Tutorial 4: Operating Systems - Kernel Level Threads in C++

Yonghao Lee

April 23, 2025

Introduction - Multi-threading

Modern computers are equipped with multiple cores, with each core residing in a CPU. A computer may also contain multiple CPUs. These cores execute commands in true parallel fashion.

In our discussion of user-level threads last week, we noted that they are better described as a mechanism for jumping between sections of code. This approach does not involve truly parallel execution; rather, thread 1 runs until it stops, then thread 2 takes over, creating a switching pattern. However, true parallelism becomes possible with kernel-level threads on multi-core computers, where multiple threads can execute simultaneously on different cores.

This capability opens up the world of parallel programming, which we will explore in the coming weeks. Parallel programming is considerably more complex than regular sequential programming, as it utilizes shared memory resources.

The Idea Behind Multi-threading

The main idea of multi-threading is to allow multiple tasks in a single program to run in true parallel fashion, where the tasks are user-defined.

As different parts of the code are processed simultaneously, the program benefits from the multiple resources of the computer and may execute faster.

Consider the example of tabs in a web browser like Chrome, where each tab operates independently yet within the same application.

Sometimes, synchronization between threads becomes necessary to coordinate their activities.

Recall - Kernel Level Threads

Threads are executions of small sequences of program instructions.

User-level threads are managed by the user, who jumps between different executions to create the illusion that things run in parallel. These threads run concurrently but not in parallel.

In contrast, kernel-level threads are managed by the operating system and take advantage of the computer's multiple resources. The OS manages these resources and is aware of the threads' existence.

Implementation in C++

The `thread` Class

A thread is represented by an instance of the `std::thread` class. This is a templated class, which means it can work with different types of callable objects.

The `thread` header is located in `<thread>` and was introduced with the C++11 standard.

Creating Threads in C++

A thread runs a function, and this function may call other functions. Creating a thread is accomplished by constructing a `std::thread` object. Upon construction, the thread enters the operating system's scheduling queue immediately (meaning it becomes available for execution by the CPU scheduler).

The default empty constructor creates a non-executing thread object that represents no thread of execution.

We look at an incorrect example:

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

int x = 0;

void thread_function()
{
    for(int i = 0; i < 1000000; i++)
    {
        x++;
    }
}

int main(void)
{
    std::cout << "Creating Threads" << std::endl;
    std::thread t1(thread_function);
    std::thread t2(thread_function);
    t1.join();
    t2.join();
    std::cout << "x = " << x << std::endl;
    return 0;
}
```

Figure 1: Incorrect Thread Example

Figure 2: Running the Incorrect Example

Recall that each process has at least one thread running. In the example, we have 3 threads created: the main thread, t1, and t2.

Note that after t1 and t2 are created, the main thread does not stop and it could reach the end first before anything can be printed.

Since a thread may not be destructed before it is joined, this is illegal.

The crash occurred because when the main function returns, it attempts to destroy the thread objects t1 and t2. However, if threads are still running when you try to destroy them, the program calls `std::terminate()` which aborts execution.

We thus use `join()`. Joining 2 threads means one thread waits until the second one finishes.

The sequence looks like this:

1. Main thread creates t1 and t2
2. All three threads (main, t1, t2) begin running simultaneously
3. When the main thread reaches `t1.join()`, it pauses and waits for t1 to finish
4. After t1 finishes, the main thread continues and reaches `t2.join()`
5. The main thread then waits for t2 to finish
6. After t2 finishes, the main thread continues to the end of the program

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

void thread_func(int x, int y)
{
    std::cout << "x: " << x << ", y: " << y << std::endl;
}

int main(void)
{
    std::cout << "Creating Threads" << std::endl;
    std::thread t1(thread_func, 1, -1);
    std::thread t2(thread_func, 2, -2);

    t1.join();
    t2.join();
    return 0;
}
```

Figure 3: Using `join()` Method

```
parallels@fedora-linux-40:~/Desktop/Parallels Shared Folders/Home/Documents/cppt
est$ ./test
Creating Threads
x: x: 21, y: -2, y: -1

parallels@fedora-linux-40:~/Desktop/Parallels Shared Folders/Home/Documents/cppt
est$ ./test
Creating Threads
x: 1, y: -1
x: 2, y: -2
parallels@fedora-linux-40:~/Desktop/Parallels Shared Folders/Home/Documents/cppt
est$
```

Figure 4: Non-reproducible Result Due to Race Condition

Note that the output shows what happens when both threads execute their print statements almost simultaneously, caused by lack of proper synchronization.

