**Lab 1 SIMPSH**

#include <stdio.h>

#include <unistd.h>

#include <getopt.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <signal.h>

#include <fcntl.h>

#include <time.h>

#include <sys/resource.h>

#include <sys/wait.h>

#define APPEND O\_APPEND

#define CLOEXEC O\_CLOEXEC

#define CREAT O\_CREAT

#define DIRECTORY O\_DIRECTORY

#define DSYNC O\_DSYNC

#define EXCL O\_EXCL

#define NOFOLLOW O\_NOFOLLOW

#define NONBLOCK O\_NONBLOCK

#define RSYNC O\_RSYNC

#define SYNC O\_SYNC

#define TRUNC O\_TRUNC

void sighandler(int signum)

{

fprintf(stderr, "Sighandler: Signal %d caught.\n", signum);

exit(signum);

}

int max(int num1, int num2)

{

return (num1 > num2 ? num1 : num2);

}

void print\_message(char c, char \*\* argv, int optind)

{

if(c == 'f')

printf("%s\n", argv[optind-1]);

if(c == 'o')

printf("%s %s\n", argv[optind - 2], optarg);

fflush(stdout);

}

struct commandargs

{

pid\_t pid;

char \* argstring;

};

struct time

{

struct timeval utime;

struct timeval stime;

};

/\* Subtract the ‘struct timeval’ values X and Y,

storing the result in RESULT.

Return 1 if the difference is negative, otherwise 0. \*/

int timeval\_subtract (struct timeval \*result, struct timeval \*x, struct timeval \*y)

{

/\* Perform the carry for the later subtraction by updating y. \*/

if (x->tv\_usec < y->tv\_usec) {

int nsec = (y->tv\_usec - x->tv\_usec) / 1000000 + 1;

y->tv\_usec -= 1000000 \* nsec;

y->tv\_sec += nsec;

}

if (x->tv\_usec - y->tv\_usec > 1000000) {

int nsec = (x->tv\_usec - y->tv\_usec) / 1000000;

y->tv\_usec += 1000000 \* nsec;

y->tv\_sec -= nsec;

}

/\* Compute the time remaining to wait.

tv\_usec is certainly positive. \*/

result->tv\_sec = x->tv\_sec - y->tv\_sec;

result->tv\_usec = x->tv\_usec - y->tv\_usec;

/\* Return 1 if result is negative. \*/

return x->tv\_sec < y->tv\_sec;

}

void start\_timer(int kind, struct time \* startTime)

{

struct rusage usage;

int time = getrusage(kind, &usage);

startTime->utime = usage.ru\_utime;

startTime->stime = usage.ru\_stime;

}

void end\_timer(int kind, struct time \* startTime, char \* str)

{

struct rusage usage;

int num = getrusage(kind, &usage);

struct time endTime;

struct time totalTime;

endTime.utime = usage.ru\_utime;

endTime.stime = usage.ru\_stime;

timeval\_subtract(&totalTime.utime, &endTime.utime, &startTime->utime);

timeval\_subtract(&totalTime.stime, &endTime.stime, &startTime->stime);

printf("%s: User time: %ld.%ds | System time: %ld.%lds \n",

str,

totalTime.utime.tv\_sec, totalTime.utime.tv\_usec,

totalTime.stime.tv\_sec, totalTime.stime.tv\_usec);

fflush(stdout);

}

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

{

//declare file descriptor array and possible flags

int filearr\_size = 10;

int \* filelist = (int\*) malloc(10 \* sizeof(int));

int filepos = 0, verbose = 0, exitnum = 0;

//intialize process array for when running commands

int pro\_size = 10, pro\_pos = 0;

pid\_t \* pro\_arr = (pid\_t \*) malloc(10 \* sizeof(pid\_t));

int c = 0, option\_index = 0, fileflags = 0;

int pipefile[2];

char \* str = NULL;

int cmdarray\_size = 10;

int terminated = 0;

int signum = 0;

int commandargs\_size = 0;

struct commandargs \* commandstruct = malloc(100\*sizeof(struct commandargs));

int profile = 0;

struct time \* childStartTime = malloc(sizeof(struct time));

struct time \* startTime = malloc(sizeof(struct time));

struct time \* shelltime = malloc(sizeof(struct time));

struct time \* childshelltime = malloc(sizeof(struct time));

//long\_options class to read through different options

static struct option long\_options[] = {

//File flags

{"append", no\_argument, 0, APPEND},

{"cloexec", no\_argument, 0, CLOEXEC},

{"creat", no\_argument, 0, CREAT},

{"directory", no\_argument, 0, DIRECTORY},

{"dsync", no\_argument, 0, DSYNC},

{"excl", no\_argument, 0, EXCL},

{"nofollow", no\_argument, 0, NOFOLLOW},

{"nonblock", no\_argument, 0, NONBLOCK},

{"rync", no\_argument, 0, RSYNC},

{"sync", no\_argument, 0, SYNC},

{"trunc", no\_argument, 0, TRUNC},

//Open Files

{"rdonly", required\_argument, 0, 'r'},

{"rdwr", required\_argument, 0, 'R'},

{"wronly", required\_argument, 0, 'w'},

{"pipe", no\_argument, 0, 'p'},

//Subcommand options

{"command", required\_argument, 0, 'c'},

{"wait", no\_argument, 0, 'W'},

//Misc options

{"close", required\_argument, 0, 'C'},

{"verbose", no\_argument, 0, 'v'},

{"profile", no\_argument, 0, 'P'},

{"abort", no\_argument, 0, 'a'},

{"catch", required\_argument, 0, 'b'},

{"ignore", required\_argument, 0, 'i'},

{"default", required\_argument, 0, 'd'},

{"pause", no\_argument, 0, 'e'},

{0,0,0,0}

};

//check to make sure both process array and file descriptor array are malloc'ed correctly

if(!pro\_arr || !filelist)

{

fprintf(stderr, "Error: Could not allocate dynamic memory. \n");

exit(1);

}

//main while loop to parse through each option

while((c = getopt\_long(argc, argv, "", long\_options, &option\_index)) != -1)

{

switch (c)

{

case APPEND: case CLOEXEC: case CREAT: case DIRECTORY:

case DSYNC: case EXCL: case NOFOLLOW: case NONBLOCK:

case SYNC: case TRUNC:

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

if(verbose)

print\_message('f', argv, optind);

fileflags |= c;

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-1]);

break;

case 'r': //rdonly

{

if(verbose)

print\_message('o', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

//open file in file description array using O\_RDONLY flag

filelist[filepos] = open(optarg, fileflags | O\_RDONLY, S\_IRUSR | S\_IWUSR | S\_IROTH);

if(filelist[filepos] == -1) //check for read error, and set flags

{

fprintf(stderr, "RDONLY Error: %s, Unable to open file %s. \n", strerror(errno), optarg);

exitnum = 1;

}

filepos++;

fileflags = 0;

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-2]);

break;

}

case 'R': //RDWR

{

if(verbose)

print\_message('o', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

//open file in file description array using O\_RDWR flag

filelist[filepos] = open(optarg, fileflags | O\_RDWR, S\_IRUSR | S\_IWUSR | S\_IROTH);

if(filelist[filepos] == -1) //check for read error, and set flags

{

fprintf(stderr, "RDWR Error: %s, Unable to open file %s. \n", strerror(errno), optarg);

exitnum = 1;

}

filepos++;

fileflags = 0;

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-2]);

break;

}

case 'w': //wronly, same process as above

{

if(verbose)

print\_message('o', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

filelist[filepos] = open(optarg, fileflags | O\_WRONLY, S\_IRUSR | S\_IWUSR | S\_IROTH);

if(filelist[filepos] == -1)

{

fprintf(stderr, "WRONLY Error: %s Unable to open file %s. \n", strerror(errno), optarg);

exitnum = 1;

}

filepos++;

fileflags = 0;

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-2]);

break;

}

case 'p':

{

if(verbose)

print\_message('f', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

if(pipe(pipefile) == 0)

{

filelist[filepos] = pipefile[0]; //read

filepos++;

filelist[filepos] = pipefile[1]; //write

filepos ++;

}

else

{

fprintf(stderr, "Pipe Error: %s Unable to open file %s. \n", strerror(errno), optarg);

exitnum = 1;

}

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-1]);

break;

}

case 'c': //command

{

int start = optind;

int cur = start;

int end;

int input = atoi(argv[start-1]); //set input fd to one before optind

int output = atoi(argv[start]); //set output fd to where optind is located

int error = atoi(argv[start+1]); //set error fd to one after optind

char \* cmd = argv[start+2]; //starting command after fd

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

//check for any file descriptor errors

if(input < 0 || output < 0 || error < 0 || input >= filepos || output >= filepos || error >= filepos)

{

fprintf(stderr, "Error: Invalid I/O or Error file descriptor \n");

exit(1);

}

while(cur < argc)

