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:	Maughan	Student ID:	19513869
Other name(s):	Alec Christopher		
Unit name:	Operating Systems	Unit ID:	COMP2006
Lecturer / unit coordinator:	Sie Teng Soh	Tutor:	Sie Teng Soh
Date of submission:	11/05/2020	Which assignment ?	Operating Systems 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: Alec M. Date of Signature: 11/05/2020

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

Report

Contents

Discussion on Mutual Exclusion and Sharing

I achieved my buffer using a Buffer.c and Buffer.h file that store and perform operations on the buffer. The buffer operates as a circular FIFO queue array. The header file type defines a struct containing a 2D array of size m by 2. Each entry is an array of source floor and destination floor. The struct also contains integers detailing the maximum size (defined by m), the head index, the tail index, the number of populated entries, and a somewhat unrelated boolean that specifies if the requester task has completed it's file reading and enqueuing. In the .c file are two constructors for heap allocation (buffer_init) and shared memory allocation and mapping (buffer_open) as well as two respective deconstructors for head deallocation (buffer_destroy) and shared memory deallocation (buffer_close). The queue mutators buffer_enqueue and buffer_dequeue perform circular queue functions, and there are accessors for checking if the queue is full or empty. The variable that specifies if the requester task is complete is controlled with buffer_setComplete and biffer_isComplete.

For Part A (Threads) I used POSIX Pthreads to create threads and shared memory in the global scope. I used pthread condition variables and pthread mutex variables to achieve thread synchronisation. My shared variables included:

- The 't' value (Time taken for lifts to move)
- A buffer pointer struct
- A mutex variable for accessing the buffer
- A file pointer to the sim_out file
- A mutex variable for accessing the sim_out file pointer
- A condition variable for the buffer being full
- A condition variable for the buffer being empty
- An int counter for the total number of moves and an int counter for the total number of requests (Both use the same mutex variable as sim_out file pointer).

In one lift operation, this is the basic course of events:

- Obtain mutex lock on buffer (Enter critical section for buffer)
- If the buffer is empty, wait on the empty condition variable
- Dequeue once from the buffer
- Unlock the mutex on buffer (Exit critical section for buffer)
- Signal the full condition variable
- Calculate values needed for logging
- Obtain mutex lock on sim_out
- Increment the total moves counter
- Write data into sim_out as per the assignment spec
- Unlock the mutex on sim_out
- Sleep for t seconds and move current floor to the source floor of the dequeued request
- Sleep for t seconds and move current floor to the destination floor of the dequeued request
- Obtain mutex lock on buffer (Enter critical section for buffer)
- Check if another lift operation is required, or if the requester is complete and the buffer is empty
- Release the mutex lock on buffer (Exit critical section for buffer)
- Repeat if needed

In one Lift-R operation, this is the basic course of events:

- Read one request from sim_input
- If values read are illegal, show an error and exit gracefully

- Obtain mutex lock on buffer (Enter critical section for buffer)
- If the buffer is full, wait on the full condition variable
- Enqueue once onto the buffer
- Unlock the mutex on buffer (Exit critical section for buffer)
- Signal empty condition variable
- Obtain mutex lock for sim_out (Enter critical section for sim_out)
- Increment the total requests counter
- Write data into sim_out as per the assignment spec
- Unlock mutex on sim_out
- Repeat if not at the end of sim_input
- If not repeating, set the buffer to 'complete', telling the lifts to exit once the buffer is empty (Use buffer mutex to do this operation)

For Part B (Processes) I used shm_open, ftruncate and mmap to create shared memory and fork to create processes. I stored a file descriptor and a pointer for each shared memory object used and #defined a string name for each. I used POSIX unnamed semaphores (With sem_init) to achieve process synchronisation. My shared variables included

- A buffer pointer struct
- A binary semaphore for accessing the buffer
- A binary semaphore for accessing sim_out
- A counting semaphore for the buffer being full
- A counting semaphore for the buffer being empty
- An int counter for the total number of moves and an int counter for the total number of requests (Both use the same semaphore as sim_out).

In one lift operation, this is the basic course of events:

- Wait on the empty semaphore (Wait while the buffer is empty)
- Wait on the buffer semaphore (Obtain lock, enter critical section for buffer)
- Dequeue once from the buffer
- Check for 'poisoning' (Explained later), if poisoned, re-enqueue the poisoned entry, post the buffer and empty semaphores and exit gracefully
- Post the buffer semaphore (Unlock buffer, exit critical section for buffer)
- Post the full semaphore
- Calculate values needed for logging
- Wait on the sim_out semaphore (Enter critical section for sim_out)
- Increment the total moves counter
- Open sim_out for appending
- Write data into sim_out as per the assignment spec
- Close sim_out
- Post the sim_out semaphore (Exit critical section for sim_out)
- Sleep for t seconds and move current floor to the source floor of the dequeued request
- Sleep for t seconds and move current floor to the destination floor of the dequeued request
- Wait on the buffer semaphore (Enter critical section for buffer)
- Check if another lift operation is required, or if the requester is complete and the buffer is empty
- Post the buffer semaphore (Exit critical section for buffer)
- Repeat if needed

In one Lift-R operation, this is the basic course of events:

- Read one request from sim_input
- If values read are illegal, show an error and exit gracefully
- Wait on the full semaphore (Wait while the buffer is full)

- Wait on the buffer semaphore (Enter critical section for buffer)
- Enqueue once onto the buffer
- Post the buffer semaphore (Exit critical section for buffer)
- Post the empty semaphore
- Wait on the sim_out semaphore (Enter critical section for sim_out)
- Increment the total requests counter
- Open sim_out for appending
- Write data into sim_out as per the assignment spec
- Close sim_out
- Post the sim_out semaphore (Exit critical section for sim_out)
- Repeat if not at the end of sim_input
- If not repeating, set the buffer to ‘complete’, telling the lifts to exit once the buffer is empty and also ‘poison’ the queue by enqueueing (-1,-1), telling the lifts to exit once they read this entry (Use buffer semaphore to do these operations).

Issues and Testing

To test, I created various sim_input files and ran them through parts A and B multiple times to test that the program dealt with illegal sim_input files (No. of lines <50 or >100) and illegal floor values (<1 or >20) and that it created expected results in it’s sim_out file. I also tested various values of m and t in the command line arguments. The sample outputs are in the provided file under the samples directory.

