```
/*NAME:SHASHANK VENKAT
EMAIL: shashank18@g.ucla.edu
ID: 705303381
*/

#DESCRIPTION OF PROJECT IS UNDER "MULTITHREADING WITH HASHTABLES ON
RESUME"

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <getopt.h>
#include <time.h>
#include <stdbool.h>
#include <string.h>
#include <errno.h>
#include <sched.h>
#include <signal.h>
#include "SortedList.h"

#define BILLION 1000000000L;


long numThreads = 1;
long numIterations = 1;
long numSubLists = 1;

pthread_mutex_t* mutexLocks;
int* spinLocks;

long timeDiff = 0;


SortedList_t* listHeads;
SortedListElement_t* elements;
int* hashValues;

int isM = 1;
int isS = 1;

int isI = 1;
int isD = 1;
int isL = 1;

int isYield;

int opt_yield = 0;

void segfaultHandler(int sigNum) {
```

```
        if(sigNum == SIGSEGV){
                fprintf(stderr, "Segmentation Fault.\n");
                exit(2);
        }
}


long hash(const char *str){ //Hash function is taken from https://
stackoverflow.com/questions/7666509/hash-function-for-string
        unsigned long hash = 5381;
        int c;

        while(0 != (c = *str++))
                hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

        hash = hash % numSubLists; //need to mod by number of sublists
to get a key value between 0 and numSublists - 1
        return hash;
}

void initSubLists(){
        listHeads = malloc(sizeof(SortedList_t)*numSubLists); //malloc
all the sublists
        if(listHeads == NULL){
                fprintf(stderr, "Error in mallocing the lists.\n");
                exit(1);
        }
        for(long i = 0; i < numSubLists; i++){ //loop through the
sublists and make each head point to itself and set the keys equal to
NULL
                listHeads[i].prev = &listHeads[i];
                listHeads[i].next = &listHeads[i];
                listHeads[i].key = NULL;
        }
}

void initLocks(){
        if(isM == 1){
                mutexLocks =
malloc(sizeof(pthread_mutex_t)*numSubLists);
                if(mutexLocks == NULL){
                        fprintf(stderr, "Error in mallocing mutex
locks.\n");
                        exit(1);
                }
        }
        else if(isS == 1){
                spinLocks = malloc(sizeof(int) * numSubLists);
                if(spinLocks == NULL){
                        fprintf(stderr, "Error in mallocing mutex
```

```
locks.\n");
                        exit(1);
                }
        }
        for(long i = 0; i < numSubLists; i++){
                if(isM == 1){
                        if(pthread_mutex_init(&mutexLocks[i], NULL) <
0){
                                fprintf(stderr, "Unable to
initialize mutex lock.\n");
                                exit(1);
                        }
                }
                else if(isS == 1){
                        spinLocks[i] = 0;
                }
        }
}

void initElements(long numElements){
        elements = (SortedListElement_t*)
malloc(sizeof(SortedListElement_t) * numElements);
        if(elements == NULL){
                fprintf(stderr, "Error in mallocing elements of the
list.\n");
                exit(1);
        }
}

void createKeys(long numElements){
        char* randomKey;
        for(long i = 0; i < numElements; i++){
                randomKey = (char*) malloc(sizeof(char) * 10);
                for(int i = 0; i < 9; i++){ //create a random value
made up of ten characters, 9 of which are randomly selected
                        randomKey[i] = 'A' + rand() % 26;
                }
                randomKey[9] = '\0';
                elements[i].key = randomKey;

        }
}

void initKeyList(int numElements){
        hashValues = malloc(sizeof(long) * numElements);
        for(long i = 0; i < numElements; i++){
                hashValues[i] = hash(elements[i].key);
        }
}
```

```
void* threadFunc(void* id){
        long timeDiffSec = 0;
        long timeDiffNS = 0;
        timeDiff = 0;
        timeDiff++;

        struct timespec start, stop;
        int threadID = *((int*) id);


        long startIndex = threadID * numIterations;
        long i;

        for(i = startIndex; i < startIndex + numIterations; i++){
                //fprintf(stderr, "elementKey: %s\n iteration:
%ld\n", elements[i].key, i);
                if(isM == 1){
                        if(clock_gettime(CLOCK_MONOTONIC, &start) ==
-1){
                                fprintf(stderr, "Error in starting
the montonic time.\n");
                                exit(1);
                        }

