Winter 2017 March 22nd, 2017

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. File System Redux	[10]	_____	_____
2. Distributed Bubble Sort	[20]	_____	_____
3. Event Barriers	[15]	_____	_____
4. Thread Pool With Thunk Priorities	[30]	_____	_____
5. Closest Points	[15]	_____	_____
6. Concurrency, Networking Redux	[20]	_____	_____
Total	**[110]**	_____	_____

Solution 1: File System Redux

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. A variant of the **mkdir** system call—called **mkdirat**—is equivalent to **mkdir** unless the **pathname** argument is relative. In the case where the provided path argument is relative, the directory to be created is relative to the directory associated with the provided descriptor. Leveraging your understanding of the file descriptor tables, the file entry table, and vnode entry table, and the layers of the file system, explain how **mkdirat** is best implemented.

```
int mkdirat(int dirfd, const char *pathname, mode_t mode);
```

Drill through **dirfd** to the relevant vnode entry to extract the inode number for the base directory. Resolve the relative **pathname** as you normally would from the initial inumber, configuring new inodes and appending new **struct dirent**s to the end of directory payloads as needed until all new components are part of the file system. [0, 1, or 2 points on sliding scale]

b. According to its man page, the **scandir** function scans the directory **dirp**, calling **filter** on each directory entry. Entries for which **filter** returns nonzero are stored in C strings allocated using **malloc**, sorted using **qsort** with the supplied **compar** comparator, and placed in a **NULL**-terminated **namelist** array, which is allocated using **malloc**.

```
int scandir(const char *dirp, struct dirent ***namelist,
            int (*filter)(const struct dirent *),
            int (*compar)(const struct dirent **, const struct dirent **));
```

If you know how **mkdir** and **mkdirat** crawl over and manipulate the filesystem, you fundamentally know how **scandir** must be implemented. However, **mkdir** and **mkdirat** are system calls, whereas **scandir** is not. Why are **mkdir** and **mkdirat** system calls, whereas **scandir** is just a regular library function?

The majority of what **mkdir** and **mkdirat** do requires kernel mode (permission checks, file system changes, etc.). The majority of what **scandir** does can be done without special permissions (malloc'ing, callback functions, etc.), and only file system accesses can be managed in terms of existing system calls. [0, 1, or 2 points on sliding scale]

c. During the first two weeks of the quarter, we introduced two design principles—naming and layering—to describe the architecture and implementation of file systems. Briefly explain how naming and layering come up during our discussion of networking.

- naming: DNS lookup algorithm as described in lecture to resolve domain names to IP addresses [1 point for this example or one equally strong]
- layering: TCP/IP model where application layer (socket, bind, connect), is implemented above transport layer (introduces port numbers), which is implemented above internet layer (introduces IP addresss), and is implemented above link layer (which provided support for Bluetooth, wifi, Ethernet, etc.) [1 point for just mentioning TCP/IP or something else that convinces you they know of these layers]

d. [2 points] To speed up the implementation of assign1's **`file_getblock`** and **`inode_ilookup`**, you update the implementation of **`diskimg_readsector`** to include an in-memory cache of the 1024 most recently accesses sectors. The installation of sector cache decreases the average running time of **`file_getblock`** by 16% for small files, but decreases the average running time of **`file_getblock`** by 45% for large files. Explain why the cache is so much more helpful for large files.

For large files, **`file_getblock`** spends majority of its time drilling through same sectors housing singly and doubly indirect block numbers. If those sectors are hot and present in the cache, execution time improves dramatically. [0, 1, or 2 points on sliding scale]

e. When we introduced the pipe as an extension to the file descriptor abstraction, we implemented **`subprocess`** as follows:

```
subprocess_t subprocess(const char *command) {
  int fds[2];
  pipe(fds);
  subprocess_t process = { fork(), fds[1] };
  if (process.pid == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]); close(fds[1]);
    char *argv[] = {"/bin/sh", "-c", (char *) command, NULL};
    execvp(argv[0], argv);
  }
  close(fds[0]);
  return process;
}
```

Now consider the following very short program that trivially relies on **`subprocess`**:

```
int main(int argc, char *argv[]) {
  subprocess_t sps[3];
  for (size_t i = 0; i < 3; i++) sps[i] = subprocess("sort");
  for (size_t i = 0; i < 3; i++) dprintf(sps[i].supplyfd, "b\nc\ne\na\nd\n");
  for (size_t i = 0; i < 3; i++) close(sps[i].supplyfd);
  for (size_t i = 0; i < 3; i++) waitpid(sps[i].pid, NULL, 0);
  return 0;
}
```

When run, the program prints out 15 lines as expected. But **`valgrind`** complains that **`sps[1].pid`** and **`sps[2].pid`** leave a combined three extra file descriptors open. Why are those processes leaking descriptors, and how can the implementation of **`subprocess`** be updated to fix it?

