Contents

[*1.* Threads and Processes 2](#_Toc78223388)

[*a.* Introduction 2](#_Toc78223389)

[*b.* Concurrent Vs Parallel Execution 4](#_Toc78223390)

[*c.* Execution scheduling 5](#_Toc78223391)

[*d.* Thread life cycle 8](#_Toc78223392)

[*e.* Detached thread 10](#_Toc78223393)

[*2.* Mutual Exclusion 14](#_Toc78223394)

[*a.* Data race 14](#_Toc78223395)

[*b.* Mutual exclusion 15](#_Toc78223396)

[*c.* Atomic objects 18](#_Toc78223397)

[*3.* Locks 19](#_Toc78223398)

[*a.* Recursive mutex 19](#_Toc78223399)

[*b.* Reentrant mutex 19](#_Toc78223400)

[*c.* Try lock 20](#_Toc78223401)

[*d.* Shared mutex 22](#_Toc78223402)

[*4.* Liveness 24](#_Toc78223403)

[*a.* Deadlock 24](#_Toc78223404)

[*b.* Abandoned lock 27](#_Toc78223405)

[*c.* Starvation 28](#_Toc78223406)

[*d.* Livelock 30](#_Toc78223407)

## Threads and Processes

### 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.
* Threads can only exist as part of a process.
* Threads 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.

#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;

}

### Concurrent Vs Parallel Execution

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

Example:

┌─────────────────────────────┐ Concurrent Tasks

│ │ ┌────────────────┐ ┌───────────────┐

│ Salad Recipe │ │ │ │ │

│ │ │ Chop Lettuce │ │ Chop Tomatoes │

│ 1. Chop Lettuce. │ │ │ │ │

│ 2. Chop Cucumbers. │ └────────────────┘ └───────────────┘

│ 3. Chop Tomatoes. ├────────►

│ 4. Chop Onions. │ ┌────────────────┐ ┌───────────────┐

│ 5. Mix Chopped Vegetables. │ │ │ │ │

│ 6. Add Dressing. │ │ Chop Cucumbers │ │ Chop Onions │

│ │ │ │ │ │

└─────────────────────────────┘ └────────────────┘ └───────────────┘

Order independent

Parallel Execution: Simultaneous Execution, this requires parallel hardware.

┌────────────────────────────────────────────────────┐

│ │

│ │ │

│ Concurrency │ Parallelism │

│ ───────────── │ ───────────── │

│ Program Structure │ Simultaneous Execution │

│ │ │

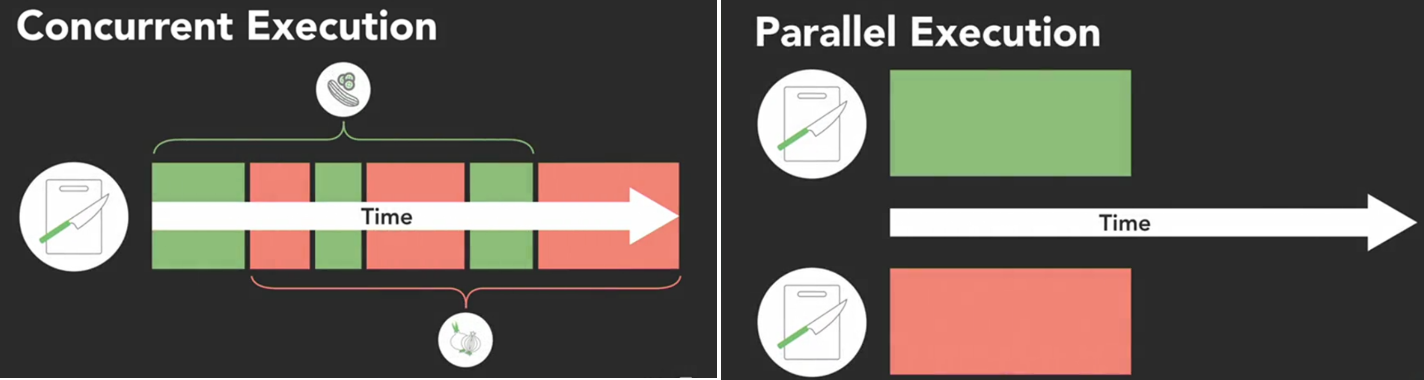
│ Dealing with multiple │ Doing multiple thigs at │

