CMPE 202
Software Systems Engineering
February 4 Class Meeting

Engineering Extended Studies
San Jose State University

Spring 2026
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

Today

- Go over Assignment #1
- Introduction to multithreaded programming
  - Shared resources
  - Critical regions
  - Mutexes
  - Locks
  - Atomic variables
  - Condition variables

C++ support for multithreading is a major topic that involves a complex library of objects and functions. This introduction only presents the most basic ideas.

- The `auto` keyword and the `decltype` pseudo-function.

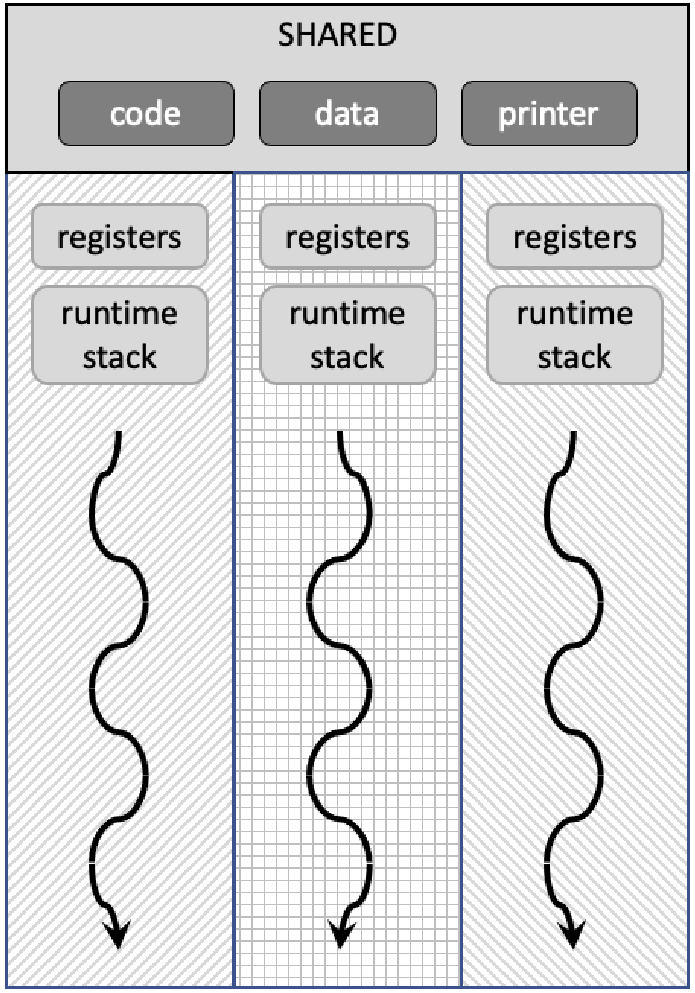
Intro to Multithreaded Programming

- Knowing how to design and develop multithreaded applications allows us to take advantage of a computer's ability to handle multiple threads of execution.
  - A properly designed multithreaded program can take better advantage of today's multicore computers.

- A thread is a path the computer takes through the code as it executes your program.
  - Multithreading means the computer is running multiple paths simultaneously.

Conceptual View of Multithreading

- An application's code, data, and external resources are shared among the threads.
- Each thread has its own runtime stack and a copy of the machine registers.
- The main thread of a program can spawn (create) child threads.



Simultaneous: Concurrency vs. Parallelism

- Example: Planning for a dinner party.
- Concurrent work by yourself:
  - Switch among simultaneous tasks of cleaning, cooking, setting the table, etc.
  - You can physically do only one task at a time.
  - Coordinate use of resources (mops, stove, pots and pans, etc.).
- Parallel work with friends:
  - Friends do multiple tasks simultaneously.
  - Must coordinate use of resources.

Critical Regions and Mutual Exclusion

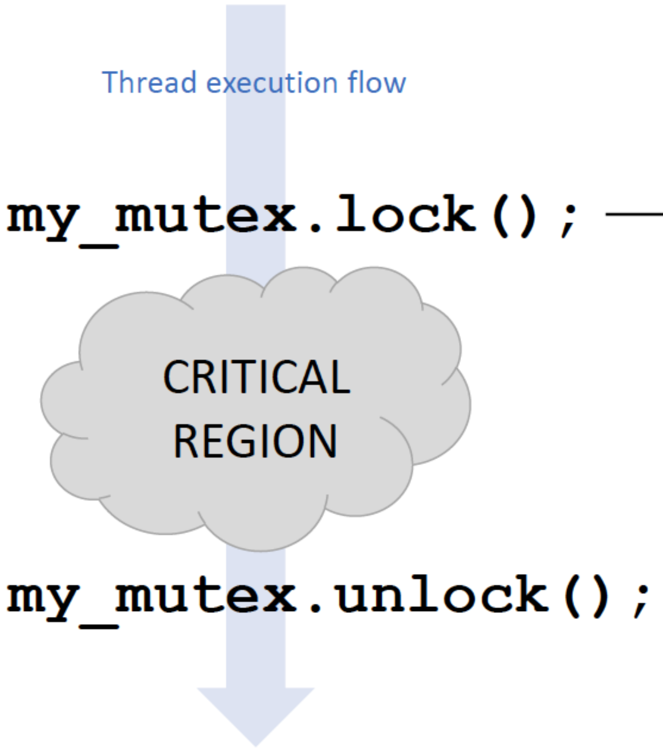
- A multithreaded program must protect a shared resource so that multiple threads don’t step on each other.
  - Examples: data in memory, a printer, etc.

- The critical region of a thread is its <u>code</u> that accesses a share resource.

- Mutual exclusion allows only <u>one thread at a time to be in its critical region</u>.

