1:

/\* CS 415 Project 3- dtsv1.c

\* Name: Carter Young;

\* Duck ID: cartery;

\* UO: 951690164;

\* Completed: 05/28/23;

\*

\* This is my own work. Conversed sparingly re: the project with Freddy Lopez,

\* Mason Kline, and Sydney Whiting.

\*/

#include "BXP/bxp.h"

#include <assert.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#define UNUSED \_\_attribute\_\_((unused))

#define SERVICE "DTS"

#define HOST "localhost"

#define PORT "19999"

void \*svcFxn(void \*args) {

BXPService bxps = (BXPService)args;

BXPEndpoint ep;

char query[1024], response[1025];

unsigned qlen;

while((qlen = bxp\_query(bxps, &ep, query, 1024)) > 0) {

sprintf(response,"1%s", query);

bxp\_response(bxps, &ep, response, qlen+1);

}

return NULL;

}

int main(UNUSED int argc, UNUSED char \*argv[]) {

BXPService bxps;

pthread\_t svcThread;

assert(bxp\_init(19999, 1)); // Bind to port 19999

bxps = bxp\_offer("DTS"); // Offer service named "DTS"

assert(! pthread\_create(&svcThread, NULL, svcFxn, (void \*)bxps));

pthread\_join(svcThread, NULL); // Wait for the service thread to finish

return 0;

}

2:

/\* CS 415 Project 3- dtsv2.c

\* Name: Carter Young;

\* Duck ID: cartery;

\* UO: 951690164;

\* Completed: 05/28/23;

\*

\* This is my own work. Conversed sparingly re: the project with Freddy Lopez,

\* Mason Kline, and Sydney Whiting.

\*/

#include "BXP/bxp.h"

#include <pthread.h>

#include <assert.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define UNUSED \_\_attribute\_\_((unused))

#define SERVICE "DTS"

#define HOST "localhost"

#define PORT "19999"

BXPService \*bxps;

int checkCommand(char \*command, int numArgs) {

if (strcmp(command, "OneShot") == 0 && numArgs == 7) {

return 1;

} else if (strcmp(command, "Repeat") == 0 && numArgs == 9) {

return 1;

} else if (strcmp(command, "Cancel") == 0 && numArgs == 2) {

return 1;

}

return 0;

}

void \*svcFxn(void \*args) {

BXPService bxps = (BXPService)args;

BXPEndpoint ep;

char query[1024], response[1025], original\_query[1024];

unsigned qlen;

while ((qlen = bxp\_query(bxps, &ep, query, 1024)) > 0) {

char\* command;

int numArgs = 1;

// Save the original query before tokenizing

strncpy(original\_query, query, 1024);

command = strtok(query, "|");

while (strtok(NULL, "|") != NULL) {

numArgs++;

}

if(checkCommand(command, numArgs)){

// Prepend "1" to the response when the command is valid

sprintf(response, "1%s", original\_query);

} else {

// Prepend "0" to the response when the command is invalid

sprintf(response, "0%s", original\_query);

}

unsigned rlen = strlen(response) + 1;

assert(bxp\_response(bxps, &ep, response, rlen));

}

return NULL;

}

int main(UNUSED int argc, UNUSED char \*argv[]) {

BXPService bxps;

pthread\_t svcThread;

assert(bxp\_init(19999, 1)); // Bind to port 19999

bxps = bxp\_offer("DTS"); // Offer service named "DTS"

assert(! pthread\_create(&svcThread, NULL, svcFxn, (void \*)bxps));

pthread\_join(svcThread, NULL); // Wait for the service thread to finish

return 0;

}

3:

/\* CS 415 Project 3- dtsv3.c

\* Name: Carter Young;

\* Duck ID: cartery;

\* UO: 951690164;

\* Completed: 05/31/23;

\*

\* This is my own work. Conversed sparingly re: the project with Freddy Lopez,

\* Mason Kline, and Sydney Whiting.

