1 Objective

This exercise will apply concepts of multithreading and synchronization. You are to extend the
website codebase provided for you. Before changing anything, please understand the current source
code, and ask questions on piazza (@piazza.com/class#spring2013/csci3907os). Again, in
order to allocate memory and deallocate memory, you must use malloc and free from stdlib.h. To
create threads you must use pthread_create and to wait on, clean up and retrieve information from a
terminated thread you must use pthread_join from pthread.h.

2 Thread Management via POSIX Threads (pthreads)

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void (*start_routine) (void *), void *arg);
```

pthread_create starts a new thread in the calling process and returns the new thread identifier. The
new thread may terminate either by calling pthread_exit or by returning from the start_routine.

```c
void pthread_exit(void *retval);
```

pthread_exit terminates a calling thread safely and returns a value via retval which is available to
another thread in the same process that calls pthread_join.

```c
int pthread_join(pthread_t thread, void **retval);
```

pthread_join waits for the thread specified by thread to terminate. If that thread has already
terminated, pthread_join returns immediately.

For detailed information, refer to the man pages on pthread_create, pthread_exit, and
pthread_join.

3 Assignment

You will add a thread pool to the current webpage. The pool of threads will be “workers” and a single
“master” should be in an “accept loop”, that accepts (creates) connections, and adds them to a
data-structure that is read from by the worker threads. When a worker receives a connection, it will
process the request. After servicing the request, that worker will go back to attempting to get additional requests from the data-structure.

You will implement two data-structure and synchronization mechanisms for communication between the master and worker threads. First, you will use mutexes and condition variables so that threads workers will block waiting for data, and the master will wake them up to service the request. Second, you will implement a lock-free stack data-structure using the compare and swap atomic instruction to pass data between the threads.

4 Thread Pool with Mutexes and Condition Variables

Below is provided a brief overview of pthread functions for Mutexes and Condition Variables. However, do not consider this overview to contain comprehensive information regarding these functions. You are expected to consult man pages or online resources on each function for detailed information.

4.1 Mutex

```c
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
```

pthread_mutex_init creates the mutex pointed to by `mutex` with the attributes pointed to by `attr`. If `attr` is NULL, the mutex is created with default attributes. A mutex must be initialized before it is used. Attempting to initialize an already initialized mutex results in undefined behavior. Returns 0 on success; otherwise, returns an error number.

```c
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

pthread_mutex_destroy deletes the mutex pointed to by `mutex`. Referencing an uninitialized or deleted mutex is undefined. Returns 0 on success; otherwise, returns an error number.

```c
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

pthread_mutex_lock locks the mutex pointed to by `mutex`. If `mutex` is already locked, the calling thread will block until `mutex` is available. Returns 0 on success; otherwise, returns an error number.

```c
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

pthread_mutex_unlock releases the mutex pointed to by `mutex`. Returns 0 on success; otherwise, returns an error number.

4.2 Condition Variables

```c
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
```

pthread_cond_init creates the condition variable pointed to by `cond` with the attributes pointed to by `attr`. If `attr` is NULL, the condition variable is created with default attributes. A condition variable must be initialized before it is used. Attempting to initialize an already initialized condition variable results in undefined behavior. Returns 0 on success; otherwise, returns an error number.
int pthread_cond_destroy(pthread_cond_t *cond)

pthread_cond_destroy deletes the condition variable pointed to by cond. Referencing an uninitialized or deleted condition variable is undefined. Returns 0 on success; otherwise, returns an error number.

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)

pthread_cond_wait blocks on a condition variable. Must be called on a locked mutex or the behavior is undefined. Returns 0 on success; otherwise, returns an error number.

int pthread_cond_signal(pthread_cond_t *cond)

pthread_cond_signal unblocks at least one thread blocked on the condition variable pointed to by cond. If more than one thread is blocked on the condition variable, the scheduling policy determines which threads are unblocked. Returns 0 on success; otherwise, returns an error number.