pthread_mutex_lock(&mutexLocks[hashValues[i]]);
                        if(clock_gettime(CLOCK_MONOTONIC, &stop) ==
-1){
                                fprintf(stderr, "Error in ending the
montonic time.\n");
                                exit(1);
                        }

                        timeDiffSec = (stop.tv_sec - start.tv_sec) *
BILLION; //find the time difference in nanoseconds
                        timeDiffNS = stop.tv_nsec - start.tv_nsec;
                        timeDiff += timeDiffSec + timeDiffNS;

                        SortedList_insert(&listHeads[hashValues[i]],
&elements[i]);

pthread_mutex_unlock(&mutexLocks[hashValues[i]]);
                }
                else if(isS == 1){
                        if(clock_gettime(CLOCK_MONOTONIC, &start) ==
-1){
                                fprintf(stderr, "Error in starting
the montonic time.\n");
                                exit(1);
                        }
```

```
while(__sync_lock_test_and_set(&spinLocks[hashValues[i]], 1));
                        if(clock_gettime(CLOCK_MONOTONIC, &stop) ==
-1){
                                fprintf(stderr, "Error in ending the
montonic time.\n");
                                exit(1);
                        }

                        timeDiffSec = (stop.tv_sec - start.tv_sec) *
BILLION; //find the time difference in nanoseconds
                        timeDiffNS = stop.tv_nsec - start.tv_nsec;
                        timeDiff += timeDiffSec;
                        timeDiff += timeDiffNS;
                        SortedList_insert(&listHeads[hashValues[i]],
&elements[i]);

__sync_lock_release(&spinLocks[hashValues[i]]);
                }
                else{
                        SortedList_insert(&listHeads[hashValues[i]],
&elements[i]);
                }
        }

        int totalLength = 0;
        if(isM == 1){
                for(i = 0; i < numSubLists; i++){
                        if(clock_gettime(CLOCK_MONOTONIC, &start) ==
-1){
                                fprintf(stderr, "Error in starting
the montonic time.\n");
                                exit(1);
                        }

pthread_mutex_lock(&mutexLocks[hashValues[i]]);
                        if(clock_gettime(CLOCK_MONOTONIC, &stop) ==
-1){
                                fprintf(stderr, "Error in ending the
montonic time.\n");
                                exit(1);
                        }

                        timeDiffSec = (stop.tv_sec - start.tv_sec) *
BILLION; //find the time difference in nanoseconds
                        timeDiffNS = stop.tv_nsec - start.tv_nsec;
                        timeDiff += timeDiffSec;
                        timeDiff += timeDiffNS;

                        int length =
SortedList_length(&listHeads[hashValues[i]]);
```

```
                        if(length < 0){
                                fprintf(stderr, "Error in the length
of Sublist: %ld", i);
                                exit(1);
                        }
                        totalLength += length;

pthread_mutex_unlock(&mutexLocks[hashValues[i]]);
                }
        }
        else if(isS == 1){
                for(i = 0; i < numSubLists; i++){
                        if(clock_gettime(CLOCK_MONOTONIC, &start) ==
-1){
                                fprintf(stderr, "Error in starting
the montonic time.\n");
                                exit(1);
                        }

while(__sync_lock_test_and_set(&spinLocks[hashValues[i]], 1));
                        if(clock_gettime(CLOCK_MONOTONIC, &stop) ==
-1){
                                fprintf(stderr, "Error in ending the
montonic time.\n");
                                exit(1);
                        }

                        timeDiffSec = (stop.tv_sec - start.tv_sec) *
BILLION; //find the time difference in nanoseconds
                        timeDiffNS = stop.tv_nsec - start.tv_nsec;
                        timeDiff += timeDiffSec;
                        timeDiff += timeDiffNS;

                        int length =
SortedList_length(&listHeads[hashValues[i]]);
                        if(length < 0){
                                fprintf(stderr, "Error in the length
of Sublist: %ld", i);
                                exit(1);
                        }
                        totalLength += length;


__sync_lock_release(&spinLocks[hashValues[i]]);
                }
        }

        if(totalLength < 0){
                fprintf(stderr, "Error with the length in threadFunc.
