

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

```
1  /*
2  ** client.c -- a stream socket client demo
3  */
4
5  #include <stdio.h>
6  #include <poll.h>
7  #include <stdlib.h>
8  #include <unistd.h>
9  #include <errno.h>
10 #include <string.h>
11 #include <netdb.h>
12 #include <sys/types.h>
13 #include <netinet/in.h>
14 #include <sys/socket.h>
15
16 #include <arpa/inet.h>
17
18 #define PORT "9034" // the port client will be connecting to
19
20 #define MAXDATASIZE 100 // max number of bytes we can get at once
21
22 // get sockaddr, IPv4 or IPv6:
23 void *get_in_addr(struct sockaddr *sa)
24 {
25     if (sa->sa_family == AF_INET) {
26         return &(((struct sockaddr_in*)sa)->sin_addr);
27     }
28
29     return &(((struct sockaddr_in6*)sa)->sin6_addr);
30 }
31
32 int main(int argc, char *argv[])
33 {
34     int sockfd, numbytes;
35     char buf[MAXDATASIZE];
36     struct addrinfo hints, *servinfo, *p;
37     int rv;
38     char s[INET6_ADDRSTRLEN];
39
40     if (argc != 2) {
41         fprintf(stderr, "usage: client hostname\n");
42         exit(1);
```

```

43     }
44
45     memset(&hints, 0, sizeof hints);
46     hints.ai_family = AF_UNSPEC;
47     hints.ai_socktype = SOCK_STREAM;
48
49     if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
50         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
51         return 1;
52     }
53
54     // loop through all the results and connect to the first we can
55     for(p = servinfo; p != NULL; p = p->ai_next) {
56         if ((sockfd = socket(p->ai_family, p->ai_socktype,
57                             p->ai_protocol)) == -1) {
58             perror("client: socket");
59             continue;
60         }
61
62         if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
63             perror("client: connect");
64             close(sockfd);
65             continue;
66         }
67
68         break;
69     }
70
71     if (p == NULL) {
72         fprintf(stderr, "client: failed to connect\n");
73         return 2;
74     }
75
76     inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
77             s, sizeof s);
78     printf("client: connecting to %s\n", s);
79
80     freeaddrinfo(servinfo); // all done with this structure
81     struct pollfd pfd[2];
82     pfd[0].fd = 0; // Stdin
83     pfd[0].events = POLLIN;
84     pfd[1].fd = sockfd;
85     pfd[1].events = POLLIN;
86     for (;;) {
87         poll ( pfd, 2, -1);
88         if (pfd[0].revents & POLLIN) {
89             numbytes = read (0, buf, MAXDATASIZE-1);
90             send ( sockfd, buf, numbytes,0);
91         } else {
92             numbytes = recv (sockfd, buf, MAXDATASIZE-1,0);
93             buf[numbytes]=0;
94             printf ("server:%s\n",buf);
95         }
96     }
97
98     close(sockfd);
99
100     return 0;
101 }
102

```

Assignments\Assignment4\hash.c

```

1 //
2 // Created by alon on 5/18/23.
3 //
4 /*
5  * base code copied from https://www.geeksforgeeks.org/implementation-of-hash-table-in-c-
6  * using-separate-chaining/
7  * and changed to fit for us
8  */
9
10 #include "hash.h"
11
12 // like constructor
13 void setNode(struct node* node, int key , handler_t value)
14 {
15     node->key = key;
16     node->value = value;
17     node->next = NULL;
18 }
19
20 // like constructor
21 void initializeHashMap(struct hashMap* mp)
22 {
23     // Default capacity in this case
24     mp->capacity = 100;
25     mp->numOfElements = 0;
26
27     // array of size = 1
28     mp->arr = (struct node**)malloc(sizeof(struct node*)
29                                     * mp->capacity);
30 }
31
32 void increase_HashMap(struct hashMap* mp,int new_capacity){
33     struct node ** new_arr = realloc(mp->arr,sizeof(struct node*) * new_capacity);
34     free(mp->arr);
35     mp->arr = new_arr;
36 }
37
38 int hashFunction(struct hashMap* mp, int key)
39 {
40     int bucketIndex;
41     int sum = 0, factor = 31;
42     for (int i = 0; i < key; i++) {
43         sum = ((sum % mp->capacity)
44               + (key * factor) % mp->capacity)
45               % mp->capacity;
46
47         // factor = factor * prime
48         // number....(prime
49         // number) ^ x
50         factor = ((factor % __INT16_MAX__)
51                  * (31 % __INT16_MAX__))
52                  % __INT16_MAX__;
53     }
54 }

