Operating Systems Assignment

ELIJAH COMBES

Document version: 1.1 (2015-11-15)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Combes	Student ID:	19142519
Other name(s):	Elijah		
Unit name:	Operating Systems	Unit ID:	COMP2006
Lecturer / unit coordinator:	Dr Sie Teng Soh	Tutor:	
Date of submission:	18/05/2020	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: Elijah Combes Date of signature: 18/05/2020

(By submitting this form, you indicate that you agree with all the above text.)

Contents

Source Code

LiftSimulator.h

```
// LiftSimulator.c header file containing struct and fucntion declarations and
 definitions

/* each represents an index of an array that this data is stored in
   all are variables that must be kept track of by each lift */
#define CURRENT_FLOOR 0
#define PREVIOUS_FLOOR 1
#define TOTAL_MOVEMENT 2
#define REQUESTS_SERVED 3
#define REQUEST_MOVEMENT 4
```

```
#define LIFT_NUMBER 5

/* data */
#define NUM_TASKS 4 // 4 threads ( lift_R, lift1, lift2, lift3)

void *request(void *ptr);
int validateThreadCreation(int s);
```

LiftSimulatorA.c

```
/*  AUTHOR: Elijah Combes
    NAME: LiftSimulatorA.c
    PURPOSE: Multithreaded program to simulate the operations of three lifts/e
levators running concurrently
            to service requests from different floors from a building. Writes
the details of each operation
            and each request to a file( "sim_out" ) and terminates when all re
quests from the input file
            and in the buffer of requests have been serviced.
*/

#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#include "BufferOperations.h"
#include "LiftSimulator.h"
#include "FileReader.h"


struct thread_info {    /* Used as argument to thread_start() */
           pthread_t thread_id;        /* ID returned by pthread_create() */
           int       thread_num;       /* Application-defined thread # */
           char     *argv_string;      /* From command-line argument */
       };

void *lift(void *ptr);
int getThreadId(pthread_t thread);

Buffer *buf; // pointer to buffer
pthread_mutex_t lock1, lock2; // lock for access to the buffer
pthread_cond_t bufferEmpty;
pthread_cond_t bufferFull;
struct thread_info *tInfo;
int *t; // time
int *totalReq; // total number of requests
```

```
int *totalMove; // total number of movements by all lifts
int done = FALSE;

int main(int argc, char *argv[])
{
    int s,i,j;
    pthread_attr_t attr;
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);
    pthread_cond_init(&bufferEmpty, NULL);
    pthread_cond_init(&bufferFull, NULL);
    pthread_t tid[NUM_TASKS]; // stores thread id's
   // Request *requests;
    t = (int*)malloc(sizeof(int));

    if( argc != 3)
    {
        printf("Please provide the buffer size(m) and time(t)\n");
        printf("run using ./LiftSimulator m t\n");
    }
    else
    {
        buf = (Buffer*)malloc(sizeof(Buffer)); // create space for Buffer
        buf->size = atoi(argv[1]);
        *t = atoi(argv[2]);
        totalMove = (int*)malloc(sizeof(int));
        totalReq = (int*)malloc(sizeof(int));
        if(buf->size < 1)
        {
            printf("Error: please ensure the buffer size is >= 1\n");
        }
        else if(*t < 0)
        {
            printf("Error: time must be >= 0\n");
        }
        else
        {
            // initalising buffer
            buf->requests = (Request*)malloc(sizeof(Request) * (buf-
>size)); // create space to hold each request

            buf->numReq = 0; // initialise number of requests to 0
            buf->head = -1;
            buf->tail = -1;
            s = pthread_attr_init(&attr);

            tInfo = calloc(4,sizeof(struct thread_info)); // create space for
each of the 4 threads info
```

```
            s = pthread_create(&tInfo[0].thread_id,NULL,&request, buf-
>requests); // create Lift_R request thread
            validateThreadCreation(s);

            for(i = 1; i < NUM_TASKS; i++)
            {
                s = pthread_create(&tInfo[i].thread_id,NULL,&lift, &tInfo[i]);
 // create 3 threads, one for each lift
                tInfo[i].thread_num = i;
                validateThreadCreation(s);
            }

            pthread_join(tInfo[0].thread_id,NULL); // join threads so main mus
t wait for their completion

            pthread_join(tInfo[2].thread_id,NULL);
            pthread_join(tInfo[1].thread_id,NULL);
            pthread_join(tInfo[3].thread_id,NULL);
        }
       // pthread_exit(NULL);
        pthread_mutex_destroy(&lock1);
        free(buf);
        free(tInfo);
        writeTotals(totalMove,totalReq); // writes totdal to file
        free(totalMove);
        free(totalReq);
    }
    free(t);
    printf("Simulation Complete\n");
    printf("Please refer to the file 'sim_out' for results\n");
    return 0;
}

/*  NAME: request
    IMPORTS: ptr(void*) - a pointer to a buffer to store data about requests
    EXPORTS: ptr(void*)
    PURPOSE: thread function to open the input file "sim_input" to read in dat
a about a request and save this data in a queue(buf.requests)
             stored in a Buffer. The request data is then written to a file "s
im_out".If the buffer is full then it waits for a request to be removed from t
he queue and
             then proceeds to read in another request and add it to the queue.
 Process repeats until the end of the
             input file is reached and there are no more requests to add. */
void *request(void *ptr)
{
    char* fileName = "sim_input";
    FILE *f;
```

```
    int i = 1;
    int requestNo;
    int numLines = getNumLines(fileName); // represents the number of lines le
ft to read

    Request *newRequest;
    Request *req = (Request*)ptr; // convert the void pointer from thread crea
tion to the requests buffer
    f = fopen(fileName,"r");