Part A works flawlessly as far as I could tell, although Part B is only about 90% there, it occasionally shows behaviour in sim_out suggesting that more entries are being enqueued than the buffer size should support. Apart from this occasional hiccup, both programs show no memory leaks in valgrind and perform their behaviour as detailed in the assignment specification.

Sample Files

See submission directory ‘samples’ and see README inside.

Source Code

README.txt

```
Lift Simulator Readme

--------------------------------------------------------------------------------

Curtin University - Operating Systems (COMP2006) 2020 S1
Assignment - Lift Simulator
Alec Maughan 19513869

--------------------------------------------------------------------------------

Simulates 3 consumer lifts serving floor requests created by a producer using a
bounded buffer. Implemented using both threads (Part A) and processes (Part B)

--------------------------------------------------------------------------------

Compile entire program with `make`

--------------------------------------------------------------------------------

Run part A with `./lift_sim_A m t`
Run part B with `./lift_sim_B m t`

m is the maximum size of buffer
t is the time taken for a lift to move in seconds

--------------------------------------------------------------------------------

Samples in /samples directory, see /samples/README.txt
```

makefile

```
# Lift Simulator Makefile -------------------------------------------------------

# Variables ---------------------------------------------------------------------

gcc   = gcc -Wall -ansi -Werror -pedantic
link  = -lrt -pthread
execA = lift_sim_A
execB = lift_sim_B
all   = Buffer.o ${execA} ${execB}

# All ---------------------------------------------------------------------------

all: ${all}

# Compilation -------------------------------------------------------------------

Buffer.o : Buffer.c Buffer.h
	${gcc} Buffer.c -c ${link}

${execA} : ${execA}.c ${execA}.h Buffer.o
	${gcc} Buffer.o ${execA}.c -o ${execA} ${link}

${execB} : ${execB}.c ${execB}.h Buffer.o
	${gcc} ${execB}.c Buffer.o -o ${execB} ${link}

# Clean -------------------------------------------------------------------------

clean :
	rm ${all}
```

Buffer.h

```
/* Control and store FIFO array buffer initialised with either heap or shared
 * memory allocation */

#ifndef BUFFER_H

    #define BUFFER_H

    #define BUFF_NAME  "/lift_buffer"       /* Shared name of buffer          */
    #define ARRAY_NAME "/lift_buffer_array" /* Shared name of array in buffer */

    /* Buffer struct */
    typedef struct
    {
        int** array;    /* 2D malloc Array, size [size][2]                    */
        int   size;     /* Size of 1st dimension of array                     */
        int   head;     /* Index of starting element                          */
        int   tail;     /* Index of last element                              */
        int   count;    /* Number of populated elements                       */
        int   complete; /* Boolean if requester is finished                   */
    } buffer;

    /* Function forward declarations */
    buffer* buffer_init       (int);
    buffer* buffer_open       (int, int*);
    int     buffer_isEmpty    (buffer*);
    int     buffer_isFull     (buffer*);
    int     buffer_isComplete (buffer*);
    int     buffer_enqueue    (buffer*, int,  int );
    int     buffer_dequeue    (buffer*, int*, int*);
    void    buffer_setComplete(buffer*);
    void    buffer_destroy    (buffer*);
    void    buffer_close      (buffer*, int*);

#endif
```

