CS110 Final Examination Solution

This is a closed book, closed note, closed computer exam (although you are allowed to use your two double-sided cheat sheets). You have 180 minutes to complete all problems. You don't need to **`#include`** any header files, 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, call me at 415-205-2242 should you have any questions.

Good luck!

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

Last Name: Key

First Name: Answer

I accept the letter and spirit of the honor code.

[signed] _____

		Score	Grader
1. **`farm`**, Take II	[12]	_____	_____
2. Multiprocessing Redux	[10]	_____	_____
3. **`cv::wait_until`**	[6]	_____	_____
4. Timer and One-Shot Functions	[30]	_____	_____
5. Networking Redux	[12]	_____	_____
Total	**[70]**	_____	_____

Solution 1: `farm`, Take II [Median Grade: 9 out of 12]

One of the short answer questions on your midterm asked why **`farm`** could have been easily implemented without **`SIGCHLD`** handlers, whereas **`stsh`** really needed them. To drive that point home, you're going to implement a simplified version of **`farm`** all over again, but without any signal handlers.

Recall farm's architecture relied on the existence of a Python program called factor.py, which looked like this:

```
self_halting = len(sys.argv) > 1 and sys.argv[1] == '--self-halting'
pid = os.getpid()
while True:
    if self_halting: os.kill(pid, signal.SIGSTOP)
    try: num = int(raw_input())
    except EOFError: break;
    start = time.time()
    response = factorization(num) # omitted for brevity
    stop = time.time()
    print '%s [pid: %d, time: %g seconds]' % (response, pid, stop - start)
```

When invoked from the command line without the **`--self-halting`** flag, it reads all lines from standard input and publishes the factorizations of each number. Need a reminder? Here you go:

```
poohbear@myth14$ printf "12345678\n1357999\n7483921\n" | python ./factor.py
12345678 = 2 * 3 * 3 * 47 * 14593 [pid: 25309, time: 0.899008 seconds]
1357999 = 389 * 3491 [pid: 25309, time: 0.098738 seconds]
7483921 = 89 * 84089 [pid: 25309, time: 0.548174 seconds]
poohbear@myth14$
```

The option is present so that one or more processes running **`factor.py`** can be monitored and orchestrated by a program like **`farm`**. Both the Assignment 3 **`farm`** and this one work like this:

```
poohbear@myth14$ printf "12345678\n1357999\n7483921\n" | ./farm
1357999 = 389 * 3491 [pid: 25324, time: 0.161889 seconds]
7483921 = 89 * 84089 [pid: 25322, time: 0.586688 seconds]
12345678 = 2 * 3 * 3 * 47 * 14593 [pid: 25321, time: 0.90889 seconds]
poohbear@myth14$
```

The **`farm`** you're to implement here has a slightly different decomposition, but it's otherwise operationally the same as the version you coded for Assignment 3. I'll provide that decomposition, identify simplifications, and then let you loose to code.

Assume a working **subprocess struct** (with constructor) has been provided and the following **main** function has been given to you:

```
struct subprocess {
   subprocess(char *argv[], bool supply, bool ingest);
   pid_t pid;
   int supplyfd; // -1 if constructor's supply value was false
   int ingestfd; // -1 if constructor's ingest value was false
};

static const size_t kNumWorkers = 8; // always 8, regardless of CPU count
static const char *kWorkerArguments[] = {"./factor.py", "--self-halting", NULL};
int main(int argc, char *argv[]) {
   map<pid_t, int> workers;
   launchWorkers(workers);
   broadcastNumbers(workers);
   instructAllWorkersToExit(workers);
   waitForAllWorkersToExit(workers);
   return 0;
}
```

You're to implement each of the four helper function according to specification. Details will be provided for each part, but some initial simplifications are stated up front.

- The number of workers is hard-coded to be **kNumWorkers = 8**, regardless of CPU count.
- Your solution won't rely on signal handlers at all. As you'll see shortly, it's not technically necessary.
- You'll rely on the C++ **map** to associate pids of processes running **factor.py** to the supply descriptors that **farm** can write to to feed that process's standard input.

