Mutex Introduction Exercises

Copyright © James Raynard 2020

Mutex introduction

- Explain what is meant by the terms "critical section" and "mutex" in relation to multi-threaded programs
  - "Critical section" describes code that must execute without being interfered with by other parts of the program
  - In multi-threaded programs, critical sections usually arise when shared data can be modified by different threads
  - A "mutex" is a "mutual exclusion object"
  - It has two states, locked and unlocked

Copyright © James Raynard 2020

Mutex introduction

- How can mutexes be used with critical sections?
  - We can associate a mutex with a critical section
  - Each thread that accesses the critical section locks the mutex before entering it and unlocks it on leaving the critical section
  - If the mutex is already locked when the thread is about to enter the critical section, the thread must wait until the mutex is unlocked
  - If all threads respect this convention, each thread's access "happens before" the next thread's access
  - Correct use of mutexes prevents data races

Copyright © James Raynard 2020

Mutex introduction

- Briefly describe the C++ std::mutex
  - std::mutex is a class which implements a mutual exclusion object
  - It is defined in <mutex>
  - It has three main member functions
  - lock() tries to lock the mutex. If it is already locked, it waits until the mutex is unlocked, then locks it
  - try_lock tries to lock the mutex. If it is already locked, it returns immediately
  - unlock() unlocks the mutex

Copyright © James Raynard 2020

Rewrite using std::mutex

- Rewrite the "scrambled output" program using a mutex to protect the output operations
- Verify that the output is not scrambled when there are ten concurrent threads running

Copyright © James Raynard 2020

Rewrite using std::mutex

- What happens if mutex::unlock is not called?
  - One set of output appears (for example, "abc") before the program hangs
- Explain the results
  - The threads launch and call mutex::lock()
  - One thread gets the lock and its call returns immediately
  - The other threads wait for their call to return
  - The running thread prints its output
  - The running thread goes to the second iteration of its loop
  - The running thread calls mutex::lock() again
  - The mutex is already locked, so it waits for the call to return
  - All the threads are deadlocked - the mutex will never be unlocked
  - The main thread, which joins on these deadlocked threads, is also deadlocked

Copyright © James Raynard 2020

Rewrite using std::mutex

- Alter your program so that an exception is thrown between the output statement and the unlock call
- Add a catch handler at the end of the loop to handle the exception
- What happens when you run the program? Explain your results.
  - The program prints "abc", then the output from the exception handler, then hangs
  - The exception is thrown on the first iteration in the first thread which runs
  - When the exception is thrown, the destructors are called for all objects in scope and the program jumps into the catch handler
  - The unlock call is never executed
  - All the threads which are waiting for the lock will be blocked
  - main() is joined on these blocked threads and will be blocked as well

Copyright © James Raynard 2020

Mutexes and data

- Implement a simple thread-safe wrapper for the std::vector class, Vector
  - This only stores ints
  - It has a push_back() member function. This uses a mutex to protect calls to the std::vector push_back()
  - Provide a print() function to display all its elements
- Write a thread entry function that calls push_back 5 times, with a 50ms sleep after each call
- Write a program with a global Vector instance that starts ten of these threads and then prints out all the elements of the Vector
- Replace the Vector instance with a standard vector<int>

Copyright © James Raynard 2020

Mutexes and data

- Explain your results
  - With the threadsafe Vector class, the program runs normally
  - With std::vector, the program exhibits undefined behaviour (on my system, it crashed and did not display any output)
  - This is because a data race can occur if multiple threads access the same std::vector object
  - In Vector, accesses to the underlying vector are protected by a mutex, which imposes a "happens before" relationship between thread accesses
  - A data race cannot happen with a shared Vector object, but can happen with an std::vector

Copyright © James Raynard 2020

try_lock()

- Write a program which runs two task functions in separate threads
- The first task function locks a mutex, sleeps for 500ms and releases the lock
- The second task function sleeps for 100ms, then calls try_lock() to lock the mutex. If unsuccessful, it sleeps again for 100ms and calls try_lock() again. If successful, it unlocks the mutex
- Add suitable print statements and run the program

Copyright © James Raynard 2020

try_lock()

- What do you observe?
  - Task1 gets the lock first (because of the sleep in Task2)
  - Task2 repeatedly calls try_lock unsuccessfully, because it is locked by Task1
  - Finally, Task1 releases the lock and Task2's try_lock() call succeeds

```
Task1 trying to get lock
Task1 has lock
Task2 trying to get lock
Task2 could not get lock
Task2 could not get lock
Task2 could not get lock
Task2 could not get lock
Task1 releasing lock
Task2 has lock
```

Copyright © James Raynard 2020