CS11 Advanced C++

Winter 2020 – Threads

C++ Threading: Before C++11

- Before C++11, language specification didn't say anything about multithreaded programs
- No guarantee of portability of multithreaded programs across multiple platforms
- Rely on OS threading support (e.g. POSIX pthreads, Windows threads)
- For more advanced multithreading programs, e.g. lock-free data structures
  - Often need assembly language snippets and other language tricks to make code behave properly
- Not ideal, but definitely usable

C++ Threading: C++11 and Later

- C++11 adds language support for multithreading
  - Language specification is now written from the perspective of multithreaded programs
- C++ Standard Library includes both basic facilities and higher-level abstractions for multithreaded programs
  - Thread, mutex, condition variable classes
  - Helper classes and functions for managing them
  - Atomic data types with various memory-ordering guarantees
  - Other abstractions, e.g. async. tasks and futures

C++ **`std::thread`** Class

- The **`std::thread`** class is a wrapper around an operating-system thread

```
#include <thread>
```

- Constructor takes a function and a set of arguments
  - A new thread is started, then the thread calls the function with the arguments

- Thread terminates when the function returns

`thread` Constructor

- Constructor is a function template:

```
template <class Fn, class... Args>
explicit thread(Fn&& fn, Args&&... args)
```

- **`Fn`** can be a pointer to a function, or any object that implements **`operator()`**
  - i.e. anything that can be called like a function
- **`Args`** is a **parameter pack**
  - C++11 added **variadic templates** to the language
  - Allows the **`thread`** constructor to take as many arguments as **`Fn`** requires

`thread` Constructor (2)

- Constructor is a function template:

```
template <class Fn, class... Args>
explicit thread(Fn&& fn, Args&&... args)
```

- Constructor is marked **`explicit`** to prevent **implicit conversions** from functions into threads
  - Cases where the function **`Fn`** takes no arguments
  - e.g. without **`explicit`**, this would start a thread:

```
thread t = rand;
```

- All arguments to the thread constructor must also be **move-constructible**
  - They must have a move constructor or copy constructor

`thread` Constructor (3)

- **`std::thread`** also has a default constructor
- The default-initialized **`thread`** object doesn't represent a thread of execution
- You can't do much with the object except assign another **`thread`** to it (more on this in a moment)

`thread` Copy/Move Operations

- **Only one `std::thread` object may represent a given OS-level thread of execution**
  - It doesn't make sense for two **`std::thread`** objects to represent the same OS-level thread
- **`std::thread`** does not provide either a copy constructor or a copy-assignment operator
- **`std::thread`** defines a move constructor

  `thread(thread &&x)`
  - Allows us to create functions that return thread objects, store thread objects into collections, etc.

`thread` Copy/Move Operations

- Similarly, **`std::thread`** defines a move-assignment operator
  - Ensures that only one thread object represents each thread of execution
- Example:

```
thread t1(f, a, b, c); // Calls f(a, b, c)
thread t2;

t2 = t1; // Contents of t1 are moved to t2!
         // After assignment, t1 is empty!
```

Joining and Detaching Threads

- Can wait for a thread to terminate by calling its **`join()`** member-function
  - **`join()`** doesn't return until the thread of execution terminates
- Also, threads hold some system resources that must be reclaimed at termination
  - A stack, exit status, etc.
- Calling **`join()`** causes these resources to be reclaimed
- If a program doesn't care when a thread terminates, can call **`detach()`** on it
  - Cannot call **`join()`** on the thread after **`detach()`**ing it
  - The thread's resources are cleaned up automatically

Joinable Threads

- Can only call **`join()`** on threads that are **joinable**
  - The **`joinable()`** method reports if it is joinable
- A **`thread`** object is not joinable if:
  - The **`thread`** object was initialized via the default ctor, and no other **`thread`** object was moved into it
  - The **`thread`** object's contents were moved into another **`thread`** object
  - Either **`join()`** or **`detach()`** has been called on the **`thread`** object
- **If a `thread` object is joinable when its destructor is called, your program will be terminated!**

Removing Copy Constructors

- Before C++11, the usual way to remove special member functions was to make them private
  - e.g. copy constructor, assignment operator, destructor, new operator, etc.

```
class NotCopyable {
    NotCopyable(const NotCopyable &) { }
public:
    ...  // Other stuff
};
```

- Attempts to copy objects of this type will produce a compile error about accessing private members

Removing Copy Constructors (2)

- C++11 introduces a new way to remove special functions:

```
class NotCopyable {
public:
    NotCopyable(const NotCopyable &) = delete;
    ...
};
```

- Can apply this to copy and/or move constructors, copy and/or move assignment operators

Threads and Functions that Throw

- What happens if a thread's function throws?

```
void f() {
    throw exception();
}

// Start a thread running f
thread t(f);
```

- In C++, if an exception is unhandled, the runtime calls the **`std::terminate()`** function
  - Default impl calls **`abort()`** – terminates the process
  - (you can also replace default **`terminate()`** handler)
- **If a thread's function throws an exception, the default behavior is to kill your entire program!**