    printf("\n");
    if(f == NULL)
    {
        printf("Error opening file. \n");
    }
    else
    {
        // loop whilst the input file still has lines to be read
        while( numLines > 0 )
        {
            // adds a request to the buffer
            pthread_mutex_lock(&lock1); // lock mutex to access CS
            if(!bufferIsFull(buf))
            {
                addRequest(buf); // moves head/tail pointers in buffer
                printf("\n");
                readInputLine(f, &(*(req + buf-
>tail)) ); // reads in new request and adds to buffer
                requestNo++;
                requestWrite( &req[buf-
>tail], &requestNo ); // write the new request to the file
                numLines--
; // decrement numLines when a line is read from the file

                pthread_cond_signal(&bufferFull); // allows lift's to take fro
m the buffer when numReq > 0 && numReq <= 5
                pthread_mutex_unlock(&lock1); // unlock mutex so lift's can ac
quire it
            }
            else
            {
                while(bufferIsFull(buf))
                {
                    pthread_cond_signal(&bufferFull); // signal to lift's that
 the buffer is full/ not empty
                    pthread_cond_wait(&bufferEmpty, &lock1);   // unlock mutex
 and wait until the buffer is empty/not full to acquire it  again
                }
```

```
                pthread_mutex_unlock(&lock1); // unlocks utex as it is already
 acquired at top of while loop
            }
        }
        done = TRUE; // sets done to true so lifts can exit their loop once fi
nished
    }
    fclose(f);
    return ptr;
}

/*  NAME: lift
    IMPORTS: ptr(void*)
    EXPORTS: ptr
    PURPOSE: thread function to simulate operations of a lift. Removes a reque
st from the queue and moves from the source floor
            to the destination floor( by altering data stored in *liftData). T
his data is written to a file for each operation
            that the lift performs( each request ). The function waits whilst
there are no requests
            in the queue. */
void *lift(void *ptr)
{
    int *liftData;
    liftData = (int*)malloc(sizeof(int) * 6); // pointer to store individual l
ift data, indexes defined in "LiftSimulator.h"
    Request* curRequest;

    int tid = getThreadId(pthread_self()); // gets the thread id of this threa
d
    liftData[LIFT_NUMBER] = tid;

    liftData[PREVIOUS_FLOOR] = 1; //start lift at Floor 1

    while( !done || !bufferIsEmpty(buf)) // loop whilst there are still reques
ts to be read from file or buffer is not empty
    {

        pthread_mutex_lock(&lock1); // acquire the mutex
        if( !bufferIsEmpty(buf) )
        {
            curRequest = removeRequest(buf); // gets first request in the q an
d removes it from buffer

            liftData[REQUEST_MOVEMENT] = abs(liftData[CURRENT_FLOOR] - curRequ
est->source) +
                                         abs(curRequest->source - curRequest-
>dest); // calculates floor movement for this particular request
```

```
            // Move Lift
            liftData[CURRENT_FLOOR] = curRequest-
>dest; //floor that lift ends at after moving
            liftData[REQUESTS_SERVED]++; // updates number of requests served
by this lift
            liftData[TOTAL_MOVEMENT] = liftData[TOTAL_MOVEMENT] + liftData[REQ
UEST_MOVEMENT]; // updates total movement

            fileWrite(liftData, curRequest); // write the data for this reques
t to the file

            liftData[PREVIOUS_FLOOR] = liftData[CURRENT_FLOOR]; // update prev
ious floor

            pthread_cond_signal(&bufferFull); // allows other lifts to acquire
 mutex
            pthread_cond_signal(&bufferEmpty); // allows LiftR to add more req
uests to the buffer3
            pthread_mutex_unlock(&lock1); // unlocks mutex so other threads ca
n acquire it

            sleep(*t); // sleep for *t seconds
        }
        else
        {
                while( bufferIsEmpty(buf) ) // loop while the buffer is empty
                {
                    pthread_cond_signal(&bufferEmpty); // signal to liftR that
 the buffer is empty
                    if(done)
                    {
                        break;
                    }
                    pthread_cond_wait(&bufferFull,&lock1); // wait for the buf
fer to be full/ not empty + unlock mutex
                }
                pthread_mutex_unlock(&lock1); // unlock mutex
        }
    }
    *totalMove = *totalMove + liftData[TOTAL_MOVEMENT]; // add this particular
 lift's total movement to the total movement of all lifts
    *totalReq = *totalReq +liftData[REQUESTS_SERVED]; // add this particular l
ift's # of requests served to the # served by all lifts

    free(liftData);   // free malloc'd space
    pthread_exit(NULL);
    return ptr;
}
```

```
/*  NAME: validateThreadCreation
    IMPORTS: s(int) - status returned by pthread_create when making a thread
    EXPORTS: s(int)
    PURPOSE: notifies user of an error during thread creation if the status !=
 0
            at this stage should do more error handling
*/
int validateThreadCreation(int s)
{
    if (s != 0)
        printf("Error: pthread_create\n");
    return s;
}

 /*  NAME: getThreadId
     IMPORTS: thread(pthread_t)
     EXPORTS: int
     PURPOSE: mainly for debugging, converts the thread id into the correspond
ing thread number
             that was assigned to it in main (ie 1, 2 or 3, representing each
of the three lifts)
             returns 100(error) if the thread could not be found. */
 int getThreadId(pthread_t thread)
 {
    int i,s;
     for(i = 0; i < 4;i++)
     {
         s = pthread_equal(thread, tInfo[i].thread_id);
         if(s > 0)
         {
             return tInfo[i].thread_num;
         }
     }
    return 100;
 }
```

