Design Document: Project1:-Multithreaded Web Server

By Team# G15

a. Team Members

1. Kalpesh Kagresha (kkagresh@buffalo.edu)
2. Laxman Bharadwaj Kandlakunta (laxmanbh@buffalo.edu)
3. Xuhui Jin (xuhuijin@buffalo.edu)

b. The **Responsibilities** of each team members are as follows

1. Laxman Bharadwaj Kandlakunta: HTTP Request and Response, Listener Thread, Bugs Handling, Logging
2. Xuhui Jin: Scheduler Thread
3. Kalpesh Kagresha : Multithreading, Bugs Handling

Main Components

1. Server: Initialization and Parameter Configuration Step
2. Listener: Accept Incoming Requests
3. Scheduler: Schedules and Assign tasks to thread pool
4. Multithread / Thread pool: Executes the request (i.e. Send the requested file of list of files over the network).

Data Structures:

The following data structures are used in the ThreadPool (Execution Threads) implementation

1. struct job_details

```
{
    int client_socket; // It stores the client socket file descriptor
    long filesize;// Size of File requested(useful for scheduling algorithm such as SJF)
   char *request;// GET or HEAD Request string
};
```

The above structure gives details about the job requested by client. It will be prepared once we store the job data in ready queue.

2. struct th_job

```
{
   struct th_job *link; // This link stores the pointer to next job in thread queue
   struct job_details *param; //Job Details/Parameters
};
```

The above structure stores the link between the threads as well as job details.

3. struct sched_tp_queue // queue for scheduler and threads
{
   struct job_request *front;// Front pointer of Job queue
   struct job_request *rear;// Rear pointer of Job queue
   int NoOfJobs; //Total no. of jobs in queue
   sem_t *sem_queue; //Semaphore Pointer to queue for mutual exclusion between execution threads

};
The above structure gives details (such as front, rear, #jobs and semaphore) about the thread queue. Jobs enter in this queue through scheduler. Note that this queue is different from the ready queue which is flooded by the listener thread.

4. struct threadpool
{
   pthread_t *thread_ptr; //Pointer to threads
   int NoOfThreads; //Total No. of threads specified by the numThreads or n execution threads
};

This structure gives details about the n execution threads.

The below two data structure are used by listener and scheduler thread

5. struct job_request
{
   int client_socket;//client socket
   long filesize; //filesize
   struct job_request *link; //pointer to the next job
   char request[];// contains the entire request
   char clientipaddress[];//IP address of the remote client for logging purpose
   char reqrecvtime[];//the time at which this request arrives for logging purpose
};

6. struct list_sched_job_queue //the queue for listener and scheduler i.e. ready queue
{
   struct job_request *front;
   struct job_request *rear;
};

Listener Thread

Once the server starts, it continuously listens on a given port. At this point of time, we initialize the following variables/ call methods

1. scheduler_init( ): FCFS or SJF
2. threadpool_init():

We then create listener thread using the following statement

pthread_create(&listener_thread, &attr, listener_thread_func, (void*)&server_socket);

void* listener_thread_func(void* arg)

The main objective of this function is to accept request from client and store them in ready queue. Data structure such as **struct job_request** are used by the listener thread to store incoming request in **struct list_sched_job_queue.**

Scheduler Thread

Created by using following

pthread_create(&scheduler_thread,&attr,scheduler_thread_func,NULL);

This thread is created along with the listener thread. But the main difference between them is that scheduler begins scheduling tasks only after't' seconds. Once active, it retrieves job from the ready queue and assign to thread pool. Function such as "**retrieve_job()**" in scheduler.c file picks job from the ready queue depending the scheduling mode specified. The data structure used by listener thread are also utilized by this thread.

