Lab Handout 5: Threads vs Processes, 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 or Hemanth directly if they have questions that can't be properly addressed without being physically present for a discussion section. The lab checkoff sheet for all students–both on-campus and off–can be found right here.

Before starting, go ahead and clone the `lab5` folder, which contains the test framework for the `EventBarrier` class discussed in Problem 4.

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

Problem 1: Threads vs Processes

According to some CAs, a good number of students are pondering the pros and cons of threads versus processes. Some have even asked what the pros and cons are. To provide some answers, we'll lead you through a collection of short answer questions about multiprocessing and multithreading that focuses more on the big picture.

- What does it mean when we say that a process has a private address space?
- What are the advantages of a private address space?
- What are the disadvantages?
- What programming directives have we used in prior assignments and discussion section handouts to circumvent address space privacy?
- In what cases do the processes whose private address spaces are being publicized have any say in the matter?
- When architecting a larger program like `farm` or `stsh` that relies on multiprocessing, what did we need to do to exchange information across process boundaries?
- Can a process be used to execute multiple executables? Restated, can it `execvp` twice to run multiple programs?
- Threads are often called lightweight processes. In what sense are they processes? And why the lightweight distinction?
- Threads are often called virtual processes as well. In what sense are threads an example of virtualization?
- Threads running within the same process all share the same address space. What are the advantages and disadvantages of allowing threads to access pretty much all of virtual memory?

Each thread within a larger process is given a thread id, also called a **tid**. In fact, the thread id concept is just an extension of the process id. For singly threaded processes, the pid and the main thread's tid are precisely the same. If a process with pid 12345 creates three additional threads beyond the main thread (and no other processes or threads within other processes are created), then the tid of the main thread would be 12345, and the thread ids of the three other threads would be 12346, 12347, and 12348.

- What are the advantages of leveraging the pid abstraction for thread ids?
- What happens if you pass a thread id that isn't a process id to `waitpid`?
- What happens if you pass a thread id to `sched_setaffinity`?
- What are the advantages of requiring that a thread always be assigned to the same CPU?

In some situations, the decision to rely on multithreading instead of multiprocessing is dictated solely by whether the code to be run apart from the main thread is available in executable or in library form. But in other situations, we have a choice.

- Why might you prefer multithreading over multiprocessing if both are reasonably good options?
- Why might you prefer multiprocessing over multithreading if both are reasonably good options?
- What happens if a thread within a larger process calls `fork`?
- What happens if a thread within a larger process calls `execvp`?

And some final questions about how `farm`, `aggregate`, and `stsh` could have been implemented:

- Assume you know nothing about multiprocessing but needed to emulate the functionality of `farm`. How could you use multithreading to design an equally parallel solution without ever relying on multiprocessing?
- Inversely, how could you have implemented `aggregate` to rely on multiprocessing (without threading) to arrive at an equally parallel solution?
- Could multithreading have contributed to the implementation of `stsh` in any meaningful way? Why or why not?

Problem 2: Threading Short Answer Questions

Here are some more short answer questions about the specifics of threads and the directives that help threads communicate.

- Is `i--` thread safe on a single core machine? Why or why not?
- What's the difference between a `mutex` and a `semaphore` with an initial value of 1? Can one be substituted for the other?

- What is the `lock_guard` class used for, and why is it useful?
- What is busy waiting? Is it ever a good idea? Does your answer to the good-idea question depend on the number of CPUs your multithreaded application has access to?
- As it turns out, the semaphore's constructor allows a negative number to be passed in, as with `semaphore s(-11)`. Identify a scenario where -11 might be a sensible initial value.
- What would the implementation of `semaphore::signal(size_t increase = 1)` need to look like if we wanted to allow a `semaphore`'s encapsulated value to be promoted by the `increase` amount? Note that `increase` defaults to 1, so that this version could just replace the standard `semaphore::signal` that's officially exported by the `semaphore` abstraction.
- What's the multiprocessing equivalent of the `mutex`?
- What's the multiprocessing equivalent of the `condition_variable_any`?

Problem 3: 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:

```
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++;
  }
  ```

- 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?
- 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 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?
- 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?

Problem 4: 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.