Buffer.c

```
/* Control and store FIFO array buffer initialised with either heap or shared
 * memory allocation */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <errno.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

#include "Buffer.h"

/* Create a new buffer using malloc allocation */
buffer* buffer_init(int m)
{
    int     i;    /* For loop index                                          */
    buffer* buff; /* The new buffer                                          */

    buff           = (buffer*)malloc(sizeof(buffer));
    buff->array    = (int**)  malloc(sizeof(int*) * m);
    buff->size     = m;
    buff->count    = 0;
    buff->head     = 0;
    buff->tail     = 0;
    buff->complete = 0;

    for(i = 0; i < buff->size; i++)
        buff->array[i] = (int*)malloc(2 * sizeof(int));

    return buff;
}

/* Create a new buffer using shared memory allocation */
buffer* buffer_open(int m, int* fd)
{
    int i;
    buffer* buff;
    char entry_str[32];

    /* Allocate the buffer */
    fd[0] = shm_open(BUFF_NAME, O_CREAT | O_RDWR, 0666);
    if(fd[0] == -1) perror("Open failed on buffer");
    if(ftruncate(fd[0], sizeof(buffer)) == -1) perror("Buff truncate");
    buff = (buffer*)mmap(0, sizeof(buffer), PROT_READ | PROT_WRITE,
        MAP_SHARED, fd[0], 0);
    if(buff == MAP_FAILED) perror("Map failed on buffer");

    /* Allocate the array */
    fd[1] = shm_open(ARRAY_NAME, O_CREAT | O_RDWR, 0666);
    if(fd[1] == -1) perror("Open failed on buffer array");
    ftruncate(fd[1], sizeof(int*) * m);
    buff->array = (int**)mmap(0, sizeof(int*) * m, PROT_READ | PROT_WRITE,
        MAP_SHARED, fd[1], 0);
    if(buff->array == MAP_FAILED) perror("Map failed on buffer array");
    for(i = 0; i < m; i++)
    {
        sprintf(entry_str, "%s_entry_%d", ARRAY_NAME, i);
        fd[i+2] = shm_open(entry_str, O_CREAT | O_RDWR, 0666);
        if(fd[i+2] == -1) perror("Open failed on buffer array i");
        ftruncate(fd[i+2], sizeof(int) * 2);
        buff->array[i] = (int*)mmap(0, sizeof(int) * 2, PROT_READ | PROT_WRITE,
            MAP_SHARED, fd[i+2], 0);
        if(buff->array[i] == MAP_FAILED) perror("Map failed on buffer array i");
    }

    /* Set other variables */
    buff->size     = m;
    buff->count    = 0;
    buff->head     = 0;
    buff->tail     = 0;
    buff->complete = 0;

    return buff;
}

/* Check if a buffer is empty */
int buffer_isEmpty(buffer* buff)
{
    return buff->count == 0;
}

/* Check if a buffer is full */
int buffer_isFull(buffer* buff)
{
    return buff->count == buff->size;
}
```

```
/* Check if a buffer has been marked as complete */
int buffer_isComplete(buffer* buff)
{
    return buff->complete;
}

/* Enqueue a src and dest num onto the end of the buffer */
int buffer_enqueue(buffer* buff, int srcNum, int destNum)
{
    /* First make sure buffer is not full */
    if(buffer_isFull(buff)) return 0;

    /* Push values */
    buff->array[buff->tail][0] = srcNum;
    buff->array[buff->tail][1] = destNum;

    buff->count++;

    /* Either increment tail index or loop around */
    if(buff->tail == (buff->size - 1)) buff->tail = 0; /* Tail needs to loop  */
    else                               buff->tail++;   /* Tail can increment  */

    return 1;
}

/* Dequeue a src and dest num from the front of the buffer */
int buffer_dequeue(buffer* buff, int* srcNum, int* destNum)
{
    /* First make sure buffer is not empty */
    if(buffer_isEmpty(buff)) return 0;

    /* Pull values */
    *srcNum  = buff->array[buff->head][0];
    *destNum = buff->array[buff->head][1];

    buff->count--;

    /* Either increment head index or loop around */
    if(buff->head == (buff->size - 1)) buff->head = 0; /* Head needs to loop  */
    else                               buff->head++;   /* Head can increment  */

    return 1;
}

/* Set a buffer as complete */
```

```
void buffer_setComplete(buffer* buff)
{
    buff->complete = 1;
}

/* Free all associated memory of a buffer */
void buffer_destroy(buffer* buff)
{
    int i;

    for(i = 0; i < buff->size; i++) free(buff->array[i]);
    free(buff->array);
    free(buff);
    buff = NULL;
}

/* Clean up all associated memory of a buffer */
void buffer_close(buffer* buff, int* fd)
{
    int i;
    char entry_str[32]; /* Name of each 2nd dimension array in buff->array */

    for(i = 0; i < buff->size; i++)
    {
        sprintf(entry_str, "%s_entry_%d", ARRAY_NAME, i);

        if(munmap(buff->array[i], sizeof(int*)) == -1)
            perror("Munmap error on buffer array element");

        if(close(fd[i+2]) == -1) perror("Close error on buffer array element");

        if(shm_unlink(entry_str) == -1) perror("Unlink error on array elem");
    }
    if(munmap(buff->array, sizeof(int**)) == -1)
        perror("Munmap error on buffer array");
    if(close(fd[1]) == -1)
        perror("Close error on buffer array");
    if(shm_unlink(ARRAY_NAME) == -1)
        perror("Unlink error on buffer array");

    if(munmap(buff, sizeof(buffer)) == -1)
        perror("Munmap error on buffer");
    if(close(fd[0]) == -1)
        perror("Close error on buffer");
    if(shm_unlink(BUFF_NAME) == -1)
        perror("Unlink error on buffer");
}
```

lift_sim_A.h

```
/* Simulates 3 consumer lifts serving floor requests created by a producer using
 * a bounded buffer. Implemented using threads with POSIX pthreads */

#ifndef LIFT_SIM_A_H

    #define LIFT_SIM_A_H

    /* Function forward declarations */
    void* lift    (void*);
    void* request (void*);

#endif
```

