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:	Singh	Student ID:	18944183
Other name(s):	Anurag		
Unit name:	Operating Systems	Unit ID:	COMP2006
Lecturer / unit coordinator:	Soh	Tutor:	Arlen
Date of submission:	17/05/2020	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: Anurag Singh Date of signature: 17/05/2020

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

Operating Systems

Lift Simulator - Report

Anurag Singh
18944183

May 17, 2020

Contents

1 Mutual Exclusion and Shared Resources **1**

1.1 Threads . . . 1

1.2 Processes . . . 2

2 Testing **3**

2.1 Processes . . . 3

2.1.1 Example 1 . . . 3

2.1.2 Example 2 . . . 3

2.1.3 Example 3 . . . 3

2.2 Threads . . . 4

2.2.1 Example 1 . . . 4

2.2.2 Example 2 . . . 4

2.2.3 Example 3 . . . 4

A Source Code **5**

A.1 README.md . . . 5

A.2 Common . . . 6

A.2.1 common.h . . . 6

A.2.2 file_io . . . 6

A.2.3 request.h . . . 8

A.3 Processes . . . 9

A.3.1 cqueue . . . 9

A.3.2 lift_main . . . 10

A.3.3 memory . . . 13

A.3.4 scheduler_main . . . 16

A.3.5 lift_sim_B . . . 18

A.4 Threads . . . 21

A.4.1 lift . . . 21

A.4.2 log . . . 24

A.4.3 queue . . . 26

A.4.4 scheduler . . . 30

A.4.5 lift_sim_A . . . 33

1 Mutual Exclusion and Shared Resources

1.1 Threads

Mutual exclusion is done using pthread mutexes the queue struct contains a mutex that is used to protect the data inside the queue, this mutex must be acquired before manipulating queue data to ensure mutual exclusion, this queue is shared between the scheduler thread and the 3 lift threads, and if any of them wish to mutate the queue in any way (removing or adding) they must first acquire mutex.

2 conditions are also used to allow some basic communication between threads, the conditions are when the queue is empty or full, if the queue is empty the lift threads wait until the scheduler adds something to the queue, and signals the condition waking up a sleeping thread to reacquire the mutex and to process the data, and in the case of the queue been full.

```
...
pthread_mutex_lock(&l->queue->mutex);
...

while(l->queue->empty && !l->queue->finished) {
    pthread_cond_wait(&l->queue->cond_empty, &l->queue->mutex);
}

if(!l->queue->empty) {
    request_t *r = queue_remove(l->queue);
    pthread_cond_signal(&l->queue->cond_full);
    pthread_mutex_unlock(&l->queue->mutex);
...
```

The scheduler waits on a full condition which is signaled by any of the 3 lift threads when they remove something from the queue, letting the scheduler know that there is space now and it can wake up an reacquire the mutex to add more into the queue.

```
...
pthread_mutex_lock(&s->queue->mutex);

while(s->queue->full) {
    pthread_cond_wait(&s->queue->cond_full, &s->queue->mutex);
}

request_t *r = file_read_line(s->input);
if(r) {
    queue_add(s->queue, r);
...
pthread_mutex_unlock(&s->queue->mutex);
...
```

1.2 Processes

A logic similar to threads exists in processes but requires the use of shared memory and semaphores since processes do not share memory mappings the way threads do.

I used POSIX shared memory and semaphores in order to implement shared resources and mutual exclusion. The bounded buffer problem can be solved through the use of 3 semaphores, one acting as a signal for full, another as a signal for empty, and the final used as a mutex to protect the shared memory (I also had another to protected accesses to the output file). The semaphores are initialised to different values.

```
1 sem_init(&ptr->semaphore.empty, 1, m);
2 sem_init(&ptr->semaphore.full, 1, 0);
3 sem_init(&ptr->semaphore.mutex, 1, 1);
4 sem_init(&ptr->semaphore.file, 1, 1);
```

The empty semaphore marks how many empty locations are in the buffer, the full semaphore marks how many are full, the mutex is used as a binary semaphore to protect the shared memory and same for the file semaphore to protect the file IO.

