Winter 2014: CS110 Final Examination Solution

Solution 1: `fork` and `signal`

a. Which single line of the output is incorrect, and what should it be?

The last line must print a **`1`**, since the original parent process is never signaled, and hence its counter is incremented by one just one time.

Criteria for Problem 1a: 2 points

- 2 points for correct answer, 0 points for incorrect answer

b. Which two lines might be exchanged by another test run? Why can that happen?

The first two lines might be exchanged, since the child's first **`printf`** and the grandchild's first **`printf`** can execute in either order. This can happen because there are no **`waitpid`** or **`kill`** calls prior to either of them, so each can progress toward their first **`printf`** without being blocked or stalled by another. (It's possible other pairs of lines are exchanged, but the published **`counter`**s would be different.)

Criteria for Problem 1b: 2 points

- 2 points for correct answer, 0 points for incorrect answer

c. Which line might be missing altogether from another test run? Why can that happen?

The **`7500`** might be missing, because the grandchild could exit before the child signals it.

Criteria for Problem 1c: 2 points

- 2 points for correct answer, 0 points for incorrect answer

Solution 2: `ThreadPool` and Office Hours

a. First, declare your global variables, ensuring that all of them are properly initialized. Because each of the TA's can work concurrently with different students, you need to maintain an array of **`struct`**s—one **`struct`** for each TA! You'll also need a few isolated global variables.

You should only need to make use of primitive types, **`mutex`**es, and **`semaphore`**s (no **`condition_variable`**s can be used for this problem.) For this problem you can just note what each of the global variables and **`struct`** fields should be initialized to if they aren't properly initialized by default.

```
static struct ta {
  mutex available;
  semaphore requested;
  semaphore finished;
  size_t races; // doesn't need to be initialized
} tas[kNumTAs];

static semaphore availablePowerOutlets(kNumPowerOutlets);
static size_t numStudents = kNumStudents;
static mutex numStudentsLock;
```

Criteria for Problem 2a: 7 points

- Includes a field in each **`struct`** to ensure at most one student is being helped by a TA at any one time: 1 point
- Includes a field in each **`struct`** where the number of race conditions can be shared with the student being helped: 1 point
- Includes a semaphore in each **`struct`** that can be used to signal the TA that a student is present: 1 point
- Includes a second **`semaphore`** in each **`struct`** that can be used to signal a student that his code has been reviewed: 1 point
- Includes a global **`semaphore`** to manage the number of available outlets: 1 point (0 points if global **`int`** with companion **`mutex`**—leads to busy waiting)
- Global count on the number of students: 1 point (if they use **`atomic<int>`**, ok!)
- Global **`mutex`** guarding the global count: 1 point (not needed for **`atomic<int>`**)

b. Using this and the next page, present your implementation of the **`ta`** and **`student`** thread routines. Be sure to avoid race conditions, deadlock, and busy waiting.

```
static void ta(size_t id) {
  while (true) {
    tas[id].requested.wait();
    if (numStudents == 0) return;
    tas[id].races = review();
    tas[id].finished.signal();
    grade();
  }
}
```

Criteria for Problem 2b, `ta` routine: 6 points

- Efficiently blocks until a student wakes her up: 2 points
- Examines the global student count and returns on 0: 1 point
- **`review`**s the student code and places value in location that its student knows about: 1 point
- notifies the student that his code has been reviewed: 2 points
- Non-CS110 issues like **`grade`**, **`while (true)`**, etc: 0 points

```
static void student() {
  size_t races = 1;
  availablePowerOutlets.wait();
  for (size_t i = 0; races > 0 && i < kMaxNumRounds; i++) {
    debug();
    size_t id = random();
    tas[id].available.lock();
    tas[id].requested.signal();
    tas[id].finished.wait();
    races = tas[id].races;
    tas[id].available.unlock();
  }

  if (races == 0) squeal();
  submit();
  availablePowerOutlets.signal();
  lock_guard<mutex> lg(numStudentsLock);
  numStudents--;
  if (numStudents == 0)
    for (struct ta& ta: tas)
      ta.requested.signal();
}
```

Criteria for Problem 2b, `student` routine: 12 points

- Efficiently waits for a power outlet to become available: 1 point
- Selects a random TA and efficiently blocks until he becomes available: 2 points
- Signals the TA that she is needed for a code review: 2 points
- Waits until the TA has reviewed his code: 1 point
- Pulls the number of race conditions into his scope so he can act on it: 1 point
- Allows the TA to move on to other students: 1 point
- Eventually stops coding: 0 points
- **`signal`**s that one more power outlet is available: 1 point
- Atomically decrements the number of students needing help: 1 point
- **`signal`**s all of the TAs when he notices all students have been helped: 2 points
- **`squeal`**, **`submit`**, etc are all for back story, but not worth anything: 0 points

Solution 3: Read-Write Locks

a. 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.

