Multithreading

C++ Concurrency Support

Below are some of the useful methods and objects I found in documentation for the C++ Concurrency Support Library. This is of course not an exhaustive list, and only covers what I've used in the examples later on this page.

std::mutex is a C++11 object which offers mutually exclusive ownership over shared resources within a C++ program. The object is not directly associated with the shared resources via construction or any other initialization, but instead the object is used to block program execution should another thread want to step into a block of code that would attempt to modify a resource that is currently in use elsewhere.

Mutex locks provide many functions, and after instatiation we can use these functions to enforce ownership of shared resources.

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>

int main(const int argc, const char * argv[]) {
  std::cout << "main() thread id: " << std::this_thread::get_id() << std::endl;

  static std::mutex mtx_A, mtx_B;

  std::thread thread_A([]()->void {
    mtx_A.lock(); // Call mutex member function std::mutex::lock()
    std::cout << std::this_thread::get_id() << " thread_A: Lock A\n";
    std::this_thread::sleep_for(std::chrono::seconds(3));
    mtx_A.unlock(); // Call mutex member function std::mutex::unlock()
    std::cout << std::this_thread::get_id() << " thread_A: Unlock A\n";
  });

  std::thread thread_B([]()->void {
```

```
    std::this_thread::sleep_for(std::chrono::seconds(1));
    mtx_A.lock();
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    std::cout << std::this_thread::get_id() << " thread_B: Lock A\n";
    std::this_thread::sleep_for(std::chrono::seconds(3));
    mtx_A.unlock();
    std::cout << std::this_thread::get_id() << " thread_B: Unlock A\n";
  });

  thread_A.join();
  thread_B.join();
  return 0;
}
```

The output of this example should show the general idea. If you run this code, you will notice one thread takes ownership, waits a few seconds, releases the object, and then the same is repeated for the next thread.

```
main() thread id: 140533678282560
140533678278400 thread_A: Lock A
140533678278400 thread_A: Unlock A
140533669885696 thread_B: Lock A
140533669885696 thread_B: Unlock A
```

You might notice my use of the `std::this_thread` namespace, specifically std::this_thread::get_id() and std::this_thread::sleep_for().I also use the std::chrono library often to define time periods for the program to wait. For writing examples and testing multithreaded functionality of C++, these methods will prove very useful, but keep in mind they are just to support *examples* - you typically wouldn't want your program to wait for several seconds, and thus many of these examples can be 'fixed' by removing these methods. For instance, the livelock example on this page often results in no livelock at all if I hadn't intentionally synchronized the iteration of loops within the opposing threads. Sometimes you may see a livelock occur in that example naturally, but more often one thread executes at a *slightly* different time, which breaks from the livelock and the program exits normally. The examples serve as proof of concept, and not real world problems or scenarios.

std::lock is a function available in C++11 which handles the locking of N mutex objects, avoiding the case of deadlock when any one of them is unavailable. This means that if any one of the mutex objects passed to the call to `std::lock`, the program execution will be blocked until the resource is available. When it becomes available, the resources are locked and execution continues.

std::try_lock is a function available in C++11 which is similar to `std::lock` in that is handles locking N objects for us and avoiding deadlock. The major difference here is what you would expect, the

return value. This function returns `-1` if the resources have been successfully locked, and `0` if not. This means it can be used to programmatically react to resources that are currently unavailable.

RAII: Resource Acquisition Is Initialization descirbes the implementation of certain objects in C++ which encapsulate the initialization and deconstruction of required resources which may be shared or unavailable at any point in time. There are many examples of objects that follow the RAII concept in C++ concurrency libraries, listed below are only a few of these. RAII applies to heap memory management, which is a much more common practice which programmers of all skill levels would already be familiar with.

std::lock_guard is a C++11 object useful for managing a *single* mutex lock. The `lock_guard` object has a constructor which supports a single mutex as an argument, and on construction this object will attempt to lock the mutex, obtaining ownership of it. If the mutex is already owned by another object, the thread or program execution will be blocked at this call and will not proceed until the resource has been released by the current owning object. When the program leaves the scope of an owning `lock_guard` object, the mutex is automatically released - we do not need to remember to call `unlock` or any similar function.

std::scoped_lock is a C++17 object useful for managing N mutex locks, which means `scoped_lock` has a constructor that supports N arguments where each is a mutex we want to obtain ownership of. A caveat to this object is that the constructor also supports 0 arguments, which means the programmer can construct a `scoped_lock` that doesn't ever lock a single mutex. The compiler will not complain, whereas with `lock_guard` the compiler will reject any attempt to construct the object without providing a mutex to lock. Similar to `lock_guard`, when the program leaves the scope of a `scoped_lock` object, it will automatically call `unlock` on all mutex locks it has ownership of.

std::unique_lock provides a C++11 method of deferred mutex locking, which supports constructing locks without requiring the resource to be immediately available, using lock tags such as std::defer_lock. Notably, `unique_lock` is used in conjuction with `std::condition_variable`, which provides sychronization support via notifications for blocked threads.

std::condition_variable is used alongside of `unique_lock`, where there is a shared value that is modified to to signal the unblocking of a previously blocked thread. This is useful for synchronizing events or jobs to continue processing only once the program has reached a valid state. An example of using this method of synchronization can be seen in the code below.

