**lab4**

#define AIO\_PORT 1

#define GPIO\_PIN 60

volatile sig\_atomic\_t flag = 1;

/\* global variables \*/

float sample\_rate = 1.0;

int logfd = -1;

int stop = 0;

int off = 0;

int tflag = 'F';

FILE\* f\_logfd;

//4c

char\* id;

char\* host;

int port = 0;

int sockfd = 1;

SSL\* ssl\_client = 0;

void sig\_handler(int signum) { if (signum == SIGINT) flag = 0;}

void button\_handler() { flag = 0; }

int stoi(char\* str) {

char\* endptr = NULL;

int ret = (int) strtol(str, &endptr, 10);

if (endptr == str || ret < 0)

return -1;

return ret;

}

void \_shutdown() {

/\* get current time \*/

struct timeval curr;

if (gettimeofday(&curr, NULL) < 0) {

fprintf(stderr, "Error: gettimeofday()\n");

exit(1);

}

char timebuf[64];

strftime(timebuf,64,"%T", localtime(&(curr.tv\_sec)));

if (f\_logfd) fprintf(f\_logfd, "%s SHUTDOWN\n", timebuf);

char buf[256];

sprintf(buf, "%s SHUTDOWN\n", timebuf);

SSL\_write(ssl\_client, buf, strlen(buf));

exit(0);

}

/\* convert sensor temperature \*/

float c\_to\_f(float cdegree) { return cdegree\*(9.0/5.0)+32.0; }

/\* process input from terminal \*/

void execute(char\* command, int nbytes) {

if (nbytes <= 0) return;

char cmd[256];

strncpy(cmd, command, strlen(command));

int i = 0;

int j = 0;

for (j = 0; j < nbytes; j++) {

if (command[j] != '\n') continue;

// report

cmd[j] = '\0';

if (logfd != -1) dprintf(logfd, "%s\n", cmd+i);

// process

if (strncmp(cmd+i, "SCALE=F", 7) == 0)

tflag = 'F';

else if (strncmp(cmd+i, "SCALE=C", 7) == 0)

tflag = 'C';

else if (strncmp(cmd+i, "STOP", 4) == 0) //stop generating reports but continue processing input commands. If the program is not generating reports, merely log receipt of the command

stop = 1;

else if (strncmp(cmd+i, "START", 5) == 0) //if stopped, resume generating reports. If the program is not stopped, merely log receipt of the command.

stop = 0;

else if (strncmp(cmd+i, "OFF", 3) == 0) { //like pressing the button, output (and log) a time-stamped SHUTDOWN message and exit

off = 1;

\_shutdown();

}

else if (strlen(cmd) >= 3) { //change the number of seconds between reporting intervals. It is acceptable if this command does not take effect until after the next report

char prefix\_l[] = "LOG";

int lg = 1;

int k;

for (k = 0; k < 3; k++) {

if (cmd[i+k] != prefix\_l[k])

lg = 0;

}

char prefix\_p[] = "PERIOD=";

int prd = 1;

if (lg == 0 && strlen(cmd) >= 7) {

for (k = 0; k < 7; k++)

if (cmd[i+k] != prefix\_p[k])

prd = 0;

if (prd == 1) {

int ret = stoi(cmd+i+7);

if (ret < 0)

fprintf(stderr, "Please provide a valid period\n");

else

sample\_rate = ret;

}

else {

fprintf(stderr, "Please provide a valid command\n");

exit(1);

}

}

}

else {

fprintf(stderr, "Please provide a valid command: %s\n", cmd+i);

exit(1);

}

i = j+1;

}

}

void tcp\_connection() {

/\* check host name and id \*/

if (!id || !host) {

fprintf(stderr, "Error: no host address/id\n");

exit(1);

}

/\* Create a socket point \*/

struct sockaddr\_in serv\_addr;

struct hostent \*server;

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd < 0) {

perror("Error: opening socket");

exit(2);

}

server = gethostbyname(host);

if (server == NULL) {

fprintf(stderr,"Error: no such host\n");

exit(2);

}

memset((char \*) &serv\_addr, 0, sizeof(serv\_addr));

serv\_addr.sin\_family = AF\_INET;

memcpy((char \*)&serv\_addr.sin\_addr.s\_addr, (char \*)server->h\_addr, server->h\_length);

serv\_addr.sin\_port = htons(port);

/\* Now connect to the server \*/

if (connect(sockfd, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr)) < 0) {

perror("Error: connecting");

exit(2);

}

}

void ssl\_connection() {

SSL\_library\_init();

SSL\_load\_error\_strings();

OpenSSL\_add\_all\_algorithms();

SSL\_CTX\* ssl\_context = SSL\_CTX\_new(TLSv1\_client\_method());

if (!ssl\_context) {

fprintf(stderr, "Error: ssl context\n");

exit(2);

}

ssl\_client = SSL\_new(ssl\_context);

if (!ssl\_client) {

fprintf(stderr, "Error: ssl client\n");

exit(2);

}

if (!SSL\_set\_fd(ssl\_client, sockfd)) {

fprintf(stderr, "Error: ssl\_set\_fd\n");

exit(2);

}

if (SSL\_connect(ssl\_client) != 1) {

fprintf(stderr, "Error: connection fail\n");

exit(2);

}

/\* send (and log) an ID terminated with a newline \*/

if (f\_logfd) fprintf(f\_logfd, "ID=%s\n", id);

char buf[256];

sprintf(buf, "ID=%s\n", id);

SSL\_write(ssl\_client, buf, strlen(buf));

}

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

{

/\* declare variables \*/

int opt = 0;

static struct option long\_options[] = {

{ "period", required\_argument, NULL, 'p' },

{ "scale", required\_argument, NULL, 's' },

{ "log", required\_argument, NULL, 'l' },

{ "id", required\_argument, NULL, 'i' },

{ "host", required\_argument, NULL, 'h' },

{ 0,0,0,0 }

};

while (1) {

opt = getopt\_long(argc, argv, "", long\_options, NULL);

if (opt == -1)

break;

switch(opt) {

case 'p': {

sample\_rate = stoi(optarg);

if (sample\_rate < 0) {

fprintf(stderr, "Error: invalid period\n");

exit(1);

}

break;

}

case 's': {

if (strcmp(optarg,"C") == 0)

tflag = 'C';

break;

}

case 'l': {

logfd = open(optarg, O\_CREAT|O\_WRONLY|O\_APPEND, S\_IRWXU);

if (logfd == -1) {

fprintf(stderr, "%s\n", "Error: opening log file");

exit(1);

}

f\_logfd = fopen(optarg, "a+");

if (f\_logfd == NULL) {

fprintf(stderr, "Error: %s\n", strerror(errno));

exit(1);

}

break;

}

case 'i': {

if (strlen(optarg) != 9) {

fprintf(stderr, "Error: Please provide a valid id\n");

exit(1);

}

id = malloc(10);

strcpy(id, optarg);

id[9] = '\0';

break;

}

case 'h': {

if (strlen(optarg) == 0) {

fprintf(stderr, "Please provide a valid host address\n");

exit(1);

}

host = optarg;

break;

}

default: {

fprintf(stderr, "%s\n", "Error: invalid option");

exit(1);