\n");
```

```
                exit(2);
        }



        SortedListElement_t* e;
        for(i = startIndex; i < startIndex + numIterations; i++){
                if(isM == 1){
                        if(clock_gettime(CLOCK_MONOTONIC, &start) ==
-1){
                                fprintf(stderr, "Error in starting
the montonic time.\n");
                                exit(1);
                        }

pthread_mutex_lock(&mutexLocks[hashValues[i]]);
                        if(clock_gettime(CLOCK_MONOTONIC, &stop) ==
-1){
                                fprintf(stderr, "Error in ending the
montonic time.\n");
                                exit(1);
                        }

                        timeDiffSec = (stop.tv_sec - start.tv_sec) *
BILLION; //find the time difference in nanoseconds
                        timeDiffNS = stop.tv_nsec - start.tv_nsec;
                        timeDiff += timeDiffSec;
                        timeDiff += timeDiffNS;
                        e =
SortedList_lookup(&listHeads[hashValues[i]], elements[i].key);

                        if(e != NULL){
                                if(SortedList_delete(e) != 0){
                                        fprintf(stderr, "Error with
deleting an element.\n");
                                        exit(2);
                                }
                        }
                        else{
                                fprintf(stderr, "Error with looking
up the element.\n");
                                exit(2);
                        }

pthread_mutex_unlock(&mutexLocks[hashValues[i]]);
                }
                else if(isS == 1){
                        if(clock_gettime(CLOCK_MONOTONIC, &start) ==
-1){
                                fprintf(stderr, "Error in starting
```

```
the montonic time.\n");
                                exit(1);
                        }

while(__sync_lock_test_and_set(&spinLocks[hashValues[i]], 1));
                        if(clock_gettime(CLOCK_MONOTONIC, &stop) ==
-1){
                                fprintf(stderr, "Error in ending the
montonic time.\n");
                                exit(1);
                        }

                        timeDiffSec = (stop.tv_sec - start.tv_sec) *
BILLION; //find the time difference in nanoseconds
                        timeDiffNS = stop.tv_nsec - start.tv_nsec;
                        timeDiff += timeDiffSec;
                        timeDiff += timeDiffNS;

                        e =
SortedList_lookup(&listHeads[hashValues[i]], elements[i].key);

                        if(e != NULL){
                                if(SortedList_delete(e) != 0){
                                        fprintf(stderr, "Error with
deleting an element.\n");
                                        exit(2);
                                }
                        }
                        else{
                                fprintf(stderr, "Error with looking
up the element.\n");
                                exit(2);
                        }


__sync_lock_release(&spinLocks[hashValues[i]]);
                }
                else{
                        e =
SortedList_lookup(&listHeads[hashValues[i]], elements[i].key);

                }
        }
        return NULL;
}


void printToCSV(char* yieldType, char* syncType, long numOps, long
threadTime, long avgTime, long mutexTime){
        fprintf(stdout, "list-%s-%s,%ld,%ld,%ld,%ld,%ld,%ld,%ld\n",
```

```
yieldType, syncType, numThreads, numIterations, numSubLists, numOps,
threadTime, avgTime, mutexTime);
}

char* getYieldType(){
        if(isI == 1){
                if(isD == 1){
                        if(isL == 1)
                                return "idl";
                        else
                                return "id";
                }
                else
                        if(isL == 1)
                                return "il";
                        else
                                return "i";
        }
        else if(isD == 1){
                if(isL == 1)
                        return "dl";
                else
                        return "d";
        }
        else if(isL == 1){
                return "l";
        }
        else{
                return "none";
        }
}

char* getSyncType(){
        if(isS == 1){
                return "s";
        }
        else if(isM == 1){
                return "m";
        }
        else{
                return "none";
        }
}



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

        signal(SIGSEGV, segfaultHandler);
```

```
        isM--;
        isS--;

    isI--;
        isD--;
    isL--;

    char inputLock;

    int c;
        int isThread = 1;
        isThread--;

    int isIteration = 1;
        isIteration--;

        int isSubLists = 1;
        isSubLists--;

        isYield = 1;
        isYield--;

        int isSync = 1;
        isSync--;


        while(1){
                int option_index = 0;

                static struct option long_options[] = {
                        {"threads", required_argument, 0, 0},
                        {"iterations", required_argument, 0, 0},
                        {"yield", required_argument, 0, 0},
           {"sync", required_argument, 0, 0},
                        {"lists", required_argument, 0, 0},
                        {0,     0,      0,      0}
                };
                c = getopt_long(argc, argv, "", long_options,
&option_index);
                if( c == -1) break;
                switch(c){
                        case 0:

if(strcmp(long_options[option_index].name, "threads") == 0){
                                        isThread = 1;
                                        numThreads = atoi(optarg);
                                        if(numThreads <= 0)
numThreads = 1;
                                }
                                else
```

```
if(strcmp(long_options[option_index].name, "iterations") == 0){
                                        isIteration = 1;
                                        numIterations =
atoi(optarg);
                                        if(numIterations <= 0)
numIterations = 1;
                                }
                                else
if(strcmp(long_options[option_index].name, "lists") == 0){
                                        isSubLists = 1;
                                        numSubLists = atoi(optarg);
                                        if(numSubLists <= 0)
numSubLists = 1;
                                }
                                else
if(strcmp(long_options[option_index].name, "yield") == 0){
                                        isYield = 1;
                                        for(size_t i = 0; i <
strlen(optarg); i++){
                                                if(optarg[i] ==
'i'){
                                                        opt_yield |
= INSERT_YIELD;
                                                        isI = 1;
                                                }
                                                else if(optarg[i]
== 'd'){
                                                        opt_yield |
= DELETE_YIELD;
                                                        isD = 1;
                                                }
                                                else if(optarg[i]
== 'l'){
                                                        opt_yield |
= LOOKUP_YIELD;
                                                        isL = 1;
                                                }
                                                else{

fprintf(stderr, "Incorrect arguments given for the --yield option.
