MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

C++ std::threads

Daniel Playne
d.p.playne@massey.ac.nz

Reminder

In the last lesson we looked at some of the basics of C++ threading.

- `thread`
- `mutex`
- `timed_mutex`
- `recursive_mutex`
- `recursive_timed_mutex`
- `lock`

C++ `std::threads` - Mutexes

Mutexes are generally not intended to be used directly and more commonly utlised through `lock_guard` or `std::unique_lock`.

A `lock_guard` is an RAII (Resource Allocation Is Initialisation) mechanism for taking ownership of a mutex for the duration of a scoped block.

A `unique_lock` is a general-purpose mutex ownership wrapper that supports deferred locking, time-constrained attempts at locking, recursive locking, transfer of ownership and use with condition variables.

C++ std::threads - lock_guard

When a `lock_guard` object is created, it will attempt to take ownership of the given mutex. When the `lock_guard` leaves scope and is destructed, it will release the mutex.

```
std::mutex mtx;

void thread_function(int id) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::lock_guard<std::mutex> lck(mtx);
    std::cout << "Thread: " << id << std::endl;
}

int main() {
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; i++) {
        threads.push_back(std::thread(thread_function, i));
    }
    for(std::thread &t : threads) {
        t.join();
    }
}
```

C++ std::threads - lock_guard

One of the main advantages of using a `lock_guard` is that they ensure mutexes are properly unlocked when an exception occurs.

If a `lock_guard` is created inside a try block and an exception occurs. The mutex will be properly released when the exception-handling piece of code is executed and the `lock_guard` leaves scope.

C++ std::threads - lock_guard

```
std::mutex mtx;

void print_maybe(int i) {
    int r = rand();
    if(r % 2 == 0) {
        std::cout << i << std::endl;
    } else {
        throw "Unlucky";
    }
}

void thread_function(int id) {
    try {
        std::lock_guard<std::mutex> lock(mtx);
        print_maybe(id);
    } catch(const char *s) {
        // Handle exception
    }
}

int main() {
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; i++) {
        threads.push_back(std::thread(thread_function, i));
    }
    for(std::thread &t : threads) {
        t.join();
    }
}
```

C++ std::lock_guard

There is another constructor for lock_guard

```
lock_guard(mutex_type &m, std::adopt_lock_t t)
```

This constructor will create a lock_guard and adopt the ownership of the mutex without trying to lock it.

Adopting the lock of a mutex should only ever be used if the thread already owns the mutex (behaviour is undefined otherwise).

C++ - std::lock_guard

The adopt_lock constructor allows lock_guards to be used in conjunction with std::lock. First the necessary mutexes are locked, then lock_guards are constructed that adopt those locks.

```
std::mutex mtx1, mtx2, mtx3;

void thread_function_a() {
   std::lock(mtx1, mtx2, mtx3);
   std::lock_guard<std::mutex> lck1(mtx1, std::adopt_lock);
   std::lock_guard<std::mutex> lck2(mtx2, std::adopt_lock);
   std::lock_guard<std::mutex> lck3(mtx3, std::adopt_lock);

   std::cout << "Thread a" << std::endl;
}

void thread_function_b() {
   std::lock(mtx3, mtx2, mtx1);
   std::lock_guard<std::mutex> lck1(mtx1, std::adopt_lock);
   std::lock_guard<std::mutex> lck2(mtx2, std::adopt_lock);
   std::lock_guard<std::mutex> lck3(mtx3, std::adopt_lock);

   std::cout << "Thread b" << std::endl;
}
```

C++ `unique_lock`

C++ also has `unique_lock` which supports *move semantics* to allow a `unique_lock` to be *moved*.

This feature can be useful when a thread that currently holds a mutex wishes to pass ownership of that mutex to another thread.

The original thread can *move* the `unique_lock` to the other thread and ensure that no other thread obtains ownership of it in between.

C++ unique_lock

Moving a unique_lock to another thread:

```
std::mutex mtx1;

void thread_function(int id, std::unique_lock<std::mutex> &&lck) {
   std::this_thread::sleep_for(std::chrono::seconds(1));
   std::cout << "Thread " << id << " running." << std::endl;
   std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main() {
   // Create Threads
   std::vector<std::thread> threads;
   for(int i = 0; i < 5; i++) {
      // Get unique lock
      std::unique_lock<std::mutex> lck(mtx1);

      // Create Thread (move unique lock to thread)
      std::cout << "Lanching thread " << i << "." << std::endl;
      threads.push_back(std::thread(thread_function, i, std::move(lck)));
   }

   // Join Threads
   for(std::thread &t : threads) {
      t.join();
   }
}
```

C++ `condition_variable`

The C++ threading API (before C++20) does not support *semaphores*.

Instead a `condition_variable` can be used to block a thread until *notified* by another thread.

While a thread is waiting on a `condition_variable` it will be blocked and not scheduled on the CPU.

C++ condition_variable

condition_variable provides the following functions:

- wait - Wait until notified
- wait_for - Wait until notified or timeout
- wait_until - Wait until notified or time point is reached
- notify_one - Notify one thread waiting on condition variable.
- notify_all - Notify all threads waiting on condition variable.