The “Current Thread”

- C++ also has a **`std::this_thread`** namespace
  - Provides several helper functions for performing operations on the current thread
- Suspend the thread for a period of time:

```
void this_thread::sleep_for(
    const std::chrono::duration &duration)
```

  - Sleep for a certain duration

```
void this_thread::sleep_until(
    const std::chrono::time_point &time)
```

  - Sleep until a specific point in time

The “Current Thread” (2)

- Can also yield the processor to another thread

  `void this_thread::yield()`

  - Allows other threads of execution to be scheduled on the processor
  - Can be helpful if the current thread has more work to do, but wants to give other threads a chance to progress as well

Synchronization Primitives

- C++11 Standard Library also includes mutexes and condition variables
- **Mutexes** are mutual-exclusion locks
  - Used to ensure exclusive access to shared resources
- **Condition variables** can be used to passively wait for some condition to become true
  - Another thread can notify threads waiting on a condition variable, to wake them up
  - Can also timeout, in case condition doesn't become true within a specified time limit

Mutexes

- C++ provides several different kinds of mutexes

  `#include <mutex>`

- **`std::mutex`**
  - **`void lock()`** – locks the mutex, blocking if not available
  - **`bool try_lock()`** – attempts to lock the mutex, but returns immediately if it is not available
    - Returns **`true`** if lock acquired; **`false`** otherwise
  - **`void unlock()`** – unlocks a previously locked mutex
- Mutex cannot be assigned or copied (makes sense)
- **This class <u>does not</u> support multiple "recursive" (i.e. nested) lock requests from the same thread!**
  - May deadlock, or may cause an exception

Mutexes (2)

- **`std::recursive_mutex`**
  - Same set of member functions as **`std::mutex`**
- Supports multiple nested ("recursive") lock requests from the same thread
  - Implementation is slightly more complex than **`std::mutex`**, to detect recursive lock requests
- Ownership of mutex extends from first **`lock()`** call to last matching **`unlock()`** call
  - Nested calls to **`lock()`** and **`unlock()`** won't cause deadlocks or errors

Mutexes (3)

- Also **`timed_mutex`** and **`recursive_timed_mutex`**
- These classes support lock-requests with a timeout

- **`bool try_lock_for(`**
  `const std::chrono::duration &duration)`
  - Try to acquire lock for a certain duration, then give up
- **`bool try_lock_until(`**
  `const std::chrono::time_point &timeout_time)`
  - Try to acquire lock until a specific point in time, then give up

Managing Mutexes

- Mutexes can present some management issues
- Example: countdown latch (a.k.a. barrier)
  - For coordinating a group of cooperating threads
- Initialize the latch with a number of threads
- When threads call **`count_down()`**:
  - Count of remaining threads is decremented
  - Calling thread is blocked until the latch's count reaches 0
- No threads in the group are allowed to progress until all threads have checked in on the latch

Managing Mutexes (2)

```
class CountdownLatch {
    // How many threads still need to check in?
    int count;
    mutex m;
public:
    CountdownLatch(int count) : count{count} {}

    void count_down() {
        m.lock();
        if (count <= 0)
            throw invalid_usage();
        count--;

        ...  // Somehow, wait for count to hit 0
        m.unlock();
    }
};
```

Need to unlock mutex before throwing!

Managing Mutexes (3)

- C++ provides classes for managing mutex locks based on variable scoping
- Recall: when an object goes out of scope, its destructor is called
  - Allows object to release any resources it holds
  - Allows us to implement the "Resource Acquisition Is Initialization" (RAII) pattern
  - This is used to implement "smart pointers" which automatically delete the memory they point to
- Idea: Why not manage mutexes in the same way?
  - A wrapper class that represents an acquired lock
  - When the object goes out of scope, it calls **`unlock()`**

Managing Mutexes (4)

- The **`std::lock_guard<class Mutex>`** template manages a mutex-lock
  - Also declared in **`<mutex>`** header
  - You can specify the kind of mutex being managed in the template parameter
- **`lock_guard(Mutex &m)`** constructor acquires a lock on **`m`**
- When the **`lock_guard`** object passes out of scope, **`m.unlock()`** is called automatically
- Can use this class to implement lock management that is correct even in the context of exceptions

Managing Mutexes (5)

- Our countdown latch, revisited:

```
void count_down() {
    lock_guard<mutex> lock(m);
    if (count <= 0)
        throw invalid_usage();
    count--;

    ...  // Somehow, wait for count to hit 0
}
```

lock goes out of scope, calls **m.unlock()** automatically

- Now, when **lock** goes out of scope, **m** is unlocked automatically! Even when exceptions are thrown!

