Lab Solution 5: Read-Write Locks, Event Barriers

Students are encouraged to share their ideas in the #lab5 Slack channel. SCPD students are welcome to reach out to me directly if they have questions that can't be properly addressed without being physically present for a discussion section.

Before starting, go ahead and clone the `lab5` folder, which contains the test framework for the `EventBarrier` class discussed in Problem 2.

```
poohbear@myth15:~$ hg clone /usr/class/cs110/repos/lab5/shared lab5
poohbear@myth15:~$ cd lab5
poohbear@myth15:~$ make
```

Solution 1: Read-Write Locks

The read-write lock (implemented by the `rwlock` class) is a `mutex`-like class with three `public` methods:

```


 private:
  // object state omitted
};class rwlock {
 public:
  rwlock();
  void acquireAsReader();
  void acquireAsWriter();
  void release();

 private:
  // object state omitted
};
```

Any number of threads can acquire the lock as a reader without blocking one another. However, if a thread acquires the lock as a **writer**, then all other `acquireAsReader` and `acquireAsWriter` requests block until the writer releases the lock. Waiting for the write lock will block until all readers release the lock so that the writer is guaranteed exclusive access to the resource being protected. This is useful if, say, you use some shared data structure that only very periodically needs to be modified. All reads from the data structure require you to hold the reader lock (so as many threads as you want can read the data structure at once), but any writes require you to hold the writer lock (giving the writing thread exclusive access).

The implementation ensures that as soon as one thread **tries** to get the writer lock, all other threads trying to acquire the lock–either as a reader or a writer–block until that writer gets the locks and releases it. That means the state of the lock can be one of three things:

- **Ready**, meaning that no one is trying to get the write lock.
- **Pending**, meaning that someone is trying to get the write lock but is waiting for all the readers to finish.
- **Writing**, meaning that someone is writing.

The leanest implementation I could come up with relies on two `mutex`es and two `condition_variable_any`s. Here is the full interface for the `rwlock` class:

```
 private:
  int numReaders;
  enum { Ready, Pending, Writing } writeState;
  mutex readLock, stateLock;
  condition_variable_any readCond, stateCond;
};class rwlock {
 public:
  rwlock(): numReaders(0), writeState(Ready) {}
  void acquireAsReader();
  void acquireAsWriter();
  void release();

 private:
  int numReaders;
  enum { Ready, Pending, Writing } writeState;
  mutex readLock, stateLock;
  condition_variable_any readCond, stateCond;
};
```

And here are the implementations of the three `public` methods:

```cpp
void rwlock::acquireAsReader() {
  lock_guard<mutex> lgs(stateLock);
  stateCond.wait(stateLock, [this]{ return writeState == Ready; });
  lock_guard<mutex> lgr(readLock);
  numReaders++;
}

void rwlock::acquireAsWriter() {
  stateLock.lock();
  stateCond.wait(stateLock, [this]{ return writeState == Ready; });
  writeState = Pending;
  stateLock.unlock();
  lock_guard<mutex> lgr(readLock);
  readCond.wait(readLock, [this]{ return numReaders == 0; });
  writeState = Writing;
}

void rwlock::release() {
  stateLock.lock();
  if (writeState == Writing) {
    writeState = Ready;
    stateLock.unlock();
    stateCond.notify_all();
    return;
  }

  stateLock.unlock();
  lock_guard<mutex> lgr(readLock);
  numReaders--;
  if (numReaders == 0) readCond.notify_one();
}
```

Very carefully study the implementation of the three methods, and answer the questions that appear below. This lab problem is designed to force you to really internalize the `condition_variable_any`–the most difficult concept in the entire threading segment of the course, in my opinion–and understand how it works.

- The implementation of `acquireAsReader` acquires the `stateLock` (via the `lock_guard`) before it does anything else, and it doesn't release the `stateLock` until the method exits. Why can't the implementation be this instead?

  ```
  void rwlock::acquireAsReader() {
    stateLock.lock();
    stateCond.wait(stateLock, [this]{ return writeState == Ready; });
    stateLock.unlock();
    lock_guard<mutex> lgr(readLock);
    numReaders++;
  }
  ```