Criteria for Problem 3a: 2 points

- 2 points for perfectly clear, correct answer
- 1 point for ambiguous answer that could be interpreted as correct
- 0 points for incorrect answer or answer that's clearly too long

b. 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`**.

Criteria for Problem 3b: 2 points

- 2 points for perfectly clear, correct answer
- 1 point for ambiguous answer that could be interpreted as correct
- 0 points for incorrect answer or answer that's clearly too long

c. 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.

Criteria for Problem 3c: 3 points

- 3 points for perfectly clear, correct answer
- 1 point for ambiguous answer that could be interpreted as correct
- 0 points for incorrect answer or answer that's clearly too long

d. 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.

Criteria for Problem 3d: 3 points

- 3 points for perfectly clear, correct answer
- 1 point for ambiguous answer that could be interpreted as correct
- 0 points for incorrect answer or answer that's clearly too long

e. [5 points] 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?

We should accept a huge variety of answers on this problem, because it was intended to be open-ended and an opportunity to communicate advanced understanding of threading, race conditions, deadlock threat, and concurrency primitives.

One advantage: fewer **`mutex`**es and **`condition_variable_any`**s 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 **`pthread`**s) 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.

Criteria for Problem 3e, advantage: 2 points

- Any reasonably stated advantage: 2 points
- Any vague answer that's short on detail: 1 point
- Leaving it blank: 0 points

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, so it's okay.)

Criteria for Problem 3e, advantage: 3 points

- Either changing the interface, maintaining a set of thread ids, or something else that would allow a thread to know if it's upgrading: 1 point
- Conveying any reasonably intelligent approach to implementing the upgrade concept: 2 points

Solution 4: Primary DNS Servers

a. Turn to the next page and complete the implementation of **`buildRequestHandler`** and the second method (you'll choose the name and the list of parameters) that constructs the routine to be executed within the **`outboundRequests`** pool.

```
function<void(void)> DNSServer::buildRequestHandler(int client) {
  return [this, client] {
    sockbuf rsb(client);
    iosockstream rss(&rsb);
    vector<string> names = pullAllNames(rss);
    map<string, vector<string>> workerRequests = compileMap(names);
    semaphore forwardedRequestsCompleted;
    mutex rssLock;
    for (const auto& request: workerRequests) {
      outboundRequests.schedule(
         buildForwardHandler(rss, rssLock,
                             request.first, request.second,
                             forwardedRequestsCompleted));
    }

    for (size_t i = 0; i < workerRequests.size(); i++)
      forwardedRequestsCompleted.wait();
    rss << endl;
  };
}
```

Criteria for Problem 4, `buildRequestHandler` method: 7 points

- Declares the shared **`mutex`** that protect secondary threads trying to publish partial responses to the original connection (or that some other solution is needed to guard against race conditions): 1 point
- Correctly uses the **`mutex`**: 1 point
- Declares a **`semaphore`** that can be signaled by the outbound threads so the primary thread knows when all secondary requests have responded: 1 point
- Correctly waits for the **`semaphore`** to be signaled once by each of the workers: 2 points
- Correctly schedules **some** helper thunk on **`outboundRequests`** for each of the secondaries needed: 1 point
- Aspects of passing by reference to sub-thunk-constructing method are correct: 1 point

```
function<void(void)>
DNSServer::buildForwardHandler(iosockstream& rss, mutex& rssLock,
                               const string& address,
                               const vector<string>& names,
                               semaphore& parent) {
  return [&] {
    int forward = createClientSocket(address, kDefaultPort);
    sockbuf fsb(forward);
    iosockstream fss(&fsb);
    for (const string& name: names) fss << name << endl;
    fss << endl;
    lock_guard<mutex> lg(rssLock);
    while (true) {
      string response;
      getline(fss, response);
      if (fss.fail()) break;
      rss << response << endl;
    }
    parent.signal();
  };
}
```

Criteria for Problem 4, `buildForwardHandler` method: 6 points

- Correctly captures all of the parameters needed (don't worry about the syntax—not important to get right without **`g++`**): 1 point
- Correctly calls **`createClientSocket`** and wraps an **`iosockstream`** around its return value: 1 point
- Writes the request to the secondary connection: 1 point
- Ingests the secondary response and writes it back to the client, but only after ensuring that no one else is writing to the same client connection (e.g. using **`mutex`**): 2 points
- Signals the parent that the partial response has been posted to the original client socket: 1 point

b. The architecture description is adamant that everything be done off of the main thread by making use of **`inboundRequests`**. What's so important about this type of system that we want to get everything off of the main thread as quickly as possible?