}

}

}

/\* port number \*/

port = stoi(argv[argc-1]);

/\* install signal handler \*/

signal(SIGINT, sig\_handler);

/\* initialize mraa aio and gpio \*/

uint16\_t value = 0;

mraa\_aio\_context aio;

mraa\_gpio\_context gpio;

aio = mraa\_aio\_init(AIO\_PORT);

gpio = mraa\_gpio\_init(GPIO\_PIN);

if (aio == NULL || gpio == NULL) {

fprintf(stderr, "Error: mraa initialization\n");

exit(1);

}

mraa\_gpio\_dir(gpio, MRAA\_GPIO\_OUT);

mraa\_gpio\_isr(gpio, MRAA\_GPIO\_EDGE\_RISING, (void\*)&button\_handler, NULL);

tcp\_connection();

ssl\_connection();

/\* set up polling \*/

struct pollfd p[1];

p[0].fd = sockfd;

p[0].events = POLLIN | POLLHUP | POLLERR; // data to read/hng up/error condition

/\* get start time \*/

struct timeval start\_tv;

if (gettimeofday(&start\_tv, NULL) < 0) {

fprintf(stderr, "Error: gettimeofday()\n");

exit(1);

}

while (flag == 1) {

/\* get current time \*/

struct timeval curr;

if (gettimeofday(&curr, NULL) < 0) {

fprintf(stderr, "Error: gettimeofday()\n");

exit(1);

}

long count\_down = curr.tv\_sec - start\_tv.tv\_sec;

/\* sensor read \*/

if (count\_down >= sample\_rate) {

char timebuf[64];

strftime(timebuf,64,"%T", localtime(&(curr.tv\_sec)));

/\* read temperature \*/

value = mraa\_aio\_read(aio);

float R = (1023.0/value - 1.0)\*100000;

float temp = 1.0/(log(R/100000.0)/4275.0+1/298.15)-273.15;

if (tflag == 'F') temp = c\_to\_f(temp);

/\* report \*/

if (stop == 0) {

char buf[256];

sprintf(buf, "%s %.1f\n", timebuf, temp);

SSL\_write(ssl\_client, buf, strlen(buf));

if (logfd != -1) fprintf(f\_logfd, "%s %.1f\n", timebuf, temp);

}

start\_tv = curr; // update current time

}

/\* polling: read from terminal \*/

int poll\_status = poll(p, 1, 512);

if (poll\_status > 0) {

char command[256];

if ((p[0].revents&POLLIN) == POLLIN) {

int r = SSL\_read(ssl\_client, command, strlen(command));

if (r < 0)

fprintf(stderr, "Error: read()\n");

execute(command, r);

}

}

if (off == 1) \_shutdown();

}

\_shutdown();

close(logfd);

mraa\_aio\_close(aio);

mraa\_gpio\_close(gpio);

SSL\_shutdown(ssl\_client);

SSL\_free(ssl\_client);

return 0;

}

**lab3a**

/\* global variables \*/

int fd = 0;

int bcount = 0;

int icount = 0;

int bsize = 0;

int isize = 0;

int ipergroup = 0;

\_\_u32 fbcount = 0;

\_\_u32 ficount = 0;

\_\_u32 fb\_bitmap = 0;

\_\_u32 fi\_bitmap = 0;

\_\_u32 itable = 0;

/\* calculate the block offset given the block id \*/

int blk\_offset(int blk\_id) {return 1024 + (blk\_id-1)\*bsize;}

void \_error(const char\* str) {

fprintf(stderr, "%s\n", str);

close(fd);

exit(2);

}

void superblock() {

/\* superblock has a size of 1024 at offset 1024 \*/

struct ext2\_super\_block s;

if (pread(fd, &s, 1024, 1024) < 0)

\_error("pread() fails\n");

if (s.s\_magic != 0xEF53)

\_error("Magic number can't match\n");

bcount = s.s\_blocks\_count;

icount = s.s\_inodes\_count;

/\* bsize = EXT2\_MIN\_BLOCK\_SIZE << s\_log\_block\_size \*/

bsize = EXT2\_MIN\_BLOCK\_SIZE << s.s\_log\_block\_size;

isize = s.s\_inode\_size;

ipergroup = s.s\_inodes\_per\_group;

fprintf(stdout, "%s,%d,%d,%d,%d,%d,%d,%d\n", "SUPERBLOCK",

bcount,icount,bsize,isize,

s.s\_blocks\_per\_group,

ipergroup,

s.s\_first\_ino);

}

void group() {

struct ext2\_group\_desc g;

/\* group descriptor is located after superblock \*/

if (pread(fd, &g, sizeof(struct ext2\_group\_desc), 1024+bsize) < 0)

\_error("pread() fails\n");

fbcount = g.bg\_free\_blocks\_count;

ficount = g.bg\_free\_inodes\_count;

fb\_bitmap = g.bg\_block\_bitmap;

fi\_bitmap = g.bg\_inode\_bitmap;

itable = g.bg\_inode\_table;

/\* Produce a line for the single group with nine fields \*/

fprintf(stdout, "%s,%d,%d,%d,%d,%d,%d,%d,%d\n", "GROUP", 0, bcount, icount,

g.bg\_free\_blocks\_count,

g.bg\_free\_inodes\_count,

fb\_bitmap, fi\_bitmap, itable);

}

void bfree() {

/\* Scan the free block bitmap \*/

/\* bsize is represented in bytes and each bit represents a block \*/

/\* there are in total bcount/8 bytes \*/

for (int i = 0; i < bcount/8; i++) {

char bitmap\_onebyte;

if (pread(fd, &bitmap\_onebyte, 1, fb\_bitmap\*bsize+i) < 0)

\_error("pread() fails\n");

/\* 1 byte = 8 bits \*/

for (int j = 0; j < 8; j++) {

/\* 1 means "used" and 0 "free/available" \*/

if ((bitmap\_onebyte & (1 << j)) == 0)

fprintf(stdout, "%s,%d\n", "BFREE", i\*8+j+1);

}

}

}

void ifree() {

/\* Scan the free inode bitmap, similar to block bitmap \*/

for (int i = 0; i < bcount/8; i++) {

char bitmap\_onebyte;

if (pread(fd, &bitmap\_onebyte, 1, fi\_bitmap\*bsize+i) < 0)

\_error("pread() fails\n");

for (int j = 0; j < 8; j++) {

if ((bitmap\_onebyte & (1 << j)) == 0)

fprintf(stdout, "%s,%d\n", "IFREE", i\*8+j+1);

}

}

}

void inode() {

struct ext2\_inode inod;

for (int i = 0; i < ipergroup; i++) {

if (pread(fd, &inod, sizeof(struct ext2\_inode), itable\*bsize+i\*isize) < 0)

\_error("pread() fails\n");

if (inod.i\_links\_count == 0 || inod.i\_mode == 0)

continue;

/\* file type \*/

char file\_type = '?';

if ((inod.i\_mode & 0xF000) == 0x8000) file\_type = 'f';

if ((inod.i\_mode & 0xF000) == 0xA000) file\_type = 's';

if ((inod.i\_mode & 0xF000) == 0x4000) file\_type = 'd';