{

//parse through option until next "--" option

if(argv[cur][0] == '-' && argv[cur][1] == '-')

{

end = cur - 1;

break;

}

cur++;

}

//if current index is at end, set end to one before.

if(cur == argc)

end = argc - 1;

int optlen = end - start - 2; //create local variable for length of option

char \*\* optarray = malloc((optlen+1) \* sizeof(char\*));

if(optarray) //check to see if array is malloc'ed correctly

{

//set first part of array to the function

optarray[0] = cmd;

int i = 1;

for(; i <= optlen; i++) //insert options into optarray

{

optarray[i] = argv[start + 2 + i];

}

optarray[i] = NULL; //set end of array to NULL

if(verbose)

{

printf("%s %d %d %d", argv[optind-2], input, output, error);

int inc = 0;

while(inc <= optlen)

{

printf(" %s", optarray[inc]);

fflush(stdout);

inc++;

}

printf("\n");

fflush(stdout);

}

int inc = 1;

int index = 0;

int cmdstring\_size = 40;

char\* cmdstring = malloc(cmdstring\_size\*sizeof(char));

while(index < strlen(cmd))

{

cmdstring[index] = cmd[index];

index++;

}

cmdstring[index] = ' ';

index ++;

while(inc <= optlen)

{

for(int j = 0; j < strlen(argv[start + 2 + inc]); j ++)

{

if(cmdstring\_size - index < 15)

{

cmdarray\_size += 40;

cmdstring = realloc(cmdstring, cmdstring\_size\*sizeof(char));

}

cmdstring[index] = argv[start + 2 + inc][j];

index++;

}

cmdstring[index] = ' ';

index++;

inc ++;

}

//check for closed files

if(filelist[input] == -1 || filelist[output] == -1 || filelist[error] == -1)

{

fprintf(stderr, "Error: File is unable to be opened.\n");

exit(1);

}

if(cmd[0] != '-' && cmd != NULL)

{

pro\_arr[pro\_pos] = fork(); //fork processes in the process array to run

if(pro\_arr[pro\_pos] >= 0) //check to make sure the fork was sucessful

{

if(pro\_arr[pro\_pos] == 0) // if the process is a child

{

//duplicate each file descriptor

dup2(filelist[input], 0);

dup2(filelist[output], 1);

dup2(filelist[error], 2);

int j = 0;

while(j < filepos)

{

close(filelist[j]); //close original file descriptors

j++;

}

//execvp runs the function, check to see ran correctly

if (execvp(optarray[0], optarray) == -1)

{

fprintf(stderr, "Error %s: Could not execute command. \n", strerror(errno));

exitnum = 1;

}

}

commandstruct[commandargs\_size].argstring = cmdstring;

commandstruct[commandargs\_size].pid = pro\_arr[pro\_pos];

commandargs\_size++;

}

else //else return an error

{

fprintf(stderr, "Error: Could not create child process. \n");

exitnum = 1;;

}

}

else //else return an error

{

fprintf(stderr, "Error: Invalid Command. \n");

exitnum = 1;;

}

}

else

{

fprintf(stderr, "Error: Could not allocate memory. \n");

exitnum = 1;

}

pro\_pos++; //increase process array index each time goes through

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-2]);

break;

}

case 'W': //wait

{

if(verbose)

print\_message('f', argv, optind);

if(profile)

{

start\_timer(RUSAGE\_SELF, startTime);

start\_timer(RUSAGE\_CHILDREN, startTime);

}

while(terminated < pro\_pos)

{

int wstatus;

int cstatus = 0;

pid\_t finprocess = waitpid(-1, &wstatus, 0);

if(WIFEXITED(wstatus))

{

cstatus = WEXITSTATUS(wstatus);

printf("exit %d ", cstatus);

}

if(WIFSIGNALED(wstatus))

{

cstatus = WTERMSIG(wstatus);

printf("signal %d ", cstatus);

signum = max(signum, cstatus);

}

exitnum = max(cstatus, exitnum);

int i = 0;

for(; i < commandargs\_size; i ++)

{

if(commandstruct[i].pid == finprocess)

{

printf("%s\n", commandstruct[i].argstring);

fflush(stdout);

break;

}

}

terminated ++;

}

// free(commandstruct);

commandargs\_size = 0;

fflush(stdout);

commandstruct = malloc(100\*sizeof(int));

if(profile)

{

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-1]);

end\_timer(RUSAGE\_CHILDREN, startTime, "Children Processes");

}

break;

}

case 'C': //close

{

if(verbose)

print\_message('f', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

int index = atoi(optarg);

if(close(filelist[index]) == -1)

{

fprintf(stderr, "Error: Could not close file correctly. \n");

exitnum = 1;

}

filelist[index] = -1;

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-2]);

break;

}

case 'v': //verbose

{

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

verbose = 1;

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-1]);

break;

}

case 'P': //profile

{

if(verbose)

print\_message('f', argv, optind);

profile = 1;

break;

}

case 'a': //abort

{

if(verbose)

print\_message('f', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

\*str = 'l';

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-1]);

break;

}

case 'b': //catch

{

if(verbose)

print\_message('o', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

signal(atoi(optarg), sighandler);

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-2]);

break;

}

case 'i': //ignore

{

if(verbose)

print\_message('o', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

signal(atoi(optarg), SIG\_IGN);

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-2]);

break;

}

case 'd': //default

{

if(verbose)

print\_message('o', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

signal(atoi(optarg), SIG\_DFL);

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-2]);

break;

}

case 'e': //pause

{

if(verbose)

print\_message('f', argv, optind);

if(profile)

start\_timer(RUSAGE\_SELF, startTime);

pause();

if(profile)

end\_timer(RUSAGE\_SELF, startTime, (char\*) argv[optind-1]);

break;

}

default:

break;

}

if(filearr\_size == filepos) //check for max size of file descriptor array

{

filearr\_size += 10;

filelist = realloc(filelist, filearr\_size\*sizeof(int));

}

if(pro\_size == pro\_pos) //check for max size of process array

{

pro\_size += 10;

pro\_arr = realloc(pro\_arr, pro\_size\*sizeof(int));

}

for(int l = optind; l < argc; l ++) //increment index of optind to next '--'

{

if(argv[optind][0] == '-' && argv[optind][1] == '-')

break;

optind++;

}

}

if(signum != 0)

{

signal(signum, SIG\_DFL);

raise(signum);

}

exit(exitnum);

}

**LAB2\_ADD.C**

/\*

NAME: Luke Jung

EMAIL: lukejung@ucla.edu

UID: 904982644

\*/

#include <getopt.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#include <pthread.h>

#include <sched.h>

#define THREADS 't'

#define ITERATIONS 'i'

#define SYNC 'x'

#define YIELD 'y'

#define NONE 'n'

#define MUTEX 'm'

#define SPINLOCK 's'

#define CAS 'c'

static int yield = 0;

static long long counter = 0;

static int spinlock = 0;

static int iterations = 1;

static pthread\_mutex\_t lock;

static char options = NONE;

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

{

long long sum = \*pointer + value;

if(yield)

sched\_yield();

\*pointer = sum;

}

void add\_controller(int num)

{

switch(options)

{

case NONE:

{

for(int i = 0; i < iterations; i ++)

{

add(&counter, num);

}

break;

}

case MUTEX:

{

for(int i = 0; i < iterations; i ++)

{

pthread\_mutex\_lock(&lock);

add(&counter, num);

pthread\_mutex\_unlock(&lock);

}

break;

}

case SPINLOCK:

{

for(int i = 0; i < iterations; i ++)

{

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

continue;

add(&counter, num);

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

}

break;

}

case CAS:

{

for(int i = 0; i < iterations; i ++)

{

int temp1, temp2;

do

{

temp1 = counter;

if(yield)

sched\_yield();

temp2 = temp1 + num;

}

while(\_\_sync\_val\_compare\_and\_swap(&counter, temp1, temp2) != temp1);

}

break;

}

default:

break;

}

}

void \*run\_threads()

{

add\_controller(1);

add\_controller(-1);

return NULL;

}

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

{

struct option long\_options[] =

{

{"threads", required\_argument, 0, THREADS},

{"iterations", required\_argument, 0, ITERATIONS},

{"sync", required\_argument, 0, SYNC},

{"yield", no\_argument, 0, YIELD},

{0, 0, 0, 0}

};

int c = 0;

char \* optionstr = malloc(sizeof(char) \* 20);

sprintf(optionstr, "add");

int threads = 1;

while((c = getopt\_long(argc, argv, "", long\_options, 0)) != -1)

{

switch(c)

{

case THREADS:

{

threads = atoi(optarg);

break;

}

case ITERATIONS:

{

iterations = atoi(optarg);

break;

}

case YIELD:

{

yield = 1;

strcat(optionstr, "-yield");

break;

}

case SYNC:

{

options = optarg[0];

if(options != MUTEX && options != SPINLOCK && options != CAS)

