Winter 2017 March 22nd, 2017

CS110 Final Examination

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): _____**`@stanford.edu`**

Last Name: _____

First Name: _____

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]**	_____	_____

Relevant Prototypes

```
// filesystem access
int open(const char *path, int oflag, ...);       // returns descriptor
ssize_t read(int fd, char buffer[], size_t len);  // returns num read, 0 at eof
ssize_t write(int fd, char buffer[], size_t len); // returns num written
int close(int fd); // ignore retval
int pipe(int fds[]); // argument should be array of length 2, ignore retval
int dup2(int old, int new); // ignore retval
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2

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

```
template <typename T>
class list {
public:
   list();
   void push_back(const T& elem);
   T& front();
   void pop_front();
};

class thread {
public:
   thread();
   thread(Routine routine, ...);
   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 Pred>
     void wait(mutex& m, Pred p);
   void notify_one();
   void notify_all();
};
```

```
template <typename T>
class vector {
public:
   vector();
   template <typename InputIter>
     vector(InputIter begin, InputIter end);
   size_t size() const;
   void push_back(const T& elem);
   T& operator[](size_t i);
   const T& operator[](size_t i) const;
   iterator begin();
   iterator end();
   const_iterator cbegin();
   const_iterator cend();
};

template <typename U, typename T>
struct pair<U, T> {
   U first;
   V second;
};

template <typename U, typename T>
pair<U, T>
make_pair<U, T>(const U& u, const T& t);

template <typename InputIter,
          typename OutputIter,
          typename Pred>
OutputIter
remove_copy_if(InputIter begin1, InputIter end1,
               OutputIter begin2, Pred pred);

template <typename InputIter, typename Pred>
void sort(InputIter begin, InputIter end,
          Pred pred);
```

Problem 1: File System Redux [10 points]

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] 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);
```

b. [2 points] 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?

c. [2 points] 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.

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.

e. [2 points] 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?

Problem 2: Distributed Bubble Sort [20 points]

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. [2 points] 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.

b. [2 points] 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?

c. [2 points] 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**.

d. [2 points] 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**?

e. [2 points] 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 your response.

f. [2 points] 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?

g. [2 points] 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.

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?

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

j. [2 points] 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.

Problem 3: Event Barriers [15 points]

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. [0 points] 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 ☺




};
```

b. [2 points] Implement the **`EventBarrier`** constructor in the space below. Your implementation should be very short.

```
/**
 * Constructor: EventBarrier
 * -------------------------
 * Constructs an event barrier to block zero or more
 * consumer threads until the barrier is lifted.
 */
EventBarrier::EventBarrier() {
```

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

```
/**
 * Method: wait
 * ------------
 * Causes the calling thread to wait until
 * the barrier it lifted.  If the barrier is
 * currently up, then wait returns immediately
 * without blocking.
 */
void EventBarrier::wait() {
```

d. [5 points] 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.

```
/**
 * Method: lift
 * ------------
 * Lifts the barrier and notifies all consumers that they can
 * continue.  lift itself blocks until all consumers have
 * advanced past the barrier.
 */
void EventBarrier::lift() {
```

e. [5 points] 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.

```
/**
 * Method: past
 * ------------
 * Called by a consumer to inform the producer and all other consumers
 * that it's responded to the event and moved past the barrier.
 * past blocks until all other consumer have also invoked past, so all
 * consumers effectively return from past at the same time.
 */
void EventBarrier::past() {
```

Problem 4: Thread Pool With Thunk Priorities [15 points]

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. [0 points] 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:
  ThreadPool();
  void schedule(const function<void(void)>& thunk, size_t priority);
  void wait();
  ~ThreadPool();

 private:
  void dispatcher();
  void worker(size_t workerID);
  thread dt;
  struct {
    bool isAvailable = true;
    function<void(void)> thunk;
    thread thread;
    // add other data members as needed
  } workers[128];
  // add other data members and helper methods as needed
};
```

b. [5 points] 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() {
```

c. [5 points] 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() {
```

d. [5 points] 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) {
```

e. [5 points] 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 < 128);
```

