CS110 Final Assessment Solution

This is a closed book, closed note, closed computer exam, and you have 180 minutes to complete all problems. You don’t need to \texttt{include} any header files, and you needn’t guard against any errors or system call failures unless specifically instructed to do so. Understand that most points are awarded for concepts taught in CS110.

Good luck!

SUNet ID (username): _____________________@stanford.edu

Last Name: ____________________________________________

First Name: __________________________________________

I accept the letter and spirit of the honor code.

[signed] ________________________________________________

\textbf{Problems:}

1. \texttt{splayline} \hspace{1cm} 10 points
2. Synchronization, Races, and Signals \hspace{1cm} 8 points
3. Pipes and Condition Variables \hspace{1cm} 12 points
4. Dispatcher-Free \texttt{ThreadPools} \hspace{1cm} 15 points
5. Batched HTTP Requests \hspace{1cm} 10 points

55 points total

A reference sheet of function signatures and constants is included at the back of the assessment.
**Problem 1: splayline [10 points]**

Let's revisit the splay function from Assessment 2, which comes with the following prototype:

\[
\text{int splay(int pullfds[], size_t len);}\]

How does splay work? Consider these two lines of code:

\[
\text{int pullfds[8];}
\text{int pushfd = splay(pullfds, 8);}\]

The splay call above updates each of the eight pullfds entries with read-oriented descriptors and returns a write-oriented descriptor that's caught in pushfd. Bytes published to pushfd are copied eight times, and each of the eight copies is readable through the descriptors residing in pullfds.

For this problem, you're to assume you have a working splay function so you can implement splayline, which comes with the following prototype:

\[
\text{void splayline(char *pushargv[], const vector<char *[]> & pullargvs);}\]

splayline spawns a total of pullargvs.size() + 1 child processes, where one is run with the arguments in pushargv, and the remaining are each run with a different one of the argument vectors in pullargvs. All argument arrays are NULL-terminated. The standard output of the process running pushargv should be fed as the standard input to each of the remaining child processes running with pullargvs[i].

You may assume that all system calls succeed, and you needn't do any error checking. Your implementation of splayline should create all of the child processes, rewire file descriptors as described, close all unused descriptors (in both the parent and all of the children), and wait for all child processes to run to completion before returning. For convenience, you can assume that all descriptors surfaced and returned by splay will automatically close when execvp is called. The processes should run in parallel as much as possible.

Place your implementation of your splayline function on the next page.
Place your implementation of your **splayline** function in the space below.

```c
// from previous page, for reference
int splay(int pullfds[], size_t len);

void splayline(char *pushargv[], const vector<char *]> & pullargvs) {
    int pullfds[pullargvs.size()];
    int pushfd = splay(pullfds, pullargvs.size());
    pid_t pullpids[pullargvs.size()];

    for (size_t i = 0; i < pullargvs.size(); i++) {
        pullpids[i] = fork();
        if (pullpids[i] == 0) {
            dup2(pullfds[i], STDIN_FILENO);
            execvp(pullargvs[i][0], pullargvs[i]);
        }
        close(pullfds[i]);
    }

    pid_t pushpid = fork();
    if (pushpid == 0) {
        dup2(pushfd, STDOUT_FILENO);
        execvp(pushargv[0], pushargv);
    }
    close(pushfd);

    for (pid_t pullpid: pullpids) waitpid(pullpid, NULL, 0);
    waitpid(pushpid, NULL, 0);
}
```

We’ve also put a working version of **splay** and **splayline** up on [cplayground](https://cplayground).
Problem 2: Synchronization, Race Conditions, and Signals [8 points]

Answer each of the four short answer questions, limiting your responses to 50 or so words.

a. [2 points] The starter code for assign3’s trace includes the following code near the beginning of main. The execvp call transforms the child process to run the executable being traced, though that isn’t the focus here. We’re concerned more with the child’s decision to call raise(SIGSTOP) and for the parent’s decision to call waitpid where it does.

```c
pid_t pid = fork();
if (pid == 0) {
    ptrace(PTRACE_TRACEME);
    raise(SIGSTOP);
    execvp(argv[numFlags + 1], argv + numFlags + 1);
    return 0;
}

waitpid(pid, NULL, WUNTRACED);
kill(pid, SIGCONT); // continue self-halted child process
```

Why are the raise and waitpid calls necessary if trace is to successfully track all of the child’s system calls, and what could have happened had they been omitted?