lift_sim_A.c

```
/* Simulates 3 consumer lifts serving floor requests created by a producer using
 * a bounded buffer. Implemented using threads with POSIX pthreads */

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

#include "Buffer.h"
#include "lift_sim_A.h"

#define LIFTS 3 /* Number of lifts */

buffer*         buff;              /* The buffer                             */
pthread_mutex_t buffMutex;         /* Mutex lock for accessing buffer        */
pthread_cond_t  buffFull;          /* Condition variable for full buffer     */
pthread_cond_t  buffEmpty;         /* Condition variable for empty buffer    */
FILE*           sim_out;           /* Shared file ptr to sim_out for logging  */
pthread_mutex_t logMutex;          /* Mutex lock for sim_out and globalTot's  */
int             t;                 /* Time taken to move lift, given in args  */
int             globalTotMoves;    /* Total number of moves done by lifts     */
int             globalTotRequests; /* Total number of requests handled        */

int main(int argc, char* argv[])
{
    pthread_t liftThreads[LIFTS]; /* Threads for lifts 1 to LIFTS            */
    pthread_t requestThread;      /* Thread for request                      */
    int       liftNums[LIFTS];    /* Numbers 1 to LIFTS for lift() parameter  */
    int       i;                  /* For loop index                          */
    int       m;                  /* Buffer size, given in args              */
    int       threadError;        /* Return value of pthread_create()        */
    FILE*     sim_in;             /* sim_input file to count lines of        */
    int       lineNo         = 0; /* Number of lines counted in sim_input     */

    /* Handle command line arguments */
    if(argc != 3)
    {
        perror("Error, Invalid number of command line arguments");
        pthread_exit(NULL);
    }
    m = atoi(argv[1]);
    t = atoi(argv[2]);
    if(m < 1 || t < 0)
    {
        perror("Error, Invalid coomand line arguments");
        pthread_exit(NULL);
    }

    /* Count lines in sim_input */
    sim_in = fopen("sim_input", "r");
    if(sim_in == NULL)
    {
        perror("Error, sim_input file could not be opened");
        pthread_exit(NULL);
    }
    else if(ferror(sim_in))
    {
        perror("Error in opening sim_input");
        fclose(sim_in);
        sim_in = NULL;
        pthread_exit(NULL);
    }
    while(!feof(sim_in)) if(fgetc(sim_in) == '\n') lineNo++;
    fclose(sim_in);
    if(lineNo < 50 || lineNo > 100)
    {
        perror("Error, sim_input must be between 50 and 100 lines");
        pthread_exit(NULL);
    }

    /* Create buffer */
    buff = buffer_init(m);

    /* Create mutex and condition variables */
    pthread_mutex_init(&logMutex,  NULL);
    pthread_mutex_init(&buffMutex, NULL);
    pthread_cond_init (&buffFull,  NULL);
    pthread_cond_init (&buffEmpty, NULL);

    /* Open sim_out logging file and check for errors */
    sim_out = fopen("sim_out", "w");
    if(sim_out == NULL)
    {
        perror("Error, sim_out file could not be opened");
        pthread_exit(NULL);
    }
    else if(ferror(sim_out))
    {
        perror("Error in opening sim_out");
        fclose(sim_out);
        sim_out = NULL;
        pthread_exit(NULL);
    }

    /* Request thread creation and error checking */
    threadError = pthread_create(
        &requestThread, /* pthread_t ptr to request thread                    */
        NULL,           /* attr, NULL means use default attributes            */
        request,        /* function ptr to start routine request()            */
        NULL            /* argument to give to lift(), NULL for none          */
    );
    if(threadError)
    {
        perror("Error, request thread could not be created");
        pthread_exit(NULL);
    }

    /* Lift thread creation and error checking */
    for(i = 0; i < LIFTS; i++)
    {
        liftNums[i] = i + 1;
            /* If we just passed &i to the lift, i might increment before the
            lift set's its lift number to *i, so we must allocate a separate
            int for each lift */

        threadError = pthread_create(
            &liftThreads[i],    /* pthread_t ptr to lift thread               */
            NULL,               /* attr, NULL means use default attributes    */
            lift,               /* function ptr to start routine lift()       */
            (void*)&liftNums[i] /* argument to give to lift() - lift number   */
        );
        if(threadError)
        {
            perror("Error, lift thread could not be created");
            pthread_exit(NULL);
        }
    }

    /* Join all threads to wait until they terminate before cleaning up */
    pthread_join(requestThread, NULL);
    for(i = 0; i < LIFTS; i++) pthread_join(liftThreads[i], NULL);

    /* Print final stats to sim_out */
    fprintf(sim_out, "Total number of requests: %d\n", globalTotRequests);
    fprintf(sim_out, "Total number of movements: %d\n", globalTotMoves);
```

```
    /* Close file and free heap memory */
    fclose               (sim_out);
    buffer_destroy       (buff);
    pthread_mutex_destroy(&logMutex);
    pthread_mutex_destroy(&buffMutex);
    pthread_cond_destroy (&buffEmpty);
    pthread_cond_destroy (&buffFull);

    pthread_exit(NULL);
}

/* Represents Lift-1 to Lift-N, consumer thread that dequeues requests from the
 * buffer                                                                     */
void* lift(void* liftNumPtr)
{
    int liftNum;          /* This lift's number, 1 to LIFTS                   */
    int flr          = 1; /* Current floor                                    */
    int srcFlr;           /* Source floor of current request                  */
    int destFlr;          /* Destination floor of current request             */
    int requestNum   = 0; /* Number of requests served                        */
    int move         = 0; /* Number of floors moved this request              */
    int totMoves     = 0; /* Total number of floors moved                     */
    int done         = 0; /* Boolean for the lift being finished              */

    /* Assign lift number from void* parameter */
    liftNum = *((int*)liftNumPtr);

    while(!done)
    {
        /* Obtain lock for buffer */
        pthread_mutex_lock(&buffMutex);

        /* Wait if the buffer is empty */
        if(buffer_isEmpty(buff)) pthread_cond_wait(&buffEmpty, &buffMutex);

        /* Dequeue once from buffer */
        buffer_dequeue(buff, &srcFlr, &destFlr);

        /* Unlock buffer mutex */
        pthread_mutex_unlock(&buffMutex);

        /* Stop request() from waiting */
        pthread_cond_signal(&buffFull);

        /* Calculate values for sim_out */
        move = abs(flr - srcFlr) + abs(srcFlr - destFlr);
        totMoves += move;
        requestNum++;

        /* Obtain lock for appending to sim_out */
        pthread_mutex_lock(&logMutex);

        /* Increment global total moves, uses logMutex */
        globalTotMoves += move;

        /* Write log to sim_out */
        fprintf(sim_out, "Lift-%d Operation\n",                liftNum        );
        fprintf(sim_out, "Previous position: Floor %d\n",      flr            );
        fprintf(sim_out, "Request: Floor %d to Floor %d\n",    srcFlr, destFlr);
        fprintf(sim_out, "Detail operations:\n"                               );
        fprintf(sim_out, "\tGo from Floor %d to Floor %d\n",   flr,    srcFlr );
        fprintf(sim_out, "\tGo from Floor %d to Floor %d\n",   srcFlr, destFlr);
        fprintf(sim_out, "\t#movement for this request: %d\n", move           );
        fprintf(sim_out, "\t#request: %d\n",                   requestNum     );
        fprintf(sim_out, "\tTotal #movement: %d\n",            totMoves       );
        fprintf(sim_out, "Current position: Floor %d\n\n",     destFlr        );

        /* Release lock on sim_out */
        pthread_mutex_unlock(&logMutex);

        /* Move from current floor to srcFloor */
        sleep(t);
        flr = srcFlr;

        /* Move from current floor to destFloor */
        sleep(t);
        flr = destFlr;

        /* Check if we need to loop again */
        pthread_mutex_lock(&buffMutex);
        if(buffer_isEmpty(buff) && buffer_isComplete(buff)) done = 1;
        pthread_mutex_unlock(&buffMutex);
    }

    pthread_exit(0);
}

/* Represents Lift-R, producer thread that enqueues requests onto the buffer  */
void* request(void* nullPtr)
{
    FILE* sim_in;  /* sim_input file ptr                                      */
    int   srcFlr;  /* Destination floor read from sim_in                      */
    int   destFlr; /* Source floor read from sim_in                           */

    /* Open sim_input */
    sim_in = fopen("sim_input", "r");
    if(sim_in == NULL)
    {
        perror("Error, sim_input file could not be opened");
        buffer_setComplete(buff);
        pthread_exit(0);
    }
    else if(ferror(sim_in))
    {
        perror("Error in opening sim_input");
        buffer_setComplete(buff);
        fclose(sim_in);
        sim_in = NULL;
        pthread_exit(0);
    }

    while(!feof(sim_in))
    {
        /* Read one line from sim_input */
        fscanf(sim_in, "%d %d", &srcFlr, &destFlr);

        /* Make sure read values are legal */
        if(srcFlr < 1 || srcFlr > 20 || destFlr < 1 || destFlr > 20)
        {
            perror("Illegal floor values in sim_input");
            buffer_setComplete(buff);
            fclose(sim_in);
            pthread_exit(0);
        }

        /* Obtain mutex lock on buffer */
        pthread_mutex_lock(&buffMutex);

        /* Wait if the buffer is full */
        if(buffer_isFull(buff)) pthread_cond_wait(&buffFull, &buffMutex);

        /* Enqueue once into the buffer */
        buffer_enqueue(buff, srcFlr, destFlr);

        /* Unlock buffer mutex */
        pthread_mutex_unlock(&buffMutex);

        /* Wake up lifts */
```

```
        pthread_cond_signal(&buffEmpty);

        /* Obtain mutex lock for sim_out */
        pthread_mutex_lock(&logMutex);

        /* Increment global total num of requests, uses logMutex */
        globalTotRequests++;

        /* Print to sim_out */
        fprintf(sim_out, "--------------------------------------------\n");
        fprintf(sim_out, "New Lift Request From Floor %d to Floor %d\n",
            srcFlr, destFlr);
        fprintf(sim_out, "Request No: %d\n", globalTotRequests);
        fprintf(sim_out, "--------------------------------------------\n\n");

        /* Unlock sim_out mutex */
        pthread_mutex_unlock(&logMutex);
    }

    /* Set lifts to close once the buffer is empty */

    pthread_mutex_lock(&buffMutex);
    buffer_setComplete(buff);
    pthread_mutex_unlock(&buffMutex);

    fclose(sim_in);

    pthread_exit(0);
}
```