{

fprintf(stderr, "Error: Invalid Sync Option. \n");

exit(1);

}

if(options == MUTEX)

{

if(pthread\_mutex\_init(&lock, NULL))

{

fprintf(stderr, "Error: Mutex Initialization failed. \n");

exit(1);

}

}

break;

}

default:

{

fprintf(stderr, "Error: Invalid option");

exit(1);

break;

}

}

}

if(options == NONE)

{

strcat(optionstr, "-none");

}

else

sprintf(optionstr + strlen(optionstr), "-%c", options);

pthread\_t \* id = (pthread\_t \*) malloc(sizeof(pthread\_t) \* threads);

struct timespec start;

clock\_gettime(CLOCK\_MONOTONIC, &start);

int i = 0;

for(; i < threads; i ++)

{

if(pthread\_create(&id[i], NULL, &run\_threads, NULL))

{

fprintf(stderr, "Error: Creation of Threads failed. \n");

free(id);

exit(1);

}

}

int j = 0;

for(; j < threads; j++)

{

if(pthread\_join(id[j], NULL))

{

fprintf(stderr, "Error: Joining Threads failed. \n");

exit(1);

}

}

struct timespec stop;

clock\_gettime(CLOCK\_MONOTONIC, &stop);

long long time = stop.tv\_nsec - start.tv\_nsec;

time += (stop.tv\_sec - start.tv\_sec) \* 1000000000;

long long ops = threads \* iterations \* 2;

printf("%s,%d,%d,%lld,%lld,%lld,%lld\n", optionstr, threads, iterations, ops,

time, time / ops, counter);

free(id);

free(optionstr);

if(options == MUTEX)

{

pthread\_mutex\_destroy(&lock);

}

if(counter != 0)

exit(1);

else

exit(0);

}

**LAB2\_LIST.C**

#include "SortedList.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

#include <getopt.h>

#include <signal.h>

#include <pthread.h>

#include <time.h>

#define THREADS 't'

#define ITERATIONS 'i'

#define SYNC 'x'

#define YIELD 'y'

#define NONE 'n'

#define MUTEX 'm'

#define SPINLOCK 's'

#define LISTS 'l'

#define ID\_YIELD 0x03

#define IL\_YIELD 0x05

#define DL\_YIELD 0x06

#define IDL\_YIELD 0x07

SortedList\_t \* lists;

SortedListElement\_t \* elements;

int iterations = 1;

pthread\_mutex\_t lock;

char options = NONE;

int threads;

int spinlock;

int numlists = 1;

long long \* threadTimes;

int opt\_yield = 0;

int element\_len = 0;

void seg\_fault()

{

fprintf(stderr, "Error: Seg fault. \n");

exit(2);

}

//Hash fucnction from http://www.partow.net/programming/hashfunctions/#top

unsigned long hash(const char \*str)

{

unsigned int b = 378551;

unsigned int a = 63689;

unsigned int hash = 0;

unsigned int i = 0;

for (i = 0; i < 10; ++str, ++i)

{

hash = hash \* a + (\*str);

a = a \* b;

}

return hash % numlists;

}

void \* run\_threads(void \* list)

{

struct timespec start;

struct timespec end;

int num = (\*(int\*) list);

for(int i = num; i < element\_len; i += threads)

{

int index = hash(elements[i].key);

switch(options)

{

case MUTEX:

{

clock\_gettime(CLOCK\_MONOTONIC, &start);

pthread\_mutex\_lock(&lock);

clock\_gettime(CLOCK\_MONOTONIC, &end);

long long time = end.tv\_nsec - start.tv\_nsec;

time += (end.tv\_sec - start.tv\_sec) \* 1000000000;

threadTimes[num] += time;

SortedList\_insert(&lists[index], &elements[i]);

pthread\_mutex\_unlock(&lock);

break;

}

case SPINLOCK:

{

clock\_gettime(CLOCK\_MONOTONIC, &start);

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

clock\_gettime(CLOCK\_MONOTONIC, &end);

long long time = end.tv\_nsec - start.tv\_nsec;

time += (end.tv\_sec - start.tv\_sec) \* 1000000000;

threadTimes[num] += time;

SortedList\_insert(&lists[index], &elements[i]);

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

break;

}

case NONE:

{

SortedList\_insert(&lists[index], &elements[i]);

break;

}

}

}

SortedListElement\_t \* el;

for(int i = num; i < element\_len; i += threads)

{

int index = hash(elements[i].key);

switch(options)

{

case MUTEX:

{

clock\_gettime(CLOCK\_MONOTONIC, &start);

pthread\_mutex\_lock(&lock);

clock\_gettime(CLOCK\_MONOTONIC, &end);

long long time = end.tv\_nsec - start.tv\_nsec;

time += (end.tv\_sec - start.tv\_sec) \* 1000000000;

threadTimes[num] += time;

el = SortedList\_lookup(&lists[index], elements[i].key);

if(el)

{

SortedList\_delete(el);

}

pthread\_mutex\_unlock(&lock);

break;

}

case SPINLOCK:

{

clock\_gettime(CLOCK\_MONOTONIC, &start);

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

clock\_gettime(CLOCK\_MONOTONIC, &end);

long long time = end.tv\_nsec - start.tv\_nsec;

time += (end.tv\_sec - start.tv\_sec) \* 1000000000;

threadTimes[num] += time;

el = SortedList\_lookup(&lists[index], elements[i].key);

if(el)

{

SortedList\_delete(el);

}

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

break;

}

case NONE:

{

el = SortedList\_lookup(&lists[index], elements[i].key);

if(el)

{

SortedList\_delete(el);

}

break;

}

}

}

return NULL;

}

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

{

struct option long\_options[] =

{

{"threads", required\_argument, 0, THREADS},

{"iterations", required\_argument, 0, ITERATIONS},

{"sync", required\_argument, 0, SYNC},

{"yield", required\_argument, 0, YIELD},

{"lists", required\_argument, 0, LISTS},

{0, 0, 0, 0}

};

int c = 0;

opt\_yield = 0;

threads = 1;

signal(SIGSEGV, seg\_fault);

while((c = getopt\_long(argc, argv, "", long\_options, 0)) != -1)

{

switch(c)

{

case THREADS:

{

threads = atoi(optarg);

break;

}

case ITERATIONS:

{

iterations = atoi(optarg);

break;

}

case YIELD:

{

int len = strlen(optarg);

for(int i = 0; i < len; i ++)

{

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

opt\_yield |= DELETE\_YIELD;

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

opt\_yield |= INSERT\_YIELD;

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

opt\_yield |= LOOKUP\_YIELD;

}

break;

}

case SYNC:

{

options = optarg[0];

if(options != MUTEX && options != SPINLOCK)

{

fprintf(stderr, "Error: Invalid Sync Option. \n");

exit(1);

}

break;

}

case LISTS:

{

numlists = atoi(optarg);

break;

}

default:

{

fprintf(stderr, "Error: Invalid option. \n");

exit(1);

break;

}

}

}

if(options == MUTEX)

{

if(pthread\_mutex\_init(&lock, NULL))

{

fprintf(stderr, "Error: Mutex Initialization failed. \n");

exit(1);

}

}

element\_len = threads \* iterations;

elements = malloc(element\_len \* sizeof(SortedListElement\_t));

if(elements == NULL)

{

fprintf(stderr, "Error: Could not allocate memory for list. \n");

exit(1);

}

char \*\* keys = malloc(element\_len \* sizeof(char\*));

if(keys == NULL)

{

fprintf(stderr, "Error: Could not allocate memory for keys array. \n");

exit(1);

}

for(int i = 0; i < element\_len; i ++)

{

keys[i] = malloc(10 \* sizeof(char));

if(keys[i] == NULL)

{

fprintf(stderr, "Error: Could not allocate memory for keys. \n");

exit(1);

}

for(int j = 0; j < 9; j ++)

{

keys[i][j] = rand() % 28 + 'A';

}

keys[i][9] = '\0';

elements[i].key = keys[i];

}

lists = malloc(numlists \* sizeof(SortedList\_t));

for(int i = 0; i < numlists; i ++)

{

lists[i].prev = &lists[i];

lists[i].next = &lists[i];

lists[i].key = NULL;

}

pthread\_t \* id = malloc(sizeof(pthread\_t) \* threads);

int \* idthreads = malloc(sizeof(int) \* threads);

threadTimes = malloc(sizeof(long long) \* threads);

for(int i = 0; i < threads; i ++)

{

threadTimes[i] = 0;

}

struct timespec start;

clock\_gettime(CLOCK\_MONOTONIC, &start);

for(int i = 0; i < threads; i ++)

{

idthreads[i] = i;

int n = pthread\_create(&id[i], NULL, &run\_threads, &idthreads[i]);

if(n)

{

fprintf(stderr, "Error: Creation of Threads failed. \n");

free(id);

exit(1);