The child halts before reaching execvp, else it might execute it (and perhaps some system calls beyond) before the parent is able to observe them. The synchronization provided by raise, waitpid, and kill in the parent is needed to manually advance the child, one system call at a time, and extract register values while doing so.

b. [2 points] Your assign4’s stsh implementation relied on synchronous signal handling, signal blocking, and sigwait in order to manage all of the relevant signals—SIGCHLD, SIGINT, SIGTSTP, in particular—that might be delivered during execution. Why is it important for all three of these signals to be blocked during the entire lifetime of the shell?

We don’t want signals to arrive and be consumed—even by a default handler—before or after the sigwait, else our inline signal handling code won’t always be executed. sigwait should be the only way for a signal of interest to be consumed and surfaced.
c. [2 points] Assume you launch the following background job at the `stsh` prompt, wait five seconds, and then forcibly terminate the second process via `sray`, as with:

```bash
stsh> sleep 100 | sleep 100 | sleep 100 &
[1] 10000 10001 10002
stsh> slay 10001
```

Are the two other `sleep` processes still running after the `sray` call? Or are one or both of them terminated as well? Explain your answer.

*Both the leading and the trailing sleep calls would execute to completion without interruption, because neither relies on standard input or output. (Aside: Had the first process been writing to standard out, the OS would have sent it a `SIGPIPE` and, by default, terminated it. And had the third process depended on standard in, standard in would have returned EOF and prompted it to execute accordingly.*)

d. [2 points] Briefly explain why two processes—each running `emacs`—are capable of coexisting and running concurrently even though they each believe they own all of computer memory and initially manipulate the same exact addresses?

*The addresses each `emacs` process owns are virtual addresses, and the OS maps virtual addresses in any one process to physical addresses that no other processes map to. This is the power of virtual memory and virtualization.*
Problem 3: Pipes, Producer/Consumer, and Condition Variables [12 points]

Presented below is a C++ class that might be used by the OS to support pipes. The class pipe interface is defined on the left, and the implementation of the two key methods are presented to the interface’s right:

class pipe {
public:
    void write(char ch);
    char read();
private:
    mutex m;
    condition_variable_any cvwrite, cvread;
    static const size_t kBufferSize = 8192;
    char buffer[kBufferSize];
    size_t count = 0, rpos = 0, wpos = 0;
};

void pipe::write(char ch) {
    unique_lock<mutex> ul(m);
    while (count == kBufferSize) cvread.wait(ul);
    count++;
    buffer[wpos++] = ch;
    if (wpos == kBufferSize) wpos = 0;
    cvwrite.notify_one();
}

char pipe::read() {
    unique_lock<mutex> ul(m);
    while (count == 0) cvwrite.wait(ul);
    count--;
    char ch = buffer[rpos++];
    if (rpos == kBufferSize) rpos = 0;
    cvread.notify_one();
    return ch;
}

The interplay between the read and write methods should remind you of the producer-consumer example covered in lecture, where we have a shared buffer that can be read from or written to. However, here we use condition variables instead of semaphores to guarantee that an excess of at most kBufferSize characters can build up in the circular buffer without blocking. The class presented above is thread-safe, so that multiple threads can read and write without fear of race condition or deadlock.

a. [2 points] The condition_variable_any::wait method expects a locked mutex as its first argument, and the condition variable locks and unlocks this mutex for us. Explain why the condition variable must unlock the mutex for us in wait instead of the caller unlocking right before calling wait.

To unlock to mutex before committing to wait is to introduce a race condition, as another thread might change the condition—there’s no lock preventing that from happening—before the waiting thread goes to sleep.

b. [2 points] Clearly and succinctly explain why, for any condition variable, the while loop around the wait call can’t be replaced by an if test on the same condition. Limit your response to 50 or so words.