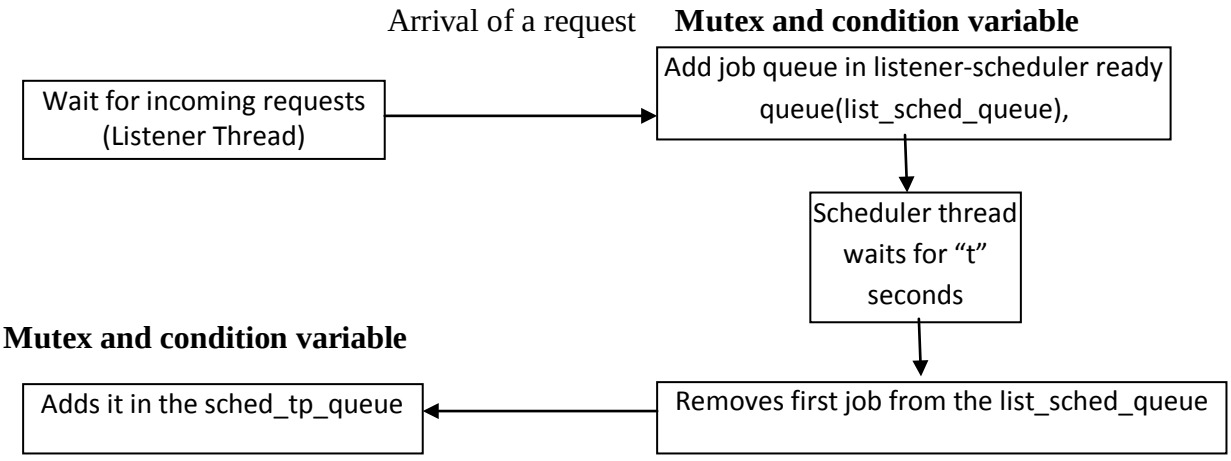

Fig 1.2 Listener- Scheduler Thread Process

Implemetation of Execution Threads/ Thread Pool

1. **struct sched_tp_queue* threadpool_init(int noofThreads, int flag)**
   This function creates pointer to threadpool and initializes #threads. N execution threads are assigned request_assign method. This function is responsible for handling the request received by the client.

2. **void *request_assign(void *arg)**
   This function executes in a infinite loop until the thread pool is alive. Each thread checks if there are any jobs available in thread queue. If there aren't any, then they wait for incoming jobs examining the following line

   sem_wait(tp->job_queue->sem_queue)
   The sem_wait function locks the semaphore referenced by sem_queue by performing lock operation on that semaphore. The state of the semaphore remains lock until the sem_post() is executed and returns successfully.

   In case if the sem_wait is unsuccessful, then current thread pops the job from front of the queue and handles the request for file/ directory. Mutual Exclusion is preserved using mutex lock and unlock operation on the queue.

   pthread_mutex_lock(&mutex);
   //removes the front job in from the queue and calls the request handle method.
   pthread_mutex_unlock(&mutex);

   The above operation picks the job from front of the queue and removes it. By locking mutex variable, only single thread will be allowed to access the queue. This is how the race conditions are avoided.

3. **struct sched_tp_queue* threadpool_assign_task(int cs,char buffer[],long sz,char cip[],char reqrecv[])**

   This function creates a job to be inserted in thread queue. It is invoked by the scheduler. In order to ensure mutual exclusion, we use following lines

   pthread_mutex_lock(&mutex)
   //adding job to the sched_tp_queue
   sem_post(tp->job_queue->sem_queue);
   pthread_mutex_unlock(&mutex)

   The first line locks the mutex variable which blocks all calling threads. It implies, at a time only one thread is allowed to add a job to queue, therefore avoiding race conditions. Most importantly, sem_post() is invoked which signals thread waiting for the jobs to be added in the queue.

4. **void threadpool_finish(struct threadpool* tp)**
   This method releases all threads waiting for the semaphore with no job, destroys semaphore, waits for all threads to finish and free the memory allocated for the above data structures.

5. **void request_handle(struct job_details *parameter)**
   It serves the HTTP GET/ HEAD request and sends the appropriate message/ file /directory lists as response to the request.

