::exception& e) {
            std::cout << "Exception caught: " << e.what() << '\n';
      }
}
```

- The constructor works the same way. We give the type of the `mutex` as parameter, and the `mutex` object as argument. The constructor will wait until the `mutex` is locked before it returns.
- Then it is safe to enter the critical section, because we know that we are the only `thread` which is executing this code.
- And now, with the `std::unique_lock`, we can unlock the `mutex` immediately after the critical section.
- If we don't `unlock()` the mutex immediately after the critical section, the `thread` will go to `50ms` sleep while having the `mutex`, no other `thread` is able to enter the critical section.
- What are the main differences between `std::lock_guard` and `std::unique_lock`?
    - `std::lock_guard` has only a constructor and destructor

  - `std::unique_lock` has other member functions, including `lock()` and `unlock()`
  - `std::lock_guard`'s constructor can only perform a blocking lock.
  - `std::unique_lock` 's constructor has a number of options for locking
    - `std::try_to_lock`
    - `std::defer_lock`
    - `std::adopt_lock`
- `std::unique_lock` is a move only `class` (Similar to `unique_ptr`). We cannot copy objects, but we can move them. And when that happens, the ownership of the locked `mutex` is transferred, from one object to another.

c. Recursive mutex

- Only one `thread` at a time can own or have a ***lock*** on it and only that `thread` can access the shared resource.
- If I attempt to ***lock*** the ***mutex*** while another `thread` has it, my `thread` will be blocked, and I need to wait until the other `thread` unlocks the ***mutex*** so it becomes available.
- If my `thread` is in a recursive call and it has already ***locked*** a ***mutex*** and if I attempt to ***lock*** the ***mutex***, it doesn't appear to be available so my `thread` will just have to wait too. My `thread` can't unlock the ***mutex*** while I'm blocked waiting on it and I'll be waiting on the ***mutex*** forever because I'll never be able to ***unlock*** it.
- If a `thread` tries to lock a ***mutex*** that it's already locked, it'll enter into a waiting list for that ***mutex***, which results in something called a ***deadlock*** because no other `thread` can unlock that ***mutex***.
- There may be times when a program needs to ***lock*** a ***mutex*** multiple times before ***unlocking*** it. In that case, you should use a ***reentrant mutex*** to prevent this type of problem.

d. Reentrant mutex

- A ***reentrant mutex*** is a particular type of ***mutex*** that can be ***locked multiple times*** by the same process or `thread`.
- Internally, the ***reentrant mutex*** keeps track of how many times it's been locked by the owning `thread` and it has to be ***unlocked*** an equal number of times before another `thread` can ***lock*** it.

```cpp
#include <iostream>
#include <thread>       // Needed for thread
#include <mutex>        // Needed for recursive_mutex

unsigned int garlic_count = 0;
unsigned int potato_count = 0;
std::recursive_mutex pencil;

void add_garlic() {
    pencil.lock();
    garlic_count++;
    pencil.unlock();
}

void add_potato() {
    pencil.lock();
    potato_count++;
    add_garlic();
    pencil.unlock();
}

void shopper()
{
    for (int i = 0; i < 10000; i++)
    {
        add_garlic();
        add_potato();
    }
}

int main()
{
    std::thread me(shopper);
    std::thread myWife(shopper);
    me.join();
    myWife.join();
    printf("We should buy %u garlic.\n", garlic_count);
    printf("We should buy %u potatoes.\n", potato_count);
    return 0;
}
```

e. Try lock

- Suppose if a `thread` tries to lock a `mutex` which is already locked by another `thread`, it will be in the waiting list of that `mutex` (blocked). If the `thread` had some other activities to perform, it cannot perform as it is blocked.
- So, rather than using the standard ***locking*** method to acquire the `mutex`, we will use what's called ***try lock***, or ***try enter***, which is a non-blocking version of the ***lock*** or acquire method.
- It returns immediately, and one of two things will happen.

I. If the mutex you're trying to lock is available, it will get locked and the method will return true.

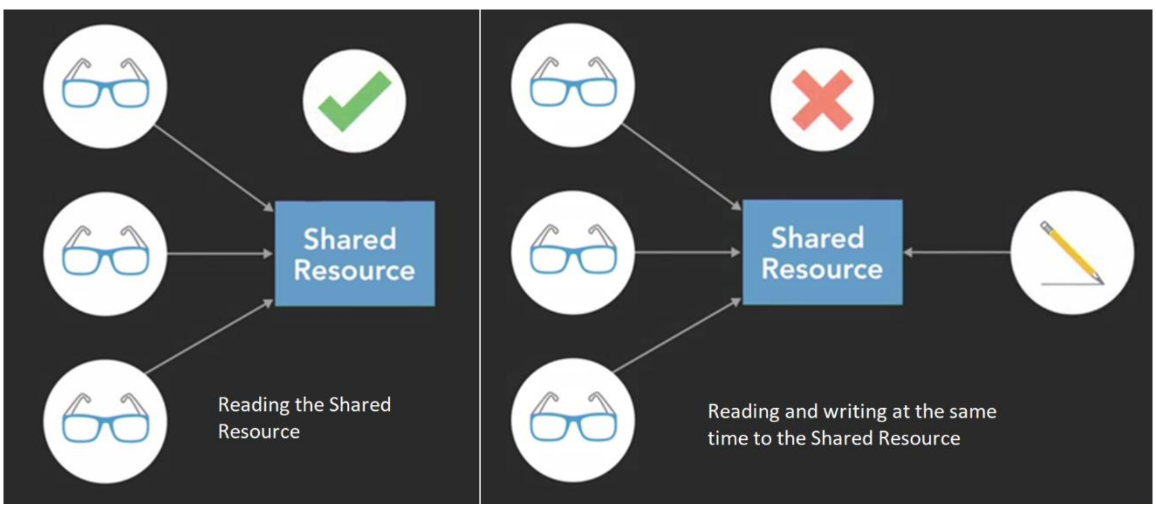
II. If the mutex is already possessed by another thread, the trial lock method will immediately return false.

- That return value of true or false lets the thread know whether or not it was successful in acquiring the lock.

```
#include <iostream>
#include <thread>       // Needed for thread
#include <mutex>        // Needed for mutex
#include <chrono>

unsigned int items_on_notepad = 0;
std::mutex pencil;

void shopper(const char* name)
{
    int items_to_add = 0;
    while (items_on_notepad <= 20)
    {
        if (items_to_add && pencil.try_lock())
        {
            //pencil.lock();
            items_on_notepad += items_to_add;
            printf("%s added %u item(s) to notepad.\n", name, items_to_add);
            items_to_add = 0;
            std::this_thread::sleep_for(std::chrono::milliseconds(300));
            pencil.unlock();
        }
        else
        {
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
            items_to_add++;
            printf("%s found something else to buy.\n", name);
        }
    }
}

int main()
{
    std::thread me(shopper, "I");
    std::thread myWife(shopper, "My Wife");
    auto start_time = std::chrono::steady_clock::now();
    me.join();
    myWife.join();
    auto elapsed_time = std::chrono::duration_cast<std::chrono::milliseconds>
        (std::chrono::steady_clock::now() - start_time).count();
    printf("Elapsed Time: %.2f seconds\n", elapsed_time / 1000.0);
    return 0;
}
```

f. The `std::shared_mutex` class

- We use a ***lock*** or `mutex` to protect a ***critical section*** of code to defend against data races, which can occur when multiple `thread`s are concurrently accessing the same location in memory and at least one of those `thread`s is writing to that location.
- That second part is key because if we have a bunch of `thread`s and none of them are writing, they're all just want to read from the same location, that's fine.
- It's okay to let multiple `thread`s read the same shared value as long as no one else can change it. They'll all safely see the same thing. Danger only exists when you add a `thread` that's writing to the mix.
- When we use a basic ***lock*** or `mutex` to protect the shared resource, we limit access so that only one of the `thread`s can use it at a time regardless of whether that `thread` is reading or writing or both.
- That works but it's not necessarily the most efficient way to do things, especially when there are lots of `thread`s that only need to read. This is where ***reader-writer locks*** can be useful.



- A ***reader-writer lock*** or ***shared*** `mutex` can be ***locked*** in one of two ways.
  I. It can be ***locked*** in a **shared read mode** that allows multiple `thread`s that only need to read simultaneously to ***lock*** it.
  II. It can be ***locked*** in an **exclusive write mode** that limits access to only one `thread` at a time, allowing that `thread` to safely write to the shared resource.

```
#include <shared_mutex> // Needed for Shared Mutex

char weekdays[7][10] = { "Sunday", "Monday", "Tuesday", "Wednesday",
                         "Thursday", "Friday", "Saturday" };
int today = 0;
std::shared_mutex marker;
void calendar_reader(const int id) {
    for (int i = 0; i < 7; i++) {
        marker.lock_shared();   // Multiple threads are reading here.
        printf("Reader-%d sees today is %s\n", id, weekdays[today]);
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        marker.unlock_shared();
    }
}
void calendar_writer(const int id) {
    for (int i = 0; i < 7; i++) {
        marker.lock();          // Only one thread will be writing.
        today = (today + 1) % 7;
        printf("Writer-%d updated date to %s\n", id, weekdays[today]);
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        marker.unlock();
    }
}
int main() {
    constexpr auto number_of_readers = 10;
    constexpr auto number_of_writers = 2;

    std::vector<std::thread> writers;
    for (unsigned int i = 0; i < number_of_writers; i++)
        writers.emplace_back(calendar_writer, i);

    std::vector<std::thread> readers;
    for (unsigned int i = 0; i < number_of_readers; i++)
        readers.emplace_back(calendar_reader, i);

    for (unsigned int i = 0; i < number_of_readers; i++)
        readers[i].join();

    for (unsigned int i = 0; i < number_of_writers; i++)
        writers[i].join();
    return 0;
}
```

- Instead of using `std::shared_mutex` directly, we can use `std::lock_guard` or `std::unique_lock` wrapper `class`es.

  I. To obtain an exclusive lock

  - We give `std::shared_mutex` as the type parameter to `std::lock_guard` or `std::unique_lock`, and an object as the constructor argument.

- This will create an object, in which the shared mutex has an exclusive lock and this means that only the thread which created that object can execute in a critical section. Other threads must wait for this thread to release the lock before they can enter a critical section.
- The thread can only acquire an exclusive lock when the mutex is unlocked. If there are other threads which have locks, either shared or exclusive locks, then this thread must wait, until all those locks have been released.

II. To acquire a shared lock,

- We create an object of the std::shared_lock class. With the std::shared_mutex as parameter and a std::shared_mutex object as constructor argument and this will create a std::shared_lock object, where the mutex has a shared lock.
- So, a thread which has a shared lock can enter a critical section. It can only acquire this shared lock if there are no threads which have exclusive locks.
- If there is a thread which has an exclusive lock, then the thread will have to wait, until that thread releases the exclusive lock.
- So, an exclusive lock will lock out every other thread.

```
std::shared_mutex marker;

void calendar_reader(const int id) {
    for (int i = 0; i < 7; i++) {
        std::shared_lock<std::shared_mutex> lock(marker);
        printf("Reader-%d sees today is %s\n", id, weekdays[today]);
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}
void calendar_writer(const int id) {
    for (int i = 0; i < 7; i++) {
        std::lock_guard<std::shared_mutex> lock(marker);
        today = (today + 1) % 7;
        printf("Writer-%d updated date to %s\n", id, weekdays[today]);
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}
```

g. Spinlocks

- A *spinlock* is a type of synchronization primitive used to protect shared resources.
- It gets its name from the way `thread`s "spin" in a loop while waiting to acquire the lock, rather than putting themselves to sleep.

How Spinlocks Work

- When a `thread` tries to acquire a spinlock:
  1. If the lock is already held by another `thread`, the `thread` continuously checks (or spins) in a loop, waiting for the lock to be released.
  2. Once the lock becomes available, the spinning `thread` will acquire it and proceed with its work.
- In other words, a `thread` holding a spinlock will continue running, which avoids the overhead of a context switch (switching between `thread`s) but uses CPU resources while it spins.

Implementation

```
class SpinLock {
public:
    void lock()
    {
        // Attempts to acquire the lock by setting isLocked to true.
        // If isLocked was already true, it means the lock is
        // held by another thread.
        while (isLocked.exchange(true, std::memory_order_acquire))
        {
            // Temporarily give up place in the CPU’s execution
            // queue to allow other threads to make progress
            // and possibly release the lock.
            std::this_thread::yield();
        }
    }

    void unlock() {
        // Sets isLocked to false, releasing the lock.
        isLocked.store(false, std::memory_order_release);
    }

private:
    std::atomic<bool> isLocked = { false };
};
```

```cpp
// Shared counter and SpinLock instance
int counter = 0;
SpinLock spinlock;

void increment(int numIterations) {
    for (int i = 0; i < numIterations; ++i) {
        spinlock.lock();  // Acquire the lock
        ++counter;        // Increment the shared counter
        spinlock.unlock(); // Release the lock
    }
}

int main() {
    int numThreads = 10;
    int numIterations = 5000;

    // Create multiple threads to increment the counter
    std::vector<std::thread> threads;
    for (int i = 0; i < numThreads; ++i) {
        threads.emplace_back(increment, numIterations);
    }

    // Join all threads
    for (auto& thread : threads) {
        thread.join();
    }

    // Output the final counter value
    std::cout << "Final counter value: " << counter << std::endl;

    return 0;
}
```

- The `std::memory_order_acquire`
    1. **Purpose**: Ensures that any read or write operations that happen before an **acquire** operation on one thread are visible to that thread after the acquire operation. This ordering is useful when **acquiring** a resource or lock to ensure that **this** thread **can see all changes made by the** thread **that released it**.
- The `std::memory_order_release`
    1. **Purpose**: Ensures that all write operations in the current thread happen before the release operation, making them visible to other threads that

perform an acquire operation on the same `atomic` variable. This ordering is useful when **releasing a lock or updating a flag** to signal to other `thread`s that changes are now visible.

- The `std::memory_order_relaxed`
  1. **Purpose**: Provides no synchronization or ordering guarantees. It allows `atomic` operations to happen without ordering constraints, meaning it only guarantees atomicity of the operation itself without impacting visibility of other variables.

Summary Table

Memory Order	Purpose	Typical Use Case
`memory_order_acquire`	Ensures visibility of prior writes from another thread before acquiring a resource.	Reading a lock or flag
`memory_order_release`	Ensures all previous writes are visible to other threads after releasing a resource.	Writing a lock or flag
`memory_order_relaxed`	Provides atomicity without visibility or ordering constraints.	Non-critical updates like counters

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

// Shared data variable
std::atomic<int> data{ 0 };
// Flag to indicate data is ready
std::atomic<bool> ready{ false };

// Producer function
void producer() {
     // Write data
     data.store(42, std::memory_order_relaxed);
     // Release memory order
     ready.store(true, std::memory_order_release);
}

// Consumer function
void consumer() {
     while (!ready.load(std::memory_order_acquire)) {
          // Busy-wait until data is ready
```

```
    }
    // When we reach this point, we know data is safe to access
    printf("Data: %d\n", data.load(std::memory_order_relaxed));
}

int main() {
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();

    return 0;
}
```

4. Liveness

a. Shared Data

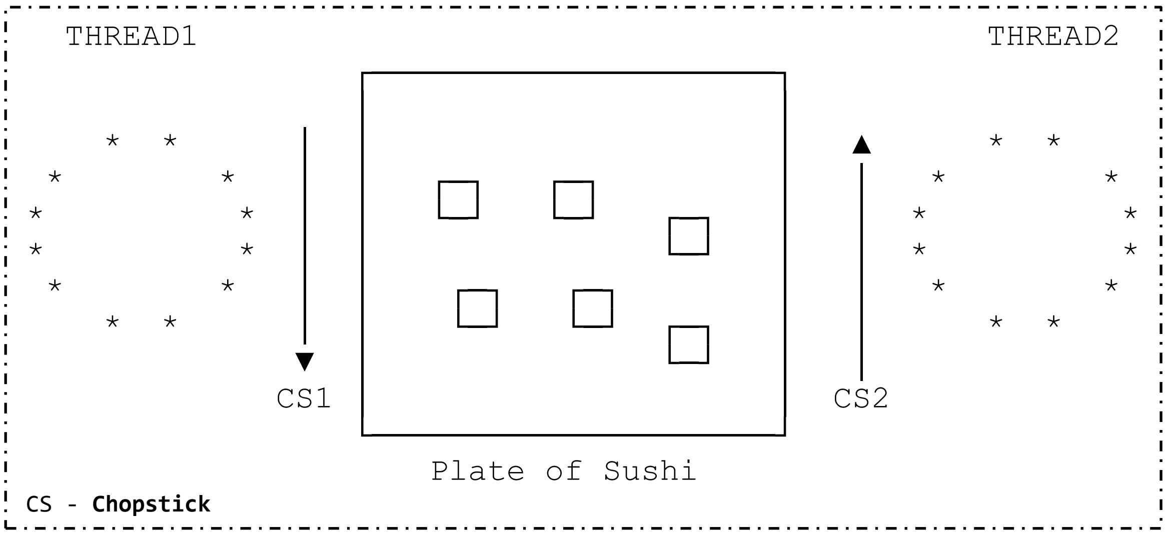
- Shared data can take a variety of forms.
    1. Global variable.
        - Which can be accessed by any code in the program.
    2. The `static` variable, at `namespace` scope.
        - Accessed by any code which can see its declaration.
    3. The `class` member, which is declared `static`.
        - If the `class` has member functions which access that `static` member, then it can be accessed by any code which calls those member functions.
        - If the `class` member has `public` accessibility, then any code which can see the `class` definition will be able to access it.
    4. Local variable, which is declared `static`.
        - That can be accessed by any code which calls the function.
- Initialization of Shared Data.
    1. Global variable
    2. The `static` variable, at `namespace` scope.
    3. The `class` member, which is declared `static`.
        - They are all initialized when the program starts up, before the `main()` function is called.
        - At that point, there is only one `thread` running, and there cannot be a data race. Because a data race requires multiple `thread`s, and we only have one `thread`.
    4. Local variable, which is declared `static`.
        - The initialization takes place when the program is running, and this will occur when the program reaches the declaration for the first time.

    - For example, if we have a function with a `static` local variable. The first time that the program reaches this declaration, it will call the constructor. In a single `thread`ed program, that is no problem.
- But what happens if we have multiple `thread`s? We could have a situation where both `thread`s try to call the constructor concurrently.

```
void function()
{
      ...
      // Static Local Variable
      static std::string str("ABC");
      ...
}
```

b. Deadlock

- A classic example that's used to illustrate synchronization issues when multiple `thread`s are competing for multiple locks is the **dining philosopher's** problem.