`std::thread` Tips

Such problems we encountered make using threads uncomfortable; therefore, from C++20 onwards, we may also use `std::jthread`, which automatically joins the thread in destruction.

We may also use `std::this_thread` which has a few useful methods like `yield()`, which hints the scheduler to run another thread instead of the calling thread, and `sleep_for()` which sleeps for the given amount of time. We are going to see these in the examples to come.

Note that it is also possible to initialize a thread in construction with a lambda expression.

Variable Accessibility

Each thread running a function has its own stack; therefore, they do not share local variables. Yet, it is important to note that threads can access other threads' variables if given their address. This is because threads still live in the same process address space. Keep in mind that when the thread terminates, the variables are destroyed.

Usually, shared variables are allocated dynamically on the heap, or in the data segment (global variables).

Synchronization Issues in Multi-threading

Race Conditions

We demonstrate the race condition by using the following example:

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

int x = 0;

void thread_function()
{
    for(int i = 0; i < 1000000; i++)
    {
        x++;
    }
}

int main(void)
{
    std::cout << "Creating Threads" << std::endl;
    std::thread t1(thread_function);
    std::thread t2(thread_function);
    t1.join();
    t2.join();
    std::cout << "x = " << x << std::endl;
    return 0;
}
```

Figure 5: Mutual Exclusion Example

When we run the program multiple times, we see the results vary all the time, yet we expected x to be equal to 4000000:

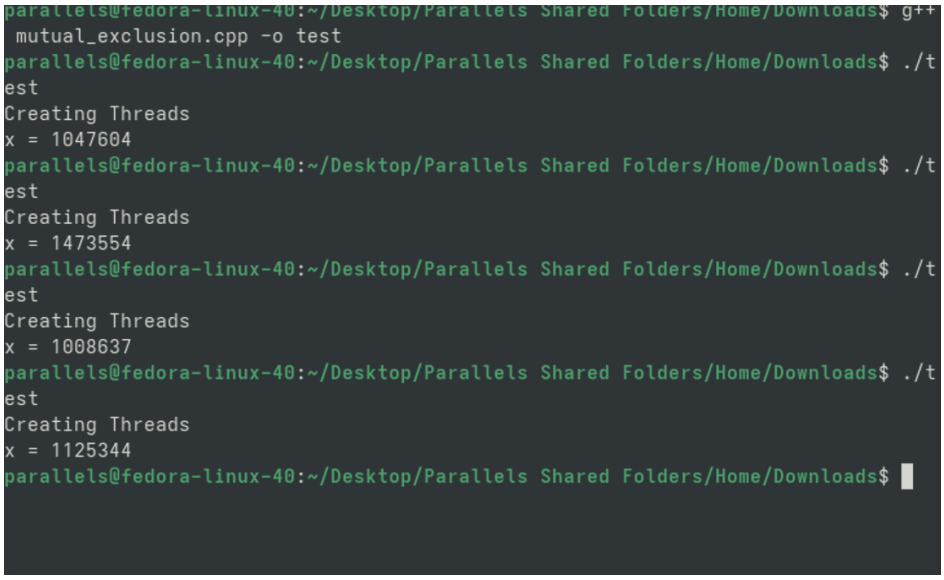


Figure 6: Race Condition Results

Understanding the Problem

The problem comes from the fact that the instruction `x++` is not an atomic operation and it consists of 3 assembly commands:

1. Load value of x from memory into register
2. Increment the value in the register
3. Store the value back to memory location of x

Let's look at the following example:

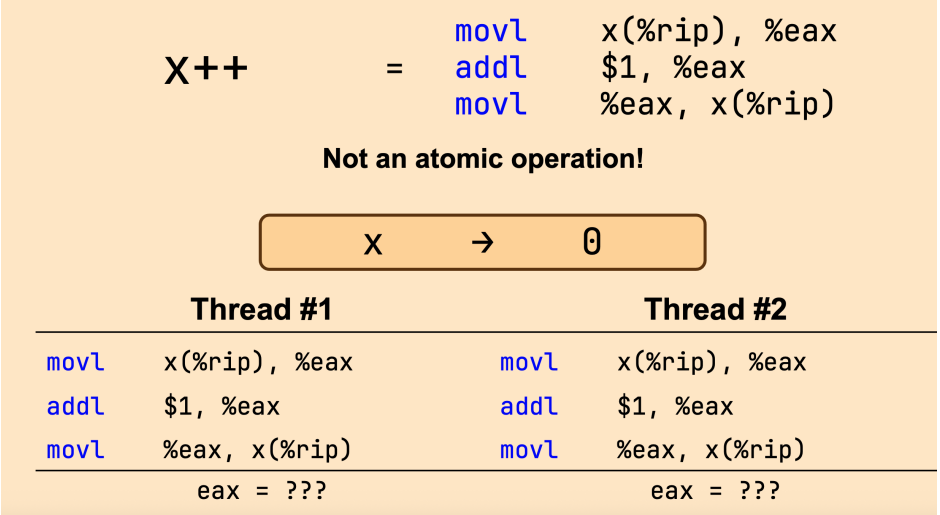


Figure 7: Mutual Exclusion - 2 Threads

As shown in the figure above, the increment operation `x++` gets translated into three separate assembly instructions:

1. `movl x(%rip), %eax` – Load the value of x from memory into the EAX register
2. `addl $1, %eax` – Add 1 to the value in the EAX register
3. `movl %eax, x(%rip)` – Store the value from EAX back to memory location of x

When multiple threads execute this non-atomic operation concurrently, a race condition occurs. Here's how it happens:

1. Initially, the shared variable x contains the value 0.
2. Thread #1 starts executing and loads the value of x (which is 0) into its EAX register.
3. Thread #2 also starts executing and loads the same value of x (0) into its EAX register.