\*/

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <pthread.h>

#include <assert.h>

#include <unistd.h>

#include <sys/time.h>

#include <valgrind/valgrind.h>

#include "BXP/bxp.h"

#include "ADTs/ADTdefs.h"

#include "ADTs/iterator.h"

#include "ADTs/heapprioqueue.h"

#include "ADTs/hashmap.h"

static unsigned long next\_sid = 0;

#define UNUSED \_\_attribute\_\_ ((unused))

#define MAX\_CMD\_LENGTH 256

#define SERVICE "DTS"

#define PORT 19999

#define HOST "localhost"

BXPService \*bxps;

typedef struct event {

char \*name;

unsigned long event\_time;

unsigned long clid;

unsigned long sid;

char \*host;

char \*service;

unsigned int port;

int cancelled;

int hashmap\_lookup;

time\_t secs;

time\_t usecs;

} Event; // event information

PrioQueue \*pqueue;

Map \*hmap;

pthread\_mutex\_t event\_mutex;

// create an event

Event\* event\_create() {

Event\* event = (Event\*)malloc(sizeof(Event));

event->sid = next\_sid++;

event->hashmap\_lookup = 0;

return event;

}

// compare function

int event\_compare(void\* sid1, void\* sid2) {

// cast to an unsigned long pointer, then dereference that pointer with \*

return (\*(unsigned long\*)sid1) - (\*(unsigned long\*)sid2);

}

// free function

void event\_delete(void\* event) {

free((Event\*)event);

}

// hash function

long event\_hash(void\* sid, long N) {

return (\*(unsigned long\*)sid) % N;

}

void handle\_oneshot(char\* command, BXPService bxps, BXPEndpoint\* ep) {

Event\* event = event\_create();

// Parse the command

char\* tokens[7]; // "OneShot", <id>, <secs>, <usecs>, <ip>, <service>, <port>

char\* token = strtok(command, "|");

int i = 0;

while (token != NULL && i < 8) {

tokens[i] = token;

token = strtok(NULL, "|");

i++;

}

// Allocate memory for the unique identifiers (sid and time)

int\* clid\_ptr = malloc(sizeof(int));

// Assign the values from tokens to the identifiers

\*clid\_ptr = atoi(tokens[1]);

// Assign the event attributes

event->name = strdup(tokens[0]);

event->secs = atoi(tokens[2]);

event->usecs = atoi(tokens[3]);

event->clid = \*clid\_ptr;

event->event\_time = ((event->secs \* 1000000) + event->usecs); // convert to usecs for granularity

event->host = strdup(tokens[4]);

event->service = strdup(tokens[5]);

event->port = atoi(tokens[6]);

event->cancelled = 0;

// Insert the event into the priority queue and the hash map

// Using the sid as the key

pthread\_mutex\_lock(&event\_mutex);

pqueue->insert(pqueue, (void \*)&(event->event\_time), (void\*)(event)); // Use the time as the key

hmap->put(hmap, (void \*)&(event->sid), (void\*)(event));

pthread\_mutex\_unlock(&event\_mutex);

// Prepare the response

char response[1025];

sprintf(response, "1%08lu", event->sid);

// Send the response to the client

if(!bxp\_response(bxps, ep, response, strlen(response) + 1)){

fprintf(stderr, "Failed to send response\n");

exit(EXIT\_FAILURE);

}

}

void handle\_cancel(char\* command, BXPService bxps, BXPEndpoint\* ep) {

Event \*event;

char\* tokens[2]; // "Cancel", <sid>

char\* token = strtok(command, "|");

int i = 0;

while (token != NULL && i < 2) {

tokens[i] = token;

token = strtok(NULL, "|");

i++;

