Myhttpd is a multithreaded web server. It binds to a given port on the given address and waits for incoming HTTP/1.0 requests. When it receives a new request from the client it responds to it using HTTP/1.0 protocol. Being a multithreaded server it has different threads to perform different jobs. Firstly it has 1 thread which listens to clients continuously and adds a new request to ready queue. 2nd thread schedules the ready queue and assigns the request to N execution threads.- number of execution threads changes as per user’s input.

Person# 50096836

swapnesh@buffalo.edu

Multi-threaded Web Server

Swapnesh Gandhi

1. Data Structures used:

For the ready queue linked list implementation of a queue is used. Listening and scheduling threads share this queue data structure and both write to it. Therefore, writing and reading to queue is identified as “Critical Section” and are protected by binary semaphores.

The structure of the queue is-

1. Whole request
2. Command (GET, HEAD etc.)
3. File or Directory name.
4. Size of the file
5. Incoming timestamp

For SJF scheduling the request need to be sorted w.r.t. size of the file, this is done by sorting the whole linked list.

To assign the requests to the execution threads first scheduling thread must know whether the thread is free or busy. For this purpose, I created an integer array which gets updated, the scheduler checks this array and finds the free thread, if any.

To wake the threads I have used an array of conditional variables one conditional variable each for the execution thread.

The request to be executed is written to an array of structure by the scheduling thread, thus one structure per each thread ensures that each thread can read from its own structure without interfering with other threads.

All of the shared data structures like queue, array of structure, array for thread status are protected by mutex locks.

1. Context switches implementation:

I find the relation between listening and scheduling thread similar to producer consumer problem. When listening thread adds an item to ready queue it signals the scheduler which may be waiting. Scheduler then schedules the ready queue and assigns the request to threads.

If scheduler is woken up and while it was still doing some work listening thread may signal the scheduler. But it will not receive the signal as it was not in wait condition. This situation may lead to deadlock, to avoid it scheduler checks whether the queue is empty and then it will wait if it finds the queue empty.

The relation between scheduler and execution threads is similar to sleeping barber problem. Where the scheduler acts like the customer and execution threads act like barbers. The execution threads initially go to sleep till scheduler wakes one of them using signal on conditional variables. To make synchronization simpler scheduler in my implementation only wakes one of the execution threads at a time. The scheduler finds the free thread by keeping track of thread status array, which is updated by both scheduler and threads. The threads when woken up take request related data from their own structure, which must be written by scheduler before waking them up, and use serve the request and log the request if specified by the user.

Execution threads

Listening thread

Scheduler thread

Race conditions handling:

As there are multiple shared data structures present multiple situation where race condition is possible exist. To deal with these situations, I have either tried to avoid them using less complex approaches or used mutexes which protect the critical region.

1. To avoid having race condition w.r.t queue, I have always manipulated the data in queue only while using mutexes.
2. The temporary structure which is used by scheduler and execution thread is protected by mutexes as well.
3. The thread status array which updates busy or free status of thread is protected by mutex.
4. By creating a simple flow of data in queue I have tried to reduce the amount of interaction and dependence of these threads with each other, as shown in the diagram above.

Advantages:

1. Separate data structure for execution threads and ready queue ensures situations with race conditions are minimal.
2. All the potential race conditions and deadlock situations are avoided by careful use of mutexes and conditional variables.
3. As no thread in server is busy waiting, the performance of the server is good.

Disadvantage:

1. The separate data structure for thread will mean that the memory required will be more as well this increasing space complexity.

References:

1. For socket programming I referred to <http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#twotypes> from time to time.
2. For pthread implementation I referred to <https://computing.llnl.gov/tutorials/pthreads>
3. For miscellaneous parts of projetcs like finding file size, modified time etc. I referred to www.stackoverflow.com
4. For implementation of queue using linked list referred to book - Data structures through C by G.S. Baluja
5. Referred to recitation and lecture slides.