Second call to **`subprocess`** leads to **`fork`** call that duplicates **`sps[0].supplyfd`**, and **`subprocess`** doesn't know of it, doesn't know to close it. Third call to **`subprocess`** duplicates **`sps[0].supplyfd`**, **`sps[1].supplyfd`**. [1 point]

The fix? Many solutions, but all require **`subprocess`** track or be reminded of descriptors it opened that haven't been closed, so it can close them in the forked child process. [1 point, be open to many legitimate ideas]

Solution 2: Distributed Bubble Sort

Consider the following program on a 128-core machine, which launches 127 peer processes to cooperatively sort an array of 128 randomly generated numbers. (The implementations of **`createSharedArray`**, **`printArray`**, **`freeSharedArray`**, and **`swap`** are omitted for brevity.)

```
static void foreverSwap(int& first, int& second) {
  while (true) {
    raise(SIGSTOP);
    if (first > second) swap(first, second);
  }
}

static void launchAllSwappers(int numbers[], pid_t pids[], size_t count) {
  for (size_t i = 0; i < count; i++) {
    pids[i] = fork();
    if (pids[i] == 0) foreverSwap(numbers[i], numbers[i + 1]);
  }
}

static void startAndWait(pid_t pids[], size_t start, size_t stop) {
  for (size_t i = start; i < stop; i += 2) kill(pids[i], SIGCONT);
  for (size_t i = start; i < stop; i += 2) waitpid(pids[i], NULL, WUNTRACED);
}

static void manipulateSwappers(pid_t pids[], size_t count) {
  for (size_t pass = 0; pass <= count/2; pass++) {
    startAndWait(pids, 0, count);
    startAndWait(pids, 1, count);
  }

  for (size_t i = 0; i < count; i++) kill(pids[i], SIGKILL);
  for (size_t i = 0; i < count; i++) waitpid(pids[i], NULL, 0);
}

static void bubblesort(int numbers[], size_t length) {
  size_t numProcesses = length - 1;
  pid_t pids[numProcesses];
  launchAllSwappers(numbers, pids, numProcesses);
  manipulateSwappers(pids, numProcesses);
}

const size_t kNumElements = 128;
int main(int argc, char *argv[]) {
  for (size_t test = 0; test < 10000; test++) {
    int *numbers = createSharedArray(kNumElements);
    printArray("Before: ", numbers, kNumElements);
    bubblesort(numbers, kNumElements);
    printArray("After:  ", numbers, kNumElements);
    assert(is_sorted(numbers, numbers + kNumElements));
    freeSharedArray(numbers, kNumElements);
  }
  return 0;
}
```

The system presented on the previous page is a nod to distributed programming and whether parallelism can reduce the asymptotic running time of an algorithm (in this case, a variation on bubble sort). The above main function exercises the parallel implementation of **`bubblesort`** 10000 times. You may assume the implementation of **`bubblesort`** is correct and that the program never fails its one assert statement.

This system works because the address in the **`numbers`** variable is cloned across the 127 **`fork`** calls, and this particular address **maps to the same set of physical addresses in all 128 processes**. The implementation of **`createSharedArray`** relies on something called *memory mapping*, and allows segments of memory to be shared across multiple processes.

We'll lead you through a series of short questions—some easy, some less easy—to test your multiprocessing sensibilities and to understand why the asymptotic running time of an algorithm can often be improved in a parallel programming world.

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. Examine the implementation of **`foreverSwap`** on the previous page, and notice that each one conditionally reorders the integers referenced by **`first`** and **`second`**. Explain why the **`raise(SIGSTOP)`** call is needed.

   Each one relies on signal handling to inform the manipulator that it's waiting for permission to continue swap, since only a subset of swappers can be swapping at any one time. [0, 1, or 2 points on sliding scale]

b. The main thread of execution instructs all even-indexed swappers to continue, conditionally swap, and then self-halt before allowing all odd-indexed swappers to do the same thing. Why not let all of them operate at the same time?

   Even-indexed swappers never manipulate each other's numbers, so it's safe to let them swap simultaneously. Permit neighboring swappers to examine and swap simultaneously and you have race conditions. [0, 1, or 2 points on sliding scale]

c. Explain why the **bubblesort** implementation is correct by describing where the smallest and the largest elements in **numbers** can be after each call to **startAndWait**.

Each call to **startAndWait** (except for the first one) always moves the smallest number to the left one slot and the largest number to the right. After **length** calls, the smallest must be at index 0 and the largest must be at index **length – 1**. [0, 1, or 2 points on sliding scale]

d. Consider a change to the **startAndWait** implementation where the two **for** loops are merged into one, as with this:

```
static void startAndWait(pid_t pids[], size_t start, size_t stop) {
  for (size_t i = start; i < stop; i += 2) {
    kill(pids[i], SIGCONT);
    waitpid(pids[i], NULL, WUNTRACED);
  }
}
```

How does this impact the correctness and performance of **startAndWait**?

Doesn't impact correctness from a functionality standpoint, but it serializes all swapping and removes virtually all parallelism. [0, 1, or 2 points on sliding scale, be open to the claim that the serialization is a correctness issue, though]

e. Consider another change to the original **startAndWait** implementation where the first argument to **waitpid** is -1 instead of **pids[i]**. Does this introduce a bug? Explain.