4. Thread #1 adds 1 to its EAX register, making it 1. Before it stores the value back to the memory location of x.
5. Thread #2 also adds 1 to its EAX register, making it 1 as well.
6. Thread #1 stores the value 1 from its EAX register back to the memory location of x.
7. Thread #2 also stores the value 1 from its EAX register back to the memory location of x, overwriting the previous update by Thread #1.

After both threads have executed their operations, the final value of x is 1, not 2 as expected. The increment performed by Thread #1 has effectively been lost due to Thread #2 overwriting it with its own calculation based on the original value.

Solutions to Race Conditions

Atomic Operations in C++

To fix this problem, we have atomic operations such as fetch-and-add which we have seen in the lecture. Atomic operations ensure that the entire operation completes as a single, indivisible unit without interruption from other threads.

C++ offers a convenient atomic wrapper class called `std::atomic` in the `<atomic>` header. Using this wrapper makes a variable "atomic" in a very particular sense, preventing race conditions when multiple threads attempt to modify the same variable simultaneously.

The `std::atomic` wrapper provides several atomic operations, including:

- **Load & Store**: Atomic reading and writing of values
- **CAS (Compare and Swap)**: Atomically compares the value with an expected value and swaps it if they match

- **Fetch & Add**: Atomically adds a value and returns the previous value

Additionally, the atomic wrapper overrides various operators to make them work atomically:

- Arithmetic compound assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`
- Bitwise compound assignment operators: `|=`, `&=`, `^=`
- Increment and decrement operators: `++`, `--`

Important Considerations

It's crucial to understand that not all expressions involving atomic variables are guaranteed to be atomic. For example:

- `x += 5;` is atomic when x is an `std::atomic<int>`
- `x = x + 5;` is NOT atomic, even when x is an `std::atomic<int>`

This is because the right-hand side (RHS) of the second expression involves reading the value, performing an operation, and then assigning, which is not atomic as a whole.

Example Implementation

Here's how we can fix our race condition example using atomic variables:

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

std::atomic<int> x = 0;

void thread_function()
{
    for(int i = 0; i < 1000000; i++)
    {
        x++;
    }
}

int main(void)
{
    std::cout << "Creating Threads" << std::endl;
    std::thread t1(thread_function);
    std::thread t2(thread_function);
    t1.join();
    t2.join();
    std::cout << "x = " << x << std::endl;
    return 0;
}

```

Figure 8: Atomic Mutual Exclusion

With this implementation, the `x++` operation is now atomic, ensuring that all increments are properly counted without race conditions. The program will consistently output the expected value of 2,000,000.

