VSCODE_PRINT_SCRIPT_TAGS

Selected files

7 printable files

```
Assignments\Assignment4\client_tst.c
Assignments\Assignment4\hash.c
Assignments\Assignment4\makefile
Assignments\Assignment4\hash.h
Assignments\Assignment4\react_server.c
Assignments\Assignment4\st_reactor.c
Assignments\Assignment4\st_reactor.h
```

Assignments\Assignment4\client_tst.c

```c
/*
** client.c -- a stream socket client demo
*/

#include <stdio.h>
#include <poll.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#include <arpa/inet.h>

#define PORT "9034" // the port client will be connecting to

#define MAXDATASIZE 100 // max number of bytes we can get at once

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct addrinfo hints, *servinfo, *p;
    int rv;
    char s[INET6_ADDRSTRLEN];

    if (argc != 2) {
        fprintf(stderr,"usage: client hostname\n");
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and connect to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                             p->ai_protocol)) == -1) {
            perror("client: socket");
            continue;
        }

        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            perror("client: connect");
            close(sockfd);
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "client: failed to connect\n");
        return 2;
    }

    inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
              s, sizeof s);
    printf("client: connecting to %s\n", s);

    freeaddrinfo(servinfo); // all done with this structure
    struct pollfd pfds[2];
    pfds[0].fd  = 0; // Stdin
    pfds[0].events = POLLIN;
    pfds[1].fd = sockfd;
    pfds[1].events = POLLIN;
    for (;;) {
        poll ( pfds, 2, -1);
        if (pfds[0].revents & POLLIN) {
            numbytes = read (0, buf, MAXDATASIZE-1);
            send ( sockfd, buf, numbytes,0);
        } else {
            numbytes = recv (sockfd, buf, MAXDATASIZE-1,0);
            buf[numbytes]=0;
            printf ("server:%s\n",buf);
        }
    }

    close(sockfd);

    return 0;
}

```

Assignments\Assignment4\hash.c

```c
//
// Created by alon on 5/18/23.
//
/*
 * base code copied from https://www.geeksforgeeks.org/implementation-of-hash-table-in-c-using-separate-chaining/
 * and changed to fit for us
 */
#include "hash.h"

// like constructor
void setNode(struct node* node, int key , handler_t value)
{
    node->key = key;
    node->value = value;
    node->next = NULL;
};


// like constructor
void initializeHashMap(struct hashMap* mp)
{

    // Default capacity in this case
    mp->capacity = 100;
    mp->numOfElements = 0;

    // array of size = 1
    mp->arr = (struct node**)malloc(sizeof(struct node*)
                                    * mp->capacity);

}

void increase_HashMap(struct hashMap* mp,int new_capacity){
    struct node ** new_arr = realloc(mp->arr,sizeof(struct node*) * new_capacity);
    free(mp->arr);
    mp->arr = new_arr;
}

int hashFunction(struct hashMap* mp, int key)
{
    int bucketIndex;
    int sum = 0, factor = 31;
    for (int i = 0; i < key; i++) {

        sum = ((sum % mp->capacity)
               + (key * factor) % mp->capacity)
              % mp->capacity;

        // factor = factor * prime
        // number....(prime
        // number) ^ x
        factor = ((factor % __INT16_MAX__)
                  * (31 % __INT16_MAX__))
                 % __INT16_MAX__;
    }

    bucketIndex = sum;
    return bucketIndex;
}

void insert(struct hashMap* mp, int key, handler_t value)
{

    // Getting bucket index for the given
    // key - value pair
    int bucketIndex = hashFunction(mp, key);
    // Creating a new node
    struct node* newNode = (struct node*)malloc(sizeof(struct node));



    // Setting value of node
    setNode(newNode, key, value);

    // Bucket index is empty....no collision
    if (mp->arr[bucketIndex] == NULL) {
        mp->arr[bucketIndex] = newNode;
    }

        // Collision
    else {

        // Adding newNode at the head of
        // linked list which is present
        // at bucket index....insertion at
        // head in linked list
        newNode->next = mp->arr[bucketIndex];
        mp->arr[bucketIndex] = newNode;
    }
    mp->numOfElements++;
}

void delete (struct hashMap* mp, int key)
{

    // Getting bucket index for the
    // given key
    int bucketIndex = hashFunction(mp, key);

    struct node* prevNode = NULL;

    // Points to the head of
    // linked list present at
    // bucket index
    struct node* currNode = mp->arr[bucketIndex];

    while (currNode != NULL) {

        // Key is matched at delete this
        // node from linked list
        if ( key == currNode->key ) {

            // Head node
            // deletion
            if (currNode == mp->arr[bucketIndex]) {
                mp->arr[bucketIndex] = currNode->next;
            }

                // Last node or middle node
            else {
                prevNode->next = currNode->next;
            }
            free(currNode);
            break;
        }
        prevNode = currNode;
        currNode = currNode->next;
    }
}

handler_t search(struct hashMap* mp, int key)
{

    // Getting the bucket index
    // for the given key
    int bucketIndex = hashFunction(mp, key);

    // Head of the linked list
    // present at bucket index
    struct node* bucketHead = mp->arr[bucketIndex];
    while (bucketHead != NULL) {

        // Key is found in the hashMap
        if (bucketHead->key == key) {
            return bucketHead->value;
        }
        bucketHead = bucketHead->next;
    }

    // If no key found in the hashMap
    // equal to the given key
    // returning NULL
    return NULL;
}