Please only give 'i', 'd', 'l', or some combination as the argument.
\n");
                                                exit(1);
                                                }
                                        }
                }
                                else
if(strcmp(long_options[option_index].name, "sync") == 0){
                                        isSync = 1;
                                        inputLock = *optarg;
```

```
                                        switch(inputLock){
                                                case 'm':
                                                        isM = 1;
                                                        /
*if(pthread_mutex_init(&mutex, NULL) != 0){

fprintf(stderr, "Unable to initialize mutex lock.\n");

exit(1);
                                                        }*/
                                                        break;
                                                case 's':
                                                        isS = 1;
                                                        break;
                                                default:

fprintf(stderr, "Incorrect arguments given for the --sync option.
Please only give 'm', 's', or 'c' as the argument.\n");
                                                        exit(1);
                                                        break;
                                                }
                                        }
                                        break;
                                default:
                                        fprintf(stderr, "Error: Please only
use arguments --threads and --iterations\n");
                                        exit(1);
                                        break;
                        }
                }

                long numElements = numThreads * numIterations;
                //fprintf(stderr, "here\n");


                srand(time(NULL)); //creates new seed of random number
generator


                initSubLists();
                if(isS==1 || isM==1) initLocks();
                initElements(numElements); //initializes the pool of elements
                createKeys(numElements); //creates the random keys for each
element
                initKeyList(numElements); //initializes the list of hash
values for each element

                struct timespec start, stop;

                //check that getting the monotonic time doesn't cause any
```

```
problems
        if(clock_gettime(CLOCK_MONOTONIC, &start) == -1){
                fprintf(stderr, "Error in starting the montonic time.
\n");
                exit(1);
        }


        //allocate space to create the pthread_t objects
        pthread_t *threads = malloc((sizeof(pthread_t)) * numThreads);
        if(threads == NULL){
                fprintf(stderr, "Error in allocating memory for the
threads.\n");
                exit(1);
        }

        int threadID[numThreads];
        for(long i = 0; i < numThreads; i++){
                threadID[i] = i;
                //fprintf(stderr, "%ld\n", i);
                int checkThread = pthread_create(&threads[i], NULL,
threadFunc, &threadID[i]);
                if(checkThread != 0){
                        fprintf(stderr, "Error in creating thread:
%s\n", strerror(errno));
                        exit(1);
                }
        }



        for(int i = 0; i < numThreads; i++){
                int checkJoin = pthread_join(threads[i], NULL);
                if(checkJoin != 0){
                        fprintf(stderr, "Error in joining threads
together.\n");
                        exit(1);
                }
        }



        if(clock_gettime(CLOCK_MONOTONIC, &stop) == -1){
                fprintf(stderr, "Error in ending the montonic time.
\n");
                exit(1);
        }

        long threadTime;
```

```
        long timeDiffSec = (stop.tv_sec - start.tv_sec) * BILLION; //
find the time difference in nanoseconds
        long timeDiffNS = stop.tv_nsec - start.tv_nsec;
        threadTime = timeDiffSec + timeDiffNS;



        long operations = numThreads * numIterations * 3;
        long numLockOps = (2*numIterations+1) * numThreads;
        long avgTime = threadTime / operations;
        long lockContention = timeDiff / numLockOps;


        char* yieldType = getYieldType();
        char* syncType = getSyncType();
        printToCSV(yieldType, syncType, operations, threadTime,
avgTime, lockContention);

        free(elements);
        free(listHeads);
        if(isM == 1)
                free(mutexLocks);
        else if(isS == 1)
                free(spinLocks);
        free(hashValues);

        return 0;

}
```