- In this scenario, the **THREAD1** and **THREAD2** are two philosophers doing what philosophers do best, thinking and eating. **THREAD1** and **THREAD2** both need to access a shared resource, this plate of sushi, and each time one of the `thread`s takes a piece of sushi, it is modifying its value, the number of pieces that are left.
- The act of taking sushi from the plate is a **critical section**.
- So, to protect it we will use a mutual exclusion process using these two chopsticks as `mutex`es.
- When **THREAD1** want to take a bite of sushi, it will first pick up the chopstick (`CS1`) closest to **THREAD1** to acquire a lock on it.
- Then **THREAD1** pick up the farther chopstick (`CS2`). Now it has possession of both locks. **THREAD1** in the critical section.
- So **THREAD1** take a piece of sushi and then put down the far chopstick (`CS2`) to release lock on it and then the close chopstick (`CS1`) and finally, since **THREAD1** is a philosopher, it will go back to philosophizing.
- When **THREAD2** starts running, it will acquire the chopstick closest (`CS2`) to **THREAD2** and then the one further away (`CS1`).

- **THREAD2** take a piece of sushi and then release the far chopstick (`CS1`) and then the one closer (`CS2`) to **THREAD2**.
- As dining philosophers, **THREAD1** and **THREAD2** both continue to alternate between eating and thinking.
- But since these are operating as concurrent `thread`s, neither one of the `thread`s knows when the other one wants to eat or think and that can lead to problems.
- If **THREAD1** get hungry again and pick up the chopstick closest (`CS1`) to **THREAD1**. and **THREAD2** also gets hungry and pick up the close chopstick (`CS2`), we've come to an impasse.
- **THREAD1** and **THREAD2** both acquired one of the two locks that they need. So, both stuck waiting on the other `thread` to release the other lock to make progress.
- This is one example of a situation called deadlock.
- Each member of a group is waiting for some other member to take action and as a result, neither member is able to make progress.
- Well, the deadlock occurred because **THREAD1** and **THREAD2** both reached for the chopstick which is closest it first.
- But if we prioritize these locks so that **THREAD1** and **THREAD2** both try to acquire `CS1` chopstick first then we won't have this problem because **THREAD1** and **THREAD2** both be competing for the same chopstick (`CS1`) first to lock.

Deadlock Code

```
int sushi_count = 5000;

void philosopher(std::mutex& first_chopstick, std::mutex& second_chopstick)
{
    while (sushi_count > 0)
    {
        first_chopstick.lock();
        second_chopstick.lock();
        if (sushi_count)
            sushi_count--;
        second_chopstick.unlock();
        first_chopstick.unlock();
    }
}

int main()
{
    std::mutex CS1, CS2;
    std::thread THREAD1(philosopher, std::ref(CS1), std::ref(CS2));
    std::thread THREAD2(philosopher, std::ref(CS2), std::ref(CS1));
    THREAD1.join();
    THREAD2.join();
    printf("The philosophers are done eating.\n");
    return 0;
}
```

Deadlock Code – Fixed (Using `std::scoped_lock`)

```
int sushi_count = 5000;

void philosopher(std::mutex& first_chopstick, std::mutex& second_chopstick)
{
    while (sushi_count > 0)
    {
        std::scoped_lock lock(first_chopstick, second_chopstick);
        if (sushi_count)
            sushi_count--;
    }
}

int main()
{
    std::mutex CS1, CS2;
    std::thread THREAD1(philosopher, std::ref(CS1), std::ref(CS2));
    std::thread THREAD2(philosopher, std::ref(CS1), std::ref(CS2));
    THREAD1.join();
    THREAD2.join();
    printf("The philosophers are done eating.\n");
    return 0;
}
```

c. Abandoned lock

- If one thread, or process, acquires a lock and then terminates because of some unexpected reason, it may not automatically release the lock before it disappears.
- That leaves other tasks stuck waiting for a lock that will never be released.

Abandoned lock - Example Code

```
#include <iostream>
#include <thread>        // Needed for thread
#include <mutex>

int sushi_count = 5000;

void philosopher(std::mutex& chopsticks)
{
    while (sushi_count > 0)
    {
        chopsticks.lock();
        if (sushi_count)
            sushi_count--;

        if (sushi_count == 10)
        {
            printf("This philosopher has had enough!\n");
            break;
        }
        chopsticks.unlock();
    }
}

int main()
{
    std::mutex chopsticks;
    std::thread THREAD1(philosopher, std::ref(chopsticks));
    std::thread THREAD2(philosopher, std::ref(chopsticks));
    THREAD1.join();
    THREAD2.join();
    printf("The philosophers are done eating.\n");
    return 0;
}
```

Abandoned lock - Fixed (Using `std::scoped_lock`)

```cpp
#include <iostream>
#include <thread>
#include <mutex>
int sushi_count = 5000;

void philosopher(std::mutex& chopsticks) {
    while (sushi_count > 0) {
        std::scoped_lock lock(chopsticks);
        if (sushi_count)
            sushi_count--;

        if (sushi_count == 10) {
            printf("This philosopher has had enough!\n");
            break;
        }
    }
}
int main() {
    std::mutex chopsticks;
    std::thread THREAD1(philosopher, std::ref(chopsticks));
    std::thread THREAD2(philosopher, std::ref(chopsticks));
    THREAD1.join();
    THREAD2.join();
    printf("The philosophers are done eating.\n");
    return 0;
}
```

d. Starvation

- It would be nice if **THREAD1** and **THREAD2** took turns acquiring and releasing the pair of chopsticks so they could each take an equal amount of sushi from the shared plate.
- But that's not guaranteed to happen. The operating system decides when each of `thread`s gets scheduled to execute and depending on the timing of that, it can lead to problems.
- If **THREAD2** puts down the chopsticks to release its lock on the critical section, but **THREAD1** doesn't get a chance to acquire them before **THREAD2** takes them again, then **THREAD1** be stuck waiting again until **THREAD2** takes another piece.
- If that happens occasionally, it's probably not a big deal. But if it happens regularly, Then **THREAD1** going to **starve**.

- **Starvation** occurs when a thread is unable to gain access to a necessary resource and is therefore unable to make progress. If another greedy thread is frequently holding a lock on the shared resource, then the starved thread won't get a change to execute.
- How different thread priorities get treated will depend on the operating system, but generally, higher priority threads will be scheduled to execute more often and that can leave low priority thread to starve.
- Another thing that can lead to starvation is having ***too many concurrent threads***.

Starvation – Example Code

```
int sushi_count = 5000;
void philosopher(std::mutex& chopsticks) {
    int sushi_eaten = 0;
    while (sushi_count > 0) {
        std::scoped_lock lock(chopsticks);
        if (sushi_count)  {
            sushi_count--;
            sushi_eaten++;
        }
    }
    printf("Philosopher %d ate %d.\n", std::this_thread::get_id(), sushi_eaten);
}
int main() {
    std::mutex chopsticks;
    std::array<std::thread, 200> philosophers;
    for (size_t i = 0; i < philosophers.size(); i++)
        philosophers[i] = std::thread(philosopher, std::ref(chopsticks));

    for (size_t i = 0; i < philosophers.size(); i++)
        philosophers[i].join();

    printf("The philosophers are done eating.\n");
    return 0;
}
/*
Philosopher 2380 ate 3878.
Philosopher 27852 ate 867.
Philosopher 32864 ate 255.
Philosopher 31304 ate 0.
Philosopher 37208 ate 0.
Philosopher 31248 ate 0.
Philosopher 36644 ate 0.
Philosopher 17672 ate 0.
Philosopher 14012 ate 0.
.
*/
```

How Starvation Can Occur

1. **High Contention**: With 200 philosophers (`thread`s) trying to access the same `mutex` (`chopsticks`), there can be significant contention for the lock. Only one philosopher can hold the lock at a time.
2. **Resource Allocation**: If many philosophers are continuously trying to access the `mutex`, some may end up waiting for an extended period to gain access to the critical section. If they never get a chance to lock the `mutex` because others are continuously acquiring it, they may starve.
3. **Lack of Fairness**: The current implementation does not include any mechanism to ensure fairness in accessing the shared resource. In scenarios with high contention, some `thread`s may be perpetually denied access to the `mutex` due to other `thread`s continuously acquiring it.
4. **Potential Outcome**: While some philosophers can successfully decrement `sushi_count` and eat sushi, others may be stuck waiting for the lock indefinitely if they are consistently preempted or if the scheduler prioritizes other `thread`s.

e. Livelock

- A ***livelock*** looks similar to a ***deadlock*** in the sense that two `thread`s are blocking each other from making progress. But the difference is that the `thread`s in a livelock are actively trying to resolve the problem.
- A livelock can occur when two or more `thread`s are designed to respond to the actions of each other.
- Both `thread`s are busy doing something, but the combination of their efforts prevent them from making progress and accomplishing anything useful. The program will never reach the end.
- The ironic thing about livelocks is that they're often caused by algorithms that are intended to detect and recover from deadlock.
- If one or more process or `thread` takes action to resolve the deadlock, then those `thread`s can end up being overly polite and stuck in a livelock.

Livelock – Example Code

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

int sushi_count = 5000;

void philosopher(std::mutex& first_chopstick, std::mutex& second_chopstick)
{
    while (sushi_count > 0)
    {
        first_chopstick.lock();
        printf("Thread id: %d \t, Sushi Count: %d\n", std::this_thread::get_id(),
sushi_count);
        if (!second_chopstick.try_lock())
            first_chopstick.unlock();
        else {
            if (sushi_count)
                sushi_count--;

            second_chopstick.unlock();
            first_chopstick.unlock();
        }
    }
}

int main()
{
    std::mutex chopstick_a, chopstick_b;

    std::thread THREAD1(philosopher, std::ref(chopstick_a), std::ref(chopstick_b));
    std::thread THREAD2(philosopher, std::ref(chopstick_b), std::ref(chopstick_a));
    std::thread THREAD3(philosopher, std::ref(chopstick_a), std::ref(chopstick_b));
    std::thread THREAD4(philosopher, std::ref(chopstick_b), std::ref(chopstick_a));

    THREAD1.join();
    THREAD2.join();
    THREAD3.join();
    THREAD4.join();

    printf("The philosophers are done eating.\n");
    return 0;
}
```

Livelock Explanation

1. **Philosopher Behavior**:
    - Each philosopher tries to lock the first chopstick (`mutex`).
    - If they successfully lock the first chopstick, they attempt to lock the second chopstick using `try_lock()`.

   - If they can't lock the second chopstick (because another philosopher is holding it), they unlock the first chopstick and start over. This is the key to the livelock.
2. **Endless Loop**:
   - If all philosophers keep attempting to pick up chopsticks, they may find themselves in a situation where they are continuously unlocking and re-locking the first chopstick without making any progress.
   - This leads to a livelock because the system is active (philosophers are running and trying to eat), but no philosopher can actually eat sushi.
3. **Dynamic State**:
   - In a livelock situation, the state of the system keeps changing, but none of the philosophers are making progress, which differentiates it from a deadlock, where the system is completely inactive.

Livelock – Fixed (Using `std::this_thread::yield()`)

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

int sushi_count = 5000;

void philosopher(std::mutex& first_chopstick, std::mutex& second_chopstick)
{
    while (sushi_count > 0)
    {
        first_chopstick.lock();
        if (!second_chopstick.try_lock())
        {
            first_chopstick.unlock();
            std::this_thread::yield();
        }
        else
        {
            if (sushi_count)
                sushi_count--;

            second_chopstick.unlock();
            first_chopstick.unlock();
        }
    }
}
```

```
int main()
{
    std::mutex chopstick_a, chopstick_b;

    std::thread THREAD1(philosopher, std::ref(chopstick_a), std::ref(chopstick_b));
    std::thread THREAD2(philosopher, std::ref(chopstick_b), std::ref(chopstick_a));
    std::thread THREAD3(philosopher, std::ref(chopstick_a), std::ref(chopstick_b));
    std::thread THREAD4(philosopher, std::ref(chopstick_b), std::ref(chopstick_a));

    THREAD1.join();
    THREAD2.join();
    THREAD3.join();
    THREAD4.join();

    printf("The philosophers are done eating.\n");
    return 0;
}
```

How It Works

1. **Philosopher Behavior**:
   - Each philosopher tries to lock the first chopstick (`first_chopstick`).
   - If they successfully lock the first chopstick, they then attempt to lock the second chopstick (`second_chopstick`) using `try_lock()`.
   - If they cannot lock the second chopstick (meaning another philosopher holds it), they unlock the first chopstick.
2. **Yielding Control**:
   - After unlocking the first chopstick due to the failure to lock the second chopstick, the philosopher calls `std::this_thread::yield()`. **This function gives up the remainder of the thread's time slice, allowing other threads to run.**
   - By yielding control, the current thread allows other philosophers (or threads) to attempt to acquire the chopsticks. This reduces contention and allows the system to make progress, preventing the livelock situation.
3. **Progress**:
   - Since the threads are yielding, it helps to break the cycle where each philosopher is constantly attempting to grab the chopsticks. When they yield, other philosophers have a chance to eat, thus allowing some sushi to be consumed and eventually reducing the `sushi_count`.

5. Synchronization

- A slow cooker full of hot soup is shared between two people who must take turns to ensure each gets a fair portion. In this analogy, two `thread`s compete for access to a shared resource (the soup), with the slow cooker lid acting as a `mutex`. Only the `thread` holding the lid can check the soup level, determine whose turn it is, and take a serving.
- One `thread` may waste energy repeatedly acquiring the `mutex`, just to check if it's its turn—this is called ***busy waiting*** or ***spinning***. It will keep checking until the other `thread` eventually takes its turn and the `mutex` becomes available. This inefficiency highlights a limitation of `mutex`es: while they prevent simultaneous access, they don't provide a way for `thread`s to signal each other.

```
int soup_servings = 10;
std::mutex slow_cooker_lid;

void hungry_person(int id) {
    int put_lid_back = 0;
    while (soup_servings > 0) {
        // Pick up the slow cooker lid
        std::unique_lock<std::mutex> lid_lock(slow_cooker_lid);
        // Is it your turn to take soup?
        if ((id == soup_servings % 2) && (soup_servings > 0))
            soup_servings--; // it's your turn; take some soup!
        else
            put_lid_back++; // It's not your turn; put the lid back...
    }
    printf("Person %d put the lid back %u times.\n", id, put_lid_back);
}
int main() {
    std::thread hungry_threads[2];
    for (int i = 0; i < 2; i++)
        hungry_threads[i] = std::thread(hungry_person, i);
    for (auto &ht : hungry_threads)
        ht.join();
    return 0;
}
/*
Person 0 put the lid back 150 times.
Person 1 put the lid back 1854 times.
*/
```

Busy Waiting / Spinning

- The above code demonstrates **busy waiting** (or **spinning**) because each thread repeatedly checks the soup_servings variable even when it's not their turn to take soup, holding the mutex for each check.
- **Busy Waiting / Spinning**: This refers to a thread repeatedly checking a condition in a loop, even if that condition isn't immediately met. The thread occupies CPU time while waiting, which can lead to inefficiencies, especially if it holds a lock during the check.
- Each thread locks the mutex just to check if it's their turn (`id == soup_servings % 2`) which could block the other thread, causing CPU cycles to be wasted.

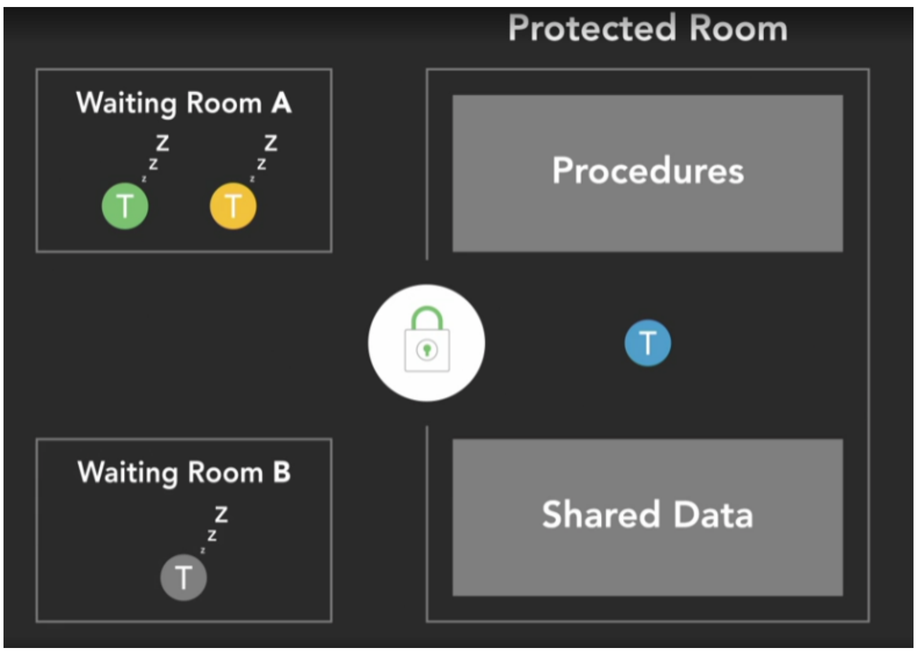
a. The std::condition_variable

- We can use a **condition variable** to coordinate thread actions more effectively.
- A condition variable **acts as a queue where threads can wait for a certain condition to occur. It **works with a** mutex to form a **monitor**, *a construct that ensures mutual exclusion and allows threads to wait until a condition is met*, then signals waiting threads when the condition changes.
- Think of a **monitor** as a **protected room with procedures and shared data**. Only one thread can enter at a time, with the mutex acting as a lock on the door. Other threads wait in a "waiting room" (the condition variable) until the condition they're waiting for is true. When a thread inside the monitor completes its task, it signals a condition and releases the lock. One waiting thread is then allowed to acquire the lock, enter the room, and proceed with the critical section.

How It Works

- **Mutex Acquired**: A thread (person) wants to enter the protected room. They acquire the mutex (lock on the door), allowing them to enter.
- **Condition Check**: Once inside, the thread checks a specific condition related to the shared data or procedures. If the condition is not met, they need to wait.

- **Condition Variable**: The `thread` waits in the corresponding waiting room (condition variable).
- **Signal**: When another `thread` inside the protected room modifies the shared data or completes a procedure that satisfies the condition, it signals the waiting room, indicating the condition has changed.
- **Mutex Released and Entry**: One `thread` from the waiting room (the waiting room associated with the signaled condition) is allowed to acquire the `mutex` (lock), enter the protected room, and proceed.