}

}

for(int j = 0; j < threads; j++)

{

int n = pthread\_join(id[j], NULL);

if(n)

{

fprintf(stderr, "Error: Joining Threads failed. \n");

exit(1);

}

}

struct timespec stop;

clock\_gettime(CLOCK\_MONOTONIC, &stop);

fprintf(stdout, "list");

switch(opt\_yield)

{

case DELETE\_YIELD:

fprintf(stdout, "-d");

break;

case INSERT\_YIELD:

fprintf(stdout, "-i");

break;

case LOOKUP\_YIELD:

fprintf(stdout, "-l");

break;

case ID\_YIELD:

fprintf(stdout, "-id");

break;

case IL\_YIELD:

fprintf(stdout, "-il");

break;

case DL\_YIELD:

fprintf(stdout, "-dl");

break;

case IDL\_YIELD:

fprintf(stdout, "-idl");

break;

default:

fprintf(stdout, "-none");

break;

}

switch(options)

{

case MUTEX:

fprintf(stdout, "-m");

break;

case SPINLOCK:

fprintf(stdout, "-s");

break;

default:

fprintf(stdout, "-none");

break;

}

long long time = stop.tv\_nsec - start.tv\_nsec;

time += (stop.tv\_sec - start.tv\_sec) \* 1000000000;

long long ops = threads \* iterations \* 3;

long long total = 0;

for(int i = 0; i < threads; i ++)

{

total += threadTimes[i];

}

long long averageTime = total / threads;

printf(",%d,%d,%d,%lld,%lld,%lld,%lld\n", threads, iterations, numlists,

ops, time, time / ops, averageTime);

free(id);

free(elements);

if(options == MUTEX)

{

pthread\_mutex\_destroy(&lock);

}

exit(0);

}

**Lab 3a**

/\*

NAME: Luke Jung, Justin Jeon

EMAIL: lukejung@ucla.edu, justinjeon151@gmail.com

ID: 904982644, 205008698

\*/

#include <math.h>

#include <stdio.h>

#include <stdint.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdint.h>

#include <time.h>

#include <sys/types.h>

#include "ext2\_fs.h"

#define BYTE\_SIZE 8

// globals

int fd;

struct ext2\_super\_block super;

uint32\_t groups;

uint32\_t blocks\_count;

uint32\_t inodes\_count;

uint32\_t block\_size;

uint32\_t inode\_size;

uint32\_t blocks\_per\_group;

uint32\_t inodes\_per\_group;

uint32\_t first\_data\_block;

// function declaration

void superblock();

void format\_time(time\_t sec, char \* output);

void group\_summary(uint32\_t group);

void free\_block\_entries(uint32\_t block\_bitmap, uint32\_t group);

void free\_inode\_entries(uint32\_t inode\_bitmap, uint32\_t inode\_table, uint32\_t group);

void inode\_summary(uint32\_t inode\_table, uint32\_t index, uint32\_t curr);

void directory\_entries(uint32\_t inode, uint32\_t block);

void indirect\_blocks(uint32\_t curr, uint32\_t i\_block, int level, uint32\_t offset);

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

{

if (argc > 2)

{

fprintf(stderr, "Error: Too many arguments.\n");

exit(1);

}

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

if (fd == -1)

{

fprintf(stderr, "Error: Unable to open file.\n");

exit(1);

}

superblock();

for (uint32\_t i = 0; i < groups; i++)

group\_summary(i);

exit(0);

}

void superblock()

{

// pread to read offset of where the superblock is

pread(fd, &super, sizeof(super), 1024);

blocks\_count = super.s\_blocks\_count;

inodes\_count = super.s\_inodes\_count;

// equation found from http://cs.smith.edu/~nhowe/262/oldlabs/ext2.html

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

inode\_size = super.s\_inode\_size;

blocks\_per\_group = super.s\_blocks\_per\_group;

inodes\_per\_group = super.s\_inodes\_per\_group;

first\_data\_block = super.s\_first\_data\_block;

printf("SUPERBLOCK,%u,%u,%u,%u,%u,%u,%u\n",

blocks\_count, // total number of blocks

inodes\_count, // total number of inodes

block\_size, // block size

inode\_size, // inode size

blocks\_per\_group, // blocks per group

inodes\_per\_group, // inodes per group

super.s\_first\_ino // first non-reserved inode

);

// round up number of groups

groups = ceil((double) blocks\_count/blocks\_per\_group);

}

void format\_time(time\_t sec, char \* output)

{

struct tm \* time = gmtime(&sec);

strftime(output, 18, "%m/%d/%y %H:%M:%S", time);

}

void group\_summary(uint32\_t group)

{

struct ext2\_group\_desc group\_desc;

// pread offset away from superblock, then however many group sizes

pread(fd, &group\_desc, sizeof(group\_desc), 1024 + block\_size + group \* sizeof(group\_desc));

uint32\_t num\_blocks = super.s\_blocks\_per\_group;

uint32\_t remainder\_blocks = super.s\_blocks\_count % num\_blocks;

if (remainder\_blocks && group == groups - 1)

{

num\_blocks = remainder\_blocks;

}

uint32\_t num\_inodes = super.s\_inodes\_per\_group;

uint32\_t remainder\_inodes = super.s\_inodes\_count % num\_inodes;

if (remainder\_inodes && group == groups - 1 )

{

num\_inodes = remainder\_inodes;

}

fprintf(stdout, "GROUP,%u,%u,%u,%u,%u,%u,%u,%u\n",

group, // group number

num\_blocks, // total number of blocks in this group

num\_inodes, //total number of inodes in this group

group\_desc.bg\_free\_blocks\_count, // number of free blcoks

group\_desc.bg\_free\_inodes\_count, //number of free inodes

group\_desc.bg\_block\_bitmap, // block number of free blcok bitmap

group\_desc.bg\_inode\_bitmap, // block number of free inode bitmap

group\_desc.bg\_inode\_table // block number of first block of inode in group

);

// check for free entries now that group bitmaps have been extracted

free\_block\_entries(group\_desc.bg\_block\_bitmap, group);

free\_inode\_entries(group\_desc.bg\_inode\_bitmap, group\_desc.bg\_inode\_table, group);

}

void free\_block\_entries(uint32\_t block\_bitmap, uint32\_t group)

{

char buf[block\_size];

// pread at offset where bitmap is located

pread(fd, buf, block\_size, block\_bitmap \* block\_size);

uint32\_t start = first\_data\_block + group \* blocks\_per\_group;

// doubly listed for loop to parse through bitmap

for(uint32\_t i = 0; i < block\_size; i ++)

{

char c = buf[i];

for(uint32\_t j = 0; j < BYTE\_SIZE; j ++)

{

// if c == 0, report which are free

if(!(c & 1))

fprintf(stdout, "BFREE,%u\n", start);

start++;

c >>= 1;

}

}

}

void free\_inode\_entries(uint32\_t inode\_bitmap, uint32\_t inode\_table, uint32\_t group)

{

uint32\_t length = inodes\_per\_group / BYTE\_SIZE;

char buf[length];

// pread to where inode bitmpap is located

pread(fd, buf, length, inode\_bitmap \* block\_size);

uint32\_t first\_block = group \* inodes\_per\_group + 1;

uint32\_t curr\_block = first\_block;

for(uint32\_t i = 0; i < length; i ++)

{

char c = buf[i];

for(uint32\_t j = 0; j < BYTE\_SIZE; j ++)

{

if(!(c & 1)) // if first bit is 0, print block

fprintf(stdout, "IFREE,%u\n", curr\_block);

else // if block is being used, then print summary info for it

inode\_summary(inode\_table, curr\_block - first\_block, curr\_block);

curr\_block ++;

c >>= 1;

}

}

}

void inode\_summary(uint32\_t inode\_table, uint32\_t index, uint32\_t curr)

{

struct ext2\_inode inode;

// pread inode that is being used to see what kind of inode it is

pread(fd, &inode, sizeof(inode), inode\_table \* block\_size + index \* sizeof(inode));

// check what kind of inode it is and how many links

uint16\_t mode = inode.i\_mode;

uint16\_t links = inode.i\_links\_count;

char file = 'n';

if (mode && links)

{

// Use '&' because the mode can be combined. List of constants can be found in

// https://www.nongnu.org/ext2-doc/ext2.html#I-MODE

if((mode & 0x8000) == 0x8000)

file = 'f';

else if((mode & 0x4000) == 0x4000)

file = 'd';

else if((mode & 0xA000) == 0xA000)

file = 's';

char create[18], modified[18], access[18];

format\_time(inode.i\_ctime, create);

format\_time(inode.i\_mtime, modified);

format\_time(inode.i\_atime, access);

fprintf(stdout, "INODE,%u,%c,%o,%u,%u,%u,%s,%s,%s,%u,%u",

curr, // inode number

file, // file type

inode.i\_mode & 0x0FFF, // mode (take only the low 12 bits)

inode.i\_uid, // owner

inode.i\_gid, // group

inode.i\_links\_count, // link count

create, //time of last I-Node change

modified, // modification time

access, //time of last access

inode.i\_size, // file size

inode.i\_blocks //number of blocks

);

