Winter 2014: CS110 Final Examination

This is a closed book, closed note, closed computer exam. You have 180 minutes to complete all problems. You don't need to **`#include`** any libraries, and you needn't guard against any errors unless specifically instructed to do so. Understand that the majority of points are awarded for concepts taught in CS110. If you're taking the exam remotely, you can telephone me at 415-205-2242.

Good luck!

SUNet ID (username): _____**`@stanford.edu`**

Last Name: _____

First Name: _____

I accept the letter and spirit of the honor code. I've neither given nor received aid on this exam. I pledge to write more neatly than I ever have in my entire life.

[signed] _____

		Score	Grader
1. **`fork`** and **`signal`**	[6]	_____	_____
2. **`ThreadPool`** Mania	[25]	_____	_____
3. Read/Write Locks	[15]	_____	_____
4. Primary DNS Node	[25]	_____	_____
5. Short Answers	[9]	_____	_____
Total	**[80]**	_____	_____

Relevant Prototypes

```
// exceptional control flow and multiprocessing
pid_t fork();
pid_t waitpid(pid_t pid, int *status, int flags);
typedef void (*sighandler_t)(int sig);
sighandler_t signal(int signum, sighandler_t handler); // ignore retval
int execvp(const char *path, char *argv[]); // ignore retval

// thread
class thread {
public:
   thread(...); // first argument is thread routine, its args come afterwards
   void join();
};

class mutex {
public:
   mutex();
   void lock();
   void unlock();
};

class semaphore {
public:
   semaphore(int count = 0);
   void wait();
   void signal();
};

class condition_variable_any {
public:
   template <typename Mutex, typename Pred>
      void wait(Mutex& m, Pred pred);
   void notify_one();
   void notify_all();
};

class ThreadPool {
public:
   ThreadPool(size_t numThreads);
   void schedule(function<void(void)>& thunk>;
   void wait();
};

vector<T> class supports, among other things,
   • void push_back(const T& elem);
   • const T& operator[](size_t index);
   • size_t size() const;

map<Key, Value> class supports, among other things,
   • Value& operator[](const Key& key);
   • size_t size() const;

Create a client connection to a server:
   int createClientSocket(const string& serverName, unsigned short port);

sockbuf instances constructed via sockbuf::sockbuf(int socket);
iosockstream instances constructed via iosockstream::iosockstream(sockbuf *sb);
```

Problem 1: `fork` and `signal` [6 points]

Consider the following program:

```
static pid_t pid;
static int counter = 0;

static void handlerOne(int sig) {
  counter += 7000;
  printf("counter = %d\n", counter);
}

static void handlerTwo(int sig) {
  counter += 20000;
  handlerOne(sig);
  kill(pid, SIGUSR1);
}

int main(int argc, char *argv[]) {
  signal(SIGUSR1, handlerOne);
  if ((pid = fork()) == 0) {
    signal(SIGUSR1, handlerTwo);
    if ((pid = fork()) == 0) {
      counter += 500;
      printf("counter = %d\n", counter);
      signal(SIGUSR1, handlerOne);
      kill(getppid(), SIGUSR1);
      exit(0);
    }
    counter += 30;
    printf("counter = %d\n", counter);
  }
  counter++;
  waitpid(pid, NULL, 0);
  printf("counter = %d\n", counter);
  return 0;
}
```

The above program is capable of printing out something very close to the following:

```
counter = 500
counter = 30
counter = 27031
counter = 7500
counter = 27031
counter = 2
```

a. [2 points] Which single line of the output is incorrect, and what should it be?

b. [2 points] Which two lines might be exchanged by another test run? Why can that happen?

c. [2 points] Which line might be missing altogether from another test run? Why can that happen?

Problem 2: `ThreadPool` and Office Hours [25 points]

You are developing a multithreaded C++ program to simulate CS110 office hours at Lathrop Library the night before the **`ThreadPool`** assignment is due. All seven CS110 TA's are there to point out your race conditions so you can debug your work. (In fact, all seven TA's stay until the very last CS110 student leaves—they heart CS110 students.)