Implement the new farm solution over the course of the next several pages.

a. [3 points] Implement the **launchWorkers** function, which spawns the proper number of subprocesses running self-halting workers and populates the initially empty map with pid/descriptor pairs. Your implementation should be very short.

```
static void launchWorkers(map<pid_t, int>& workers) {
  for (size_t i = 0; i < kNumWorkers; i++) {
    subprocess_t sp =
       subprocess(const_cast<char **>(kWorkerArguments), true, false);
    workers[sp.pid] = sp.supplyfd;
  }

  assert(workers.size() == kNumWorkers);
}
```

Problem 1a Criteria: 3 points

- Properly invokes **subprocess** with the correct arguments: 2 points
- Properly inserts the correct pid/fd pairs into the map: 1 point

b. [3 points] Next implement the **broadcastNumbers** function, which forwards each line of its standard input to the next available worker. You may assume each line consists of a single number that can be forwarded as is to the worker. You should rely on **dprintf** to publish directly to the relevant file descriptor that leads to each worker once it's identified as available. Again, there are no **SIGCHLD** handlers, so you can do everything needed directly in **broadcastNumbers**. **broadcastNumbers** should return once, um, all numbers have been broadcasted. ☺

```
static void broadcastNumbers(const map<pid_t, int>& workers) {
  while (true) {
    string line;
    getline(cin, line);
    if (cin.fail()) break;
    size_t endpos;
    long long num = stoll(line, &endpos);
    if (endpos != line.size()) break;
    int status;
    pid_t pid = waitpid(-1, &status, WUNTRACED);
    assert(WIFSTOPPED(status));
    auto iter = workers.find(pid);
    assert(iter != workers.cend());
    int supplyfd = iter->second;
    dprintf(supplyfd, "%lld\n", num);
    kill(pid, SIGCONT); // this and previous line can be in either order
  }
} // you did not need to do any of this error checking
```

Problem 1b Criteria: 3 points

- Properly waits for a worker to self-halt: 1 point
- Properly writes the line (either as a C string or a number) to the relevant **supplyfd**: 1 point: 1 point
- Prompts the selected process to continue using **kill**: 1 point

c. [3 points] Present your implementation for **`instructAllWorkersToExit`**, which waits for all workers to self-halt one last time, and then instructs each of them to exit (but doesn't wait for them to actually exit).

```
static void instructAllWorkersToExit(const map<pid_t, int>& workers) {
  for (size_t i = 0; i < kNumWorkers; i++) {
    int status;
    pid_t pid = waitpid(-1, &status, WUNTRACED);
    assert(WIFSTOPPED(status));
    auto iter = workers.find(pid);
    assert(iter != workers.cend());
    close(iter->second);
  }

  for (const pair<pid_t, int>& p: workers)
     kill(p.first, SIGCONT);
} // you did not need to do any of this error checking
```

Problem 1c Criteria: 3 points

- Properly waits for all workers to self-halt: 1 point
- Properly closes the associated descriptor as worker pids surface: 1 point
- Prompts the selected process to continue using kill without confusing the previous **`waitpid`** calls (two loops are needed unless more work is done during the first loop): 1 point

d. [3 points] Finally, present your implementation of **`waitForAllWorkersToExit`**, which does precisely that: it only returns once it's confirmed that all workers have exited.

```
static void waitForAllWorkersToExit(map<pid_t, int>& workers) {
  for (size_t i = 0; i < kNumWorkers; i++) {
    int status;
    pid_t pid = waitpid(-1, &status, 0);
    assert(WIFEXITED(status));
    auto iter = workers.find(pid);
    assert(iter != workers.cend());
    workers.erase(iter);
    assert(workers.find(pid) == workers.cend());
  }
  assert(workers.empty());
} // you did not need to do any of this error checking
```

Problem 1d Criteria: 3 points

- Properly waits for all workers to exit (where the third argument to **waitpid** absolutely must be different): 3 points (1 point for calling **`waitpid`** with the correct first two arguments, 1 point for the correct third argument)

Solution 2: Multiprocessing Redux [Median Grade: 6 out of 10]

Unless otherwise noted, your answers to the following questions should be 75 words or fewer. **Responses longer than the permitted length will receive 0 points.** You needn't write in complete sentences provided it's clear what you're saying. Full credit will only be given to the best of responses. Just because everything you write is true doesn't mean you get all the points.

a. [2 points] Even though Problem 1 didn't require it, your Assignment 3 specification was very clear that the number of workers should match the number of CPUs, that each of the workers be assigned to run on just one CPU (and always the same CPU), and that all CPUs be assigned. In general, what are the advantages to ensuring that each worker always runs on the same CPU.