lift_sim_B.h

```
/* Simulates 3 consumer lifts serving floor requests created by a producer using
 * a bounded buffer. Implemented using processes with POSIX shared memory */

#ifndef LIFT_SIM_B_H

    #define LIFT_SIM_B_H

    /* Function foward declarations */
    void lift   (int, int);
    void request();

#endif
```

lift_sim_B.c

```
/* Simulates 3 consumer lifts serving floor requests created by a producer using
 * a bounded buffer. Implemented using processes with POSIX shared memory */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <errno.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/types.h>

#include "Buffer.h"
#include "lift_sim_B.h"

#define LIFTS              3                    /* Number of lifts            */
#define SEM_BUFF_NAME     "/lift_sem_buff"     /* Shared name of buffMutex    */
#define SEM_FULL_NAME     "/lift_sem_full"     /* Shared name of buffFull     */
#define SEM_EMPTY_NAME    "/lift_sem_empty"    /* Shared name of buffEmpty    */
#define SEM_LOG_NAME      "/lift_sem_log"      /* Shared name of logMutex     */
#define TOT_MOVES_NAME    "/lift_tot_moves"    /* Shared Name of totMov       */
#define TOT_REQUESTS_NAME "/lift_tot_requests" /* Shared name of totRequests  */

int main(int argc, char* argv[])
{
    pid_t   liftIDs[LIFTS]; /* Process IDs for lifts 1 to LIFTS              */
    pid_t   requestID;      /* Process ID for request                        */
    int     i;              /* For loop index                                */
    int     m;              /* Buffer size, given in args                    */
    int     t;              /* Time taken to move lift, given in args        */
    pid_t   forkVal;        /* Return value of fork()                        */
    FILE*   sim_in;         /* sim_input file to count lines of              */
    FILE*   sim_out;        /* sim_out file ptr to write final stats to      */
    int     lineNo     = 0; /* Number of lines counted in sim_input          */
    buffer* buff;           /* The buffer shared memory obj ptr              */
    int*    fd_buffArr;     /* File descriptors for buffer shared memory     */
    int*    totMov;         /* Total moves done by lifts shared obj ptr      */
    int     fd_totMov;      /* File descriptor for totMov                    */
    int*    totReq;         /* Total requests served shared obj ptr          */
    int     fd_totReq;      /* File descriptor for totReq                    */
    sem_t*  buffMutex;      /* Binary semaphore for accessing buffer         */
```

```
int     fd_buffMutex;   /* File descriptor for buffMutex                  */
sem_t*  buffFull;       /* Counting semaphore for full, init to m         */
int     fd_buffFull;    /* File descriptor for buffFull                   */
sem_t*  buffEmpty;      /* Counting semaphore for empty, init to m        */
int     fd_buffEmpty;   /* File descriptor for buffEmpty                  */
sem_t*  logMutex;       /* Binary semaphore for accessing sim_out         */
int     fd_logMutex;    /* File descriptor for logMutex                   */

/* Handle command line arguments */
if(argc != 3)
{
    perror("Error: Invalid number of arguments");
    return 1;
}
m = atoi(argv[1]);
t = atoi(argv[2]);
if(m < 1 || t < 0)
{
    perror("Error, invalid command line arguments");
    return 1;
}

/* Count lines in sim_input */
sim_in = fopen("sim_input", "r");
if(sim_in == NULL)
{
    perror("Error, sim_input file could not be opened");
    return 1;
}
else if(ferror(sim_in))
{
    perror("Error in opening sim_input");
    fclose(sim_in);
    sim_in = NULL;
    return 1;
}
while(!feof(sim_in)) if(fgetc(sim_in) == '\n') lineNo++;
fclose(sim_in);
if(lineNo < 50 || lineNo > 100)
{
    perror("Error, sim_input must be between 50 and 100 lines");
    return 1;
}

/* Clear out sim_out since we are appending to it */
unlink("sim_out");
```

```
/* Initialise semaphores */
fd_buffMutex = shm_open(SEM_BUFF_NAME,  O_CREAT | O_RDWR, 0666);
fd_buffFull  = shm_open(SEM_FULL_NAME,  O_CREAT | O_RDWR, 0666);
fd_buffEmpty = shm_open(SEM_EMPTY_NAME, O_CREAT | O_RDWR, 0666);
fd_logMutex  = shm_open(SEM_LOG_NAME,   O_CREAT | O_RDWR, 0666);
if(fd_buffMutex == -1) perror("Open failed on buffMutex");
if(fd_buffFull  == -1) perror("Open failed on buffFull" );
if(fd_buffEmpty == -1) perror("Open failed on buffEmpty");
if(fd_logMutex  == -1) perror("Open failed on logMutex" );
ftruncate(fd_buffMutex, sizeof(sem_t));
ftruncate(fd_buffFull,  sizeof(sem_t));
ftruncate(fd_buffEmpty, sizeof(sem_t));
ftruncate(fd_logMutex,  sizeof(sem_t));
buffMutex = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_buffMutex, 0);
buffFull  = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_buffFull,  0);
buffEmpty = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_buffEmpty, 0);
logMutex  = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_logMutex,  0);
if(buffMutex == MAP_FAILED) perror("Map failed on buffMutex");
if(buffFull  == MAP_FAILED) perror("Map failed on buffFull" );
if(buffEmpty == MAP_FAILED) perror("Map failed on buffEmpty");
if(logMutex  == MAP_FAILED) perror("Map failed on logMutex" );
if(sem_init(buffMutex, 1, 1) == -1) perror("Sem init failed on buffMutex");
if(sem_init(buffFull,  1, 0) == -1) perror("Sem init failed on buffFull" );
if(sem_init(buffEmpty, 1, m) == -1) perror("Sem init failed on buffEmpty");