- Condition variables support three primary operations:
    - Wait
    - Signal
    - Broadcast.
- Before using a condition variable, a `thread` acquires the associated `mutex`, checks the condition, and if it's not met, it releases the `mutex` and **waits.** This allows another `thread` to acquire the lock and proceed.
- When a `thread` completes its work, it uses signal (or notify) to wake a **single** waiting `thread` or **broadcast** (or notify all) to wake all waiting `thread`s.

```cpp
int soup_servings = 10;
std::mutex slow_cooker_lid;
std::condition_variable soup_taken;
constexpr auto TWO_THREADS = 2;
constexpr auto MANY_THREADS = 5;

void hungry_person(int id) {
    int put_lid_back = 0;
    while (soup_servings > 0)
    {
        // Pick up the slow cooker lid.
        std::unique_lock<std::mutex> lid_lock(slow_cooker_lid);
        // Is it your turn to take soup?
        while ((id != soup_servings % MANY_THREADS) && (soup_servings > 0)) {
            put_lid_back++; // It's not your turn; Put the lid back...
            soup_taken.wait(lid_lock); // ...Release the lock and wait...
        }
        if (soup_servings > 0) {
            soup_servings--; // It's your turn; Take some soup!
            lid_lock.unlock(); // Put back the lid
            //soup_taken.notify_one(); // Notify another thread to take their turn
            soup_taken.notify_all(); // Notifies all other therads.
        }
    }
    printf("Person %d put the lid back %u times.\n", id, put_lid_back);
}

int main() {
    std::thread hungry_threads[MANY_THREADS];
    for (int i = 0; i < MANY_THREADS; i++) {
        hungry_threads[i] = std::thread(hungry_person, i);
    }
    for (auto& ht : hungry_threads) {
        ht.join();
    }
}
```

A mutex used to manage exclusive access to the shared soup_servings

A std::condition_variable that the threads use to communicate whose turn it is.

std::unique_lock immediately locks slow_cooker_lid upon creation. This ensures that only one thread at a time can hold the "lid"

If there are only 2 threads.

b. Condition variables with a predicate

- In the below code, there is a **`reader()`** and **`writer()`**.
- The **`writer`** `thread` is writing data to a shared data and the **`reader`** is reading from the shared data.
- We are using `condition_variable` to synchronize the between them.

```cpp
std::string sdata;
std::mutex mut;
std::condition_variable cv;
using namespace std::literals;

void reader() {
    printf("Reader thread locking the mutex.\n");
    std::unique_lock<std::mutex> lock(mut);
    std::cout << "Reader thread has locked the mutex.\n";
    std::cout << "Waiting on the condition!\n";
    cv.wait(lock);
    std::cout << "Reader wakes up as it receives notification!\n";
    std::cout << "Data is read and it: " << sdata << "\n";
}

void writer() {
    {
        printf("Writer is locking the mutex.\n");
        std::lock_guard<std::mutex> lock(mut);
        std::cout << "Writer has locked the mutex.\n";
        std::this_thread::sleep_for(2s);
        std::cout << "Modifying the shared data.\n";
        sdata = "Hello World!";
    }
    std::cout << "Notifying other thread.\n";
    cv.notify_one();
}
int main() {
    sdata = "Empty";
    std::thread r(reader);
    std::thread w(writer);
    r.join();
    w.join();
    return 0;
}
```

- Below is the output…

```
/*
Reader thread locking the mutex.
Reader thread has locked the mutex.
Waiting on the condition!
Writer is locking the mutex.
Writer has locked the mutex.
Modifying the shared data.
Notifying other thread.
Reader wakes up as it receives notification!
Data is read and it: Hello World!
*/
```

- But there is a problem with this code...
- If you change around the order in which the threads start up. For example, in `main()` you create the `writer` thread object, before the `reader` thread object.
- It can happen that the `writer` calls `notify()` before the `reader` calls `wait()` and, in that case, the notification goes out when there is nobody to receive it.
- It is like knocking on somebody's door, when they are not at home or throwing a ball, when there is nobody to catch it.
- So, the condition_variable is notified when there are no threads waiting.
- The `reader` will never be woken up, because this `wait` call will block until the condition_variable is notified, and the `reader` will probably be blocked forever. That is known as a **lost wake up**.

```
int main() {
    sdata = "Empty";

    std::thread w(writer);
    std::this_thread::sleep_for(500ms);
    std::thread r(reader);

    r.join();
    w.join();
    return 0;
}
/*
Writer is locking the mutex.
Writer has locked the mutex.
Modifying the shared data.
Notifying other thread.
Reader thread locking the mutex.
Reader thread has locked the mutex.
Waiting on the condition!
*/
```

- Fortunately, there is a way to solve this **lost wakeup** problem and also “**spurious wakeup**” problems.

The `wait()` with predicate…

- A **predicate** is function or function object that takes one or more arguments and `return`**s a Boolean** value (`true` or `false`).
- The `wait` function takes an optional second argument, which is a predicate, and, usually, this predicate will check a shared bool.
- So, the idea is that this `bool` is normally `false`, but the `writer` `thread` will set the `bool` to `true` when it sends a notification.
- So, the `reader` can use that, to check if there is a notification that is in the pipeline.
- The `reader` `thread` will call this predicate and it will only call `wait()` if the predicate `return`s `false`, so there is no notification. So, the `reader` `thread` will `wait` for one to turn up.

```
std::string sdata;
bool condition = false;
std::mutex mut;
std::condition_variable cv;
using namespace std::literals;
void reader() {
    printf("Reader thread locking the mutex.\n");
    std::unique_lock<std::mutex> lock(mut);
    std::cout << "Reader thread has locked the mutex.\n";
    std::cout << "Waiting on the condition!\n";
    cv.wait(lock, []() {
        return condition;
        });
    std::cout << "Reader wakes up as it receives notification!\n";
    std::cout << "Data is read and it: " << sdata << "\n";
}
void writer() {
    {
        printf("Writer is locking the mutex.\n");
        std::lock_guard<std::mutex> lock(mut);
        std::cout << "Writer has locked the mutex.\n";
        std::this_thread::sleep_for(100ms);
        std::cout << "Modifying the shared data.\n";
        sdata = "Hello World!";
        condition = true;
    }
    std::cout << "Notifying other thread.\n";
    cv.notify_one();
}
```

```
int main() {
    sdata = "Empty";
    std::thread w(writer);
    std::this_thread::sleep_for(500ms);
    std::thread r(reader);
    r.join();
    w.join();
    return 0;
}
/*
Writer is locking the mutex.
Writer has locked the mutex.
Modifying the shared data.
Notifying other thread.
Reader thread locking the mutex.
Reader thread has locked the mutex.
Waiting on the condition!
Reader wakes up as it receives notification!
Data is read and it: Hello World!
*/
```

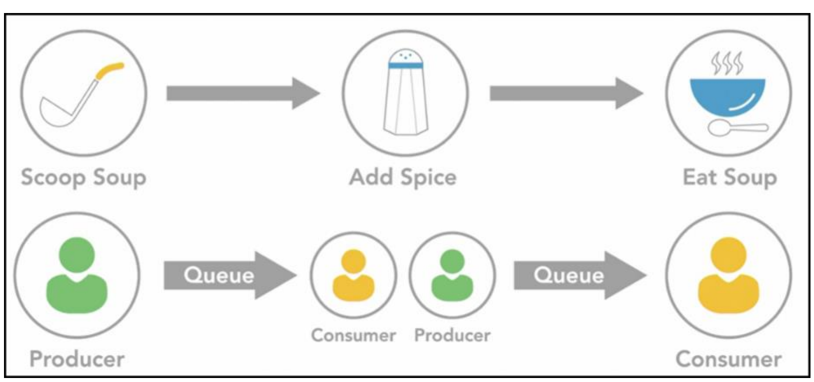
The `condition_variable::wait_for()` and `wait_until()`

- The sequence of steps in `cv.wait_for()` is as follows:
    1. Mutex Acquisition:
        - Ensure the `std::unique_lock` owns the `mutex` before calling `wait_for()`.
    2. Conditional Wait:
        - Temporarily release the `mutex`.
        - Wait for a notification or the specified timeout duration.
    3. Mutex Reacquisition:
        - Automatically re-lock the `mutex` when the `thread` is unblocked.
    4. Predicate Check:
        - If the predicate evaluates to `false`, go back to Conditional Wait (step 2).
        - If the predicate is `true`, or the timeout elapses, proceed.
    5. Timeout Handling:
        - If the timeout elapses before the predicate becomes `true`, `return std::cv_status::timeout`.

c. Producer-consumer

Use Case 1 of Condition Variable

- **Queue Operations**: A `queue` operates on a First-In-First-Out (FIFO) principle, where items are removed in the same order they're added. The producer adds bowls of soup (data items) to one end, and the consumer takes bowls from the other end.
- **Challenges in Synchronization**: In a multi-threaded producer-consumer setup, the `queue` is a shared resource that requires mutual exclusion. A `mutex` ensures only one `thread` can add or remove items at any time. Additionally, we must prevent the producer from adding items when the `queue` is full and prevent the consumer from removing items when it's empty. Some programming languages offer thread-safe `queue` implementations that handle these issues; otherwise, a `mutex` and condition variables can be used to implement a synchronized `queue`.
- **Producer and Consumer Rates**: The producer might be a steady or bursty stream of data that cannot be paused, such as streaming data. If production exceeds consumption, the `queue` may overflow, potentially causing data loss. Unbounded `queue`s can help but are ultimately limited by physical memory. Ideally, the average rate of production should be less than the consumption rate to avoid overflow.
- **Parallel Consumers**: Adding more consumer `thread`s can help balance high production rates. For instance, if two consumer `thread`s work together, they may be able to keep up with the producer.
- **Expanding to a Pipeline**: In a more complex scenario, processing may require multiple steps, like seasoning the soup before consumption. This setup can form a pipeline, where each processing step has its own consumer-producer pair, connected by `queue`s. In this case, each stage in the pipeline processes data in parallel, maintaining an optimal rate to prevent bottlenecks.



```
class ServingLine {
    std::queue<int> soup_queue;
    std::mutex ladle;
    std::condition_variable soup_served;
public:
    void serve_soup(int i) {
        std::unique_lock<std::mutex> ladle_lock(ladle);
        soup_queue.push(i);
        ladle_lock.unlock();
        soup_served.notify_one();
    }
    int take_soup() {
        std::unique_lock<std::mutex> ladle_lock(ladle);
        while (soup_queue.empty()) {
            soup_served.wait(ladle_lock);
        }
        int bowl = soup_queue.front();
        soup_queue.pop();
        return bowl;
    }
};

ServingLine serving_line = ServingLine();

void soup_producer() {
    for (int i = 0; i < 10000; i++) { // serve a 10,000 bowls of soup
        serving_line.serve_soup(1);
    }
    serving_line.serve_soup(-1); // indicate no more soup
    printf("Producer is done serving soup!\n");
}
void soup_consumer() {
    int soup_eaten = 0;
    while (true) {
        int bowl = serving_line.take_soup();
        if (bowl == -1) { // check for last bowl of soup
            printf("Consumer ate %d bowls of soup.\n", soup_eaten);
            // Put back last bowl for other consumers. When one
            // consumer encounters the sentinel value -1, it recognizes
            // that the producer has finished and stops consuming.
            serving_line.serve_soup(-1);
            return;
        }
        else {
            soup_eaten += bowl; // eat the soup
        }
    }
}
int main() {
    std::thread olivia(soup_producer);
    std::thread barron(soup_consumer);
    std::thread steve(soup_consumer);
    olivia.join();
    barron.join();
    steve.join();
}
```

Components of the Code

1. **Class ServingLine**: This `class` represents the `queue` (or serving line) of soup bowls.
   - `serve_soup(int i)`: The producer `soup_producer()` calls this function to add a bowl to the `soup_queue`. After pushing a bowl onto the `queue`, it unlocks the `ladle` `mutex` and notifies a waiting consumer `thread` via `soup_served` condition variable.
   - `take_soup()`: The consumer calls this function to take a bowl of soup from the `soup_queue`. If the `queue` is empty, the consumer waits on `soup_served` until notified by the producer. When notified, the consumer takes a bowl from the front of the `queue` and returns it.
2. **Data Members of `ServingLine`**:
   - `std::queue<int> soup_queue`: Represents the `queue` of soup bowls.
   - `std::mutex ladle`: Ensures only one `thread` accesses the `queue` at a time.
   - `std::condition_variable`: Notifies consumers when a new bowl of soup is added to the `queue`.
3. **Global `ServingLine` Object**:
   - `serving_line` is an instance of `ServingLine`, shared between the producer and consumer `thread`s.

Functions

1. `soup_producer()`:
   - This function simulates a producer that serves soup.
   - It adds `10,000` bowls to the `queue` by calling `serving_line.serve_soup(1)` in a loop.

  - After serving `10000` bowls, it adds a final bowl with `-1`, a sentinel value indicating no more soup will be served.
  - This sentinel allows consumers to detect when they should stop consuming.

2. `soup_consumer()`:
  - Each consumer continuously retrieves bowls of soup by calling `serving_line.take_soup()`.
  - When a consumer encounters a bowl with the value `-1`, it knows production has stopped:
    - It logs the total soup it ate.
    - Puts the `-1` sentinel back onto the queue for other consumers, allowing each to detect the end of production.
  - Consumers log the total bowls they consumed before stopping.

3. **Key Concurrency Features**

- **Mutex (`ladle`)**: Controls exclusive access to `soup_queue` for both the producer and consumers, ensuring data consistency.
- **Condition Variable (`soup_served`)**: Synchronizes producer-consumer communication, allowing consumers to wait until the producer adds a new bowl of soup when the `queue` is empty.

Use Case 2 of Condition Variable

```
const int MAX_QUEUE_SIZE = 5;
std::queue<int> shared_queue;
std::mutex queue_mutex;
std::condition_variable not_full, not_empty;
// Producer function: adds items to the queue
void producer(int id) {
    int item = 0;
    while (true) {
        std::unique_lock<std::mutex> lock(queue_mutex);
        // Wait if the queue is full
        not_full.wait(lock, [] { return shared_queue.size() < MAX_QUEUE_SIZE; });
        // Add item to the queue
        item++;
        shared_queue.push(item);
        std::cout << "Producer " << id << " produced item " << item << "\n";
        // Notify one consumer thread that an item is available
        not_empty.notify_one();
        lock.unlock();
        // Simulate work
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}
// Consumer function: removes items from the queue
void consumer(int id) {
    while (true) {
        std::unique_lock<std::mutex> lock(queue_mutex);
        // Wait if the queue is empty
        not_empty.wait(lock, [] { return !shared_queue.empty(); });
        // Remove item from the queue
        int item = shared_queue.front();
        shared_queue.pop();
        std::cout << "Consumer " << id << " consumed item " << item << "\n";
        // Notify one producer thread that space is available
        not_full.notify_one();
        lock.unlock();
        // Simulate work
        std::this_thread::sleep_for(std::chrono::milliseconds(150));
    }
}

int main() {
    std::thread producers[2], consumers[2];
    // Start producer threads
    for (int i = 0; i < 2; ++i)
        producers[i] = std::thread(producer, i + 1);
    // Start consumer threads
    for (int i = 0; i < 2; ++i)
        consumers[i] = std::thread(consumer, i + 1);
    // Join threads
    for (int i = 0; i < 2; ++i) {
        producers[i].join();
        consumers[i].join();
    }
    return 0;
}
```

How It Works

1. Mutex (`queue_mutex`):
    - This ensures mutual exclusion for accessing `shared_queue`. Only one `thread` can modify the `queue` at a time.
2. Condition Variables (`not_full`, `not_empty`):
    - `not_full` is used by `producer()` `thread`s to wait if the `queue` is full, preventing overflow.
    - `not_empty` is used by `consumer()` `thread`s to wait if the `queue` is empty, preventing underflow.
3. Producer Logic:
    - Each producer locks the `mutex` and checks if the `queue` has space.
    - If the `queue` is full, it waits on `not_full`.
    - When space is available, the producer adds an item and notifies a waiting consumer via `not_empty`.
4. Consumer Logic:
    - Each consumer locks the `mutex` and checks if there are items in the `queue`.
    - If the `queue` is empty, it waits on `not_empty`.
    - When items are available, the consumer removes an item and notifies a waiting producer via `not_full`.

d. The `futures` and `promises`

- The C++ `thread` `class` does not provide a direct way to transfer data from one `thread` to another. Task functions can `return` values, but those will just be ignored.
- There are ways to share data between `thread`s. We have seen,
    - Shared variable where one `thread` will update this variable, and then other `thread`s can read the new value of the variable.
- We have been using `mutex`es, which will protect the access to the shared variable.
- We have used `condition_variables` to coordinate `thread`s.
- However, there are a lot of situations where you just want to be able to send a value from one `thread` to another.

- C++ provides some `class`es for doing that.
  - The `future` `class` and the `promise` `class`.
  - We use a `future` and a `promise` `class` together, and between them they will set up a shared state between two `thread`s.
  - This is like a one-way communication channel and we can use this shared state, to transfer data from one `thread` to another.
  - These `class`es have member functions which will help in sharing data.
  - This means we do not need to have any shared data variables.
- With `future`s and `promise`s, we use what is called a "Producer-Consumer" model.
  - We have a producer `thread`, which is going to generate some result.
    - The producer `thread` has a `promise` object associated with it.
    - The constructor of the `promise` will create a `future` object, which is associated with it, and it will set up the shared state between the `promise` and the `future`.
  - We have a consumer `thread`, which is going to wait for this result.
    - The consumer `thread` has a `future` object associated with it.
    - We do not create a `future` object directly.
    - We can obtain `future` object from a `promise` object or an `async`hronous operation will return a `future`.

```
std::promise<int> prom;
std::future<int> fut = prom.get_future();
```

  - The producer `thread` will generate this result.
  - Then it will store the result in the shared state.
  - The consumer `thread` will read the result, from this shared state.
    - The consumer `thread` will call a member function of the `future` object and that member function is going to block, until the result becomes available.
- We can also use `exception`s with `future`s and `promise`s.