}

// Get event from hM

unsigned long sid = atol(tokens[1]);

pthread\_mutex\_lock(&event\_mutex);

if(hmap->get(hmap, (void \*)(&sid), (void \*\*)(&event)) != 0) {

// Set event as cancelled

event->cancelled = 1;

hmap->remove(hmap, (void \*)(&sid));

// send response to client

char response[1025];

sprintf(response, "1%08lu", sid);

assert(bxp\_response(bxps, ep, response, strlen(response) + 1));

pthread\_mutex\_unlock(&event\_mutex);

} else {

pthread\_mutex\_unlock(&event\_mutex);

printf("sid does not exist.\n");

// send error message to client

char response[1025];

sprintf(response, "Error: sid %lu does not exist.", sid);

if(!bxp\_response(bxps, ep, response, strlen(response) + 1)){

fprintf(stderr, "Failed to send response\n");

}

}

}

void notify\_client(Event\* event) {

BXPConnection bxpc = bxp\_connect(event->host, event->port, event->service, 1, 1);

char req[MAX\_CMD\_LENGTH];

unsigned int reqlen = (unsigned int)sprintf(req, "%ld", event->clid);

char resp[MAX\_CMD\_LENGTH];

unsigned int resplen;

bxp\_call(bxpc, req, reqlen, resp, sizeof(resp), &resplen);

bxp\_disconnect(bxpc);

}

void \*svcFxn(void \*args) {

BXPService bxps = (BXPService)args;

BXPEndpoint ep;

char command[1024];

char response[1025];

unsigned len;

while ((len = bxp\_query(bxps, &ep, command, sizeof(command))) > 0) {

if (strncmp(command, "OneShot", 7) == 0) {

handle\_oneshot(command, bxps, &ep);

} else if (strncmp(command, "Cancel", 6) == 0) {

handle\_cancel(command, bxps, &ep);

} else if (strncmp(command, "Repeat", 6) == 0) {

sprintf(response, "0%s", command);

assert(bxp\_response(bxps, &ep, response, strlen(response) + 1));

} else {

printf("Unknown command: %s\n", command);

}

}

return NULL;

}

void \*timeFxn() {

while(1) {

pthread\_mutex\_lock(&event\_mutex);

if (pqueue->isEmpty(pqueue)) {

pthread\_mutex\_unlock(&event\_mutex);

usleep(100000); // sleep for 100 ms

continue;

}

// Get current time

struct timeval tp;

gettimeofday(&tp, NULL);

unsigned long current\_time = tp.tv\_sec \* 1000000 + tp.tv\_usec;

// Get the event with the smallest time from the priority queue without removing it

Event\* next\_event;

void \*priority;

void \*value;

if(!pqueue->min(pqueue, &priority, &value)) {

pthread\_mutex\_unlock(&event\_mutex);

usleep(100000); // sleep for 100 ms

continue;

}

next\_event = (Event\*)value;

if(next\_event->event\_time <= current\_time) {

// The event is due, so we can remove it from the queue

pqueue->removeMin(pqueue, &priority, &value);

hmap->remove(hmap, (void \*)&(next\_event->sid));

pthread\_mutex\_unlock(&event\_mutex);

// If the event is cancelled, just delete it

if(next\_event->cancelled) {

event\_delete(next\_event);

} else {

// If the event is not cancelled, fire the event

printf("Event fired: %lu|%s|%s|%u\n", next\_event->clid, next\_event->host, next\_event->service, next\_event->port);

notify\_client(next\_event); // Notify when we fire an event

event\_delete(next\_event); // Free the event after it's fired

}

} else {

pthread\_mutex\_unlock(&event\_mutex);

usleep(100000); // sleep for 100 ms

}

}

return NULL;

}

int main(UNUSED int argc, UNUSED char \*argv[]) {

BXPService bxps;

// Initialize mutexes

pthread\_mutex\_init(&event\_mutex, NULL);

// Initialize threads

pthread\_t svcThread;

pthread\_t timeThread;