```
/*##############################################################################
############
## Author: Shaun Reed                                                        ##
## Legal: All Content (c) 2022 Shaun Reed, all rights reserved             ##
## About: An example of condition_variables in multithreaded C++           ##
##                                                                         ##
## Contact: shaunrd0@gmail.com  | URL: www.shaunreed.com | GitHub: shaunrd0   ##
```

```
##########################################################################
#############
*/

#include <chrono>
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

static std::mutex mtx;
std::condition_variable cv;
bool processing = false;

// Starts a job that waits for kick-off from main
// + When job finishes, handoff result back to main via processing bool
void job(int32_t & shared) {
  std::unique_lock uniqueLock(mtx);
  cv.wait(uniqueLock, []()->bool {return processing;});
  std::cout << std::this_thread::get_id()
          << " thread_A: Initial value of shared = " << shared << std::endl;
  while (shared < INT32_MAX) {
    shared++;
  }
  // We're no longer processing data
  processing = false;
  std::cout << std::this_thread::get_id()
          << " thread_A: Done working." << std::endl;
  uniqueLock.unlock(); // Important! Unlock uniqueLock before we notify
  // Notify main that we've finished, so it can proceed
  cv.notify_one();
}

int main(const int argc, const char * argv[]) {
  std::cout << "main() thread id: " << std::this_thread::get_id() << std::endl;

  int32_t share = 0;
  std::thread thread_A(job, std::ref(share));

  mtx.lock();
```

```
    std::this_thread::sleep_for(std::chrono::seconds(3));
    share = INT32_MAX / 2;
    processing = true;
    mtx.unlock();
    // Notify thread_A that its work can begin
    cv.notify_one();

    // Wait until thread_A finishes its work
    std::unique_lock uniqueLock(mtx);
    // Block execution until we are not processing
    cv.wait(uniqueLock, []()->bool { return !processing;});
    std::cout << std::this_thread::get_id() << " main(): final value of shared = "
              << share << std::endl;
    thread_A.join();

    return 0;
}
```

The output of this program shows that the value of `share` in the main function is what we would expect upon the unblocking of `thread_A`. Since this value was passed to the threads job using `std::ref`, it is a reference to the same value and the modifications made by `thread_A` are seen in main as well.