If a CPU is dedicated to a process, then its L1 caches are much more likely to retain previously fetched data and speed up computation, because L1 caches are almost always dedicated to a single CPU. [1 point for mentioning caching, 1 point for the appropriate amount of detail as to why.]

b. [2 points] Assume that the following function has been installed to handle all **SIGCHLD** signals for the lifetime of a program.

```
static void reapChildren(int unused) {
   while (true) {
      int status;
      pid_t pid = waitpid(-1, &status, WNOHANG);
      if (pid <= 0) break;
      assert(WIFEXITED(status) || WIFSIGNALED(status));
      printf("%d has exited.\n", pid);
   }
}
```

Note, however, that **waitpid** isn't outfitted to identify stopped and restarted processes. Does **reapChildren** get invoked as child processes are stopped and continued? Explain your answer.

It certainly gets invoked, because **SIGCHLD**s are sent for all child state processes. The way **waitpid** determines whether stopped or continued processes are identified.

c. [2 points] Recall that Assignment 1 and several discussion section exercises relied on memory mapping (via **mmap**) to share data across process boundaries. One example, drawn from a discussion section handout, is presented below:

```
static int *createSharedArray(size_t length) {
  int *numbers =
    static_cast<int *>(mmap(NULL, length * sizeof(int),
                            PROT_READ | PROT_WRITE,
                            MAP_SHARED | MAP_ANONYMOUS, -1, 0));
  RandomGenerator rgen;
  for (size_t i = 0; i < length; i++)
    numbers[i] = rgen.getNextInt(kMinValue, kMaxValue);
  return numbers;
}
```

Suppose you want to share a C++ container across process boundaries so that parent and child can agree on a single structure to store all shared data.

```
vector<int> *createSharedVector() {
   vector<int> *numbers =
      static_cast<vector<int> *>(mmap(NULL, sizeof(vector<int>),
                                      PROT_READ | PROT_WRITE,
                                      MAP_SHARED | MAP_ANONYMOUS, -1, 0));
   return number;
}
```

Concurrency issues aside, why is it difficult to share the location of a data structure like a **vector<int>** across process boundaries and expect it to work?

The vector object will certainly be stored in shared memory, but the dynamically allocated array storing vector element won't be, because it's establish using the traditional **malloc**/**operator new[]** directives. [2 points for being clear that the dynamically allocated memory is not itself memory mapped by the implementation.]

d. [2 points] **`execvp`** relies on **`mmap`** to map the process's text segment to the assembly code instructions packed within the executable. The implementation of **`execvp`** is then free to either load all assembly code instructions into main memory, or to lazily load them on an as-needed basis. Briefly describe an advantage to each approach.

- Lazy loading conserves physical memory in the cases where most of the code never runs, and allows that physical memory to be used for other processes: 1 point
- The accumulation of time spent loading everything once will be less than the sum of all lazy load times, and that's relevant when virtually all user code is executed at some point: 1 point

e. [2 points] The process scheduler relies on runnable and blocked queues to categorize processes. How exactly does this categorization lead to better CPU utilization?

By distinguishing between runnable and blocked and only allowing runnable processes to be assigned to a CPU, you dedicate more CPU time to the processes that can get work done, thereby decreasing the amount of time some blocked processes remain blocked (e.g. blocked on **`waitpid`**, waiting for **`kill`** call, etc). [2 points for emphasizing that only runnable processes can make use of the CPU.]

Solution 3: `cv.wait_until` [Median Grade: 2 out of 6]

The **`condition_variable_any`** class includes two overloaded **`wait_until`** methods, which operate much like the two **`wait`** methods do, except that each returns after a certain amount of time, even if the conditional variable hasn't been signaled.