Establishing connections with and communicating with remote hosts involves slow system calls that could interfere with the main thread's ability to accept new connections.

Criteria for Problem 4b: 3 points

- 3 points for clear, correct response
- 2 points for a clear, correct response with a minor error
- 1 point for a vague response that could be interpreted as correct
- 0 points for an incorrect response, a response that's true but irrelevant, or a response that's clearly too long

c. [3 points] Why does the implementation use two **`ThreadPool`**s instead of one?

The inbound threads could hog the thread pool, preventing any outbound connection threads—the threads inbound threads depend on to get their work done—from making any progress. If outbound threads can't get work done, inbound threads never finish and never vacate the thread pool.

Criteria for Problem 4c: 3 points

- 3 points for clear, correct response
- 2 points for a clear, correct response with a minor error
- 1 point for a vague response that could be interpreted as correct
- 0 points for an incorrect response, a response that's true but irrelevant, or a response that's clearly too long

d. The translation algorithm used by our **`DNSServer`** is very similar to the translation algorithm used to translate absolute pathnames (e.g. **`/usr/class/cs110/bin/submit`**) to inode numbers. Describe three of these similarities.

- Both turn something human-readable into something more computer-compatible.
- Both manage a lookup of a number based on the tokenization of a name.
- Both have well-defined bootstrap locations (well-defined inode number for the root directory, well-known IP addresses of the primary servers consulted by initial DNS lookup)

Criteria for Problem 4d: 3 points

- 1 point for each clearly presented answer (there are other good answers, of course; only grade the first three.)

e. [3 points] The number of threads running between the two **`ThreadPool`**s can be as high as 64, even though there are only two processors on the myth machines. Why is 64 a reasonable number of threads for this type of application? For what type of application would 64 be an absurdly large number of threads?

All threads in this example are network bound, which means they'll spend the vast majority of their time waiting on work to be done off their own CPU. 64 would be too large if the threads ran routines that were local-CPU-bound.

Criteria for Problem 4c: 3 points

- 3 points for clear, correct response
- 2 points for a clear, correct response with a minor error
- 1 point for a vague response that could be interpreted as correct
- 0 points for an incorrect response, a response that's true but irrelevant, or a response that's clearly too long

Solution 5: Short Answers

a. When implementing **`proxy`**, we could have relied on multiprocessing instead of multithreading to support concurrent transactions. Very briefly describe one advantage of the multiprocessing approach over the multithreading approach, and briefly describe one disadvantage.

- Advantage: private address spaces and memory protection that comes with separate processes
- Disadvantage: difficult to share and synchronize on resources

Criteria for Problem 5a: 2 points

- Clearly and succinctly presents a genuine advantage: 1 point
- Clearly and succinctly presents a valid disadvantage: 1 point

b. Recall that virtualization is a systems principle where either many hardware resources are made to appear like one, or one hardware resource is made to appear like many. List four **distinct** forms of virtualization—implemented by the OS, by your **`assign6`** codebase, or by some other system—that contribute to the overall implementation of your Assignment 6 MapReduce system.

- Virtual-to-physical addressing: allows multiple processes to run.
- Threads as virtual processes: allows **`mr`** to manage concurrent connections to workers.
- AFS grafts independent file systems into to one: makes file sharing trivial
- Distributed workers: collectively work to produce a result the primary server could have generated standalone.

Criteria for Problem 5b: 4 points

- 1 point for each distinct example that truly contributes to MapReduce.

c. Request-response is one of the fundamental methods different computers (or different modules on the same computer) use to communicate and/or exchange information. One computer/module sends a request, and another responds. List three protocols, modules, or systems that rely on request/response that also contribute directly to your MapReduce implementation for Assignment 6.

- DNS, SSH, and or custom messaging protocols (can count as two or even all three)
- AFS
- system calls like **`socket`**, **`read`**, **`bind`**, etcetera

Criteria for Problem 5c: 3 points

- 1 point for each distinct, valid example