│ things at once │ Once │

│ │ │

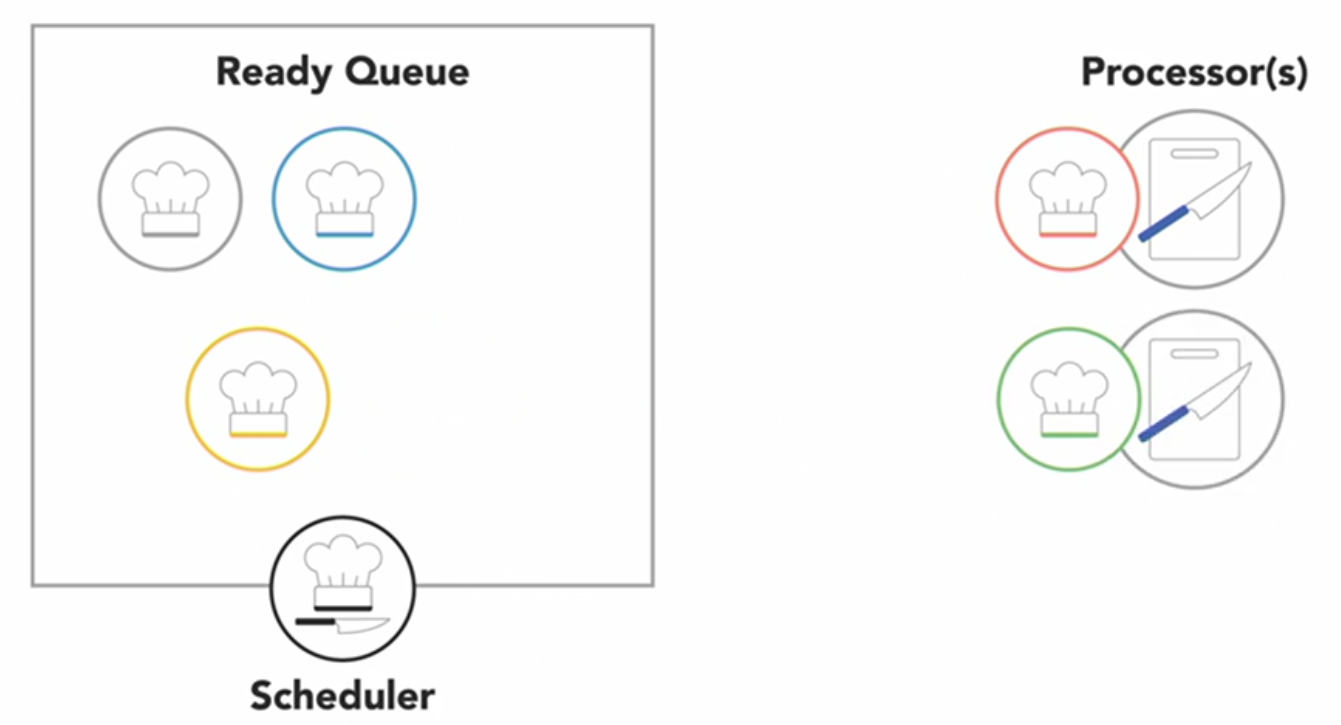
│ │

└────────────────────────────────────────────────────┘

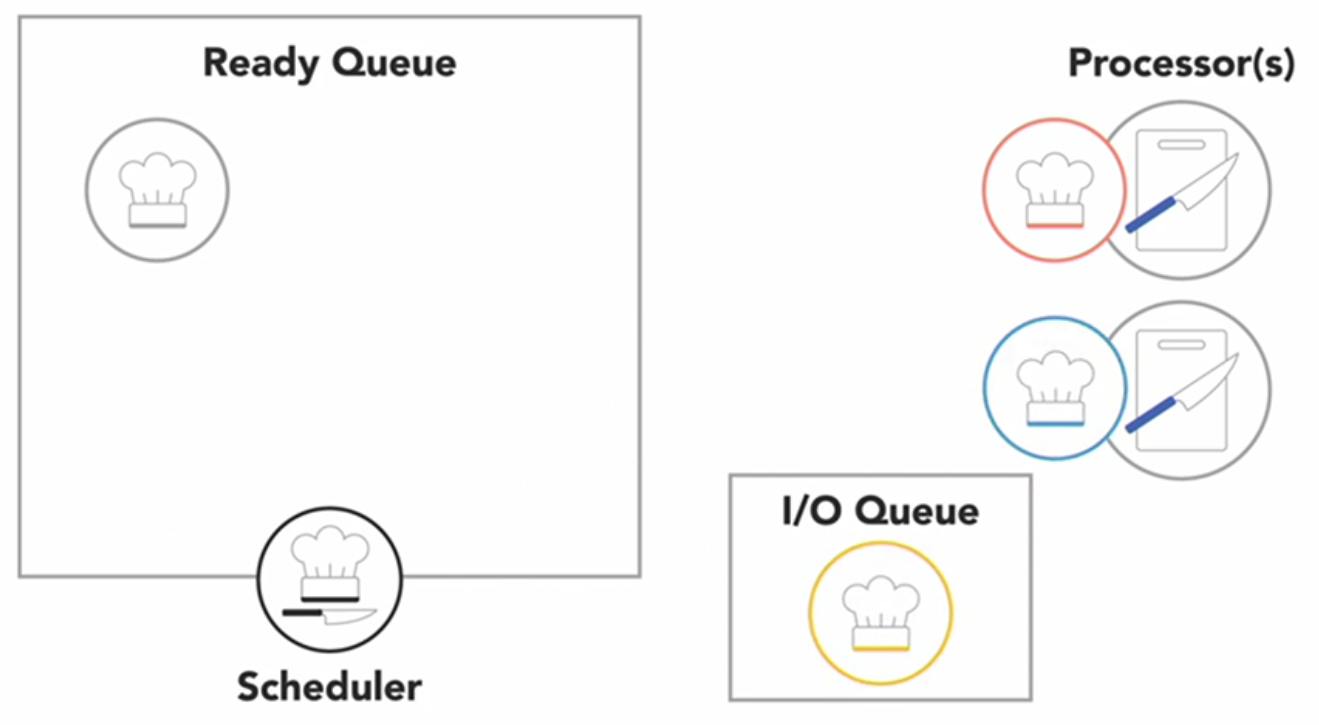


### Execution scheduling

* Scheduling 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 a 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 has to state the state or context of the process that was running so it can be resumed later.
* And it has to 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++;

}

//cout << name << " chopped " << vegetable\_count << " vegetables." << endl;

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.

### 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 threads to help out, referred to as its child threads, which are part of the same process but execute independently to do other tasks. Those threads can spawn their own children if needed.
* As each of those threads 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, threads 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