The scheduler must wait on the empty mutex to be available in order to add something into the shared memory, and once it has added something the scheduler posts the full mutex to allow a lift process to wake up and take one from the queue, once the lift is done it posts another empty, which will allow the scheduler back on if the queue was completely full prior.

The mutex semaphore is used to protect the shared memory by enforcing mutual exclusion by only allowing a single process to modify the data in the shared memory.

In terms of the shared resources, there was an array of requests which was used as a circular queue, and the semaphores were also shared, along with some statistics that need to be printed such as total requests and the movement counts of each individual lift.

These resources were shared through the use of the POSIX shared memory API, this API required you to open a shared memory object with a name, use `ftruncate` to set the size of the shared memory object, and to use `mmap` to map the memory object into the address space of the current process. This was done once to create the shared memory by the main, and then each process mapped the same block of shared memory into their own address space.

Once the processes have been `wait`ed on, the shared memory is unmapped and the memory objects are unlinked.

The output `FILE` isn't included in the shared memory since according to the manual page for `fork`, the open file descriptors are shared across child processes, therefore the parent simply opens it, and each child will close it when they're done, and same for the parent.

2 Testing

2.1 Processes

2.1.1 Example 1

Testing with a buffer size larger than the number of threads, and relatively small sleep time to see if the program behaves as expected. Through manual inspection of the output, it can be surmised that the program works as expected.

Command `./lift_sim_B 4 2 test/inputs/test2.txt`

Input `test/inputs/test2.txt`

Output `examples/process/test2.out`

2.1.2 Example 2

Testing with a significantly larger buffer size but with a sleep time of 0 to see if mutual exclusion still exists in the case of the "work" being done quickly. Manual inspection reveals the expected output, and since there was a final output, it means there was no deadlock while acquiring resources.

Command `./lift_sim_B 10 0 test/inputs/test7.txt`

Input `test/inputs/test7.txt`

Output `examples/process/test7.out`

2.1.3 Example 3

A very large buffer size with a long sleep time to demonstrate another possible situation where the program works as expected.

Command `./lift_sim_B 20 5 test/inputs/test21.txt`

Input `test/inputs/test21.txt`

Output `examples/process/test21.out`

2.2 Threads

Each test file is a different size between the allowed values of 50 and 100.

2.2.1 Example 1

A buffer size equal to to the number of worker threads with a relatively short "working" time.

Command `./lift_sim_A 3 1 test/inputs/test34.txt`

Input `test/inputs/test34.txt`

Output `examples/threads/test34.out`

2.2.2 Example 2

Larger buffer with the same time.

Command `./lift_sim_A 5 1 test/inputs/test1.txt`

Input `test/inputs/test1.txt`

Output `examples/threads/test1.out`

2.2.3 Example 3

Even larger buffer with a zero time to ensure mutual exclusion happens even without the threads sleeping.

Command `./lift_sim_A 10 0 test/inputs/test10.txt`

Input `test/inputs/test10.txt`

Output `examples/threads/test10.out`

A Source Code

A.1 README.md

```
# Lift Simulator

### Anurag Singh

## Introduction

Lift Simulator is a solution to the Operating Systems (COMP2006) assignment in Semester 1 2020.

## Purpose

To demonstrate multi-threading, and inter process/thread communication, and solving the critical section problem.

## Building

Running `make` will build 2 executables, `lift_sim_A` and `lift_sim_B` which refer to threads and processes implementations respectively.

## Usage

Both lift_sim_A, and lift_sim_B are used the same way, the program will write to the sim_out file as it runs.