Mutex

- A mutex is an object that enforces mutual exclusion.



This thread attempts to lock **my_mutex** which is guarding the critical region.

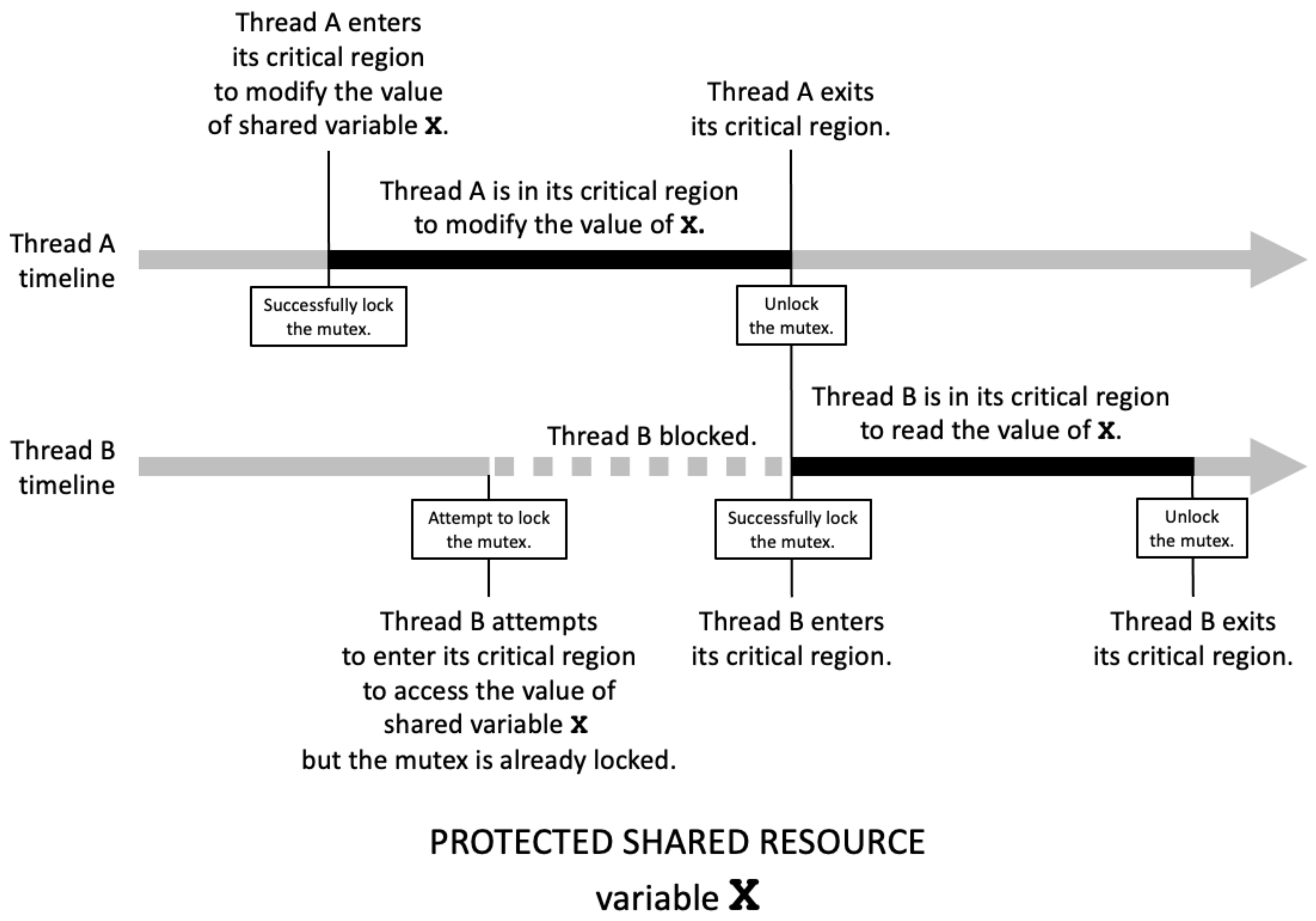
If **my_mutex** is already locked by another thread, then this thread blocks.

When **my_mutex** is unlocked by the other thread, then this thread again attempts to lock **my_mutex**.

If **my_mutex** is not locked, then this thread locks it and enters its critical region. No other thread can enter its critical region that is also guarded by **my_mutex**.

This thread must unlock **my_mutex** when it exits its critical region to give another thread that's blocked a chance to lock the mutex and enter its critical region.

