Winter 2019 March 20th, 2019

# 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. And if you're taking the exam remotely, please scan and email a copy of your completed exam to **`jerry@cs.stanford.edu`**.

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. Forks | [10] | _____ | _____ |
| 2. Threading | [18] | _____ | _____ |
| 3. High Performance, Pre-forking Servers | [27] | _____ | _____ |
| 4. Short Answers | [20] | _____ | _____ |
| **Total** | **[75]** | _____ | _____ |

## Relevant Prototypes

```
// filesystem access
int close(int fd); // ignore retval
int dup(int fd);   //
int dup2(int oldfd, int newfd); // ignore retval
int pipe(int fds[]); // ignore retval
int pipe2(int fds[], int flags); // ignore retval, flags typically O_CLOEXEC
#define STDIN_FILENO 0
#define STDOUT_FILENO 1

// exceptional control flow and multiprocessing
pid_t fork();
pid_t waitpid(pid_t pid, int *status, int flags);
int execvp(const char *path, char *argv[]); // ignore retval
int kill(pid_t pid, int signal); // ignore retval
typedef void (*sighandler_t)(int sig);
sighandler_t signal(int signum, sighandler_t handler); // ignore retval
int sigemptyset(sigset_t *set); // ignore retval
int sigaddset(sigset_t *set, int sig); // ignore retval
int sigprocmask(int how, const sigset_t *set, sigset_t *old); // ignore retval

#define WIFEXITED(status)   // macro
#define WIFSTOPPED(status)  // macro

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

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

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

class ThreadPool {
public:
   ThreadPool(size_t numThreads);
   void schedule(Thunk t);
   void wait();
};
```

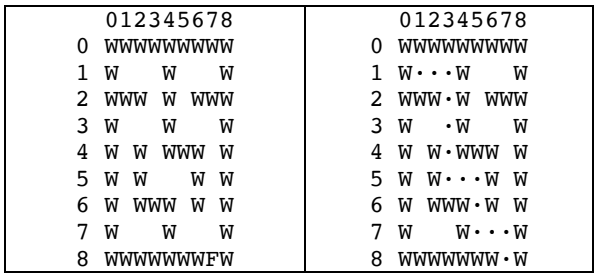
```
template <typename T>
class vector {
public:
   size_t size() const;
   void push_back(const T& elem);
   T& operator[](size_t i);
   const T& operator[](size_t i) const;
};

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

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

Template <typename Key, typename Value>
class map {
public:
   // iter points to pair<Key, Value>
   size_t size() const;
   iter find(const Key& k);
   Value& operator[](const Key& k);
};
```

**Problem 1: Maze Solving with Forks [10 points]**




The diagram above-left shows a maze, where W denotes walls, spaces represent open paths, and the F in the bottom right is the finish. The numbers denote the (x,y) coordinates of the maze, with the Finish at coordinate (7,8). The diagram on the right shows the same maze with dots (·) representing the solved maze path from top left to the finish on the bottom right. Maze-solving can be performed with a depth-first search, which is often accomplished recursively.

Another way to think about solving a maze is to simply check every possible direction (North, South, East, and West) and fork processes to handle each direction where an open path is present, with each process keeping track of its own solution path. It is also necessary to keep track of visited positions, to avoid infinite loops.

For this problem, you will have three types, with the functions or variables you need shown:

```
enum Location {PATH, WALL, FINISH};

struct coordinate {
    int x,y;
};

class Maze {
    public:
        Location getVal(coordinate c);
};
```

You can assume that the `coordinate` definition also has the required operator overloads to enable storage in a container, such as a `set`.

As a refresher on `enum`s and structs, you can use a `Location` and `getVal()` as follows:

```
Location loc = getVal({1,2});
if (loc == PATH) { … }
```

Given the above definitions, write the following function:

```
bool solveMazeFork(Maze &maze, coordinate start, vector<coordinate> &path);
```

Your function should keep track of the solution path in the `path` vector, and should create a fork at each potential path. For example, in the above maze it is obvious that the solution starting from coordinate (1,1) only has one possible direction (east), so your program would only fork once to handle that condition, with the fork progressing to coordinate (2,1). The same is true at coordinate (2,1), which can only progress to (3,1), because (2,1) has already been visited. At (3,1), the only path is to (3,2) (by going south), and from (3,2) you must go to (3,3). However, at (3,3), there are two possible directions: (2,3) (by going west), and (3,4) (by going south). At this point, your function should fork two processes, with one investigating the path at (2,3), and the other investigating the path at (3,4).