```
main() thread id: 139826095839040
139826095834880 thread_A: Initial value of shared = 1073741823
139826095834880 thread_A: Done working.
139826095839040 main(): final value of shared = 2147483647
```

std::basic_osyncstream provides a C++20 way to synchronize outputs to a shared stream, which could be constructed with `std::basic_osyncstream out(std::cout);` and use directly as if it were `std::cout`. Unfortunately many compilers don't support this yet, but worth mentioning. See C++ compiler support for more up to date information on compiler support in regards to more modern C++ versions.

For now, these are all the methods and objects I'll use in the examples below, but there is surely always more to learn and I will return to update this page in the future as I improve upon or learn new multithreading techniques and practices in C++.

Process Lifetime

Threads should not be confused with processes. A process can own several threads, while any given thread can only be owned by a single process. In general, the lifetime of a process is outlined below, where `New-(Admitted)->Ready` describes the transisition between the `New` and `Ready` process states following the `Admitted` signal from the OS that created the thread.

- New
  - New-(Admitted)->Ready
- Ready
  - Ready-(Dispatch)->Running
- Running
  - Running-(Interrupt)->Ready
  - Running-(Wait for I/O or Event)->Watiting
  - Running-(Exit)->Terminated
- Waiting
  - Waiting-(I/O or Event Completed)->Ready
- Terminated

Thread Lifetime

When we call `std::thread` and construct a new thread, we request a new thread from the OS, which is supplied threads by the kernel, which manages the resources available to us by our hardware.

In general, operating systems have the following states for threads that they're managing.

- Creation - Thread has been created but not allocated any resources yet
- Ready - The thread has resources available to it, and is ready to be assigned some work
- Running - The thread is running a job, which may result in any of the below states, which will subsequently return to the Ready state, which transitions to a Running thread.
  - Waiting - The thread is waiting for some event or signal to return to the Ready state
  - Delayed - The thread has been delayed, usually to support more important processing happening on other threads
  - Blocked - The thread requires a resource which is currently in use by another thread, and is blocked until this resource is made available
- Finished - The thread is finished, and is joined back into the owning process

These are the general steps taken by processses, including the operating system, when threads are created, used, and destroyed. To destroy a thread we must first join it back into the owning process which releases the resources back to the kernel for allocating to other threads.

Examples

The sections below contain programs that show examples for various problems and solutions in multithreaded applications. These are just general examples and not real world problems, so many of the problems were intentionally created to showcase a situation that can occur. If only interested

in the source code, check out my Git repository shaunrd0/klips where I practice general programming.

Race Conditions

The following program is an example of the `problem()` and `solution()` to race conditions in C++. This is a basic example, and just shows what *can* happen if two or more threads happen to access the same variable simultaneously. Since the value of `x` is shared, and this value is incremented within a `for` loop by all 5 threads, the value of `x` can be the same when the increment is applied in some undefined number of threads.

This means that each time we run the program, the output of `problem()` will vary from 1000000-5000000. This variation in final output is due to the undefined number of threads that simultaneously access `x` which incrementing it within the loop. In contrast, each time we run the `solution()`, the value will always be the expected output of 5000000, because we have utilized the concurrency features of C++ appropriately.

```
/*##################################################################
############
## Author: Shaun Reed                                            ##
## Legal: All Content (c) 2022 Shaun Reed, all rights reserved    ##
## About: An example of a race condition problem and solution     ##
##                                                               ##
## Contact: shaunrd0@gmail.com  | URL: www.shaunreed.com | GitHub: shaunrd0   ##
##################################################################
#############
*/

#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

void problem() {
  std::vector<std::thread> threads;
  const uint8_t thread_count = 5;
  // With no mutex lock, the final value will vary in the range 1000000-5000000
  // + Threads will modify x simultaneously, so some iterations will be lost
  // + x will have same initial value entering this loop on different threads
  uint32_t x = 0;
  for (uint8_t i = 0; i < thread_count; i++) {
    threads.emplace_back([&x](){
```

```
      for (uint32_t i = 0; i < 1000000; i++) {
        x = x + 1;
      };
    });
  }
  // Ensure the function doesn't continue until all threads are finished
  // + There's no issue here, the issue is in how `x` is accessed above
  for (auto &thread : threads) thread.join();
  std::cout << x << std::endl;
}

// Create mutex lock to prevent threads from modifying same value simultaneously
static std::mutex mtx;
void solution() {
  std::vector<std::thread> threads;
  const uint8_t thread_count = 5;
  uint32_t x = 0;
  for (uint8_t i = 0; i < thread_count; i++) {
    threads.emplace_back([&x](){
      // The first thread that arrives here will 'lock' other threads from passing
      // + Once first thread finishes, the next thread will resume
      // + This process repeats until all threads finish
      std::lock_guard<std::mutex> lock(mtx);
      for (uint32_t i = 0; i < 1000000; i++) {
        x = x + 1;
      };
    });
  }
  // Ensure the function doesn't continue until all threads are finished
  for (auto &thread : threads) thread.join();
  std::cout << x << std::endl;
}

int main(const int argc, const char * argv[]) {
  // Result will vary from 1000000-5000000
  problem();

