C++ Concurrent Programming

Peter Chapin

CIS-3012, C++ Programming

Vermont State University

Thread?

- A *thread of execution* (or just *thread*) is the sequence of statements executed by the processor (or *processing element*)
- In a *single-threaded* program, there is just one thread
  - It calls `main` and follows the flow of the program until it exits, causing the program to end
- In a *multi-threaded program*, there are multiple threads
  - The main thread is the one that calls `main`
  - During its lifetime, the main thread might start other threads
  - Each additional thread calls a top-level function for that thread called the *thread function*

Program Termination

- The program ends when the main thread returns from `main`
  - This is true even if there are other threads active. They are aborted at once
- The program ends if any thread calls std::exit
  - This is true even if there are other threads active. They are aborted at once
- The program ends if any thread throws an exception that it does not handle
  - This is true even if there are other threads active. They are aborted at once
- In general, *it is best to arrange for the clean termination of all threads before trying to terminate the program*

Single-Threaded

- In a single-threaded process:
  - The OS starts the main thread
  - The main thread calls constructors of global objects
  - The main thread calls `main`
  - **The program is executed**
  - The main thread returns from main
  - The main thread calls destructors of global objects
  - The main thread informs the OS that the process has ended

An exception here might skip the return from `main`.
However, the destructors of global objects still get called.

Multi-Threaded

- In a single-threaded process:
  - The OS starts the main thread
  - The main thread calls constructors of global objects
  - The main thread calls `main`
  - Additional threads get created by the main thread (or by each other)
  - **The program is executed**
  - Wait for all additional threads to cleanly terminate
  - The main thread returns from main
  - The main thread calls destructors of global objects
  - The main thread informs the OS that the process has ended

Unhandled Exceptions

- Because an unhandled exception in a thread will terminate the entire program, consider catching all exceptions in the thread function

```
// This function is the top-level function of some thread
void f( int x, int y )
{
    try {
        // The main logic of the thread
    }
    catch( ... ) {
        // The thread tried to throw an unhandled exception.
        // Log the event, and let the thread end normally?
    }
    // The thread ends when this function returns
}
```

Processor Stack

- Every thread has its own stack. This means:
  - Local variables are unique to the thread, even if two threads execute the very same function (local variables on on the stack)
  - When an exception is thrown, it is the stack of the throwing thread that is unwound
  - An exception can be happening in one thread while other threads are executing normally. This does not (necessarily) cause any problems
  - The stack could overflow in one thread (causing UB), but not the others

Global Variables

- Local variables (and function parameters) *are not* shared between threads, even when two threads execute the same function (every thread has its own stack)
- Global variables *are* shared between threads
- Heap data is (potentially) shared between threads
  - That is, objects allocated with **new** can potentially be accessed by multiple threads, if pointers to such objects are shared
- *Thread-local storage* is global storage that is only visible to a particular thread. **Outside the scope of these slides**

Debugging Multi-Threaded Programs

- Is hard!
  - By default, most debuggers will stop only one thread. The other threads run at full speed as you single-step through the program.
  - You typically can stop all threads, and switch between them manually to single step each one
  - A breakpoint will likely stop the thread that hits it, but not the others, although your debugger may give you the option to stop all threads when any of them breaks
- Interpreting what is going on can be very difficult

Debugging Multi-Threaded Programs

- Is **very** hard!!
  - Many thread related errors arise because of timing problems between the threads (called *race conditions*)
  - Unfortunately, thread timing is non-deterministic and extremely difficult (aka impossible) to reproduce at will
  - A problem that is reasonably reliable in the deployed system may go away when you try to debug because of changes in thread timing
  - Even adding a debugging print will change the relative execution speed of the threads and can mask bugs
- Many multi-threaded programs are deployed with bugs like these!

What Does It Look Like?

```
// FILE: main.cpp

#include <thread>

extern void f( int x );

int main( )
{
    // Start a thread, passing 42 to f
    std::thread t( f, 42 );

    // Do other things while f executes

    // Wait for the thread to end
    t.join( );
    return 0;
}
```

```
// FILE: helper.cpp

void f( int x )
{
    // Code executed by thread
}
```

If an exception is thrown, the call to `join` gets skipped. That might be undesirable

Using C++ 2020 `std::jthread`

```
// FILE: main.cpp

#include <thread>

extern void f( int x );

int main( )
{
    // Start a thread, passing 42 to f
    std::jthread t( f, 42 );

    // Do other things while f executes
    return 0;
}
```

```
// FILE: helper.cpp

void f( int x )
{
    // Code executed by thread
}
```

The destructor of `jthread` calls `join`.
This will happen even if an exception propagates

Another Example

```
// FILE: main.cpp

#include <thread>

extern void f( int x, double y );

int main( )
{
    // Start a thread, passing 42 to f
    std::thread t( f, 42, 3.14 );

    // Do other things while f executes

    // Wait for the thread to end
    t.join( );
    return 0;
}
```

```
// FILE: helper.cpp

void f( int x, double y )
{
    // Code executed by thread
}
```

Variable number of arguments of variable types
This works because the constructor is a *variadic template*

Finish Me!

- Topics to include:
  - Returning values from threads
  - The `std::this_thread` name space
  - `std::mutex`
  - Lock guards
  - R/W locks
  - Condition variables
  - Futures and promises