By continuing this process, one of the fork chains will eventually find the finish, at which point that process should return `true`. After your function produces all of its children, it should `wait()` for them and return `false`.

*Notes:*

1. You can assume that there will be only one solution for a maze.
2. Mazes are completely surrounded by walls except for the finish, so there is no need to check any bounds (i.e., you will always eventually hit a wall in a given direction, unless it is the finish).
3. Your function can solve the maze within a single `while()` loop, with the necessary forking. The only processes that need to continue looping are the forked processes.
4. You should keep track of coordinates you have visited as you visit them, although you do not need to communicate this information between processes, except at the time a fork is generated.
5. You may use any `std::` containers to hold information needed for this problem (but a set of visited coordinates is the only extra container my solution uses).

[10 points] Write your function on the next page.

```
bool solveMazeFork(Maze &maze, coordinate start, vector<coordinate> &path) {
```

**Problem 2: Threaded Election Vote Tallying [20 points]**

This problem simulates an election for a national popular vote in the U.S., where there are two major parties, Democrats (`DEM`) and Republicans (`REP`). The program uses threads to distribute the work.

In the U.S., voting happens at the state level, and is further distributed to the precinct (local) level on a town-by-town basis. After voting closes for the day, a precinct forwards its votes in a bundle to the state. At the state level, two people (one from each major political party, `DEM` and `REP`) must tally the votes independently, and agree on the totals (assume only two parties can accumulate votes).

If the two state vote-counters disagree on the totals, then the counting process continues, until eventually, the two counters agree on the totals.

Once a precinct's votes have been counted and verified, they will be added to the state total.

Finally, after all of the precincts for a state have added their totals to the state count, a national counter will add the votes to the national total, and report the current results.

This program models the following:

1. There is a national counter function, `countNationalVotes`, in a thread that waits for state totals to come in, at which point it updates the totals and reports on the current total. You will write most of this function.

2. There are 51 threads, using the `stateCount` function, that coordinate state voting (50 states + Washington, D.C.). Each of the 51 state threads loop through all of their precincts and start a thread for each with the `precinctCount` function, using a `precinctThreadSem` semaphore to limit the precinct threads. When a state thread has the result from all precincts, it signals `countNationalVotes`. You will write part of the `stateCount` function.

3. The precincts require one representative from each party to count independently. This happens in parallel with two counter threads, although there are a limited number of counters per party, and this is handled through two semaphores. If the counters disagree, the process repeats, until they agree. When they agree, the state totals are updated (and this could happen simultaneously, so proper locking is necessary). Before returning, the precinct signals the `stateCount` function via the `precinctThreadSem` semaphore to allow another precinct to begin counting. You will write most of this function.

**Important program details:**

```
enum Party { DEM, REP };
```

The `Party enum` defines the two parties that can collect votes.

To use the `Party` enum, you can check a vote as follows:

```
Party vote = <get a vote from a list of votes>
if (vote == DEM) { ... } else { ... }
```

```
static semaphore demCounters(kTotalStateCounters);
static semaphore repCounters(kTotalStateCounters);
```

These two semaphores represent the number of available counters for each party (i.e. the number of permissions slips is the number of counters that can currently be summoned).

```
struct stateInfo {
   string name;
   int numPrecincts;
   vector<vector<Party>> rawPrecinctVotes;
   int totalDVotes, totalRVotes;
};
```

The `stateInfo` struct holds information about one single state. `rawPrecinctVotes` holds a vector of votes for each precinct. The totals for each party are updated when each precinct is counted.

```
static vector<stateInfo> allStatesInfo;
```

The global `allStatesInfo` vector holds all the structs for the 51 states. It is global for simplicity in handling the data in the threads.

```
struct nationalCount {
   mutex listUpdateMutex;
   list<stateInfo *> finishedStates;
   semaphore waitingStates;
} nationalCount;
```

The global `nationalCount` struct is used to coordinate between the state counters and the national counter. As each state finishes counting, it pushes back a pointer to its `stateInfo` variable, which can then be used by the `countNationalVotes` to update the national data. The `listUpdateMutex` is used to lock the list when it is updated, and the `waitingStates` semaphore is used to signal the national counter that there are states that have finished.