  // Result will always be 5000000
  solution();
  return 0;
```

```
}
```

The output of this program is

```
1374956
5000000
```

Deadlocks

Deadlocks occur when two threads lock a shared resource from each other. In the example below, `thread_A` has locked `mtx_A`, and it *wants to* lock `mtx_B`. This is not possible, because `thread_B` has already locked `mtx_B`, and is now waiting to lock `mtx_A`. Neither of these locks will ever pass, because the threads both require a common resource to continue.

For the sake of the example, I provide a way out of the deadlock situation within the `problem()` function. Note that if it were not for this, the program would simply never finish execution. It would wait forever, as neither resource would ever become available to the opposing thread. I have split this code between several blocks with short explanations of each problem / solution, but all code within this section is within the same program. The full example can be found in my general programming practice repository, klips.

```
/*##################################################################
############
## Author: Shaun Reed                                            ##
## Legal: All Content (c) 2022 Shaun Reed, all rights reserved    ##
## About: An example and solution for deadlocks in C++            ##
##                                                               ##
## Contact: shaunrd0@gmail.com  | URL: www.shaunreed.com | GitHub: shaunrd0   ##
##################################################################
#############
*/

#include <chrono>
#include <iostream>
#include <mutex>
#include <sstream>
#include <thread>

static std::mutex mtx_A, mtx_B, output;

// Helper function to output thread ID and string associated with mutex name
```

```
// + This must also be thread-safe, since we want threads to produce output
// + There is no bug or issue here; This is just in support of example output
void print_safe(const std::string & s) {
  std::scoped_lock<std::mutex> scopedLock(output);
  std::cout << s << std::endl;
}

// Helper function to convert std::thread::id to string
std::string id_string(const std::thread::id & id) {
  std::stringstream stream;
  stream << id;
  return stream.str();
}

// In the two threads within this function, we have a problem
// + The mutex locks are acquired in reverse order, so they collide
// + This is called a deadlock; The program will *never* finish
void problem() {
  std::thread thread_A([]()->void {
    mtx_A.lock();
    print_safe(id_string(std::this_thread::get_id()) + " thread_A: Locked A");
    std::this_thread::sleep_for(std::chrono::seconds(1));
    mtx_B.lock(); // We can't lock B! thread_B is using it
    // The program will never reach this point in execution; We are in deadlock
    print_safe(id_string(std::this_thread::get_id())
               + " thread_A: B has been unlocked, we can proceed!\n  Locked B"
    );
    std::this_thread::sleep_for(std::chrono::seconds(1));

    print_safe(id_string(std::this_thread::get_id())
               + " thread_A: Unlocking A, B..."
    );
    mtx_A.unlock();
    mtx_B.unlock();
  });

  std::thread thread_B([]()->void {
    mtx_B.lock();
    print_safe(id_string(std::this_thread::get_id()) + " thread_B: Locked B");
    std::this_thread::sleep_for(std::chrono::seconds(1));
```

```
    mtx_A.lock(); // We can't lock A! thread_A is using it
    // The program will never reach this point in execution; We are in deadlock
    print_safe(id_string(std::this_thread::get_id())
               + " thread_B: A has been unlocked, we can proceed!\n  Locked A"
    );
    std::this_thread::sleep_for(std::chrono::seconds(1));

    print_safe(id_string(std::this_thread::get_id())
               + " thread_B: Unlocking B, A..."
    );
    mtx_B.unlock();
    mtx_A.unlock();
  });

  // This offers a way out of the deadlock, so we can proceed to the solution
  std::this_thread::sleep_for(std::chrono::seconds(2));
  char input;
  print_safe("\n"
             + id_string(std::this_thread::get_id())
             + " problem(): We are in a deadlock. \n"
             + "    Enter y/Y to continue to the solution...\n"
  );
  while (std::cin >> input) {
    if (input != 'Y' && input != 'y') continue;
    else break;
  }
  print_safe(id_string(std::this_thread::get_id())
             + " problem(): Unlocking A, B..."
  );
  mtx_A.unlock();
  mtx_B.unlock();

  thread_A.join();
  thread_B.join();
}
```

There are a few solutions we could use to work around this problem. We could use std::lock, which is available in C++11. With this approach, we still need to remember to unlock each mutex, or else we end up in a deadlock situation again. The only difference is that this time we caused it ourselves by forgetting to unlock the resource instead of two threads with a conflict of interest.