LiftSimulatorB.c

```
/*  AUTHOR: Elijah Combes
    NAME: LiftSimulatorB.c
    PURPOSE: program to simulate the operations of three lifts/elevators runni
ng concurrently
            to service requests from different floors from a building. Each li
ft is run by a seperate process. Writes the details of each operation
            and each request to a file( "sim_out" ) and terminates when all re
quests from the input file
            and in the buffer of requests have been serviced.
```

```
*/
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
  #include <sys/mman.h>
       #include <sys/stat.h>
       #include <fcntl.h>
#include <semaphore.h>

#include "BufferOperations.h"
#include "LiftSimulator.h"
#include "FileReader.h"
// struct to contain all data that needs to be in shared memory
typedef struct SharedM{
    Buffer buf; // buffer
    sem_t sem; // sempahore
    int done; // 0(FALSE) if there are still requests to read in, 1(TRUE) if a
ll requests have been read
    int totalReq; // total number of requests served
    int totalMove; // total number of movement completed
} SharedM;

int getProcessNumber(pid_t* pids); // function declarations, here as A and B u
se the same header file
void *lift(void *ptr, pid_t *pids);

int *t;

// MAIN:
//  used to create shared memory and the 4 processes used in this program to r
epresent the 3 lifts and the request creator
int main(int argc, char *argv[])
{
    int s,i,j;
    pid_t *pids;
    Request *requests;
    SharedM *sharedMem = (SharedM*)malloc(sizeof(SharedM));
    pid_t wpid;
    t = (int*)malloc(sizeof(int));

    if( argc != 3) // incorrect number of args
    {
        printf("Please provide the buffer size(m) and time(t)\n");
        printf("run using ./LiftSimulator m t\n");
    }
    else
```

```
    {
        // holds each of the process ids: main process(index 0),request(index
1 etc.) lift: 1,2,3 (ie 5 processes total)
        pids = (pid_t*)malloc(sizeof(pid_t) * NUM_TASKS + 1);

        if(atoi(argv[1]) < 1) // buffer size must be >= 1
        {
            printf("Error: please ensure the buffer size is >= 1\n");
        }
        else if(atoi(argv[2]) < 0) // time must be >= 0
        {
            printf("Error: time must be >= 0\n");
        }
        else
        {
            requests = (Request*)malloc(sizeof(Request) * atoi( argv[1] ) ); /
/create space to store buffer of requests
            // get shared memory for SharedM
            int fd =  shm_open("/sharedMem", O_RDWR | O_CREAT, 0660);
            if(fd < 0)
                printf("Error: shm_open - /x\n");
            if( ftruncate(fd, sizeof(SharedM*)) == -1)
                printf("Error: ftruncate\n");
            if((sharedMem = (SharedM*)mmap(0,sizeof(SharedM*),PROT_READ | PROT
_WRITE, MAP_SHARED, fd, 0)) == MAP_FAILED)
                printf("Error: mmap\n");
            // create shared mem for Request buffer(requests)
            fd =  shm_open("/requests", O_RDWR | O_CREAT, 0660);
            if(fd < 0)
                printf("Error: shm_open - /requests\n");
            if( ftruncate(fd, sizeof(Request*)) == -1)
                printf("Error: ftruncate\n");
            if((requests = (Request*)mmap(0,sizeof(Request*),PROT_READ | PROT_
WRITE, MAP_SHARED, fd, 0)) == MAP_FAILED)
                printf("Error: mmap\n");
            // create shared memory for process id's
            fd =  shm_open("/pids", O_RDWR | O_CREAT, 0660);
            if(fd < 0)
                printf("Error: shm_open - /pids\n");
            if( ftruncate(fd, sizeof(pid_t*)) == -1)
                printf("Error: ftruncate\n");
            if((pids = (pid_t*)mmap(0,sizeof(pid_t*),PROT_READ | PROT_WRITE, M
AP_SHARED, fd, 0)) == MAP_FAILED)
                printf("Error: mmap\n");

            pids[0] = getpid(); // set pid of main
```

```
            sharedMem-
>buf.size = atoi(argv[1]); // assign buffer size from command line
            *t = atoi(argv[2]); // time(seconds) from command line
            sharedMem-
>buf.requests = requests; // add request buffer to the main shared memory Shar
edM
            sharedMem->buf.numReq = 0;

            int semVal = sem_init(&(sharedMem-
>sem),1,1); // initialise semaphore with value 1
            if( semVal == -1 )
            {
                printf("Pid: %d, Error: sem_init\n", getpid());
            }

            //create a process for request()
            pids[1] = fork();
            if( pids[1] < 0 )
            {
                perror("Fork unsuccessful\n");
            }
            if( pids[1] == 0) // child process
            {
                pids[1] = getpid();
                request(sharedMem);
                _exit(3);

            }
                // create a process for each lift and run the lift function fo
r each one
                for(i = 2; i < NUM_TASKS + 1; i++)
                {
                    // create lift processes
                    pids[i] = fork();
                    if( pids[i] < 0 )
                    {
                        perror("Fork unsuccessful\n");
                    }
                    if( pids[i] == 0)  // child process
                    {
                        pids[i] = getpid();
                        lift(sharedMem,pids);
                        _exit(3);
                    }
                }
        }
        int status = 0;
        for(int k = 0; k < 4; k++)
```

```
        {
            wait(NULL);
        }
        writeTotals(&sharedMem->totalMove, &sharedMem->totalReq);
        sem_destroy(&(sharedMem->sem));
        shm_unlink("/requests");
        shm_unlink("/sharedMem");
        free(t);
        printf("Simulation Complete\n");
        printf("Please refer to the file 'sim_out' for results\n");
    }
    return 0;