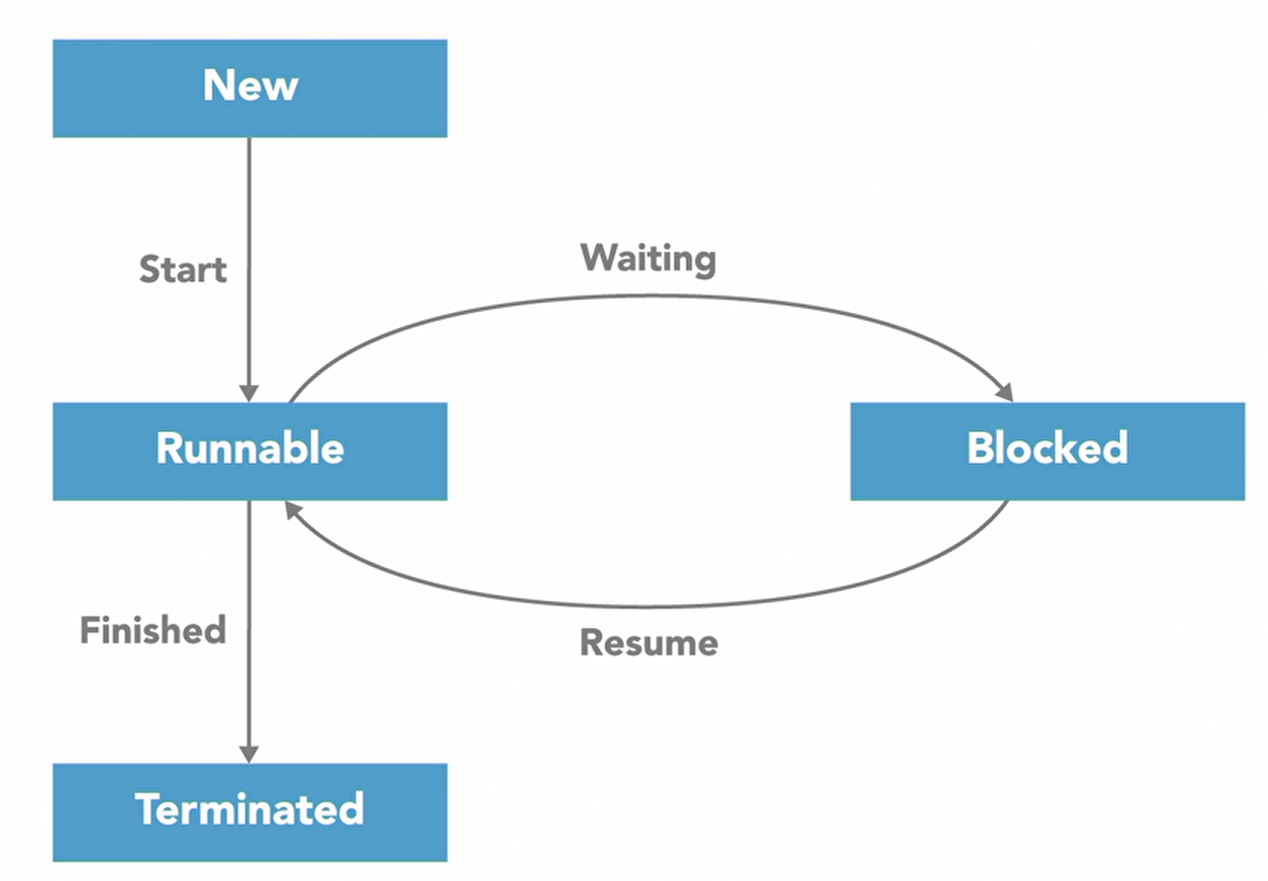
* If thread starts running, then it is in **runnable state**, which means the operating system can schedule this thread to execute.
* Through contact switches, this thread will get swapped out with other threads to run on one of the available processors.

##### 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 threads 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;

}

### Detached thread

* We often create threads 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.
* Threads 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 threads are usually spawned as non-daemon, or normal threads, and you must explicitly turn a thread into a daemon or background thread.
* When main thread is finished executing and there aren't any non-daemon threads 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;

}

## Mutual Exclusion

### Data race

* Data races are a common problem that can occur when two or more threads are concurrently accessing the same location in memory, and at least one of those threads 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 threads 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();