The first version has this interface (slightly altered so it's more easily explained):

```
bool condition_variable_any::wait_until(mutex& m, uint64_t timeout);
```

If a programmer wants to wait on a condition variable for at most 5 seconds, the he or she might rely on the following (assuming **`m`** and **`cv`** are in scope, and further assuming that **`steady_clock::now()`** returns the current time in milliseconds since the 1/1/1970 epoch):

```
lock_guard<mutex> lg(m);
bool notified = cv.wait_until(m, steady_clock::now() + 5000);
```

The **`wait_until`** call is operationally identical to **`cv.wait(m)`**, except that it returns after 5000 milliseconds if it otherwise wasn't explicitly notified. The return value is **`true`** if an only if **`wait_until`** returned because it was notified, and it returns **`false`** if the five seconds has elapsed.

A second version of **`wait_until`** is more like the version of **`wait`** we've relied on for most of our work this quarter, and it has the following prototype:

```
template <typename Pred>
bool condition_variable_any::wait_until(mutex& m, uint64_t timeout, Pred pred);
```

So, code willing to wait at most five seconds for some **`private`** data member named **`count`** to become positive might use a condition variable this way:

```
lock_guard<mutex> lg(m);
bool satisfied = cv.wait_until(m, steady_clock::now() + 5000,
                               [this] { return count > 0; });
```

If this version of **`wait_until`** returns before 5000 milliseconds has passed, then it returns **`true`**, because it must be the case that **`cv`** was notified and the supplied predicate evaluated to **`true`**. If this version of **`wait_until`** returns because 5000 milliseconds passed, then it returns whatever the predicate evaluates to just prior to return (which may be **`true`** or **`false`**).

As it turns out, the second version of **`wait_until`** can be implemented in terms of the first one. Using the next page (which is way more space than you need), present your implementation of the second **`wait_until`** assuming the first one works as described. Your implementation should be no more than 5 or so lines of code, but it's important you have a sense as to how this works, because you'll be using one or both of these **`wait_until`** methods for Problem 4.

Solution 3: `cv.wait_until` [continued]

Place your implementation in the space below.

```
template <typename Pred>
bool condition_variable_any::wait_until(mutex& m, uint64_t timeout, Pred pred) {
   while (!pred()) {
      if (!wait_util(m, timeout)) {
         return pred();
      }
   }
   return true;
}
```

Problem 3 Criteria: 6 points

- 0 points for busy waiting and not relying on other forms of wait.
- Properly identifies the situation where **`wait_until`** times out, and in that case returns true if and only if **`pred()`** evaluates to **`true`**: 2 points
- Properly identifies the situation where cv is notified, and
    - returns true if and only if **`pred()`** evaluates to **`true`**: 2 points
    - call **`wait_until`** in a loop specifying the same expiration time: 2 points

Solution 4: Timer Functions [Median Grade: 18 out of 30]

Some languages—most notably, JavaScript—allow functions to be scheduled for execution at a later time. Such functions, called **timer functions**, can be scheduled as **one-shot functions** (which means they only execute once) or as **interval functions**, which are repeatedly scheduled for execution with a fixed time delay in between each call. C++ doesn't provide native support timer functions, but those armed with the skill set acquired from ten weeks of CS110 are in a position to add it.

For this problem, you're to implement the majority of a **`timer`** class, which can be used to schedule one-shot and interval functions. To illustrate, consider the following test harness (where **`oslock`** and **`osunlock`** manipulators have been removed for code clarity):

```
int main(int argc, char *argv[]) {
   timer t;
   uint64_t ids[] = {
      t.setTimeout([]() {
         cout << "One-shot functions 1 invoked!" << endl;
      }, 1000), // invoked once, one second from now
      t.setTimeout([]() {
         cout << "One-shot functions 2 invoked!" << endl;
      }, 3000), // invoked once, three seconds from now
      t.setInterval([]() {
         cout << "Timer function 1 invoked." << endl;
      }, 0, 500), // fires every 0.5 seconds, starting now
      t.setInterval([]() {
         cout << "Timer function 2 invoked." << endl;
      }, 50, 333), // fires every third of a second, starting 50ms from now
      t.setInterval([]() {
         cout << "Timer function 3 invoked." << endl;
      }, 3900, 10), // fires every 10 ms, starting 3.9 seconds from now
   };

   sleep_for(2000); // let everything fly for two seconds
   cout  << "Killing off timer function 2 (with id " << ids[3] << ")." << endl;
   t.clear(ids[3]); // kill of the second of the three timer functions
   cout  << "Timer function 2 should be dead." << endl;
   sleep_for(2000);
   return 0;
}
```

Note that five timer functions are scheduled using **`setTimeout`** (for the one-shot functions) and **`setInterval`** (for the interval functions). Each call to **`setTimeout`** and **`setInterval`** returns an identifier that can be passed to **`clear`** to deactivate that timer function from ever running again.