```

```
55     }
56
57     bucketIndex = sum;
58     return bucketIndex;
59 }
60
61 void insert(struct hashMap* mp, int key, handler_t value)
62 {
63
64     // Getting bucket index for the given
65     // key - value pair
66     int bucketIndex = hashFunction(mp, key);
67     // Creating a new node
68     struct node* newNode = (struct node*)malloc(sizeof(struct node));
69
70
71
72     // Setting value of node
73     setNode(newNode, key, value);
74
75     // Bucket index is empty....no collision
76     if (mp->arr[bucketIndex] == NULL) {
77         mp->arr[bucketIndex] = newNode;
78     }
79
80     // Collision
81     else {
82
83         // Adding newNode at the head of
84         // linked list which is present
85         // at bucket index....insertion at
86         // head in linked list
87         newNode->next = mp->arr[bucketIndex];
88         mp->arr[bucketIndex] = newNode;
89     }
90     mp->numOfElements++;
91 }
92
93 void delete (struct hashMap* mp, int key)
94 {
95
96     // Getting bucket index for the
97     // given key
98     int bucketIndex = hashFunction(mp, key);
99
100     struct node* prevNode = NULL;
101
102     // Points to the head of
103     // linked list present at
104     // bucket index
105     struct node* currNode = mp->arr[bucketIndex];
106
107     while (currNode != NULL) {
108
109         // Key is matched at delete this
110         // node from linked list
111         if ( key == currNode->key ) {
112
113             // Head node
114             // deletion
```

```

115         if (currNode == mp->arr[bucketIndex]) {
116             mp->arr[bucketIndex] = currNode->next;
117         }
118
119         // Last node or middle node
120         else {
121             prevNode->next = currNode->next;
122         }
123         free(currNode);
124         break;
125     }
126     prevNode = currNode;
127     currNode = currNode->next;
128 }
129 }
130
131 handler_t search(struct hashMap* mp, int key)
132 {
133
134     // Getting the bucket index
135     // for the given key
136     int bucketIndex = hashFunction(mp, key);
137
138     // Head of the linked list
139     // present at bucket index
140     struct node* bucketHead = mp->arr[bucketIndex];
141     while (bucketHead != NULL) {
142
143         // Key is found in the hashMap
144         if (bucketHead->key == key) {
145             return bucketHead->value;
146         }
147         bucketHead = bucketHead->next;
148     }
149
150     // If no key found in the hashMap
151     // equal to the given key
152     // returning NULL
153     return NULL;
154 }
155
156

```

Assignments\Assignment4\makefile

```

1 all: react_server
2
3
4 react_server.o: react_server.c st_reactor.h hash.h
5     gcc -c react_server.c -o react_server.o
6
7 react_server: react_server.o st_reactor.so
8     gcc react_server.o -L. ./st_reactor.so -o react_server
9
10 st_reactor.o: st_reactor.c hash.c st_reactor.h hash.h
11     gcc -c -fpic st_reactor.c -o st_reactor.o

```

```

12
13 st_reactor.so: st_reactor.o hash.o
14     gcc -shared st_reactor.o hash.o -o st_reactor.so -lpthread
15
16 hash.o: hash.c hash.h
17     gcc -c hash.c -o hash.o
18
19 client:
20     gcc client_tst.c -o client
21
22 .PHONY: all clean
23
24 clean:
25     rm -f *.o *.so react_server

```

Assignments\Assignment4\hash.h

```

1
2
3 #ifndef HS
4 #define HS
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <string.h>
9 typedef void (*handler_t)(int fd, void * args ) ;
10
11
12 typedef struct node {
13
14     // key is string
15     int key;
16
17     // value is also string
18     handler_t value;
19     struct node* next;
20 } node, * pnode;
21
22 typedef struct hashMap {
23
24     // Current number of elements in hashMap
25     // and capacity of hashMap
26     int numOfElements, capacity;
27
28     // hold base address array of linked list
29     struct node** arr;
30 } hashMap, * PhashMap;
31
32 void setNode(struct node* node, int key , handler_t value);
33 void initializeHashMap(struct hashMap* mp);
34 void increase_HashMap(struct hashMap* mp,int new_capacity);
35 int hashFunction(struct hashMap* mp, int key);
36 void insert(struct hashMap* mp, int key, handler_t value);
37 void delete (struct hashMap* mp, int key);
38 handler_t search(struct hashMap* mp, int key);
39
40
41