```

Assignments\Assignment4\makefile

```
all: react_server


react_server.o: react_server.c st_reactor.h hash.h
	gcc -c react_server.c -o react_server.o

react_server: react_server.o st_reactor.so
	gcc react_server.o -L. ./st_reactor.so -o react_server

st_reactor.o: st_reactor.c hash.c st_reactor.h hash.h
	gcc -c -fpic st_reactor.c -o st_reactor.o
```

```

st_reactor.so: st_reactor.o hash.o
	gcc -shared st_reactor.o hash.o -o st_reactor.so -lpthread

hash.o: hash.c hash.h
	gcc -c hash.c -o hash.o

client:
	gcc client_tst.c -o client

.PHONY: all clean

clean:
	rm -f *.o *.so react_server
```

Assignments\Assignment4\hash.h

```c


#ifndef HS
#define HS

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef void (*handler_t)(int fd, void * args ) ;


typedef struct node {

    // key is string
    int key;

    // value is also string
    handler_t value;
    struct node* next;
} node, * pnode;

typedef struct hashMap {

    // Current number of elements in hashMap
    // and capacity of hashMap
    int numOfElements, capacity;

    // hold base address array of linked list
    struct node** arr;
} hashMap, * PhashMap;

void setNode(struct node* node, int key , handler_t value);
void initializeHashMap(struct hashMap* mp);
void increase_HashMap(struct hashMap* mp,int new_capacity);
int hashFunction(struct hashMap* mp, int key);
void insert(struct hashMap* mp, int key, handler_t value);
void delete (struct hashMap* mp, int key);
handler_t search(struct hashMap* mp, int key);



```

```
#endif
```

Assignments\Assignment4\react_server.c

```
/*
** selectserver.c -- a cheezy multiperson chat server
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "st_reactor.h"
#include "poll.h"

#define DEBUG(stage) printf("DEBUG: %s\n",stage)

#define PORT "9034"   // port we're listening on

void handler(int fd, void *args) {
    char buff[2048];
    memset(buff, 0, 2048);
    if (recv(fd, buff, 2048, 0) < 0) {
        printf("recv error");
    } else {
        printf("%s", buff);
    }

}

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa) {
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in *) sa)->sin_addr);
    }

    return &(((struct sockaddr_in6 *) sa)->sin6_addr);
}

int main(void) {
    reactor *worker = createReactor();
    startReactor(worker);

    fd_set master;    // master file descriptor list
    fd_set read_fds;  // temp file descriptor list for select()
    int fdmax;        // maximum file descriptor number

    int listener;     // listening socket descriptor
    int newfd;        // newly accept()ed socket descriptor
    struct sockaddr_storage remoteaddr; // client address
    socklen_t addrlen;

    char buf[256];    // buffer for client data
    int nbytes;

    char remoteIP[INET6_ADDRSTRLEN];

    int yes = 1;        // for setsockopt() SO_REUSEADDR, below
    int i, j, rv;

    struct addrinfo hints, *ai, *p;

    FD_ZERO(&master);    // clear the master and temp sets
    FD_ZERO(&read_fds);

    // get us a socket and bind it
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
        exit(1);
    }

    for (p = ai; p != NULL; p = p->ai_next) {
        listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (listener < 0) {
            continue;
        }

        // lose the pesky "address already in use" error message
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

        if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
            close(listener);
            continue;
        }

        break;
    }

    // if we got here, it means we didn't get bound
    if (p == NULL) {
        fprintf(stderr, "selectserver: failed to bind\n");
        exit(2);
    }

    freeaddrinfo(ai); // all done with this

    // listen
    if (listen(listener, 10) == -1) {
        perror("listen");
        exit(3);
    }

    // add the listener to the master set
    FD_SET(listener, &master);

    // keep track of the biggest file descriptor
    fdmax = listener; // so far, it's this one

    // main loop
    for (;;) {
        read_fds = master; // copy it
        if (select(fdmax + 1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(4);
        }

        // run through the existing connections looking for data to read
        for (i = 0; i <= fdmax; i++) {
            if (FD_ISSET(i, &read_fds)) { // we got one!!
                if (i == listener) {
                    // handle new connections
                    addrlen = sizeof remoteaddr;
                    newfd = accept(listener,
                                   (struct sockaddr *) &remoteaddr,
                                   &addrlen);

                    if (newfd == -1) {
                        perror("accept");
                    } else {
                        // TODO: change...
                        addFd(worker, newfd, &handler);
                    }
                } else {
                    // handle data from a client
                    if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
                        // got error or connection closed by client
                        if (nbytes == 0) {
                            // connection closed
                            printf("selectserver: socket %d hung up\n", i);
                        } else {
                            perror("recv");
                        }
                        close(i); // bye!
                        FD_CLR(i, &master); // remove from master set
                    } else {
                        // we got some data from a client
                        for (j = 0; j <= fdmax; j++) {
                            // send to everyone!
                            if (FD_ISSET(j, &master)) {
                                // except the listener and ourselves
                                if (j != listener && j != i) {
                                    if (send(j, buf, nbytes, 0) == -1) {
                                        perror("send");
                                    }
                                }
                            }
                        }
                    }
                } // END handle data from client
            } // END got new incoming connection
        } // END looping through file descriptors
    } // END for(;;)--and you thought it would never end!

    return 0;
}