When a thread returns from a wait call, it only knows the condition was once true, but it doesn’t know it’s true right now. This is a general truth about all calls to condition_variable_any::wait.
class pipe {
public:
    void write(char ch);
    char read();

private:
    mutex m;
    condition_variable_any cvwrite, cvread;
    static const size_t kBufferSize = 8192;
    char buffer[kBufferSize];
    size_t count = 0, rpos = 0, wpos = 0;
};

1 void pipe::write(char ch) {
2    unique_lock<mutex> ul(m);
3    while (count == kBufferSize) cvread.wait(ul);
4    count++;
5    buffer[wpos++] = ch;
6    if (wpos == kBufferSize) wpos = 0;
7    cvwrite.notify_one();
8 }
9
10 char pipe::read() {
11    unique_lock<mutex> ul(m);
12    if (count == 0) cvwrite.wait(ul);
13    count--;
14    char ch = buffer[rpos++];
15    if (rpos == kBufferSize) rpos = 0;
16    cvread.notify_one();
17    return ch;
18 }

Assuming empty buffer and count of 0:

- T1 enters read, acquires lock on m, detects 0 count, calls wait, releases m.
- T2 enters write, acquires m, adds character to buffer, increments count, but swapped off CPU before calling notify_one.
- T3 enters read, but blocks in unique_lock constructor. (T2 still owns lock on m.)
- T2 continues, calls notify_one to allow T1 to continue, releases lock on m, returns.
- T3 acquires lock on m, runs read to completion, leaving count at 0.
- T1 acquires lock on m, continues to consume nonexistent character and decrement count from 0 to -1.

d. [2 points] Assume the OS really does use the pipe class to support pipes as introduced during the multiprocessing segment of the course. Explain why the vnode table would be the data structure within the hierarchy of data structures to allocate and interact with a pipe object when a new pipe is created. Limit your response to 50 or so words.

vnode table entries are data structures that work to make everything (terminals, pipes, sockets, etc) behave like files, even when they aren’t. That way, the other data structures (open file table, descriptor tables) are insulated from the details of how the bytes are sourced and consumed.
Problem 4: Dispatcher-Free ThreadPools [15 points]

For this problem, you’re going to implement the ThreadPool all over again, although you’ll code to a slightly different design than you did for Assignment 5. In particular:

- Your ThreadPool will launch all worker threads at construction time.
- Your ThreadPool will be implemented without a dispatcher. Instead, each worker will dequeue and execute a thunk in cooperation with all other workers.
- Your ThreadPool should rely on a single mutex for all locking and synchronization needs, and as many condition_variable_any s as needed to support the inter-thread communication necessary for thread-safety and optimal use of the CPU’s cores. In particular, you shouldn’t declare any semaphores—just the one mutex and a collection of condition variables.
- We’ll ignore the implementation of the destructor.

Your implementation should otherwise code to precisely the same interface you’re already familiar with.

Presented below is a partial ThreadPool class declaration that includes all of the public entries and some of the private ones:

```cpp
class ThreadPool {
public:
    ThreadPool(size_t numWorkers);
    void schedule(const std::function<void(void)>& thunk);
    void wait();
private:
    std::list<std::function<void(void)>> thunks; // functions to execute
    size_t numActiveThunks = 0; // count of functions not fully executed yet
    std::vector<std::thread> workers;
    void worker(); // index isn’t needed for our version
    std::mutex m; // the one mutex you’re permitted to declare
    std::condition_variable_any workcv; // notify workers of work!
    std::condition_variable_any waitcv; // notify those waiting to wake!
};
```

We’ll lead you through the full implementation of this in the following parts.
a. [Points are allocated throughout parts b through e] If you inspect the partial `ThreadPool` declaration on the previous page, you’ll notice we’ve not declared any of the `condition_variable_any`s you’ll need. As you implement everything else over the course of the next few pages, add `condition_variable_any`s as needed to help coordinate thread communication with efficient blocking and zero busy waiting while avoiding needless thread wakeup. You should not declare anything other than `condition_variable_any`s.