All CS110 students arrive at Meyer library with their laptops and what they assume to be a single race condition in their otherwise perfect **`ThreadPool`** implementations. Each student waits for one of 15 power outlets to become available (because CS110 students have laptops with batteries that don't maintain a charge, so all laptops must be plugged in.) Once a student plugs in, he debugs his code and then selects a **`random`** TA to review his work. The TA counts the number of race conditions and shares that number with the student. If there aren't any race conditions, then the student **`squeal`**s with delight, runs the CS110 **`submit`** script, unplugs his laptop, and leaves. If his code still has one or more race conditions, then he repeats the debug-for-a-bit-and-get-random-TA-to-help process. If after 5 rounds the student's code still has problems, he **`submit`**s what he has, unplugs his laptop, and leaves without **`squeal`**ing.

Each of the seven TA's waits until a CS110 student gets her attention. The student shows her his assignment, she **`review`**s the code, reports the number of race conditions back to the student, **`grade`**s some **`tsh`** assignments, and then waits until another student asks for help. The very last student to leave—whether he successfully arrived at a working **`ThreadPool`** or not—wakes all of the TA's, all of whom notice there are no more students and go home. Note that each of the TA's is capable of reviewing student code and counting race conditions at the same time as other TA's, and the simulation maximizes parallelism by making sure they can do so.

Here are a collection of constants and the **`main`** function:

```
static const size_t kNumTAs = 7;
static const size_t kNumStudents = 159;
static const size_t kNumPowerOutlets = 15;
static const size_t kMaxNumRounds = 5;

int main(int argc, char *argv[]) {
  thread tas[kNumTAs];
  thread students[kNumStudents];
  for (size_t id = 0; id < kNumTAs; id++) tas[id] = thread(ta, id);
  for (thread& s: students) s = thread(student);
  for (thread& s: students) s.join();
  for (thread& t: tas) t.join();
  return 0;
};
```

Here are a few helper routines that also contribute to the simulation:

```
static size_t random(); // for student, returns random int from [0, kNumTAs – 1]
static void debug();    // for student, student debugs race conditions
static size_t review(); // for ta, review student code, return race condition count
static void grade();    // for ta, after helping student, sneak in some grading
static void squeal();   // for student, student squeals when his code works
static void submit();   // for student, student submits code before leaving
```

Assume the above helpers are already implemented, are thread-safe, and that you can just call them when you need to. You're to implement the **ta** and **student** thread routines to properly synchronize the different activities and efficiently share common resources without introducing your **own** race conditions, deadlock, or busy waiting.

a. [7 points] First, declare your global variables, ensuring that all of them are properly initialized. Because the TA's can help students independently of each other, 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 primitive types, **mutex**es, and **semaphore**s (no **condition_variable_any**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 {
   // here, list the fields needed to complete the struct definition




} tas[kNumTAs]; // declares an array of records called tas

// list any additional global variables needed to properly synchronize all threads
```

b. [18 points] 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.

Problem 2: `ThreadPool` and Office Hours [continued]

Problem 3: Read-Write Locks [15 points]

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

```
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 want some kind of mutable 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:

```
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 on the next few pages. Your answers to each of the following questions should be 50 words or less. Responses longer than 50 words will receive 0 points. You needn't write in complete sentences provided it's clear what you're saying. Understand that you don't just get all the points just because everything you say is true.

a. [2 points] 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++;
}
```

b. [2 points] 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?

c. [3 points] 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)?

d. [3 points] 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?

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?

Problem 4: Primary DNS Servers [25 points]

During the first two weeks of the course, I discussed the how DNS is used to translate domain names (e.g. **"graph.facebook.com"**, **"cs110.stanford.edu"** and **"www.wikipedia.org"**) to IP addresses (e.g. **"31.13.75.1"**, **"171.67.215.200"**, and **"198.35.26.96"**, respectively.)