```

42 | #endif

Assignments\Assignment4\react_server.c

```

1  /*
2  ** selectserver.c -- a cheezy multiperson chat server
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <unistd.h>
9  #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <netinet/in.h>
12 #include <arpa/inet.h>
13 #include <netdb.h>
14 #include "st_reactor.h"
15 #include "poll.h"
16
17 #define DEBUG(stage) printf("DEBUG: %s\n",stage)
18
19 #define PORT "9034"    // port we're listening on
20
21 void handler(int fd, void *args) {
22     char buff[2048];
23     memset(buff, 0, 2048);
24     if (recv(fd, buff, 2048, 0) < 0) {
25         printf("recv error");
26     } else {
27         printf("%s", buff);
28     }
29 }
30
31 // get sockaddr, IPv4 or IPv6:
32 void *get_in_addr(struct sockaddr *sa) {
33     if (sa->sa_family == AF_INET) {
34         return &(((struct sockaddr_in *) sa)->sin_addr);
35     }
36
37     return &(((struct sockaddr_in6 *) sa)->sin6_addr);
38 }
39
40 int main(void) {
41     reactor *worker = createReactor();
42     startReactor(worker);
43
44     fd_set master;    // master file descriptor list
45     fd_set read_fds; // temp file descriptor list for select()
46     int fdmax;        // maximum file descriptor number
47
48     int listener;      // listening socket descriptor
49     int newfd;         // newly accept()ed socket descriptor
50     struct sockaddr_storage remoteaddr; // client address
51     socklen_t addrlen;
52
53     char buf[256];      // buffer for client data

```

```
55     int nbytes;
56
57     char remoteIP[INET6_ADDRSTRLEN];
58
59     int yes = 1;          // for setsockopt() SO_REUSEADDR, below
60     int i, j, rv;
61
62     struct addrinfo hints, *ai, *p;
63
64     FD_ZERO(&master);    // clear the master and temp sets
65     FD_ZERO(&read_fds);
66
67     // get us a socket and bind it
68     memset(&hints, 0, sizeof hints);
69     hints.ai_family = AF_UNSPEC;
70     hints.ai_socktype = SOCK_STREAM;
71     hints.ai_flags = AI_PASSIVE;
72     if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
73         fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
74         exit(1);
75     }
76
77     for (p = ai; p != NULL; p = p->ai_next) {
78         listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
79         if (listener < 0) {
80             continue;
81         }
82
83         // lose the pesky "address already in use" error message
84         setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
85
86         if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
87             close(listener);
88             continue;
89         }
90
91         break;
92     }
93
94     // if we got here, it means we didn't get bound
95     if (p == NULL) {
96         fprintf(stderr, "selectserver: failed to bind\n");
97         exit(2);
98     }
99
100     freeaddrinfo(ai); // all done with this
101
102     // listen
103     if (listen(listener, 10) == -1) {
104         perror("listen");
105         exit(3);
106     }
107
108     // add the listener to the master set
109     FD_SET(listener, &master);
110
111     // keep track of the biggest file descriptor
112     fdmax = listener; // so far, it's this one
113
114     // main loop
```



```

115     for (;;) {
116         read_fds = master; // copy it
117         if (select(fdmax + 1, &read_fds, NULL, NULL, NULL) == -1) {
118             perror("select");
119             exit(4);
120         }
121
122         // run through the existing connections looking for data to read
123         for (i = 0; i <= fdmax; i++) {
124             if (FD_ISSET(i, &read_fds)) { // we got one!!
125                 if (i == listener) {
126                     // handle new connections
127                     addrlen = sizeof remoteaddr;
128                     newfd = accept(listener,
129                                   (struct sockaddr *) &remoteaddr,
130                                   &addrlen);
131
132                     if (newfd == -1) {
133                         perror("accept");
134                     } else {
135                         // TODO: change...
136                         addFd(worker, newfd, &handler);
137                     }
138                 } else {
139                     // handle data from a client
140                     if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
141                         // got error or connection closed by client
142                         if (nbytes == 0) {
143                             // connection closed
144                             printf("selectserver: socket %d hung up\n", i);
145                         } else {
146                             perror("recv");
147                         }
148                         close(i); // bye!
149                         FD_CLR(i, &master); // remove from master set
150                     } else {
151                         // we got some data from a client
152                         for (j = 0; j <= fdmax; j++) {
153                             // send to everyone!
154                             if (FD_ISSET(j, &master)) {
155                                 // except the listener and ourselves
156                                 if (j != listener && j != i) {
157                                     if (send(j, buf, nbytes, 0) == -1) {
158                                         perror("send");
159                                     }
160                                 }
161                             }
162                         }
163                     }
164                 } // END handle data from client
165             } // END got new incoming connection
166         } // END looping through file descriptors
167     } // END for(;;)--and you thought it would never end!
168
169     return 0;
170 }
171