  - *Assume just two threads:*
    - *Thread 1 calls `acquireAsReader` and is swapped off after third of five lines.*
    - *Thread 2 calls and progresses through all of `acquireAsWriter`.*
    - *Thread 1 progresses through rest of `acquireAsReader`.*
  - *We have one reader and one writer, and that's forbidden.*
- The implementation of `acquireAsWriter` acquires the `stateLock` before it does anything else and it releases the `stateLock` just before it acquires the `readLock`. Why can't `acquireAsWriter` adopt the same approach as `acquireAsReader` and just hold onto `stateLock` until the method returns?
  - *If the writer doesn't release `stateLock` before waiting for the number of readers to fall to 0, it blocks readers trying to release their locks from decrementing `numReaders`.*
- Notice that we have a single `release` method instead of `releaseAsReader` and `releaseAsWriter` methods. How does the implementation know if the thread acquired the `rwlock` as a writer instead of a reader (assuming proper use of the class)?
  - *The implementation is such that the write state can only be `Writing` when there's one write lock and zero read locks. When the write state is `Writing`, then only one thread could possibly be calling `release`, unless the class is being used improperly.*
- The implementation of release relies on `notify_all` in one place and `notify_one` in another. Why are those the correct versions of `notify` to call in each case?
  - *Any number of threads might be waiting for a `Ready` state so they can advance to acquire read locks. The writer needs to notify all of them when it releases the write lock. At most one thread–a* ***writer***–*can be waiting for the number of readers to be 0, so calling `notify_one` is sufficient.*

- A thread that owns the lock as a reader might want to upgrade its ownership of the lock to that of a writer without releasing the lock first. Besides the fact that it's a waste of time, what's the advantage of not releasing the read lock before re-acquiring it as a writer, and how could be the implementation of `acquireAsWriter` be updated so it can be called after `acquireAsReader` without an intervening release call?
    - *One advantage: fewer `mutexes` and `condition_variable_anys` need to be waited on, so the chance that the threads trying to upgrade the lock are forced to yield the processor is much, much smaller.*
    - *Another advantage: using the underlying thread (which are `pthreads`) and its support for priorities, you can give threads that are trying to upgrade higher priority so they get the processor before lower priority threads do.*
    - *Implementation idea: change the `acquireAsWriter` to accept a `bool` to state whether it holds a read lock already.*
    - *Better implementation idea: update the `rwlock` to maintain a set of thread ids (which are all the underlying `pthread_t`'s really are) that hold a read lock, and if the thread trying to upgrade finds its thread id in the set, then it knows it's upgrading. It can then wait until `numReaders == numUpgraders` instead of `numReaders == 0`. Couple this with higher thread priorities and you can ensure that exactly one upgrading thread succeeds while all others wait. It's true that all of the other upgraders technically have a read lock, but they're blocked inside `acquireAsWriter`, so they're not actually reading whatever data structure is being accessed, and that's okay.*

Solution 2: Event Barriers

An event barrier allows a group of one or more threads—we call them *consumers*—to efficiently `wait` until an event occurs (i.e. the barrier is `lift`ed by another thread, called the *producer*). The barrier is eventually restored by the producer, but only after consumers have detected the event, executed what they could only execute because the barrier was lifted, and notified the producer they've done what they need to do and moved `past` the barrier. In fact, consumers and producers efficiently block (in `lift` and `past`, respectively) until all consumers have moved past the barrier. We say an event is *in progress* while consumers are responding to and moving past it.

The `EventBarrier` implements this idea via a constructor and three zero-argument methods called `wait`, `lift`, and `past`. The `EventBarrier` requires no external synchronization, and maintains enough internal state to track the number of waiting consumers and whether an event is in progress. If a consumer arrives at the barrier while an event is in progress, `wait` returns immediately without blocking.