Thread Safety

It is important to note that C++ standard library functions and classes are not thread safe by default. This means that modification and access to the same container by multiple threads is not well defined and can cause problems such as data corruption, unexpected behavior, or program crashes.

When using containers like vectors, maps, or strings in multi-threaded code, you must provide your own synchronization mechanisms.

Thread Safety Example

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

#define RANGE 1000000

void thread_function(std::vector<int> &shared, int range)
{
    for(int i = 0; i < range; i++)
    {
        shared.push_back(i);
    }
}

int main()
{
    std::vector<int> vec;
    {
        std::jthread threads[8];
        for(int i = 0; i < 8; i++)
        {
            threads[i] = std::jthread(thread_function, std::ref(vec), RANGE);
        }
    }
    std::cout << "vec has " << vec.size() << " elements" << std::endl;
    return 0;
}
```

Figure 9: Thread Safety Example

When we run the program, we get an error:

```
maor@maor:~/os/tests/vector$ g++ -std=c++20 vector_mutual_exclusion.cpp -o test
maor@maor:~/os/tests/vector$ ./test
munmap_chunk(): invalid pointer
munmap_chunk(): invalid pointer
munmap_chunk(): invalid pointer
Aborted (core dumped)
```

Figure 10: Running the Wrong Program

In this example, the function `thread_function` receives a reference to a shared vector and a range value. Its purpose is to add `range` number of integers to the vector.

The main function creates an empty vector of integers called `vec` and opens a new block using curly braces to create a specific scope. Within this scope, we create an array of 8 `std::jthread` objects.

Each `jthread` can be thought of as a separate worker that executes independently. These threads will automatically join (wait for completion) when they go out of scope at the end of the block.

The critical issue is that all 8 threads attempt to modify the same vector simultaneously, which is made possible by passing the vector by reference using `std::ref`. However, this creates a dangerous race condition.

The problem stems from how vectors manage memory:

1. When a vector runs out of space, it allocates a new, larger block (typically twice its current capacity)

2. It then copies all elements to the new location

3. Finally, it deallocates the old memory

During this reallocation process, if one thread (T_1) is adding elements while another thread (T_2) triggers a reallocation, T_1 may end up writing to memory that is no longer valid.

This error manifests as a `munmap_chunk()` error, indicating that the program is attempting to access or free memory that was either already freed or is invalid.

A mutex is the simplest primitive synchronization mechanism to solve this problem. It is also called a lock, and it is used to protect critical sections. The `lock/acquire` operation locks the mutex so that any other thread calling `lock` will be blocked or put to sleep, while `unlock/release` releases the mutex and wakes up a waiting thread if any exists.

In our example, we can add a mutex to protect our `push_back()` operations. When a thread starts the `push_back()` operation, it first locks the mutex and only releases it after the operation completes.

C++ implements mutexes in the `std::mutex` class. This class provides an abstraction of a mutex. The mutex object must be shared among the threads. Different mutex objects represent different mutexes, each controlling access to a distinct resource.

Let's look at the following example:

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

void thread_func(std::mutex &m, int thread_num)
{
    m.lock();
    std::cout << "I am thread " << thread_num << std::endl;
    m.unlock();
}

int main()
{
    std::mutex m;
    std::jthread threads[10];
    for(int i = 0; i < 10; i++)
    {
        threads[i] = std::jthread(thread_func, std::ref(m), i);
    }
}
```

The mutex must be passed by reference, or by a pointer.
Do you understand why?

Figure 11: Mutex Example

When using a mutex to synchronize access across multiple threads, the mutex must be passed by reference or pointer. This is critical for two reasons:

1. **Shared Synchronization Point**: All threads must access the *same* mutex instance to properly coordinate access to shared resources.

2. **Non-copyable Design**: `std::mutex` is intentionally designed to be non-copyable to prevent synchronization errors.

- A single mutex instance `m` is created in `main()`.

- Ten threads are created, each receiving a reference to the same mutex using `std::ref(m)`.
- Each thread calls `lock()` on this shared mutex before printing, ensuring mutual exclusion.
- After printing, each thread calls `unlock()` to release the mutex for other threads.

If the mutex were passed by value:

- Compilation would fail because `std::mutex` is non-copyable.
- Even if it were copyable, each thread would operate on a different mutex instance, defeating the purpose of synchronization.

The `std::ref()` wrapper ensures the mutex is passed by reference, creating a shared synchronization point across all threads.