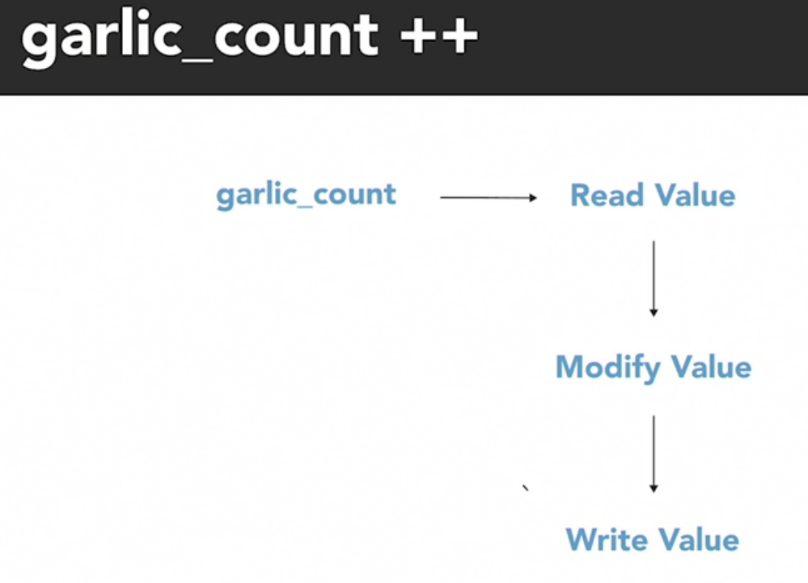
myWife.join();

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 threads end up stepping on each other's toes and unintentionally overwriting each other's changes to produce an incorrect result.

### Mutual exclusion

* Anytime multiple threads 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 threads 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 threads added garlic to the shared variable garlic\_count, because incrementing a value is actually a three-step process.
  1. Read the current value.
  2. Modify it.
  3. 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***.
* 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.

#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;

}

### 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;

}

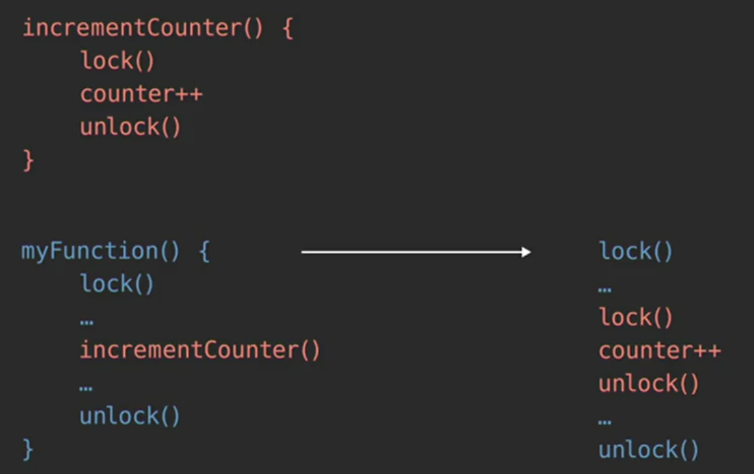
## Locks

### 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.

### 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.



#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;

}

### 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.
  1. If the ***mutex*** you're trying to lock is available, it will get locked and the method will return ***true***.
  2. 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>