Provided the only child processes are the ones created by **launchAllSwappers**, this doesn't introduce a bug. The **for** loop is question is charged with confirming all swappers have self-halted, but it doesn't need to confirm **pids[0]** has halted before **pids[2]**, etc. [0, 1, or 2 points on sliding scale]

f. The program presented doesn't rely on any custom **SIGCHLD** handlers, but instead relies on the default handler, which consumes but otherwise ignores **SIGCHLD**s. Some distributed programs (like the one above) don't really need **SIGCHLD** handlers, whereas others (like **assign3**'s **stsh**) benefit by relying on them. Why do **SIGCHLD** handlers simplify the implementation of something like **stsh** when they're evidently unnecessary for our bubble sort implementation?

bubblesort has nothing else to do but wait for swappers to self-halt, so calling **waitpid** inline makes sense. **stsh**, however, supports background processes, and therefore has other work to move on to and can't be held up by a **waitpid** call. [0, 1, or 2 points on sliding scale]

g. In spite of the fact that we never install a **SIGCHLD** handler, the operating system still sends **SIGCHLD** signals to the parent process; they're just ignored. Identify the three lines of code in the above program that lead to **SIGCHLD** signals being sent to the parent process.

raise(SIGSTOP): 1 point
kill(pids[i], SIGKILL) and **kill(pids[i], SIGCONT)**: 1 point

h. [2 points] Consider the last of the three **for** loops in **manipulateSwappers** and its call to **waitpid(pids[i], NULL, 0)**. Consider what might happen if the implementer accidentally passed in **WNOHANG** instead of 0 as the third argument. The implementer could, in principle, see **fork** failures after a few iterations of **main**'s **for** loop. Why is that?

waitpid(pids[i], NULL, WNOHANG) is nonblocking, and might return before a swapper has transitioned into zombie status. Zombie processes are still processes, and count toward the maximum number of processes allowed (around 256). That means calls to **fork** would soon fail. [0, 1, or 2 points on sliding scale]

i. Explain why the running time our distributed bubble sort is still $O(n^2)$, in spite of the parallelism built in to the swappers.

The orchestrator makes n passes over swapper pid array of length n. That's $O(n^2)$ time.

j. Explain how you might use memory mapping (the same technique used by **createSharedArray**), files, or both to update the implementation of bubblesort to run in O(n) instead of $O(n^2)$ time.

Allocate additional shared memory to make space for two **cv**s: the parent waits a constant amount of time for the swappers to conditionally swap and be notified they're all done, and each swapper waits on another condition variable the orchestrator **notify_all**s when it's the odd- or even-indexed swappers' turn to proceed. You also need space for **size_t**s, **bool**s, and **mutex**es to support the **cv**'s conditions.

Solution 3: 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 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.

Over the next three pages, detail your internal representation by filling in the **`private`** section of the **`EventBarrier`** class declaration, and then implementing the constructor and the three zero-argument methods.

a. Fill in the private section of the EventBarrier declaration below. You should only rely on **`bool`**s, **`size_t`**s, **`mutex`**es, and/or **`condition_variable_any`**s. You should have at most 6 data members (and no arrays, vectors, or containers should be needed). This part is 0 points, because it's here more or less to inform how you implement the **`EventBarrier`** in parts b through e.

```
class EventBarrier {
public:
  EventBarrier();
  void wait();
  void lift();
  void past();

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

b. Implement the **`EventBarrier`** constructor in the space below. Your implementation should be very short.

```
EventBarrier::EventBarrier() : numWaiting(0), occurring(false) {}
```

Problem 3b Criteria: 2 points

- Properly initializes **`numWaiting`** to 0: 1 point
- Properly initializes other state variables to be clear no event is in progress: 1 point

c. [3 points] Now implement **`EventBarrier::wait`** in the space below.

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

Problem 3c Criteria: 3 points

- Properly acquires lock on **`m`** for lifetime of method call: 1 point
- Increments **`numWaiting`** unconditionally: 1 point
- Efficiently (i.e. without busy waiting) waits for the event to be fired: 1 point (allow for possibility student relies on a separate **`mutex`** to guard occurring, and let this influence how you grade the problem).

d. Next, implement the **`lift`** method according. In particular, **`lift`** should inform all waiting consumers of the event, and then wait for each and every one of them to invoke the **`past`** method.

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

Problem 3d Criteria: 5 points

- Properly acquires lock on **`m`** for lifetime of method call: 1 point (if some other **`mutex`** guards **`occurring`**, they may need to acquire/release, acquire, and release so they don't hold the lock during the **`wait`** call).
- Sets **`occurring`** to be **`true`**: 1 point
- Notifies all consumers waiting on the event that the event has occurred: 1 point (give 0 points for **`notify_one`**)
- Properly blocks on the number of pre-past consumers to be 0: 1point (give zero points for busy waiting solution)
- Sets **`occurring`** to be **`false`** (thereby setting the event barrier as it was immediately after construction): 1 point

e. Finally, implement the **past** method, which is called by each consumer to inform the producer and the other consumers that they've detected the event and that it's safe to restore the barrier.

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

Problem 3e Criteria: 5 points

- Properly acquires lock on **m** for lifetime of method call (or does what's required from a **mutex** standpoint to guarantee thread safety): 1 point
- Decrements **numWaiting** (or otherwise notes one more consumer is past the barrier): 1 point
- Distinguishes between last consumer to get past the barrier and all other consumers: 1 point
- Properly requires that all by the last consumer efficiently block until the last consume is past: 1 point (0 points for busy waiting)
- Properly instructs last consumer to notify producer and all other consumers that everyone is past the barrier: 1 point (0 points for a busy waiting solution)

Solution 4: Thread Pool With Thunk Priorities

For this problem, you'll revisit the **`ThreadPool`** abstraction from Assignment 4, which is as follows:

```
class ThreadPool {
public:
  ThreadPool();
  void schedule(const function<void(void)>& thunk, size_t priority = 0);
  void wait();
  ~ThreadPool();
private:
```

The abstraction is precisely the same, except that when thunks are scheduled for execution, they are scheduled with an integer priority between 0 and 127 inclusive. Higher priority thunks are selected for execution before lower priority thunks, regardless scheduling order. And to guard against high-priority thunk starvation (i.e. the scenario where all workers are occupied with low-priority and time consuming thunks), we impose the following design restrictions:

- there are always 128 workers—the **`ThreadPool`** is constructed that way from the beginning
- worker 0 may be selected to execute thunks of any priority
- worker 1 may be selected to execute thunks of priority 1 or higher
- worker 2 may be selected to execute thunks of priority 2 or higher
- worker 3 may be selected to execute thunks of priority 3 or higher, etc.

In general, the k^{th} worker will never be selected to execute a thunk with priority less than k. That means all workers are willing to execute higher tasks (so there's more parallelism for these high-priority tasks—makes sense!), but very few workers are willing to execute low-priority tasks (also makes sense, since they're low priority, right?).

You will extend a partial design and present an entire implementation over the course of the next several pages.

a. First, complete the class declaration in the space below. You'll probably want to come back to this part from time to time to add new data members and helper methods as you provide implementations for the four public entries. Understand that this part is 0 points, except that errors here will translate to errors in later parts of the problem. Hint: we strongly advise that you add an array of 128 **list<function<void(void)>>**s.

```
class ThreadPool {
 public:
  static const size_t kMinPriority = 0;
  static const size_t kMaxPriority = 127;
  ThreadPool();
  void schedule(const function<void(void)>& thunk, size_t priority);
  void wait();
  ~ThreadPool();

 private:
  static const size_t kNumWorkers = kMaxPriority + 1;
  void dispatcher();
  void worker(size_t workerID);
  thread dt;
  struct {
    bool isAvailable = true;
    function<void(void)> thunk;
    thread wt;

    // add other data members as needed
    semaphore ready;

  } workers[kNumWorkers];

  // add other data members and helper methods as needed
  bool isRunning = true;

  std::mutex thunksLock;
  std::deque<std::function<void(void)>> thunks[kNumWorkers];

  semaphore numWaiting;
  semaphore numAvailable;
  std::condition_variable_any qualifiedWorkerAvailable;
  size_t getAvailableWorker(size_t priority);
  size_t computeHighestPriority() const;

  size_t outstanding = 0;
  std::mutex outstandingLock;
  std::condition_variable_any done;

};
```

All of my additional data members and functions are highlighted in bold. I also changed the name of the **thread** field in the worker **struct** to be **wt**, and I added a few constants to replace some magic numbers.

b. First, implement your constructor so that all 129 threads are running, and all data members that need to be initialized are initialized. We won't be fussy about C++ syntax as long as your intent is clear. Your implementation should be very short.

   Don't worry about the implementation of **`dispatcher`** and **`worker`** methods just yet, even though you need to call them. You'll implement them in later parts.

```
ThreadPool::ThreadPool(): numAvailable(kNumWorkers) {
  dt = thread([this] { dispatcher(); });
  for (size_t workerID = 0; workerID < kNumWorkers; workerID++) {
    workers[workerID].wt = thread([this](size_t workerID) {
      worker(workerID);
    }, workerID);
  }
}
```

Problem 4b Criteria: 5 points

Note: Ignore the **`numThreads`** parameter included in the prototype on the exam, or assume **`numThreads`** is 128.

- Initializes the dispatcher thread (be super lenient on syntax): 1 point
- Initializes all of the worker threads (again, be super lenient on syntax (and note they'll all be available automatically, since **`bool available = true`** was inlined in the **`.h`** files): 2 points
- Ensures that the **`numAvailable`** semaphore (or whatever they use to communicate to the dispatcher how many workers are idle) to be the maximum value of 128/**`kNumWorkers`**: 2 points (allow for each worker thread routine to signal the semaphore up front instead, so that the constructor can construct **`numAvailable`** to sit around a 0).

c. Implement the **`dispatcher`** method, which blocks until there's at least one qualified worker available to execute the highest priority task, finds qualified worker, and then prompts that worker to execute it. Your implementation here must be sensitive to the fact that a low-priority task is of no interest to workers ordered to execute only high-priority tasks. Hint: implementing a **`getAvailableWorker(size_t priority)`** method will be very helpful.