The following test program (where all `oslock`s and `osunlock`s have been removed, for brevity) and sample run illustrate how the `EventBarrier` works:

```
static void gatekeeper(EventBarrier& eb) {
  sleep(random() % 5 + 7);
  cout << "Gatekeeper raises the drawbridge gate." << endl;
  eb.lift(); // lift the drawbridge
  cout << "Gatekeeper lowers drawbridge gate knowing all have crossed." << endl;
}

static string kMinstrelNames[] = {"Peter", "Paul", "Mary"};
static const size_t kNumMinstrels = 3;
int main(int argc, char *argv[]) {
  EventBarrier drawbridge;
  thread minstrels[kNumMinstrels];
  for (size_t i = 0; i < kNumMinstrels; i++)
    minstrels[i] = thread(minstrel, kMinstrelNames[i], ref(drawbridge));
  thread g(gatekeeper, ref(drawbridge));
  for (thread& c: minstrels) c.join();
  g.join();
  return 0;
}
```

```
myth15$ ./event-barrier-test
Peter walks toward the drawbridge.
Paul walks toward the drawbridge.
Mary walks toward the drawbridge.
Mary arrives at the drawbridge gate, must wait.
Paul arrives at the drawbridge gate, must wait.
Peter arrives at the drawbridge gate, must wait.
Gatekeeper raises the drawbridge.
Paul detects drawbridge gate lifted, starts crossing.
Peter detects drawbridge gate lifted, starts crossing.
Mary detects drawbridge gate lifted, starts crossing.
```

```
Paul has crossed the bridge.
Mary has crossed the bridge.
Peter has crossed the bridge.
Gatekeeper lowers drawbridge gate knowing all have crossed.static void
minstrel(const string& name, EventBarrier& eb) {
  cout << name << " walks toward the drawbridge." << endl;
  sleep(random() % 3 + 3); // minstrels arrive at gate at different times
  cout << name << " arrives at the drawbridge gate, must wait." << endl;
  eb.wait(); // all minstrels wait until drawbridge gate is raised
  cout << name << " detects drawbridge gate lifted, starts crossing." << endl;
  sleep(random() % 3 + 2); // minstrels walk at different rates
  cout << name << " has crossed the bridge." << endl;
  eb.past();
}

static void gatekeeper(EventBarrier& eb) {
  sleep(random() % 5 + 7);
  cout << "Gatekeeper raises the drawbridge gate." << endl;
  eb.lift(); // lift the drawbridge
  cout << "Gatekeeper lowers drawbridge gate knowing all have crossed." << endl;
}

static string kMinstrelNames[] = {"Peter", "Paul", "Mary"};
static const size_t kNumMinstrels = 3;
int main(int argc, char *argv[]) {
  EventBarrier drawbridge;
  thread minstrels[kNumMinstrels];
  for (size_t i = 0; i < kNumMinstrels; i++)
    minstrels[i] = thread(minstrel, kMinstrelNames[i], ref(drawbridge));
  thread g(gatekeeper, ref(drawbridge));
  for (thread& c: minstrels) c.join();
  g.join();
  return 0;
}

myth15$ ./event-barrier-test
Peter walks toward the drawbridge.
Paul walks toward the drawbridge.
Mary walks toward the drawbridge.
Mary arrives at the drawbridge gate, must wait.
Paul arrives at the drawbridge gate, must wait.
Peter arrives at the drawbridge gate, must wait.
Gatekeeper raises the drawbridge.
Paul detects drawbridge gate lifted, starts crossing.
Peter detects drawbridge gate lifted, starts crossing.
Mary detects drawbridge gate lifted, starts crossing.
Paul has crossed the bridge.
Mary has crossed the bridge.
Peter has crossed the bridge.
Gatekeeper lowers drawbridge gate knowing all have crossed.
```

The backstory for the above sample run: three singing minstrels approach a castle only to be blocked by a drawbridge gate. The three minstrels wait until the gatekeeper lifts the gate, allowing the minstrels to cross. The gatekeeper only lowers the gate after all three minstrels have crossed the bridge, and the three minstrels only proceed toward the castle once all three have cross the bridge.

Your `lab5` folder includes `event-barrier.h`, `event-barrier.cc`, and `ebtest.cc`, and typing `make` should generate an executable called `ebtest` that you can run to ensure that the `EventBarrier` class you'll flesh out in `event-barrier.h` and `.cc` are working properly. The one exercise in this lab that has you do any coding is this one, as it expects you complete the implementation stub you've been supplied with.

- *Here's the full interface file:*

  ```
  private:
    // use the space within the dashed rectangle
    size_t numWaiting;
    bool occurring;
    std::mutex m;
  ```

```
    std::condition_variable_any cv;
};class EventBarrier {
public:
  EventBarrier();
  void wait();
  void lift();
  void past();

private:
  // use the space within the dashed rectangle
  size_t numWaiting;
  bool occurring;
  std::mutex m;
     std::condition_variable_any cv;
};
```

- *And here's my implementation:*

```


void EventBarrier::wait() {
lock_guard<mutex> lg(m);
  numWaiting++;
  cv.wait(m, [this] { return occurring; });
}

void EventBarrier::lift() {
  lock_guard<mutex> lg(m);
  occurring = true;
  cv.notify_all();
  cv.wait(m, [this] { return numWaiting == 0; });
  occurring = false;
}

void EventBarrier::past() {
  lock_guard<mutex> lg(m);
  numWaiting-;
  if (numWaiting > 0) {
    cv.wait(m, [this] { return numWaiting == 0; });
  } else {
    cv.notify_all();
  }
}EventBarrier::EventBarrier(): numWaiting(0), occurring(false) {}

void EventBarrier::wait() {
lock_guard<mutex> lg(m);
  numWaiting++;
  cv.wait(m, [this] { return occurring; });
}

void EventBarrier::lift() {
  lock_guard<mutex> lg(m);
  occurring = true;
  cv.notify_all();
  cv.wait(m, [this] { return numWaiting == 0; });
  occurring = false;
}

void EventBarrier::past() {
  lock_guard<mutex> lg(m);
  numWaiting--;
  if (numWaiting > 0) {
    cv.wait(m, [this] { return numWaiting == 0; });
  } else {
    cv.notify_all();
  }
}
```