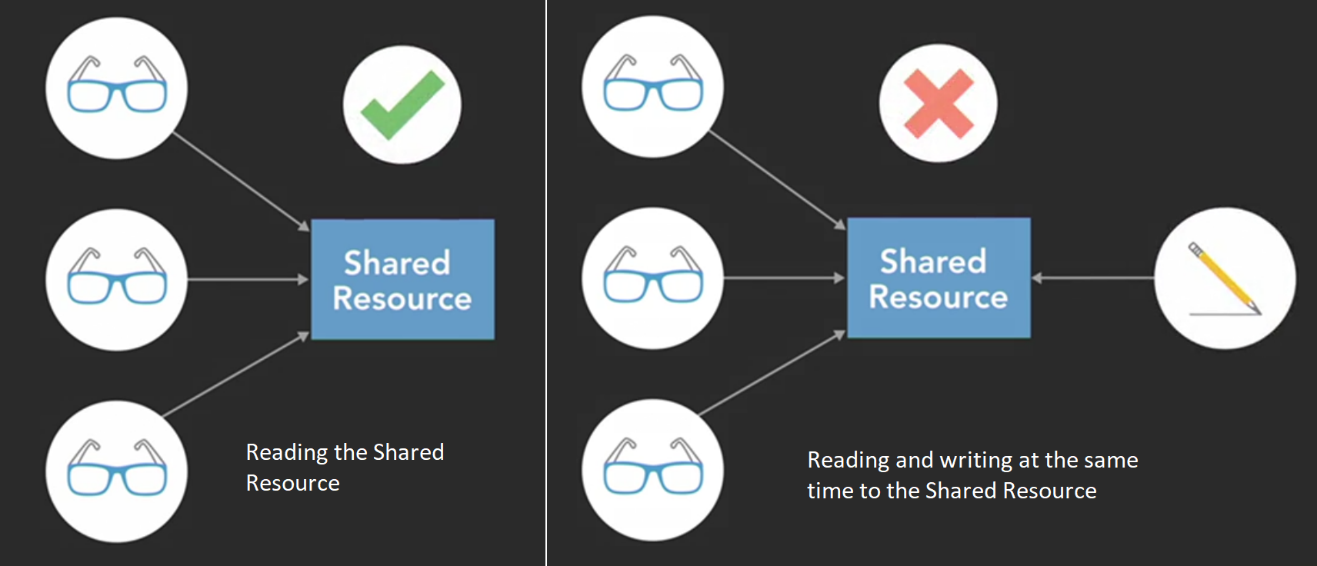
(std::chrono::steady\_clock::now() - start\_time).count();

printf("Elapsed Time: %.2f seconds\n", elapsed\_time / 1000.0);

return 0;

}

### Shared mutex

* We use a ***lock*** or ***mutex*** to protect a ***critical section*** of code to defend against data races, which can occur when multiple threads are concurrently accessing the same location in memory and at least one of those threads is writing to that location.
* That second part is key because if we have a bunch of threads 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 threads 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 threads 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 threads 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.
  1. It can be ***locked*** in a shared read mode that allows multiple threads that only need to read simultaneously to ***lock*** it.
  2. 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 <iostream>

#include <array>

#include <thread> // Needed for thread

#include <chrono> // Needed for time

#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()

{

std::array<std::thread, 10> readers;

for (unsigned int i = 0; i < readers.size(); i++)

readers[i] = std::thread(calendar\_reader, i);

std::array<std::thread, 2> writers;

for (unsigned int i = 0; i < writers.size(); i++)

writers[i] = std::thread(calendar\_writer, i);

for (unsigned int i = 0; i < readers.size(); i++)

readers[i].join();

for (unsigned int i = 0; i < writers.size(); i++)

writers[i].join();

return 0;

}

## Liveness

### Deadlock

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

THREAD1 THREAD2

┌─────────────────────┐

│ │

\* \* │ │ │ ▲ \* \*

\* \* │ │ ┌─┐ ┌─┐ │ │ \* \*

\* \* │ │ └─┘ └─┘ ┌─┐ │ │ \* \*

\* \* │ │ └─┘ │ │ \* \*

\* \* │ │ ┌─┐ ┌─┐ │ │ \* \*

\* \* │ │ └─┘ └─┘ ┌─┐ │ │ \* \*

▼ │ └─┘ │ │

CS1 │ │ CS2

└─────────────────────┘

Plate of Sushi

CS - **Chopstick**

* 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 threads 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 mutexes.
* 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 threads, neither one of the threads 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(CS2), std::ref(CS1));

THREAD1.join();

THREAD2.join();

printf("The philosophers are done eating.\n");

return 0;

}

### 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)

#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;

}

### 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 threads 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***.

#include <iostream>

#include <thread>

#include <mutex>

#include <array>

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;

}

##### Starvation – Example Code

### Livelock

* A ***livelock*** looks similar to a ***deadlock*** in the sense that two threads are blocking each other from making progress. But the difference is that the threads in a livelock are actively trying to resolve the problem.
* A livelock can occur when two or more threads are designed to respond to the actions of each other.
* Both threads are busy doing something, but the combination of their efforts prevent them from actually 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 threads 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();

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 – 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;

}