- The promise will store the exception in the shared state.
- The consumer thread will be woken up and the consumer thread can handle the exception, as appropriate.
- So, the exception is thrown in the producer, and caught in the consumer.

```
// The producer's task function takes a std::promise
// as argument by reference as it modifies the
// std::promise object.
void produce(std::promise<int>& px) {
   // Produce the result
   int x = 42;
   std::this_thread::sleep_for(2s);

   std::cout << "Promise sets shared state to " << x << '\n';
   // Store the result in the shared state
   px.set_value(x);
}

// The consumer's task function takes an
// std::future as argument.
void consume(std::future<int>& fx) {
   // Get the result from the shared state
   std::cout << "Future calling get()...\n";
   // This get() member function will block until
   // the producer has written the result
   // in the shared state
   int x = fx.get();
   std::cout << "The answer is " << x << '\n';
}
int main() {
   // Create an std::promise object
   // This creates an associated std::future object
   // and sets up a shared state between them
   std::promise<int> prom;
   // Get the future associated with the promise
   std::future<int> fut = prom.get_future();

   // The producer task function takes the promise as argument
   std::thread thr_producer(produce, std::ref(prom));
   // The consumer task function takes the future as argument
   std::thread thr_consumer(consume, std::ref(fut));

   thr_consumer.join();
   thr_producer.join();
}
```

- If we have some code in the producer which might throw an exception, we put a try block around it.
- Then we write a catch block, which will store this exception in the shared state, and to do that, we call the `set_exception()` member function of the promise.

```
void produce(std::promise<int>& px)
{
   try
   {
      using namespace std::literals;
      int x = 42;
      std::this_thread::sleep_for(2s);

      if (1)
      {
         throw std::out_of_range("Oops");
      }

      std::cout << "Promise sets shared state to " << x << '\n';
      px.set_value(x);
   }
   catch (...)
   {
      // Exception thrown - store it in the shared state
      px.set_exception(std::current_exception());
   }
}
```

- C++11 has a `current_exception()` function, which will return a pointer to the currently active exception.
- In the consumer thread, the `get()` function will wake up, but it will not return a value. Instead, it will throw the exception.
- We need to put a try block around that, and then a catch block to handle the exception.

```
void consume(std::future<int>& fx)
{
   std::cout << "Future calling get()...\n";
   try
   {
      // Get the result from the shared state - may throw
      int x = fx.get();
```

```
        std::cout << "Future returns from calling get()\n";
        std::cout << "The answer is " << x << '\n';
    }
    catch (std::exception& e)
    {
        // Exception thrown - get it from the shared state
        std::cout << "Exception caught: " << e.what() << '\n';
    }
}
```

A Shared Promise for the Future

- The `std::shared_future` is a `class template` that represents a `future` result that can be shared among multiple `thread`s. We are allowed to copy this, so we can give each `thread` its own shared `future` object and all these objects will share the same state with the `promise` object.

```
int main() {
    std::promise<int> prom;
    // Get an std::shared_future associated with the promise
    // This will move the promise's future into a shared future
    std::shared_future<int> shared_fut1 = prom.get_future();
    // Copy the shared future object
    std::shared_future<int> shared_fut2 = shared_fut1;

    // Start the threads
    // The producer task function takes the promise as argument
    std::thread thr_producer(produce, std::ref(prom));

    // Start two consumer threads
    // The consumer task function takes a shared future as argument
    // Each thread uses a different shared future object
    std::thread thr_consumer1(consume, std::ref(shared_fut1));
    std::thread thr_consumer2(consume, std::ref(shared_fut2));

    thr_consumer1.join();
    thr_consumer2.join();
    thr_producer.join();
}
```

e. Semaphore

- A **semaphore** is a synchronization tool that controls access to shared resources, allowing multiple `thread`s to access a resource simultaneously. Unlike a `mutex`, a semaphore includes a **counter** that tracks the number of times it has been acquired or released.

How It Works:

- **Acquiring a Semaphore**: As long as the semaphore's **counter** is **positive**, any `thread` can acquire it, decrementing the counter. If the counter reaches **zero**, further `thread`s attempting to acquire the semaphore are **blocked** and must **wait**.

- **Releasing a Semaphore**: When a `thread` finishes using the resource, it releases the semaphore, which **increments** the counter. If `thread`s are waiting, one is **notified** to wake up and acquire the semaphore.

  - Example: Consider a charger with two ports. This can be thought of as a semaphore with an initial value of 2 (indicating two available ports). As each device plugs in, it decreases the semaphore's value by 1. When both ports are occupied, the semaphore's value is zero, blocking any additional devices from charging. As soon as a device is unplugged, the semaphore is incremented, and a waiting device can plug in.

Types of Semaphores:

- **Counting Semaphore**: Can take values 0, 1, 2, etc., representing available resources (e.g., charger ports, server connections).

- **Binary Semaphore**: Limited to values 0 (locked) and 1 (unlocked), resembling a `mutex` but with a key distinction—semaphores can be incremented or decremented by different `thread`s, making them useful for signaling between `thread`s.

Semaphore Demo

- The C++17 standard library doesn't provide a built-in semaphore type. Although C++20 introduces `std::counting_semaphore` in its standard library, C++17 lacks this feature, so a custom `class` is needed.

```
class Semaphore {
public:
    Semaphore(unsigned long init_count) {
        count_ = init_count;
    }

    void acquire() { // decrement the internal counter
        std::unique_lock<std::mutex> lck(m_);
        while (!count_) {
            cv_.wait(lck);
        }
        count_--;
    }


    void release() { // increment the internal counter
        std::unique_lock<std::mutex> lck(m_);
        count_++;
        lck.unlock();
        cv_.notify_one();
    }

private:
    std::mutex m_;
    std::condition_variable cv_;
    unsigned long count_;
};
```

- The `Semaphore class` simulates a semaphore, a synchronization primitive that limits access to a shared resource. The constructor initializes the semaphore with a given number of "permits" (slots available), stored in `count_`.
- `acquire()` **Method**: Decreases the `count_` (number of permits) and blocks if no permits are available.
  - The `acquire()` function ensures exclusive access using `std::unique_lock`.
  - If `count_` is zero (no permits), the `thread` waits (`cv_.wait()`) until another `thread` releases a permit.
  - When a permit is available (`count_ > 0`), the counter is decremented, and the `thread` proceeds to use the resource.

- `release()` **Method**:
    - Increases `count_` (returns a permit) and signals waiting threads if any are blocked using `cv_.notify_one()`.
    - Unlocking the mutex (`lck.unlock()`) before notifying helps avoid potential deadlocks by allowing waiting threads to start acquiring right away.

```
Semaphore charger(4);

void cell_phone(int id) {
    charger.acquire();
    printf("Phone %d is charging...\n", id);
    srand(id); // charge for "random" amount between 1-3 seconds
    std::this_thread::sleep_for(std::chrono::milliseconds(rand() % 2000 + 1000));
    printf("Phone %d is DONE charging!\n", id);
    charger.release();
}
int main() {
    std::thread phones[10];
    for (int i = 0; i < 10; i++) {
        phones[i] = std::thread(cell_phone, i);
    }
    for (auto& p : phones) {
        p.join();
    }
}
```

- **Cell Phone Simulation** (`cell_phone` **Function)**
    - `cell_phone` simulates the behaviour of a phone attempting to charge:
    - It first "acquires" a permit to charge by calling `charger.acquire()`.
    - The phone then "charges" for a random period (between 1-3 seconds).
    - Finally, it releases the permit using `charger.release()` so another phone can charge.

Semaphore using boost

```
#include <boost/interprocess/sync/interprocess_semaphore.hpp>
#include <thread>
#include <chrono>
#include <cstdlib>
#include <iostream>

boost::interprocess::interprocess_semaphore charger(4);

void cell_phone(int id) {
    charger.wait();     // Acquire the semaphore
    printf("Phone %d is charging...\n", id);
    srand(id);          // Simulate charging time between 1-3 seconds
    std::this_thread::sleep_for(std::chrono::milliseconds(rand() % 2000 + 1000));
    printf("Phone %d is DONE charging!\n", id);
    charger.post();     // Release the semaphore
}
int main() {
    std::thread phones[10];
    for (int i = 0; i < 10; i++) {
        phones[i] = std::thread(cell_phone, i);
    }
    for (auto& p : phones) {
        p.join();
    }
    return 0;
}
```

6. Barriers

a. Race condition

- Data races and race conditions are two distinct issues in concurrent programming, though they're often confused due to their similar names. A **data race** occurs when multiple `thread`s access the same memory location simultaneously, and at least one `thread` is writing to it. This can cause `thread`s to overwrite each other's changes or read incorrect values. Data races are straightforward to detect with automated tools and are preventable by ensuring mutual exclusion on shared resources.

- A **race condition**, however, is a flaw in the timing or sequence of operations, causing incorrect program behaviour. Many race conditions are caused by data races, and vice versa, but they are not interdependent. It's possible to have data races without race conditions and race conditions without data races.

- For example, if two people calculate the number of bags of chips needed for a party, each taking turns to modify a shared shopping list, the final count will vary depending on the order in which their operations are executed. Despite using a `mutex` (e.g., a `pencil`) to prevent data races, the result could be incorrect because the order of calculations is not fixed.

$$(1 + 3) \times 2 = 8$$

$$(1 \times 2) + 3 = 5$$

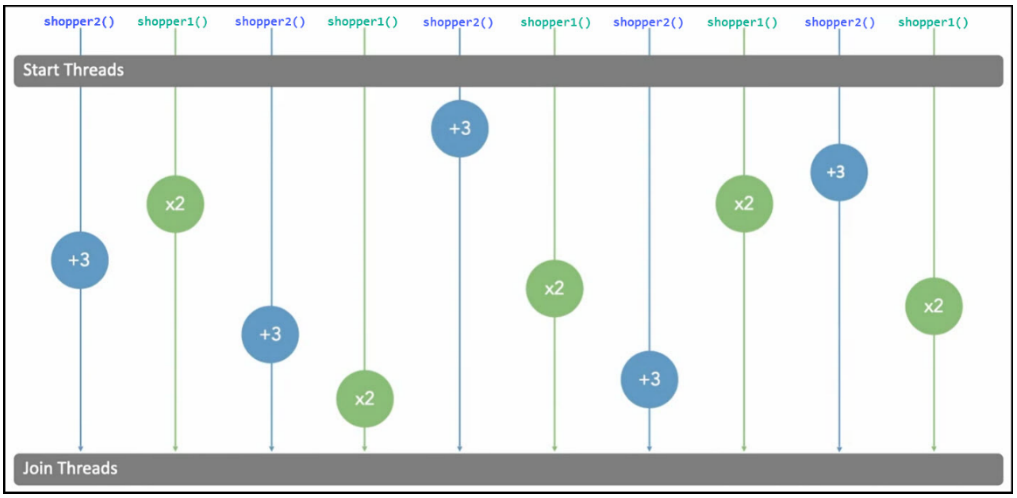
- Race conditions are challenging to detect because a program might function correctly for millions of runs, only to fail under different execution timing. Introducing sleep statements at strategic points can help reveal race conditions by altering `thread` timing. However, race conditions are often considered **heisenbugs**—bugs that change or disappear when you try to investigate them. Debugging tools that affect execution timing may prevent the race condition from appearing.

Example of race condition

```
unsigned int bags_of_chips = 1; // Initial bags on the list
std::mutex pencil;

// Simulated work
void cpu_work(unsigned long workUnits) {
    for (unsigned long i = 0; i < workUnits * 1000000; i++);
}

void shopper1() {
    cpu_work(1); // Simulate workload
    std::scoped_lock lock(pencil);
    bags_of_chips *= 2;
    //printf("Shopper1 DOUBLED the bags of chips.\n");
}

void shopper2() {
    cpu_work(1); // Simulate workload
    std::scoped_lock lock(pencil);
    bags_of_chips += 3;
    //printf("Shopper2 ADDED 3 bags of chips.\n");
}

int main() {
    std::thread shoppers[10];
    for (int i = 0; i < 10; i += 2) {
        shoppers[i] = std::thread(shopper1);
        shoppers[i + 1] = std::thread(shopper2);
    }
    for (auto& s : shoppers) s.join();
    printf("Total bags of chips needed: %u\n", bags_of_chips);
}

/*
* Run 1: Total bags of chips needed: 176
* Run 1: Total bags of chips needed: 296
*/
```

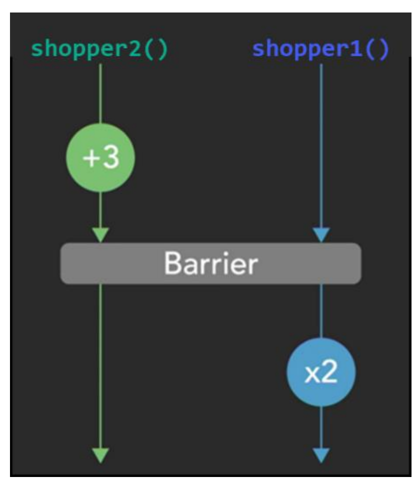
- This C++ example demonstrates a race condition in a multithreaded program. Two functions, `shopper1()` and `shopper2()`, modify the `bags_of_chips` variable, representing the total bags of chips needed for a party. The `shopper1()` function doubles the bags after some CPU work, while `shopper2()` adds three bags after similar CPU work.

- Each function locks the shared `pencil` `mutex` before modifying `bags_of_chips`, ensuring there's no data race. However, due to the variable execution order of `thread`s, the final result differs on each run, creating a **race condition**. The output varies because the order in which `thread`s are scheduled to run changes each time the code executes, even though mutual exclusion prevents simultaneous access.



b. Barrier

- To prevent a race condition, we need a way to synchronize the actions of our `thread`s, ensuring they execute operations in a specific order. A **barrier** is a synchronization point that stops all `thread`s until a specified number of them reach it, similar to players gathering in a huddle before breaking to continue the game.
- In this example, `Shopper2()` **adds three bags of chips** before the barrier, and `Shopper1()` **doubles the total afterward**. By using the barrier, the order in which `thread`s are scheduled doesn't matter, as it ensures that Olivia's addition completes before Barron's multiplication. This synchronization solves the race condition, providing a consistent result of eight bags of chips regardless of `thread` execution order.

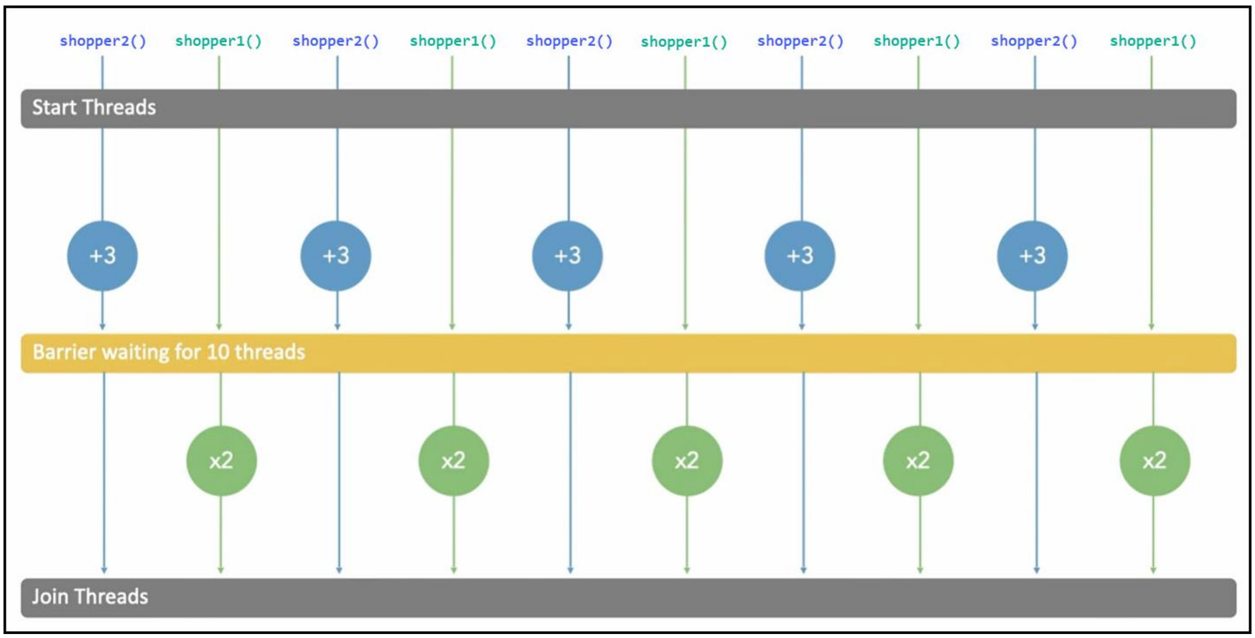


Barrier Demo

```
#include <boost/thread/barrier.hpp>

unsigned int bags_of_chips = 1; // Initial bags on the list
std::mutex pencil;
boost::barrier first_bumps(10); // Total number of threads

// Simulated work
void cpu_work(unsigned long workUnits) {
    for (unsigned long i = 0; i < workUnits * 1000000; i++);
}

void shopper1() {
    cpu_work(1); // Simulate workload
    first_bumps.wait();
    std::scoped_lock lock(pencil);
    bags_of_chips *= 2;
    printf("Shopper1 DOUBLED the bags of chips.\n");
}

void shopper2() {
    cpu_work(1); // Simulate workload
    {
        std::scoped_lock lock(pencil);
        bags_of_chips += 3;
    }
    printf("Shopper2 ADDED 3 bags of chips.\n");
    first_bumps.wait();
}

int main() {
    std::thread shoppers[10];
    for (int i = 0; i < 10; i += 2) {
        shoppers[i] = std::thread(shopper1);
        shoppers[i + 1] = std::thread(shopper2);
    }
    for (auto& s : shoppers) s.join();
    printf("Total bags of chips needed: %u\n", bags_of_chips);
}
```

- **Explanation**
- `bags_of_chips`: A shared integer that keeps track of the number of chips. It starts with an initial count of `1`.
- `pencil`: A `std::mutex` used to control access to the `bags_of_chips` variable, ensuring that only one `thread` can modify it at a time.

- `first_bumps`: A `boost::barrier` initialized to `10`, which is the total number of threads. It acts as a synchronization point that prevents `thread`s from proceeding until all `thread`s reach it.
- `shopper1()`
  - Calls `cpu_work(1)` to simulate some work.
  - Waits at the barrier (`first_bumps.wait()`), so it will pause until all `10` threads reach this point.
  - After passing the barrier, it acquires a lock on the `pencil` `mutex` to safely access `bags_of_chips`.
  - Doubles the value of `bags_of_chips` and prints a message.



- `shopper2()`
  - Calls `cpu_work(1)` for the simulated workload.
  - Enters a critical section using `std::scoped_lock` to acquire `pencil`, then adds 3 to `bags_of_chips`.
  - Prints a message indicating it has added `3` bags.
  - Waits at the barrier (`first_bumps.wait()`), ensuring it does not proceed until all `thread`s reach the barrier.

c. Latch

- Another synchronization mechanism called a **latch** allows one or more `thread`s to wait until a set of operations performed in other `thread`s is complete.
- The latch is initialized with a `count` value that `thread`s can use in two ways:
  - `Thread`s can `wait()` until the `count` reaches zero or,
  - Decrement the `count` using the `count_down()` function.