Most of the program is written below. Fill in the parts indicated by

```
/* student code below here */
```
and
```
/* student code above here */
```

**Read all the code before starting to write your own code!**

```
enum Party { DEM, REP };

static const int kNumStates = 51; // 50 + Washington, D.C.
static const int kTotalStateCounters = 10;

static semaphore demCounters(kTotalStateCounters);
static semaphore repCounters(kTotalStateCounters);

struct stateInfo {
  string name;
  int numPrecincts;
  vector<vector<Party>> rawPrecinctVotes;
  int totalDVotes, totalRVotes;
};

static vector<stateInfo> allStatesInfo;

struct nationalCount {
  mutex listUpdateMutex;
  list<stateInfo *> finishedStates;
  semaphore waitingStates;
} nationalCount;

static int readStateInfo();
static void populatePrecinctVotes();

static void countVotes(vector<Party>&rawVotes, Party p, int &totalD, int &totalR);

static void precinctCount(vector<Party>&rawVotes, int stateIndex,
       mutex &updateStateTotalsMutex, semaphore &precinctThreadSem) {
 // Complete the first part of this function, which creates two independent
 // threads that call countVotes, one for DEM with totalDfromD and
 // totalRFromD, as references, and the other for REP, with totalDfromR
 // and totalRfromR, as references. You should use the demCounters and
 // repCounters semaphores to limit the number of counters across all threads.

 stateInfo &info = allStatesInfo[stateIndex];
 while (true) { // keep counting until both parties agree on vote counts
   int totalDFromD, totalRFromD; // DEM vote counts, passed by reference
   int totalDFromR, totalRFromR; // REP vote counts, passed by reference

   /* student code below here [4 points] */
```

```
   /* student code above here */

   // complete the second part of this function, which updates
   // info.totalDVotes and info.totalRVotes in a thread-safe manner,
   // and then signals the state counter to allow another precinct to count
   if (totalDFromD == totalDFromR && totalRFromD == totalRFromR) {

   /* student code below here [5 points] */









   /* student code above here */
   break;
  }
 }
}


static void stateCount(int stateIndex) {
 // have each party count each precinct, and update total
 stateInfo &info = allStatesInfo[stateIndex];
 vector<thread> precinctThreads;
 mutex updateStateTotalsMutex;
 semaphore precinctThreadSem(10);
 for (size_t i=0; i < info.rawPrecinctVotes.size(); i++) {
     precinctThreadSem.wait();
     precinctThreads.push_back(thread([&info, i, stateIndex,
                &updateStateTotalsMutex, &precinctThreadSem]() {
                    precinctCount(info.rawPrecinctVotes[i], stateIndex,
                      updateStateTotalsMutex, precinctThreadSem);
                    }));
 }
 for (thread &t : precinctThreads) t.join();

 // at this point, all the states have finished counting
 cout << oslock << "Done counting " << info.name << ": D:"
  << info.totalDVotes << " R: " << info.totalRVotes << endl << osunlock;
```

```
 // Complete the rest of this function to allow the countNationalVotes
 // thread to update the national count based on the state totals.

 /* student code below here [4 points] */









 /* student code above here */

 cout << oslock << info.name << " has reported its totals." << endl << osunlock;
}

static void countNationalVotes() {
  int nationalDVotes = 0;
  int nationalRVotes = 0;
  cout << oslock << "National counter will update national totals." << endl << osunlock;

  for (unsigned int i = 0; i < kNumStates; i++) {
    stateInfo *info;

    // complete the rest of this function in order to add the state totals to the
    // nationalDVotes and nationalRVotes totals. The loop should only proceed
    // when signaled by a state that has completed its counting.

    /* student code below here [5 points] */









    /* student code above here */
```

```
  cout << oslock << "Added " << info->name << " to total." << endl << osunlock;
  cout << oslock << "Current Totals: " << "D: " << nationalDVotes <<
   ", R: " << nationalRVotes << endl << osunlock;
 }
 cout << oslock << "  Final Totals: " << "D: " << nationalDVotes <<
  ", R: " << nationalRVotes << endl << osunlock;
}

int main(int argc, const char *argv[]) {
  int numStates = readStateInfo();
  populatePrecinctVotes();

  thread national(countNationalVotes);
  vector<thread> states_th;

  // start 51 threads, one for each state
  for (int i = 0; i < numStates; i++) {
    states_th.push_back(thread(stateCount, i));
  }

  for (thread& state : states_th) {
    state.join();
  }

  national.join();

  return 0;
}
```