f. [5 points] 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() {
```

g. [5 points] 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.

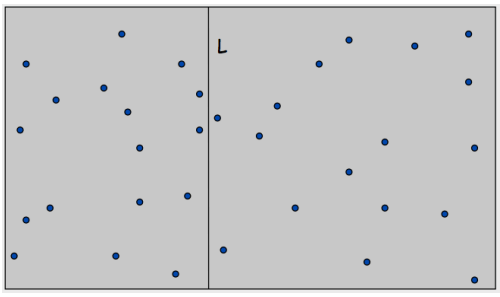
```
ThreadPool::~ThreadPool() {
```

Problem 5: Closest Points [15 points]

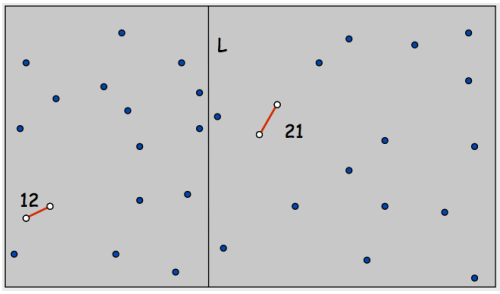
Computational geometry is a branch within computer science concerned with the data structures and algorithms easily stated in geometric terms, and has applications in data mining, computer vision, molecular modeling, air traffic control, and so forth. The closest pair problem is a common one: given a 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:

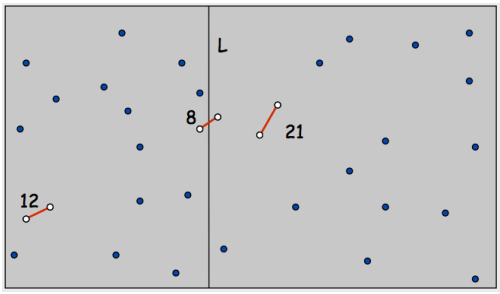
- Sort the collection of points by increasing x, and separate the sorted set into two collections of roughly the same size such that each point in the first is to the left of each and every point in the second. Pictorially, draw a vertical line L so that each side encloses roughly half the points, as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



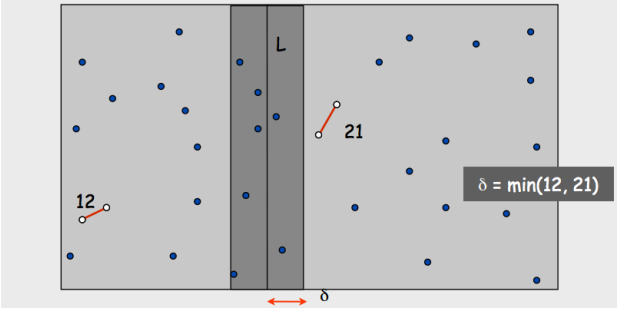
- Recursively find the closest pair in each half:



- 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.

[15 points] 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> findClosestPoints(vector<point> points) {
```

Problem 6: Concurrency and Networking Redux [20 points]

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

b. [2 points] 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.

c. [2 points] **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?

d. [2 points] 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.

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?

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

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

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.

i. [2 points] Briefly explain why a nonblocking HTTP server can manage many more open requests than a blocking one.

j. [2 points] What problem do **`epoll_create`**, **`epoll_ctl`**, and **`epoll_wait`** solve? Explain.

```
struct point {
  double x;
  double y;
};

double distance(const point& p1, const point& p2) {
  double dx = p1.x - p2.x;
  double dy = p1.y - p2.y;
  return sqrt(dx * dx + dy * dy);
}

double distance(const pair<point, point>& points) {
  return distance(points.first, points.second);
}

pair<point, point> closestOfThree(const vector<point>& points) {
  assert(points.size() > 1 && points.size() < 4);
  pair<pair<point, point>, double> nearest(make_pair(points[0], points[1]), distance(points[0], points[1]));
  if (points.size() == 2) return nearest.first;
  if (distance(points[0], points[2]) < nearest.second)
    nearest = make_pair(make_pair(points[0], points[2]), distance(points[0], points[2]));
  if (distance(points[1], points[2]) < nearest.second)
    nearest = make_pair(make_pair(points[1], points[2]), distance(points[1], points[2]));
  return nearest.first;
}

pair<point, point> closestInBand(const vector<point>& points) {
  assert(points.size() > 1);
  pair<pair<point, point>, double> nearest(make_pair(points[0], points[1]), distance(points[0], points[1]));
  for (size_t lh = 0; lh < points.size(); lh++) {
    for (size_t rh = lh + 1; rh < min(points.size(), lh + 12); rh++) {
      if (distance(points[lh], points[rh]) < nearest.second)
        nearest = make_pair(make_pair(points[lh], points[rh]), distance(points[lh], points[rh]));
    }
  }

  return nearest.first;
}

// each of these functions can be passed as the third argument to sort
bool xcompare(const point& one, const point& two) { return one.x < two.x; }
bool ycompare(const point& one, const point& two) { return one.y < two.y; }
```