/\* format time \*/

time\_t ctime = (time\_t) inod.i\_ctime;

time\_t mtime = (time\_t) inod.i\_mtime;

time\_t atime = (time\_t) inod.i\_atime;

char ctimebuf[50];

char mtimebuf[50];

char atimebuf[50];

strftime (ctimebuf,50,"%D %T",gmtime(&ctime));

strftime (mtimebuf,50,"%D %T",gmtime(&mtime));

strftime (atimebuf,50,"%D %T",gmtime(&atime));

/\* print information \*/

fprintf(stdout, "%s,%d,%c,%o,%d,%d,%d,%s,%s,%s,%d,%d,",

"INODE", i+1, file\_type, inod.i\_mode&0xFFF, inod.i\_uid, inod.i\_gid, inod.i\_links\_count,

ctimebuf, mtimebuf, atimebuf, inod.i\_size, inod.i\_blocks);

/\* print block pointers \*/

for (int j = 0; j < 14; j++) {

fprintf(stdout, "%d,", inod.i\_block[j]);

}

if ((file\_type == 'd' || file\_type == 'f')) //?? If the file length is greater than 60 bytes, print out the fifteen block numbers

fprintf(stdout, "%d\n", inod.i\_block[14]);

else if (inod.i\_size <= 60) // Symbolic link and contains <= 60 bytes

fprintf(stdout, "\n");

}

}

/\* Directories are stored as data block and referenced by an inode. \*/

void dirEnt() {

struct ext2\_inode inod;

for (int i = 0; i < ipergroup; i++) {

if (pread(fd, &inod, sizeof(struct ext2\_inode), itable\*bsize+i\*isize) < 0) // parent inode

\_error("pread() fails\n");

if (inod.i\_links\_count == 0 || inod.i\_mode == 0 || (inod.i\_mode & 0xF000) != 0x4000)

continue;

/\* Scan every data block \*/

for (int j = 0; j < 12; j++) {

struct ext2\_dir\_entry d;

int blkptr = inod.i\_block[j];

if (inod.i\_block[j] == 0)

continue;

// fprintf(stdout, "%s\n", "Enter dirEnt"); //140

/\* Each directory inode is a linked list of directory entries of block size \*/

// k = offset

for (int k = 0; k < bsize; ) {

if (pread(fd, &d, sizeof(struct ext2\_dir\_entry), k+blk\_offset(blkptr)) < 0)

\_error("pread() fails\n");

if (d.inode == 0){

k += d.rec\_len;

continue;

}

char d\_name[EXT2\_NAME\_LEN+1];

memcpy(d\_name, d.name, d.name\_len);

d\_name[d.name\_len] = '\0';

fprintf(stdout, "%s,%d,%d,%d,%d,%d,\'%s\'\n",

"DIRENT", i+1, k, d.inode, d.rec\_len, d.name\_len, d\_name);

k += d.rec\_len;

}

}

}

}

void indir\_rec(int inode, int level, int log\_offset, int blk\_id, int isDirectory) {

if (level == 0)

return;

int referenced\_blk[bsize/4]; // contain a block of block\_ids

// eg. a block of 1024B stores 256 ids.

for (int i = 0; i < bsize/4; i++) {

/\* read each block pointer in the indirect blocks \*/

if (pread(fd, &referenced\_blk[i], 4, blk\_offset(blk\_id)+4\*i) < 0)

\_error("pread() fails\n");

if (referenced\_blk[i] != 0) {

/\* print information \*/

fprintf(stdout, "%s,%d,%d,%d,%d,%d\n", "INDIRECT", inode, level,

log\_offset, blk\_id, referenced\_blk[i]);

/\* print information for directory entries \*/

if (isDirectory) {

struct ext2\_dir\_entry d;

for (int k = 0; k < bsize; ) {

if (pread(fd, &d, sizeof(struct ext2\_dir\_entry), blk\_offset(referenced\_blk[i])+k) < 0)

\_error("pread() fails\n");

if (d.inode == 0){

k += d.rec\_len;

continue;

}

char d\_name[EXT2\_NAME\_LEN+1];

memcpy(d\_name, d.name, d.name\_len);

d\_name[d.name\_len] = '\0';

fprintf(stdout, "%s,%d,%d,%d,%d,%d,\'%s\'\n",

"DIRENT", inode, blk\_offset(referenced\_blk[i])+k, d.inode, d.rec\_len, d.name\_len, d\_name);

k += d.rec\_len;

}

}

/\* recursion \*/

if (level > 1) {

indir\_rec(inode, level-1, log\_offset, referenced\_blk[i], isDirectory);

if (level == 2) log\_offset += bsize/4;

if (level == 3) log\_offset += bsize/4\*bsize/4;

}

}

log\_offset++;

}

}

void indirect() {

struct ext2\_inode inod;

for (int i = 0; i < ipergroup; i++) {

if (pread(fd, &inod, sizeof(struct ext2\_inode), itable\*bsize+i\*isize) < 0) // parent inode

\_error("pread() fails\n");

if (inod.i\_links\_count == 0 || inod.i\_mode == 0)

continue;

if ((inod.i\_mode & 0xF000) != 0x4000 && (inod.i\_mode & 0xF000) != 0x8000) // file or directory inode

continue;

int isDirectory = 0;

if ((inod.i\_mode & 0xF000) == 0x4000)

isDirectory = 1;

/\* scan the 12~15th blocks \*/

for (int j = 12; j < 15; j++) {

// A value of 0 in this array effectively terminates it

if (inod.i\_block[j] == 0)

break;

if (j==12) indir\_rec(i+1, 1, 12, inod.i\_block[j], isDirectory);// first level

if (j==13) indir\_rec(i+1, 2, 12+bsize/4, inod.i\_block[j], isDirectory);// second level

if (j==14) indir\_rec(i+1, 3, 12+bsize/4+bsize/4\*bsize/4, inod.i\_block[j], isDirectory);// third level

}

}

}

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

{

if (argc != 2) {

fprintf(stderr, "Wrong number of arguments\n");

close(fd);

exit(1);

}

fd = open(argv[1], O\_RDONLY);

if (fd < 0) {

fprintf(stderr, "Can't open file image\n");

close(fd);

exit(1);

}

superblock();

group();

bfree();

ifree();

inode();

dirEnt();

indirect();

close(fd);

return 0;

}

**lab3b**

import sys

import csv

# define constant

NONE = 0

FREE = 1

USED = 2

RESERVED = 3

DUPLICATE = 4

INVALID = 6

# error

def \_error(string):

sys.stderr.write(string+"\n")

sys.exit(1)

# my data structures

total\_blocks = 0

total\_inodes = 0

bsize = 1024

isize = 0

first\_data\_block = 0

first\_inode = 0

exit = 0

block\_state = []

inode\_state = []

parent = []

linkcount = []

refcount = []

dirent = []

free\_inode\_bm = []

alloc\_inode\_bm = []

free\_blk\_bm = []

alloc\_blk\_bm = []

# block consistency audits

def validBlk(blk):

if blk < 0 or blk > total\_blocks:

return False

return True

def isReserved(blk):

if validBlk(blk) == True and blk < first\_data\_block:

return True

return False

def block\_audit():