}

/*  NAME: request
    IMPORTS: ptr(void*) - a pointer to a shared memory
    EXPORTS: void
    PURPOSE: function to open the input file "sim_input" to read in data about
 a request and save this data in a queue(sharedMem->buf.requests)
             stored in a Buffer. The request data is then written to a file "s
im_out".If the buffer is full then it waits for a request to be removed from t
he queue and
             then proceeds to read in another request and add it to the queue.
 Process repeats until the end of the
             input file is reached and there are no more requests to add. */
void *request(void *ptr)
{
    char* fileName = "sim_input";
    FILE *f;
    int i = 1;
    int requestNo = 0;
    int numLines;

    SharedM *sharedMem = ptr;
    Buffer *buf = &(sharedMem->buf);
    Request *req = buf-
>requests; // convert the void pointer from thread creation to the requests bu
ffer

    numLines = getNumLines(fileName); //fgets in this method causes process to
 terminate
    f = fopen(fileName,"r"); // open the input file

    while( numLines > 0 )// loop whilst the input file still has lines to be r
ead
    {
        sem_wait(&(sharedMem->sem)); // wait for semaphore
        if(!bufferIsFull(buf))
```

```
        {
            addRequest(buf); // updates head/tail pointers in buffer ready for
 a new request
            printf("\n");
            readInputLine(f, &(*(req + buf-
>tail)) ); // read a request from the file and add to buffer
            requestNo++;
            requestWrite( &req[buf-
>tail], &requestNo );  // write the new request to the output file
            numLines--; // when numLines reaches 0 the last line has been read

            sem_post(&(sharedMem->sem)); // increment semaphore
        }
        else
        {
            while(bufferIsFull(buf)) //
            {
                sem_post(&(sharedMem->sem)); // increment semaphore
                sem_wait(&(sharedMem-
>sem)); // wait for semaphore/ added to ready q
            }
            sem_post(&(sharedMem->sem));
        }
    }
    sem_wait(&(sharedMem->sem));
    sharedMem->done = TRUE; // all requests have been added to the buffer
    sem_post(&(sharedMem->sem));
      fclose(f);
    return ptr;
}

/*  NAME: lift
    IMPORTS: ptr(void*) - pointer to shared memory, pids - process id's of pro
cesses created in this program
    EXPORTS: void
    PURPOSE: function to simulate operations of a lift. Removes a request from
 the queue and moves from the source floor
            to the destination floor( by altering data stored in *liftData). T
his data is written to a file for each operation
            that the lift performs( each request ). The function waits whilst
there are no requests
            in the queue. */
void *lift(void *ptr, pid_t *pids)
{
    int *liftData;
    SharedM *sharedMem = (SharedM*)ptr;
    liftData = (int*)malloc(sizeof(int) * 6);
    Request* curRequest;
```

```
    sem_wait(&(sharedMem->sem));
    liftData[LIFT_NUMBER] = getProcessNumber(pids); // gets lift number/proces
s number
    sem_post(&(sharedMem->sem));
    liftData[PREVIOUS_FLOOR] = 1; //start lift at Floor 1

    while( !(sharedMem->done) || !bufferIsEmpty(&(sharedMem-
>buf)) ) // loop whilst there are still requests to be read(ie not done), if d
one then check if there are requests left in the buffer
    {
        sem_wait(&(sharedMem->sem));
        if( !bufferIsEmpty(&(sharedMem->buf) ))
        {
            curRequest = removeRequest(&(sharedMem-
>buf)); // gets first request in the q and removes it from buffer
            liftData[REQUEST_MOVEMENT] = abs(liftData[CURRENT_FLOOR] - curRequ
est->source) +
                                         abs(curRequest->source - curRequest-
>dest); // calculates floor movement for this particular request
            //move lift
            liftData[CURRENT_FLOOR] = curRequest-
>dest; //floor that lift ends at after moving
            liftData[REQUESTS_SERVED]++; // updates number of requests served
by this lift
            liftData[TOTAL_MOVEMENT] = liftData[TOTAL_MOVEMENT] + liftData[REQ
UEST_MOVEMENT]; // updates total movement

            fileWrite(liftData, curRequest); // write the data for this reques
t to the file
            liftData[PREVIOUS_FLOOR] = liftData[CURRENT_FLOOR]; // update prev
 floor
            sem_post(&(sharedMem->sem));
            sleep(*t); // sleep for t seconds
        }
        else
        {
                while( bufferIsEmpty(&(sharedMem->buf)) )
                {
                    if(sharedMem-
>done) // if done then exit from this while loop
                    {
                        sem_post(&(sharedMem-
>sem)); // increment semaphore before exiting
                        break;
                    }
                    sem_post(&(sharedMem->sem));
                    sem_wait(&(sharedMem->sem));
```

```
                }
                sem_post(&(sharedMem->sem));
        }
    }
    sem_wait(&(sharedMem->sem));
    sharedMem->totalMove = sharedMem->totalMove + liftData[TOTAL_MOVEMENT];
    sharedMem->totalReq = sharedMem->totalReq + liftData[REQUESTS_SERVED];
    sem_post(&(sharedMem->sem));
    free(liftData);
    return ptr;
}

/*  NAME: getProcessNumber
     IMPORTS: pids - process id's for all created threads in this program
     EXPORTS: int
     PURPOSE: returns the process number of the process that calls it,
              made mainly to number each lift process */
int getProcessNumber(pid_t* pids)
{
    pid_t pid = getpid();
    int processNumber = -1;
    for(int i = 0; i < NUM_TASKS + 1; i++)
    {
        if(pids[i] == pid)
        {
            processNumber = i - 1; // minus one as Lift-1 is at index 2 etc
        }
    }
    return processNumber;
}
```