// Initialize hM

hmap = HashMap(0, 0.0, event\_hash, event\_compare, doNothing, doNothing);

// Initialize pQ

pqueue = PrioQueue\_create(event\_compare, doNothing, event\_delete);

// Connect and offer!

assert(bxp\_init(PORT, 1)); // Initialize BXP

bxps = bxp\_offer(SERVICE);

// Assign threads

assert(! pthread\_create(&svcThread, NULL, svcFxn, (void \*)bxps));

assert(! pthread\_create(&timeThread, NULL, timeFxn, (void \*)bxps));

// Join threads

pthread\_join(svcThread, NULL); // Wait for the service thread to finish

pthread\_join(timeThread, NULL);

// Cleanup data structures

hmap->destroy(hmap);

pqueue->destroy(pqueue);

// Cleanup mutexes

pthread\_mutex\_destroy(&event\_mutex);

// Check w valgrind

VALGRIND\_MONITOR\_COMMAND("leak\_check summary");

// exit!

return 0;

}

4:

/\* CS 415 Project 3- dtsv4.c

\* Name: Carter Young;

\* Duck ID: cartery;

\* UO: 951690164;

\* Completed: 05/31/23;

\*

\* This is my own work. Conversed sparingly re: the project with Freddy Lopez,

\* Mason Kline, and Sydney Whiting.

\*/

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <pthread.h>

#include <assert.h>

#include <unistd.h>

#include <sys/time.h>

#include "BXP/bxp.h"

#include "ADTs/ADTdefs.h"

#include "ADTs/iterator.h"

#include "ADTs/heapprioqueue.h"

#include "ADTs/hashmap.h"

static unsigned long next\_sid = 0;

#define UNUSED \_\_attribute\_\_ ((unused))

#define MAX\_CMD\_LENGTH 256

#define SERVICE "DTS"

#define PORT 19999

#define HOST "localhost"

BXPService \*bxps;

typedef struct event {

char \*name;

unsigned long event\_time;

unsigned long clid;

unsigned long sid;

char \*host;

int repeats;

char \*service;

unsigned int port;

int cancelled;

int hashmap\_lookup;

time\_t secs;

time\_t usecs;

} Event; // event information

PrioQueue \*pqueue;

Map \*hmap;

pthread\_mutex\_t event\_mutex;

// create an event

Event\* event\_create() {

Event\* event = (Event\*)malloc(sizeof(Event));

event->sid = next\_sid++;

event->hashmap\_lookup = 0;

return event;

}

// compare function

int event\_compare(void\* sid1, void\* sid2) {

// cast to an unsigned long pointer, then dereference that pointer with \*

return (\*(unsigned long\*)sid1) - (\*(unsigned long\*)sid2);

}

// free function

void event\_delete(void\* event) {

free((Event\*)event);

}

// hash function

long event\_hash(void\* sid, long N) {

return (\*(unsigned long\*)sid) % N;

}

void handle\_oneshot(char\* command, BXPService bxps, BXPEndpoint\* ep) {

Event\* event = event\_create();

// Parse the command

char\* tokens[7]; // "OneShot", <id>, <secs>, <usecs>, <host>, <service>, <port>

char\* token = strtok(command, "|");

int i = 0;

while (token != NULL && i < 8) {

tokens[i] = token;

token = strtok(NULL, "|");

i++;

}

// Allocate memory for the unique identifiers (sid and time)

int\* clid\_ptr = malloc(sizeof(int));

// Assign the values from tokens to the identifiers

\*clid\_ptr = atoi(tokens[1]);

// Assign the event attributes

event->name = strdup(tokens[0]);

event->secs = atoi(tokens[2]);

event->usecs = atoi(tokens[3]);

event->clid = \*clid\_ptr;

event->event\_time = ((event->secs \* 1000000) + event->usecs); // convert to usecs for granularity

event->host = strdup(tokens[4]);

event->service = strdup(tokens[5]);

event->port = atoi(tokens[6]);

event->cancelled = 0;