#print free\_blk\_bm

#print alloc\_blk\_bm

for b in range(first\_data\_block, total\_blocks):

if free\_blk\_bm[b] == 0 and alloc\_blk\_bm[b] == 0:

print "UNREFERENCED BLOCK",b

exit = 2

if free\_blk\_bm[b] == 1 and alloc\_blk\_bm[b] == 1:

print "ALLOCATED BLOCK", b, "ON FREELIST"

exit = 2

# i-node allocation audits

def validInode(inode):

if inode >=2 and inode < total\_inodes:

return True

return False

def inode\_audit():

for i in range(0, total\_inodes):

if free\_inode\_bm[i] == 1 and alloc\_inode\_bm[i] == 1:

print "ALLOCATED INODE", i, "ON FREELIST"

exit = 2

elif free\_inode\_bm[i] == 0 and alloc\_inode\_bm[i] == 0:

print "UNALLOCATED INODE", i, "NOT ON FREELIST"

exit = 2

elif alloc\_inode\_bm[i] == 1 and refcount[i] != linkcount[i]:

print "INODE", i, "HAS", refcount[i], "LINKS BUT LINKCOUNT IS", linkcount[i]

exit = 2

# directory consistency audits

def checkParent(d):

if d == 2: #root

return 2

for i in range(0, len(dirent)):

if dirent[i]["ref"] == d and dirent[i]["name"] != "\'..\'" and dirent[i]["parent"] != d:

return dirent[i]["parent"]

return -1

def dir\_audit():

for i in range(0, len(dirent)):

d = dirent[i]

if validInode(d["ref"]) == False:

print "DIRECTORY INODE", d["parent"], "NAME " + d["name"] + " INVALID INODE", d["ref"]

exit = 2

elif alloc\_inode\_bm[d["ref"]] == 0:

print "DIRECTORY INODE", d["parent"], "NAME " + d["name"] + " UNALLOCATED INODE", d["ref"]

exit = 2

elif d["name"] == "\'.\'" and d["parent"] != d["ref"]:

print "DIRECTORY INODE", d["parent"], "NAME \'.\' LINK TO INODE", d["ref"], "SHOULD BE", d["parent"]

exit = 2

elif d["name"] == "\'..\'" and d["ref"] != checkParent(d["parent"]):

print "DIRECTORY INODE", d["parent"], "NAME \'..\' LINK TO INODE", d["ref"], "SHOULD BE", checkParent(d["parent"])

exit = 2

def level\_str(level):

if level == 3 or level == 26:

return "TRIPLE INDIRECT "

elif level == 2 or level == 25:

return "DOUBLE INDIRECT "

elif level == 1 or level == 24:

return "INDIRECT "

else:

return ""

def offset(level):

if level == 3 or level == 26:

return 12+bsize/4+bsize/4\*bsize/4

elif level == 2 or level == 25:

return 12+bsize/4

elif level == 1 or level == 24:

return 12

else:

return 0

# main function

if \_\_name\_\_== "\_\_main\_\_":

# check syntax

if len(sys.argv) != 2:

\_error("Wrong number of argument")

try:

fd = open(sys.argv[1], "r")

except IOError:

\_error("Can't open the file")

# collect data from csv

with open(sys.argv[1],"rb") as csvfile:

linereader = csv.reader(csvfile)

for line in linereader:

if line[0] == "SUPERBLOCK":

total\_blocks = int(line[1])

total\_inodes = int(line[2])

bsize = int(line[3])

isize = int(line[4])

first\_inode = int(line[7])

for i in range(0, total\_blocks+1):

free\_blk\_bm.append(0)

alloc\_blk\_bm.append(0)

block\_state.append(NONE)

parent.append({ "inode":0, "level":0 })

for i in range(0, total\_inodes+1):

free\_inode\_bm.append(0)

alloc\_inode\_bm.append(0)

linkcount.append(0)

refcount.append(0)

for i in range(0, first\_inode):

alloc\_inode\_bm[i] = 1

elif line[0] == "GROUP":

first\_data\_block = int(line[8]) + isize\*total\_inodes/bsize

for i in range(0, first\_data\_block):

alloc\_blk\_bm[i] = 1

elif line[0] == "BFREE":

free\_blk\_bm[int(line[1])] = 1

block\_state[int(line[1])] = FREE

elif line[0] == "IFREE":

free\_inode\_bm[int(line[1])] = 1

elif line[0] == "INODE":

# skip symlinks with file size < 60B

if line[2] == "s" and int(line[10]) < 60:

continue

# allocated inode has nonzero link count and nonzero mode

if int(line[6]) != 0 and int(line[4]) != 0:

alloc\_inode\_bm[int(line[1])] = 1

linkcount[int(line[1])] = int(line[6])

# check block pointers in each inode

for i in range(12,27):

if int(line[i]) == 0:

continue

#INVALID

if validBlk(int(line[i])) == False:

print "INVALID " + level\_str(i) + "BLOCK " + line[i] + " IN INODE " + line[1] + " AT OFFSET", offset(i)

exit = 2

#RESERVED

elif isReserved(int(line[i])) == True:

print "RESERVED " + level\_str(i) + "BLOCK " + line[i] + " IN INODE " + line[1] + " AT OFFSET", offset(i)

exit = 2

else:

#DUPLICATE

if block\_state[int(line[i])] == USED or block\_state[int(line[i])] == DUPLICATE:

block\_state[int(line[i])] = DUPLICATE

print "DUPLICATE " + level\_str(parent[int(line[i])]["level"]) + "BLOCK " + line[i] + " IN INODE " + parent[int(line[i])]["inode"] + " AT OFFSET", offset(parent[int(line[i])]["level"])

print "DUPLICATE " + level\_str(i) + "BLOCK " + line[i] + " IN INODE " + line[1] + " AT OFFSET", offset(i)

exit = 2

else:

block\_state[int(line[i])] = USED

alloc\_blk\_bm[int(line[i])] = 1

parent[int(line[i])] = { "inode":line[1], "level":i } #inode=last\_reference #level=24~26

elif line[0] == "DIRENT":

if validInode(int(line[3])) == True:

refcount[int(line[3])] += 1

dirent.append({ "parent":int(line[1]), "ref":int(line[3]), "offset":int(line[2]), "name":line[6] })

elif line[0] == "INDIRECT":

if validBlk(int(line[5])) == False:

print "INVALID " + level\_str(int(line[2])) + "INDIRECT BLOCK " + line[5] + " IN INODE " + line[1] + " AT OFFSET " + line[3]

exit = 2

elif isReserved(int(line[5])) == True:

print "RESERVED " + level\_str(int(line[2])) + "INDIRECT BLOCK " + line[5] + " IN INODE " + line[1] + " AT OFFSET " + line[3]

exit = 2

else:

if block\_state[int(line[5])] == USED or block\_state[int(line[5])] == DUPLICATE:

block\_state[int(line[5])] = DUPLICATE

print "DUPLICATE " + level\_str(parent[int(line[5])]["level"]) + "BLOCK " + line[5] + " IN INODE " + parent[int(line[5])]["inode"] + " AT OFFSET", offset(parent[int(line[5])]["level"])

print "DUPLICATE " + level\_str(i) + "BLOCK " + line[5] + " IN INODE " + line[1] + " AT OFFSET", line[3]

exit = 2

else:

block\_state[int(line[5])] = USED

alloc\_blk\_bm[int(line[5])] = 1

parent[int(line[5])] = { "inode" : line[1], "level" : int(line[2]) }

# report audits

block\_audit()

inode\_audit()

dir\_audit()

sys.exit(exit)