// in spec, if a symbolic link with more than 60 bytes, a file type, or a directory,

// print next 15 parameters

if ( (file == 's' && inode.i\_size > 60) || file == 'f' || file == 'd' )

{

for(uint32\_t i = 0; i < EXT2\_N\_BLOCKS; i ++)

{

fprintf(stdout, ",%u", inode.i\_block[i]); // prints block addresses

}

}

fprintf(stdout, "\n");

// print directory entry blocks

for (uint32\_t i = 0; i < EXT2\_NDIR\_BLOCKS; i ++)

{

if(inode.i\_block[i])

if(file == 'd')

directory\_entries(curr, inode.i\_block[i]);

}

// print indirect references, whether it be single, double, or triple indirect

if (inode.i\_block[EXT2\_IND\_BLOCK])

indirect\_blocks(curr, inode.i\_block[EXT2\_IND\_BLOCK], 1, 12);

if (inode.i\_block[EXT2\_DIND\_BLOCK])

indirect\_blocks(curr, inode.i\_block[EXT2\_DIND\_BLOCK], 2, 268);

if (inode.i\_block[EXT2\_TIND\_BLOCK])

indirect\_blocks(curr, inode.i\_block[EXT2\_TIND\_BLOCK], 3, 65804);

}

}

void directory\_entries(uint32\_t inode, uint32\_t block)

{

struct ext2\_dir\_entry dir;

for(uint32\_t i = 0; i < block\_size; i += dir.rec\_len)

{

// pread each inode to extract info of each directory

pread(fd, &dir, sizeof(dir), block \* block\_size + i);

if(dir.inode)

{

fprintf(stdout, "DIRENT,%u,%u,%u,%u,%u,'%s'\n",

inode, // parent inod enumber

i, // logical byte offset

dir.inode, // inode number of referenced file

dir.rec\_len, // entry length

dir.name\_len, // name length

dir.name // name

);

}

}

}

void indirect\_blocks(uint32\_t curr, uint32\_t i\_block, int level, uint32\_t offset)

{

uint32\_t blocks\_count = block\_size/sizeof(uint32\_t);

uint32\_t buf[blocks\_count];

// set each value of buf to 0

for (uint32\_t i = 0; i < blocks\_count; i++)

buf[i] = 0;

// pread into where each path is located

pread(fd, buf, block\_size, 1024 + (i\_block - 1) \* block\_size);

for (uint32\_t i = 0; i < blocks\_count; i++)

{

if (buf[i])

{

printf("INDIRECT,%u,%u,%u,%u,%u\n",

curr, // I-node number of owning file

level, // level of indirection

offset+i, // logical block offset

i\_block, // block number of the indirect block

buf[i] // block number of the referenced block

);

if (level == 2)

{

// use recursion to find next address and subtract level by 1

// and increase offset on basis of level from Piazza post

indirect\_blocks(curr, buf[i], level-1, offset);

offset+=256;

}

else if (level == 3)

{

indirect\_blocks(curr, buf[i], level-1, offset);

offset+= 65536;

}

}

}

}

**Lab 3b**

#!/usr/bin/python

import sys

import csv

from sets import Set

free\_blocks = set()

allocated\_inodes = set()

free\_inodes = set()

references = dict()

global exit\_status

# for finding duplicate blocks

class BlockInfo:

def \_\_init\_\_(self, inode, offset, level):

self.inode = inode

self.offset = offset

self.level = level

# superblock class for global data

class SuperBlock:

def \_\_init\_\_(self, properties):

self.blocks\_count = int(properties[1])

self.inodes\_count = int(properties[2])

self.block\_size = int(properties[3])

self.inode\_size = int(properties[4])

self.blocks\_per\_group = int(properties[5])

self.inodes\_per\_group = int(properties[6])

self.first\_nonreserved\_inode = int(properties[7])

class Inode:

def \_\_init\_\_(self, properties):

self.inode\_num = int(properties[1])

self.type = properties[2]

self.mode = properties[3]

self.link\_count = int(properties[6])

self.addresses = []

for address in properties[12:27]:

addresses = int(address)

self.addresses.append(address)

class Dirent:

def \_\_init\_\_(self, properties):

self.name = properties[6]

self.parent\_inode = int(properties[1])

self.logical\_byte\_offset = int(properties[2])

self.file\_num = int(properties[3])

# check valid pointers in I-nodes, direct blocks, and indirect blocks

def block\_scan(properties, superblock):

inode = int(properties[1])

allocated\_inodes.add(inode)

if properties[0] == "INODE":

# don't analyze symbolic links less than or equal to 60 bytes long

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

return

# direct blocks

offset = 0

for block in properties[12:24]:

block\_address = int(block)

if block\_address:

if block\_address >= superblock.blocks\_count or block\_address < 0:

print "INVALID BLOCK {} IN INODE {} AT OFFSET {}".format(block\_address, inode, offset)

exit\_status = 2

elif block\_address > 0 and block\_address < 8:

print "RESERVED BLOCK {} IN INODE {} AT OFFSET {}".format(block\_address, inode, offset)

exit\_status = 2

else:

block\_info = BlockInfo(inode, offset, 0)

if block\_address not in references:

references[block\_address] = [ block\_info ]

else:

references[block\_address].append(block\_info)

offset = offset + 1

# single indirect block

block\_address = int(properties[24])

if block\_address:

if block\_address >= superblock.blocks\_count or block\_address < 0:

print "INVALID INDIRECT BLOCK {} IN INODE {} AT OFFSET {}".format(block\_address, inode, 12)

exit\_status = 2

elif block\_address > 0 and block\_address < 8:

print "RESERVED INDIRECT BLOCK {} IN INODE {} AT OFFSET {}".format(block\_address, inode, 12)

exit\_status = 2

else:

block\_info = BlockInfo(inode, 12, 1)

if block\_address not in references:

references[block\_address] = [ block\_info ]

else:

references[block\_address].append(block\_info)

# double indirect block

block\_address = int(properties[25])

if block\_address:

if block\_address >= superblock.blocks\_count or block\_address < 0:

print "INVALID DOUBLE INDIRECT BLOCK {} IN INODE {} AT OFFSET {}".format(block\_address, inode, 268)

exit\_status = 2

elif block\_address > 0 and block\_address < 8:

print "RESERVED DOUBLE INDIRECT BLOCK {} IN INODE {} AT OFFSET {}".format(block\_address, inode, 268)

exit\_status = 2

else:

block\_info = BlockInfo(inode, 268, 2)

if block\_address not in references:

references[block\_address] = [ block\_info ]

else:

references[block\_address].append(block\_info)

# triple indirect block

block\_address = int(properties[26])

if block\_address:

if block\_address >= superblock.blocks\_count or block\_address < 0:

print "INVALID TRIPLE INDIRECT BLOCK {} IN INODE {} AT OFFSET {}".format(block\_address, int(properties[1]),

65804)

exit\_status = 2

elif block\_address > 0 and block\_address < 8:

print "RESERVED TRIPLE INDIRECT BLOCK {} IN INODE {} AT OFFSET {}".format(block\_address, int(properties[1]),

65804)

exit\_status = 2

else:

block\_info = BlockInfo(inode, 65804, 3)

if block\_address not in references:

references[block\_address] = [ block\_info ]

else:

references[block\_address].append(block\_info)

else:

level = int(properties[2])

block\_address = int(properties[5])

offset = int(properties[3])

if block\_address:

if level == 1:

strlevel = "INDIRECT"

elif level == 2:

strlevel = "DOUBLE INDIRECT"

else:

strlevel = "TRIPLE INDIRECT"

if block\_address >= superblock.blocks\_count or block\_address < 0:

print "INVALID {} BLOCK {} IN INODE {} AT OFFSET {}".format(strlevel, block\_address, inode, offset)

exit\_status = 2

elif block\_address > 0 and block\_address < 8:

print "RESERVED {} BLOCK {} IN INODE {} AT OFFSET {}".format(strlevel, block\_address, inode, offset)

exit\_status = 2

else:

block\_info = BlockInfo(inode, offset, level)

if block\_address not in references:

references[block\_address] = [ block\_info ]

else:

references[block\_address].append(block\_info)

def directory\_scan(inodes, dirents, superblock):

parents = dict();

parents[2] = 2; #parent of root directory is root

for dirent in dirents:

if dirent.file\_num <= superblock.inodes\_count:

if dirent.name != "'..'" and dirent.name != "'.'":

parents[dirent.file\_num] = dirent.parent\_inode

for inode in inodes:

numReferences = 0;

for dirent in dirents:

if dirent.file\_num == inode.inode\_num:

numReferences+=1

if numReferences != inode.link\_count:

print "INODE {} HAS {} LINKS BUT LINKCOUNT IS {}".format(inode.inode\_num, numReferences, inode.link\_count)

exit\_status = 2

for dirent in dirents:

inodeNum = dirent.file\_num

parentNum = dirent.parent\_inode

# 1 is parentnum 3 is inode num

if dirent.name == "'.'" and inodeNum != parentNum:

print "DIRECTORY INODE {} NAME {} LINK TO INODE {} SHOULD BE {}".format(

parentNum,

dirent.name,

inodeNum,

parentNum)

exit\_status = 2

if dirent.name == "'..'" and inodeNum != parents[parentNum]:

print "DIRECTORY INODE {} NAME {} LINK TO INODE {} SHOULD BE {}".format(

parentNum,

dirent.name,

dirent.file\_num,

parents[parentNum])

exit\_status = 2

for dirent in dirents:

cur\_inode = None

for inode in inodes:

if inode.inode\_num == dirent.file\_num:

cur\_inode = inode

break

if cur\_inode == None and dirent.file\_num > 0 and dirent.file\_num < superblock.inodes\_count:

print "DIRECTORY INODE {} NAME {} UNALLOCATED INODE {}".format(dirent.parent\_inode, dirent.name, dirent.file\_num)

exit\_status = 2

elif cur\_inode == None or (cur\_inode.inode\_num < 1 or cur\_inode.inode\_num > superblock.inodes\_count):

print "DIRECTORY INODE {} NAME {} INVALID INODE {}".format(dirent.parent\_inode, dirent.name, dirent.file\_num)

exit\_status = 2

else:

allocated\_inodes.add(cur\_inode)

def main():

# pre-processing

if len(sys.argv) != 2:

sys.stderr.write("Wrong number of arguments\n")

sys.exit(1)

exit\_status = 0

superblock = None

inodes = []

dirents = []

try:

with open(sys.argv[1], 'rb') as f:

reader = csv.reader(f, delimiter=',')

for line in reader:

if line[0] == "INODE" or line[0] == "INDIRECT":

block\_scan(line, superblock)

if line[0] == "INODE":

inodes.append(Inode(line))

elif line[0] == "BFREE":

free\_blocks.add(int(line[1]))

elif line[0] == "IFREE":

free\_inodes.add(int(line[1]))

elif line[0] == "DIRENT":

dirents.append(Dirent(line))

elif line[0] == "SUPERBLOCK":

superblock = SuperBlock(line)

except IOError:

sys.stderr.write("Could not open file\n")

sys.exit(1)

directory\_scan(inodes, dirents, superblock)

# check free block list for unreferenced and allocated inconsistencies

for block in range(superblock.blocks\_count):

if block not in references and block not in free\_blocks and block >= 8:

print "UNREFERENCED BLOCK {}".format(block)

exit\_status = 2

elif block in references and block in free\_blocks:

print "ALLOCATED BLOCK {} ON FREELIST".format(block)

exit\_status = 2

# check allocated inodes list for allocated inconsistencies

for inode in allocated\_inodes:

if inode in free\_inodes:

print "ALLOCATED INODE {} ON FREELIST".format(inode)

exit\_status = 2

# check free inodes list for unreferenced inconsistencies

for inode in range(superblock.first\_nonreserved\_inode, superblock.inodes\_count):

if inode not in allocated\_inodes and inode not in free\_inodes:

print "UNALLOCATED INODE {} NOT ON FREELIST".format(inode)

exit\_status = 2

# check for duplicates

for block, block\_info\_array in references.items():

if len(block\_info\_array) > 1:

exit\_status = 2

for block\_info in block\_info\_array:

if block\_info.level == 0:

print "DUPLICATE BLOCK {} IN INODE {} AT OFFSET {}".format(block, block\_info.inode,

block\_info.offset)

elif block\_info.level == 1:

print "DUPLICATE INDIRECT BLOCK {} IN INODE {} AT OFFSET {}".format(block, block\_info.inode,

block\_info.offset)

elif block\_info.level == 2:

print "DUPLICATE DOUBLE INDIRECT BLOCK {} IN INODE {} AT OFFSET {}".format(block, block\_info.inode,

block\_info.offset)

else:

print "DUPLICATE TRIPLE INDIRECT BLOCK {} IN INODE {} AT OFFSET {}".format(block, block\_info.inode,

block\_info.offset)

sys.exit(exit\_status)

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

main()

**Lab 4b**

// NAME: Luke Jung

// EMAIL: lukejung@ucla.edu

// ID: 904982644

#include <stdlib.h>

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#include <math.h>

#include <getopt.h>

#include <poll.h>

#include <signal.h>

#include <unistd.h>

#include <sys/errno.h>

#include <sys/time.h>

#include <sys/types.h>

#include <time.h>

#include <mraa.h>

#include <mraa/aio.h>

#define A0 1

#define GPIO\_50 60

#define SCALE 100000.0

FILE\* log\_fd = 0;

int scale = 'F';

int period = 1;

int paused = 0;

int log\_flag = 0;

mraa\_aio\_context temp\_sensor;

mraa\_gpio\_context button;

char buffer[100];

time\_t next = 0;

struct tm \*now;

double measure\_temp()

{

int temp = mraa\_aio\_read(temp\_sensor);

float read = (1023.0/((float) temp) - 1.0) \* SCALE;

float celsius = 1.0/(log(read/SCALE)/4275 + 1/298.15) - 273.15;

float faren = (celsius \* 9)/5 + 32;

if (scale == 'C')

{

return celsius;

}

else

return faren;

}

void shutdown()

{

struct timeval ck;

char buffer[200];

gettimeofday(&ck, 0);

now = localtime(&ck.tv\_sec);

sprintf(buffer, "%02d:%02d:%02d SHUTDOWN\n", now->tm\_hour, now->tm\_min, now->tm\_sec);

fputs(buffer, stdout);

if (log\_flag)

fputs(buffer, log\_fd);

exit(0);

}

int checkPeriod(const char \* string)

{

char periodstr[] = "PERIOD=";

int i = 0;

for(; i < 7; i ++)

{

if(string[i] != periodstr[i])

return 0;

}

unsigned len = strlen(string);

for(unsigned j = 7; j < len - 1; j ++)

{

if (!isdigit(string[j]))

{

return 0;

}

}

char \* num = malloc((len+1) \* sizeof(int));

memcpy( num, &string[7], len);

num[len] = '\0';

return atoi(num);

}

void process(const char \* string)

{

if(!strcmp(string, "START\n"))

{

paused = 0;

if (log\_flag)

fprintf(log\_fd, string);

fflush(log\_fd);

}

else if (!strcmp(string, "STOP\n"))

{

paused = 1;

if (log\_flag)

fprintf(log\_fd, string);

fflush(log\_fd);

}

else if(!strcmp(string, "OFF\n"))

{

if (log\_flag)

fprintf(log\_fd, string);

fflush(log\_fd);

shutdown();

}

else if (!strcmp(string, "SCALE=F\n"))

{

scale = 'F';

if (log\_flag)

fprintf(log\_fd, string);

fflush(log\_fd);

}

else if (!strcmp(string, "SCALE=C\n"))

{

scale = 'C';

if (log\_flag)

fprintf(log\_fd, string);

fflush(log\_fd);

}

else if(strlen(string) > 4 && string[0] == 'L' && string[1] == 'O' && string[2] == 'G')

{

if (log\_flag)

fprintf(log\_fd, string);

fflush(log\_fd);

}

else if(string[0] == 'P' && string[1] == 'E' && string[2] == 'R')

{

int m\_period = checkPeriod(string);

if(m\_period == 0)

{

fprintf(stderr, "Error: Invalid Period.\n");

exit(1);

}

if (log\_flag)

fprintf(log\_fd, string);

fflush(log\_fd);

period = m\_period;

}

else

{

fprintf(stderr, "Error: Invalid command.\n");

exit(1);

}

}

void run(struct pollfd \*pollfds)

{

struct timeval ck;

while(1)

{

double temp = measure\_temp();

gettimeofday(&ck, 0);

// if current ck is greater or equal than last ck second plus period...b

if(ck.tv\_sec >= next && !paused)

{

now = localtime(&(ck.tv\_sec));

sprintf(buffer, "%02d:%02d:%02d %.1f\n", now->tm\_hour, now->tm\_min, now->tm\_sec, temp);

fputs(buffer, stdout);

if(log\_flag)

fputs(buffer, log\_fd);

next = ck.tv\_sec + period;

}

int j = poll(pollfds, 1, 0);

if(j < 0)

{

fprintf(stderr, "Error: Polling error.");

exit(1);

}

if(pollfds[0].revents & POLLIN)

{

char tmp[100];

fgets(tmp, 100, stdin);

process(tmp);

}

if(mraa\_gpio\_read(button))

shutdown();