**Problem 3: High Performance, Pre-forking Servers [27 points]**

Many servers are coded to an architecture that **pre-forks** a collection of worker executables to handle individual client requests. For the purposes of this problem, each worker executable is required to self-halt until instructed to proceed, read single-line requests from standard input, publish single-line responses to standard output, and repeat until its standard input reads **`EOF`**, at which point it exits. A simplified version of Assignment 3's **`factor.py`**, presented below, satisfies this requirement.

```
while True:
    os.kill(os.getpid(), signal.SIGSTOP)
    try: num = int(raw_input())
    except EOFError: break
    response = factorization(num) # returns a single line with no \n
    print response                # prints response with terminating \n
    sys.stdout.flush()            # force a flush
```

Of course, there's no reason to assume a server's workers are always **`factor.py`**. They can be any program that bows to the requirements.

```
while (true) {
    <self-halt>
    <read one-line request from stdin>
    if (<unable-to-read-data-ever-again>) break;
    <process request, generate one-line response>
    <publish to stdout, and flush>
}
```

For this problem, you should assume that every single worker executable is implemented using the above structure.

When launched, the server spawns off (i.e. pre-forks) a specific number of workers and maintains two pipes on behalf of each: a **supply** descriptor that leads to the worker's standard input, and a **ingest** descriptor that pulls from the worker's standard output. Every time the server accepts a new connection, the server schedules a thread to execute within a thread pool, and that scheduled thread reads a one-line request from the client, forwards it verbatim to an available worker, waits for the worker to handle the request, receives the one-line response from the worker, and forwards that response back to the client. It's similar to your proxy assignment, except that requests and responses are always one line, and everything—the server and all its workers—run on the same machine.

One additional feature! The server is prepared to **spawn additional workers**—up to a maximum number—if the number of client connections seems high. The server is also prepared to shut down workers when the vast majority of them are idle. This feature is a gesture to how a server might self-tune so that number of workers goes up and down with the perceived client traffic.

The **`main`** function for this server is nothing more than a wrapper around some command line parsing, a server configuration, and a method call instructing the server to loop forever. Check

out the **prefork-server.cc** file below (the implementation of **parseCommandLine** has been omitted, but you can intuit what it must do):

```
static const short kDefaultPort = 24680;
static const size_t kNumCPUs = sysconf(_SC_NPROCESSORS_ONLN);
int main(int argc, char *argv[]) {
  short port = kDefaultPort;
  size_t low = kNumCPUs, start = 2 * low, high = 4 * low;
  argv += parseCommandLine(argc, argv, port, start, low, high);
  PFServer server(port, start, low, high, argv); // argv now addresses worker argv
  server.run();
  return 0;
}
```

Assuming a working **PFServer** constructor and **run** method, the following would launch a server with 12 workers with the guarantee that even though it may add or shut down workers, the number will always be between 8 and 32, inclusive:

```
myth51:$ ./prefork-server --port 24680 --low 8 --start 12 --high 32 ./factor.py
Server listening to port 24680.
```

Until I hit ctrl-C, my server is running on **myth51**, bound to port **24680**. That means I can use my own laptop to connect to **myth51:24680** and play the role of client, as with:

```
jerry$ telnet myth51.stanford.edu 24680
Trying 171.64.15.23...
Connected to myth51.stanford.edu.
Escape character is '^]'.
123456789
123456789 = 3 * 3 * 3607 * 3803
Connection closed by foreign host.
jerry$
```