```

Assignments\Assignment4\st_reactor.c

```c
//
// Created by alon on 5/18/23.
//

#include "st_reactor.h"
#include <poll.h>
#define DEBUG(stage) printf("DEBUG: %s\n",stage)

void *thread_func(void *phash);

void *createReactor() {
    preactor new_reactor = malloc(sizeof(reactor));
    new_reactor->thread = -1;

    PhashMap hashMap = malloc(sizeof(hashMap));
    initializeHashMap(hashMap);
    new_reactor->hash = hashMap;

    new_reactor->WaitFor = WaitFor;
    new_reactor->addFd = addFd;
    new_reactor->startReactor = startReactor;
    new_reactor->stopReactor = stopReactor;

    return new_reactor;
}

void shoutDownReactor(void *this) {
    stopReactor(this);
    reactor *r = (reactor *) (this);
    free(r->hash->arr);
    free(r->hash);
    free(r);
}

void stopReactor(void *this) {
    int t = ((preactor) this)->thread;
    if (t != -1) {
        ((preactor) this)->thread = -1;
        pthread_cancel(t); // ask the thread to stop
        pthread_join(t, NULL); // wait until the thread will stop
    }
}

void startReactor(void *this) {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, ((preactor) this)->hash);
    ((preactor) this)->thread = thread;
}

void cleanupHandler(void *arg) { //from gpt,  asked it: i want to clean up allocated space 
before canceling my thread.
    // Cleanup code here
    free(arg);
}

void *thread_func(void *phash_param) {
    PhashMap phash = phash_param;
    int my_fds_num = 0;
    struct pollfd *fds = malloc(sizeof(struct pollfd) * (phash->numOfElements));
```

```
//    struct pollfd fds2[phash->numOfElements];
    int index = 0;
    // fill the fds array
    for (int i = 0; i < phash->capacity; ++i) {
        if (phash->arr[i] != NULL) {
            pnode temp = phash->arr[i];
            do {
                fds[index].fd = temp->key;
                fds[index].events = POLLIN;
                temp = temp->next;
                index++;
            } while (temp != NULL);
        }
    }
    my_fds_num = index;
    if (index != phash->numOfElements) {
        printf("missed some fds ?\n");
        printf("expected: %d, found %d\n", phash->numOfElements, my_fds_num);
    }


    pthread_cleanup_push(cleanupHandler, fds) ;

            while (1) {
                // check for cancel
                pthread_testcancel();
                //pool handle
                int err = poll(fds, my_fds_num, 1000);
                if (err < 0) {
                    printf("poll failed\n");
                }
                if (err > 0) {
                    for (int i = 0; i < my_fds_num; ++i) {
                        if (fds[i].revents && POLLIN) {
                            search(phash, fds[i].fd)(fds[i].fd, NULL); //activate the fds
function
                        }
                    }
                }
                if (my_fds_num < phash->numOfElements) {
                    // update the fds list:
                    int size = phash->numOfElements;
                    free(fds);
                    // resize fds
                    fds = malloc(sizeof(struct pollfd) * size);
                    index = 0;
                    // fill fds up
                    for (int i = 0; i < phash->capacity; ++i) {
                        if (phash->arr[i] != NULL) {
                            pnode temp = phash->arr[i];
                            do {
                                fds[index].fd = temp->key;
                                fds[index].events = POLLIN;
                                temp = temp->next;
                                index++;
                            } while (temp != NULL);
                        }
                    }
                    my_fds_num = index;
                }
```

```c
            }
    pthread_cleanup_pop(1);

}

void addFd(void *this, int fd, handler_t handler) {
    reactor * r = (reactor *)this;
    insert(r->hash,fd,handler);
}

void WaitFor(void * this){
    reactor * r  = (reactor *)this;
    pthread_join(r->thread,NULL );
}


```

Assignments\Assignment4\st_reactor.h

```c
//
// Created by alon on 5/18/23.
//

#ifndef ASSIGNMENT4_ST_REACTOR_H
#define ASSIGNMENT4_ST_REACTOR_H
#include "hash.h"
#include <pthread.h>


typedef struct reactor{
    PhashMap hash;
    int thread;

    void (* stopReactor)(void * this);
    void (*startReactor)(void* this);
    void (*addFd)(void * this,int tfd, handler_t handler);
    void (*WaitFor)(void * this);

} reactor, * preactor ;

void * createReactor();

void stopReactor(void * this);

void startReactor(void* this);

void addFd(void * this,int tfd, handler_t handler);

void WaitFor(void * this);

#endif //ASSIGNMENT4_ST_REACTOR_H
```