Note: The above data structures and functions are declared in ThreadPool_utils.h header file

Block diagram for Thread Pool Implementation

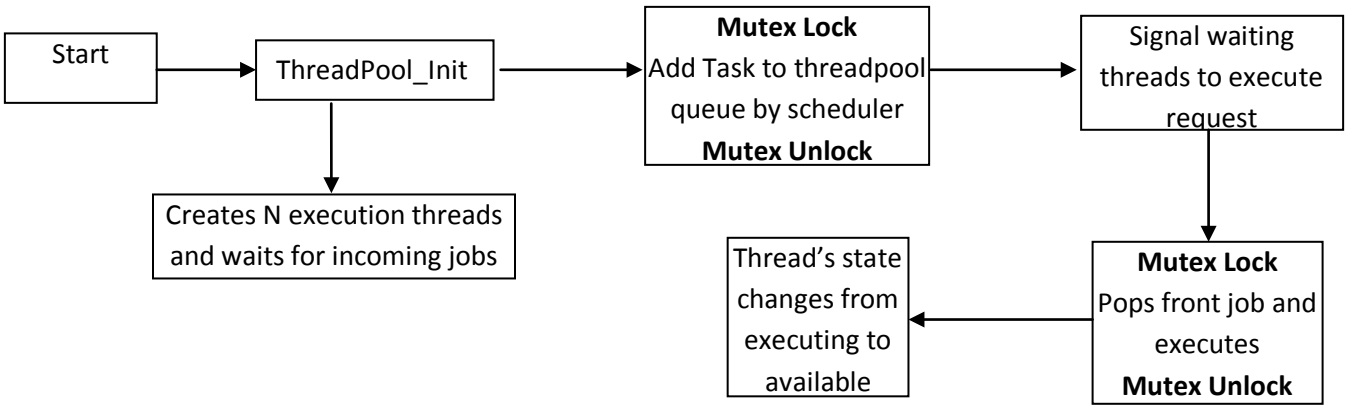

Fig 1.1 Thread Pool Implementation Steps

d. The listener thread waits for incoming connections. It adds a job in the list_sched_queue and signals the scheduler and repeats this process. Scheduler picks up a job (after waking up) and inserts it in the sched_tp_queue, repeats this process. This signals one of the already created execution threads .This execution thread handles the request and waits for new job (all others execution threads are alike). Thus the CPU performs context switching between listener and scheduler thread for listening and saving requests in ready queue. The similar switching happens between thread in the threadpool to handle the request.

e. As explained in d, the queues list_sched_queue and sched_tp_queue can be accessed by more than one thread. We have used three semaphores to control access to the list_sched_queue: a mutex ,empty and full. Mutex and sem_queue to control access to sched_tp_queue. Scheduler acquires a lock on mutex and adds a job in the queue. The threads in the threadpool are waiting for sem_queue. The scheduler, after assigning a job signals a thread to pick up the job. This thread handles the request by acquiring a mutex, thus another thread cannot access the queue simultaneously.

f. The main advantage of this design is that we are using two separate queues. Thus we avoid the situation of execution threads/scheduler thread/listener thread waiting to access the queue. In our approach, listener can add new requests while the execution threads are picking up already added requests. Another advantage is that, for SJF, the listener adds up the jobs according to the file size putting the file with shortest job at the front. So, the scheduler can always pick up the shortest job. These queues are implemented using linked lists. This (dynamic memory allocation) is efficient. The scheduler thread busy waits for‘t’ seconds. This is a disadvantage. The functionality to stop this server with a command was not implemented. It has to be stopped using CTRL+C command.

References

- **POSIX Threads Programming**
  http://pubs.opengroup.org/onlinepubs/7908799/xsh/sem_post.html
- **Linux Tutorial: POSIX Threads**
  http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- **sem_wait**
  http://pubs.opengroup.org/onlinepubs/7908799/xsh/sem_wait.html
- **Pthread_mutex_lock**
  http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_lock.html
- **sem_post**
  http://pubs.opengroup.org/onlinepubs/7908799/xsh/sem_post.html
- **pthread:mutexes**
  https://computing.llnl.gov/tutorials/pthreads/#Mutexes
- **socket programming**
  http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html
- **string functions**
  http://faculty.edcc.edu/paul.bladek/c_string_functions.htm

  Also, we have consulted stackoverflow.com for fixing the errors.