FileReader.h

```
/* Header file for FileReader.c containing struct and function defs */

int readInputLine(FILE* f,Request *);
void fileWrite();
void requestWrite( Request *curRequest, int *requestNo );
void writeTotals(int* totalMove, int* totalReq);
int getNumLines(char *fileName);

#define indent "    "
#define MAXSIZE 20
```

FileReader.c

```
/* AUTHOR: Elijah Combes
   NAME: FileReader.c
```

```c
   PURPOSE: perform file IO for the lift simulator( LiftSimulatorA/B.c )
            reading input file ("sim_input") containing Request data ( source
and destination floors )
            writing imported data to the output file "sim_out"
            - potentially could have been split up into 2 .c files to separate
 reading and writing functions */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "BufferOperations.h"
#include "LiftSimulator.h"
#include "FileReader.h"

/*  NAME: readInputLine
    IMPORTS: f (FILE*) - pointer to open file, ptr(Request*) - pointer to stor
e the request data
    EXPORTS: done (int) - indicates when the file has no more lines to read
    PURPOSE: read one line from the input file and store the data in a Request
 struct to be returned */
int readInputLine(FILE* f,Request *ptr)
{
    char *line;
    char *token;
    char *fileName = "sim_input";
    int done = FALSE;

    if(f == NULL)
    {
        perror("Error opening file '%s' ");
    }
    else
    {
        // initialises request structure source and dest values
        if(!(fgets(line,MAXSIZE,f) == NULL))
        {
            //set first element tp ptr->source
            token = strtok(line, " ");
            ptr->source = atoi(token); // set the requests source
            token = strtok(NULL," ");
            ptr->dest = atoi(token);
        }
        else
        {
            printf("End of file reached\n)");
            done = TRUE;
        }
```

```
    }
    return done;
}

/* NAME: getNumLines
   IMPORTS: fileName - name of the file to read
   EXPORTS: numLines - number of lines in the file
   PURPOSE: counts the number of lines in a file and returns # as an integer *
/
int getNumLines(char *fileName)
{
    FILE* f1;
    int numLines = 0;
    char *line1 = (char*)malloc(sizeof(char) * MAXSIZE);
    f1 = fopen(fileName,"r");
    if( f1 == NULL )
    {
        perror("Error opening file.");
    }
    else
    {
        // counts number of lines in the file
        while( !(fgets(line1,MAXSIZE,f1) == NULL))  // if next line is not nul
l
        {
            numLines++; // increment number of lines
        }
    }
    fclose(f1);
    free(line1);
    return numLines;
}

/*  NAME: requestWrite
    IMPORTS: curRequest(Request*) - current request to writeto file, requestNo
(int*) - request number
    EXPORTS: none
    PURPOSE: write data about the current request to the file "sim_out" */
void requestWrite(Request *curRequest, int *requestNo)
{
    // print to sim_out
    FILE *filePointer;
    char *outputFile = "sim_out";
    int end;

    filePointer = fopen(outputFile,"a");
    if( filePointer == NULL )
    {
```

```
        perror("Error opening file: '%s'");
    }
    else
    {
        fprintf(filePointer, "-----------------------------------------
\n");
        fprintf(filePointer, "New Lift Request From Floor %d to Floor %d\n",cu
rRequest->source, curRequest->dest);
        fprintf(filePointer, "Request No: %d\n", *requestNo);
        fprintf(filePointer, "-----------------------------------------
\n\n");
        end = fclose(filePointer);
        if( !( end == 0 ) )
        {
            printf("Error writing to file\n");
        }
    }
}

/*  NAME: fileWrite
    IMPORTS: lift(int*) - lift number, curRequest(Request*)
    EXPORTS: none
    PURPOSE: write data about the current lift operation to the output file "s
im_out"   */
void fileWrite(int *lift, Request *curRequest)
{
    FILE* f;
    char* outputFile = "sim_out";
    int end;
    f = fopen(outputFile,"a");
    if( f == NULL )
    {
        perror("Error opening file: '%s' ");
    }
    else
    {
        fprintf(f,"Lift-%d Operation\n", lift[LIFT_NUMBER]);
        fprintf(f,"Previous position: Floor %d\n",lift[PREVIOUS_FLOOR]);
        fprintf(f,"Request: Floor %d to Floor %d\n",curRequest-
>source, curRequest->dest );
        fprintf(f,"Detail operations: \n");
        fprintf(f, "    Go from Floor %d to Floor %d\n",lift[PREVIOUS_FLOOR],
curRequest->source);
        fprintf(f, "    Go from Floor %d to Floor %d\n",curRequest-
>source, curRequest->dest );
        fprintf(f, "    #movement for this request: %d\n",lift[REQUEST_MOVEMEN
T] /* reqMovement*/);
        fprintf(f, "    #request: %d\n",lift[REQUESTS_SERVED] /*totalReq*/);
```

```
        fprintf(f, "    Total #movement: %d\n",lift[TOTAL_MOVEMENT]/* totalMov
e */);
        fprintf(f, "Current position: Floor %d\n\n", lift[CURRENT_FLOOR] /* cu
rFloor */);

        end = fclose(f);
        if( !(end == 0) )
        {
            printf("Error when writing to file\n");
        }
    }
}

/*  NAME: writeTotals
    IMPORTS: totalMove(int*), totalReq(int*)
    EXPORTS: none
    PURPOSE: write the total movements and total requests for all lifts 1,2 an
d 3 to the outputFile */
void writeTotals(int* totalMove, int* totalReq)
{
    FILE *f;
    char* outputFile = "sim_out";
    f = fopen(outputFile,"a");
    if( f == NULL )
    {
        perror("Error opening file '%s'\n");
    }
    else
    {
        fprintf(f, "Total number of requests: %d\n", *totalReq);
        fprintf(f, "Total number of movements: %d\n", *totalMove);
    }
}
```