if(sem_init(logMutex,  1, 1) == -1) perror("Sem init failed on logMutex" );

/* Initialise buffer */
fd_buffArr = (int*)malloc((m + 2) * sizeof(int)); /* FDs of entire buffer */
buff       = buffer_open(m, fd_buffArr);          /* Stores FDs in array  */

/* Set up shared memory for total moves an requests */
fd_totMov = shm_open(TOT_MOVES_NAME,    O_CREAT | O_RDWR, 0666);
fd_totReq = shm_open(TOT_REQUESTS_NAME, O_CREAT | O_RDWR, 0666);
if(fd_totMov == -1) perror("Open failed on main moves counter"  );
if(fd_totReq == -1) perror("Open failed on main request counter");
ftruncate(fd_totMov, sizeof(int));
ftruncate(fd_totReq, sizeof(int));
totMov = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_totMov, 0);
totReq = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_totReq, 0);
```

```
if(totMov == MAP_FAILED) perror("Map failed on main move counter"   );
if(totReq == MAP_FAILED) perror("Map failed on main request counter");
*totMov = 0;
*totReq = 0;

/* Request process creation and error checking */
forkVal = fork();
if(forkVal == -1)
{
    /* Error */
    perror("Error, fork failed");
    return 1;
}
if(forkVal == 0)
{
    /* You are the child */
    request();
    free(fd_buffArr);
    return 0;
}
else
{
    /* You are the parent */
    requestID = forkVal;
}

/* Lift process creation and error checking */
for(i = 0; i < LIFTS; i++)
{
    forkVal = fork();
    if(forkVal == -1)
    {
        /* Error */
        perror("Error, fork failed");
        return 1;
    }
    if(forkVal == 0)
    {
        /* You are the child */
        lift(i + 1, t);
        free(fd_buffArr);
        return 0;
    }
    else
    {
        /* You are the parent */
        liftIDs[i] = forkVal;
    }
}

/* Wait until all processes terminate before cleaning up */
waitpid(requestID, NULL, 0);
for(i = 0; i < LIFTS; i++) waitpid(liftIDs[i], NULL, 0);

/* Print final stats to sim_out */
sim_out = fopen("sim_out", "a");
if(sim_out == NULL)
{
    perror("Error, sim_out file could not be opened");
    return 1;
}
else if(ferror(sim_out))
{
    perror("Error in opening sim_out");
    fclose(sim_out);
    sim_out = NULL;
    return 1;
}
fprintf(sim_out, "Total number of requests: %d\n",  *totReq  );
fprintf(sim_out, "Total number of movements: %d\n", *totMov);
fclose (sim_out);

/* Clean up */
sem_destroy(buffMutex);
sem_destroy(buffFull);
sem_destroy(buffEmpty);
sem_destroy(logMutex);

if(munmap(totMov,    sizeof(int))   == -1) perror("Munmap error totMov"   );
if(munmap(totReq,    sizeof(int))   == -1) perror("Munmap error totReq"   );
if(munmap(buffMutex, sizeof(sem_t)) == -1) perror("Munmap error buffMutex");
if(munmap(buffFull,  sizeof(sem_t)) == -1) perror("Munmap error buffFull" );
if(munmap(buffEmpty, sizeof(sem_t)) == -1) perror("Munmap error buffEmpty");
if(munmap(logMutex,  sizeof(sem_t)) == -1) perror("Munmap error logMutex" );

if(close(fd_totMov)    == -1) perror("Close error totMov"   );
if(close(fd_totReq)    == -1) perror("Close error totReq"   );
if(close(fd_buffMutex) == -1) perror("Close error buffMutex");
if(close(fd_buffFull)  == -1) perror("Close error buffFull" );
if(close(fd_buffEmpty) == -1) perror("Close error buffEmpty");
if(close(fd_logMutex)  == -1) perror("Close error logMutex" );
```

```
    if(shm_unlink(SEM_BUFF_NAME)      == -1) perror("Unlink error sem buff"     );
    if(shm_unlink(SEM_FULL_NAME)      == -1) perror("Unlink error sem full"     );
    if(shm_unlink(SEM_EMPTY_NAME)     == -1) perror("Unlink error sem empty"    );
    if(shm_unlink(SEM_LOG_NAME)       == -1) perror("Unlink error sem log"      );
    if(shm_unlink(TOT_MOVES_NAME)     == -1) perror("Unlink error tot moves"    );
    if(shm_unlink(TOT_REQUESTS_NAME) == -1) perror("Unlink error tot requests");

    buffer_close(buff, fd_buffArr);
    free        (fd_buffArr      );

    return 0;
}

/* Represents Lift-1 to Lift-N, consumer process that dequeues requests from the
 * buffer                                                                      */
void lift(int liftNum, int t)
{
    int     flr         = 1; /* Current floor                                  */
    int     srcFlr;          /* Source floor of current request                */
    int     destFlr;         /* Destination floor of current request           */
    int     requestNum  = 0; /* Number of requests served                      */
    int     move        = 0; /* Number of floors moved this request            */
    int     localTotMov = 0; /* Total number of floors moved                   */
    int     done        = 0; /* Boolean for the lift being finished            */
    int     fd_buff;         /* File descriptor for buffer shared obj          */
    buffer* buff;            /* Buffer shared obj                              */
    int     fd_totMov;       /* File descriptor for totMov shared obj          */
    int*    totMov;          /* totMov shared obj                              */
    sem_t*  buffMutex;       /* Binary semaphore for accessing buffer          */
    int     fd_buffMutex;    /* File descriptor for buffMutex                  */
    sem_t*  buffFull;        /* Counting semaphore for full, init to m         */
    int     fd_buffFull;     /* File descriptor for buffFull                   */
    sem_t*  buffEmpty;       /* Counting semaphore for empty, init to m        */