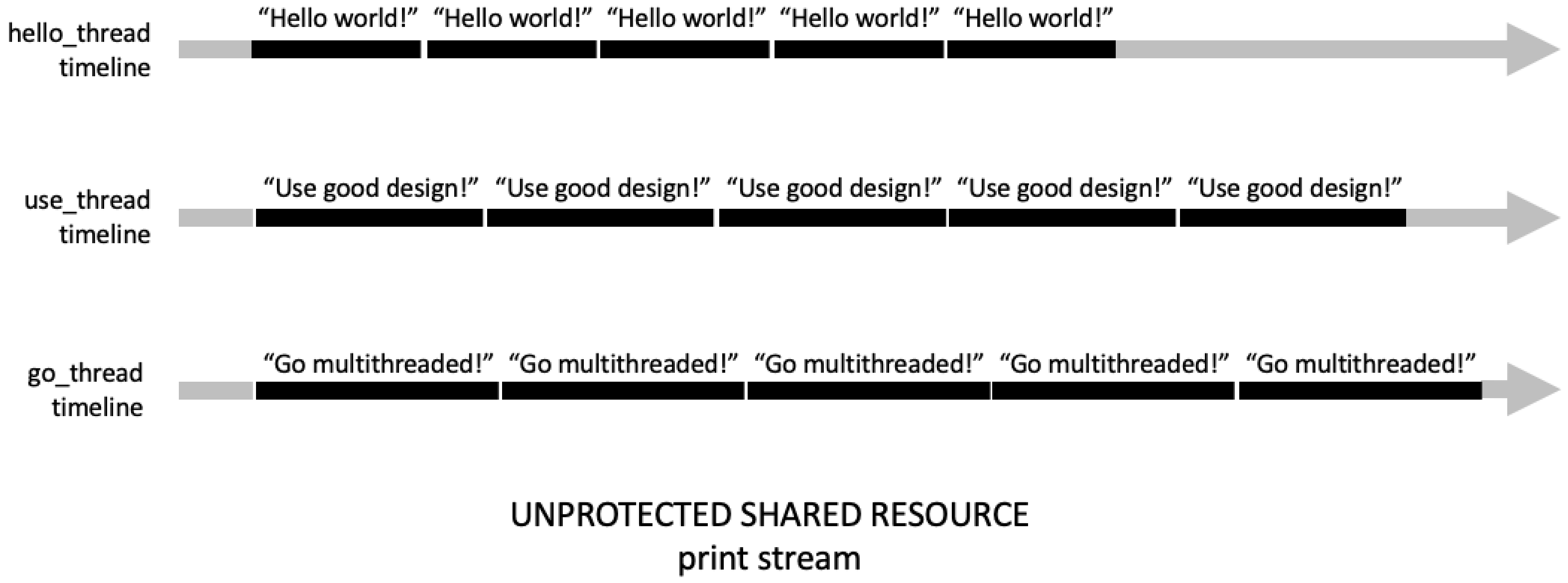
Locking and Unlocking a Mutex


Thread A enters its critical region to modify the value of shared variable **X**.
Thread A exits its critical region.
Thread A is in its critical region to modify the value of **X**.
Thread A timeline
Successfully lock the mutex.
Unlock the mutex.
Thread B is in its critical region to read the value of **X**.
Thread B blocked.
Thread B timeline
Attempt to lock the mutex.
Successfully lock the mutex.
Unlock the mutex.
Thread B attempts to enter its critical region to access the value of shared variable **X** but the mutex is already locked.
Thread B enters its critical region.
Thread B exits its critical region.
PROTECTED SHARED RESOURCE
variable **X**

The critical regions of threads A and B modify the same shared resource, variable X. A mutex object guards the critical regions. A thread that successfully locks the mutex can enter its critical region. Attempting to lock a mutex that's already locked causes the thread to block. The thread that's exiting its critical region must unlock the mutex to allow another thread to lock it.

Multithreaded Printing

- Multiple threads print simultaneously.
  - Shared resource: the output stream.
- Without mutual exclusion:


hello_thread timeline
"Hello world!" "Hello world!" "Hello world!" "Hello world!" "Hello world!"
use_thread timeline
"Use good design!" "Use good design!" "Use good design!" "Use good design!" "Use good design!"
go_thread timeline
"Go multithreaded!" "Go multithreaded!" "Go multithreaded!" "Go multithreaded!" "Go multithreaded!"
UNPROTECTED SHARED RESOURCE
print stream

Multithreaded Printing, *cont'd*

16.1/main.cpp

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

using namespace std;

const int COUNT = 5;

void print(string text);

int main(void)
{
    thread hello_thread(print, "Hello, world!\n");
    thread use_thread(print, "Use good design!\n");
    thread go_thread(print, "Go multithreaded\n");

    hello_thread.join();
    use_thread.join();
    go_thread.join();

    cout << endl << "Program done." << endl;
    return 0;
}

void print(string text)
{
    for (int i = 0; i < COUNT; i++)
    {
        for (const char &ch : text) cout << ch;
    }
}
```

Spawn child threads, each calls function `print()`.

Wait for threads to complete.

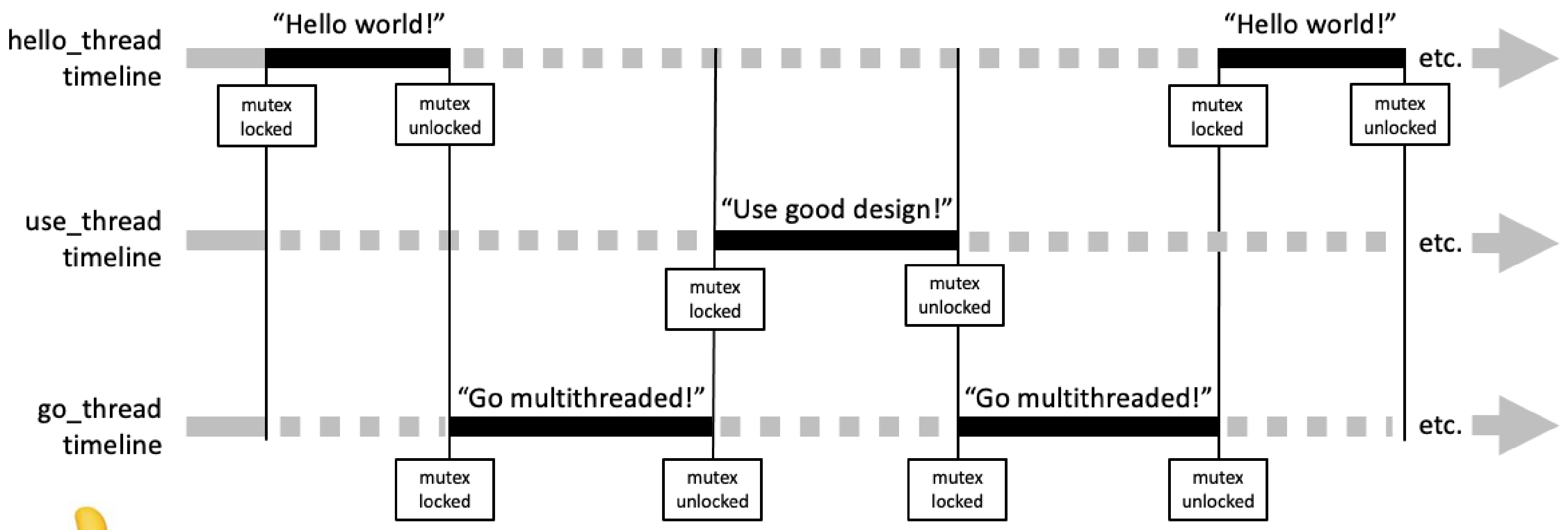
Function that each thread executes.

Output:

```
HelloUGo multithreadeds, world!
Hello, worle
G goo mod design!
Used!
He ugood llo, wordesiltithreagn!
Use gldo!
Helded
olo, worlddG!
Hel dlo o, wormulld!
tithreaded
Go multithreesign!
Use good desaded
Go mign!
Use good design!
ultithreaded

Program done.
```

Multithreaded Printing, *cont'd*



PROTECTED SHARED RESOURCE
print stream

With a mutex guarding the critical regions of the threads, only one thread at a time can print to the print stream, which is the shared resource. When a thread wants to enter its critical region, it attempts to lock the mutex. If it succeeds, it can enter its critical region and print. If it doesn't, it blocks until another thread unlocks the mutex. Then, the thread can attempt to lock the mutex again. A thread must unlock the mutex before it exits its critical region. When the mutex is unlocked, the runtime system determines which blocked thread can unlock it.

Multithreaded Printing, *cont'd*

16.2a/main.cpp

```
void print(string text)
{
    for (int i = 0; i < COUNT; i++)
    {
        print_mutex.lock();
        {
            for (const char &ch : text) cout << ch;
        }
        print_mutex.unlock();
    }
}
```

Output:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Go multithreaded!
Hello, world!
Use good design!
Use good design!
Use good design!
Use good design!
Use good design!
Go multithreaded!
Go multithreaded!
Go multithreaded!
Go multithreaded!

Program done.
```

Multithreaded Printing, *cont'd*

- The C++ runtime automatically switches among the spawned child threads.
  - Switching among threads is called context switching.
  - Whether the threads execute concurrently or in parallel (or a combination of both) is determined by the operating system.

- We can explicitly cause a context switch by calling `this_thread.yield()` to tell the currently executing thread to relinquish control to another thread.

Multithreaded Printing, *cont'd*

16.2b/main.cpp

```
void print(string text)
{
    for (int i = 0; i < COUNT; i++)
    {
        print_mutex.lock();
        {
            for (const char &ch : text) cout << ch;
        }
        print_mutex.unlock();
        this_thread::yield();
    }
}
```

Output:

```
Hello, world!
Hello, world!
Use good design!
Use good design!
Use good design!
Hello, world!
Hello, world!
Hello, world!
Go multithreaded!
Use good design!
Use good design!
Go multithreaded!
Go multithreaded!
Go multithreaded!
Go multithreaded!

Program done.
```

Slightly more intermixed thread output.

Lock Guard

- A thread that successfully locks a mutex can enter its critical region, and it must unlock the mutex before exiting the critical region.
  - Forgetting to unlock will prevent other threads from locking the mutex, possibly causing a deadlock.

- Wrap a mutex as a lock guard.
  - It automatically locks when constructed.
  - It automatically unlocks when destructed, such as when it goes out of scope.

```
lock_guard<mutex> print_lock(print_mutex);
```

Lock Guard, *cont'd*

16.2c/main.cpp

```
void print(string text)
{
    for (int i = 0; i < COUNT; i++)
    {
        lock_guard<mutex> print_lock(print_mutex);
        {
            for (const char &ch : text) cout << ch;
            this_thread::yield();
        }
    }
}
```

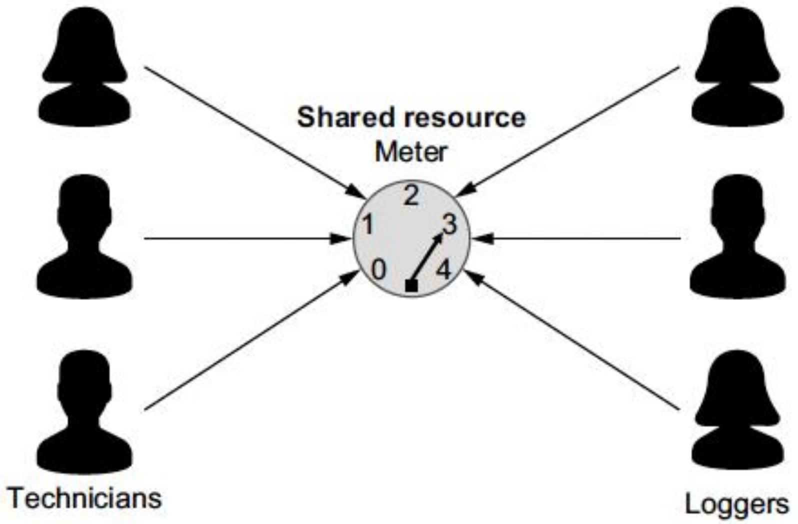
(I like to put the statements guarded by the lock guard in braces for emphasis. That's not necessary.)