It’ll likely be easier to add them as you work on parts b through e. You should add these new data members in the boxed region provided on the previous page.

b. [3 points] Implement the `ThreadPool` constructor, which should initialize all data members and spawn all `numWorkers` worker threads. Remember: thread objects can be constructed around methods, as with `thread(&ThreadPool::worker, this);

```cpp
ThreadPool::ThreadPool(size_t numWorkers) {
    for (size_t i = 0; i < numWorkers; i++) {
        workers.push_back(thread(&ThreadPool::worker, this));
    }
}
```

c. [3 points] Implement the thread-safe `schedule` method. Note that it’s not `schedule`’s responsibility to select which worker executes the supplied thunk; rather, it simply appends said thunk to the end of the thunk list and notifies the worker threads that another thunk needs to be executed.

```cpp
void ThreadPool::schedule(const function<void(void)>& thunk) {
    unique_lock ul(m);
    thunks.push_back(thunk);
    numActiveThunks++;
    if (thunks.size() == 1) workcv.notify_all();
}
```
d. [6 points] Implement the **private**, thread-safe **worker** method, which functions as the worker thread routine. **worker** should loop repeatedly and cooperate with other workers to executable threads in the order they are scheduled. You can assume that the worker loops forever and never breaks from its loop (i.e. don’t worry about a worker’s interaction with the destructor, since you’re not implementing that). Ensure that thunks are capable of executing in parallel with other thunks being executed by other workers.

```cpp
void ThreadPool::worker() {
    while (true) {
        unique_lock ul(m);
        workcv.wait(m, [this] { return thunks.size() > 0; });
        function<void(void)> thunk = thunks.front();
        thunks.pop_front();
        ul.unlock(); // important to unlock!
        thunk();
        ul.lock();
        numActiveThunks--;
        if (numActiveThunks == 0) waitcv.notify_all();
    }
}
```

e. [3 points] Implement the thread-safe **wait** method, which halts execution until all workers are idle and the thunk queue is empty.

```cpp
void ThreadPool::wait() {
    unique_lock<mutex> ul(m);
    waitcv.wait(m, [this] { return numActiveThunks == 0; });
}
```
Problem 5: Batched HTTP Requests [10 points]

Part 1 – Batched Proxy

[6 points] In your implementation of an HTTP Proxy, each HTTP request from a client was sent individually to the proxy. We could imagine an alternative where the requests are batched, meaning if a client needs to send HTTP requests to servers A, B and C, instead of forwarding 3 separate HTTP requests to the proxy for servers A, B and C, it could instead “batch” the requests together, sending one single HTTP request that contains information about each of the individual requests to A, B and C. The proxy could then send all the individual requests to A, B and C, and combine their responses into one single HTTP response that it sends back to the original client. There would be a specified format for the batched HTTP requests and responses to support this combined information. The benefit of this approach is reduced traffic between the clients and the proxy – instead of the client sending 3 requests to the proxy here, for instance, it now sends only 1.

Your task is to implement the handleBatchedRequest function, which is called by the proxy whenever it receives an incoming batched request. This function should do the following:

1. Create a batched HTTP response object.
2. Extract each of the individual requests included in the batched request and add a task to the provided thread pool for each to simultaneously connect to that server, send the request, and read the response, and add it to the batched HTTP response we are building up.
3. Once all these (and only these!) tasks finish, send the batched HTTP response back to the client. You should not assume that the pool isn’t being used for other tasks – in other words, handleBatchedRequest may be run in several threads that all share the same thread pool.