```
void ThreadPool::dispatcher() {
  while (true) {
    numWaiting.wait();
    if (!isRunning) break;
    numAvailable.wait();
    lock_guard<mutex> lg(thunksLock);
    size_t workerID, priority;
    qualifiedWorkerAvailable.wait(thunksLock, [this, &priority, &workerID] {
      priority = computeHighestPriority();
      workerID = getAvailableWorker(priority);
      return workerID < kNumWorkers;
    });
    workers[workerID].thunk = thunks[priority].front();
    thunks[priority].pop_front();
    workers[workerID].ready.signal();
  }
}

size_t ThreadPool::getAvailableWorker(size_t priority) {
  for (size_t workerID = 0; workerID <= priority; workerID++) {
    if (workers[workerID].available) {
      workers[workerID].available = false;
      return workerID;
    }
  }
  return kNumWorkers; // sentinel that no qualified worker is available
} // precondition: thunksLock is locked before call

size_t ThreadPool::computeHighestPriority() const {
  for (ssize_t priority = kNumWorkers - 1; priority >= 0; priority--) {
    if (!thunks[priority].empty()) return priority;
  }
  assert(false); // should not happen
} // precondition: thunksLock is locked before call
```

Problem 4c Criteria: 5 points

- Waits until there's a thunk to be executed and some worker, perhaps unqualified, to operate on thunks: 1 point
- Blocks until some worker notifies the dispatcher that it's qualified to take on the highest priority thunk: 3 points
    - Knows to call **`condition_variable_any::wait`**: 1 point
    - Knows that highest priority and need to be identified during the execution of the condition: 1 point
    - Properly implements the condition predicate: 1 point
- Pulls the leading thunk from the highest-priority nonempty queue and assigns it to the qualified worker: 1 point

d. Implement the **`worker`** thread routine, which waits until the **`dispatcher`** signals it to execute a task, executes that task, and then notifies any interested threads when it's fully executed.

```
void ThreadPool::worker(size_t workerID) {
  while (true) {
    workers[workerID].ready.wait();
    if (!isRunning) break;
    workers[workerID].thunk();
    thunksLock.lock();
    workers[workerID].available = true;
    thunksLock.unlock();
    numAvailable.signal();
    qualifiedWorkerAvailable.notify_one(); // see note
    lock_guard<mutex> lg(outstandingLock);
    outstanding--;
    if (outstanding == 0) done.notify_all();
  }
}
// note: we could be more discerning about when to notify, but only
// require they unconditionally notify the dispatcher that a more qualified
// worker may have just freed up
```

Problem 4d Criteria: 5 points

- Waits to hear that a thunk has been placed and should be executed: 1 point
- Executes the thunk: 0 points
- Marks itself as available once it's done, and then informs the **`dispatcher`** that one more worker is available and it may even be qualified to execute the highest priority one: 2 points
- Does whatever work is necessary to inform the thread blocked in **`wait`** that the entire pool has been drained: 2 points (one for thread safety, one for the decrement)
- Includes some sensible directive to exit: 0 points (points allocated in destructor)

e. Implement the **`schedule`** method, which accepts the task with the stated priority and schedules it to be executed, taking care to ensure that the supplied task is executed before any other yet-to-be-executed tasks with lower priorities.

```
void ThreadPool::schedule(const function<void(void)>& thunk, size_t priority) {
  assert(priority < kNumWorkers);
  outstandingLock.lock();
  outstanding++;
  outstandingLock.unlock();
  thunksLock.lock();
  thunks[priority].push_back(thunk);
  thunksLock.unlock();
  numWaiting.signal(); // inform dispatcher one more thunk to be executed
  qualifiedWorkerAvailable.notify_one();
  // thunk may be highest priority one on deck, and previously unwilling
  // workers might be willing to take this one on
}
```

Problem 4e Criteria: 5 points

- Does whatever is necessary to be clear that wait has to wait even longer before it can exit: 1 point
- Appends the incoming thunk to the back of the correct queue: 1 point
- Manages everything in a thread-safe manner: 2 points
- Signals the dispatcher that one more thunk is available: 1 point
- Bonus point for knowing to call **`notify_one`**

f. Implement the **`wait`** method, which waits until all workers are idle and no thunks are outstanding. Your implementation should be very short.

```
void ThreadPool::wait() {
  lock_guard<mutex> lg(outstandingLock);
  done.wait(outstandingLock, [this]{ return outstanding == 0; });
}
```

Problem 4f Criteria: 5 points

- Acquires the lock on the needed for the condition variable: 2 points
- Efficiently blocks until pool is fully drained: 3 points (ensure the supplied mutex protects absolutely everything in the condition being evaluated).

g. Finally, implement the **`ThreadPool`** destructor, which waits until all workers are idle and all task queues to be empty, and then tears down all threads and releases any resources acquired during the thread pool's lifetime.