`std::unique_lock` is a wrapper helper class for using mutexes in C++. It provides several advantages over directly manipulating mutex objects.

- **Calling lock() again with no blocking (recursive locking):**
  Unlike regular mutexes, which would deadlock if a thread tries to lock a mutex it already holds, `unique_lock` allows the same thread to call lock() multiple times without causing a deadlock.
- **Ownership transfer and lock swaps:**
  The lock can be moved between different objects and functions, allowing flexible design patterns.
- **Use for condition variables:**
  Works seamlessly with condition variables, allowing proper synchronization between threads.

`std::unique_lock` follows the RAII principle:

- Calling to `std::unique_lock`'s constructor automatically locks the mutex.
- Calling to `std::unique_lock`'s destructor (at the end of the scope) automatically releases the mutex, if it is locked.

It is uncommon to lock/unlock the `unique_lock` by yourself. Instead, you typically rely on RAII to handle locking and unlocking automatically.

The following code demonstrates `std::unique_lock` in a multi-threaded application:

In this example:

1. We create a single mutex `m` that will be shared by all threads.
2. We define a function `thread_func` that each thread will execute. This function:
   - Takes a reference to the shared mutex and a thread identifier

- Creates a `std::unique_lock` that automatically locks the mutex
- Outputs a message indicating which thread is running
- When the function exits, the `lock` object is destroyed, automatically releasing the mutex

The rest of the code is identical to our previous example.

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

void thread_func(std::mutex &m, int thread_num)
{
    std::unique_lock<std::mutex> lock(m);
    std::cout << "I am thread " << thread_num << std::endl;
}

int main()
{
    std::mutex m;
    std::jthread threads[10];
    for(int i = 0; i < 10; i++)
    {
        threads[i] = std::jthread(thread_func, std::ref(m), i);
    }
}
```

Figure 12: Unique Lock

Note that we don't need to explicitly call `unlock()` - it happens automatically when each thread function completes.

Semaphore Concept

A semaphore is a synchronization primitive that generalizes the concept of a mutex:

- A semaphore maintains an internal counter that represents available resources or slots
- It allows at most N threads to enter a critical section simultaneously, where N is the initial value
- Unlike a mutex, a semaphore's value can be modified at runtime (incremented before being decremented)
- When the counter reaches zero, threads attempting to acquire the semaphore will block until another thread releases it

C++20 Semaphore Types

std::counting_semaphore

`std::counting_semaphore` is a C++20 synchronization primitive with the following key properties:

- Constructor takes an initial value that determines how many threads can simultaneously acquire the semaphore
- `release()`: Increments the internal counter (also known as "up" or "post" operation)
- `acquire()`: Decrements the internal counter or blocks if counter is zero (also known as "down" or "wait" operation)

std::binary_semaphore

`std::binary_semaphore` is a specialized semaphore type with a maximum count of 1:

- It is defined as an alias template: `using binary_semaphore = counting_semaphore<1>;`
- Functions similarly to a mutex but with different ownership semantics
- Useful for simple signaling between threads

Key Differences from Mutex

- A mutex can only be locked by one thread and unlocked by the same thread
- A semaphore can be acquired and released by different threads
- Semaphores have no thread ownership concept
- Semaphores can allow multiple concurrent accesses to a critical section

Semaphore Example

Let's look at the following example demonstraing the usage of semaphore: This code demonstrates thread synchronization using a counting semaphore in C++:

```cpp
#include <iostream>
#include <thread>
#include <semaphore>

int x = 0;
std::counting_semaphore sem(0);

void thread1(void)
{
    // do some calculations
    x++;
    // do some calculations
    std::this_thread::sleep_for(std::chrono::seconds(1)); // simulate work
    x++;
    // now thread2 continues
    sem.release();
}

void thread2(void)
{
    sem.acquire();
    std::cout << "Thread 2: x = " << x << std::endl;
}

int main(void)
{
    std::thread t1(thread1);
    std::thread t2(thread2);