BufferOperations.h

```
// Header file for BufferOperations.c containing struct and function definitio
ns
typedef struct Request{
    //request data
    int source; // source floor
    int dest;
          //destination floor
} Request;

typedef struct Buffer{
    Request *requests; //pointer to array of requests
    int numReq; // number of requests currently in the buffer
```

```
    int size; // size of the buffer determined by command line input
    int head; // head of request queue
    int tail; // tail of
} Buffer;

int bufferIsFull();
struct Request* removeRequest();
void addRequest( Buffer *buf);
int bufferIsEmpty();

#define FALSE 0
#define TRUE !FALSE
```

BufferOperations.c

```
/*  AUTHOR: Elijah Combes
    PURPOSE: Contains functions to act on a Buffer allowing the array of Reque
sts in
            the Buffer to act like a circular queue */
#include <stdlib.h>
#include <stdio.h>

#include "BufferOperations.h"
#include "LiftSimulator.h"

/* NAME: addRequest
   IMPORTS: buf - the buffer containing all requests and relevant data
            newRequest - pointer to the new request to be added
   EXPORTS: newRequest - pointer to where the new request must go
   PURPOSE: returns a pointer to the space in the buffer where the new request
 should be added. updates the buffer/queue data,
            increments numReq, updates head and or tail pointers   */
void addRequest( Buffer *buf)
{
    if( bufferIsFull(buf)) // this should never be true if logic in LiftSimula
tors is correct
    {
        printf("Error buffer is full, cannot add request\n");
    }
    else
    {
        if(buf->tail == (buf->size - 1))
        {
            buf->tail = 0;
        }
        else
        {
            buf->tail = buf->tail + 1;
```

```
        }
        buf->numReq++; // increment number of requests currently in buffer
    }
    if( buf->head == -1 ) // first element ever added to buffer
    {
        buf->head++;
    }
}

/*  NAME: removeRequest
    IMPORTS: buf(Buffer*) pointer to the buffer to operate on
    EXPORTS: r(Request) - the request that has been removed from the buffer
    PURPOSE: to remove the head element(Request) in the buffer and return it t
o the calling fucntion
            also to update the head pointer to track the first element in the
queue/buffer */
Request* removeRequest(Buffer *buf)
{
    Request *r = &(buf->requests[buf->head]);
    if((buf->head + 1) == buf->size)
    {
        buf->head = 0;
    }
    else
    {
        buf->head++;
    }
    buf->numReq--;
    return r;
}

/*  NAME: bufferIsFull
    IMPORTS: buf(Buffer*)
    EXPORTS: isFull(int) - boolean defined in BufferOperations.h
    PURPOSE: check if the imported Buffer is full or not */
int bufferIsFull(Buffer *buf)
{
    int isFull = TRUE;
    if(buf->numReq < buf->size )
    {
        isFull = FALSE;
    }
    return isFull;
}

/*  NAME: bufferIsEmpty
    IMPORTS: buf(Buffer*)
    EXPORTS: isEmpty(int) - boolean defined in BufferOperations.h
```

```
    PURPOSE: check if the imported Buffer is Empty or not */
int bufferIsEmpty(Buffer *buf)
{
    int isEmpty = FALSE;
    if(buf->numReq == 0)
    {
        isEmpty = TRUE;
    }
    return isEmpty;
}
```

README and how to run/compile

---------- Lift Simulator ----------

README

Program that simulates the actions of 3 Lifts/elevators who all access a common buffer to service requests

each lift is represented by a separate thread for lift_sim_A OR a separate process for lift_sim_B.

This program is to demonstrate how processes/threads can run concurrently and access shared data

by conforming to mutual exclusion principles.

Input to the program must be specified in a file called 'sim_input'

Results of the program can be found in a file called 'sim_out' after the program has been run.

To compile:

'make all'

To Run:

Thread implementation:

'./lift_sim_A m t' - where m is the size of the buffer which must be >= 1

and t is the amount of seconds a lift takes to complete a request

Process implementation:

'./lift_sim_B m t' - where m is the size of the buffer which must be >= 1

and t is the amount of seconds a lift takes to complete a request

Mutual Exclusion

In order to achieve mutual exclusion in this lift simulator, the program has been designed so that at any given time only a maximum of one process may enter its critical section. In this case the critical section involves accessing buffer of requests which is shared among the processes of this program and includes reading and writing of that buffer, and access to the output file “sim_out”. These must be mutually exclusive in order to prevent dirty reads and overwriting data. LiftSimulatorA.c uses the pthread library to create multiple concurrent threads and synchronize them, whilst LiftSimulatorB.c creates multiple concurrent processes and uses POSIX semaphores to synchronize them. For the purpose of explaining how this is achieved I will walk through the main steps of each process, one for Lift-R and one for representing each of the lifts.

Lift_sim_A

Lift_sim_A uses a mutex lock to prevent a thread from entering its critical section if it is not in possession of the mutex. The critical section in this case involves any access to resources shared amongst the threads which includes access to the output file and mutable global variables: the buffer and the total number of requests and movements for all lifts. Therefore at any given time only a single thread may enter its critical section meaning that only one thread is accessing the shared resources at any given time.