```
ThreadPool::~ThreadPool() {
  wait();
  isRunning = false;

  numWaiting.signal();
  for (size_t i = 0; i < kNumWorkers; i++) {
    workers[i].ready.signal();
  }

  dt.join();
  for (size_t i = 0; i < kNumWorkers; i++) {
    workers[i].wt.join();
  }
}
```

Problem 4g Criteria: 5 points

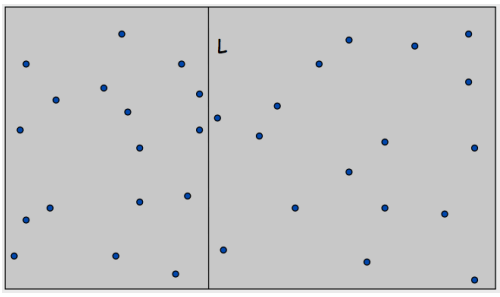
- waits for the thread pool to drain: 1 point
- marks the pool to be torn down, and fools the dispatcher and all workers into thinking there are thunks to be managed: 2 points (take off a point if the **`isRunning`** checks are missing from **`dispatcher`**, **`worker`**, or both)
- blocks until all helpers threads to exit: 2 points

Solution 5: Closest Points

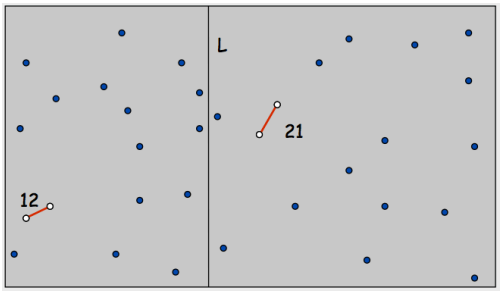
Computational geometry is a branch within computer science concerned with the data structures and algorithms easily stated in geometric terms, and has applications in data mining, computer vision, molecular modeling, air traffic control, and so forth. The closest pair problem is a common one: given a collection of points in a plane, find the two points with the smallest Euclidean distance between them.

The algorithm for doing so is a classic divide-and-conquer one:

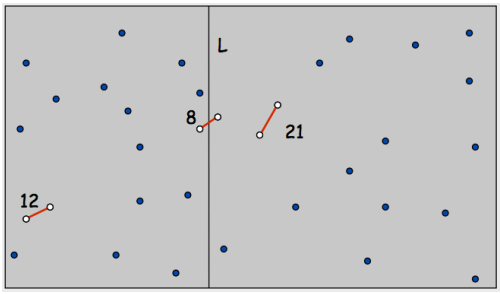
- Sort the collection of points by increasing x, and separate the sorted set into two collections of roughly the same size such that each point in the first is to the left of each and every point in the second. Pictorially, draw a vertical line L so that each side encloses roughly half the points, as with this:



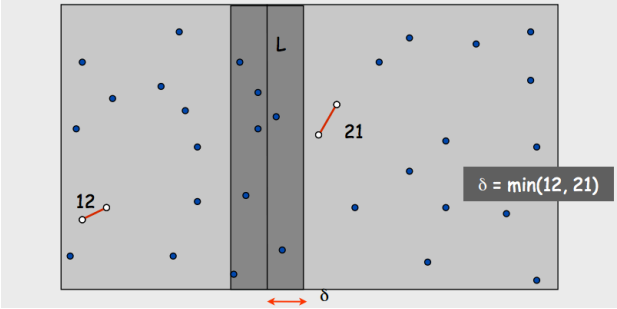
- Recursively find the closest pair in each half:



- Find the closest pair with one endpoint on each side, and then return the closest of the three:



The task of finding the closest pair with an endpoint on either side can be done very quickly, because you only need consider points within a distance δ of the original separator (i.e. the line drawn as L in the first illustration on the previous page), where δ is the smaller of the two recursively discovered distances on either side of L.



To find the two closest points with endpoints on either side, you can:

- filter out those points outside the highlighted band of width 2δ,
- sort all points by y coordinate
- find, by brute force, the two points within the band that are closest.

The last of these three bullet points sounds like an $O(n^2)$ algorithm, but it can be done in O(n) time is you accept the following true statement, stated without proof:

> If p_0 is the point with the smallest y coordinate within 2δ–band, p_1 is the point with the second smallest y coordinate within the 2δ–band, p_2 is the point with the third smallest y coordinate within the 2δ–band, and so forth, then the distance between p_k and p_{k+12} is greater than δ for any value of k. Restated, if all of the points in the 2δ–band are sorted by y-coordinate and collected in a vector, each point need only be compared to the 12 points after following it.

For this problem, you're going to rely on your **`ThreadPool`** from Question 4 to implement a multithreaded implementation of **`findClosestPoints`**, which returns a **`pair<point, point>`** containing the two closest points, using the algorithm outlined above. You should make use of the generic containers and algorithms detailed on page 2, and the custom **`point`** definition and helper routines spelled out on the very last page of the exam.

One of the algorithms on page 2, **`remove_copy_if`**, is best illustrated by example:

```
static void removeEvens(std::vector<int>& numbers) {
  vector<int> odds;
  remove_copy_if(numbers.begin(), numbers.end(),
                 back_inserter(odds), [](int n) { return n % 2 == 0;});
  numbers = odds;
}
```

You'll rely on to **`remove_copy_if`** to remove points whose x coordinates are out of range.

Use the rest of this page to implement a multithread version of **`findClosestPoints`**, which accepts an unsorted array of points, and returns the pair of points that are the closest. Your implementation should make use of the priority-oriented **`ThreadPool`** interface from Problem 4 so that each of the two recursive calls can be executed in parallel, without drama.