Thread Coordination at a Higher Level

- Some multithreaded applications must coordinate the operations of the threads at a higher level as they enter and leave their critical regions.
- Without this higher level of coordination, the shared resources are not properly protected.

The Classic Reader-Writer Problem

- Multiple technicians (the writers) each attempts to set the meter's value.
- Meanwhile, several loggers (the readers) each attempts to read and log the meter's current setting.
- What is the shared resource?
- What are the critical regions?



Three technicians simultaneously attempt to set the meter's value; meanwhile, three loggers attempt to read and log the meter's current setting. Only one technician is allowed to be setting the meter at a time. No logger is allowed to read the meter while a technician is setting it. However, unless a technician is setting the meter, multiple loggers are allowed to be reading it at the same time. But while a logger is reading the meter, no technician is allowed to be setting it.

Reader-Writer Problem: Writers

- Only one technician thread can set (write to) the meter at a time.
  - Assume that the process of setting the meter to level *n* takes *n* seconds.
  - This process cannot be interrupted.

- While a technician thread is in the process of setting the meter:
  - No other technician thread can set the meter.
  - No logger thread can log the current value of the meter.

Reader-Writer Problem: Readers

- However, <u>multiple</u> logger threads can be logging (reading) the current value of the meter.
  - This is allowed because reading the meter doesn’t change its state.
  - Assume the process to read a meter takes 1 second.
- While any logger thread is logging the meter, no technician thread can be setting the meter.
- No logger thread can start to log the meter when a technician thread is setting the meter.

Reader-Writer Problem: Locks

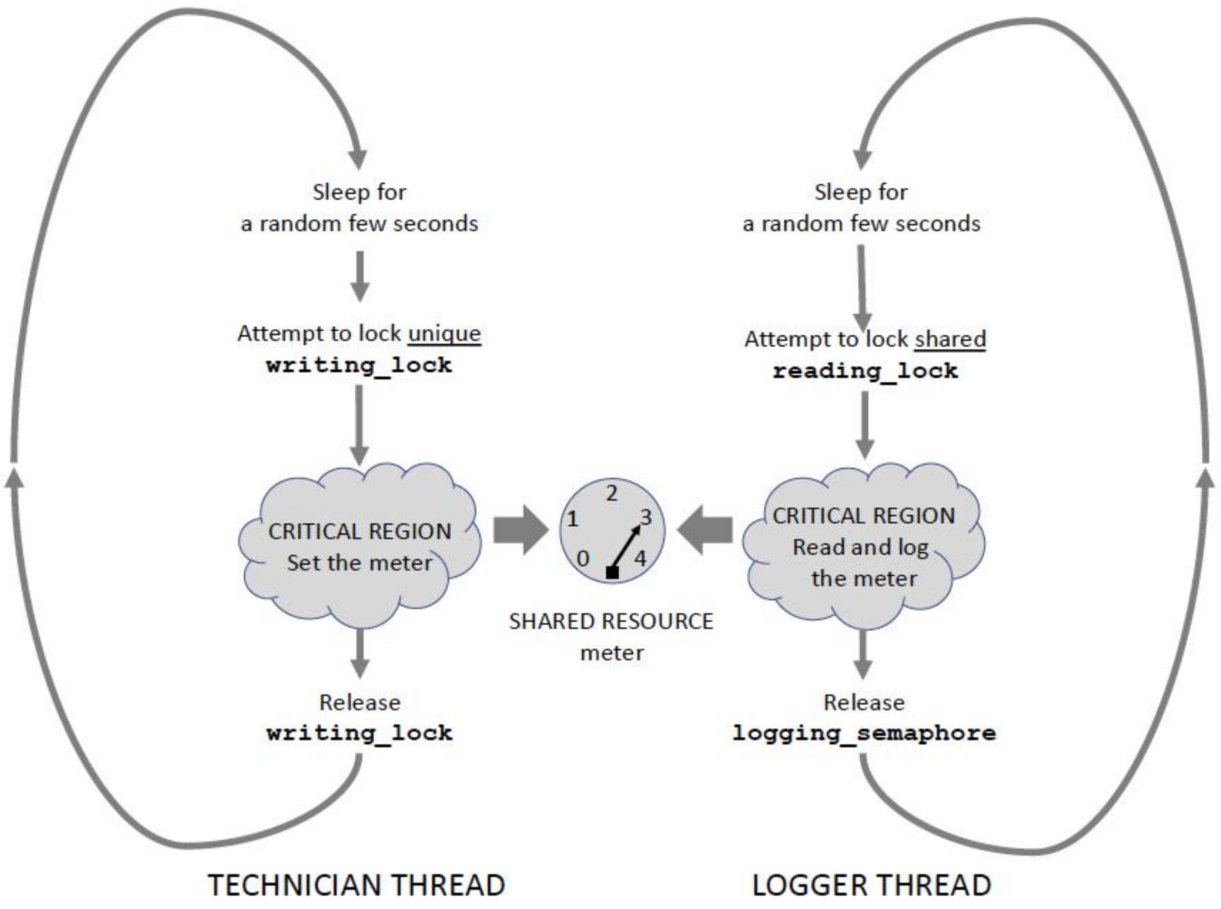
- Use a shared mutex to protect the shared resource, the meter. `shared_mutex meter_mutex;`

- Wrap the mutex as a unique lock to guard a critical region by allowing only one thread to lock it and no threads to lock a shared lock.

```
unique_lock<shared_mutex> writing_lock(meter_mutex);
```

- Wrap the mutex as a shared lock to guard a critical region by allowing multiple threads to lock it, but no other threads can lock the unique lock. `shared_lock<shared_mutex> reading_lock(meter_mutex);`

The Classic Reader Writer Problem, *cont'd*


Sleep for a random few seconds
Attempt to lock unique writing_lock
CRITICAL REGION Set the meter
Release writing_lock
TECHNICIAN THREAD
2
1
3
0
4
SHARED RESOURCE meter
Sleep for a random few seconds
Attempt to lock shared reading_lock
CRITICAL REGION Read and log the meter
Release logging_semaphore
LOGGER THREAD

The technician threads and the logger threads each loop and attempt to enter their critical regions to access the meter, which is the shared resource. The unique lock prevents another thread from setting or reading the meter. The shared lock allows multiple logger threads to read and log the meter while keeping out a technician thread.