// Insert the event into the priority queue and the hash map

// Using the sid as the key

pthread\_mutex\_lock(&event\_mutex);

pqueue->insert(pqueue, (void \*)&(event->event\_time), (void\*)(event)); // Use the time as the key

hmap->put(hmap, (void \*)&(event->sid), (void\*)(event));

pthread\_mutex\_unlock(&event\_mutex);

// Prepare the response

char response[1025];

sprintf(response, "1%08lu", event->sid);

// Send the response to the client

if(!bxp\_response(bxps, ep, response, strlen(response) + 1)){

fprintf(stderr, "Failed to send response\n");

exit(EXIT\_FAILURE);

}

//printf("OneShot Event scheduled\n");

}

void handle\_cancel(char\* command, BXPService bxps, BXPEndpoint\* ep) {

Event \*event;

char\* tokens[2]; // "Cancel", <sid>

char\* token = strtok(command, "|");

int i = 0;

while (token != NULL && i < 2) {

tokens[i] = token;

token = strtok(NULL, "|");

i++;

}

// Get event from hM

unsigned long sid = atol(tokens[1]);

pthread\_mutex\_lock(&event\_mutex);

if(hmap->get(hmap, (void \*)(&sid), (void \*\*)(&event)) != 0) {

// Set event as cancelled

event->cancelled = 1;

hmap->remove(hmap, (void \*)(&sid));

// send response to client

char response[1025];

sprintf(response, "1%08lu", sid);

assert(bxp\_response(bxps, ep, response, strlen(response) + 1));

pthread\_mutex\_unlock(&event\_mutex);

} else {

pthread\_mutex\_unlock(&event\_mutex);

printf("sid does not exist.\n");

// send error message to client

char response[1025];

sprintf(response, "Error: sid %lu does not exist.", sid);

if(!bxp\_response(bxps, ep, response, strlen(response) + 1)){

fprintf(stderr, "Failed to send response\n");

}

}

}

void handle\_repeat(char\* command, BXPService bxps, BXPEndpoint\* ep) {

Event\* event = event\_create();

// Parse the command

char\* tokens[8]; // "Repeat", <id>, <secs>, <usecs>, <repeats>, <host>, <service>, <port>

char\* token = strtok(command, "|");

int i = 0;

while (token != NULL && i < 9) {

tokens[i] = token;

token = strtok(NULL, "|");

i++;

}

// Allocate memory for the unique identifiers (sid and time)

int\* clid\_ptr = malloc(sizeof(int));

// Assign the values from tokens to the identifiers

\*clid\_ptr = atoi(tokens[1]);

// Assign the event attributes

event->name = strdup(tokens[0]);

event->secs = atoi(tokens[2]);

event->usecs = atoi(tokens[3]);

event->repeats = atoi(tokens[4]);

event->clid = \*clid\_ptr;

event->event\_time = ((event->secs \* 1000000) + event->usecs); // convert to usecs for granularity

event->host = strdup(tokens[5]);

event->service = strdup(tokens[6]);

event->port = atoi(tokens[7]);

event->cancelled = 0;

// Insert the event into the priority queue and the hash map

// Using the sid as the key

pthread\_mutex\_lock(&event\_mutex);

pqueue->insert(pqueue, (void \*)&(event->event\_time), (void\*)(event)); // Use the time as the key

hmap->put(hmap, (void \*)&(event->sid), (void\*)(event));

pthread\_mutex\_unlock(&event\_mutex);

// Prepare the response

char response[1025];

sprintf(response, "1%08lu", event->sid);

// Send the response to the client

if(!bxp\_response(bxps, ep, response, strlen(response) + 1)){

fprintf(stderr, "Failed to send response\n");

exit(EXIT\_FAILURE);