```
$ ./lift_sim_{A,B} [max buffer size] [time per sleep] [input file]
```

Testing

Various test files are included in the `test/inputs` directory, with some sample outputs available in the `examples` directory.

Python scripts in the testing directory are for testing the program through the input files quickly.
```

A.2 Common

A.2.1 common.h

```
#ifndef COMMON_H
#define COMMON_H
/**
 * Some common definitions between
 * threads and proceses implementation
 */
#define OUTPUT_FILENAME         "sim_out"
#define TOTAL_THREADS           4
#define TOTAL_LIFTS             3
#define MAX_FLOOR               20
#define MIN_FLOOR               1
#define MIN_INPUT               50
#define MAX_INPUT               100

#endif
```

A.2.2 file_io

```
#ifndef FILE_IO_H
#define FILE_IO_H
#include <stdio.h>
#include <stdbool.h>

#include "request.h"

request_t* file_read_line(FILE *file);
bool file_validate(FILE *file);
#endif
```

```
/**
 * @file file_io.c
 * @author Anurag Singh (18944183)
 *
 * @date 24-04-20
 *
 */

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

#include "common.h"
```

```c
#include "file_io.h"
#include "request.h"

/**
 * This function reads a line
 * from the input file, parses it
 * in the format specified, and
 * does a simple sanity check to
 * ensure that its valid, before
 * malloc'ing a struct and returning
 * it containing the values read.
 *
 * @param file to read the line from
 * @return malloc'd request_t* containing the values
 */
request_t* file_read_line(FILE *file)
{
    int src, dest;
    char buffer[BUFSIZ] = {0};

    if(!fgets(buffer, BUFSIZ, file)) {
        return NULL;
    }

    int ret = sscanf(buffer, "%d %d", &src, &dest);
    if(ret != 2) {
        // incase 2 values weren't read from the line
        return NULL;
    }

    // bounds check the input
    if(src > MAX_FLOOR || dest > MAX_FLOOR || src < MIN_FLOOR || dest <
    ↪  MIN_FLOOR) {
        return NULL;
    }

    request_t *req = malloc(sizeof(request_t));
    req->src = src;
    req->dest = dest;

    return req;
}

/**
 * Validates that the input file has
 * only valid lift locations and
 * has a valid size between the min and
```

```
 * max inclusive
 *
 * @param file to be validated
 * @return true if the file is valid
 * @return false if the file is invalid
 */
bool file_validate(FILE *file)
{
    request_t *req;
    int count = 0;

    while((req = file_read_line(file))) {
        count++;
        free(req);
    }

    // If we didn't reach eof
    // it means there was an invalid line
    if(!feof(file)) {
        return false;
    }

    // If the overall count is less than or more than
    // its also invalid
    if(count > MAX_INPUT || count < MIN_INPUT) {
        return false;
    }

    rewind(file);

    return true;
}
```

A.2.3 request.h

```
#ifndef REQUEST_H
#define REQUEST_H

/**
 * Each request is stored as a struct
 */
typedef struct request {
    int src;
    int dest;
} request_t;

```

```
#endif
```

A.3 Processes

A.3.1 cqueue

```
#ifndef CQUEUE_H
#define CQUEUE_H
#include "memory.h"

void sm_cqueue_add(struct shared_memory *sm, request_t r);
request_t sm_cqueue_remove(struct shared_memory *sm);
#endif
```

```
/**
 * @file cqueue.c
 * @author Anurag Singh (18944183)
 *
 * @date 24-04-20
 *
 * A simple circular queue "implementation"
 *
 */

#include <common/request.h>

#include "memory.h"

/**
 * Add a request to the queue stored in
 * the shared memory param
 *
 * @param sm shared memory struct
 * @param r request to add to queue
 */
void sm_cqueue_add(struct shared_memory *sm, request_t r)
{
    if(sm->head == -1) {
        sm->head = 0;
        sm->tail = 0;
    } else {
        if(sm->tail == sm->max - 1) {
            sm->tail = 0;
        } else {
            sm->tail++;
```

```
        }
    }

    sm->current++;
    if(sm->current == sm->max) {
        sm->full = true;
    }
    sm->empty = false;
    sm->requests[sm->tail] = r;
}

/**
 * Remove a request from the queue
 *
 * @param sm memory to remove from
 * @return request_t
 */
request_t sm_cqueue_remove(struct shared_memory *sm)
{
    request_t r = sm->requests[sm->head];

    if(sm->head == sm->tail) {
        sm->head = -1;
        sm->tail = -1;
    } else {
        if(sm->head == sm->max - 1) {
            sm->head = 0;
        } else {
            sm->head++;
        }
    }

    sm->current--;
    if(sm->current == 0) {
        sm->empty = true;
    }
    sm->full = false;

    return r;
}
```