Key Differences Between Latch and Barrier:

- A **barrier** releases when a specified number of `thread`s waiting on it has been reached.
- A **latch** releases when the `count_down` function has been called enough times to reduce the `count` to `zero`.

```
#include <thread>
#include <mutex>
#include <boost/thread/latch.hpp>

unsigned int bags_of_chips = 1; // Start with one bag on the list
std::mutex pencil;

// Initialize latch with count of 5. One thread waits on this
// latch count to become zero shopper1() and another thread
// shopper2() decrements the count.
boost::latch fist_bump(5);

void cpu_work(unsigned long workUnits) {
    unsigned long x = 0;
    for (unsigned long i = 0; i < workUnits * 1000000; i++) {
        x++; // Simulate work
    }
}

void shopper1() {
    cpu_work(1);
    fist_bump.wait();   // Wait until all shopper2() finish
    std::scoped_lock<std::mutex> lock(pencil);
    bags_of_chips *= 2;
    printf("Shopper1 DOUBLED the bags of chips.\n");
}
```

```cpp
void shopper2() {
    cpu_work(1);
    {
        std::scoped_lock<std::mutex> lock(pencil);
        bags_of_chips += 3;
    }
    printf("Shopper2 ADDED 3 bags of chips.\n");
    fist_bump.count_down(); // Decrement latch count
}

int main() {
    std::thread shoppers[10];
    for (int i = 0; i < 10; i += 2) {
        shoppers[i] = std::thread(shopper1);
        shoppers[i + 1] = std::thread(shopper2);
    }
    for (auto& s : shoppers) {
        s.join(); // Wait for all threads to finish
    }
    printf("We need to buy %u bags of chips.\n", bags_of_chips);
}
```

- This program simulates two types of shoppers: `shopper1()` and `shopper2()`, who contribute to a shared resource (`bags_of_chips`). The program uses a `boost::latch` to synchronize their actions.
- The `unsigned int bags_of_chips = 1`: This variable keeps track of the total number of bags of chips. It starts with one bag.
- `std::mutex pencil`: A `mutex` that ensures exclusive access to `bags_of_chips` when it is being modified.
- `boost::latch fist_bump(5)`: A `latch` initialized with a count of 5. This is used to synchronize the two types of shoppers, where `shopper2()` will decrement the latch count.
- `void shopper1()`: This function simulates the work done by `shopper1`.
    - It waits on the `latch` using `fist_bump.wait()`, which blocks until the `latch` count reaches zero (i.e., until enough `shopper2` `thread`s have called `count_down()`).

  - Once the wait is over, it acquires a `lock` on the `pencil mutex` to ensure exclusive access to `bags_of_chips`, doubles its value, and prints a message indicating the action taken.
- `void shopper2()`: This function simulates the work done by `shopper2`.
  - It then locks the `pencil mutex`, adds `3` to `bags_of_chips`, and prints a message.
  - After updating the bag count, it decrements the `latch` count using `fist_bump.count_down()`, signaling that one `shopper2()` has finished their work.

7. Asynchronous Programming

- **Asynchronous programming** and **multithreading** are two distinct approaches for handling concurrency, each suited to specific types of problems.
- Asynchronous Programming:
  - Uses non-blocking operations to allow a program to continue execution without waiting for a task to complete.
  - Typically leverages `std::future`, `std::async`, and callbacks for handling tasks asynchronously.
  - Execution is task-based, and tasks may or may not run on separate `thread`s.
  - May run tasks on a single `thread` or a `thread` pool, depending on the implementation (e.g., `std::async` can run tasks on a new `thread` or reuse an existing one).
  - Typically does not require direct `thread` management.
  - Tasks yield control (e.g., using `std::future::get`) until they are ready, allowing the `main` `thread` to continue.
  - More lightweight as it doesn't always require creating new `thread`s.
  - Ideal for I/O-bound tasks (e.g., network requests, file I/O) where the CPU would otherwise sit idle waiting for results.
- Multithreading:
  - Involves explicitly creating and managing `thread`s of execution using `std::thread`.
  - `Thread`s can execute code in parallel, often on separate CPU cores. Requires explicit `thread` management by the programmer using `std::thread` or higher-level abstractions like `std::mutex` or `std::lock_guard`.
  - `Thread`s execute independently, and you need synchronization mechanisms to avoid data races and deadlocks.
  - Typically used for true parallelism on multi-core systems.
  - More resource-intensive since each `thread` requires its own stack and incurs overhead for context switching.

- Better suited for CPU-bound tasks where tasks can utilize multiple cores for faster computation.

a. The std::packaged_task class

- It packages up everything that we need for executing a task. It will contain all the code for the task that is stored in a callable object member.
- There is also a member which is a promise object, and that is used for storing the result of the task.
- The packaged_task is a template class.
  - The type parameter is the signature of the callable object.
  - The constructor takes the callable object as argument.

```
int Sumup(int x, int y) {
   return x + y;
}

int main() {
   std::packaged_task<int(int, int)> task(Sumup);
}
```

- The packaged_task is also a functor class. It overloads the function call operator. This will invoke the callable object. It will execute the code for the task and then it will store the return value, in the promise object member.
- The class has a get_future() member. This will return a future object, which is associated with the promise member. We can use that, to find the result of the task.
- Like most of the classes in the C++ thread library, this is a move-only class.

```
int Sumup(int x, int y) { return x + y; }

int main() {
   std::packaged_task<int(int, int)> task(Sumup);
   std::future<int> fut = task.get_future();
   task(6, 7);
   std::cout << "Result: " << fut.get() << "\n";
}
```

- The above code runs synchronously. We can also run it in a different `thread`, **asynchronously**.
- To do that, we provide the task as the first argument to the constructor of the `std::thread`.

```
int Sumup(int x, int y) {
   std::this_thread::sleep_for(std::chrono::seconds(2));
   return x + y;
}

int main() {
   std::packaged_task<int(int, int)> task(Sumup);
   std::future<int> fut = task.get_future();

   std::thread t(std::move(task), 6, 7);
   std::cout << "Waiting for result: " << fut.get() << "\n";

   t.join();
   return 0;
}
```

b. The `std::async()` function

- With `async`, we can start a task which will run in the background, and our `thread` can carry on doing other work, while that task is running on a different `thread`.
- It is also possible to run the task synchronously on the same `thread`.
- The first argument is the task function, the entry point function of the `thread`, and any arguments which follow that will be forwarded to the task function.

```
int Sumup(int x, int y) {
   std::this_thread::sleep_for(std::chrono::seconds(2));
   return x + y;
}

int main() {
   std::future<int> fut = std::async(Sumup, 6, 7);
   std::cout << "Waiting for result: " << fut.get() << "\n";
   return 0;
}
```

- Program which finds the Fibonacci of n^{th} number asynchronously.

```
unsigned long long fibonacci(unsigned long long n) {
   if (n <= 1)
      return 1;
   return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
   constexpr int fib = 42;
   std::cout << "Calling Fibobacci(" << fib << ")\n";
   auto fut = std::async(fibonacci, fib);

   while (fut.wait_for(std::chrono::seconds(1)) !=
      std::future_status::ready)
   {
      std::cout << "Waiting for the Fibonacci result!\n";
   }

   std::cout << "Fibobacci("<< fib << "): "
      << fut.get() << "\n";
   return 0;
}
```

- If the task function throws an exception, then that exception will be stored in the future object and when we call **get()**, then the future will rethrow the exception.

```
int producer()
{
   int x = 40;
   std::this_thread::sleep_for(std::chrono::seconds(2));

   if (true)
      throw std::out_of_range("Out of range");

   std::cout << "Produced value: " << x << "\n";
   return x;
}
```

```
int main() {

   auto fut = std::async(producer);

   std::cout << "Doing some operation\n";
   try
   {
      auto result = fut.get();
      std::cout << "Value from produced: " << result << "\n";
   }
   catch (std::exception& e)
   {
      std::cout << "Exception Caught: " << e.what() << "\n";
   }

   return 0;
}
```

Launch options for the `std::async()`

- The `async` function takes a callable object as argument, and runs it as a task.
- It may do that by starting a new `thread` for the task, or it may run the task in the same `thread` which called `async()`.
- There is a launch flag we can use, to control this behavior.
    - `std::launch::async`
    - `std::launch::deferred`
    - Both flags are set.
- `std::launch::async`
    - With the `std::launch::async` a new `thread` will be started up for the task.
    - So, when we call the `async` function, it `return`s immediately with a `future` object, and the task will run in the background.
- `std::launch::deferred`
    - With `std::launch::deferred` option, the task does not run until someone calls `get()` on the `return`ed `future` object.
- Both bitflags are set
    - These are bitflags. So, we can set both of them, by using the OR operator.
    - In that case, the implementation will decide which policy to use.

c. Choosing a thread object

- There are three different ways of starting a task:
  - The `std::thread` object.
  - The `std::packaged_task` object.
  - The `std::async()` function.
- Which one to use depends on particular problem.

Feature	`std::thread`	`std::packaged_task`	`std::async`
Control over execution	Full control over thread creation	Explicit scheduling of task execution	Minimal control; runtime decides
Task result handling	Manual	`std::future` integration	Built-in `std::future` support
Ease of use	Moderate	Moderate	High
Custom thread pools	Works well	Ideal for integration	Not suitable
Best for	Long-running tasks, fine-grained control	Task scheduling, integration with pools	Simple async tasks

d. Asynchronous Programming vs. Multithreading

Asynchronous Programming:

- Focuses on the **when** a task is executed (non-blocking, independent of the `main` task flow) rather than **how** it is executed (in a `thread` or not).
- The key idea is that tasks are started and can be completed later without blocking the `main` program flow.
- It is about **task** management—tasks are scheduled and executed independently of the calling code.

Multithreading:

- Focuses on **how** tasks are executed—by creating multiple `thread`s that execute in parallel.
- It is about using multiple execution contexts (`thread`s) for simultaneous work, which often requires `thread` management and synchronization.

Why `std::async` is Asynchronous Programming

- The `thread` Usage is an Implementation Detail:
  - When you use `std::async`, you are asking the runtime to execute a task asynchronously. Whether the task runs on the same `thread`, a new `thread`, or some `thread` from a `thread` pool is up to the implementation.
- For example:
  - With `std::launch::async`, a new `thread` is created.
  - With `std::launch::deferred`, the task runs synchronously in the same `thread`.
- From the perspective of the programmer, the key feature is that the calling `thread` is not blocked—the task executes independently.
- While `std::async` may create a new `thread` under the hood, it is not necessarily multithreading:
- When `std::launch::deferred` is used, no additional `thread`s are involved, but it's still asynchronous.
- Asynchronous programming focuses on the logical concurrency of tasks, whereas multithreading emphasizes the physical parallelism of `thread`s.
- The `std::async` is considered asynchronous programming because it abstracts away how tasks are executed and allows non-blocking task execution.
- Multithreading is a specific implementation detail where tasks are run on separate `thread`s.
- Asynchronous programming is a broader concept that may or may not involve `thread`s, while multithreading explicitly focuses on using multiple `thread`s for parallel execution.

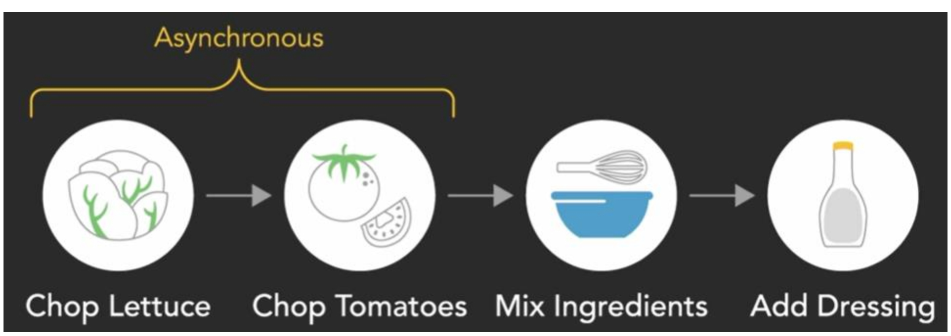
8. Asynchronous Tasks

- The key to parallel programming is identifying which steps in a program can be executed in parallel and coordinating them effectively. One tool that helps to model is computational graph.

a. Computational graph

- A computational graph helps model the relationships between these steps.
    - For example, making a simple salad involves tasks like
        - Chopping lettuce
        - Chopping tomatoes
        - Mixing the chopped ingredients
        - Adding dressing.
    - Each task can be represented as a node in a graph, with arrows indicating progression from one task to another. A task can only start executing after all its prerequisite tasks are complete.

- When arranged sequentially, these tasks represent a single path of execution that could be implemented as one `thread` or process.
- However, chopping lettuce and tomatoes can occur asynchronously, allowing them to be executed in parallel.

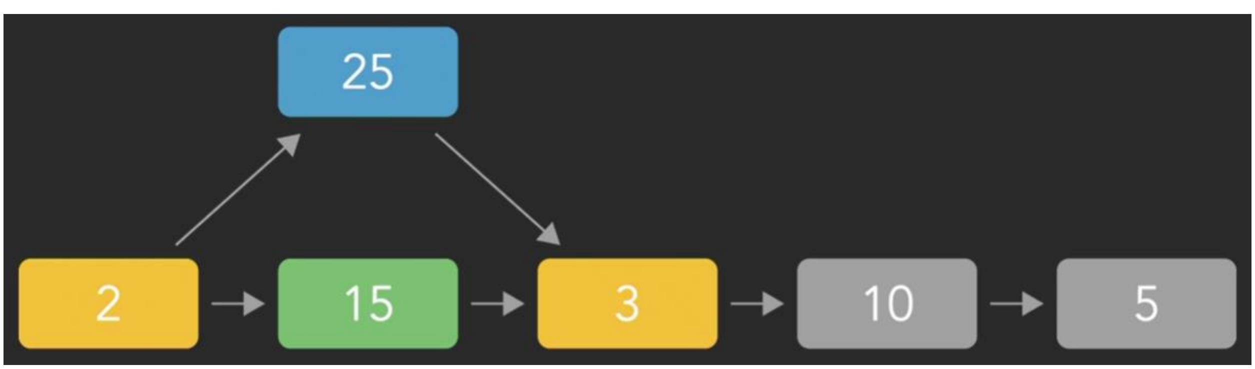


- By adding a "spawn or fork" node, we indicate that both chopping tasks can begin at any time after the spawn operation.
- There is a dependency between these chopping tasks and the mixing task, which requires a "sync or join" node to ensure it only executes once both chopping tasks are complete.

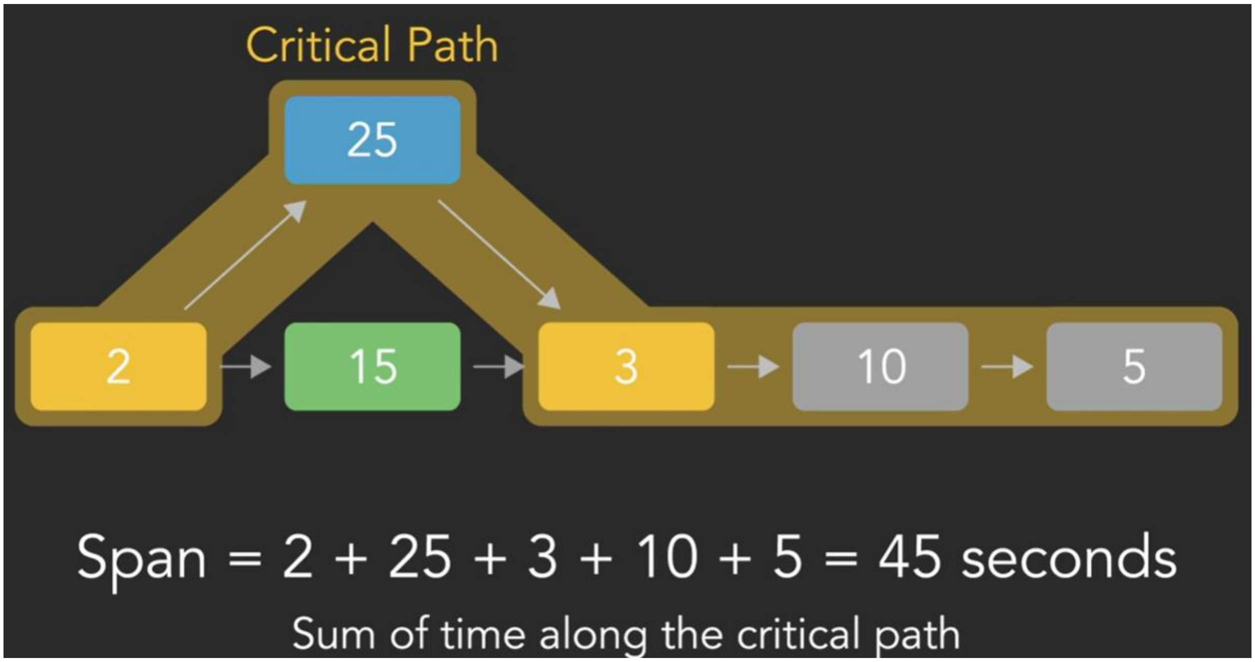
- This diagram represents a directed acyclic graph (DAG), where each edge is directed from one node to another, and there are no loops.

Purpose

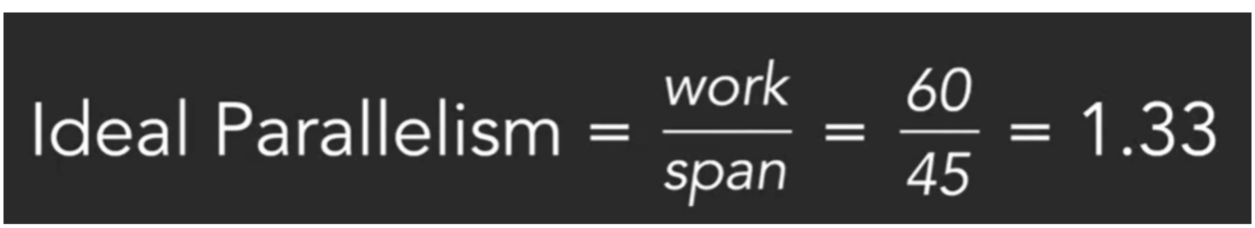
- The purpose of computational graphs is to provide an abstract representation of a program, helping to visualize task relationships and dependencies while assessing potential parallelism.
- Each node represents a task with an associated execution time. The total execution times for all nodes yield a metric called **work**, indicating the time to execute all tasks on a single processor (e.g., 60 seconds).



