NAME: VIVEKANANDH VEL RATHINAM

UB ID: 50098075

E-MAIL: VVELRATH@BUFFALO.EDU

PROJECT I – DESIGN DOCUMENT

DATA STRUCTURES USED FOR IMPLEMENTING QUEUING, SCHEDULING, MULTITHREADING AND SYNCHRONIZATION COMPONENTS OF THE SERVER

1) A structure encapsulating the following information of a single request.
   a) Client Socket ID
   b) Requested File name
   c) Size of the requested file
   d) Arrival time of the request
   e) Time when the request is served
   f) Status of the served request
   g) First line of the request
   h) Link to another request structure

```
struct queue
{
int acceptfd;
char filename[1024];
long long int filesize;
char *arrivaltime;
char *exectime;
char *firstline;
int status;
struct queue *link;
};
```

2) A singly linked list is implemented which links all the requests in the form of a queue. Each request structure has a link to another request structure. There are two ways on how the linked list is created
   a) FCFS – This linked list is created based on the arrival time of the request

```
if(front==NULL)
{
rear=newnode;
front=rear;
}
else
{
rear->link=newnode;
rear=newnode;
}
```

b) SJF – This linked list is created based on the requested file size. The request which has the smallest file size gets the most priority. The linked list is ordered according to increasing file sizes.

```
if(front==NULL)
{
        rear=newnode;
        front=rear;
}
else
{
        if(newnode->filesize<=front->filesize)
        {
                newnode->link=front;
                front=newnode;
        }
        else if(newnode->filesize>rear->filesize)
        {
                rear->link=newnode;
                rear=newnode;
        }
        else
        {
                temp=front;
                while(temp!=NULL)
                {
                if(newnode->filesize<=temp->filesize)
                {
                temp_previous->link=newnode;
                newnode->link=temp;
                break;
                }
                temp_previous = temp;
                temp=temp->link;
                }
        }
}
```

3) POSIX Threads

Three threads are created in this project

a) Listening thread – This thread takes care of the queuing of the requests
b) Scheduling thread – This thread takes out one request at a time from the queue and makes it available for the worker threads.

c) Worker threads – These are the threads which serve the requests popped out by the scheduling thread

```
pthread_t listening_thread;
pthread_create(&listening_thread,NULL,&listener,NULL);
```

4) Mutual Exclusion Locks – Locks are placed on the critical sections of the code to avoid race conditions.

```
pthread_mutex_t qmutex = PTHREAD_MUTEX_INITIALIZER;
```

5) Condition variables – These are used by the threads to perform the waiting and signaling operation on requests.

```
pthread_cond_t  cond_schedule  = PTHREAD_COND_INITIALIZER;
```

IMPLEMENTATION OF CONTEXT SWITCHES BETWEEN THREADS

Context switches between threads are implemented using condition variables. When the server is started, the scheduler thread and all the worker threads wait on a condition.

There are two condition variables used in this project

a) cond_schedule – This variable is used by the listening thread and the scheduling thread by signaling and waiting respectively. The listening thread signals this condition to the scheduling thread when it inserts a request into the queue. The scheduling thread waits on this condition to make sure that the queue is not empty.

Scheduling thread - pthread_cond_wait(&cond_schedule);

Listening thread - pthread_cond_signal(&cond_schedule);

b) cond_serve – This variable is used by the scheduling thread and the worker threads by signaling and waiting respectively. The scheduling thread pops out one request from the queue and signals this condition. All the worker threads wait on this condition after serving each request. The condition signaled by the scheduling thread is received by one of the worker threads which are waiting on this condition.

Worker threads - pthread_cond_wait(&cond_serve);

Scheduling thread - pthread_cond_signal(&cond_serve);

HOW ARE RACE CONDITIONS AVOIDED IN THIS PROJECT

Race conditions are avoided using mutual exclusion locks. The queue used in this project has to be accessed by the listening and scheduling threads. When the listening thread is inserting a request into the queue, the scheduling thread should not pop a request from the queue. These race conditions are avoided by placing mutual exclusion locks in the critical sections in each of the threads. The mutex lock is unlocked after the execution of the critical section has ended by one of the threads.

ADVANTAGES AND DISADVANTAGES IN THE PROJECT DESIGN

Advantages:

a) For Shortest Job First algorithm, the request is inserted in the appropriate place in the queue to make sure the queue is sorted. This minimizes the sorting time when compared to sorting the whole queue each time where all the request nodes are moved. In this project, only one node has to be inserted and there are no change of links.

Disadvantages:

a) All the project code is written in one single file which makes it difficult to understand the project design.

ONLINE RESOURCES USED FOR PREPARING THE PROJECT

1) http://www.cplusplus.com/forum/unices/16005/
2) http://beej.us/guide/bgnet/output/html/multipage/index.html
3) https://gist.github.com/kyunghoj/6993541