If I'm the only one hitting my server and I manually connect four or more times (and I'm slow about it), I'll see evidence that my server self-detects its unpopularity and shuts down workers.

```
myth51:$ ./prefork-server --port 24680 --low 8 --start 12 --high 32 ./factor.py
Server listening to port 24680.
Lots of idle workers... shutting one down.
Lots of idle workers... shutting one down.
Lots of idle workers... shutting one down.
Lots of idle workers... shutting one down.
```

If I now write a multithreaded program that flash mobs the server with a huge number of simultaneous requests, it begins to add more workers, up to a maximum.

```
myth51:$ ./prefork-server --port 24680 --low 8 --start 12 --high 32 ./factor.py
Server listening to port 24680.
Lots of idle workers... shutting one down.
Lots of idle workers... shutting one down.
Lots of idle workers... shutting one down.
Lots of idle workers... shutting one down.
Not enough workers... spawning one more.
Not enough workers... spawning one more.
// 22 more lines like the one above
```

For this problem, you're given a full class declaration and a partial implementation. Your job is to provide implementations for those helper methods we've not provided.

The definition of the **PFServer** class is presented below. You won't need to add anything to the **.h** file (in fact, you're not permitted to), but you'll provide implementations for the last four **private** methods, the prototypes of which are highlighted in **bold**.

```
class PFServer {
 public:
  PFServer(short port, size_t start, size_t low, size_t high, char *argv[]);
  void run();

 private:
  short port;
  size_t start, low, high;
  char **argv;

  int server;
  std::map<pid_t, std::pair<int, int>> bridges;
  std::list<pid_t> available;
  std::mutex m;
  semaphore numAvailable;
  ThreadPool pool;

  void spawnInitialWorkers();
  void optimizeNumWorkers();
  bool shouldSpawnWorker();
  bool shouldShutdownWorker();

  void spawnWorker();
  void handleRequest(int client);
  void shutdownWorker();
  void markWorkersAsAvailable();
};
```

The **port**, **start**, **low**, **high**, **argv**, and **server** fields should be self-explanatory once you see the code I provide. As for the others:

- The **bridges** maps stores all of the worker pids, and associates each pid to the supply and ingest descriptors described earlier.
- The **available** list maintains a FIFO queue of pids, each of which identifies some worker that's self-halted and staged to be fed a single-line request from the server once its time comes.
- **m** is used to guard access to all the **private** data members that might be accessed by two or more threads at the same time.
- **numAvailable** wraps as many permission slips are there are pids in **available**.
- **pool** is used the same way threads pool were used for Assignments 7 and 8—to handle as many incoming requests as possible while allowing the main thread to accept connections with minimal interruption.

Presented below are the implementations of the class entries we've provided. We omit the implementations of **blockSignals** and **unblockSignals**, since you can infer what they must do without having to see code for them. We rely on a new version of **createServerSocket** to create a server socket that's automatically closed across **execvp** boundaries. And we rely on a second version of **accept**—you'll see it as **accept4**—because that one can be called, as I have called it, to return sockets that also auto-close at **execvp**.

You'll be implementing the four helper functions you see in bold.

```
PFServer::PFServer(short port, size_t start, size_t low, size_t high, char *argv[]):
port(port), start(start), low(low), high(high), argv(argv), pool(high) {

  signal(SIGCHLD, [this](int unused) { markWorkersAsAvailable(); });
  server = createServerSocket(port);
  cout << "Server listening to port " << port << "." << endl;
}

void PFServer::run() {
   spawnInitialWorkers();
   while (true) {
      int client = accept4(server, NULL, NULL, SOCK_CLOEXEC);
      pool.schedule([this, client] { handleRequest(client); });
   }
}

void PFServer::spawnInitialWorkers() {
   blockSignals({SIGCHLD});
   for (size_t i = 0; i < start; i++) spawnWorker();
   unblockSignals({SIGCHLD});
}

bool PFServer::shouldSpawnWorker() {
   return available.empty() && bridges.size() < high;
} // if all workers are working, create one more (up to a limit)

bool PFServer::shouldShutdownWorker() {
   return bridges.size() > low && bridges.size() - available.size() == 1;
} // if all other workers seem to be idle, kill one off

void PFServer::optimizeNumWorkers() {
   if (shouldSpawnWorker()) {
      cout << "Not enough workers... spawning one more." << osunlock;
      spawnWorker();
   } else if (shouldShutdownWorker()) {
      cout << "Lots of idle workers... shutting one down." << endl;
      shutdownWorker();
   }
}
```