- The longest path through the graph, known as the **critical path**, represents the series of sequential operations. Summing the execution times along this path gives us the **span**, indicating the shortest possible execution time with full parallelization (e.g., 45 seconds).



- The ratio of work to span indicates the program's ideal parallelism—how much faster the parallel version can run compared to a sequential version on a single processor.



- In this case, a parallelism ratio of 1.33 suggests that the parallel version can be at most 33% faster than the sequential version. Although total work may not be reducible, minimizing the critical path is essential in designing efficient parallel algorithms, as the span determines the shortest execution time.

b. Thread pool

- After identifying the tasks in a program that can run asynchronously, one approach to execute them in parallel is to create independent `thread`s for each task.
- For example, preparing a salad with various chopped ingredients could involve creating separate `thread`s for each chopping task (lettuce, tomatoes, cucumbers, etc.).
- However, if there are many tasks but only a limited number of processors, creating a new `thread` for each task can lead to inefficiency due to the **overhead involved in** `thread` **creation and management**.
- Instead, a `thread` pool can be more efficient. A `thread` pool maintains a fixed set of worker `thread`s that can repeatedly take on new tasks from a `queue`.
    - As each task completes, the worker `thread` picks up the next task in line.
    - By reusing existing `thread`s, a thread pool reduces the delay associated with creating new `threads`, making it especially effective for short tasks where `thread` creation time could otherwise exceed the task execution time.

Without Thread Pool

```
#include <thread>

void vegetable_chopper(int vegetable_id) {
    printf("Thread %d chopped vegetable %d.\n",
        std::this_thread::get_id(), vegetable_id);
}

int main() {
    std::thread choppers[100];
    for (int i = 0; i < 100; i++) {
        choppers[i] = std::thread(vegetable_chopper, i);
    }
    for (auto& c : choppers) {
        c.join();
    }
}
```

Key Features of Thread Pools:

- **Efficient Resource Management**: By reusing a limited number of `thread`s, `thread` pools reduce the overhead of frequent `thread` creation and destruction.
- **Task Queuing**: New tasks are added to a `queue` instead of creating a new `thread` each time. `Thread`s from the pool pick up tasks from this `queue` as they complete their current work.
- **Concurrency Control**: The pool size limits the number of active `thread`s, which prevents excessive CPU or memory consumption.
- **Improved Responsiveness**: Since `thread`s are already created, the time needed to start a new task is reduced, improving program responsiveness.

Thread Pool Demo

```
#include <boost/asio.hpp>
#include <iostream>
#include <thread>

// Function that represents a task, in this case,
// "chopping a vegetable"
void vegetable_chopper(int vegetable_id) {
    // Prints the thread ID and vegetable ID for tracking
    printf("Thread %d chopped vegetable %d.\n",
        std::this_thread::get_id(), vegetable_id);
}

int main() {
    // Initialize a thread pool with 4 threads.
    boost::asio::thread_pool pool(4);

    // Submit 100 tasks (chop 100 vegetables) to the thread pool.
    for (int i = 0; i < 100; i++) {
        boost::asio::post(
            pool,
            // Lambda function that captures 'i' and calls
            // 'vegetable_chopper' with it. Each task represents
            // chopping a vegetable identified by 'i'.
            [i]() { vegetable_chopper(i); }
        );
    }

    // Wait for all tasks to complete before exiting on pool.
    pool.join();
}
```

- C++ does not include a native `thread` pool in the Standard Library, so Boost's `asio::thread_pool class` is used here to demonstrate how `thread` pools work.
- The `vegetable_chopper()` function simply prints the `thread` ID and a vegetable ID.
- In `main()`, we create a `thread` pool with `4` `thread`s and then **submit** `100` tasks to it using `boost::asio::post()`. This `post()` function takes
  - **Execution Context**: The first argument is an execution context, which typically specifies where the task will be executed. In the case of a `thread` **pool**, the execution context is the `thread` pool object itself (e.g., pool in this example).
    - `pool`: This specifies the `thread` pool context, so the post function knows where to schedule the task.
  - **Completion Handler** (Callable Object): The second argument is a callable object (like a lambda function, a function pointer, or a `std::function`). This callable defines the task to be executed.
    - `[i]() { vegetable_chopper(i); }`: This lambda function is the task. It captures the variable `i` by value, and when invoked, it calls `vegetable_chopper(i)`.
- The `thread` pool reuses the same `4` `thread`s to execute all tasks efficiently, reducing the overhead associated with creating and destroying individual `thread`s.

c. Future

- When launching asynchronous tasks, you can keep multiple operations running at once. For example, if one task counts the number of vegetables in the pantry, other tasks can proceed without waiting for the count. To eventually retrieve that count once it's ready, a `future` is used.
- A `future` is like a placeholder for a result that isn't available right away but will be ready eventually. It provides a way to access the outcome of an asynchronous operation. Think of it as an "IOU (**I Owe You**)" for the result.

```cpp
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

// Function to simulate counting vegetables
int count_vegetables() {
    printf("Counting vegetables...\n");
    // Simulate a delay
    std::this_thread::sleep_for(std::chrono::seconds(3));
    return 42; // Return a sample count
}

int main() {
    printf("Asking how many vegetables are in the pantry.\n");

    // Launch an asynchronous task to count vegetables
    std::future<int> result = std::async(std::launch::async,
        count_vegetables);

    // While the counting is happening, the program can
    // perform other tasks
    printf("Doing other things while waiting for result...\n");

    // Get the result of the asynchronous operation
    // (this will block until the result is ready)
    printf("The count of vegetables is: %d.\n", result.get());
    return 0;
}
```

Explanation:

- Function `count_vegetables()`: This function simulates the task of counting vegetables. It includes a sleep call to represent a time-consuming operation (like counting), and returns a fixed value (`42`) as the count of vegetables.
- Function `main()`:
    - It uses `std::async` to launch the `count_vegetables()` function asynchronously.
        - `std::async`:
            - This is a function in the C++ Standard Library that is used to launch a function asynchronously (in a separate `thread`).

- It takes at least two parameters:
  - A launch policy
  - Function to execute.
- The result is stored in a `std::future<int>` object named `result`. This allows the program to continue executing while waiting for the result of the asynchronous task.
- While the counting is happening, the program can perform other operations (represented by the message about doing other things).
- Finally, `result.get()` is called to retrieve the count of vegetables. This call will block (pause execution) until the result is ready, at which point it prints the count.

d. Divide and Conquer

- Divide and conquer algorithms are well-suited for parallel execution across multiple processors. They work by:
  I. **Dividing**: Breaking a large problem into smaller subproblems of roughly equal size.
  II. **Conquering**: Recursively solving each subproblem.
  III. **Combining**: Merging the solutions of the subproblems to produce the overall solution.
- The structure of divide and conquer code typically includes an `if`/`else` statement.
  - If the algorithm reaches a base case – a small enough problem that can be solved directly—it proceeds to solve it.
  - Else,
    - It divides the problem into two smaller parts (left and right)
    - Solve the left problem using divide-and-conquer.
    - Solve the right problem using divide-and-conquer.
    - Combine the solution to left and right problems.
- For example, consider summing a large array of shopping receipts.
  - A sequential approach involves iterating through the array and accumulating values.

- However, with a divide and conquer strategy, you can split the receipts into two halves, with each half processed by a different `thread`. This can be further subdivided into smaller groups until reaching a base case, such as a predefined number of receipts. At this point, you can sum the groups and combine the results to get the final total.

- While divide and conquer algorithms can be parallelized effectively, it's essential to consider the problem size and operation complexity. The overhead of parallelization may outweigh the benefits in some cases.

```
/**
 * Recursively sum the range of numbers between low and high.
 */
#include <cstdio>
#include <future>

 // Recursive function to sum numbers in the range [lo, hi)
unsigned long long recursive_sum(unsigned int lo,
    unsigned int hi,
    unsigned int depth = 0)
{
    // Base case: sum numbers sequentially if depth exceeds threshold
    if (depth > 3) {
        unsigned long long sum = 0;
        for (auto i = lo; i < hi; i++) {
            sum += i;  // Summing the numbers
        }
        return sum;
    }
    else { // Divide and conquer
        auto mid = (hi + lo) / 2; // Calculate the middle index
        // Asynchronously calculate the left half
        auto left = std::async(std::launch::async,
            recursive_sum, lo, mid, depth + 1);
        // Calculate the right half in the current thread
        auto right = recursive_sum(mid, hi, depth + 1);
        return left.get() + right; // Return the combined sum
    }
}

int main() {
    // Calculate the total sum from 0 to 1 billion
    unsigned long long total = recursive_sum(0, 1000000000);
    printf("Total: %llu\n", total); // Output the result
}
```

Understanding Thread Launching

- Initial Call: The main function calls `recursive_sum(0, 1000000000)`, starting the recursion with a depth of 0.
- **Base Case**: If the recursion depth exceeds 3, the function switches to sequential summation, using a single `thread` for the loop.
- **Recursive Case**: Divides the range into two halves.
  - Asynchronously launches a `thread` to calculate the sum of the left half.
  - Recursively calculates the sum of the right half in the current `thread`.
  - Waits for the asynchronous task to finish and combines the results.
- `Threa`d Launch Analysis:
  - Initial Call: One `thread` is launched for the entire range.
  - First Recursive Level: Two `thread`s are launched: one for the left half and one for the right half.
  - Second Recursive Level: Four `thread`s are launched, two for each of the previous halves.
  - Third Recursive Level: Eight `thread`s are launched.
- However, since the depth threshold is 3, the recursion stops at this level.
- Therefore, a total of `1 + 2 + 4 + 8 = 15` `thread`s is launched in this specific implementation.

9. Evaluating Parallel Performance

a. Speedup, latency, and throughput

- Using multiple processors for parallel execution can serve two main purposes:
    - Increasing the problem size that can be handled in a given time (Weak Scaling). Or
    - Completing a task faster (Strong Scaling).
- When we talk about *increasing the problem size*, we're referring to situations where more processors are added to handle a larger task, rather than finishing the same task more quickly.
- **Weak scaling** is a concept in parallel computing where you add more processors to handle a larger overall task in the same amount of time, without increasing the workload per processor.
    - For example, if one processor decorates 10 cupcakes in an hour, adding another processor doubles the output to 20 cupcakes per hour, while each still decorates 10.
- **Strong scaling** is a concept in parallel computing where parallel processors work on the same task to reduce its completion time.
    - For example, if one processor takes an hour to decorate 10 cupcakes, two processors can split the work and finish it in 30 minutes.

Throughput

- Throughput refers to the amount of work completed in a given amount of time.
- It measures how many tasks or units of work (like operations, jobs, or requests) a system can handle per unit of time, typically expressed in units like "tasks per second" or "requests per hour."

$$Throughput = \frac{(\#task)}{time}$$

- For example, imagine a task where a processor decorates cupcakes. If one processor can decorate 10 cupcakes per hour, the throughput is **10 cupcakes per hour**. Adding a second processor doubles the throughput to **20 cupcakes per hour**—the system can now complete twice as much work in the same time.

- One processor can decorate 10 cupcakes per hour.

$$Throughput\ for\ 1\ processor\ = \frac{10\ cupcakes}{1\ hour} = 10\ cupcakes\ per\ hour$$

- Two processors each work at the same rate, so together they decorate 20 cupcakes per hour.

$$Throughput\ for\ 2\ processor\ = \frac{20\ cupcakes}{1\ hour} = 20\ cupcakes\ per\ hour$$

- Three processors would work together to decorate 30 cupcakes per hour.

$$Throughput\ for\ 3\ processor\ = \frac{30\ cupcakes}{1\ hour} = 30\ cupcakes\ per\ hour$$

Latency

- Latency is the time it takes to complete a single unit of work from start to finish.
- Latency is typically measured in units of time (e.g., milliseconds, seconds) and represents the delay before a system can complete a specific operation.

$$Latency\ L = \frac{Number\ of\ Tasks\ Completed}{Total\ Time\ Taken} = \frac{task}{time}$$

- For example, if a processor takes 60 minutes to decorate 10 cupcakes, then latency L is…

$$L = \frac{60\ minutes}{10\ cupcakes} = 6\ minutes$$

Throughput vs. Latency

- Throughput is different from *latency*:
  - **Throughput** is about how much work is completed in a certain period.
  - **Latency** is the time it takes to complete a single unit of work from start to finish.

Speedup

- Speedup is a measure of how much faster a **parallel system** can perform a task compared to **a single-processor** (sequential) system.
- It's a metric used to evaluate the **effectiveness** of parallel processing.

$$Speedup\ \boldsymbol{S} = \frac{(sequential\ execution\ time)}{(parallel\ execution\ time\ with\ N\ wokrers)} = \frac{T_{sequential}}{T_{parallel}}$$

- Where…
  - $T_{sequential}$ - is the time it takes to complete the task using one processor (sequential execution).
  - $T_{parallel}$ - is the time it takes to complete the same task using multiple processors (parallel execution).
- Example: if a task takes 100 minutes with one processor, and the same task takes 25 minutes using 4 processors:

$$Speedup\ S = \frac{T_{sequential}}{T_{parallel}} = \frac{100}{25} = 4$$

Interpreting Speedup

- Ideal Speedup:
  - Ideally, if you double the number of processors, the time required to complete the task should halve, resulting in a speedup that matches the number of processors.
  - For example, with 2 processors, an ideal speedup would be $S = 2$ and with 4 processors $S = 4$, and so on.
- Superlinear Speedup (rare):
  - Sometimes, adding more processors achieves even greater than ideal speedup due to factors like caching benefits. For example, with 2 processors, a speedup of $S = 2.5$ might be observed. However, this is rare in practice.
- Sublinear Speedup (common):
  - In most real-world cases, **some parts of the task cannot be parallelized** due to dependencies or shared resources. This limits the achievable speedup.

- For example, with 4 processors, you might only achieve $S = 3$ due to overhead or sequential portions of the task.

b. Amdahl's Law

- Amdahl's Law, formulated by the computer scientist Gene Amdahl, provides an equation for **estimating the potential speedup of a program when parallelized**.
- It considers the portion of the program that can be parallelized (P) and the speedup (S) achieved by running that portion on multiple processors.

$$Overall\ Speedup = \frac{1}{(1-P)+\frac{P}{S}} \qquad Where \ldots$$

  - P – Portion of the program that is parallelizable.
  - S – Speedup of the parallelized portion.
- For example,
  - If 95% of a cupcake-decorating task can be parallelized.
  - For this 95%, using 2 processors produces a speedup of 2, then…

$$Overall\ Speedup = \frac{1}{(1-0.95)+\frac{0.95}{2}} \approx 1.9$$

  - If we add 3 processors which would result in a speedup of 3 for that 95%, then…

$$Overall\ Speedup = \frac{1}{(1-0.95)+\frac{0.95}{3}} \approx 2.7$$

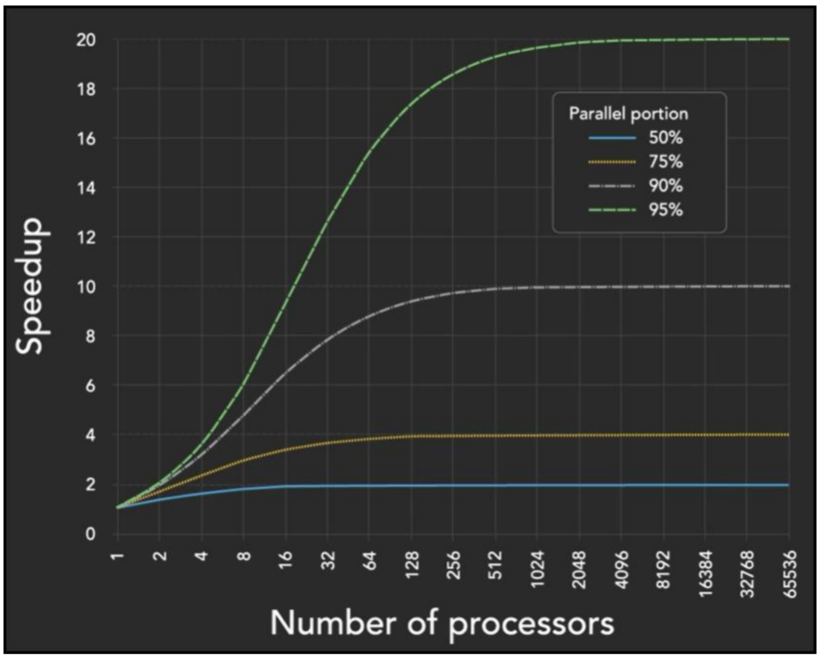
  - If we add 4 processors which would result in a speedup of 4 for that 95%, then…

$$Overall\ Speedup = \frac{1}{(1-0.95)+\frac{0.95}{4}} \approx 3.5$$

- Suppose if we add 1000 processors to speedup the 95% of the parallelized program, instinctively we would expect around overall speedup of ≈1000, but according to Amdahl's law, overall speedup will be around 19.6.

$$Overall\ Speedup = \frac{1}{(1-0.95)+\frac{0.95}{1000}} \approx 19.6$$

- The sequential portion (5%) creates an upper limit on achievable speedup, meaning that beyond a certain point, additional processors yield diminishing returns.



- Amdahl's Law demonstrates that, for programs with a small sequential portion, parallelizing is highly beneficial. However, if only 50% of a program can be parallelized, the maximum speedup, regardless of processor count, is limited to 2.
- This emphasizes that parallelization is only worthwhile for programs with a **high degree of parallelizable code**. While it may be tempting to parallelize all programs, Amdahl's Law highlights that the costs and overhead of parallelizing may sometimes outweigh the performance benefits.

c. Measure speedup

- To estimate speedup after parallelizing a program, we can measure it empirically.
- Speedup is calculated as the ratio of sequential execution time to parallel execution time, so we need two separate measurements.
  - First, measure how long the program takes with the best possible sequential implementation.
  - Then, measure the parallel implementation's execution time.
- For example, if the sequential program takes 25 seconds and the parallel version (using two processors) takes 17 seconds, the speedup is 25/17 = 1.47, indicating a 47% performance improvement.
- However, if more processors are added with only a minor increase in speedup, efficiency decreases.

Efficiency

- Efficiency in parallel computing is a measure of how effectively the available processors are used to speed up a program.
- It is calculated as the ratio of the speedup achieved by parallelizing the program to the number of processors used.

$$Efficiency = \frac{Speedup}{Number\ of\ Processors}$$ where:

  - **Speedup** is the improvement in execution time due to parallelization, calculated as $Speedup = \frac{T_{sequential}}{T_{parallel}}$
  - **Number of Processors** is the total number of processors or cores used in the parallel execution.
  - For example, if a parallel program achieves a speedup of 1.8 using 3 processors, the efficiency is:

$$Efficiency = \frac{Speedup}{Number\ of\ Processors} = \frac{1.8}{3} = 0.6 = 60\%$$

- This means that each processor is contributing only 60% of its potential toward the speedup, possibly due to overhead, waiting times, or limited parallelism in the task.