Lift-R: Once the main while loop is entered, pthread_mutex_lock(&lock1), is called to acquire the mutex. Once acquired Lift-R can execute in its critical section, in which it first checks if the buffer is full, if so it begins by reading in a request from the input file “sim_input” and adding that request to the buffer and also writing that request details to the output file. When completed a call to pthread_cond_signal(&bufferFull) is initiated to indicate to the other threads(lifts) that there is > 0 requests in the buffer. This signal is important as otherwise the other threads would be sat waiting for notification that the buffer is not empty causing a deadlock. If the buffer was not full after the first lock was acquired then lift-R checks the condition on an inner while loop where it checks if the buffer is full, if so lift-r must wait until one of the other threads removes a request from the buffer before it can continue to add requests. To do this, first pthread_cond_signal(&bufferFull) is called to notify the other threads/lifts that the buffer is full and they may begin to remove requests once they gain the mutex lock. Then pthread_cond_wait(&bufferEmpty, &lock1) is called to make lift-R release the mutex ,so other threads have a chance to acquire it, and so that lift-R waits at this point until it is signaled by one of the lifts that the buffer is no longer full(&bufferEmpty). Once signaled it will regain the mutex, check the while condition once more and since the buffer will not be full it will exit the while loop and call pthread_mutex_unlock(&lock1) to unlock the mutex as when the outer while loop repeats it will re acquire the mutex. This process repeats until there are no more requests left to add to the buffer.

Lift-1,2,3: The lift function follows a similar design to request() as it mainly contains an outer while loop with and if-else statement inside and an inner while loop in the else statement. Once the outer while loop is entered, pthread_mutex_lock(&lock1) is called to acquire the mutex allowing this lift to enter its critical section. The lift first checks if the buffer has > 0 requests in it, if so it will remove a request from the buffer, update some local values to simulate movement of the lift and then write data about itself and the request that it removed to the output file. Once completed pthread_cond_signal(&bufferFull) will be called to signal to the other lift threads that may be waiting to check if the buffer is full or not. Then pthread_cond_signal(&bufferEmpty) is called to

notify lift-R that the buffer has < the buffer size requests in it(ie. Buffer not full). These two signals allow all threads to run concurrently instead of having lift-R to fill the buffer entirely before another thread can execute or having lift-1,2 or 3 empty a buffer entirely by itself before another thread can execute in critical section. Next pthread_mutex_unlock(&lock1) is called to release the mutex so the other threads can potentially acquire it. If the buffer is empty then the lift will signal to lift-R and release the mutex and wait/sleep until the buffer is no longer empty. Once every request has been serviced, a lift will exit the outer while loop and will acquire a lock to access "totalMove" and "totalReq" which are global variables common to all the threads so this access must be mutually exclusive. The mutex is released after these values are modified and now all critical section tasks are complete.

Lift_sim_B

Lift_sim_B uses POSIX semaphores to ensure mutual exclusion. The semaphore is initialized with a value of 1 which allows one process at a time to enter critical section. Lift_sim_B uses mmap to create a region of shared memory called sharedMem that can be shared amongst each process lift-R, lift1,2 and 3. This shared memory is just a pointer to a struct containing: the buffer, semaphore and three integer values that must be shared among processes and access to this pointer is mutually exclusive. Again Lift_sim_B uses a similar design to lift_sim_A in the sense that both request() and lift() functions involve an outer while loop containing an if-else statement and an inner while loop inside the else block. For simplicity of explaining I will refer to the semaphore as "&sem" in code examples.

Lift-R: Inside the outer while loop sem_wait(&sem) is called to decrement the semaphore value which will either allow lift-R to enter critical section(semaphore will decrease from 1 to 0) or add lift-R to the waiting queue by making the semaphore value negative. Once allowed to enter critical section the semaphore value will be < 1 which means every other process that calls sem_wait(&sem) will be added to the waiting queue and unable to access the buffer until lift-R is finished. Lift-R will then check the if condition to see whether there is space available in the buffer, if so, it will add a new request to the buffer and write the request data to the output file. When finished sem_post(&sem) is called to increment the semaphore which will allow the process at the front of the waiting queue to enter critical section. If there is no space available in the buffer then the process will check if this is true and will enter the inner while loop if so. Here the process will increment the semaphore to allow another process to enter its critical section and then call sem_wait(&sem) to be added to the waiting queue as checking the while condition(while(bufferIsFull(buf))) must be mutually exclusive as it accesses the buffer. This sequence of checking if the buffer is full, incrementing the semaphore(sem_post(&sem)) and decrementing the semaphore(sem_wait(&sem)) allows lift-R to continually access the buffer to see how many requests are in it without blocking any another process from accessing it indefinitely. Finally when the inner while loop exits sem_post(&sem) is called to increment the semaphore.

Lift 1,2,3: Inside the inner while loop sem_wait(&sem) is called to decrement the semaphore and allow one lift to enter critical section if no other processes are in the waiting queue. Then the if condition is checked to see if the buffer is not empty. If true this lift will remove a request from the buffer, move the lift and write the request data to the output file before incrementing the semaphore with sem_post(&sem) to allow another process to enter its critical section. If the buffer was instead empty then this lift will enter the inner while loop and increment the semaphore allowing other processes to run in their critical section and then it will decrement the semaphore and add itself to the waiting queue as it must continually check if the buffer is still empty or not and

this must be mutually exclusive as it accesses the buffer. This process repeats until there are no requests left in the input file ant the buffer.