16.3/Meter.cpp

Technician

```
void Meter::set_meter(const int thread_id)
{
    for (int turn = 1; turn <= TECHNICIAN_TURNS; turn++)
    {
        int sleep_time = rand()%(  MAX_TECHNICIAN_SLEEP_TIME
                                 - MIN_TECHNICIAN_SLEEP_TIME)
                              + MIN_TECHNICIAN_SLEEP_TIME;
        this_thread::sleep_for(chrono::seconds(sleep_time));

        {
            unique_lock<shared_mutex> writing_lock(meter_mutex);
            {
                printf("%02d TECH #%d:", elapsed_seconds(), thread_id);

                setting = rand()%MAX_SETTING_LEVEL + 1;

                for (int n = 1; n <= setting; n++)
                {
                    this_thread::sleep_for(
                            chrono::seconds(TECHNICIAN_SET_TIME));
                    printf("%2d", n); cout.flush();
                }
                cout << endl;

                ok_to_log = true;  // allow logger threads to operate

                if (turn == TECHNICIAN_TURNS)
                {
                    printf("%02d TECH #%d: done!\n",
                            elapsed_seconds(), thread_id);

                    active_technicians_count--;
                }
            }
        }
    }
}
```

When a **unique_lock** is successfully locked, no other thread can lock it or lock a **shared_lock**.

One second per setting level.

A shared lock automatically unlocks when it goes out of scope (like a lock guard).

The Classic Reader Writer Problem: Logger

16.3/Meter.cpp

```
void Meter::log_meter(const int thread_id)
{
    while (active_technicians_count > 0)
    {
        this_thread::sleep_for(
            chrono::seconds(rand()%MAX_LOGGER_SLEEP_TIME + 1));

        // OK to log after the meter is set the first time.
        if (ok_to_log)
        {
            shared_lock<shared_mutex> reading_lock(meter_mutex);
            {
                lock_guard<mutex> printing_lock(print_mutex);
                {
                    printf("%*s%02d LOGGER #%d: logging %d\n",
                           LOGGER_PRINT_MARGIN, " ",
                           elapsed_seconds(), thread_id, setting);
                }
            }
        }
    }
}
```

When a **shared_lock** is successfully locked, other threads can also lock it, but no other thread can be locked with a **unique_lock**.

Why the lock guard for the **printf()**?

A shared lock automatically unlocks when it goes out of scope (like a lock guard).

A Multithreading Analogy: The Long Train Ride

- You are on a long train ride.
- You don’t want to miss your final stop.

- Solutions?
  - Stay awake the whole time.
  - Take short naps and check each time you wake up.
  - Set your phone to alarm you shortly before arrival.

- Problems with these solutions?
- Is there a better solution?

Break

Atomic Variables

- A C++ atomic variable enables thread-safe operations on the variable in a multithreaded program without the use of mutexes and locks.
  - Example: An atomic integer variable:

    ```
    atomic<int> active_producers_count;
    ```

  - To set its value:

    ```
    active_producers_count.store(producer_count);
    ```

  - To access its value:

    ```
    active_producers_count.load()
    ```

  - Because it’s atomic, you can perform operations on the variable without using mutexes and locks:

    ```
    --active_producers_count;
    ```

A Multithreading Analogy: The Long Train Ride

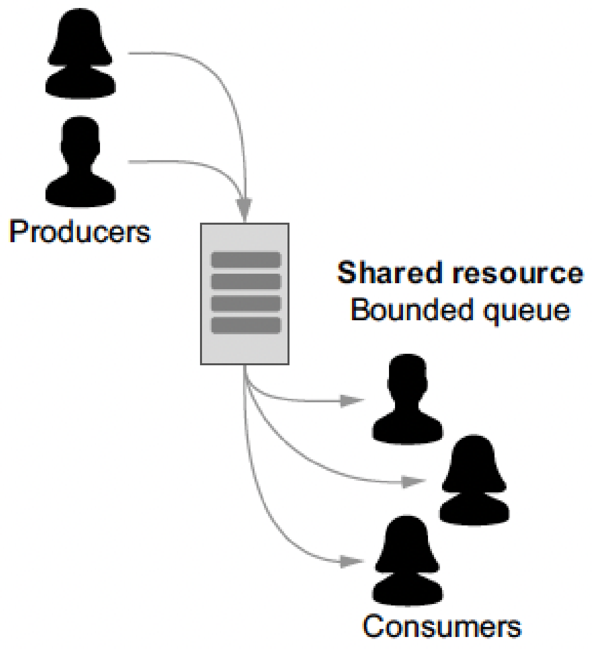
- You are on a long train ride.
- You don’t want to miss your final stop.

- Solutions?
  - Stay awake the whole time.
  - Take short naps and check each time you wake up.
  - Set your phone to alarm you shortly before arrival.

- A better solution:
  - Go ahead and sleep.
  - Have someone wake you up at your final stop.