A.3.2 lift_main

```
#ifndef LIFT_MAIN_H
#define LIFT_MAIN_H
#include <stdio.h>
```

```c
int lift_main(int lift_num, int sleep_time, FILE *output);

#endif
```

```c
/**
 * @file lift_main.c
 * @author Anurag Singh (18944183)
 *
 * @date 24-04-20
 *
 * The function called when a lift process needs
 * to be created. Effectively the "main" of the lift.
 *
 */

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

#include "memory.h"
#include "cqueue.h"
#include "lift_main.h"

int lift_main(int lift_num, int sleep_time, FILE *output)
{
    int previous_floor = 1;
    int current_floor = 1;
    int request_num = 0;
    // Get access to the shared memory
    int shm_fd = shm_open(SHARED_MEMORY_NAME, O_RDWR, 0666);
    int req_fd = shm_open(SHARED_MEMORY_REQUESTS, O_RDWR, 0666);
    struct shared_memory *sm = mmap(0, sizeof(struct shared_memory),
    ↪  PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    sm->requests = mmap(0, sizeof(request_t) * sm->max, PROT_READ |
    ↪  PROT_WRITE, MAP_SHARED, req_fd, 0);

    while(1) {
        sem_wait(&sm->semaphore.mutex);
        if(sm->empty && sm->finished) {
            sem_post(&sm->semaphore.full);
            sem_post(&sm->semaphore.mutex);
            break;
```

```
42          }
43          sem_post(&sm->semaphore.mutex);
44          // we release the mutex after checking
45          // if the process should die, because
46          // to solve the bounded buffer problem
47          // we need to wait on the full semaphore
48          // which will be posted by the producer
49          // waiting while keeping holding the
50          // "mutex" will prevent progress, and likely deadlock
51
52
53          sem_wait(&sm->semaphore.full);
54          sem_wait(&sm->semaphore.mutex);
55
56          if(!sm->empty) {
57              request_t r = sm_cqueue_remove(sm);
58              sem_wait(&sm->semaphore.file);
59              request_num++;
60              previous_floor = current_floor;
61              current_floor = r.dest;
62              int r_movement = abs(previous_floor - r.src) + abs(r.src -
                ↪  current_floor);
63              sm->lift_movements[lift_num - 1] += r_movement;
64
65              fprintf(output, "Lift-%d Operation\n"
66                              "Previous Floor: %d\n"
67                              "Request: Floor %d to %d\n"
68                              "Details: \n"
69                              "\tGo from Floor %d to %d\n"
70                              "\tGo from Floor %d to %d\n"
71                              "\t# Movements: %d\n"
72                              "\tRequest No: %d\n"
73                              "\tTotal # Movement: %d\n"
74                              "Current Position: Floor %d\n\n",
75                              lift_num, previous_floor, r.src, r.dest,
76                              previous_floor, r.src, r.src, current_floor,
77                              r_movement, request_num,
                                ↪  sm->lift_movements[lift_num - 1],
78                              current_floor);
79              fflush(output);
80
81              sem_post(&sm->semaphore.file);
82              sem_post(&sm->semaphore.mutex);
83              sem_post(&sm->semaphore.empty);
84
85              // release the mutex since the critical section
86              // doesn't involve the work we do (in this case)
```

```
87              // pretend to work
88              sleep(sleep_time);
89          } else {
90              sem_post(&sm->semaphore.mutex);
91              sem_post(&sm->semaphore.empty);
92          }
93      }
94
95
96      fclose(output);
97      return 0;
98  }
```