Note that the first one-shot function should execute at t = 1 second and the second one-shot function should execute at t = 3 seconds. The first interval function is invoked immediately, and repeatedly executes every 0.5 seconds until the four-second lifetime of the test program ends. The second interval function is delayed for 50 ms, but one it starts, is executes every third of a second until it's deactivated at t = 2 seconds. And that third interval function doesn't start until

the program has just 100 milliseconds left, but one it starts, it sneaks in 10 executions before the program ends.

When the above test program is executed, we get the following output:

```
Timer function 1 invoked.
Timer function 2 invoked.
Timer function 2 invoked.
Timer function 1 invoked.
Timer function 2 invoked.
One-shot functions 1 invoked!
Timer function 1 invoked.
Timer function 2 invoked.
Timer function 2 invoked.
Timer function 1 invoked.
Timer function 2 invoked.
Killing off timer function 2 (with id 3).
Timer function 2 should be dead.
Timer function 1 invoked.
Timer function 1 invoked.
One-shot functions 2 invoked!
Timer function 1 invoked.
Timer function 1 invoked.
Timer function 3 invoked.
Timer function 3 invoked.
Timer function 3 invoked.
Timer function 3 invoked.
Timer function 3 invoked.
Timer function 3 invoked.
Timer function 3 invoked.
Timer function 3 invoked.
Timer function 3 invoked.
Timer function 3 invoked.
```

Over the course of the next few pages, you're going to implement the **`timer`** class. By doing so, you'll be able to show off your ability to manage exposed threads, **`ThreadPool`**s, thunks, **`mutex`**es and condition variables. To simplify the problem and to be clear what we're testing, we'll provide the entire class declaration (i.e. what would be presented in the **`timer.h`** file) and then lead you through a series of problems to flesh out the **`.cc`**.

Here is the reduced **timer.h** file, where all **private** data structures and members have been decided for you:

```
class timer {
public:
   timer();
   ~timer();

   uint64_t setInterval(const std::function<void(void)>& thunk,
                        uint64_t when, uint64_t period);
   uint64_t setTimeout(const std::function<void(void)>& thunk, uint64_t when);
   void clear(uint64_t id);

private:
   struct event {
      uint64_t id;
      uint64_t when;
      uint64_t period;
      std::function<void(void)> thunk;
   };

   std::map<uint64_t, event> active;
   std::map<uint64_t, std::vector<event *>> queue;

   std::atomic<uint64_t> next;
   std::mutex m;
   std::condition_variable_any cv;
   bool working; // normally true, set to false during teardown

   std::thread dispatcher;
   ThreadPool pool;

   void dispatch();
};
```

Here's how each of the private data members contribute to the overall implementation:

- The event record bundles information about an active timer function: its unique **id**, the time **when** it should be executed, the **period** dictating how often it's executed (this stores a 0 if the timer function is one-shot), and a **thunk** itself.
- The **active** map stores all of the active timer functions, where the keys are `event` ids and the values are the full copies of the **event**s with those ids.
- The **queue** map is effectively a priority queue. The keys are future times when one or more events should be invoked, and the values are the sequence of events that should be invoked when that time arrives. And rather than store independent copies of the events again, we store pointers to the relevant events formally owned by the active map. When **queue** has one or more key/value pairs, **queue.begin()** returns an iterator to the collection of events with the smallest **when** times, because **map**s are backed by binary search trees.
- **next** is an **atomic**, just like the **atomic** we used in the **ice-cream-parlor** simulation, that provides thread-safe postfix **++** (which is the only **atomic** operation you need)

- **m** and **cv** are the only two exposed concurrency directives you can use to make the timer class thread-safe and to foster communication between the OS and the various timer methods you'll implement.
- **dispatcher** is a thread that operates much like the **ThreadPool** dispatcher does. It exists to loop for the lifetime of the timer, potentially blocking with each iteration until there's information suggesting that one or more timer functions should be executed.
- **pool** is an **ThreadPool** of size 8 where all thunks are scheduled to execute when their time comes.
- **dispatch** is the thread routine installed into the dispatcher thread.