Managing Mutexes (6)

- **`std::lock_guard`** is for very simple lock/unlock scenarios

- The **`std::unique_lock`** wrapper is a much more general-purpose lock wrapper
  - It also ensures that the mutex will be released when the wrapper goes out of scope
  - Also exposes most mutex operations directly on the wrapper object

Waiting for All Threads

- Need to wait for all threads to call **`count_down()`**

```
void count_down() {
    lock_guard<mutex> lock(m);
    if (count <= 0)
        throw invalid_usage();
    count--;

    ...  // Somehow, wait for count to hit 0
}
```

- Need to unlock the mutex while we wait, so other threads can also call **`count_down()`**
- **`lock_guard`** can't do this, but **`unique_lock`** can

Waiting for All Threads (2)

- Need to wait for all threads to call **`count_down()`**

```
void count_down() {
    unique_lock<mutex> lock(m);
    if (count <= 0)
        throw invalid_usage();
    count--;

    // Wait for count to hit 0
    while (count > 0) {
        lock.unlock();
        std::this_thread::yield();
        lock.lock();
    }
}
```

All access to count is guarded by the locked mutex

Current thread yields the CPU to other running threads between checks

Waiting for All Threads (3)

- Problem: This code **actively waits** for a condition

```
void count_down() {
    unique_lock<mutex> lock(m);
    if (count <= 0)
        throw invalid_usage();
    count--;

    // Wait for count to hit 0
    while (count > 0) {
        lock.unlock();
        std::this_thread::yield();
        lock.lock();
    }
}
```

  - A very expensive way to wait on other threads
  - Wastes CPU time that other threads could use to do actual work!
  - All threads blocked on the latch must actively wait on the latch

Passive Waiting

- Should almost never require active polling like this!
  - A rare exception is when interacting directly with hardware, e.g. in an embedded system
- Strongly prefer **passive waiting**: the thread is suspended while waiting for some condition to occur
- The waiting thread doesn't know when the condition will occur…
  - Another thread will eventually do something, which allows a waiting thread to proceed
  - Need a way for one thread to notify the other, when it is capable of proceeding
- Can implement this with **condition variables**

Passive Waiting (2)

- Condition variables allow threads to be notified when some condition becomes true
  - The specific condition depends on the program's needs
  - The condition variable is simply a synchronization primitive
- Condition variables must be used in the context of a mutex
  - The two communicating threads are manipulating shared state…
  - The mutex protects that shared state

Waiting Threads

- Need to have both a mutex and a condition variable to use:

```
#include <mutex>
#include <condition_variable>
mutex m;
condition_variable cv;
```

- Waiting thread can wait on the condition variable, after it acquires a lock on the mutex

```
unique_lock<mutex> lock(m);
cv.wait(lock); // Suspends the calling thread
```

- Must use a **`unique_lock`** with condition variables!
  - Allows the condition variable to manipulate the mutex
  - Mutex is unlocked right before the thread suspends, so that other threads can acquire the mutex and access the shared state
  - Mutex will be re-locked before **`cv.wait(lock)`** returns

Notifying Threads

- A notifying thread can use the condition variable to notify one waiting thread, or all waiting threads

```
cv.notify_one();
cv.notify_all();
```

  - If no threads are waiting, these are no-ops
- Note: no mutex-lock is required when notifying
  - Usually, the notifier will also hold a mutex when notifying
- Only call **`notify_all()`** when multiple threads can progress! Otherwise it's just inefficient.
  - If only one thread can progress, call **`notify_one()`** instead

Countdown Latch, Revised

- Add a condition variable to our countdown latch

```
class CountdownLatch {
    // How many threads still need to check in?
    int count;
    mutex m;
    condition_variable cv_done;
    ...
};
```

- Use the condition variable to wait passively until all threads have entered the latch
  - When all have entered the latch, wake everybody up!
  - The last thread to enter will have this responsibility

Countdown Latch, Revised (2)

- Update our **`count_down()`** function to passively wait:

```
void count_down() {
    unique_lock<mutex> lock(m);
    if (count <= 0)
        throw invalid_usage();
    count--;

    if (count == 0) {
        // Last thread to enter latch; notify everyone!
        cv_done.notify_all();
    }
    else {
        // Not the last thread to enter latch; wait to
        // be waken up by the last thread to enter.
        while (count > 0)
            cv_done.wait(lock);
    }
}
```

Countdown Latch, Revised (3)

- *Why do we wait in a loop?!*

```
void count_down() {
    ...
    else {
        // Not the last thread to enter latch; wait to
        // be waken up by the last thread to enter.
        while (count > 0)
            cv_done.wait(lock);
    }
}
```

- Some condition-variable implementations may *spuriously wake up* waiting threads
- **<u>Always</u> wait in a loop, until the required condition <u>actually</u> becomes true.**
  - Don't assume that just because the thread woke up, the condition is going to be true.

C++ Multithreading

- C++ includes many other multithreading facilities
- Threads, mutexes and condition variables are sufficient for most multithreaded programs
- Nonetheless, additional facilities also exist, when you need them
  - Atomic types, futures, async, etc.