A.3.3 memory

```
1   #ifndef MEMORY_H
2   #define MEMORY_H
3   #include <stdbool.h>
4   #include <semaphore.h>
5
6   #include <common/request.h>
7   #include <common/common.h>
8
9   #define SHARED_MEMORY_NAME          "shared_mem"
10  #define SHARED_MEMORY_REQUESTS      "requests_list"
11
12  /**
13   * kinda like the stack of
14   * memory that'll be shared
15   * across processes, also contains
16   * the shared semaphores in a separate struct
17   */
18  struct shared_memory {
19      request_t *requests;
20      int max;
21      int current;
22      int head;
23      int tail;
24
25      int total_requests;
26      int lift_movements[TOTAL_LIFTS];
27
28      bool finished;
29      bool empty;
30      bool full;
31
```

```
    struct {
        sem_t full;
        sem_t empty;
        sem_t mutex;
        sem_t file;
    } semaphore;
};

struct shared_memory* shared_mem_create(int m);
void shared_memory_destroy(struct shared_memory *sm, int m);

#endif
```

```
/**
 * @file memory.c
 * @author Anurag Singh (18944183)
 *
 * @date 24-04-20
 *
 * Handles allocating and freeing shared memory
 *
 */

// ftruncate
#define _POSIX_C_SOURCE 200112L
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <common/request.h>
#include <common/common.h>

#include "memory.h"

/**
 * Create the shared memory
 * initialising all fields to sane defaults
 * Uses the POSIX shared memory API
 *
 * @param m buffer size
 * @return struct shared_memory*
 */
struct shared_memory* shared_mem_create(int m)
{
```

```
35      int shm_fd = shm_open(SHARED_MEMORY_NAME, O_CREAT | O_RDWR, 0666);
36      ftruncate(shm_fd, sizeof(struct shared_memory));
37      struct shared_memory *ptr = mmap(0, sizeof(struct shared_memory),
        ↪  PROT_WRITE, MAP_SHARED, shm_fd, 0);
38
39      int req_list_fd = shm_open(SHARED_MEMORY_REQUESTS, O_CREAT | O_RDWR,
        ↪  0666);
40      ftruncate(req_list_fd, sizeof(request_t) * m);
41      ptr->requests = mmap(0, sizeof(request_t) * m, PROT_WRITE, MAP_SHARED,
        ↪  req_list_fd, 0);
42
43      // i think the memory is zero'd out
44      // but just incase, we need sane defaults
45      ptr->current = 0;
46      ptr->max = m;
47      ptr->finished = false;
48      ptr->empty = true;
49      ptr->full = false;
50      ptr->head = -1;
51      ptr->tail = -1;
52      ptr->total_requests = 0;
53      for(int i = 0; i < TOTAL_LIFTS; i++) {
54          ptr->lift_movements[i] = 0;
55      }
56
57      sem_init(&ptr->semaphore.empty, 1, m);
58      sem_init(&ptr->semaphore.full, 1, 0);
59      sem_init(&ptr->semaphore.mutex, 1, 1);
60      sem_init(&ptr->semaphore.file, 1, 1);
61
62      return ptr;
63  }
64
65  /**
66   * Unlinks and unmaps all of the shared memory
67   *
68   * @param sm shared memory block
69   * @param m buffer size (munmap needs this)
70   */
71  void shared_memory_destroy(struct shared_memory *sm, int m)
72  {
73      sem_destroy(&sm->semaphore.mutex);
74      sem_destroy(&sm->semaphore.empty);
75      sem_destroy(&sm->semaphore.full);
76      sem_destroy(&sm->semaphore.file);
77
78      munmap(sm->requests, sizeof(request_t) * m);
```

```
79      munmap(sm, sizeof(struct shared_memory));
80
81      shm_unlink(SHARED_MEMORY_REQUESTS);
82      shm_unlink(SHARED_MEMORY_NAME);
83  }
```