Over the course of the next several pages, you're to implement all of the timer's public methods, save for the **setTimeout** method, which I'll implement for you. Even though the exam has you implement all of these public entries in a prescribed order, you need to ensure that all of them communicate as needed and that all threads make efficient use of the CPU without any busy waiting. That'll require you to flip back and forth between some of the parts to make sure those communication channels are in place and there's zero opportunity for busy waiting, race conditions, or deadlock.

a. [4 points] Implement the **timer** constructor, which sets the id counter to 0, marks the **timer** class as actively **working**, sets the thread pool size to be 8, and installs **dispatch** as the thread routine inside the **dispatcher** thread. When the constructor returns, both maps should be empty, and the **dispatch** method should have started executing. (You'll implement **dispatch** in a few pages.)

```
static const size_t kPoolSize = 8;
timer::timer(): next(0), working(true), pool(kPoolSize) {
   dispatcher = thread([this] { dispatch(); });
}
```

Problem 4a Criteria: 4 points

- Properly initializes **next** to be 0: 1 point
- Properly initializes **working** to be **true**: 1 point
- Properly constructs thread pool on the initialization list: 1 point
- Properly installed **dispatch** method as the thread routine for **dispatcher** using the correct syntax: 1 point

b. [8 points] Next, implement the **setInterval** method, which accepts the provided thunk, creates a new **event** on that thunk's behalf, updates both **active** and **queue** accordingly, and returns the id given to the timer function.

- Your implementation must be thread-safe and not cause concurrency issues in the other methods.
- Assume that **steady_clock::now()** returns the current time, in milliseconds, since the 1/1/70 epoch, as a **uint64_t**.
- Note that the **when** time passed to **setInterval** is really a time-from-now value (e.g. 500 ms), whereas the **when** time stored in the relevant event (and as a key in the **queue** map) is a time-from-the-epoch value (e.g. **steady_clock::now()** + 500ms).
- If the period value is 0, that's permitted, and it means that the interval function is really just a one-shot timer function. In fact, the implementation of **setTimeout(thunk, 500)** is really just a wrapper around a call to **setInterval(thunk, 500, 0)**.
- Understand that many events may end up having the same when times, which is why the **queue** map associated time points with a **vector<event *>** instead of an isolated **event ***.
- Before **setInterval** returns, the **dispatcher** thread may need to wake up if it's blocked, but should only wake up if there's a good reason. (You'll likely figure this part out only after you've worked out **dispatch**).

```
uint64_t timer::setInterval(const function<void(void)>& thunk,
                            uint64_t when, uint64_t period) {
   event inst = {
      next++, steady_clock::now() + when, period, thunk
   };
   lock_guard<mutex> lg(m);
   bool shouldSignal = queue.empty() || inst.when < queue.cbegin()->first;
   active[inst.id] = inst;
   queue[inst.when].push_back(&active[inst.id]);
   if (shouldSignal) cv.notify_one();
   return inst.id;
}
```

Problem 4b Criteria: 8 points

- Constructs an event on behalf of the supplied data: 3 points
  - Properly extracts current value of **next** and then increments it: 1 point
  - Properly computes the **when** time: 1 point
  - Properly writes in **period** and **thunk** unmodified: 1 point
- Properly acquires the lock on **m** to protect access to **queue** and **active**: 2 points
  - 1 point for acquiring it using the correct syntax (either using a **lock_guard** or exposed **lock** and **unlock** calls): 1 point
  - 1 point for acquiring it for only as long as necessary: 1 point
- Correctly inserts **inst** and **&active[inst.id]** into the **active** and **queue** maps: 1 point (with 0 points given if they store **&inst**, which is the only subtle part of these two lines)
- Correctly calls **cv.notify_one()** (or **_all**, though comment if they do): 2 points