The Classic Producer-Consumer Problem

- Some multithreaded applications require greater <u>synchronization</u> among their threads.
  - Mutexes and locks alone may not be sufficient.
- Example: The producer–consumer application has multiple <u>producer threads</u> and <u>consumer threads</u>.
  - Simultaneously, the <u>producer threads</u> enter values into a data container such as a queue, and the <u>consumer threads</u> remove values from the queue.
  - The queue has <u>limited capacity</u>. It can only hold a certain number of elements.



Producer-Consumer Problem, *cont'd*

- Shared resource ???
- Critical regions ???

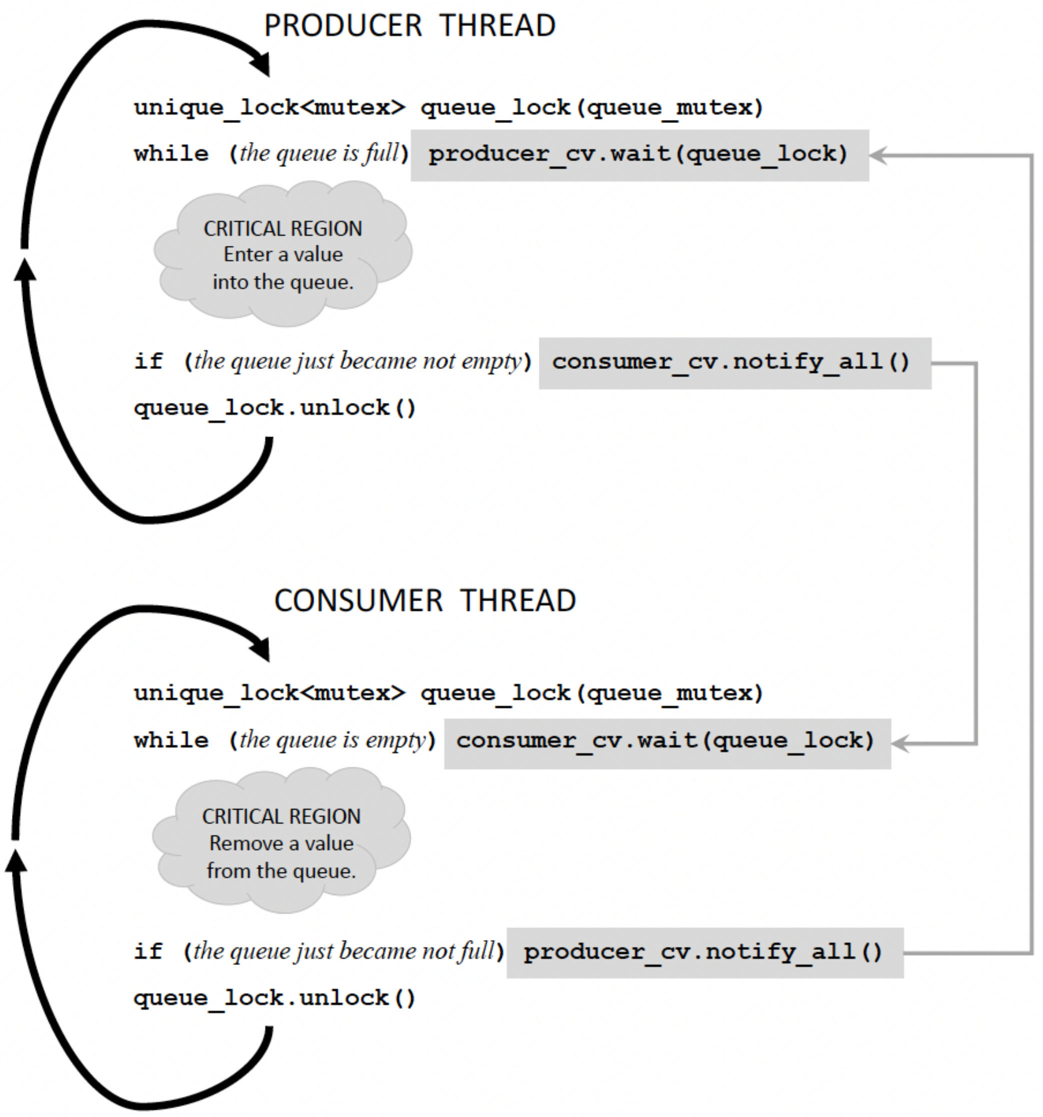
- We need a mutex to guard the critical regions.

- Queue state
  - Assume that function `data.size()` returns the number of values currently in the shared queue and that constant `CAPACITY` is the limited capacity of the queue.

Producer-Consumer Problem, *cont'd*

- Synchronization is needed!
  - `data.size() == CAPACITY`: If the queue is <u>currently full</u>, the <u>producer threads must wait</u> until the queue is no longer full before entering values.
  - `data.size() == 0`: If the queue is <u>currently empty</u>, the <u>consumer threads must wait</u> until the queue is no longer empty before removing values.
  - `data.size() == 1`: If the queue <u>just became not empty</u>, that is, a producer thread entered a value into the queue that was empty, that producer thread must <u>notify all the consumer threads</u> that were waiting for the queue to become not empty.
  - `data.size() == CAPACITY-1`: If the queue <u>just became not full</u>, that is, a consumer thread removed a value from the queue that was full, that consumer thread must <u>notify all the producer threads</u> that were waiting for the queue to become not full.

Condition Variables for Synchronization



PRODUCER THREAD

```
unique_lock<mutex> queue_lock(queue_mutex)
while (the queue is full) producer_cv.wait(queue_lock)
```

```
if (the queue just became not empty) consumer_cv.notify_all()
queue_lock.unlock()
```

CONSUMER THREAD

```
unique_lock<mutex> queue_lock(queue_mutex)
while (the queue is empty) consumer_cv.wait(queue_lock)
```

```
if (the queue just became not full) producer_cv.notify_all()
queue_lock.unlock()
```

The producer and consumer threads each creates a local unique lock object **`queue_lock`** from **`queue_mutex`** to guard its critical region.