**lab2-add**

/\* global variables \*/

long long counter = 0;

int opt\_yield = 0;

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

static int sl = 0;

/\* basic add routine \*/

void add(long long \*pointer, long long value) {

long long sum = \*pointer + value;

if (opt\_yield){

sched\_yield();

}

\*pointer = sum;

}

/\* thread routine helper \*/

void thread\_add\_helper(int niterations, int s\_o, int x) {

for (int i = 0; i < niterations; i++) {

switch(s\_o) {

case 0: {

add(&counter, x);

break;

}

//mutex

case 'm': {

if (pthread\_mutex\_lock(&lock) != 0) {

fprintf(stderr, "pthread\_mutex\_lock(): %s\n", strerror(errno));

exit(1);

}

add(&counter, x);

if (pthread\_mutex\_unlock(&lock) != 0) {

fprintf(stderr, "pthread\_mutex\_unlock(): %s\n", strerror(errno));

exit(1);

}

break;

}

//spin lock

case 's': {

while (\_\_sync\_lock\_test\_and\_set(&sl, 1));

add(&counter, x);

\_\_sync\_lock\_release(&sl);

break;

}

//compare and swap

case 'c': {

int newv = 0;

int oldv = 0;

//A full memory barrier is created when this function is invoked.

do {

oldv = counter;

newv = oldv + x;

} while ((\_\_sync\_val\_compare\_and\_swap(&counter, oldv, newv)) != oldv);

break;

}

}

}

}

/\* thread routine \*/

void\* thread\_add(int thread\_args[]) {

//int args[2] = thread\_args;

int niterations = thread\_args[0];

int sync\_f = thread\_args[1];

thread\_add\_helper(niterations, sync\_f, 1);

thread\_add\_helper(niterations, sync\_f, -1);

return NULL;

}

/\* helper functions \*/

int strtoint(char\* optarg) {

char\* endptr = NULL;

int ret = (int) strtol(optarg, &endptr, 10);

if (endptr == optarg || ret < 0)

ret = 1;

return ret;

}

void diff(struct timespec\* s, struct timespec\* e, struct timespec\* res) {

if (e->tv\_nsec < s->tv\_nsec) {

res->tv\_sec = e->tv\_sec - s->tv\_sec - 1;

res->tv\_nsec = e->tv\_nsec+1000000000 - s->tv\_nsec;

}

else {

res->tv\_sec = e->tv\_sec - s->tv\_sec;

res->tv\_nsec = e->tv\_nsec - s->tv\_nsec;

}

fprintf(stdout, "start time: %ld\n", s->tv\_sec\*1000000000+s->tv\_nsec);

fprintf(stdout, "end time: %ld\n", e->tv\_sec\*1000000000+e->tv\_nsec);

}

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

/\* declare options \*/

static struct option long\_options[] = {

{ "iterations", optional\_argument, NULL, 'i' },

{ "threads", optional\_argument, NULL, 't' },

{ "sync", required\_argument, NULL, 's' },

{ "yield", no\_argument, NULL, 'y' },

{ 0,0,0,0 }

};

/\* declare variables\*/

int opt = 0;

int sync\_opt = 0;

char test\_name[20];

int niterations = 1;

int nthreads = 1;

int noperations = 0;

int t\_args[2];

struct timespec tp\_start\_main;

struct timespec tp\_end\_main;

long long rt\_ntime;

long long avg\_ntime;

int mutexflag = 0;

int spinflag = 0;

int cflag = 0;

while(1) {

opt = getopt\_long(argc, argv, "", long\_options, NULL);

if (opt == -1)

break;

switch(opt) {

/\* --iterations=# \*/

case 'i': {

if (!optarg)

nthreads = 1;

niterations = strtoint(optarg);

break;

}

/\* --threads=# \*/

case 't': {

if (!optarg)

nthreads = 1;

nthreads = strtoint(optarg);

break;

}

/\* --sync=# \*/

case 's': {

sync\_opt = optarg[0];

switch(sync\_opt) {

case 'm':

mutexflag = 1;

break;

case 's':

spinflag = 1;

break;

case 'c':

cflag = 1;

break;

default:

fprintf(stderr, "Unrecognized sync option %c\n", sync\_opt);

exit(1);

}

break;

}

/\* --yield \*/

case 'y': {

opt\_yield = 1;

break;

}

default: {

fprintf(stderr, "Unrecognized argument: %s\n", argv[optind-1]);

exit(1);

}

}

}

/\* test names \*/

if (opt\_yield) {

if (!mutexflag && !spinflag && !cflag)

strcpy(test\_name, "add-yield-none");

else if (mutexflag)

strcpy(test\_name, "add-yield-m");

else if (spinflag)

strcpy(test\_name, "add-yield-s");

else if (cflag)

strcpy(test\_name, "add-yield-c");

}

else {

if (!mutexflag && !spinflag && !cflag)

strcpy(test\_name, "add-none");

else if (mutexflag)

strcpy(test\_name, "add-m");

else if (spinflag)

strcpy(test\_name, "add-s");

else if (cflag)

strcpy(test\_name, "add-c");

}

pthread\_t threads[nthreads];

noperations = 2\*nthreads\*niterations;

t\_args[0] = niterations;

t\_args[1] = sync\_opt;

/\* get starting time \*/

if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_start\_main) < 0) {

fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

exit(1);

}

/\* create threads \*/

for (int i = 0; i < nthreads; i++) {

if (pthread\_create(&threads[i], NULL,(void\*)&thread\_add, &t\_args) != 0){

fprintf(stderr, "pthread\_create() error: %s\n", strerror(errno));

exit(1);

}

}

/\* join threads \*/

for (int i = 0; i < nthreads; i++) {

void\* retVal;

if (pthread\_join(threads[i], &retVal) != 0){

fprintf(stderr, "pthread\_join() error: %s\n", strerror(errno));

exit(1);

}

}

/\* get ending time \*/

if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_end\_main) < 0) {

fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

exit(1);

}

rt\_ntime = (tp\_end\_main.tv\_sec - tp\_start\_main.tv\_sec) \* 1000000000;

rt\_ntime += tp\_end\_main.tv\_nsec;

rt\_ntime -= tp\_start\_main.tv\_nsec;

if (noperations != 0)

avg\_ntime = rt\_ntime / noperations;

else

avg\_ntime = rt\_ntime;

/\* print results to csv \*/

fprintf(stdout, "%s,%d,%d,%d,%lld,%lld,%lld\n",

test\_name, nthreads, niterations, noperations, rt\_ntime, avg\_ntime, counter);

}

**lab2-list**

/\* global variables \*/

int opt\_yield = 0;

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

static int sl = 0;

static int niterations = 1;

static int nthreads = 1;

static int sync\_opt = 0;

static SortedList\_t\* list;

static SortedListElement\_t\*\* elements\_list;

/\* handling segmentation fault \*/

void signal\_handler() {

fprintf(stderr, "Caught segmentation fault\n");

exit(2);

