SYSTEMS PROGRAMMING
CSE 344

#final_project

jwan hussein
151044078

data structure :

linked list :

```
struct NODE{
        int data ;
        struct NODE *next ;
};

struct mylinkedlist{
        int size ;
        struct NODE *head;
};
struct mylinkedlist create_LLIST();      /* create a linked list */
struct NODE* add_NODE(struct mylinkedlist *l , int data);  /* adding a
                                                node to the end of the list */
void free_list(struct mylinkedlist *l);            /* free the list */
int linkedlist_index_of(const struct mylinkedlist l , int data);     /* checkd if
the data is in the list . If found then return the node number if not return -1 ;
void add_NODE_front(struct mylinkedlist *l , int data);     /* adding a node
                                                        to the front of the list */
```

QUEUE :

which is implemented using the linked list above .

```
struct myqueue{
        int size ;
        struct mylinkedlist queue;
        struct NODE *front ;
        struct NODE *rear ;
};
struct myqueue create_QUEUE();    /* create a queue */
void enqueue(struct myqueue* q , int data);  /* add element to the queue */
int dequeue(struct myqueue* q ); /* remove element from the queue */
int is_empty_queue(const struct myqueue q);  /* if queue is empty return 1 ,
                                                        if not return 0 */
void free_queue(struct myqueue q); /* free the queue */
```

the queue is used only when calculating the path in the graph .

graph :

implemented using the linked list .

```
struct mygraph{
        int V;
        int E;
        int cap;
        struct mylinkedlist* adjlist ;
        int *visited ;
};
/* create a graph  */
struct mygraph create_graph();
/* add an edge to the graph */
void add_edge(struct mygraph *g , int src , int dest);
/* return the index of a vertex in the adjlist , -1 if not in the list*/
int index_of(const struct mygraph g  , int vertex);
/* free the graph */
void free_graph(struct mygraph *g);
/* return the path from src to dest */
struct mylinkedlist bfs(struct mygraph *g , int src , int dest);
/* check if there is a path between src and dest */
int is_connected(struct mygraph *g , int src , int dest ,
                                                int* dist , int* pred);
```

CACHE :

the cache or the database that going to live in the memory while the server is in execution .

```
struct CACHE{
        int S;
        int D;
        struct mylinkedlist l;
        struct CACHE *next ;
};

struct mycache{
        int size ;
        struct CACHE *head ;
};
/* create a new cache  */
struct mycache create_cache();
/* add the path inside the list into the cache */
void add_path(struct mycache *c , struct mylinkedlist l );
/* free the cache */
void free_cache(struct mycache *c);
/* search the cache for the path return 1 if the path was found and 0 otherwise  */
struct mylinkedlist is_path_in_cache(const struct mycache c,int S,int D);
```

linked list to keep information about the threads

```
struct thread_arg{
        int id ;                                /* thread id */
        int fd ;                                /* the file descriptor returned from call to
                                                        accept(); */
        int logfd ;                             /* log file descriptor */
        int state ;                              /* if -1 thread is available , else is occupied */
        pthread_cond_t cond_var ;
        struct thread_arg *next;
};
struct info{
        int size;
        struct thread_arg* head ;
        struct thread_arg** pointers ;          /* each pointer pointing to a node in the list */
};
```

this structure is used to keep information about the threads pool .

the server

the main thread :

the server starts in daemon mode , which mean it is being executed in the background with to controlling terminal .
The server using named semaphore to prevent double instantiation , when the server launched it initialize the named semaphore and and does not destroy it until the server is terminated by SIGINT signal .
After that the server loads the graph form the input file to the memory , create the thread pool , create the cache .
After every thing is ready the server start to create a socket first call socket() then call bind() then call listen() , if any error was occurred in those steps then a warning msg will be printed to the log file then in infinite loop waits for incoming requests using the accept() function .

pseudo code for the infinite loop of the server

```
do{
        lock the server_mutex
        while (number_of_occupied_thread >= 75% of the thread pool size  && min != max){
                help = 1  ;    /* help is a signal that the resizer thread waits for */
                signal the resizer thread
                cond_wait(server_cond, server_mutex )
        }
        if( number_of_occupied_thread == max ){
                /* the resizer cannot help in this situation so the main thread have to wait until a thread
                become available */
                cond_wait(server_cond, server_mutex);
        }
        unlock the server_mutex
        accept() ;
        lock the server_mutex
        search for an available thread to handle the request
        number_of_occupied_thread  ++ ;
        change the thread state to 1 ;
        broad cast thread cond var ;
        unlock the server_mutex

}while(!interrupt)
```

thread pool :

pseudo code

```
while(!interrupt){
        lock the server_mutex
        while(thread state == -1 ){
                /* means that no request was delegated to the thead  */
                cond wait(thread cond var , server_mutex )
        }
        /* the thread got the request */
        unlock the server_mutex
        read the request body
        ////////////////////////////////////////////////////////////////////////////////////////////////////
        pthread_mutex_lock( &cache_mutex );
        while((AW+WW) > 0 && !interupt){
                WR++;
                pthread_cond_wait(&oktoread,&cache_mutex);
                WR--;
        }
        AR++;
        pthread_mutex_unlock( &cache_mutex );

        check the cache for the path                    /*   a thread in this step is considered as a reader */

        pthread_mutex_lock( &cache_mutex );
        AR--;
        if(AR == 0 && WW > 0){
                pthread_cond_broadcast(&oktowrite);
        }else if(WR > 0 && WW == 0){
                pthread_cond_broadcast(&oktoread);
        }
        pthread_mutex_unlock( &cache_mutex );
        ////////////////////////////////////////////////////////////////////////////////////////////////////
        if (the thread find the path in the cache)
                 then it responds to client and wait for another request
        else {
                in this case the thread is calculating the path then responds to client ,
        if the path was available
        ////////////////// here the thread is considered as a writer //////////////////////
        pthread_mutex_lock(&cache_mutex);
        while((AW+AR) > 0 && !interupt){
                WW++;
                pthread_cond_wait(&oktowrite,&cache_mutex);
                WW--;
        }
        AW++;
        pthread_mutex_unlock(&cache_mutex);

        adding the path into the cache ;

        pthread_mutex_lock(&cache_mutex);
        AW--;
        if(WW>0){
                pthread_cond_broadcast(&oktowrite);
        }else if(WR > 0){
                pthread_cond_broadcast(&oktoread);
        }
        pthread_mutex_unlock(&cache_mutex);
        ////////////////////////////////////////////////////////////////////////////////////////////////////
        }

        lock the server_mutex
        changing the thread state to be available
        close the connection
        number_of_occupied_thread-- ;
        broadcast the  server_cond ;
        unlock the server_mutex


}
```

resizer_thread :

```
pseudo code
while(!interrupt){
        lock the server_mutex
        while (help == 0){
                cond_wait(resizer_cond , server_mutex)
        }
        if(min == max){
                /* the case where there is no more space for extending the pool */
                help == 0 ;
                broadcast server_cond
                unlock the server mutex
        }
        else {
                min += min*0.25;
                extending the pool by 25%
                help = 0 ;

                broadcast server_cond
                unlock the server mutex
        }


}
```

client

the client process calls socket() and then try to connect as soon as the connection was established the client requests a path from the server and then wait for the response from the server