```
// std::lock will lock N mutex locks
// + If either is in use, execution will block until both are available to lock
void solution_A() {
  std::thread thread_A([]()->void {
    std::lock(mtx_A, mtx_B);
    print_safe(id_string(std::this_thread::get_id()) + ": Locked A, B");
    std::this_thread::sleep_for(std::chrono::seconds(1));

    print_safe(id_string(std::this_thread::get_id()) + ": Unlocking A, B...");
    mtx_A.unlock();
    mtx_B.unlock();
  });

  std::thread thread_B([]()->void {
    std::lock(mtx_B, mtx_A);
    print_safe(id_string(std::this_thread::get_id()) + ": Locked B, A");
    std::this_thread::sleep_for(std::chrono::seconds(1));

    print_safe(id_string(std::this_thread::get_id()) + ": Unlocking B, A...");
    mtx_B.unlock();
    mtx_A.unlock();
  });

  thread_A.join();
  thread_B.join();
}
```

We could also use std::lock_guard, which is also available in C++11. The benefit to constructing this object is that we are not required to remember to unlock either mutex. `lock_guard` has a constructor which takes a *single* argument for *one* mutex to manage for us. This means we need two `lock_guard` objects to work around this deadlock situation.

```
// std::lock_guard is a C++11 object which can be constructed with 1 mutex
// + When the program leaves the scope of the guard, the mutex is unlocked
void solution_B() {
  std::thread thread_A([]()->void {
    // lock_guard will handle unlocking when program leaves this scope
    std::lock_guard<std::mutex> guard_A(mtx_A), guard_B(mtx_B);
    print_safe(id_string(std::this_thread::get_id()) + ": Locked A, B");
    std::this_thread::sleep_for(std::chrono::seconds(1));
```

```
    print_safe(id_string(std::this_thread::get_id()) + ": Unlocking A, B...");
    // We don't need to explicitly unlock either mutex
  });

  std::thread thread_B([]()->void {
    std::lock_guard<std::mutex> guard_B(mtx_B), guard_A(mtx_A);
    print_safe(id_string(std::this_thread::get_id()) + ": Locked B, A");
    std::this_thread::sleep_for(std::chrono::seconds(1));

    print_safe(id_string(std::this_thread::get_id()) + ": Unlocking B, A...");
    // We don't need to explicitly unlock either mutex
  });

  thread_A.join();
  thread_B.join();
}
```

A third and final example of a solution to deadlocks is using the std::scoped_lock object available in C++17. This object has a constructor that takes N arguments, each one being a mutex we want the object to manage for us. This means for N mutex locks, we only need *one* `scoped_lock` object. When the program leaves the scope of this object, all mutex locks will be unlocked for us, so we don't need to remember to do this ourselves.

```
// std::scoped_lock is a C++17 object that can be constructed with N mutex
// + When the program leaves this scope, all N mutex will be unlocked
void solution_C() {
  std::thread thread_A([]()->void {
    // scoped_lock will handle unlocking when program leaves this scope
    std::scoped_lock scopedLock(mtx_A, mtx_B);
    print_safe(id_string(std::this_thread::get_id()) + ": Locked A, B");
    std::this_thread::sleep_for(std::chrono::seconds(1));

    print_safe(id_string(std::this_thread::get_id()) + ": Unlocking A, B...");
    // We don't need to explicitly unlock either mutex
  });

  std::thread thread_B([]()->void {
    std::scoped_lock scopedLock(mtx_B, mtx_A);
    print_safe(id_string(std::this_thread::get_id()) + ": Locked B, A");
```

```
    std::this_thread::sleep_for(std::chrono::seconds(1));

    print_safe(id_string(std::this_thread::get_id()) + ": Unlocking B, A...");
    // We don't need to explicitly unlock either mutex
  });

  thread_A.join();
  thread_B.join();
}

int main(const int argc, const char * argv[]) {
  std::cout << "main() thread id: " << std::this_thread::get_id() << std::endl;

  problem();

  print_safe("\nsolution_A, using std::lock\n");
  solution_A();

  print_safe("\nsolution_B, using std::lock_guard\n");
  solution_B();

  print_safe("\nsolution_C, using std::scoped_lock\n");
  solution_C();

  return 0;
}
```

The output of this program is

```
main() thread id: 140625995487040
140625995482880 thread_A: Locked A
140625987090176 thread_B: Locked B