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

    return 0;
}
```

Figure 13: Semaphore

- x: Global variable initialized to 0
- sem: Counting semaphore initialized to 0

Execution Flow

$$\textbf{thread1:} \begin{cases} x \leftarrow x + 1 \quad \text{(increment } x \text{ to 1)} \\ \text{sleep for 1 second} \\ x \leftarrow x + 1 \quad \text{(increment } x \text{ to 2)} \\ sem.release() \quad \text{(increment semaphore from 0 to 1)} \end{cases} \tag{1}$$

$$\textbf{thread2:} \begin{cases} sem.acquire() \quad \text{(wait until semaphore} > 0\text{, then decrement it)} \\ \text{print value of } x \end{cases} \tag{2}$$

The semaphore sem enforces the following constraint:

- $thread2$ can only proceed after $thread1$ has completed its work and called $sem.release()$

- This ensures $thread2$ always sees $x = 2$

If semaphore were initialized to 2 instead of 0:

- Both threads could proceed immediately without waiting
- No synchronization would occur
- $thread2$ could print $x = 0$, $x = 1$, or $x = 2$ depending on execution timing
- A race condition would exist

Condition Variables

Basic Concept

A condition variable (also called a **monitor**) is a synchronization object used to control conditional entrance to a section. It provides a mechanism for threads to wait until a specific condition occurs.

Operations on Condition Variables

Threads have two primary operations with condition variables:

- **Wait**: Threads wait before entering a section
- **Signal**: Another thread signals to waiting threads
  - **Signal/Notify**: Notifies only a single thread that it is permitted to enter
  - **Broadcast**: Notifies all waiting threads that they are permitted to continue

Condition Variables with Mutex

Condition variables are used as a "sleep"-"wake up" mechanism inside a critical section. The typical pattern is:

1. Before waiting, acquire a mutex and enter the first part of the critical section
2. Release the mutex before waiting (to avoid blocking other threads from entering the critical section)
3. Wait for another thread to wake us up
4. When waking up, reacquire the mutex
5. Continue execution in the critical section with the mutex locked
6. Release the mutex at the end of the critical section

Code Example

Listing 1: Pseudocode for condition variable usage

```
// Pseudocode for condition variable usage
mutex.lock();          // Acquire mutex to enter critical section
while (!condition) {   // Check condition
    cv.wait(mutex);    // Wait releases mutex and blocks until
       signaled
}                      // On wake, mutex is automatically reacquired
// Critical section continues...
mutex.unlock();        // Release mutex when done
```

Condition Variable in C++

The C++ standard library implements condition variables through the `std::condition_variable` class, which provides a synchronization primitive that enables threads to wait until a particular condition occurs.

Key Methods

- **wait**

```
    wait(std::unique_lock<std::mutex> &lock)
```

  - Takes a `unique_lock` reference as a parameter (RAII wrapper around a mutex)
  - Atomically unlocks the mutex contained in the lock
  - Blocks the current thread until the condition variable is notified
  - When the thread wakes up, it automatically reacquires the mutex before returning

- **notify_one**

```
    notify_one(void)
```

  - Wakes up a single waiting thread
  - If multiple threads are waiting, it's usually unspecified which one will be awakened
  - If no threads are waiting, this call does nothing

- **notify_all**

```
    notify_all(void)
```

  - Wakes up all waiting threads
  - All threads that were waiting will compete for the mutex when they wake up
  - If no threads are waiting, this call does nothing

Usage Pattern

The C++ implementation follows the standard condition variable pattern but enhances safety through the RAII principle with the `unique_lock` class. This ensures proper mutex locking/unlocking even in the presence of exceptions.

Listing 2: Condition Variable Usage Pattern

```
std::mutex mtx;
std::condition_variable cv;
bool condition = false;

// Waiting thread
{
    std::unique_lock<std::mutex> lock(mtx);
    while (!condition) {
        cv.wait(lock);  // Releases lock and waits
    }                   // Lock is automatically reacquired on wake
    // Process with condition true and lock held
}  // Lock is automatically released when unique_lock goes out of
   scope

// Notifying thread
{
    std::unique_lock<std::mutex> lock(mtx);
    condition = true;
    cv.notify_one();  // Or cv.notify_all()
}
```

Spurious Wakeups

An important consideration when using condition variables is the phenomenon of **spurious wakeups**:

- According to the C++ standard, a thread in a `wait` call may be awakened spuriously, even if no thread explicitly called `notify_one()` or `notify_all()`.
- Spurious wakeups can occur due to implementation details of the operating system or hardware.
- A spurious wakeup might happen when:
  - No notification was sent at all
  - A notification was sent but already woke up another thread

Handling Spurious Wakeups

Although spurious wakeups are rare, they must be handled for reliable concurrent code:

- Always check the actual condition in a loop (not just an if statement)
- If the condition is not met after waking up, call `wait` again

Example

Listing 3: Handling Spurious Wakeups