*10.*Designing Parallel Programs

- To design a parallel program, a common four-step methodology is
    - Partitioning
    - Communication
    - Agglomeration
    - Mapping.
- This method is useful for complex programs on large-scale parallel systems, though the concepts are beneficial for all parallel programming.

a. Partitioning

- Break the problem into small tasks that can be distributed to multiple processors. At this stage, we're not yet considering the available hardware. The goal is to decompose the problem into as many independent tasks as possible.
- Two main approaches for partitioning are:
    - **Domain (or Data) Decomposition**: Focuses on dividing the data into small, preferably equal partitions, and assigning computations to these partitions. For instance, if decorating cupcakes, we could split the tray so each person decorates a different half.
    - Example: Image Processing
        - Suppose we need to apply a filter to each pixel of a high-resolution image. Each pixel's processing can be handled independently, making it a good candidate for parallelization using domain decomposition.
        - **Divide the Image**: Split the image into smaller sections or blocks, for example, dividing a 1000x1000 pixel image into four 500x500 pixel sections.
        - **Assign Tasks**: Each processor (or core) is assigned one of these sections to process independently.
        - **Process Data in Parallel**: Each processor applies the filter to its assigned pixels without needing data from other sections. After all processors have finished, the results can be combined to form the final filtered image.

- In this case:
    - **Data Partition**: The image (data) is divided into smaller, manageable sections.
    - **Computation**: The filter operation is applied independently to each section, making it parallelizable.
- **Functional Decomposition**: Focuses on dividing tasks based on different functions or steps, then considering data requirements as secondary. In the cupcake example, we might separate tasks like baking, frosting, and decorating.
- Example: Web Server Request Processing
- Consider a web server that processes incoming user requests. Each request involves several distinct steps:
    - Authenticate User: Verify the user's credentials.
    - Fetch Data: Retrieve requested data from the database.
    - Process Data: Apply any transformations or computations to the data.
    - Format Response: Package the data into a suitable format (e.g., JSON or HTML).
    - Send Response: Deliver the formatted response back to the user.
- Using Functional Decomposition, each of these steps can be handled by separate, specialized tasks or threads:
    - Task 1: An authentication `thread` or function verifies the user's identity.
    - Task 2: A Database Retrieval task fetches data based on the user's request.
    - Task 3: A Processing task performs any required computations on the data.
    - Task 4: A Formatting task prepares the data for output.
    - Task 5: A Response Handler sends the formatted data back to the user.

- Using both domain and functional decomposition can provide complementary perspectives and reveal optimization opportunities. Generally, programmers start with domain decomposition, but exploring both approaches can lead to more efficient designs.

b. Communication

- After decomposing a problem into tasks, the next step is establishing communication, which involves coordinating execution and data sharing between tasks.
- Key Points on Communication in Parallel Programs:
    - **Need for Communication**: Not all tasks require communication. For independent tasks (like frosting separate cupcakes), no data exchange is needed, making them easily parallelizable. However, some tasks, like creating a rainbow pattern across cupcakes, require coordination, as each task depends on its neighbor's output.
    - **Types of Communication**:
        - Point-to-Point: Useful for tasks needing data from a small number of neighbors, where one task acts as a producer and the other as a consumer.
        - Broadcasting and Gathering: Suitable for larger groups of tasks. For instance, one task may broadcast data to multiple tasks, or scatter data pieces to different members and gather their results.
    - **Communication Growth and Scaling**: As the number of tasks increases, point-to-point communication may not scale well, potentially creating bottlenecks. Strategies like divide-and-conquer help distribute the communication load.
    - **Synchronous vs. Asynchronous**:
        - Synchronous (Blocking): Tasks wait until communication completes before continuing, which can lead to idle time.
        - Asynchronous (Non-blocking): Tasks can continue working after sending a message, regardless of when the recipient receives it, improving efficiency.

- **Factors to Consider**:
  - Processing Overhead: Communication requires processing time, which could otherwise be used for task execution.
  - Latency: The time it takes for data to travel from sender to receiver.
  - Bandwidth: The data transfer rate, often not critical for desktop applications but essential for distributed systems.
- For desktop programs, factors like latency and bandwidth are typically less impactful, but they become crucial in large distributed systems.

c. Agglomeration

- Agglomeration is the stage of design where tasks created in the initial partitioning phase are **grouped or combined to improve efficiency** on a **specific hardware setup**.
- Granularity:
  - Fine-Grained Parallelism: Splits a program into many small tasks, allowing for even load distribution across processors. However, it has high communication overhead, resulting in a lower computation-to-communication ratio.
  - Coarse-Grained Parallelism: Divides work into fewer, larger tasks, which reduces communication overhead but can lead to load imbalance, with some tasks idle while others are active.
  - Most programs benefit from medium-grained parallelism, balancing task size and communication.
- Example - Frosting Cupcakes:
  - Originally, we assigned a task for each cupcake, resulting in 12 tasks and 34 communication events—too many for only two processors.
  - By combining tasks, each processor now frosts six cupcakes, reducing communication to just two events. Although each event conveys more data, the program now matches the processor count and is more efficient.
- Scalability:
  - Avoid hard-coding the number of tasks to keep the program adaptable to different processor counts.

  - Design with flexibility to adjust task granularity using compile-time or runtime parameters, which helps the program scale efficiently with available resources.
- This agglomeration process strikes a balance between communication efficiency and computational load, optimizing the program for a specific system while maintaining flexibility for scaling.

d. Mapping

- The mapping stage is the final step in the parallel design process, where tasks are assigned to specific processors for execution.
- Key Points on Mapping:
  - **Applicability**: Mapping is relevant mainly in distributed systems or specialized hardware setups with multiple parallel processors. On a single-processor system or systems with automated scheduling, the operating system manages task allocation, making explicit mapping unnecessary.
  - Goals of Mapping:
    - **Minimizing Execution Time**: The goal of mapping is typically to reduce the program's total runtime.
    - **Concurrency vs. Locality**: Tasks that can run concurrently are mapped to different processors to maximize parallelism. Conversely, tasks that frequently communicate are often mapped to the same processor to reduce communication delays and increase locality.
  - **Strategies**:
    - **Static Mapping**: For a fixed number of tasks and predictable workloads, static mapping assigns tasks once and maintains that allocation.
    - **Dynamic Load Balancing**: When task workloads or communication patterns change during execution, dynamic load balancing adjusts the mapping periodically to maintain efficiency.
- In summary, mapping is the process of assigning tasks to processors, balancing concurrency and locality to optimize execution time. The full parallel design process

involves partitioning tasks, establishing communication, agglomerating for efficiency, and finally mapping tasks for execution based on the hardware setup.

11. Practice Problems

a. Simple Counter with Multiple Threads

- Create a shared counter that multiple `threads` increment. Start with two `threads`, each incrementing the counter `1000` times. Use a `std::mutex` to prevent data races. Extend this to see the impact of adding more `threads`.

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

mutex mut;
int counter = 0;

void incrementCounter(int iterations) {
	for (auto i = 0; i < iterations; i++) {
		//lock_guard<mutex> lock(mut);
		scoped_lock<mutex> lock(mut);
		counter++;
	}
}
int main(void) {
	int numberOfThreads = 4;
	vector<thread> threads;

	for (auto i = 0; i < numberOfThreads; i++)
		threads.emplace_back(incrementCounter, 1000);

	for (auto i = 0; i < numberOfThreads; i++)
		threads[i].join();

	printf("Counter Value: %d\n", counter);
	return 0;
}
```

- Both `std::lock_guard` and `std::scoped_lock` are used in C++ for RAII-style (Resource Acquisition Is Initialization) locking, which ensures that a `mutex` is locked when the object is created and automatically unlocked when the object goes out of scope.

- Multiple Mutexes Support:
    - `std::lock_guard` can only lock a single `mutex`.
    - `std::scoped_lock` can lock multiple `mutex`es at once, which makes it useful in situations where you need to acquire multiple locks at the same time and avoid deadlock.
- Deadlock Avoidance:
    - When locking multiple `mutex`es, `std::scoped_lock` avoids deadlocks by locking them in a consistent order internally.
    - If you use multiple `std::lock_guard`s to lock different `mutex`es, there is a potential for deadlock if another `thread` locks the `mutex`es in a different order.

b. Summing an Array

- Split a large array into chunks and use multiple `thread`s to calculate the sum of each chunk. Then combine the partial results in the main `thread` to get the total sum. This can help you practice dividing work among `thread`s and managing results.

```
#include <iostream>
#include <vector>
#include <thread>
#include <future>

using namespace std;

int calculate_sum(const vector<int>& chunk) {
	int sum = 0;
	for (int num : chunk)
		sum += num;
	return sum;
}

int main() {
	int size = 100000;
	vector<int> large_array;
	for (auto i = 0; i < size; i++)
		large_array.push_back(rand() % 100 + 1);

	int num_threads = 4;

	int chunk_size = large_array.size() / num_threads;
	vector<future<int>> futures;
```

```
    for (int i = 0; i < num_threads; ++i) {
        int start_idx = i * chunk_size;
        int end_idx = (i + 1) * chunk_size;

        if (i == num_threads - 1)
            end_idx = large_array.size();

        vector<int> chunk(large_array.begin() + start_idx,
            large_array.begin() + end_idx);
        futures.push_back(async(launch::async, calculate_sum, chunk));
    }

    int total_sum = 0;
    for (auto i : large_array)
        total_sum += i;
    printf("Serial - Total sum: %d\n", total_sum);

    total_sum = 0;
    for (auto& future : futures)
        total_sum += future.get();

    printf("Paraller - Total sum: %d\n", total_sum);

    return 0;
}
```

- Explanation:
    - The code `rand() % 100 + 1`, generates a random integer between 1 and 100.
    - `futures.push_back(async(launch::async, calculate_sum, chunk));`
        - Creates an Asynchronous Task with `async`:
        - `std::async` is used to start a new asynchronous task. This means the function `calculate_sum` will execute in a separate `thread`, allowing it to run concurrently with other `thread`s.
        - The argument `launch::async` ensures that async launches a new `thread` specifically for this task (i.e., the task is run asynchronously and not deferred until accessed). This option guarantees that the work will start immediately in a separate `thread`.
        - Calls `calculate_sum` on a Chunk of Data.

- Stores the Result in a `future<int>`:
    - `async` returns a `future<int>`, which is an object representing the result of the asynchronous operation. This future will eventually hold the result of `calculate_sum(chunk)` once the `thread` finishes execution.
- Adds the `future` to futures:
    - `futures.push_back()` adds the `future` to the `futures` vector. This way, each `future` corresponding to a chunk's sum is stored in a single collection, allowing the program to process or retrieve all results in the same loop later.

- After all chunks have been processed in parallel by different `thread`s, the program **waits** for each `thread`'s result using `future.get()` in a loop.

c. File I/O with Threads

- Write a program that reads lines from a large text file and processes each line (e.g., count words or characters) using multiple `thread`s. Each `thread` processes a separate chunk of lines. This will involve file I/O and `thread` coordination.

```
using namespace std;

struct LineProcessor {
    vector<string> lines;
    int word_count = 0;
    int char_count = 0;

    void process() {
        for (const string& line : lines) {
            // Read from the string as if it were a file.
            istringstream iss(line);
            string word;
            // Reads the next word from the stream
            // and stores it in the word variable.
            while (iss >> word) {
                word_count++;
                char_count += word.length();
            }
        }
    }
};
```

```
int main() {
    // Open the file.
    ifstream file("large_file.txt");
    // Checks if a file opened successfully
    if (!file.is_open()) {
        cerr << "Error opening file" << endl;
        return EXIT_FAILURE;
    }

    // Number of CPU cores on the machine.
    vector<LineProcessor> processors(thread::hardware_concurrency());
    // Divide lines among processors
    string line;
    int i = 0;
    while (getline(file, line)) {
        processors[i % processors.size()].lines.push_back(line);
        i++;
    }
    // Create and start threads
    vector<thread> threads;
    for (LineProcessor& processor : processors)
        threads.emplace_back(&LineProcessor::process, &processor);

    // Wait for threads to finish
    for (thread& t : threads) {
        t.join();
    }

    // Combine results from all processors
    int total_word_count = 0;
    int total_char_count = 0;
    for (const LineProcessor& processor : processors) {
        total_word_count += processor.word_count;
        total_char_count += processor.char_count;
    }

    std::cout << "Total word count: " << total_word_count << endl;
    std::cout << "Total character count: " << total_char_count << endl;

    return 0;
}
```

- Struct Definition: `LineProcessor`
    - The program defines a structure, `LineProcessor`, which contains:

        - `lines`: A `vector` to store the lines of text assigned to each `LineProcessor` instance.
        - `word_count` and `char_count`: Integers to store the word and character counts, respectively, for each `LineProcessor`.
    - It also has a `process()` method that:
        - Loops through each `line` in `lines`.
        - Uses `istringstream` to split each line into words.
        - Increments `word_count` by `1` for each word found.
        - Adds the length of each word to `char_count` to keep a running total of characters.
- Starting Threads
    - Each `LineProcessor` has a `thread` associated with it that calls its `process()` method:

```
for (LineProcessor& processor : processors)
    threads.emplace_back(&LineProcessor::process, &processor);
```

    - This line creates a new `thread` for each `LineProcessor` instance, and the `process()` method is executed by each `thread` concurrently, allowing word and character counting to happen in parallel.
    - `&LineProcessor::process`: This is a pointer to the process member function of the `LineProcessor` class. Since process is a member function, it needs an instance of `LineProcessor` to operate on (in this case, `processor`).
    - `&processor`: This is a pointer to the processor instance of `LineProcessor`. It's passed as an argument because the `thread` needs to know which instance of `LineProcessor` to call the `process` function on.
    - The `emplace_back` function here creates a new `std::thread` object that calls `processor.process()` (the `process` method on the `processor` instance) when the `thread` starts running.

d. Producer-Consumer Problem

- Implement a producer-consumer model using two `thread`s: one producing integers and placing them in a shared buffer, and another consuming them from the buffer. Use a `std::condition_variable` along with `std::mutex` to synchronize access to the buffer.

```
std::queue<int> buffer;
constexpr int buffer_size = 10;
std::mutex mut;
std::condition_variable cv;
bool done = false;

void producer() {
	for (auto i = 0; i < 100; i++) {
		std::unique_lock<std::mutex> lock(mut);
		cv.wait(lock, []() {return buffer.size() < buffer_size; });
		buffer.push(i);
		printf("Produced :%d\n", i);
		cv.notify_one();
	}

	{
		std::unique_lock<std::mutex> lock(mut);
		done = true;
	}
	cv.notify_all();
}

void consumer() {
	while (true)
	{
		std::unique_lock<std::mutex> lock(mut);
		cv.wait(lock, []() { return !buffer.empty() || done; });

		if (!buffer.empty()) {
			auto value = buffer.front();
			printf("Consumed :%d\n", value);
			buffer.pop();
			cv.notify_one();
		}
		else if (done)
			break;
	}
}
```

```
int main()
{
	std::thread t1(producer);
	std::thread t2(consumer);

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

	return 0;
}
```

- Explanation:
- Global Variables:
    - `std::queue<int>  buffer;`: This queue acts as the shared buffer for produced integers.
    - `constexpr int buffer_size = 10;`: The maximum size of the buffer.
    - `std::mutex mut;`: A `mutex` used to protect access to the shared buffer.
    - `std::condition_variable  cv;`: A condition variable used for `thread` synchronization.
    - `bool  done  =  false;`: A flag indicating when the producer is finished producing.
- Producer Function:
    - The producer function runs a loop that produces 100 integers (from 0 to 99).
    - Within each iteration, it `locks` the `mutex` using `std::unique_lock<std::mutex> lock(mut);`.
    - It then waits for the condition that there is space in the buffer (`buffer.size() < buffer_size`).
    - Once there is space, it pushes the produced integer onto the queue and prints the produced value.
    - After producing an item, it notifies one waiting consumer `thread` with `cv.notify_one()`.

- After finishing the loop, it locks the `mutex` again, sets `done = true` to signal that production is complete, and calls `cv.notify_all()` to wake up any waiting consumers.
- If there are multiple consumer threads, using `notify_one()` might leave some consumers waiting indefinitely if they do not get a chance to wake up. `notify_all()` guarantees that all consumers can make a decision about whether to continue consuming or to exit.

- Consumer Function:
  - The `consumer` function runs in an infinite loop.
  - It locks the `mutex` and waits for the condition that either the buffer is not empty or production is done (`!buffer.empty() || done;`).
  - If there are items in the buffer, it consumes the front item, prints the consumed value, and removes it from the queue. It then notifies one waiting producer.
  - If the consumer finds that the production is done (`done` is `true`) and the buffer is `empty`, it breaks the loop and ends the function.

e. Print Odd and Even Numbers

- C++ program that demonstrates how to use two threads to print odd and even numbers. One `thread` is responsible for printing even numbers, while the other prints odd numbers.

