Atomic Workshop

Lazy initialization

- We saw earlier in the course how to perform the following lazy initialization with thread-safety by using double-checked locking with std::call_once

```
some_type *ptr{ nullptr };      // Variable to be lazily initialized

void process() {
   if (!ptr)                    // First time variable has been used
      ptr = new some_type;      // Initialize it
   ptr->do_it();                // Use it
}
```

- In this exercise, make the lazy initialization thread-safe by using an atomic type
- Write a simple program to exercise your code

Lock-free queue

- Consider the following code to implement a simple queue without any internal or external locks
- The queue is only accessed by two threads
  - A Producer thread inserts elements into the queue
  - A Consumer thread removes elements from the queue
- The code is carefully designed so that the Consumer and Producer threads never work on adjacent elements
  - This ensures that the two threads always work on different parts of the queue

Lock-free queue

- The queue has two iterators, iHead and iTail
  - iHead points to the element before the oldest element
  - iTail points to the element before the newest (most recently added)
  - When the Producer thread adds an element, it increments iTail
  - When the Consumer thread removes an element, it increments iHead
- Only the Producer thread modifies the queue
  - As well as inserting elements, the Producer queue is responsible for erasing elements that the Consumer thread has removed
- The Producer thread never erases the iHead element
  - This maintains separation between the threads

Lock-free queue class

```
template <typename T>
struct LockFreeQueue {
  private:
    std::list<T> list;
    typename std::list<T>::iterator iHead, iTail;

  public:
    LockFreeQueue() {
        list.push_back(T());              // Add dummy separator
        iHead = list.begin();
        iTail = list.end();
    }
```

Producer task member function

```
void Produce(const T& t) {
   list.push_back(t);                  // Add the new item
   iTail = list.end();                 // Publish it
   list.erase(list.begin(), iHead);    // Trim unused nodes
}
```

Consumer task member function

```
bool Consume(T& t) {
    auto iNext = iHead;
    ++iNext;
    if (iNext != iTail) {          // If queue is not empty
        iHead = iNext;             // Publish that we took an item
        t = *iHead;                // Copy it back to the caller
        return true;               // And report success
    }
    return false;                  // Else report queue was empty
}
}; // End of class definition
```

Exercise

- Why is the "dummy separator necessary?
- Add a member function to print out all the elements
- Add a main function which calls Produce and Consume in separate threads
- Write a loop which runs the Produce and Consume threads and, once they have completed, calls Print
- Increase the number of iterations until you observe a race condition
- Explain why the race condition occurs
- Can the race condition be avoided by using atomic variables?