    int     fd_buffEmpty;    /* File descriptor for buffEmpty                  */
    sem_t*  logMutex;        /* Binary semaphore for accessing sim_out         */
    int     fd_logMutex;     /* File descriptor for logMutex                   */
    FILE*   sim_out;         /* File ptr to log output file to append to       */

    /* Set up shared buffer and total moves counter */
    fd_buff   = shm_open(BUFF_NAME,      O_RDWR, 0666);
    fd_totMov = shm_open(TOT_MOVES_NAME, O_RDWR, 0666);
    if(fd_buff   == -1) perror("Open failed on lift buffer"       );
    if(fd_totMov == -1) perror("Open failed on lift moves counter");
    buff   = (buffer*)mmap(0, sizeof(buffer), PROT_READ | PROT_WRITE,
        MAP_SHARED, fd_buff,   0);
    totMov = (int*)   mmap(0, sizeof(int),    PROT_READ | PROT_WRITE,
        MAP_SHARED, fd_totMov, 0);
    if(buff   == MAP_FAILED) perror("Map failed on lift buffer"       );
    if(totMov == MAP_FAILED) perror("Map failed on lift moves counter");

    /* Open semaphores */
    fd_buffMutex = shm_open(SEM_BUFF_NAME,  O_RDWR, 0666);
    fd_buffEmpty = shm_open(SEM_FULL_NAME,  O_RDWR, 0666);
    fd_buffFull  = shm_open(SEM_EMPTY_NAME, O_RDWR, 0666);
    fd_logMutex  = shm_open(SEM_LOG_NAME,   O_RDWR, 0666);
    if(fd_buffMutex == -1) perror("Open failed on buffMutex");
    if(fd_buffFull  == -1) perror("Open failed on buffFull" );
    if(fd_buffEmpty == -1) perror("Open failed on buffEmpty");
    if(fd_logMutex  == -1) perror("Open failed on logMutex" );
    buffMutex = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
        MAP_SHARED, fd_buffMutex, 0);
    buffFull  = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
        MAP_SHARED, fd_buffFull,  0);
    buffEmpty = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
        MAP_SHARED, fd_buffEmpty, 0);
    logMutex  = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
        MAP_SHARED, fd_logMutex,  0);
    if(buffMutex == MAP_FAILED) perror("Map failed on buffMutex");
    if(buffFull  == MAP_FAILED) perror("Map failed on buffFull" );
    if(buffEmpty == MAP_FAILED) perror("Map failed on buffEmpty");
    if(logMutex  == MAP_FAILED) perror("Map failed on logMutex" );

    while(!done)
    {
        /* Wait if the buffer is empty */
        sem_wait(buffEmpty);

        /* Obtain lock for buffer */
        sem_wait(buffMutex);

        /* Dequeue once from buffer */
        buffer_dequeue(buff, &srcFlr, &destFlr);

        /* Check for 'poisoning' */
        if(srcFlr == -1 && destFlr == -1)
        {
            buffer_enqueue(buff, -1, -1);
            sem_post(buffEmpty);
            sem_post(buffMutex);
            return;
        }
```

```
/* Unlock buffer mutex */
sem_post(buffMutex);

/* Stop request() from waiting */
sem_post(buffFull);

/* Calculate values for sim_out */
move = abs(flr - srcFlr) + abs(srcFlr - destFlr);
localTotMov += move;
requestNum++;

/* Obtain lock for appending to sim_out */
sem_wait(logMutex);

/* Increment shared total moves, uses logMutex */
(*totMov) += move;

/* Write log to sim_out */
sim_out = fopen("sim_out", "a");
if(sim_out == NULL)
{
    perror("Error, sim_out file could not be opened");
    return;
}
else if(ferror(sim_out))
{
    perror("Error in opening sim_out");
    fclose(sim_out);
    sim_out = NULL;
    return;
}
fprintf(sim_out, "Lift-%d Operation\n",                liftNum        );
fprintf(sim_out, "Previous position: Floor %d\n",      flr            );
fprintf(sim_out, "Request: Floor %d to Floor %d\n",    srcFlr, destFlr);
fprintf(sim_out, "Detail operations:\n"                               );
fprintf(sim_out, "\tGo from Floor %d to Floor %d\n",   flr,    srcFlr );
fprintf(sim_out, "\tGo from Floor %d to Floor %d\n",   srcFlr, destFlr);
fprintf(sim_out, "\t#movement for this request: %d\n", move           );
fprintf(sim_out, "\t#request: %d\n",                   requestNum     );
fprintf(sim_out, "\tTotal #movement: %d\n",            localTotMov    );
fprintf(sim_out, "Current position: Floor %d\n\n",     destFlr        );
fclose(sim_out);

/* Release lock on sim_out */
sem_post(logMutex);
```

```
        /* Move from current floor to srcFloor */
        sleep(t);
        flr = srcFlr;

        /* Move from current floor to destFloor */
        sleep(t);
        flr = destFlr;

        /* Check if we need to loop again */
        sem_wait(buffMutex);
        if(buffer_isEmpty(buff) && buffer_isComplete(buff)) done = 1;
        sem_post(buffMutex);
    }

    /* Clean up */
    munmap(buff,      sizeof(buffer));
    munmap(totMov,    sizeof(int)   );
    munmap(buffMutex, sizeof(sem_t) );
    munmap(buffFull,  sizeof(sem_t) );
    munmap(buffEmpty, sizeof(sem_t) );
    munmap(logMutex,  sizeof(sem_t) );

    close(fd_totMov   );
    close(fd_buff     );
    close(fd_buffMutex);
    close(fd_buffFull );
    close(fd_buffEmpty);
    close(fd_logMutex );
}

/* Represents Lift-R, producer process that enqueues requests onto the buffer */
void request()
{
    FILE*   sim_in;       /* sim_input file ptr                              */
    int     srcFlr;       /* Destination floor read from sim_in              */
    int     destFlr;      /* Source floor read from sim_in                   */
    int     fd_buff;      /* File descriptor for buffer shared obj           */