A.3.4 scheduler_main

```
1   #ifndef SCHEDULER_MAIN_H
2   #define SCHEDULER_MAIN_H
3   #include <stdio.h>
4   int scheduler_main(const char *filename, FILE *output);
5   #endif
```

```
1   /**
2    * @file scheduler_main.c
3    * @author Anurag Singh (18944183)
4    *
5    * @date 24-04-20
6    *
7    * the function called after fork to create a scheduler child
8    *
9    */
10
11  #include <stdio.h>
12  #include <stdlib.h>
13  #include <unistd.h>
14  #include <sys/types.h>
15  #include <sys/wait.h>
16  #include <sys/mman.h>
17  #include <sys/stat.h>
18  #include <fcntl.h>
19
20  #include <common/file_io.h>
21
22  #include "memory.h"
23  #include "cqueue.h"
24  #include "scheduler_main.h"
25
26  int scheduler_main(const char *filename, FILE *output)
27  {
28      bool finished = false;
29      int shm_fd = shm_open(SHARED_MEMORY_NAME, O_RDWR, 0666);
30      int req_fd = shm_open(SHARED_MEMORY_REQUESTS, O_RDWR, 0666);
31      struct shared_memory *sm = mmap(0, sizeof(struct shared_memory),
        ↪  PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

```
32      sm->requests = mmap(0, sizeof(request_t) * sm->max, PROT_READ |
        ↪  PROT_WRITE, MAP_SHARED, req_fd, 0);
33
34      // Make sure we can actually open the input file
35      // Else we mark as finished, and post the full mutex
36      // to allow each process to wake up and die
37      FILE *fp = fopen(filename, "r");
38      if(!fp) {
39          sem_wait(&sm->semaphore.mutex);
40          sm->finished = true;
41          sm->empty = true;
42          sem_post(&sm->semaphore.mutex);
43          sem_post(&sm->semaphore.full);
44          perror("input file");
45          fclose(output);
46          return EXIT_FAILURE;
47      }
48
49      while(!finished) {
50          request_t *r = file_read_line(fp);
51          if(r) {
52              sem_wait(&sm->semaphore.empty);
53              sem_wait(&sm->semaphore.mutex);
54              sm->total_requests++;
55              sm_cqueue_add(sm, *r);
56
57              sem_wait(&sm->semaphore.file);
58              fprintf(output, "------------------------------\n"
59                              "New Lift Request from %d to %d\n"
60                              "Request No: %d\n"
61                              "------------------------------\n\n", r->src,
                                ↪  r->dest, sm->total_requests);
62              fflush(output);
63              sem_post(&sm->semaphore.file);
64
65              sem_post(&sm->semaphore.mutex);
66              sem_post(&sm->semaphore.full);
67
68              free(r);
69          } else {
70              sm->finished = true;
71              finished = true;
72              // wake up anything sleeping before i die
73              sem_post(&sm->semaphore.full);
74          }
75
76      }
```

```
77
78      fclose(fp);
79      fclose(output);
80
81      return 0;
82  }
```

A.3.5 lift_sim_B

```
1   /**
2    * @file lift_sim_B.c
3    * @author Anurag Singh (18944183)
4    *
5    * @date 24-04-20
6    *
7    * The actual main of the process part
8    * Loads up shared memory, and setups up
9    * to fork and create child processes,
10   * while also waiting for them to
11   * die so we can clean up afterwards
12   *
13   */
14
15  #include <stdio.h>
16  #include <stdlib.h>
17  #include <unistd.h>
18  #include <string.h>
19  #include <stdbool.h>
20  #include <sys/types.h>
21  #include <sys/wait.h>
22
23  #include <common/request.h>
24  #include <common/file_io.h>
25  #include <common/common.h>
26
27  #include "memory.h"
28  #include "lift_main.h"
29  #include "scheduler_main.h"
30
31  int main(int argc, const char *argv[])
32  {
33      if(argc < 4) {
34          fprintf(stderr, "usage: %s [buffer size] [time] [filename]\n",
            ↪  argv[0]);
35          return EXIT_FAILURE;
36      }
```