}

}

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

{

static struct option long\_options[] =

{

{"scale", required\_argument, 0, 's'},

{"period", required\_argument, 0, 'p'},

{"log", required\_argument, 0, 'l'},

{0, 0, 0, 0}

};

int c, option\_index, num;

while((c = getopt\_long(argc, argv, "", long\_options, &option\_index)) != -1)

{

switch (c)

{

case 's':

{

if((optarg[0] != 'C' || optarg[0] != 'F') && strlen(optarg) == 1)

{

scale = optarg[0];

}

else

fprintf(stderr, "Error: Invalid Scale.\n");

break;

}

case 'p':

{

num = atoi(optarg);

if(num > 0)

{

period = num;

}

else

fprintf(stderr, "Error: Negative Period.\n");

break;

}

case 'l':

{

log\_fd = fopen(optarg, "w");

if(log\_fd == NULL)

{

fprintf(stderr, "Error: Could not open file.\n");

}

log\_flag = 1;

break;

}

default:

fprintf(stderr, "Error: Bogus argument.\n");

exit(1);

}

}

// initialize senosrs on beaglebone

temp\_sensor = mraa\_aio\_init(A0);

button = mraa\_gpio\_init(GPIO\_50);

// initialize button to be an input

mraa\_gpio\_dir(button, MRAA\_GPIO\_IN);

struct pollfd pollfds[1];

pollfds[0].fd = STDIN\_FILENO;

pollfds[0].events = POLLIN | POLLHUP | POLLERR;

run(pollfds);

mraa\_aio\_close(temp\_sensor);

mraa\_gpio\_close(button);

return 0;

}

**Lab 4c TCP**

#define \_POSIX\_C\_SOURCE 200809L

#define h\_addr h\_addr\_list[0]

#include <stdlib.h>

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#include <math.h>

#include <getopt.h>

#include <poll.h>

#include <signal.h>

#include <unistd.h>

#include <sys/errno.h>

#include <sys/time.h>

#include <sys/types.h>

#include <time.h>

#include <mraa.h>

#include <mraa/aio.h>

#include <netdb.h>

#include <netinet/in.h>

#include <ctype.h>

#include <sys/socket.h>

#define A0 1

#define SERVER 1

#define SCALE 100000.0

FILE\* log\_fd = 0;

int scale = 'F';

int period = 1;

int paused = 0;

int log\_flag = 0;

mraa\_aio\_context temp\_sensor;

time\_t next = 0;

struct tm \*now;

// server vars

int port = -1;

struct sockaddr\_in serv\_address;

struct hostent \* server;

char \* host = "";

char \* id = "";

int sock;

void printer(const char \* string, int server)

{

if (server)

dprintf(sock, "%s\n", string);

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

fprintf(log\_fd, "%s\n", string);

fflush(log\_fd);

}

double measure\_temp()

{

int temp = mraa\_aio\_read(temp\_sensor);

float read = (1023.0/((float) temp) - 1.0) \* SCALE;

float celsius = 1.0/(log(read/SCALE)/4275 + 1/298.15) - 273.15;

float faren = (celsius \* 9)/5 + 32;

if (scale == 'C')

{

return celsius;

}

else

return faren;

}

void m\_shutdown()

{

struct timeval ck;

char buffer[200];

gettimeofday(&ck, 0);

now = localtime(&ck.tv\_sec);

sprintf(buffer, "%02d:%02d:%02d SHUTDOWN\n", now->tm\_hour, now->tm\_min, now->tm\_sec);

printer(buffer, SERVER);

exit(0);

}

int checkPeriod(const char \* string)

{

char periodstr[] = "PERIOD=";

int i = 0;

for(; i < 7; i ++)

{

if(string[i] != periodstr[i])

return 0;

}

unsigned len = strlen(string);

for(unsigned j = 7; j < len - 1; j ++)

{

if (!isdigit(string[j]))

{

return 0;

}

}

char \* num = malloc((len+1) \* sizeof(int));

memcpy( num, &string[7], len);

num[len] = '\0';

return atoi(num);

}

void process(const char \* string)

{

while(\*string == '\t' || \*string == ' ')

{

string ++;

}

if(!strcmp(string, "START"))

{

paused = 0;

printer(string, 0);

}

else if (!strcmp(string, "STOP"))

{

paused = 1;

printer(string, 0);

}

else if(!strcmp(string, "OFF"))

{

printer(string, 0);

m\_shutdown();

}

else if (!strcmp(string, "SCALE=F"))

{

scale = 'F';

printer(string, 0);

}

else if (!strcmp(string, "SCALE=C"))

{

scale = 'C';

printer(string, 0);

}

else if(strlen(string) > 4 && string[0] == 'L' && string[1] == 'O' && string[2] == 'G')

{

printer(string, 0);

}

else if(string[0] == 'P' && string[1] == 'E' && string[2] == 'R')

{

int m\_period = checkPeriod(string);

if(m\_period == 0)

{

fprintf(stderr, "Error: Invalid Period.\n");

exit(1);

}

period = m\_period;

printer(string, 0);

}

else

{

fprintf(stderr, "Error: Invalid command %s.\n", string);

exit(1);

}

}

void get\_server\_input()

{

char \* input = (char \*) malloc(sizeof(char) \* 1024);

char \* copy = input; // create a copy of the input string to parse

// ret is the number of bytes that read caught

int ret = read(sock, input, 256);

if (ret > 0)

input[ret] = 0; // where text for buffer ends, place a 0

while (copy < &input[ret]) // while the copy's address is before the end of the input

{

// create incr string to find each the end of each word

char\* incr = copy;

while (incr < &input[ret] && \*incr != '\n')

{

incr++;

}

\*incr = 0;

process(copy); // process null string, then each new word that comes up

copy = &incr[1];

}

}

void connect\_to\_server()

{

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

if(sock < 0)

{

fprintf(stderr, "Error: Client could not create socket.\n");

exit(1);

}

server = gethostbyname(host);

if(server == NULL)

{

fprintf(stderr, "Error: Host not found.\n");

}

unsigned size = sizeof(serv\_address);

memset((void \*) &serv\_address, 0, size);

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

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

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

int num = connect(sock, (struct sockaddr \*) &serv\_address, size);

if ( num < 0 )

{

fprintf(stderr, "Error: Client could not connect to server.\n");

exit(1);

}

}

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

{

static struct option long\_options[] =

{

{"scale", required\_argument, 0, 's'},

{"period", required\_argument, 0, 'p'},

{"log", required\_argument, 0, 'l'},

{"host", required\_argument, 0, 'h'},

{"id", required\_argument, 0, 'i'},

{0, 0, 0, 0}

};

int c, option\_index, num;

while((c = getopt\_long(argc, argv, "", long\_options, &option\_index)) != -1)

{

switch (c)

{

case 's':

{

if((optarg[0] != 'C' || optarg[0] != 'F') && strlen(optarg) == 1)

{

scale = optarg[0];

}

else

fprintf(stderr, "Error: Invalid Scale.\n");

break;

}

case 'p':

{

num = atoi(optarg);

if(num > 0)

{

period = num;

}

else

fprintf(stderr, "Error: Negative Period.\n");

break;

}

case 'l':

{

log\_fd = fopen(optarg, "w+");

if(log\_fd == NULL)

{

fprintf(stderr, "Error: Could not open log file.\n");

exit(1);

}

log\_flag = 1;

break;

}

case 'h':

{

host = optarg;

if (strlen(host) == 0)

{

fprintf(stderr, "Error: Host Argument is empty.\n");

exit(1);

}

break;

}

case 'i':

{

id = optarg;

if (strlen(id) != 9)

{

fprintf(stderr, "Error: ID is not 9 digits.\n");

exit(1);

}

break;

}

default:

fprintf(stderr, "Error: Bogus argument.\n");

exit(1);

}

}

if (optind < argc)

{

port = atoi(argv[optind]);

if (port < 1)

{

fprintf(stderr, "Error: Port is Invalid.\n");

exit(1);

}

}

connect\_to\_server();

char buf[20];

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

printer(buf, SERVER);

// initialize senosrs on beaglebone

temp\_sensor = mraa\_aio\_init(A0);

struct pollfd pollfds[1];

pollfds[0].fd = sock;

pollfds[0].events = POLLIN | POLLHUP | POLLERR;

struct timeval ck;

char buffer[100];

while(1)

{

gettimeofday(&ck, 0);

// if current ck is greater or equal than last ck second plus period...

if(ck.tv\_sec >= next && !paused)

{

double temp = measure\_temp();

now = localtime(&(ck.tv\_sec));

sprintf(buffer, "%02d:%02d:%02d %.1f", now->tm\_hour, now->tm\_min, now->tm\_sec, temp);

printer(buffer, SERVER);

next = ck.tv\_sec + period;

}

int poll\_ret = poll(pollfds, 1, 0);

if(poll\_ret < 0)

{

fprintf(stderr, "Error: Polling error.");

exit(1);