    buffer* buff;         /* Buffer shared obj                               */
    int     fd_totReq;    /* File descriptor for totMov shared obj           */
    int*    totReq;       /* totReq shared obj                               */
    sem_t*  buffMutex;    /* Binary semaphore for accessing buffer           */
    int     fd_buffMutex; /* File descriptor for buffMutex                   */
    sem_t*  buffFull;     /* Counting semaphore for full, init to m          */
    int     fd_buffFull;  /* File descriptor for buffFull                    */
    sem_t*  buffEmpty;    /* Counting semaphore for empty, init to m         */
    int     fd_buffEmpty; /* File descriptor for buffEmpty                   */
```

```
sem_t*  logMutex;     /* Binary semaphore for accessing sim_out           */
int     fd_logMutex;  /* File descriptor for logMutex                     */
FILE*   sim_out;      /* File ptr to log output file to append to         */

/* Set up shared buffer and  total request counter */
fd_buff   = shm_open(BUFF_NAME,         O_RDWR, 0666);
fd_totReq = shm_open(TOT_REQUESTS_NAME, O_RDWR, 0666);
if(fd_buff   == -1) perror("Open failed on request buffer" );
if(fd_totReq == -1) perror("Open failed on request counter");
buff   = (buffer*)mmap(0, sizeof(buffer), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_buff,   0);
totReq = (int*)   mmap(0, sizeof(int),    PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_totReq, 0);
if(buff   == MAP_FAILED) perror("Map failed on requester buffer"         );
if(totReq == MAP_FAILED) perror("Map failed on requester's total counter");

/* Open semaphores */
fd_buffMutex = shm_open(SEM_BUFF_NAME,  O_RDWR, 0666);
fd_buffEmpty = shm_open(SEM_FULL_NAME,  O_RDWR, 0666);
fd_buffFull  = shm_open(SEM_EMPTY_NAME, O_RDWR, 0666);
fd_logMutex  = shm_open(SEM_LOG_NAME,   O_RDWR, 0666);
if(fd_buffMutex == -1) perror("Open failed on buffMutex");
if(fd_buffFull  == -1) perror("Open failed on buffFull" );
if(fd_buffEmpty == -1) perror("Open failed on buffEmpty");
if(fd_logMutex  == -1) perror("Open failed on logMutex" );
buffMutex = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_buffMutex, 0);
buffFull  = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_buffFull,  0);
buffEmpty = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_buffEmpty, 0);
logMutex  = (sem_t*)mmap(0, sizeof(sem_t), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_logMutex,  0);
if(buffMutex == MAP_FAILED) perror("Map failed on buffMutex");
if(buffFull  == MAP_FAILED) perror("Map failed on buffFull" );
if(buffEmpty == MAP_FAILED) perror("Map failed on buffEmpty");
if(logMutex  == MAP_FAILED) perror("Map failed on logMutex" );

/* Open sim_input */
sim_in = fopen("sim_input", "r");
if(sim_in == NULL)
{
    perror("Error, sim_input file could not be opened");
    buffer_setComplete(buff);
    return;
}
```

```
else if(ferror(sim_in))
{
    perror("Error in opening sim_input");
    buffer_setComplete(buff);
    fclose(sim_in);
    sim_in = NULL;
    return;
}

while(!feof(sim_in))
{
    /* Read one line from sim_input */
    fscanf(sim_in, "%d %d", &srcFlr, &destFlr);

    /* Make sure read values are legal */
    if(srcFlr < 1 || srcFlr > 20 || destFlr < 1 || destFlr > 20)
    {
        perror("Illegal floor values in sim_input");
        buffer_setComplete(buff);
        fclose(sim_in);
        return;
    }

    /* Wait if the buffer is full */
    sem_wait(buffFull);

    /* Obtain mutex lock on buffer */
    sem_wait(buffMutex);

    /* Enqueue once into the buffer */
    if(!buffer_enqueue(buff, srcFlr, destFlr))
    {
        perror("Buffer enqueue failed");
        sem_post(buffMutex);
        return;
    }

    /* Unlock buffer mutex */
    sem_post(buffMutex);

    /* Wake up a lift */
    sem_post(buffEmpty);

    /* Obtain mutex lock for sim_out */
    sem_wait(logMutex);
```

```
        /* Increment shared total num of requests, uses logMutex */
        (*totReq)++;

        /* Print to sim_out */
        sim_out = fopen("sim_out", "a");
        if(sim_out == NULL)
        {
            perror("Error, sim_out file could not be opened");
            return;
        }
        else if(ferror(sim_out))
        {
            perror("Error in opening sim_out");
            fclose(sim_out);
            sim_out = NULL;
            return;
        }
        fprintf(sim_out, "--------------------------------------------\n");
        fprintf(sim_out, "New Lift Request From Floor %d to Floor %d\n",
            srcFlr, destFlr);
        fprintf(sim_out, "Request No: %d\n", *totReq);
        fprintf(sim_out, "--------------------------------------------\n\n");
        fclose(sim_out);

        /* Unlock sim_out mutex */
        sem_post(logMutex);
    }

    /* Set lifts to close once the buffer is empty */
    sem_wait(buffMutex);
    buffer_setComplete(buff);
    buffer_enqueue(buff, -1, -1);
    sem_post(buffEmpty);
    sem_post(buffMutex);

    /* Clean up */
    fclose(sim_in);

    munmap(buff,      sizeof(buffer));
    munmap(totReq,    sizeof(int)   );
    munmap(buffMutex, sizeof(sem_t) );
    munmap(buffFull,  sizeof(sem_t) );
    munmap(buffEmpty, sizeof(sem_t) );
    munmap(logMutex,  sizeof(sem_t) );

    close(fd_buff     );
    close(fd_totReq   );
    close(fd_buffMutex);
    close(fd_buffFull );
    close(fd_buffEmpty);
    close(fd_logMutex );
}
```