```
std::mutex mtx;                     // Mutex for synchronizing access
std::condition_variable cv;         // Condition variable for signalling
bool evenTurn = true;               // Flag to indicate whose turn it is

void printEven(int limit) {
    for (int i = 0; i <= limit; i += 2) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [] { return evenTurn; }); // Wait for the even turn
        std::cout << i << " ";                   // Print even number
        evenTurn = false;                        // Switch to odd turn
        cv.notify_one();                         // Notify the other thread
    }
}
```

```
void printOdd(int limit) {
    for (int i = 1; i <= limit; i += 2) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [] { return !evenTurn; }); // Wait for the odd turn
        std::cout << i << " ";                   // Print odd number
        evenTurn = true;                         // Switch to even turn
        cv.notify_one();                         // Notify the other thread
    }
}

int main() {
    const int limit = 20; // Define the upper limit for printing numbers

    // Create threads for printing odd and even numbers
    std::thread evenThread(printEven, limit);
    std::thread oddThread(printOdd, limit);

    // Join threads to the main thread
    evenThread.join();
    oddThread.join();

    return 0;
}
```

- Explanation:
    - Global Variables:
        - `std::mutex mtx;`: A `mutex` to protect shared data and ensure `thread`-safe operations.
        - `std::condition_variable cv;`: A condition variable to synchronize the threads.
        - `bool evenTurn`: A flag that indicates whose turn it is to print (`true` for even, `false` for odd).
    - Function `printEven`:
        - This function prints even numbers from `0` up to the specified `limit`.
        - It waits for its turn using `cv.wait(lock, [] { return evenTurn; });` to check if it's the even `thread`'s turn.
        - After printing an even number, it sets `evenTurn` to `false`, indicating that it's now the odd `thread`'s turn, and notifies the other `thread`.

- Function `printOdd`:
  - This function prints odd numbers from `1` up to the specified `limit`.
  - It waits for its turn using `cv.wait(lock, [] { return !evenTurn; });` to check if it's the odd `thread`'s turn.
  - After printing an odd number, it sets `evenTurn` to `true`, indicating that it's now the even `thread`'s turn, and notifies the other `thread`.
- Behavior of `std::condition_variable`
  - A `std::condition_variable` is used to block a `thread` until a certain condition is met, usually in synchronization scenarios. It essentially allows a `thread` to "`wait`" for an event, like a signal from another `thread`, without active polling (busy-waiting).
  - When using `std::condition_variable`, there are a few important points about its operation:
    - Wait Operation: The `wait()` function is designed to suspend the calling `thread` while atomically releasing the `mutex`. This means that during the wait period, the `mutex` must be unlocked so that other `thread`s can acquire it and possibly update the condition.
    - Reacquisition: Once the condition is satisfied (notified by another `thread`), `wait()` will reacquire the `mutex` before returning. This ensures that the waiting `thread` has exclusive access to shared data when it resumes execution.
- Requirements for the `wait()` Function in `std::`condition_variable
  - The signature for `cv.wait()` typically looks like this:

    ```
    cv.wait(lock, predicate);
    ```

    - The first argument must be a `std::unique_lock` object, which the `std::condition_variable` needs for proper operation. Specifically, `std::unique_lock` supports the following:
  - Release and reacquisition of the `mutex`, allowing `wait()` to unlock the `mutex` while the `thread` is suspended, then lock it again when the `wait` ends.

  - Locking flexibility: std::unique_lock can be locked, unlocked, and relocked as needed, which std::condition_variable leverages during its wait process.
- Why std::lock_guard or std::scoped_lock are Incompatible
  - std::lock_guard and std::scoped_lock are RAII-style locks, designed for simpler, more limited locking scenarios. They have specific traits that make them incompatible with std::condition_variable:
  - No Ability to Unlock and Re-lock
    - std::lock_guard and std::scoped_lock acquire a lock upon construction and release it only when they go out of scope.
    - Neither class provides a way to explicitly unlock and relock the mutex. This is crucial for std::condition_variable, which needs to release the lock while waiting.

f. Matrix Multiplication

- Implement matrix multiplication with multithreading. Divide the result matrix by rows and assign each thread to calculate a portion of the matrix. Each thread computes its assigned rows, storing results in a shared matrix.

```
#include <memory>
#include <thread>
#include <vector>

void printMatrix(long** mat, size_t row, size_t col)
{
	for (size_t i = 0; i < row; i++)
	{
		for (size_t j = 0; j < col; j++)
		{
			printf("%d ", mat[i][j]);
		}
		printf("\n");
	}
}

void parallel_worker(long** A, size_t num_rows_a, size_t num_cols_a,
```

```
    long** B, size_t num_rows_b, size_t num_cols_b,
    long** C, size_t start_row_c, size_t end_row_c) {
    for (size_t i = start_row_c; i < end_row_c; i++) {
        for (size_t j = 0; j < num_cols_b; j++) {
            C[i][j] = 0;
            for (size_t k = 0; k < num_cols_a; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void matrix_multiply(long** A, size_t num_rows_a, size_t num_cols_a,
    long** B, size_t num_rows_b, size_t num_cols_b,
    long** C) {
    size_t number_of_workers = std::thread::hardware_concurrency();
    printf("Concurrent Hardware: %ld\n", number_of_workers);
    size_t chunk_size = (num_rows_a + number_of_workers - 1)
        / number_of_workers;

    std::vector<std::thread> threads(number_of_workers);
    for (size_t t = 0; t < number_of_workers; t++) {
        size_t start_row_c = std::min(t * chunk_size, num_rows_a);
        size_t end_row_c = std::min((t + 1) * chunk_size, num_rows_a);

        if (start_row_c < end_row_c)
        {
            threads[t] = std::thread(parallel_worker,
                A, num_rows_a, num_cols_a,
                B, num_rows_b, num_cols_b,
                C, start_row_c, end_row_c);
        }
    }

    // Join all threads to ensure completion
    for (size_t t = 0; t < number_of_workers; t++) {
        if (threads[t].joinable()) {
            threads[t].join();
        }
    }
}

int main() {
    const size_t NUM_ROWS_A = 10;
```

```
const size_t NUM_COLS_A = 10;
const size_t NUM_ROWS_B = NUM_COLS_A;
const size_t NUM_COLS_B = 10;

long** A = (long**)malloc(NUM_ROWS_A * sizeof(long*));
if (A == nullptr)
     exit(EXIT_FAILURE);

for (size_t i = 0; i < NUM_ROWS_A; i++)
{
     A[i] = (long*)malloc(NUM_COLS_A * sizeof(long));
     if (A[i] == nullptr)
          exit(EXIT_FAILURE);

     for (size_t j = 0; j < NUM_COLS_A; j++)
          A[i][j] = rand() % 10;
}

long** B = (long**)malloc(NUM_ROWS_B * sizeof(long*));
if (B == nullptr)
     exit(EXIT_FAILURE);

for (size_t i = 0; i < NUM_ROWS_B; i++)
{
     B[i] = (long*)malloc(NUM_COLS_B * sizeof(long));
     if (B[i] == nullptr)
          exit(EXIT_FAILURE);

     for (size_t j = 0; j < NUM_COLS_B; j++)
          B[i][j] = rand() % 10;
}

long** result = (long**)malloc(NUM_ROWS_A * sizeof(long*));

if (result == nullptr)
     exit(EXIT_FAILURE);
for (size_t i = 0; i < NUM_ROWS_B; i++)
{
     result[i] = (long*)malloc(NUM_COLS_B * sizeof(long));
     if (result[i] == nullptr)
          exit(EXIT_FAILURE);
}

matrix_multiply(A, NUM_ROWS_A, NUM_COLS_A,
```

```
            B, NUM_ROWS_B, NUM_COLS_B, result);

     printMatrix(A, NUM_ROWS_A, NUM_COLS_A);
     printf("\n");
     printMatrix(B, NUM_ROWS_B, NUM_COLS_B);
     printf("\n");
     printMatrix(result, NUM_ROWS_A, NUM_COLS_B);
     printf("\n");
     return 0;
}
```

- Explanation of chunk size calculation:

```
for (size_t t = 0; t < number_of_workers; t++) {
     size_t start_row_c = std::min(t * chunk_size, num_rows_a);
     size_t end_row_c = std::min((t + 1) * chunk_size, num_rows_a);
...
```

- Suppose we have:
    - `num_rows_a = 10` (10 rows in total).
    - `number_of_workers = 3` (3 threads or workers).
    - $chunksize = \frac{(\text{num\_rows\_a} + \text{number\_of\_workers} - 1)}{\text{number\_of\_workers}} = \frac{(10 + 3 - 1)}{3} = 4$ (so each `thread` will handle a chunk of up to 4 rows).
- Thread 0 (t = 0)
    - `start_row_c = std::min(0 * 4, 10) = std::min(0, 10) = 0`
    - `end_row_c = std::min((0 + 1) * 4, 10) = std::min(4, 10) = 4`
    - Thread `0` will process rows `[0, 4)`.
- Thread 1 (t = 1)
    - `start_row_c = std::min(1 * 4, 10) = std::min(4, 10) = 4`
    - `end_row_c = std::min((1 + 1) * 4, 10) = std::min(8, 10) = 8`
    - Thread `1` will process rows `[4, 8)`.
- Thread 2 (t = 2)
    - `start_row_c = std::min(2 * 4, 10) = std::min(8, 10) = 8`
    - `end_row_c = std::min((2 + 1) * 4, 10) = std::min(12, 10) = 10`
    - Thread `2` will process rows `[8, 10)`.

g. Simple Thread Pool

- A `thread` pool is a design pattern that manages a pool of worker `thread`s to perform multiple tasks concurrently.
- Instead of creating a new `thread` for each task, the `thread` pool reuses a fixed set of `thread`s, which improves performance, especially for applications that need to handle many small tasks.
- Here's a step-by-step breakdown of how a `thread` pool works:

1. Initialization of the Thread Pool
   - When the `thread` pool is created, a specified number of worker `thread`s are also created and stored in a collection (e.g., a `vector` or `list`).
   - Each worker `thread` starts running, usually in an infinite loop where it waits for tasks to be assigned.
2. Worker Threads Waiting for Tasks
   - Worker `thread`s do not immediately perform work. Instead, they wait for tasks by blocking on a `condition variable`.
   - This waiting state prevents the `thread`s from consuming CPU resources while no tasks are available.
3. Task Enqueuing
   - When a new task is added to the pool, it is placed in a task `queue`. This task `queue` is shared among all worker `thread`s.
   - Each task is typically represented as a function or callable object that encapsulates the work to be done.
4. Notification of Worker Threads
   - After enqueuing a task, the `thread` pool signals one (or more) of the waiting worker `thread`s to start processing.
   - The `condition variable` is notified to wake up one of the `thread`s that are waiting for tasks.
5. Worker Thread Task Execution
   - When a worker `thread` is notified, it locks the `queue` and checks for tasks.

- If a task is available, the thread retrieves it from the queue, releases the lock, and then executes the task.
- Once the task is completed, the worker thread goes back to waiting for more tasks.

6. Reuse of Threads
   - After a thread finishes a task, it does not terminate. Instead, it goes back to the waiting state, ready to take on another task.
   - This reuse of threads reduces the overhead of constantly creating and destroying threads, which is beneficial for performance.
7. Graceful Shutdown
   - When the thread pool is destroyed, it signals all worker threads to stop processing.
   - This usually involves setting a flag (e.g., stop = true) and notifying all worker threads so they can exit their loops and finish execution.
   - Finally, the main thread (or destructor) joins each worker thread to ensure all threads are cleaned up before the program ends.
8. Advantages of Using a Thread Pool
   - Efficiency: Reduces the overhead of thread creation and destruction.
   - Resource Management: Limits the number of concurrent threads, preventing excessive resource usage.
   - Scalability: Allows for many tasks to be managed and executed concurrently in an organized manner.

```
#include <iostream>
#include <vector>
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>

class SimpleThreadPool
{
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> tasks;
    std::mutex queueMutex;
    std::condition_variable condition;
    bool stop;
```

```
public:
    SimpleThreadPool(size_t numThreads) : stop(false)
    {
        // Start worker threads
        for (size_t i = 0; i < numThreads; ++i)
        {
            workers.emplace_back(&SimpleThreadPool::worker, this);
        }
    }
    // Worker function that each thread will run
    void worker()
    {
        while (true)
        {
            std::function<void()> task;
            // Lock to safely access the task queue
            std::unique_lock<std::mutex> lock(queueMutex);
            // Wait for a task or a stop signal
            condition.wait(lock, [this] {
                return stop || !tasks.empty();
                });

            // If we're stopping and there are no more tasks, exit
            if (stop && tasks.empty()) return;

            // Get the next task from the queue
            task = tasks.front();
            tasks.pop();

            // Unlock the mutex before running the task
            lock.unlock();
            // Execute the task
            task();
        }
    }

    // Method to add a new task to the queue
    void enqueueTask(std::function<void()> task)
    {
        {
            // Lock to safely add a task to the queue
            std::unique_lock<std::mutex> lock(queueMutex);
            tasks.push(task);
        }
        // Notify one of the waiting threads that there is a new task
        condition.notify_one();
    }
```

```
    // Destructor to stop all threads and clean up
    ~SimpleThreadPool()
    {
        {
            // Lock to set the stop flag
            std::unique_lock<std::mutex> lock(queueMutex);
            stop = true;
        }
        // Wake up all threads to let them finish
        condition.notify_all();
        // Join all threads to ensure they complete before destruction
        for (std::thread& worker : workers)
        {
            worker.join();
        }
    }
};

int main()
{
    // Create a thread pool with 2 threads
    SimpleThreadPool pool(2);

    // Add tasks to the pool
    pool.enqueueTask(
        []()
        {
            std::cout << "Task 1 running." << std::endl;
        }
    );
    pool.enqueueTask(
        []()
        {
            std::cout << "Task 2 running." << std::endl;
        }
    );

    std::this_thread::sleep_for(std::chrono::seconds(1));
    return 0;
}
```

- The `worker()` Method:
    - We've created a member function `worker` that contains the code which each `thread` will execute.

- Thread Creation:
    - To `workers.emplace_back()`, we now pass `&SimpleThreadPool::worker` (which is the address of the `worker` member function) along with this to call the member function.
- Thread Execution:
    - Each `worker` `thread` will run the `worker` method, which contains the loop for waiting for and executing tasks.

*12.*Summary

Threads and Processes Introduction

- `Thread`s are lightweight units of a process; processes can have multiple `thread`s.
- `Thread`s within a process share memory, while processes have separate memory spaces.

Concurrent vs Parallel Execution

- **Concurrent execution**: Multiple tasks handled by the CPU seemingly simultaneously.
- **Parallel execution**: Multiple tasks executed truly simultaneously across multiple CPU cores.

Execution Scheduling

- OS schedules `thread`s based on priority, round-robin, or other algorithms.
- OS time-slices the CPU for each `thread` to manage fair CPU usage.

Thread Life Cycle (New, Runnable, Blocked, Terminated)

1. **New**: `Thread` created but not yet started.
2. **Runnable**: `Thread` ready to run, waiting for CPU time.
3. **Blocked**: `Thread` waiting for a resource (e.g., I/O).
4. **Terminated**: `Thread` execution completed.

Detached Thread (Using `t.detach()`)

1. Include `<thread>` header.
2. Declare a `thread`: `std::thread t(function);`
3. Call `t.detach();` to run the `thread` independently.

4. Parent `thread` doesn't need to wait for the detached `thread`.

Mutual Exclusion (Mutex)

1. Include `<mutex>` header.
2. Declare a `std::mutex` object.
3. Use `mutex.lock()` to acquire lock, ensuring exclusive access.
4. After accessing the critical section, use `mutex.unlock()` to release.

Atomic Objects (`std`::atomic<>)

1. Include `<atomic>` header.
2. Use `std::atomic<T>` for variables where `T` is a data type like `int`.
3. Use atomic operations (e.g., increment or compare-and-swap) directly on `std::atomic` variables.

Recursive Mutex

1. Include `<mutex>`.
2. Declare a `std::recursive_mutex` if reentrant locking is needed (i.e., allowing the same `thread` to lock multiple times).
3. Lock and unlock as needed within the same `thread`.

Reentrant Mutex (`std::recursive_mutex`)

1. Use `std::recursive_mutex` for functions calling themselves (recursively) that need locking.
2. Call `.lock()` and `.unlock()` multiple times safely within the same `thread`.

Try Lock (`mutex.try_lock()`)

1. Include `<mutex>`.
2. Declare a `std::mutex`.
3. Use `mutex.try_lock()` to attempt locking without blocking; it returns `true` if lock is successful, `false` otherwise.

Shared Mutex (`std::shared_mutex` for Reader-Writer Lock)

1. Include `<shared_mutex>`.
2. Declare a `std::shared_mutex`.
3. For reading: Use `mutex.lock_shared()` and `mutex.unlock_shared()`.
4. For writing: Use `mutex.lock()` and `mutex.unlock()`.

Liveness

Deadlock (`std::scoped_lock`)

1. Use multiple `mutex`es with `std::scoped_lock` for deadlock-free locking.
2. Include `<mutex>`.
3. Use `std::scoped_lock` to acquire all locks at once to avoid deadlocks.

Abandoned Lock (`std::scoped_lock`)

1. Avoid scenarios where a `thread` abandons a lock without releasing it.
2. Ensure `std::scoped_lock` destructors release locks on exit.

Starvation

1. Ensure fairness in resource allocation.
2. Avoid `thread` priority that indefinitely blocks other `thread`s.

Livelock (`std::this_thread::yield()`)

1. Use `std::this_thread::yield()` to avoid livelock by releasing the CPU.
2. Re-attempt the operation until the resource becomes available.

Synchronization

Condition Variable

1. Include `<condition_variable>`.
2. Declare `std::condition_variable` and `std::mutex`.
3. Use `condition_var.wait(lock)` to wait for a signal.
4. Use `condition_var.notify_one()` or `condition_var.notify_all()` to wake `thread`s.

Producer-Consumer

1. Set up a `queue` (e.g., `std::queue`).
2. Producer `thread` adds items to the `queue` and signals a consumer with `condition_var.notify_one()`.
3. Consumer waits with `condition_var.wait()` and processes items when available.

Semaphore

1. Include `<semaphore>` (C++20).
2. Declare a `std::counting_semaphore` with an initial count.
3. Use `.acquire()` to decrement and wait.
4. Use `.release()` to increment and signal availability.

Thread Pools:

- Efficiently manage `thread` creation and destruction.
- Reuse `thread`s for multiple tasks to reduce overhead.

Future and Promise:

- Asynchronous programming paradigm.
- Allows for concurrent execution and retrieval of results.

Parallel Algorithms:

- Utilize multiple cores for faster execution of algorithms.
- Examples: Parallel sorting, parallel numerical computations.

Thread-Local Storage:

- Store `thread`-specific data, isolating it from other `thread`s.
- Useful for maintaining `thread`-specific state.

Advanced Synchronization Techniques:

Barriers:

- Synchronize multiple `thread`s at a specific point.
- Useful for parallel algorithms where `thread`s need to wait for each other.

Spinlocks:

- Low-level synchronization primitive.
- Continuously check a lock until it becomes available.
- Best suited for short critical sections.

Best Practices and Considerations:

Thread Safety:

- Design data structures and algorithms to be `thread`-safe.
- Use appropriate synchronization mechanisms to protect shared resources.

Performance Optimization:

- Profile your code to identify bottlenecks.
- Minimize synchronization overhead.

- Consider using parallel algorithms and `thread` pools.

Error Handling:

- Handle exceptions and errors gracefully in multithreaded environments.
- Use appropriate error handling mechanisms.

Debugging:

- Utilize debugging tools to identify and fix issues in multithreaded code.
- Consider using `thread`-specific logging and tracing.

Real-world Applications:

Web Servers:

- Handle multiple client requests concurrently.

Game Engines:

- Render graphics, process physics, and handle input simultaneously.

Database Systems:

- Execute multiple queries and transactions concurrently.

Scientific Simulations:

- Perform complex calculations in parallel.