4.3 Thread Pool with Condition Variables

When the option is set to a value of one (1), the server should “use a thread pool with condition variables”. You are to implement the code to support this behavior. This thread pool model must use blocking operations.

A limited number of connections (bounded by the maximum size of the queue) can be open at a given time in this mode. You must use a ring buffer, or linked list for your queue. Make the defined variable for the ring buffer size, BUFFER_LENGTH, equal to 256. Use the combination of pthread_mutex_t and pthread_cond_t (i.e. mutexes and condition variables) to enable threads to block when:

- the master attempts to insert into the ring buffer and it is full.
- a worker thread attempts to dequeue, and the queue is empty.

Condition variables enable you to block when there are conditions such as "the queue is full" or "the queue is empty", while enabling other threads to access the shared data-structure (as opposed to blocking while holding a mutex, preventing other threads from accessing the shared resource). In using mutexes and condition variables, you will essentially be using monitors. It is assumed that you will investigate these concepts on your own.

5 Thread Pool via Lock-Free Data-structure and Atomic Instructions

When the option is set to a value of two (2), the server should "use atomic instructions to implement a task queue". You are to implement the code to support this behavior.

In this mode, we wish to eliminate the overhead of creating a new thread every time the server receives a request for HTTP content. To do so, you will have a single main thread that accepts on the socket and thus receives new HTTP requests. You will have to create MAX_CONCURRENCY number of worker threads
when the server initializes, and when a client requests HTTP service, the main thread will pass the client request on to a single worker thread. That worker thread will process the request, generate its content, and close the connection to the client. The difficult part of this assignment is sending the client request (including which file descriptor to use for the communication with the client) from the master thread to a worker thread. This is a typical instance where synchronization is required as you have concurrent or parallel access to shared memory.

You will use lock-free synchronization to implement this communication. You will need to implement a stack using a singly-linked list (e.g. Programming Exercise 1). Each node in the stack holds the client information the worker requires to process the HTTP request (e.g. fd), and a pointer to the next node (or NULL if there are no more nodes). Each node is of type `struct request`. You will have a single shared global pointer to the first node in the list called `requests`. You will use an atomic instruction to modify this variable to implement the following functions:

```c
struct request *get_request(void);
void put_request(struct request *r);
```

The master calls `put_request` every time a new client request arrives. Each worker has an infinite loop that calls `get_request`, and then processes the request. `get_request` itself will only return when a request is available. This means that when a request is not available, the thread will be spinning in `get_request`.

The atomic instruction you will use is the compare and swap instruction (cas). See cas.h for the function that can be called to use this instruction and for documentation on how cas works.

### 6 Testing

Your server will be tested with `httperf` which can be downloaded at http://code.google.com/p/httperf/downloads/list. We highly suggest that you test with this program as it will stress test the server at a level impossible to recreate with a browser or wget. An example of using `httperf` follows:

```
httperf --port=8080 --server=localhost --num-conns=10000 --burst-len=100
```

Note: you will need to run `httperf` on the same machine where the server is running, or change the `--server=x` parameter as appropriate.

### 7 Submission

You will create a zip archive containing **your entire simple webserver directory** (please run a `make clean` before zipping the directory to remove any build products) and you will submit this zip file to Blackboard in the folder for Programming Assignment 1 by the appointed time indicated at the top of this document (Note that you have an extended two weeks to complete this assignment, but DO NOT WAIT to complete the assignment. More time does not imply procrastinate). The zip file will be named with the following naming convention:
csci_3907_ws_<your GWNet Id>.zip

Where <your GWNet Id> is the user name that you use to log into Blackboard.

8 Evaluation

You are obligated to complete this exercise on your own by the Code of Academic Integrity (http://www.gwu.edu/~ntegrity/code.html).

The conciseness and simplicity of your code will affect your evaluation. Find a balance between the number of lines of code and the simplicity of each line.

Your code will be evaluated in terms of the comprehensiveness and descriptiveness of comments. A block comment preceding the function definitions you implement should describe the function including any special information about the input parameters and output result. Line comments should clarify variable usage and complex logic. Do not assume that either you or others will understand code that you have written.