Thread-safe Linked List Solutions

Thread-safety

- A program has a linked list object which is shared between multiple threads
- You have been asked to write a function which traverses the elements of the linked list
- What issues do you need to consider in relation to data races?
  - When traversing a linked list, we go from an node to its following node by derefencing its "next" pointer
  - We have a data race if another thread can modify the data in the node while we are reading it

Thread-safety

- Another thread could insert a new following node after we get the next pointer address, but before we dereference it. Our traversal function would not be aware of this node. This may be a problem, because we do not accurately report the nodes in the list, but it will not cause data corruption
- Another thread could delete the following node after we get the next pointer address, but before we dereference it. Our traversal function would then dereference a pointer to an object which no longer exists. The program's behaviour will be undefined and it could behave unpredictably or crash

List Traversal

- Explain how adding a mutex member to the node can make your traversal function thread-safe
  - We lock the mutex when we first access the node
  - This prevents other threads from modifying the node's data while we are accessing it (assuming they respect the mutex)
  - This also prevents other threads from modifying the node's "next" pointer
  - However, if we unlock the current node before locking the following node, it is possible for another thread to interleave and erase the following node
  - We therefore need to lock the mutex in both the node we are currently accessing and its following node

Hand-over-hand Locking

- What is meant by "hand-over-hand" locking?
  - "Hand-over-hand" locking is a technique for working with concurrent list structures
  - It uses two locks
  - The mutexes in the current node "A" and the following node "B" are locked
  - When the thread has finished its work on node A, it unlocks A's mutex and locks the node following B, which is "C"
  - B's mutex remains locked
  - The thread then begins working on node B
  - It then locks C's following mutex, "D". C's mutex remains locked
  - In effect, the lock on the current node is transferred to the following node's following node
  - The effect is similar to a man climbing a ladder

Hand-over-hand Locking

- Which member functions of std::unique_lock are particularly helpful when implementing hand-over-hand locking?
  - The assignment operator
  - std::unique_lock(A->mut) = std::unique_lock(C->mut)
  - Atomically unlocks the mutex in A and locks the mutex in C
  - The swap member function
  - Atomically swaps the locks on two mutexes

Thread-safe List Traversal

- Modify the linked link class you implemented in the previous lecture's exercises, so that its traversal member function is thread-safe
- Write a program which
  - Creates an object of this list class and populates it, in the main thread
  - Starts two threads which call the traversal member function
- Check that your program compiles and runs correctly