C++ condition_variable

Example:

```
std::mutex mtx;
std::condition_variable cond;
bool ready = false;

void thread_function(int id) {
   std::unique_lock<std::mutex> lock(mtx);
   std::cout << "Thread " << id << " waiting." << std::endl;
   while(!ready) cond.wait(lock);
   std::cout << "Thread " << id << " completed." << std::endl;
}

int main() {
   std::vector<std::thread> threads;
   for(int i = 0; i < 5; ++i) {
      threads.push_back(std::thread(thread_function, i));
   }
   std::this_thread::sleep_for (std::chrono::seconds(3));
   ready = true;
   cond.notify_all();
   for(std::thread &t : threads) {
      t.join();
   }
}
```

C++ `condition_variable`

In the following thread function, the thread obtains a lock on the mutex `mtx` which will be held until the call to `cond.wait(lock)`.

When the thread calls `wait` it will release the lock. When the thread is *notified* it will wake up and obtain a lock on `mtx` before continuing execution.

```
std::mutex mtx;
std::condition_variable cond;
bool ready = false;

void thread_function(int id) {
   std::unique_lock<std::mutex> lock(mtx);
   std::cout << "Thread " << id << " waiting." << std::endl;
   while(!ready) cond.wait(lock);
   ...
}
```

C++ `condition_variable`

Note the following line is used when *waiting* on a condition variable.

```
while(!ready) cond.wait(lock);
```

In general, threads are only woken up when another thread calls either `notify_one` or `notify_all`. However, some implementations may produce *spurious* calls to wake up threads.

It is the programmer's responsibility to ensure the correct conditions to resume execution have been met.

Example - Game of Life

To continue our Game of Life example we should look at how to implement our *phase-parallel* simulator.

We will need to explicitly create threads that will each update one section of the grid.

After each new generation is computed, the threads must synchronise with each other before continuing onto the next generation.

Example - Game of Life

First a sequential implementation

```
unsigned char *buffer[2];
buffer[0] = new unsigned char[N*N];
buffer[1] = new unsigned char[N*N];

int r = 0; // Read/Write Index
int w = 1;

for(int k = 0; k < N*N; ++k) {   // Initialise
   buffer[r][k] = rand() % 2;
}

for(int ig = 0; ig < G; ++ig) {          // For each Generation
   for(int iy = 0; iy < N; ++iy) {       // For each Row
      for(int ix = 0; ix < N; ++ix) {    // For each Column
         // Count neighbours
         int count = count_neighbours(buffer[r], ix, iy, N);

         // Update
         buffer[w][iy*N + ix] = gol_update(buffer[r][iy*N+ix], count);
      }
   }

   // Swap buffers
   r = !r;
   w = !w;
}
```

Game of Life Functions:

```
// Count neighbours
int count_neighbours(unsigned char *buffer, int ix, int iy, int N) {
   // Neighbouring Indexes
   int ym1 = (iy == 0) ? N-1 : iy-1;
   int yp1 = (iy == N-1) ? 0 : iy+1;
   int xm1 = (ix == 0) ? N-1 : ix-1;
   int xp1 = (ix == N-1) ? 0 : ix+1;

   // Count neighbours
   return buffer[ym1 * N + xm1] + buffer[ym1 * N + ix] + buffer[ym1 * N + xp1] +
          buffer[ iy * N + xm1] +                       buffer[ iy * N + xp1] +
          buffer[yp1 * N + xm1] + buffer[yp1 * N + ix] + buffer[yp1 * N + xp1];
}

// Game of Life Rules
unsigned char gol_update(unsigned char status, int count) {
   if((status == 0) && (count == 3)) {
      return 1;
   } else if((status == 1) && (count == 2 || count == 3)) {
      return 1;
   } else {
      return 0;
   }
}
```

Example - Game of Life

Creating threads to update different sections of the grid is relatively straightforward.

There is still the question of how many threads to use, which will we avoid by making the user decide.

```
std::thread *threads = new std::thread[T];

for(int t = 0; t < T; ++t) {
   threads[t] = std::thread(gol, ...);
}
```

Example - Game of Life

```
// Thread Function
void gol(unsigned char *buffer[2], int N, int G, int id, int num_threads) {
  int r = 0;  // Read/Write index
  int w = 1;

  int start = ( id    * N) / num_threads;     // Start Row
  int end   = ((id+1) * N) / num_threads;     // End Row

   for(int ig = 0; ig < G; ++ig) {            // For each Generation
      for(int iy = start; iy < end; ++iy) {   // For each Row (for this thread)
         for(int ix = 0; ix < N; ++ix) {      // For each Columm
            // Count neighbours
            int count = count_neighbours(buffer[r], ix, iy, N);

            // Update
            buffer[w][iy*N + ix] = gol_update(buffer[r][iy*N+ix], count);
         }
      }

      // Barrier
      barrier(ig+1, num_threads);

      // Swap buffers
      r = !r;
      w = !w;
   }
}
```

C++ `condition_variable`

To ensure that all the threads have updated their sections of the grid before continuing onto the next generation, we have used the following function:

```
barrier(ig+1, num_threads)
```

Barriers are only defined in the C++20 standard and may not be available in many compilers.

We will need to implement our own barrier function.

Example - Game of Life

```
std::condition_variable barrier_condition;
std::mutex barrier_mutex;

int barrier_count = 0;
int barrier_generation = 0;

void barrier(int g, int num_threads) {
   // Lock Mutex
   std::unique_lock<std::mutex> lock(barrier_mutex);

   // Increment barrier counter
   ++barrier_count;

   if(barrier_count == num_threads) {
      // Update barrier generation
      barrier_generation = g;

      // Reset barrier count
      barrier_count = 0;

      // Notify all waiting threads
      barrier_condition.notify_all();
   } else {
      // Wait
      barrier_condition.wait(lock, [=]() {
         return barrier_generation == g;
      });
   }
}
```

Summary

- `lock_guard`
- `unique_lock`
- `condition_variable`
- Game of Life implementation