All of these functions ensure mutual exclusion as with any shared memory, whether that be global variables for threads or memory created with mmap for processes, the task must have either the mutex lock or after decrementing the semaphore it must be equal to 0 meaning that 0 more processes can enter their critical section and one process(the one that decremented the semaphore to 0) is currently executing in its critical section. Any other reads of memory are either not shared between the tasks or are shared but are constants or immutable within the program.

Known error cases/bugs

Description for any cases where program is not working correctly

- When program starts the buffer is filled entirely before any requests are removed, program still works correctly however I not sure if we were required to make it so the lifts can begin taking a request as soon as the first request is added. This may be an issue as if you test the program with a buffer size >= the total number of requests, every request will be added to the buffer before any is serviced.
- If tested with a large input file (ie 100 requests) and the value of t = 0 one lift will do the majority of requests. I believe its just because the processes are running so fast that only one is being allowed to enter critical section over and over again. Still works correctly, its just that one lift may serve 80 requests by itself which is not ideal. This works fine when t = 1, the requests served is spread much more evenly.

Sample Input and Output

The program was tested with two main input files, one containing 50 requests and one containing 100 requests and also two main values of “t” 0 and 1 as increasing t further than that just uses more time and doesn’t change the results much. Examples of these can be found below containing the final output of each lift and of the total requests served and movement. Also the entire output files will be in a file with the rest of the program and I will indicate each one’s name below:

INPUT: ./lift_sim_A 5 0 **OUTPUT FILE NAME:** sim_out1

OUTPUT:

```
Lift-2 Operation
Previous position: Floor 4
Request: Floor 1 to Floor 12
Detail operations:
    Go from Floor 4 to Floor 1
    Go from Floor 1 to Floor 12
    #movement for this request: 14
    #request: 15
    Total #movement: 208
Current position: Floor 12

Lift-3 Operation
Previous position: Floor 12
Request: Floor 12 to Floor 3
Detail operations:
    Go from Floor 12 to Floor 12
    Go from Floor 12 to Floor 3
    #movement for this request: 9
    #request: 24
    Total #movement: 299
Current position: Floor 3

Lift-1 Operation
```

```
Lift-1 Operation
Previous position: Floor 15
Request: Floor 7 to Floor 3
Detail operations:
    Go from Floor 15 to Floor 7
    Go from Floor 7 to Floor 3
    #movement for this request: 12
    #request: 11
    Total #movement: 151
Current position: Floor 3

Total number of requests: 50
Total number of movements: 658
```

INPUT: ./ lift_sim_B 5 0 **OUTPUT FILE NAME:** sim_outB1 **OUTPUT:**

```
Lift-3 Operation
Previous position: Floor 8
Request: Floor 6 to Floor 15
Detail operations:
    Go from Floor 8 to Floor 6
    Go from Floor 6 to Floor 15
    #movement for this request: 11
    #request: 15
    Total #movement: 191
Current position: Floor 15

Lift-1 Operation
Previous position: Floor 3
Request: Floor 1 to Floor 12
Detail operations:
    Go from Floor 3 to Floor 1
    Go from Floor 1 to Floor 12
    #movement for this request: 13
    #request: 20
    Total #movement: 258
Current position: Floor 12
```

```
Lift-2 Operation
Previous position: Floor 12
Request: Floor 12 to Floor 3
Detail operations:
    Go from Floor 12 to Floor 12
    Go from Floor 12 to Floor 3
    #movement for this request: 9
    #request: 15
    Total #movement: 229
Current position: Floor 3

Total number of requests: 50
Total number of movements: 678
```

INPUT: ./lift_sim_A 10 1 **OUTPUT FILE NAME:** sim_outA3

OUTPUT:

```
Lift-3 Operation
Previous position: Floor 17
Request: Floor 16 to Floor 20
Detail operations:
    Go from Floor 17 to Floor 16
    Go from Floor 16 to Floor 20
    #movement for this request: 5
    #request: 31
    Total #movement: 444
Current position: Floor 20

Lift-2 Operation
Previous position: Floor 11
Request: Floor 4 to Floor 8
Detail operations:
    Go from Floor 11 to Floor 4
    Go from Floor 4 to Floor 8
    #movement for this request: 11
    #request: 34
    Total #movement: 510
Current position: Floor 8
```

```
Lift-1 Operation
Previous position: Floor 11
Request: Floor 19 to Floor 13
Detail operations:
    Go from Floor 11 to Floor 19
    Go from Floor 19 to Floor 13
    #movement for this request: 14
    #request: 35
    Total #movement: 473
Current position: Floor 13

Total number of requests: 100
Total number of movements: 1427
```

INPUT: ./lift_sim_B 10 1 **OUTPUT FILE NAME:** sim_outB3

OUTPUT:

```
Lift-1 Operation
Previous position: Floor 10
Request: Floor 17 to Floor 11
Detail operations:
    Go from Floor 10 to Floor 17
    Go from Floor 17 to Floor 11
    #movement for this request: 13
    #request: 33
    Total #movement: 433
Current position: Floor 11

Lift-3 Operation
Previous position: Floor 17
Request: Floor 16 to Floor 20
Detail operations:
    Go from Floor 17 to Floor 16
    Go from Floor 16 to Floor 20
    #movement for this request: 5
    #request: 33
    Total #movement: 538
Current position: Floor 20
```

```
Lift-2 Operation
Previous position: Floor 11
Request: Floor 4 to Floor 8
Detail operations:
    Go from Floor 11 to Floor 4
    Go from Floor 4 to Floor 8
    #movement for this request: 11
    #request: 34
    Total #movement: 474
Current position: Floor 8

Total number of requests: 100
Total number of movements: 1445
```