    - 1 point for doing so because the maps were empty but isn't any more: 1 point
    - 1 point for doing so because there's a new leader in the **queue**: 1 point
- If they forget to actually return the id, then just comment, not interesting from a CS110 perspective

c. [6 points] Now implement the thread-safe **clear** method, which updates the **active** and **queue** maps to exclude all traces of the relevant timer function. If the relevant timer function is executing at the time **clear** is called, then let it execute without waiting for it to finish, but make sure it doesn't execute again. And if the supplied id isn't present, then return without doing anything.

```
void timer::clear(uint64_t id) {
   lock_guard<mutex> lg(m);
   auto iter = active.find(id);
   if (iter == active.cend()) return; // say we cleared it :)
   event& e = iter->second;
   auto& events = queue[e.when];
   for (size_t i = 0; i < events.size(); i++) {
      if (events[i] == &e) {
         events.erase(events.begin() + i);
         if (events.empty()) {
            queue.erase(e.when); // take it out of the queue
         }
         break;
      }
   }
   active.erase(iter);
   if (queue.empty()) cv.notify_one();
}
```

Problem 4c Criteria: 6 points

- Acquires the lock as they did for part b: 1 point
- Correctly searches the active map given the id using the correct syntax: 1 point
- Correctly returns if the id wasn't found in the map: 1 point
- Properly searches relevant **event *** vector, removing the event and, if necessary, removing the empty vector: 2 points
- Removing the id/event pair from **active**: 1 point (0 if removed prematurely)
- Signaling is technically optional, since the **wait_until** in **executeThoseDue** will still time out, and will be actively signaled in the destructor or through **setTimeout** if the maps are now empty. My preference for calling it is that I want the dispatcher to be blocked in the more appropriate **cv.wait** call, since it models the new situation more accurately): 0 points

d. [8 points] Implement the granddaddy of all **timer** methods: the thread-safe **private dispatch** method, which was installed as **dispatcher**'s thread routine at construction time. **dispatch** repeatedly loops, conditionally blocking with each iteration until there's good reason to proceed. If when it wakes up it notices the current time is greater than the **when** times of one or more events, it executes all those events by scheduling their thunks in the thread pool. Events for one-shot functions are then removed from **active** and **queue**, and events in place for interval functions are updated with new **when** times, and **queue** is updated to reflect the new trigger times. (Note: this method will make use of the **wait_until** methods introduced in Problem 3).

```
void timer::dispatch() {
   lock_guard<mutex> lg(m);
   while (true) {
      if (queue.empty()) {
         cv.wait(m, [this] { return !working || !queue.empty();});
      } else {
         cv.wait_until(m, queue.cbegin()->first);
      }
      if (!working) break;
      executeThoseDue();
   }
}

void timer::executeThoseDue() {
   timestamp now = steady_clock::now();
   while (!queue.empty() && now >= queue.cbegin()->first) {
      auto& events = queue.cbegin()->second;
      for (size_t i = 0; i < events.size(); i++) {
         if (events[i]->period > 0) {
            events[i]->when = now + events[i]->period;
            queue[events[i]->when].push_back(events[i]);
         } else {
            active.erase(events[i]->id);
         }
         pool.schedule(events[i]->thunk);
      }
      queue.erase(queue.cbegin());
   }
}
```

Problem 4d Criteria: 8 points

- Properly acquires the lock on m for the lifetime of the thread routine (save for the times it's released **while** in **wait** and **wait_until** calls): 1 point
- Properly blocks using a traditional **wait** when the queue is empty: 2 points
- Properly blocks using the new **wait_until** call if the **queue** isn't empty and we're waiting for a specific time point to be reached: 2 points
- Properly schedules all thunks that should be invoked: 1 point (forgive them if they don't have a **while** loop and consider a second round of times—that's difficult to know)
- Properly reschedules all events to be repeated: 1 point
- Properly removes all expired time points from **queue**: 1 point

e. [4 points] And finally, implement the **timer** destructor, which informs **dispatcher** that the surrounding object is being torn down and no other timer functions should be executed (although currently executing ones are permitted to finish). The destructor then waits for **dispatcher** to exit and the thread pool to drain before exiting.

```
timer::~timer() {
   m.lock();
   working = false;
   m.unlock();
   cv.notify_one();
   dispatcher.join();
}
```

Problem 4e Criteria: 4 points

- Properly sets working to be **false** without race condition: 1 point
- Properly signals the **cv** to work: 1 point
- Properly calls **join** on dispatcher: 2 points
    - Properly **join**s at all: 1 point
    - Properly **join**s at a point where the lock on **m** isn't being held: 1 point
- No problem if they call **wait** on the thread pool, but it's already done for them by a working **ThreadPool** destructor, so it's not needed.