The network protocol I'm inventing for this problem requires that all translation requests take the following form:

```
graph.facebook.com
cs110.stanford.edu
cs107.stanford.edu
www.wikipedia.org
famo.us
imap.gmail.com
www.google.com
math.harvard.edu
<blank line>
```

In particular, 1 or more well formed domain names are listed, one per line, to form the entire request. Each line ends in a **'\n'**, and a single blank line (consisting of nothing more than a standalone **'\n'**) marks the end of the request.

The server manages all of the translations and publishes a response back over the same connection. For each line in the request there is a corresponding line in the response, which might look like this:

```
graph.facebook.com
cs110.stanford.edu
cs107.stanford.edu
www.wikipedia.org
famo.us
www.litterati.org
imap.gmail.com
www.google.com
math.harvard.edu
<blank line>
www.wikipedia.org -> 198.35.26.96
www.litterati.org -> 54.215.147.83
graph.facebook.com -> 31.13.75.1
imap.gmail.com -> 74.125.129.109
www.google.com -> 74.125.239.148
cs110.stanford.edu -> 74.67.215.200
cs107.stanford.edu -> 74.67.215.200
math.harvard.edu -> 140.247.39.51
famo.us -> 75.101.163.44
<blank line>
```

Each line in the response details one of the address translations, and it's only required that each domain name in the request appear somewhere in the response (imposing no order in particular.) Each line of the response ends with **'\n'**, and a blank line marks the end of the entire response.

The translations are actually managed by a collection of servers in a large distributed system, but there is a single **primary** server that intercepts all requests and forwards each of them on to secondary servers that specialize in the translation a certain domain name category (e.g. there's a secondary server that knows how to translate **`.com`** domain names, and another that translates **`.edu`** domain names, and so forth). Each of these secondary servers in turn forwards their requests to third-tier servers (e.g. one that manages all **`.stanford.edu`** translations, another that manages all **`.harvard.edu`** translations, and so forth) until end servers that store a small, static map of translations are reached. These end servers post responses to the requests, and those responses—responses which conform to the same response protocol—are collated to post aggregate responses to their requesters, etcetera. All of these servers—primary, secondary, tertiary, and end servers—respect the same network protocol.

For this problem, you're going to complete the implementation of a **primary** DNS server. The primary DNS server doesn't store any of its own translations, but it does know the IP addresses of all of the secondary servers. The primary can ingest the entire request, partition them into domain name categories (e.g. maintain a **`vector`** of **`.com`** names that should be forwarded to one secondary, a second **`vector`** of **`.edu`** names that should be forwarded to another secondary, and so forth), wait for all responses from all secondaries, and then post a primary response that is the accumulation of all secondary responses.

The interface for the **`DNSServer`** class—the class that models a primary server—looks like this:

```
class DNSServer {
public:
  DNSServer();
  void runServer();

private:
  int server;
  ThreadPool inboundRequests;
  ThreadPool outboundRequests;
  struct worker {
    worker(const string& p, const string& a): pattern(p), address(a) {}
    regex pattern;
    string address;
  };
  vector<worker> workers;

  function<void(void)> buildRequestHandler(int client);
  // other methods that decompose buildRequestHandler
};
```

A server maintains a server socket (which always listens to the host's port 54321), maintains two thread pools (one that manages incoming connections to get them off the main thread as quickly as possible, and a second, larger one to maintain the collection of all forwarded requests), and a list of workers, which helps identify the IP address of the next-tier server than should handle a particular category of domain names.

The **DNSServer** constructor looks like this:

```
static const size_t kInboundTPSize = 16;
static const size_t kOutboundTPSize = 48;
static const unsigned short kDefaultPort = 54321;
static const int kBacklog = 128;

DNSServer::DNSServer():
    inboundRequests(kInboundTPSize), outboundRequests(kOutboundTPSize) {

  server = createServerSocket(kDefaultPort, kBacklog);
  workers.push_back(worker("\\.com$", "8.9.21.140")); // matches .com addresses
  workers.push_back(worker("\\.net$", "45.45.1.17")); // matches .net addresses
  // hundreds more...
}
```

The **workers** field manages a collection of regex/IP-address pairs. When a domain name matches a regex, that domain name (along with other names that match the same regex) can be forwarded to the secondary server at the corresponding IP address as part of one big distributed, divide-and-conquer architecture.