```
// Thread 0 (Notifying thread)
std::unique_lock<std::mutex> lock(mtx);
x++;
if(x == 4) {
    cv.notify_all();
}

// Thread 1 (Waiting thread)
std::unique_lock<std::mutex> lock(mtx);
x++;
while(x != 4) {  // Note the while loop, not an if statement
    cv.wait(lock);
}
```

The key pattern is using a `while` loop instead of an `if` statement. This ensures that if Thread 1 wakes up spuriously when the condition is still not met, it will return to waiting instead of incorrectly proceeding.

Barrier

A barrier is a synchronization mechanism where all threads must reach a specific point before any can proceed past it. Unlike primitive synchronization mechanisms (mutexes, semaphores), barriers must be implemented using these primitives.

Implementation Components

- **Shared Counter**: An integer variable (named `arrived` in our C++ implementation) that tracks how many threads have reached the barrier.
- **Condition Variable (CV)**: A synchronization object (`std::condition_variable` in C++) that allows threads to wait until signaled.
- **Mutex**: Used to protect access to the shared counter (`std::mutex` in our implementation).
- **Thread Count**: A constant value (`num_threads`) representing the total number of threads that must arrive at the barrier.

Implementation Algorithm

```
1: procedure BARRIER(n)                                   ▷ n is total number of threads
2:     Initialize counter = 0                             ▷ Named arrived in our C++ code
3:     Initialize condition variable cv
4:     Initialize mutex m
5:     function barrier_wait():
6:         lock(m)                                        ▷ Using std::unique_lock in C++
7:         counter = counter + 1
8:         arrived = counter      ▷ In C++: int currently_arrived = ++this->arrived
9:         if arrived < n then
10:            cv.wait(m)                                 ▷ Releases m and blocks until signaled
11:        else
12:            counter = 0                                ▷ Reset for reuse in C++: this->arrived = 0
13:            cv.broadcast()                             ▷ In C++: this->cv.notify_all()
14:        end if
15:        unlock(m)                                      ▷ Automatic with std::unique_lock in C++
16:    end function
17: end procedure
```

C++ Implementation Details

In our C++ implementation, the barrier consists of a class with the following structure:

- **Constructor**: Takes the number of threads and initializes counters:

```cpp
Barrier::Barrier(int num_threads)
{
    this->arrived = 0; // number of threads that have arrived
    this->num_threads = num_threads; // total number of threads
}
```

- **Wait Method**: Implements the core barrier synchronization logic:

```cpp
void Barrier::wait()
{
    // acquire mutex to increment 'arrived'
    std::unique_lock<std::mutex> lock(this->mtx);
    // now the mutex is locked
    int currently_arrived = ++this->arrived;
    if(currently_arrived == this->num_threads)
    { // if all threads have arrived, notify all (broadcast)
        this->arrived = 0; // reset this->arrived
        this->cv.notify_all();
    }
```

```
        else
        { // wait for broadcast, release the lock
            while(this->arrived != 0)
            {
                this->cv.wait(lock);
            }
        } // reaching this point, the mutex is locked.
        // the mutex is automatically released since we used
           std::unique_lock
    }
```

Explanation

When a thread calls `barrier_wait()`:

1. It acquires the mutex to safely access the shared counter using `std::unique_lock`.
2. It increments the counter, indicating it has arrived at the barrier.
3. It checks if it's the last thread to arrive (if `currently_arrived == num_threads`).
4. If not the last thread:
   - It enters a while loop that checks if `arrived` is still non-zero.
   - It waits on the condition variable, which atomically releases the mutex and blocks the thread.
   - When woken up, it rechecks the while condition to ensure proper wakeup.
5. If it is the last thread:
   - It resets the `arrived` counter to 0, allowing the barrier to be reused.
   - It broadcasts on the condition variable, waking up all waiting threads.
6. Finally, the mutex is released automatically when the `std::unique_lock` goes out of scope.

Implementation Note

The C++ implementation uses a while loop with a condition check (`while(this->arrived != 0)`) to ensure correct behavior against spurious wakeups, which can occur with condition variables. This is more robust than just using `if` as in the pseudocode.

Testing the Barrier

The barrier can be tested with multiple threads that sleep for varying times before reaching the barrier:

```
auto thread_function = [](int i, Barrier &barrier)
{
    std::this_thread::sleep_for(std::chrono::seconds(i));
    std::cout << "Thread " << i << " is waiting at the barrier." <<
       std::endl;
    barrier.wait();
    std::cout << "Thread " << i << " has passed the barrier." << std
       ::endl;
};
```

When this test runs, you will observe:

1. Threads arriving at different times based on their sleep duration
2. All threads waiting until the last one arrives
3. All threads proceeding past the barrier simultaneously

This ensures all threads are synchronized at the barrier point - no thread can proceed until all have arrived.

If we run the code testing the barrier, we get the result we expect:

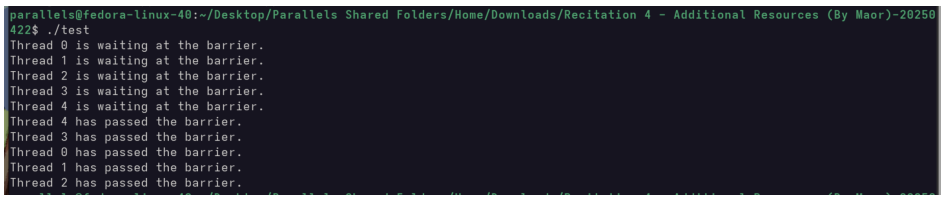


Figure 14: Barrier Test

Thread Local Storage (TLS)

- **Thread-local storage** is a memory-management technique that gives each thread its own copy of variables that would normally live in the data segment.
  - Recall: the data segment holds *static* and *global* variables.
- Unlike ordinary static or global variables (which are shared among all threads), **thread-local** variables are *not* shared.
  - Each thread gets and uses its own independent copy.

Thread Local Storage (TLS) in C++

- In C++, the keyword `thread_local` declares a variable as "one copy per thread."

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

thread_local int x;

void thread_func(int tid)
{
    x = tid;
    std::this_thread::sleep_for(std::chrono::microseconds(200));
    // t2 probably already updated x
    // we would expect x to be 1
    std::cout << "x = " << x << std::endl;
}

int main()
{
    std::jthread t1(thread_func, 0);
    // t2 delays and updates x probably after t1
    std::this_thread::sleep_for(std::chrono::microseconds(100));
    std::jthread t2(thread_func, 1);
    return 0;
}
```

Figure 15: Thread Local Storage Example

When we run it:

```
parallels@fedora-linux-40:~/Desktop/Parallels Shared Folders/Home/Downloads/Recitation 4 - Additional Resources (By Maor)-20250
422$ g++ -std=c++20 thread_local.cpp -o main
parallels@fedora-linux-40:~/Desktop/Parallels Shared Folders/Home/Downloads/Recitation 4 - Additional Resources (By Maor)-20250
422$ ./main
x = 0
x = 1
parallels@fedora-linux-40:~/Desktop/Parallels Shared Folders/Home/Downloads/Recitation 4 - Additional Resources (By Maor)-20250
422$
```

Figure 16: TLS Execution Result

How it works

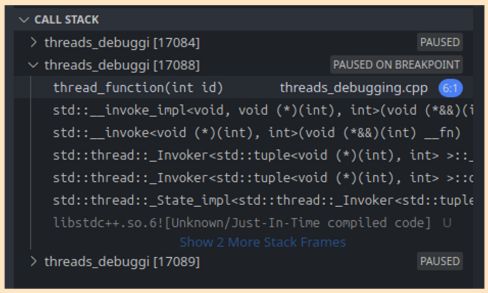
- `t1` sets its `x` to 0, then sleeps.
- `t2` starts later, sets its own `x` to 1.
- When each wakes up and prints, `t1` prints "x = 0" and `t2` prints "x = 1" — no locking needed.

Debugging with Threads

We can debug our multi-threaded programs, and we should note that print does not necessarily print to the screen at the same moment, and print causes delays, which can affect the race conditions we want to debug. If we use breakpoints, they are not individual for each thread, since each time a thread reaches a breakpoint, the whole process stops.

We can see the stack of all the currently running threads, and remember that threads start to run the moment they are created, so if the breakpoint is immediate we may not see all our created threads. We can alternate between different threads and trace their stack individually.

- Vscode:



- CLion:

Figure 17: Debug with threads