a. [7 points] Implement the **spawnWorker** method, which creates a new child executable running whatever the **argv** data field stores. **spawnWorker** should initialize two descriptors—one that feeds the new worker's standard input, and a second that pulls from the new worker's standard output—and ensure the descriptor pair is properly catalogued in **bridges**. Your implementation should assume that **SIGCHLD** is being blocked and that no other threads are capable of modifying any of the server's **private** data fields. Don't orphan any descriptors and configure your descriptors so their reference counts are never artificially inflated. Your implementation should **not** modify the **available** queue, since you need to wait for the worker to self-halt and trigger your **SIGCHLD** handler—you'll implement that next—to mark it at available. You may not use any **subprocess** function written in lecture or in an assignment unless you reimplement it and adapt it to the needs of this problem.

```
void PFServer::spawnWorker() {
```

b. [6 points] Next, implement the **markWorkersAsAvailable** method, which was installed as part of the **SIGCHLD** handler in the **PFServer** constructor. You may assume no other threads have a lock on **m** when this executes, so that you can freely update **private** data members as needed. Your **SIGCHLD** handler will be triggered to interrupt the currently executing thread whenever child processes halt, continue, or exit; so be sure your handler works for all three types of state changes.

```
void PFServer::markWorkersAsAvailable() {
```

c. [4 points] Next up is **shutdownWorker**, which gets called when **optimizeNumWorkers** (which we provided) decides the server has too many idle workers and wants to shut one down. Your implementation should assume that **SIGCHLD** is blocked and that the lock on **m** is held by the thread calling **shutdownWorker**. Your implementation should pluck some pid from the **available** queue, prompt that worker to exit gracefully, close down any descriptors that are no longer relevant, and remove any trace of the worker from the **bridges** map.

```
void PFServer::shutdownWorker() {
```

d. [4 points] To simplify the implementation of **handleRequest**, you're going to implement a short utility function, and you're going to implement it two different ways. The prototype of the function looks like this:

```
static void forwardBytes(int source, int sink);
```

Its implementation is easy to describe: Keep reading characters from **source** and forwarding them to the **sink** until you read and forward a newline character. Your implementation should work regardless of how long the sequence of bytes ends up being.

You're to implement it two different ways, just to show us you have a handle on low-level I/O and also on the **sockstream** classes used in Assignments 7 and 8.

- [2 points] The first way should use low-level I/O, exposed **read** and **write** calls, and a character buffer of size 8 to shovel bytes from **source** to **sink**. You should not close **source** or **sink**.

```
static void forwardBytes(int source, int sink) {
```

- [2 points] The second way should rely on **sockbuf**s and **iosockstream**s to do precisely the same thing. The implementation is short, though it's tricky, since you need to ensure that **source** and **sink** aren't closed at the end. This implementation should not include any exposed calls to **read** or **write**.

```
static void forwardBytes(int source, int sink) {
```

e. [6 points] Finally, present the implementation of **handleRequest**, which waits for a worker to become available, enlists that worker, and managed all communication between the client and the worker to fully service the client. After you've selected a worker and removed it from the queue (but before you start tunneling bytes back and forth between the client and the worker), you should call **optimizeNumWorkers** to quickly decide if the number of workers should be increased or decreased by one.