140625995487040 problem(): We are in a deadlock.
    Enter y/Y to continue to the solution...

y
140625995487040 problem(): Unlocking A, B...
140625995482880 thread_A: B has been unlocked, we can proceed!
  Locked B
```

```
140625987090176 thread_B: A has been unlocked, we can proceed!
  Locked A
140625995482880 thread_A: Unlocking A, B...
140625987090176 thread_B: Unlocking B, A...

solution_A, using std::lock

140625987090176: Locked A, B
140625987090176: Unlocking A, B...
140625995482880: Locked B, A
140625995482880: Unlocking B, A...

solution_B, using std::lock_guard

140625995482880: Locked A, B
140625995482880: Unlocking A, B...
140625987090176: Locked B, A
140625987090176: Unlocking B, A...

solution_C, using std::scoped_lock

140625987090176: Locked A, B
140625987090176: Unlocking A, B...
140625995482880: Locked B, A
140625995482880: Unlocking B, A...

Process finished with exit code 0
```

All that said, why would one choose to use std::lock_guard over std::scoped_lock? If there is only a need to lock a single resource, the use of `scoped_lock` just isn't necessary. An added benefit to using `lock_guard` is it's availability in C++11, whereas `scoped_lock` isn't available until C++17. Also, the constructor for `scoped_lock` *can* technically accept 0 arguments. This means that one can forget to pass the mutex locks required for thread-safe code, and the compiler will happily accept their code. By using `lock_guard` when possible, we enlist the compiler's help in avoiding a small oversight in the use of `scoped_lock`.

Livelocks

A livelock occurs when `thread_A` has ownership of `mtx_A`, and `thread_B` has ownership of `mtx_B`. Each thread attempts to lock the opposing mutex, and when the thread realizes the mutex is already locked, they attempt to take corrective action and unlock their resource. The intention is to

free up the resource for the other thread so it can complete it's work, but in some cases the threads continually lock and unlock their resources, running into the same problem each time.

This is a weird one to provide an example for, and I had to do some synchonization between the loops in `thread_A` and `thread_B` to make the results consistent. The threads will only enter livelock for 5 iterations, and then `thread_B` will give up, so the example can continue to show the solution. This is kindof an odd case, since we have intentionally synchronized the loops to produce a livelock situation, but I think the example shows the general idea.

```
/*##################################################################
############
## Author: Shaun Reed                                            ##
## Legal: All Content (c) 2022 Shaun Reed, all rights reserved     ##
## About: An example and solution for livelocks in C++            ##
##                                                               ##
## Contact: shaunrd0@gmail.com  | URL: www.shaunreed.com | GitHub: shaunrd0   ##
##################################################################
#############
*/

#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

static std::mutex mtx_A, mtx_B, output;

// Helper function to output thread ID and string associated with mutex name
// + This must also be thread-safe, since we want threads to produce output
// + There is no bug or issue here; This is just in support of example output
void print_safe(const std::string & s) {
  std::scoped_lock<std::mutex> scopedLock(output);
  std::cout << s << std::endl;
}

void problem() {
  // Construct a vector with 5 agreed-upon times to synchronize loops in threads
  typedef std::chrono::time_point<std::chrono::steady_clock,
      std::chrono::steady_clock::duration> time_point;
  std::vector<time_point> waitTime(6);
```

```
for (uint8_t i = 0; i < 6; i++) {
  waitTime[i] = std::chrono::steady_clock::now()+std::chrono::seconds(1+i);
}

std::thread thread_A([waitTime]()->void {
  uint8_t count = 0; // Used to select time slot from waitTime vector
  bool done = false;
  while (!done) {
    count++;
    std::lock_guard l(mtx_A);
    std::cout << std::this_thread::get_id() << " thread_A: Lock A\n";
    // Wait until the next time slot to continue
    // + Helps to show example of livelock by ensuring B is not available
    std::this_thread::sleep_until(waitTime[count]);
    std::cout << std::this_thread::get_id() << " thread_A: Requesting B\n";
    if (mtx_B.try_lock()) {
      done = true;
      std::cout << std::this_thread::get_id()
                << " thread_A: Acquired locks for A, B! Done.\n";
    }
    else {
      std::cout << std::this_thread::get_id()
                << " thread_A: Can't lock B, unlocking A\n";
    }
  }
  mtx_B.unlock();
});