}

/\* thread routine helper \*/

/\* inserts them all into a (single shared-by-all-threads) list \*/

void thread\_list\_insert(int t\_ID) {

fprintf(stderr, "%d\n", t\_ID);

for (int i = t\_ID; i < niterations\*nthreads; i+=nthreads) {

SortedList\_insert(list, elements\_list[i]);

}

}

/\* gets the list length \*/

void thread\_list\_length() {

int l = SortedList\_length(list);

if (l < 0) {

fprintf(stderr, "Corrupted list with length: %d\n", l);

exit(2);

}

}

/\* looks up and deletes each of the keys it had previously inserted \*/

void thread\_list\_del(int t\_ID) {

for (int i = t\_ID; i < niterations\*nthreads; i+=nthreads) {

SortedListElement\_t\* del = SortedList\_lookup(list, elements\_list[i]->key);

if (!del) {

fprintf(stderr, "Lookup failed\n");

exit(2);

}

if (SortedList\_delete(del) == 1) {

fprintf(stderr, "Deletion failed\n");

exit(2);

}

}

}

/\* thread routine \*/

void\* thread\_list(int\* t\_ID) {

int t = \*t\_ID;

switch(sync\_opt) {

case 0: {

thread\_list\_insert(t);

thread\_list\_length();

thread\_list\_del(t);

break;

}

//mutex

case 'm': {

if (pthread\_mutex\_lock(&lock) != 0) {

fprintf(stderr, "pthread\_mutex\_lock(): %s\n", strerror(errno));

exit(1);

}

thread\_list\_insert(t);

thread\_list\_length();

thread\_list\_del(t);

if (pthread\_mutex\_unlock(&lock) != 0) {

fprintf(stderr, "pthread\_mutex\_unlock(): %s\n", strerror(errno));

exit(1);

}

break;

}

//spin lock

case 's': {

while (\_\_sync\_lock\_test\_and\_set(&sl, 1));

thread\_list\_insert(t);

thread\_list\_length();

thread\_list\_del(t);

\_\_sync\_lock\_release(&sl);

break;

}

default: {

fprintf(stderr, "Wrong option of sync: %c\n", sync\_opt);

break;

}

}

pthread\_exit(NULL);

}

/\* helper functions \*/

int strtoint(char\* optarg) {

char\* endptr = NULL;

int ret = (int) strtol(optarg, &endptr, 10);

if (endptr == optarg || ret < 0)

ret = 1;

return ret;

}

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

/\* handle segmentation fault \*/

signal(SIGSEGV, signal\_handler);

/\* declare options \*/

static struct option long\_options[] = {

{ "iterations", optional\_argument, NULL, 'i' },

{ "threads", optional\_argument, NULL, 't' },

{ "sync", required\_argument, NULL, 's' },

{ "yield", required\_argument, NULL, 'y' },

{ 0,0,0,0 }

};

/\* declare variables\*/

int opt = 0;

//int sync\_opt = 0;

char test\_name[20];

strcpy(test\_name, "list");

long noperations = 0;

struct timespec tp\_start\_main;

struct timespec tp\_end\_main;

long long rt\_ntime;

long long avg\_ntime;

int yflag = 0;

char\* yield\_str = malloc(6);

char\* sync\_str;

while(1) {

opt = getopt\_long(argc, argv, "", long\_options, NULL);

if (opt == -1)

break;

switch(opt) {

/\* --iterations=# \*/

case 'i': {

if (!optarg)

nthreads = 1;

niterations = strtoint(optarg);

break;

}

/\* --threads=# \*/

case 't': {

if (!optarg)

nthreads = 1;

nthreads = strtoint(optarg);

break;

}

/\* --sync=# \*/

case 's': {

if (!optarg)

sync\_opt = 0;

else

sync\_opt = optarg[0];

switch(sync\_opt) {

case 'm': {

//mutexflag = 1;

sync\_str = "-m";

break;

}

case 's':

//spinflag = 1;

sync\_str = "-s";

break;

case 'c':

//cflag = 1;

sync\_str = "-c";

break;

default:

fprintf(stderr, "Unrecognized sync option %c\n", sync\_opt);

exit(1);

}

break;

}

/\* --yield \*/

case 'y': {

yflag = 1;

if (!optarg)

yield\_str = strcpy(yield\_str, "-none");

else if (strlen(optarg) > 4) {

fprintf(stderr, "Wrong option of yield: %s\n", optarg);

exit(1);

}

else {

yield\_str[0] = '-';

int index = 1;

for (int i = 0; i < 3; i++) {

if (optarg[i] == 'i') {

opt\_yield |= INSERT\_YIELD;

yield\_str[index++] = 'i';

}

else if (optarg[i] == 'd') {

opt\_yield |= DELETE\_YIELD;

yield\_str[index++] = 'd';

}

else if (optarg[i] == 'l') {

opt\_yield |= LOOKUP\_YIELD;

yield\_str[index++] = 'l';

}

}

yield\_str[index] = '\0';

}

break;

}

default: {

fprintf(stderr, "Unrecognized argument: %s\n", argv[optind-1]);

exit(1);

}

}

}

/\* test names \*/

if (yflag == 0)

yield\_str = strcpy(yield\_str, "-none");

if (sync\_opt == 0)

sync\_str = "-none";

strcat(test\_name, yield\_str);

strcat(test\_name, sync\_str);

/\* initialize an empty list \*/

list = malloc(sizeof(SortedList\_t));

if (list == NULL) {

fprintf(stderr, "Can't initialize a list\n");

exit(1);

}

list->key = NULL;

list->next = list;

list->prev = list;

/\* creates and initialize with random keys the required number of list elements \*/

char\* charset = { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" };

elements\_list = malloc(nthreads\*niterations\*sizeof(SortedListElement\_t\*));

for (int i = 0; i < nthreads\*niterations; i++) {

elements\_list[i] = malloc(sizeof(SortedListElement\_t));

char\* str = malloc(sizeof(char)\*2);

int r = rand() % 51;

str[0] = charset[r];

str[1] = '\0';

elements\_list[i]->key = str;

elements\_list[i]->prev = NULL;

elements\_list[i]->next = NULL;

}

/\* initialize thread arguments \*/

pthread\_t threads[nthreads];

noperations = 3\*nthreads\*niterations;

int thread\_IDs[nthreads];

/\* get starting time \*/

if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_start\_main) < 0) {

fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

exit(1);

}

/\* create threads \*/

for (int i = 0; i < nthreads; i++) {

thread\_IDs[i] = i;

if (pthread\_create(&threads[i], NULL, (void\*)&thread\_list, &thread\_IDs[i]) != 0){

fprintf(stderr, "pthread\_create() error: %s\n", strerror(errno));

exit(1);

}

}

/\* join threads \*/

for (int i = 0; i < nthreads; i++) {

void\* retVal;

if (pthread\_join(threads[i], &retVal) != 0){

fprintf(stderr, "pthread\_join() error: %s\n", strerror(errno));

exit(1);

}

}

/\* get ending time \*/

if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_end\_main) < 0) {

fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

exit(1);

}

rt\_ntime = (tp\_end\_main.tv\_sec - tp\_start\_main.tv\_sec) \* 1000000000;

rt\_ntime += tp\_end\_main.tv\_nsec;

rt\_ntime -= tp\_start\_main.tv\_nsec;

if (noperations != 0)

avg\_ntime = rt\_ntime / noperations;

else

avg\_ntime = rt\_ntime;