```
pair<point, point> findClosestPair(vector<point> points) {
  ThreadPool pool;
  return findClosestPair(points, pool);
}

static pair<point, point>
findClosestPair(vector<point> points, ThreadPool& pool, size_t priority = 0) {
  if (points.size() < 4) return closestOfThree(points);
  sort(points.begin(), points.end(), xcompare);

  auto begin = points.cbegin();
  auto mid = begin + points.size()/2;
  auto end = points.cend();

  semaphore numFinished(-1);
  vector<point> first(begin, mid);
  pair<point, point> nearestOfFirst;
  pool.schedule([&first, &pool, &nearestOfFirst, &numFinished, priority] {
      nearestOfFirst = findClosestPair(first, pool, priority + 1);
      numFinished.signal();
    }, priority);
  vector<point> second(mid, end);
  pair<point, point> nearestOfSecond;
  pool.schedule([&second, &pool, &nearestOfSecond, &numFinished, priority] {
      nearestOfSecond = findClosestPair(second, pool, priority + 1);
      numFinished.signal();
    }, priority);
  numFinished.wait();

  vector<point> band;
  double xseparator = mid->x;
  pair<point, point> nearest = nearestOfFirst;
  if (distance(nearestOfSecond) < distance(nearestOfFirst))
    nearest = nearestOfSecond;

  double minSoFar = distance(nearest);
  remove_copy_if(begin, end, back_inserter(band),
                 [xseparator, minSoFar](const point& p) {
      return abs(p.x - xseparator) > minSoFar;
  });

  if (band.size() <= 1) return nearest; // no deduction if missing
  sort(band.begin(), band.end(), ycompare);
  pair<point, point> nearestInBand = closestInBand(band);
  if (distance(nearestInBand) < minSoFar) nearest = nearestInBand;
  return nearest;
}
```

Problem 5 Criteria: 15 points

Reframes function to be implemented as a wrapper around the recursive formulation: 2 points

- Introduces a single thunk-prioritized thread pool to be shared throughout the entire recursive call: 1 point (0 points here if they go with a standard thread pool and it leads to deadlock—which is will)
- Passes the thread pool by reference and a base priority of 0 (either explicitly, or via default value as I have) to the wrapper: 1 point

Base Case: 1 point

Recursive Structure: Divide and Conquer: 9 points

- Properly sorts the points by x coordinate (be lenient on syntax that isn't CS110-specific): 1 point
- Creates two sub-vectors, **`first`** and **`second`**, as I have (or passes iterator endpoints along to achieve the same effect): 1 point
- Properly schedules two recursive calls: 3 points
    - Proper recursive call, minus syntax issues: 1 point
    - Reasonably close on the capture clauses: 1 point
    - Increases priority: 1 point (0 points if they use Assignment 4 thread pool and deadlock because of it)
- Properly waits for each of those two higher-priority recursive calls to complete: 3 points
    - If they introduce multiple thread pools—one per recursive call—then 0 points as the problem-wide penalty here
    - If they have one thread pool, and call wait: only 1 point out of 3 (they've seen this on at least two practice finals)
    - If they introduce a semaphore and properly wait for it to be signaled twice by the two recursively generated threads: all 3 points
- Properly collect the two closest points discovered by each of the two recursive calls: 1 point (0 points if they completely invent a new syntax, but 1 point is fine if the capture clause is just somewhat off)

Recursive Structure: Combine and Solve: 3 points

- Properly creates a band of coordinates that might contribute to the third nearest pair to be considered: 1 point (be lenient on syntax)
- Properly sorts on either x coordinate (correct) or y coordinate (incorrect, but there was a typo in the exam): 1 point
- Properly invoked findClosestInBand and returns closest of three pairs: 1 point

Solution 6: Concurrency and Networking Redux

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. Explain as clearly as possible why the C++ statement **`i++`** is not thread safe on all architectures.

Many architectures realize **`i++`** via three instructions: a load into a register, an increment on the register value, and a save out to memory. If one thread is swapped out between either pair of instructions and a second thread fully executes **`i++`**, the first thread doesn't know to back up and reload **`i`** and increment the new value instead of the old one. [2 point for clear explanation, 1 point for very good explanation with a minor flaw or omission, 0 points for incorrect or overly vague response]

b. Some students suggested that your Assignment 5 proxy implementation open one persistent connection to the secondary proxy when a secondary proxy is used, and that all requests be forwarded over that one connection. Explain why this would interfere with the second proxy's ability to handle the primary proxy's forwarded requests.

If the proxy has one forward connection, then the secondary has just one incoming connection from the primary. The secondary would essentially need to handle all forwarded requests sequentially over that one connection, and that would impact performance. [0, 1, or 2 points on sliding scale]

c. **Long polling** is a technique where new data may be pushed from server to client by relying on a long-standing HTTP request with a protracted, never-quite-finished HTTP response. Long polling can be used to provide live updates as they happen (e.g. push notifications to your smart phone) or stream large files (e.g. videos via Netflix, youtube, vimeo). Unfortunately, the long polling concept doesn't play well with your **`proxy`** implementation. Why is that?

Long poll connections congest the **`ThreadPool`** and prevent other requests from being handled. [0, 1, or 2 points on sliding scale]

d. HTTP pipelining is a technique where a client issues multiple HTTP requests over a single persistent connection instead of just one. The server can process the requests in parallel and issue its responses in the same order the requests were received. Relying on your understanding of HTTP, briefly explain why **`GET`** and **`HEAD`** requests can be safely pipelined but **`POST`** requests can't be.

HTTP requires that **`GET`** and **`HEAD`** requests be read-only, whereas **`POST`** requests can change server-side state, introducing a client-server race condition. [2 points for stating that **`POST`** requests change server-side state and introduce race conditions, 1 point for only mentioning change in server-side state, 0 points otherwise]

For parts e through h, consider a collection of routines from my own Assignment 6 solution:

```
bool MapReduceServer::surfaceNextFilePattern(string& pattern) {
  lock_guard<mutex> lg(m);
  cv.wait(m, [this]{ return !unprocessed.empty() || inflight.empty(); });
  if (unprocessed.empty()) return false;
  pattern = unprocessed.front();
  unprocessed.pop_front();
  inflight.insert(pattern);
  return true;
}

void MapReduceServer::markFilePatternAsProcessed(const string& pattern) {
  lock_guard<mutex> lg(m);
  inflight.erase(pattern);
  if (unprocessed.empty() && inflight.empty()) cv.notify_all();
}

void MapReduceServer::rescheduleFilePattern(const string& pattern) {
  lock_guard<mutex> lg(m);
  inflight.erase(pattern);
  unprocessed.push_back(pattern);
  if (unprocessed.size() == 1) cv.notify_all();
}
```

Exactly one of the three methods is called every time a mapper requests an input file name, a mapper announces that it successfully processes a file, or confesses it was unable to properly process the file.

e. [2 points] What role does the **`condition_variable_any`** named **`cv`** play? Is it strictly necessary?

The condition variable allows the server to intelligently stall until it knows whether one or more files need to be processed. [1 point]

Necessary? Depends on your definition of strict [1 point]

- not necessary if you're cool telling requesters there's nothing outstanding and allowing them to exit, provided you're okay with a smaller set of nodes living on.

- necessary if you insist on parallelism until the end or want to take consistently failing nodes offline.

f. Assume that **`markFilePatternAsProcessed`** relied on **`cv.notify_one()`** instead of **`cv.notify_all()`**. Could the server have deadlocked? Explain.

Certainly could. If only one file was in flight, then the server will inform just one of possibly many requesting threads that everything's been processed, but the other requests will be left trapped in the wait call. [0, 1, or 2 points on sliding scale]

g. [2 points] Assume that **`rescheduleFilePattern`** relied on **`cv.notify_one()`** instead of **`cv.notify_all()`**. Could the server have deadlocked? Explain.

Nope. Those requesters left hanging will continue to hang until the one notified request (along with those that never waited in the first place) collectively process all remaining files, at which point **`markFilePatternAsProcessed`** **`notify_all`** call will wake the waiters up.

h. [2 points] Assume that **`rescheduleFilePattern`** and **`markFilePatternAsProcessed`** called **`cv.notify_all()`** unconditionally without the **`if`** test guarding it. Does this cause any problems? Explain.

Doesn't impact functionality or ability to process all files. It only wakes up multiple waiters unnecessary. [0, 1, or 2 points on sliding scale]

i. Briefly explain why a nonblocking HTTP server can manage many more open requests than a blocking one.

A server can maintain tens of thousands of open connections, and if all connections are nonblocking, then it can address all open connections in a single thread of execution. If they are blocking, then each open connection must be handled in a thread, and we're limited on the number of threads that can exist at any one time. [0, 1, or 2 points on sliding scale]

j. What problem do **`epoll_create`**, **`epoll_ctl`**, and **`epoll_wait`** solve? Explain.

A nonblocking I/O application might spin on the CPU during the large windows of time when not a single descriptor of producing (via **`read`** and **`accept`**) or consuming (via **`write`**) anything. That wastes CPU time better allocated to other processes. The **`epoll`** functions allow us to efficiently block on a set of nonblocking descriptors until at least one can produce or consume something via **`accept`**, **`read`**, or **`write`**.