std::thread thread_B([waitTime]()->void {
  // As an example, enter livelock for only 5 iterations
  // + Also used to select time slot from waitTime vector
  uint8_t count = 0;
  bool done = false;
  while (!done && count < 5) {
    count++;
    std::lock_guard l(mtx_B);
    // Wait until the next time slot to continue
    // + Helps to show example of livelock by ensuring A is not available
    std::this_thread::sleep_until(waitTime[count]);
    if (mtx_A.try_lock()) {
```

```
        // The program will never reach this point in the code
        // + The only reason livelock ends is because count > 5
        done = true;
      }
    }
  });

  thread_A.join();
  thread_B.join();
}

// The solution below uses std::scoped_lock to avoid the livelock problem
void solution() {
  std::thread thread_A([]()->void {
    for (int i = 0; i < 5; i++) {
      // Increase wait time with i
      // + To encourage alternating lock ownership between threads
      std::this_thread::sleep_for(std::chrono::milliseconds(100 * i));
      std::scoped_lock l(mtx_A, mtx_B);
      std::cout << std::this_thread::get_id()
                << " thread_A: Acquired locks for A, B!" << std::endl;
    }
  });

  std::thread thread_B([]()->void {
    for (int i = 0; i < 5; i++) {
      std::this_thread::sleep_for(std::chrono::milliseconds(100 * i));
      std::scoped_lock l(mtx_B, mtx_A);
      std::cout << std::this_thread::get_id()
                << " thread_B: Acquired locks for B, A!" << std::endl;
    }
  });

  thread_A.join();
  thread_B.join();
}

int main(const int argc, const char * argv[]) {
  std::cout << "main() thread id: " << std::this_thread::get_id() << std::endl;
```

```
  problem();

  std::cout << "\nSolution:\n\n";
  solution();

  return 0;
}
```

The output of this program shows `A` requesting `B`, realizing it can't, and then releasing it's resource. What we can't see in this output is the activity of `thread_B` - this is just because it was very vertical, and I thought the output from `thread_A` was enough to show as an example.

The solution output shows the threads alternating ownership of the mutex locks between iterations.

```
main() thread id: 140684728141632
140684728137472 thread_A: Lock A
140684728137472 thread_A: Requesting B
140684728137472 thread_A: Can't lock B, unlocking A
140684728137472 thread_A: Lock A
140684728137472 thread_A: Requesting B
140684728137472 thread_A: Can't lock B, unlocking A
140684728137472 thread_A: Lock A
140684728137472 thread_A: Requesting B
140684728137472 thread_A: Can't lock B, unlocking A
140684728137472 thread_A: Lock A
140684728137472 thread_A: Requesting B
140684728137472 thread_A: Can't lock B, unlocking A
140684728137472 thread_A: Lock A
140684728137472 thread_A: Requesting B
140684728137472 thread_A: Acquired locks for A, B! Done.

Solution:

140684728137472 thread_B: Acquired locks for B, A!
140684719744768 thread_A: Acquired locks for A, B!
140684728137472 thread_B: Acquired locks for B, A!
140684719744768 thread_A: Acquired locks for A, B!
140684728137472 thread_B: Acquired locks for B, A!
140684719744768 thread_A: Acquired locks for A, B!
```

```
140684728137472 thread_B: Acquired locks for B, A!
140684719744768 thread_A: Acquired locks for A, B!
140684728137472 thread_B: Acquired locks for B, A!
140684719744768 thread_A: Acquired locks for A, B!
```

References

Some other references that I found online while working on these examples.

approxion: lock_guard vs scoped_lock

bogotobogo: multithreaded C++

deathbytape - C++ threading

acodersjourney - 20 threading mistakes

zitoc - Process Lifecycle

Revision #11
Created 1 April 2022 23:19:59 by Shaun Reed
Updated 21 April 2022 14:02:06 by Shaun Reed