The two condition variables, **`producer_cv`** and **`consumer_cv`**, work together to synchronize the producer and consumer threads.

The producer threads loop to enter values into the shared queue, and the consumer threads loop to remove values from the shared queue.

A producer thread cannot enter a value into the queue when the queue is full. A consumer thread cannot remove a value from an empty queue.

```
void Producer::produce(const int thread_id)
{
    condition_variable& consumer_cv = consumer->get_cv();

    for (int i = 0; i < producer_times; i++)
    {
        this_thread::sleep_for(chrono::seconds(rand()%producer_rate));

        {
            unique_lock<mutex> queue_lock(shared_queue.get_mutex());
            {
                while (shared_queue.is_full()) producer_cv.wait(queue_lock);

                ++data_value;
                shared_queue.enter(data_value.load());

                print_spaces(thread_id);
                printf("%4d", data_value.load());
                shared_queue.print();

                if (shared_queue.just_became_not_empty()) consumer_cv.notify_all();
            }
        }
    }

    --active_producers_count;

    print_spaces(thread_id);
    printf("   Done!\n");

    if (active_producers_count.load() == 0) consumer_cv.notify_all();
}
```

The producer attempts to enter data into the queue.

Attempt to lock the mutex.

Wait until the queue is no longer full.

Enter data into the queue.

Notify waiting consumers that the queue is no longer empty.

All the producers are done. Notify any waiting consumers.

The consumer attempts to remove data from the queue.

```
void Consumer::consume(const int thread_id)
{
    condition_variable& producer_cv = producer->get_cv();

    while (true)
    {
        this_thread::sleep_for(chrono::seconds(rand()%consumer_rate));

        {
            unique_lock<mutex> queue_lock(shared_queue.get_mutex());
            {
                while (   shared_queue.is_empty()
                       && (producer->get_active_count() > 0))
                {
                    consumer_cv.wait(queue_lock);
                }

                if (shared_queue.is_empty()) break;

                int value = shared_queue.remove();

                print_spaces(thread_id);
                printf("%4d", -value);
                shared_queue.print();

                if (shared_queue.just_became_not_full()) producer_cv.notify_all();
            }
        }
    }

    --active_consumers_count;

    print_spaces(thread_id);
    printf("   Done!\n");
}
```

Attempt to lock the mutex.

Wait until the queue is no longer empty.

Are we done?

Remove data from the queue.

Notify waiting producers that the queue is no longer full.

Condition Variable Operation

- Uniquely lock `queue_lock`.

```
unique_lock<mutex> queue_lock(shared_queue.get_mutex());
```

- The `wait()` call:

```
while (shared_queue.is_full())
{
    producer_cv.wait(queue_lock);
}
```

  - Check the `while` condition.
  - If the condition is true, unlock `queue_lock`.
  - To recheck the condition, first relock `queue_lock`.
  - If the condition is false, break out of the `while` loop with the `queue_lock` locked.

Final Notes on Multithreading

- It’s a major challenge to design and develop a multithreaded application.

- Because the runtime system usually determines the order that spawned child threads start executing, timing issues can cause random and irreproducible behaviors from one run of the application to another.
  - Debugging then becomes a nightmare.

- A deadlock can occur when all the threads are blocked, and the entire application hangs.

Final Notes on Multithreading, *cont'd*

- Increasing the number of threads does not mean that an application's execution time approaches zero!
  - Mutexes, locks, and condition variables all have their own performance costs.
  - The costs of context switching can overwhelm the benefit of having more threads.

- Designing a multithreaded application can be a <u>delicate balancing act</u> if the goal is to increase performance without increasing complexity.
  - What applications are suitable for multithreading?

The `auto` Keyword

- In a declaration of a variable that is also being initialized, the compiler can infer the type of the variable from the initialization expression.
  - type inference, AKA type determination

- Use `auto` instead of a complicated type name.
  - Examples: Instead of:

```
steady_clock::time_point start_time = steady_clock::now();
```

Use:

```
auto start_time = steady_clock::now();
```

From the initialization expression `steady_clock::now()` the compiler can infer that `start_time` has type `steady_clock::time_point`.

The **decltype** Pseudo-Function

- **decltype** takes a variable as an argument.
- Returns the <u>type</u> associated with the variable.

- Use to create another variable
  with the <u>same type</u>.
  - Ensure that two variables have the same type.
  - Example:

```
auto start_time = steady_clock::now();
decltype(start_time) end_time;
```

Assignment #2: Multithreading Experiments

- You will perform some experiments with Assignment #2 (team-based).
- Write a <u>non-recursive</u> version of Quicksort.
  - Initialize a <u>stack</u> by pushing the bounds (leftmost and rightmost index values) of the entire array.
  - Pop off the bounds of a subarray to partition it.
  - After partitioning, instead of making a recursive call on the two subarrays, push the bounds of the two subarrays onto the stack.
  - Repeat the pushing and popping until the entire array is sorted.

Assignment #2, *cont'd*

- During a Quicksort, subarrays don't overlap and therefore, they're processed independently.

- Using the <u>non-recursive</u> version of Quicksort:
  - Create a pool of threads.
  - Each thread pops the bounds of a subarray off the stack, partitions it, and pushes the bounds of the two subarrays onto the stack.
  - The threads all work simultaneously on different parts of the entire array.

- How does multithreading affect performance?