}

}

void notify\_client(Event\* event) {

BXPConnection bxpc = bxp\_connect(event->host, event->port, event->service, 1, 1);

char req[MAX\_CMD\_LENGTH];

unsigned int reqlen = (unsigned int)sprintf(req, "%ld", event->clid);

char resp[MAX\_CMD\_LENGTH];

unsigned int resplen;

bxp\_call(bxpc, req, reqlen, resp, sizeof(resp), &resplen);

bxp\_disconnect(bxpc);

}

void \*svcFxn(void \*args) {

BXPService bxps = (BXPService)args;

BXPEndpoint ep;

char command[1024];

unsigned len;

while ((len = bxp\_query(bxps, &ep, command, sizeof(command))) > 0) {

if (strncmp(command, "OneShot", 7) == 0) {

handle\_oneshot(command, bxps, &ep);

} else if (strncmp(command, "Cancel", 6) == 0) {

handle\_cancel(command, bxps, &ep);

} else if (strncmp(command, "Repeat", 6) == 0) {

handle\_repeat(command, bxps, &ep);

} else {

printf("Unknown command: %s\n", command);

}

}

return NULL;

}

void \*timeFxn() {

while(1) {

pthread\_mutex\_lock(&event\_mutex);

if (pqueue->isEmpty(pqueue)) {

pthread\_mutex\_unlock(&event\_mutex);

usleep(100000); // sleep for 100 ms

continue;

}

// Get current time

struct timeval tp;

gettimeofday(&tp, NULL);

unsigned long current\_time = tp.tv\_sec \* 1000000 + tp.tv\_usec;

// Get the event with the smallest time from the priority queue without removing it

Event\* next\_event;

void \*priority;

void \*value;

if(!pqueue->min(pqueue, &priority, &value)) {

pthread\_mutex\_unlock(&event\_mutex);

usleep(100000); // sleep for 100 ms

continue;

}

next\_event = (Event\*)value;

if(next\_event->event\_time <= current\_time) {

// The event is due, so we can remove it from the queue

pqueue->removeMin(pqueue, &priority, &value);

hmap->remove(hmap, (void \*)&(next\_event->sid));

pthread\_mutex\_unlock(&event\_mutex);

// If the event is cancelled, just delete it

if(next\_event->cancelled) {

event\_delete(next\_event);

} else {

// If the event is not cancelled, fire the event

printf("Event fired: %lu|%s|%s|%u\n", next\_event->clid, next\_event->host, next\_event->service, next\_event->port);

notify\_client(next\_event); // Notify when we fire an event

event\_delete(next\_event); // Free the event after it's fired

}

} else {

pthread\_mutex\_unlock(&event\_mutex);

usleep(100000); // sleep for 100 ms

}

}

return NULL;

}

int main(UNUSED int argc, UNUSED char \*argv[]) {

BXPService bxps;

// Initialize mutexes

pthread\_mutex\_init(&event\_mutex, NULL);

// Initialize threads

pthread\_t svcThread;

pthread\_t timeThread;

// Initialize hM

hmap = HashMap(0, 0.0, event\_hash, event\_compare, doNothing, doNothing);

// Initialize pQ

pqueue = PrioQueue\_create(event\_compare, doNothing, event\_delete);

// Connect and offer!

assert(bxp\_init(PORT, 1)); // Initialize BXP

bxps = bxp\_offer(SERVICE); // Offer service

// Assign threads

assert(! pthread\_create(&svcThread, NULL, svcFxn, (void \*)bxps));

assert(! pthread\_create(&timeThread, NULL, timeFxn, (void \*)bxps));

// Join threads

pthread\_join(svcThread, NULL); // Wait for the service thread to finish

pthread\_join(timeThread, NULL); // Wait for the time thread to finish

// Cleanup data structures

hmap->destroy(hmap);

pqueue->destroy(pqueue);

// Cleanup mutexes

pthread\_mutex\_destroy(&event\_mutex);

// exit!

return 0;

}