/\* Check final length \*/

opt\_yield = 0;

int length = SortedList\_length(list);

if (length != 0) {

fprintf(stderr, "Corrupted list with length: %d\n", length);

fprintf(stderr, "\t caused by %s, %d threads and %d iterations\n", test\_name, nthreads, niterations);

/\* free memory \*/

for (int i = 0; i < nthreads\*noperations; i++) {

free(elements\_list[i]);

}

free(elements\_list);

exit(2);

}

/\* print results to csv \*/

fprintf(stdout, "%s,%d,%d,%d,%ld,%lld,%lld\n",

test\_name, nthreads, niterations, 1, noperations, rt\_ntime, avg\_ntime);

/\* free memory \*/

for (int i = 0; i < nthreads\*niterations; i++) {

if (elements\_list) {

//free((void\* )elements\_list[i]->key);

free(elements\_list[i]);

}

}

free(elements\_list);

free(list);

free(yield\_str);

}

**lab2-list-multi**

/\* global variables \*/

int opt\_yield = 0;

int nlists = 1;

// pthread\_mutex\_t m\_lock = PTHREAD\_MUTEX\_INITIALIZER;

// static int sl = 0;

pthread\_mutex\_t\* mutex;

int\* spin;

static int niterations = 1;

static int nthreads = 1;

static int sync\_opt = 0;

//static SortedList\_t\* list; // head of sublists

static SortedList\_t\*\* sublists; // insert to each sublist

static SortedListElement\_t\*\* elements\_list;

struct timespec tp\_start\_lock;

struct timespec tp\_end\_lock;

long long wait\_time = 0;

//unsigned long long\* wait\_time;

/\* hash function \*/

unsigned int hash(const char\* key) {

unsigned int hash\_n = 5381;

int c;

while ((c = \*key++))

hash\_n = ((hash\_n << 5) + hash\_n) + c; /\* hash \* 33 + c \*/

return hash\_n;

}

/\* handling segmentation fault \*/

void signal\_handler() {

fprintf(stderr, "Caught segmentation fault\n");

exit(2);

}

/\* locking routine \*/

void lock(int l\_ID) {

switch(sync\_opt) {

case 'm': {

if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_start\_lock) < 0) {

fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

exit(1);

}

if (pthread\_mutex\_lock(&mutex[l\_ID]) != 0) {

//if (pthread\_mutex\_lock(&m\_lock) != 0) {

fprintf(stderr, "pthread\_mutex\_lock(): %s\n", strerror(errno));

exit(1);

}

/\* get ending time \*/

if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_end\_lock) < 0) {

fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

exit(1);

}

wait\_time += (tp\_end\_lock.tv\_sec - tp\_start\_lock.tv\_sec) \* 1000000000;

wait\_time += tp\_end\_lock.tv\_nsec;

wait\_time -= tp\_start\_lock.tv\_nsec;

break;

}

case 's': {

while (\_\_sync\_lock\_test\_and\_set(&spin[l\_ID], 1));

//while (\_\_sync\_lock\_test\_and\_set(&sl, 1));

break;

}

}

}

void unlock(int l\_ID) {

switch(sync\_opt) {

case 'm': {

if (pthread\_mutex\_unlock(&mutex[l\_ID]) != 0) {

//if (pthread\_mutex\_unlock(&m\_lock) != 0) {

fprintf(stderr, "pthread\_mutex\_unlock(): %s\n", strerror(errno));

exit(1);

}

break;

}

case 's': {

\_\_sync\_lock\_release(&spin[l\_ID]);

//\_\_sync\_lock\_release(&sl);

break;

}

}

}

/\* thread routine \*/

void list\_insert(int t) {

//lock(0,t);

for (int i = t; i < niterations\*nthreads; i+=nthreads) {

int list\_ind = hash(elements\_list[i]->key) % nlists;

lock(list\_ind);

SortedList\_insert(sublists[list\_ind], elements\_list[i]);

unlock(list\_ind);

}

//unlock(0);

}

void list\_length() {

//lock(0,t);

int l = 0;

for (int i = 0; i < nlists; i++) {

lock(i);

l = SortedList\_length(sublists[i]);

if (l < 0) {

fprintf(stderr, "Corrupted list with length: %d\n", l);

exit(2);

}

unlock(i);

}

//unlock(0);

}

void list\_delete(int t) {

//lock(0,t);

for (int i = t; i < niterations\*nthreads; i+=nthreads) {

int list\_ind = hash(elements\_list[i]->key) % nlists;

lock(list\_ind);

SortedListElement\_t\* del = SortedList\_lookup(sublists[list\_ind], elements\_list[i]->key);

if (!del) {

fprintf(stderr, "Lookup failed\n");

exit(2);

}

if (SortedList\_delete(del) == 1) {

fprintf(stderr, "Deletion failed\n");

exit(2);

}

unlock(list\_ind);

}

//unlock(0);

}

void thread\_list\_helper(int t) {

// switch(sync\_opt) {

// case 'm': {

// if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_start\_lock) < 0) {

// fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

// exit(1);

// }

// //if (pthread\_mutex\_lock(&mutex[l\_ID]) != 0) {

// if (pthread\_mutex\_lock(&m\_lock) != 0) {

// fprintf(stderr, "pthread\_mutex\_lock(): %s\n", strerror(errno));

// exit(1);

// }

// /\* get ending time \*/

// if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_end\_lock) < 0) {

// fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

// exit(1);

// }

// wait\_time += (tp\_end\_lock.tv\_sec - tp\_start\_lock.tv\_sec) \* 1000000000;

// wait\_time += tp\_end\_lock.tv\_nsec;

// wait\_time -= tp\_start\_lock.tv\_nsec;

// break;

// }

// case 's': {

// while (\_\_sync\_lock\_test\_and\_set(&sl, 1));

// break;

// }

// }

list\_insert(t);

list\_length();

list\_delete(t);

// switch(sync\_opt) {

// case 'm': {

// //if (pthread\_mutex\_unlock(&mutex[l\_ID]) != 0) {

// if (pthread\_mutex\_unlock(&m\_lock) != 0) {

// fprintf(stderr, "pthread\_mutex\_unlock(): %s\n", strerror(errno));

// exit(1);

// }

// break;

// }

// case 's': {

// \_\_sync\_lock\_release(&sl);

// break;

// }

// }

}

void\* thread\_list(int\* t\_ID) {

int t = \*t\_ID;

thread\_list\_helper(t);

pthread\_exit(NULL);

}

/\* helper functions \*/

int strtoint(char\* optarg) {

char\* endptr = NULL;

int ret = (int) strtol(optarg, &endptr, 10);

if (endptr == optarg || ret < 0)

ret = 1;

return ret;

}

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

/\* handle segmentation fault \*/

signal(SIGSEGV, signal\_handler);

/\* declare options \*/

static struct option long\_options[] = {

{ "iterations", optional\_argument, NULL, 'i' },

{ "threads", optional\_argument, NULL, 't' },

{ "sync", required\_argument, NULL, 's' },

{ "yield", required\_argument, NULL, 'y' },

{ "lists", required\_argument, NULL, 'l' },

{ 0,0,0,0 }

};