Solution 5: Networking Redux [Median Grade: 6 out of 12]

Unless otherwise noted, your answers to the following questions should be 75 words or fewer. **Responses longer than the permitted length will receive 0 points.** You needn't write in complete sentences provided it's clear what you're saying. Full credit will only be given to the best of responses. Just because everything you write is true doesn't mean you get all the points.

a. [2 points] Briefly but clearly describe how your proxy could be updated to block requests from a known malicious client?

The call to **`accept`** surfaces the IP address of the requester. If that IP address matches one found in a blacklist (separate from the blacklist of forbidden origin servers), then the connection is simply closed. [1 point for mentioning blacklist, 1 point for being clear that it's not the same blacklist as the one from their assignment]

b. [2 points] When the number of mappers exceeds the number of nodes, some nodes are required to run two or more mappers. How is the map-reduce server capable of managing one dedicated conversation with each worker, even when those workers are running on the same node?

Interpretation The client socket returned by the call to **`accept`** matches exactly one call to **`createClientSocket`** on the remote host, and if there are multiple calls from the same host (or even the same application), there's still one socket pair ***`per conversation`*** (not per server/host pair). [1 point for drawing attention to accept, 1 point for speaking of dedicated socket pair per connection.]

c. [2 points] Your Assignment 8 map-reduce implementation relies on multiprocessing and multithreading to manage at most 32 remote workers. Had the specification required you to manage tens of thousands of workers (on a mythical myth cluster with millions of machines), the architecture imposed by the Assignment 8 spec would have failed. Without relying on non-blocking I/O techniques, how could you have augmented the architecture of map-reduce to control tens of thousands of workers.

You could set up a hierarchy of intermediate proxies to assist the primary server in communicating with the larger number of workers. The server communicates with, say, 150 primary proxies, each of which manages conversations with 150 second proxies, each of which manages conversations with the workers. [1 point for speaking of intermediate hosts that are server/worker hybrids, 1 point for speaking speaking in terms of numbers that respect system limits.]

d. [2 points] Your map-reduce server relies on a **`ThreadPool`** of size 32 to manage conversations with up to 32 different workers at the same time. Describe the deadlock potential that could have resulted had the **`ThreadPool`** size been anything smaller than the number of workers. Hint: Think about the case where the last input file is in flight and everything else has been successfully processed.

If one worker is busy processing the last input chunk, then 31 other **`ThreadPool`** workers might be blocked mid-conversation with 31 map-reduce workers, all waiting on a **`cv`** for the input chunk count to go positive or for input and inflight counts to go zero. That **`cv`** can't be signaled if there isn't a **`ThreadPool`** worker open to accepting a message of success or failure from that first map-reduce worker. [2 points for accurately describing scenario, 1 point for describing the scenario with confused or hand-wavy language, 0 points otherwise]

e. [2 points] Explain why your Assignment 7 proxy is a form of virtualization.

The proxy makes many resources—in this case, web servers—appear to be one, and that's one of the definitions of what virtualization is. [2 points for correct response in terms of many-appear-to-be-one language.]

f. [2 points] Explain why non-blocking I/O and event-driven programming using the **`epoll`** functions allows a server to more efficiently handle a larger number of client connections than a server relying solely on multithreading.

The number of threads per process is limited to 250 or so. A server can still accept tens of thousands of connections and scale that way, but without non-blocking and event-driving techniques, you need to fully commit to the first 250 that come in and fully handle their clients before addressing the others, even if these first 250 are time consuming. [1 point for speaking of thread count limits, and 1 point for speaking of to the fact that the first 250 clients would need to be handled in full before moving on to others.]