}

if(poll\_ret)

{

get\_server\_input();

}

}

mraa\_aio\_close(temp\_sensor);

return 0;

}

**Lab 4c TLS**

#define \_POSIX\_C\_SOURCE 200809L

#define h\_addr h\_addr\_list[0]

#include <stdlib.h>

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#include <math.h>

#include <getopt.h>

#include <poll.h>

#include <signal.h>

#include <unistd.h>

#include <sys/errno.h>

#include <sys/time.h>

#include <sys/types.h>

#include <time.h>

#include <mraa.h>

#include <mraa/aio.h>

#include <netdb.h>

#include <netinet/in.h>

#include <openssl/ssl.h>

#include <ctype.h>

#include <sys/socket.h>

#define A0 1

#define SERVER 1

#define SCALE 100000.0

FILE\* log\_fd = 0;

int scale = 'F';

int period = 1;

int paused = 0;

mraa\_aio\_context temp\_sensor;

char buffer[100];

// time vars

time\_t next = 0;

struct tm \*now;

// server vars

int port = -1;

struct sockaddr\_in serv\_address;

struct hostent \* server;

char \* host = "";

char \* id = "";

int sock;

// ssl

SSL \* ssl = NULL;

void printer(const char \* string, int server)

{

if (server)

{

char buf[strlen(string) + 1];

sprintf(buf, "%s\n", string);

SSL\_write(ssl, buf, strlen(string) + 1);

}

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

fprintf(log\_fd, "%s\n", string);

fflush(log\_fd);

}

double measure\_temp()

{

int temp = mraa\_aio\_read(temp\_sensor);

float read = (1023.0/((float) temp) - 1.0) \* SCALE;

float celsius = 1.0/(log(read/SCALE)/4275 + 1/298.15) - 273.15;

float faren = (celsius \* 9)/5 + 32;

if (scale == 'C')

{

return celsius;

}

else

return faren;

}

void m\_shutdown()

{

struct timeval ck;

char buffer[200];

gettimeofday(&ck, 0);

now = localtime(&ck.tv\_sec);

sprintf(buffer, "%02d:%02d:%02d SHUTDOWN\n", now->tm\_hour, now->tm\_min, now->tm\_sec);

printer(buffer, SERVER);

// free ssl vars when shutdown

SSL\_shutdown(ssl);

SSL\_free(ssl);

exit(0);

}

int checkPeriod(const char \* string)

{

char periodstr[] = "PERIOD=";

int i = 0;

for(; i < 7; i ++)

{

if(string[i] != periodstr[i])

return 0;

}

unsigned len = strlen(string);

for(unsigned j = 7; j < len - 1; j ++)

{

if (!isdigit(string[j]))

{

return 0;

}

}

char \* num = malloc((len+1) \* sizeof(int));

memcpy( num, &string[7], len);

num[len] = '\0';

return atoi(num);

}

void process(const char \* string)

{

while(\*string == '\t' || \*string == ' ')

{

string ++;

}

if(!strcmp(string, "START"))

{

paused = 0;

printer(string, 0);

}

else if (!strcmp(string, "STOP"))

{

paused = 1;

printer(string, 0);

}

else if(!strcmp(string, "OFF"))

{

printer(string, 0);

m\_shutdown();

}

else if (!strcmp(string, "SCALE=F"))

{

scale = 'F';

printer(string, 0);

}

else if (!strcmp(string, "SCALE=C"))

{

scale = 'C';

printer(string, 0);

}

else if(strlen(string) > 4 && string[0] == 'L' && string[1] == 'O' && string[2] == 'G')

{

printer(string, 0);

}

else if(string[0] == 'P' && string[1] == 'E' && string[2] == 'R')

{

int m\_period = checkPeriod(string);

if(m\_period == 0)

{

fprintf(stderr, "Error: Invalid Period.\n");

exit(1);

}

period = m\_period;

printer(string, 0);

}

else

{

fprintf(stderr, "Error: Invalid command %s.\n", string);

exit(1);

}

}

void get\_server\_input()

{

char \* input = (char \*) malloc(sizeof(char) \* 1024);

char \* copy = input; // create a copy of the input string to parse

// ret is the number of bytes that read caught

int ret = SSL\_read(ssl, input, 256);

if (ret > 0)

input[ret] = 0; // where text for buffer ends, place a 0

while (copy < &input[ret]) // while the copy's address is before the end of the input

{

// create incr string to find each the end of each word

char\* incr = copy;

while (incr < &input[ret] && \*incr != '\n')

{

incr++;

}

\*incr = 0;

process(copy); // process null string, then each new word that comes up

copy = &incr[1];

}

}

void connect\_to\_server()

{

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

if(sock < 0)

{

fprintf(stderr, "Error: Client could not create socket.\n");

exit(1);

}

server = gethostbyname(host);

if(server == NULL)

{

fprintf(stderr, "Error: Host not found.\n");

}

unsigned size = sizeof(serv\_address);

memset((void \*) &serv\_address, 0, size);

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

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

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

int num = connect(sock, (struct sockaddr \*) &serv\_address, size);

if ( num < 0 )

{

fprintf(stderr, "Error: Client could not connect to server.\n");

exit(1);

}

// initialize SSL connection

SSL\_library\_init(); // loads encryption and hash algorithms

SSL\_load\_error\_strings(); // loads errors strings

OpenSSL\_add\_all\_algorithms(); // loads rest of algorithms

SSL\_CTX \* context = SSL\_CTX\_new(TLSv1\_client\_method()); // create new SSL\_CTX object

if(context == NULL)

{

fprintf(stderr, "Error: Could not create SSL Object. \n");

exit(2);

}

ssl = SSL\_new(context); // create new ssl struct to keep data

if(ssl == NULL)

{

fprintf(stderr, "Error: Could not initalize SSL.\n");

exit(2);

}

if (!SSL\_set\_fd(ssl, sock)) // set the ssl together with the sock fd

{

fprintf(stderr, "Error: Could not set sock fd with SSL.\n");

exit(2);

}

if (SSL\_connect(ssl) != 1) // connect ssl and make sure things run smoothly

{

fprintf(stderr, "Error: Could not connect SSL.\n");

exit(2);

}

}

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

{

static struct option long\_options[] =

{

{"scale", required\_argument, 0, 's'},

{"period", required\_argument, 0, 'p'},

{"log", required\_argument, 0, 'l'},

{"host", required\_argument, 0, 'h'},

{"id", required\_argument, 0, 'i'},

{0, 0, 0, 0}

};

int c, option\_index, num;

while((c = getopt\_long(argc, argv, "", long\_options, &option\_index)) != -1)

{

switch (c)

{

case 's':

{

if((optarg[0] != 'C' || optarg[0] != 'F') && strlen(optarg) == 1)

{

scale = optarg[0];

}

else

fprintf(stderr, "Error: Invalid Scale.\n");

break;

}

case 'p':

{

num = atoi(optarg);

if(num > 0)

{

period = num;

}

else

fprintf(stderr, "Error: Negative Period.\n");

break;

}

case 'l':

{

log\_fd = fopen(optarg, "w+");

if(log\_fd == NULL)

{

fprintf(stderr, "Error: Could not open log file.\n");

exit(1);

}

break;

}

case 'h':

{

host = optarg;

if (strlen(host) == 0)

{

fprintf(stderr, "Error: Host Argument is empty.\n");

exit(1);

}

break;

}

case 'i':

{

id = optarg;

if (strlen(id) != 9)

{

fprintf(stderr, "Error: ID is not 9 digits.\n");

exit(1);

}

break;

}

default:

fprintf(stderr, "Error: Bogus argument.\n");

exit(1);

}

}

if (optind < argc)

{

port = atoi(argv[optind]);

if (port < 1)

{

fprintf(stderr, "Error: Port is Invalid.\n");

exit(1);

}

}

connect\_to\_server();

char buf[20];

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

printer(buf, SERVER);

// initialize senosrs on beaglebone

temp\_sensor = mraa\_aio\_init(A0);

struct pollfd pollfds[1];

pollfds[0].fd = sock;

pollfds[0].events = POLLIN | POLLHUP | POLLERR;

struct timeval ck;

while(1)

{

gettimeofday(&ck, 0);

// if current ck is greater or equal than last ck second plus period...

if(ck.tv\_sec >= next && !paused)

{

double temp = measure\_temp();

now = localtime(&(ck.tv\_sec));

sprintf(buffer, "%02d:%02d:%02d %.1f", now->tm\_hour, now->tm\_min, now->tm\_sec, temp);

printer(buffer, SERVER);

next = ck.tv\_sec + period;

}

int poll\_ret = poll(pollfds, 1, 0);

if(poll\_ret < 0)

{

fprintf(stderr, "Error: Polling error.\n");

exit(1);

}

if(poll\_ret)

{

get\_server\_input();

}

}

mraa\_aio\_close(temp\_sensor);

exit(0);

}