Your implementation of **handleRequest** should guard against race conditions and deadlock, but it should also make sure to maximize parallelism without holding locks or blocking signals longer than necessary. Your implementation will make use of your **forwardBytes** function from part d.

```
void PFServer::handleRequest(int client) {
```

**Problem 4: Short Answers [20 points]**

Unless otherwise noted, your answers to the following questions should be at most 75 words. Responses conspicuously longer than 75 words 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] Descriptors can be configured so they are automatically closed when **`execvp`** is called, and whether or not a descriptor is self-closing on **`execvp`** boundaries is tracked using just one bit of information. Where is that bit stored? In the descriptor table entry? Or in the open file table entry that's referenced by the descriptor? Briefly defend your answer.

b. [2 points] The **`vfork`** system call has the same effect as **`fork`**, except that the child process created by **`vfork`** cannot modify any variables whatsoever, and the child process must either lead to a call to **`_exit`** (a special form of **`exit`**) or one of the **`exec*`** functions (e.g. **`execvp`**, **`execlp`**, **`execve`**, etc). **`vfork`** can be used in instead of **`fork`** to boost the performance of time- and resource-sensitive applications, because it doesn't create a new virtual address space for the child until **`execvp`** is called.

Explain why the implementation of **`vfork`** (as opposed to **`fork`**) necessarily suspends the parent process until the child process has either terminated or called one of the **`exec*`** functions.

c. [2 points] Your **`farm`** program Assignment 3 was careful to use **`sched_setaffinity`** to assign each of eight workers to run on a dedicated CPU. Briefly describe the benefits of doing this and why it leads to optimal execution speed.

d. [2 points] The Assignment 4 specification was clear that you needed to support pipelines (e.g. **`echo "123" | conduit --count 4 --delay 1 | wc`**). The assignment specification, however, didn't ask that you do anything special if one of the named programs (e.g. **`conduit`**) couldn't be executed (because it's missing, or it's there but it's not executable, or it is but you don't have permission to execute it).

Describe how you could use the **`ptrace`** system call to ensure that all individual commands successfully **`execvp`** before allowing them all to continue, but terminating them all without allowing any to continue beyond their **`execvp`** calls if one or more **`execvp`** calls fail.

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?

f. [6 points] When two or more threads are blocked on a call to **`mutex::lock`**, any one of them might be selected to acquire the lock once the **`mutex`** becomes available. Restated, the **`mutex`** isn't obligated to maintain any sort of FIFO queue to ensure the thread waiting longer than any other is chosen first.

A **strong** mutex, or a **`smutex`**, ensures that blocked threads are woken up in the same order they are blocked. There are many **`smutex`** implementations, and one that relies on a queue of **`condition_variable_any`**s is presented below (interface on the left, implementation on the right).

```
// smutex.h
class smutex {
 public:
  void lock();
  void unlock();

 private:
  mutex m;
  list<condition_variable_any *> queue;
};
```

```
// smutex.cc
void smutex::lock() {
  condition_variable_any cv;
  lock_guard<mutex> lg(m);
  queue.push_back(&cv);
  cv.wait(m, [this, &cv] {
    return queue.front() == &cv;
  });
  queue.pop_front();
}

void smutex::unlock() {
  lock_guard<mutex> lg(m);
  if (!queue.empty())
    queue.front()->notify_all();
}
```

Study the implementation of the **`smutex`** methods and answer the following questions:

- [2 points] Does the implementation guarantee that a thread calling **`smutex::lock`** before any others gets the lock on the **`smutex`** first? Why or why not?

- [2 points] Can the call to **`notify_all`** be replaced with a call to **`notify_one`** without impacting functionality? Very briefly defend your answer.

- [2 points] Can the **`queue.pop_front()`** line in **`smutex::lock()`** be moved so that it's the last line in **`smutex::unlock()`** instead? Why or why not?

g. [2 points] A good rule of thumb is that the number of threads used for a CPU-bound computation should equal the number of CPUs, whereas the number of threads used for an I/O-bound computation should be two to four times the number of CPUs. Explain why.

h. [2 points] During the last CS110 lecture, Chris introduced the **`epoll`** suite of functions—**`epoll_create`**, **`epoll_ctl`**, and **`epoll_wait`**—as a better way to implement nonblocking servers to handle thousands of client requests simultaneously. However, **`epoll_wait`** is a slow system call, counter to the idea of nonblocking. Why, then, is using **`epoll_wait`** a good thing?