C++ Standard Threads

Peter C. Chapin
Computer Information Systems
VERMONT TECH

April 8, 2015

Overview

- *Why?* Widespread agreement that C++ should have standardized thread handling facilities.
  - Many programs need threads.
  - Currently must use non-portable, OS-specific facilities.
  - Semantics of C++ 1998 with multiple threads unclear.
  - C++ 2011 contains thread classes roughly taken from Boost.

Basic Example

```
#include <thread>

void f( int count )
{
    ...
}

int main( )
{
    // Start thread, pass argument to thread function.
    std::thread my_thread( f, 42 );

    ... // Execute here and in f at the same time.

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

Notes On Basic Example

- Thread function requirements. . .
  - Return **`void`**.
  - Can take parameters as needed (any type).
  - Can be any callable object.
    - Pointer to function.
    - Class instance with overloaded **`operator`**`( )`.
- thread behavior. . .
  - Given callable object is *moved* (or else copied).
  - Destructor detaches (unless thread already joined).

Local Variables

Each thread has its own stack. Locals are allocated on the thread's stack; globals are shared between threads.

```
int x;  // Global data.

void f( int count )
{
    int temporary = count;  // Temporary local to the thread.
    x = temporary;          // X is shared by threads.
}

// No problem with two threads executing same function.
// Each thread has its own arguments and local variables.

std::thread thread1( f, 42 );
std::thread thread2( f, 84 );
```

Callable Class Instance

Possible to use a class instance as a callable.

```
struct Worker {
    void operator( )( int count );
};

void Worker::operator( )( int count )
{
    // Do stuff...
}

int main( )
{
    // Create an instance of a callable object.
    Worker callable;
    std::thread my_thread( callable, 42 );

    ... // Execute here and in Worker::operator( ).

    my_thread.join( );
    return 0;
}
```

Returning Values

- Thread callables must return `void`.
- Returning a value is tricky.
  - The parent thread is not waiting for the result; it is busy doing something else.
  - Return value must be stored somewhere.
- Three possibilities. . .
  - Global data (discouraged).
  - Use a pointer parameter (okay, can be tricky).
  - Members of a callable with class type (preferred).

Return Values: First Try

This doesn't quite work.

```
struct Worker {
    int result;
    void operator( )( int count );
};

void Worker::operator( )( int count ) {
    ...
    result = ...
}

int main( ) {
    Worker callable;
    std::thread my_thread( callable, 42 );
    ...
    my_thread.join( );
    // Look at callable.result here.
    return 0;
}
```

Returning Values: Second Try

Problem: The callable is *moved* (or else copied)

```
#include <functional>

int main( )
{
    Worker callable;
    std::thread my_thread( std::ref( callable ), 42 );

    ...

    my_thread.join( );
    // Now callable.result is meaningful.

    return 0;
}
```

The `std::ref` function returns a `reference_wrapper` that can be copied into the thread object and that forwards all access to the original callable.

Thread Copies

The `std::thread` class is not copyable. *What would copying a thread mean?*

But it is movable. Threads can be moved out of funtions (returned) and into "move aware" containers.

```
std::thread make_thread( )
{
    int x;

    // Do complex stuff to prepare thread arguments.
    return std::thread( f, x );
}
...

std::vector<std::thread> my_threads;
my_threads.push_back( make_thread( ) );
```

It's okay to use local variable as thread parameter because it is moved (or else copied) into the thread object.

Exceptions

If the callable passed to a thread thows an unhandled exception, `std::terminate` is called.

- Normally aborts the *entire* program.
- Probably should wrap top level thread function in a catch-all handler.

```
void thread_function( )
{
  try {
    // Do stuff.
  }
  catch( ... ) {
    // Unhandled exception reached here.
    // Do something reasonable.
    // Let function end without throwing.
  }
}
```

Hardware Concurrency

Can query library to find out how many hardware threads are available.

```
unsigned processor_count =
  std::thread::hardware_concurrency( );

if( processor_count == 0 ) {
  // Can't tell how many CPUs there are.
}
else {
  // Create as many threads as there are processors.
}
```

This is useful when doing parallel programming.

Namespace `this_thread`

Functions in `std::this_thread` affect only the calling thread.

Methods of a `std::thread` object affect the thread represented by that object (if any).

A `std::thread` object might not represent a thread because...

- ...the thread has terminated.
- ...the thread has been explicitly *detached* using the `detach` method.

Thread IDs

Each thread has a unique ID.

- Call `get_id` on the `std::thread` object.
- Call `std::this_thread::get_id` to get the ID of the current thread.

Can be used as keys in an associative container.

```
std::map<std::thread::id, string> thread_name;

thread_name[std::this_thread::get_id( )] = "Me";
thread_name[some_thread.get_id( )] = "You";
```

Sleeping

The static method `std::thread::sleep` allows a thread to block until a specified time occurs.

```
void f( )
{
  // Do stuff.

  // Wait until the time specified arrives.
  std::thread::sleep(
    std::get_system_time( ) +
    std::posix_time::milliseconds( 1000 ) );
}
```

There is also `std::this_thread_sleep`. It waits until a specified interval has elapsed.

```
std::this_thread::sleep(
  std::posix_time::milliseconds( 1000 ) );
```

Mutual Exclusion

C++ provides several “mutex” classes for locks.

- **`class`** `mutex`
  Provides basic `lock`, `try_lock`, `unlock` methods.
- **`class`** `timed_mutex`
  Allows for specifying a time out with `timed_lock`.
- **`class`** `recursive_mutex`
  Will not deadlock if the same thread locks twice.
- **`class`** `recursive_timed_mutex`
  A recursive mutex with a `timed_lock` method.

Using Mutex Objects

Mutex objects can be used directly. . .

```
#include <mutex>

// Must be shared between threads (so global).
std::mutex mutual_exclusion;

void f( )
{
  mutual_exclusion.lock( );

  // This thread now has exclusive access.

  mutual_exclusion.unlock( );
}
```

Not recommended. This isn't exception safe. The `unlock` method won't be called if an exception is thrown.

Use Template Wrappers

Instead use RAI enabled wrapper templates.

```
std::mutex mutual_ex;

void f( )
{
  // Constructor of 'guard' locks.
  std::lock_guard<std::mutex> guard( mutual_ex );

  // This thread now has exclusive access.

  // Destructor of 'guard' unlocks.
}
```

Lock is released even if an exception is thrown.

There is also a `unique_lock` template that is similar except it can be moved (for example into containers) and returned from functions. It has other features as well.

Condition Variables

Modeled after POSIX condition variables, but with a C++ interface.

- Allows a thread to *wait* until an arbitrary condition is true.
  - Another thread *notifies* the waiting thread (or all waiting threads at once).
- Proper use of condition variables is tricky.
  - Refer to a description of POSIX condition variables to understand the issues.

Example

```
std::condition_variable cond;
std::mutex mut;
bool data_ready;

void wait_for_data( )
{
  std::unique_lock<std::mutex> lock( mut );
  while( !data_ready ) cond.wait( lock );
  process_data( );
}

void prepare_data( )
{
  do_the_work( );
  {
    std::lock_guard<std::mutex> lock( mut );
    data_ready = true;
  }
  cond.notify_one( );
}
```

Conclusion

- C++ threads has more facilities than described here.
- POSIX-like functionality with convenient C++ interface.
- Boost.Thread inspired new thread features of the C++ 2011 library.

Questions?