```

Assignments\Assignment4\st_reactor.c

```

1 //
2 // Created by alon on 5/18/23.
3 //
4
5 #include "st_reactor.h"
6 #include <poll.h>
7 #define DEBUG(stage) printf("DEBUG: %s\n",stage)
8
9 void *thread_func(void *phash);
10
11 void *createReactor() {
12     preactor new_reactor = malloc(sizeof(reactor));
13     new_reactor->thread = -1;
14
15     PhashMap hashMap = malloc(sizeof(hashMap));
16     initializeHashMap(hashMap);
17     new_reactor->hash = hashMap;
18
19     new_reactor->WaitFor = WaitFor;
20     new_reactor->addFd = addFd;
21     new_reactor->startReactor = startReactor;
22     new_reactor->stopReactor = stopReactor;
23
24     return new_reactor;
25 }
26
27 void shoutDownReactor(void *this) {
28     stopReactor(this);
29     reactor *r = (reactor *) (this);
30     free(r->hash->arr);
31     free(r->hash);
32     free(r);
33 }
34
35 void stopReactor(void *this) {
36     int t = ((preactor) this)->thread;
37     if (t != -1) {
38         ((preactor) this)->thread = -1;
39         pthread_cancel(t); // ask the thread to stop
40         pthread_join(t, NULL); // wait until the thread will stop
41     }
42 }
43
44 void startReactor(void *this) {
45     pthread_t thread;
46     pthread_create(&thread, NULL, thread_func, ((preactor) this)->hash);
47     ((preactor) this)->thread = thread;
48 }
49
50 void cleanupHandler(void *arg) { //from gpt, asked it: i want to clean up allocated space
    before canceling my thread.
51     // Cleanup code here
52     free(arg);
53 }
54
55 void *thread_func(void *phash_param) {
56     PhashMap phash = phash_param;
57     int my_fds_num = 0;
58     struct pollfd *fds = malloc(sizeof(struct pollfd) * (phash->numOfElements));

```

```

59 // struct pollfd fds2[phash->numOfElements];
60 int index = 0;
61 // fill the fds array
62 for (int i = 0; i < phash->capacity; ++i) {
63     if (phash->arr[i] != NULL) {
64         pnode temp = phash->arr[i];
65         do {
66             fds[index].fd = temp->key;
67             fds[index].events = POLLIN;
68             temp = temp->next;
69             index++;
70         } while (temp != NULL);
71     }
72 }
73 my_fds_num = index;
74 if (index != phash->numOfElements) {
75     printf("missed some fds ?\n");
76     printf("expected: %d, found %d\n", phash->numOfElements, my_fds_num);
77 }
78
79
80 pthread_cleanup_push(cleanupHandler, fds) ;
81
82 while (1) {
83     // check for cancel
84     pthread_testcancel();
85     //pool handle
86     int err = poll(fds, my_fds_num, 1000);
87     if (err < 0) {
88         printf("poll failed\n");
89     }
90     if (err > 0) {
91         for (int i = 0; i < my_fds_num; ++i) {
92             if (fds[i].revents && POLLIN) {
93                 search(phash, fds[i].fd)(fds[i].fd, NULL); //activate the fds
function
94             }
95         }
96     }
97     if (my_fds_num < phash->numOfElements) {
98         // update the fds list:
99         int size = phash->numOfElements;
100         free(fds);
101         // resize fds
102         fds = malloc(sizeof(struct pollfd) * size);
103         index = 0;
104         // fill fds up
105         for (int i = 0; i < phash->capacity; ++i) {
106             if (phash->arr[i] != NULL) {
107                 pnode temp = phash->arr[i];
108                 do {
109                     fds[index].fd = temp->key;
110                     fds[index].events = POLLIN;
111                     temp = temp->next;
112                     index++;
113                 } while (temp != NULL);
114             }
115         }
116         my_fds_num = index;
117     }

```

```

118         }
119         pthread_cleanup_pop(1);
120
121     }
122
123     void addFd(void *this, int fd, handler_t handler) {
124         reactor * r = (reactor *)this;
125         insert(r->hash,fd,handler);
126     }
127
128     void WaitFor(void * this){
129         reactor * r = (reactor *)this;
130         pthread_join(r->thread,NULL );
131     }
132
133

```

Assignments\Assignment4\st_reactor.h

```

1 //
2 // Created by alon on 5/18/23.
3 //
4
5 #ifndef ASSIGNMENT4_ST_REACTOR_H
6 #define ASSIGNMENT4_ST_REACTOR_H
7 #include "hash.h"
8 #include <pthread.h>
9
10
11 typedef struct reactor{
12     PhashMap hash;
13     int thread;
14
15     void (* stopReactor)(void * this);
16     void (*startReactor)(void* this);
17     void (*addFd)(void * this,int tfd, handler_t handler);
18     void (*WaitFor)(void * this);
19
20 } reactor, * preactor ;
21
22 void * createReactor();
23
24 void stopReactor(void * this);
25
26 void startReactor(void* this);
27
28 void addFd(void * this,int tfd, handler_t handler);
29
30 void WaitFor(void * this);
31
32 #endif //ASSIGNMENT4_ST_REACTOR_H

```