/\* declare variables\*/

int opt = 0;

//int sync\_opt = 0;

char test\_name[20];

strcpy(test\_name, "list");

long noperations = 0;

struct timespec tp\_start\_main;

struct timespec tp\_end\_main;

long long rt\_ntime;

long long avg\_ntime;

long long avg\_wtime = 0;

int yflag = 0;

char\* yield\_str = malloc(6);

char\* sync\_str;

while(1) {

opt = getopt\_long(argc, argv, "", long\_options, NULL);

if (opt == -1)

break;

switch(opt) {

/\* --iterations=# \*/

case 'i': {

if (!optarg)

nthreads = 1;

niterations = strtoint(optarg);

break;

}

/\* --threads=# \*/

case 't': {

if (!optarg)

nthreads = 1;

nthreads = strtoint(optarg);

break;

}

/\* --sync=# \*/

case 's': {

if (!optarg)

sync\_opt = 0;

else

sync\_opt = optarg[0];

switch(sync\_opt) {

case 'm': {

//mutexflag = 1;

sync\_str = "-m";

break;

}

case 's':

//spinflag = 1;

sync\_str = "-s";

break;

case 'c':

//cflag = 1;

sync\_str = "-c";

break;

default:

fprintf(stderr, "Unrecognized sync option %c\n", sync\_opt);

exit(1);

}

break;

}

/\* --yield \*/

case 'y': {

yflag = 1;

if (!optarg)

yield\_str = strcpy(yield\_str, "-none");

else if (strlen(optarg) > 4) {

fprintf(stderr, "Wrong option of yield: %s\n", optarg);

exit(1);

}

else {

yield\_str[0] = '-';

int index = 1;

for (int i = 0; i < 3; i++) {

if (optarg[i] == 'i') {

opt\_yield |= INSERT\_YIELD;

yield\_str[index++] = 'i';

}

else if (optarg[i] == 'd') {

opt\_yield |= DELETE\_YIELD;

yield\_str[index++] = 'd';

}

else if (optarg[i] == 'l') {

opt\_yield |= LOOKUP\_YIELD;

yield\_str[index++] = 'l';

}

}

yield\_str[index] = '\0';

}

break;

}

/\* --lists=# \*/

case 'l': {

if (!optarg) {

fprintf(stderr, "Require argument for --lists\n");

exit(1);

}

nlists = strtoint(optarg);

break;

}

default: {

fprintf(stderr, "Unrecognized argument: %s\n", argv[optind-1]);

exit(1);

}

}

}

/\* test names \*/

if (yflag == 0)

yield\_str = strcpy(yield\_str, "-none");

if (sync\_opt == 0)

sync\_str = "-none";

strcat(test\_name, yield\_str);

strcat(test\_name, sync\_str);

// /\* initialize an empty list \*/

// list = malloc(sizeof(SortedList\_t));

// if (list == NULL) {

// fprintf(stderr, "Can't initialize a list\n");

// exit(1);

// }

// list->key = NULL;

// list->next = list;

// list->prev = list;

/\* initialize empty sublists \*/

sublists = malloc(nlists\*sizeof(SortedList\_t\*));

mutex = malloc(nlists\*sizeof(pthread\_mutex\_t));

spin = malloc(nlists\*sizeof(int));

for (int i = 0; i < nlists; i++) {

sublists[i] = malloc(sizeof(SortedList\_t));

if (!sublists[i]) {

fprintf(stderr, "Can't initialize sublists\n");

exit(1);

}

sublists[i]->key = NULL;

sublists[i]->next = sublists[i];

sublists[i]->prev = sublists[i];

mutex[i] = (pthread\_mutex\_t)PTHREAD\_MUTEX\_INITIALIZER;

spin[i] = 0;

}

/\* creates and initialize with random keys the required number of list elements \*/

char\* charset = { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" };

elements\_list = malloc(nthreads\*niterations\*sizeof(SortedListElement\_t\*));

for (int i = 0; i < nthreads\*niterations; i++) {

elements\_list[i] = malloc(sizeof(SortedListElement\_t));

if (!elements\_list[i]) {

fprintf(stderr, "Can't initialize elements\n");

exit(1);

}

char\* str = malloc(sizeof(char)\*2);

int r = rand() % 51;

str[0] = charset[r];

str[1] = '\0';

elements\_list[i]->key = str;

elements\_list[i]->prev = NULL;

elements\_list[i]->next = NULL;

}

/\* initialize thread arguments \*/

pthread\_t threads[nthreads];

noperations = 3\*nthreads\*niterations;

int thread\_IDs[nthreads];

//wait\_time = malloc(nthreads\*sizeof(unsigned long long));

// for (int i = 0; i < nthreads; i++) {

// wait\_time[i] = 0;

// }

/\* get starting time \*/

if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_start\_main) < 0) {

fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

exit(1);

}

/\* create threads \*/

for (int i = 0; i < nthreads; i++) {

thread\_IDs[i] = i;

if (pthread\_create(&threads[i], NULL, (void\*)&thread\_list, &thread\_IDs[i]) != 0){

fprintf(stderr, "pthread\_create() error: %s\n", strerror(errno));

exit(1);

}

}

/\* join threads \*/

for (int i = 0; i < nthreads; i++) {

void\* retVal;

if (pthread\_join(threads[i], &retVal) != 0){

fprintf(stderr, "pthread\_join() error: %s\n", strerror(errno));

exit(1);

}

}

/\* get ending time \*/

if (clock\_gettime(CLOCK\_MONOTONIC, &tp\_end\_main) < 0) {

fprintf(stderr, "clock\_gettime() error: %s\n", strerror(errno));

exit(1);

}

rt\_ntime = (tp\_end\_main.tv\_sec - tp\_start\_main.tv\_sec) \* 1000000000;

rt\_ntime += tp\_end\_main.tv\_nsec;

rt\_ntime -= tp\_start\_main.tv\_nsec;

/\* Check final length \*/

opt\_yield = 0;

for (int i = 0; i < nlists; i++) {

if (SortedList\_length(sublists[i]) != 0) {

fprintf(stderr, "Corrupted list\n");

fprintf(stderr, "\t caused by %s, %d threads and %d iterations\n", test\_name, nthreads, niterations);

/\* free memory \*/

for (int i = 0; i < nthreads\*noperations; i++) {

free(elements\_list[i]);

}

free(elements\_list);

exit(2);

}

}

/\* computer average wait time \*/

// unsigned long long wait\_total = 0;

// for (int i = 0; i < nthreads; i++) {

// wait\_total += wait\_time[i];

// }

// avg\_wtime = wait\_total / noperations;

avg\_wtime = wait\_time / noperations;

avg\_ntime = rt\_ntime / noperations;

/\* print results to csv \*/

fprintf(stdout, "%s,%d,%d,%d,%ld,%lld,%lld,%lld\n",

test\_name, nthreads, niterations, nlists, noperations, rt\_ntime, avg\_ntime, avg\_wtime);

/\* free memory \*/

for (int i = 0; i < nthreads\*niterations; i++) {

if (elements\_list) {

//free((void\* )elements\_list[i]->key);

free(elements\_list[i]);

}

}

free(elements\_list);

//free(list);

free(yield\_str);

}