The implementation of **runServer** is straightforward, and depends on the implementation of **buildRequestHandler**, which constructs and returns a thunk that can be executed off the main thread to ingest the response, partition and forward smaller requests on to secondary servers, collate all secondary-server responses, and then post a full response to the original request. Here is the implementation of **runServer**:

```
void DNSServer::runServer() {
  while (true) {
    int client = accept(server, NULL, NULL);
    inboundRequests.schedule(buildRequestHandler(client));
  }
}
```

If you're familiar with **ThreadPool::schedule**, you know the **buildRequestHandler** method must return a thunk—e.g. a **function<void(void)>**—that knows how to participate in the protocol-compliant discussion taking place over the **client** connection.

You're to use the next several pages to complete the implementation of **buildRequestHandler**. (You'll notice that I've included two helper methods to take care of some non-networking-related details.) Your implementation must adhere the following requirements:

- the implementation must ingest the entire request (that's what the supplied **pullAllNames** method does, so you can just call it)
- the implementation must forward all domain names in the same category (e.g. all domain names that end in **.edu**) as part of a single secondary-server request. (The provided **compileMap** method takes a list of domain names and partitions them into a **map**, where the keys are secondary server IP addresses, and the values are **vector**s of domain names that should be forwarded to that IP address as part of a single request.)

- All forwarded requests must be scheduled on the second **ThreadPool** called **outboundRequests** to maximize parallelism. The thunk scheduled on this second **ThreadPool** must be constructed by another helper method. You can choose the name and the list of parameters for this helper method.
- The function constructed by **buildRequestHandler** must wait until all secondary responses have come back before returning.
- The function constructed by **buildRequestHandler** must build the full response to the original request, write it to the **client** connection, and flush the connection before returning.
- Assume **createClientSocket** is thread-safe and works even when the hostname strings are IP address constants like "**45.45.1.17**".
- You can iterate of a **map<string, vector<string>>** using the following syntax:

```
for (const pair<string, vector<string>>& entry: m) {
   const string& address = entry.first;
   const vector<string>& names = entry.second;
   ...
```

```
vector<string> DNSServer::pullAllNames(iosockstream& rss) {
  vector<string> names;
  while (true) {
    string name;
    getline(rss, name);
    if (name.empty()) break;
    names.push_back(name);
  }

  return names;
}

map<string, vector<string>> DNSServer::compileMap(const vector<string>& names) {
  map<string, vector<string>> forwardMap;
  for (const string& n: names) {
    for (const worker& s: workers) {
      if (regex_match(n, s.pattern)) {
        forwardMap[s.address].push_back(n);
        break;
      }
    }
  }

  return forwardMap;
}
```

a. [13 points] 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] { // note that we're returning a thunk! that's okay
    sockbuf rsb(client);
    iosockstream rss(&rsb);
    vector<string> names = pullAllNames(rss);
    map<string, vector<string>> workerRequests = compileForwardMap(names);
    // complete the implementation in the space below and on the next page
```

Problem 4: Primary DNS Servers [continued]

```
// more space for your implementation of buildRequestHandler
// and your thunk-constructing helper method
```

Problem 4: Primary DNS Servers [continued]

Your answers to each of the following questions should be 50 words or less. Responses longer than 50 words will receive 0 points. You needn't write in complete sentences provided it's clear what you're saying. Understand that you don't just get all the points just because everything you say is true.

b. [3 points] 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?

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

d. [3 points] 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.

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?

Problem 5: Short Answers [9 points]

Provide answers for each of these three questions:

a. [2 points] 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.

b. [4 points] 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.

c. [3 points] 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.