To help with this implementation, we provide new types HTTPBatchedRequest (that itself contains HTTPRequests) and HTTPBatchedResponse (that itself contains multiple responses). The relevant snippets of these definitions can be found below:

```cpp
class HTTPBatchRequest {
    // returns an array of HTTPRequests contained in this batch request
    vector<HTTPRequest> getBatchedRequests();
};

class HTTPRequest {
    const std::string& getMethod() const { return method; }
    const std::string& getURL() const { return url; }
    const std::string& getServer() const { return server; }
    unsigned short getPort() const { return port; }
    const std::string& getPath() const { return path; }
    const std::string& getProtocol() const { return protocol; }
};

class HTTPBatchedResponse {
    // read the HTTP response from the given stream and add its info
    // to this batched response. Assume this is thread safe.
    void ingestSingleResponse(std::istream& instream);
};
```
Also, you can write an `HTTPRequest` or `HTTPBatchedResponse` to a stream as follows:

```cpp
socketStream << requestOrResponse << flush;
```

Write your implementation of the `handleBatchedRequest` function below. It takes in the stream to respond back to the client, the already-read-in batched request, and the thread pool to use to schedule tasks to concurrently send all the requests. You **may not create any additional threads or thread pools** in your implementation.

```cpp
void handleBatchedRequest(iosockstream& clientSocketBuffer, HTTPBatchedRequest& batchedRequest, ThreadPool& pool) {
    HTTPBatchedResponse response;
    vector<HTTPRequest> requests = batchedRequest.getBatchedRequests();
    semaphore requestThreads(-requests.size() + 1);
    for (HTTPRequest& req : requests) {
        pool.schedule([&req, &response, &requestThreads]() {
            int s = createClientSocket(req.getServer(), req.getPort());
            sockbuf sb(s);
            iosockstream ss(sb);
            ss << req << flush;
            response.ingestSingleResponse(ss);
            requestThreads.signal();
        });
    }
    requestThreads.wait();
    clientSocketBuffer << response << flush;
}
```
Part 2 – Short Answer

Complete the following short answer questions about networking. Your responses to each question should be at most 50 or so words.

a. [2 points] When discussing HTTP proxies, we said that the proxy functions both as a client and a server. Explain what the client and server roles are, and why a proxy qualifies as both.

A client is a networked program that sends requests to servers for information and reads responses. A server is a networked program that listens for incoming requests and responds to them. A proxy is both because a proxy is listening for incoming requests from web browsers and other programs (server), then forwarding those requests to their corresponding servers and reading their responses (client), and then forwarding those responses back to the original client (server).

b. [2 points] With HTTP 1.0, there is an ability for a client to use a single connection for multiple requests. To do this, the client includes the "Connection: keep-alive" header in their request, which means they will use the same connection to send another request after the current one. The server should then keep the connection open rather than closing it after sending back the response. The client later indicates when the connection can be closed by omitting this header in its final request. Describe (no code necessary) how you might modify your existing Assignment 6 implementation to support this capability.

We could add a while loop around when we receive a request, and after we handle the request, check if the request contained the keep-alive header. If it doesn’t have one, break out of the loop, simulating the old proxy behavior. Otherwise, we go back to the top of the loop and read another request from the same stream.
Prototypes of Functions and Methods

// 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 pipe2(int fds[], int flags); // second arg is 0 or O_CLOEXEC, 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);
int execvp(const char *path, char *argv[]); // ignore retval
#define WNOHANG // constant
#define WUNTRACED // constant
int kill(pid_t pid, int sig); // ignore retval
int raise(int sig); // ignore retval
sighandler_t signal(int signum, sighandler_t handler); // ignore retval
#define SIG_IGN // sighandler_t constant used to ignore signal
#define SIG_DFL // sighandler_t constant restoring default signal
int sigemptyset(sigset_t *set); // ignore retval
int sigaddset(sigset_t *set, int sig); // ignore retval
int sigprocmask(int how, sigset_t *changes, sigset_t *existing); // ignore retval
int sigwait(sigset_t *blockset, int *delivered); // ignore retval

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

template <typename T>
class list { public:
    bool empty() const;
    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 Mutex> void wait(Mutex& m);
    template <typename Mutex, typename Pred> void wait(Mutex& m, Pred pred);
    void notify_one();
    void notify_all();
};

int createClientSocket(const string& hostname, unsigned short port);
int createServerSocket(unsigned short port);

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