```
37
38      int m = atoi(argv[1]);
39      int lift_time = atoi(argv[2]);
40
41      if(m <= 0 && lift_time < 0) {
42          fprintf(stderr, "buffer size > 0, time >= 0\n");
43          return EXIT_FAILURE;
44      }
45
46      FILE *output = fopen(OUTPUT_FILENAME, "w");
47      if(!output) {
48          perror("output file");
49          return EXIT_FAILURE;
50      }
51
52      FILE *input = fopen(argv[3], "r");
53      if(!input) {
54          perror("input file");
55          return EXIT_FAILURE;
56      }
57
58      bool valid = file_validate(input);
59      fclose(input);
60
61      if(!valid) {
62          fclose(output);
63
64          fprintf(stderr, "invalid input file: %s\n", argv[3]);
65          return EXIT_FAILURE;
66      }
67
68      // Shared Memory
69      struct shared_memory *sm = shared_mem_create(m);
70
71      pid_t lifts[TOTAL_LIFTS];
72      pid_t scheduler;
73      int total_children = 0;
74
75      // TODO: Add error checking to fork()
76      for(int i = 0; i < TOTAL_LIFTS; i++) {
77          if((lifts[i] = fork()) == 0) {
78              // Child
79              // Each child will immediately exit main by returning
80              total_children++;
81              return lift_main(i + 1, lift_time, output);
82          } else if(lifts[i] < 0) {
83              // Signal to end all processes that might have started
```

```
            sem_wait(&sm->semaphore.mutex);
            sm->empty = true;
            sm->finished = true;
            sem_post(&sm->semaphore.mutex);

            // Wait to prevent zombies
            for(int j = 0; j < total_children; j++) {
                waitpid(lifts[j], NULL, 0);
            }

            perror("fork");
            // Clean up
            fclose(output);
            shared_memory_destroy(sm, m);
            return EXIT_FAILURE;
        }
    }

    // Create scheduler
    if((scheduler = fork()) == 0) {
        // Child
        return scheduler_main(argv[3], output);
    } else if(scheduler < 0) {
        sem_wait(&sm->semaphore.mutex);
        sm->empty = true;
        sm->finished = true;
        sem_post(&sm->semaphore.mutex);
    }

    // Parent
    for(int i = 0; i < TOTAL_LIFTS; i++) {
        waitpid(lifts[i], NULL, 0);
    }

    // If the scheduler fork worked
    if(scheduler > 0) {
        waitpid(scheduler, NULL, 0);

        fprintf(output, "Total Number of Requests: %d\n"
                        "Total Number of Movements: %d\n",
                        sm->total_requests, sm->lift_movements[0]
                                          + sm->lift_movements[1]
                                          + sm->lift_movements[2]);

    }

    fclose(output);
```

```
131     shared_memory_destroy(sm, m);
132
133     return 0;
134 }
```

A.4 Threads

A.4.1 lift

```
1  #ifndef LIFT_H
2  #define LIFT_H
3
4  #include "queue.h"
5  #include "log.h"
6
7  struct lift {
8      int id;
9      int lift_time;
10     int total_movements;
11     struct queue *queue;
12     struct log *logger;
13 };
14
15 struct lift* lift_init(struct queue *queue, int lift_time, struct log
   ↪  *logger, int id);
16 void lift_free(struct lift *l);
17 void *lift(void *ptr);
18 #endif
```

```
1  /**
2   * @file lift.c
3   * @author Anurag Singh (18944183)
4   *
5   * @date 09-05-20
6   *
7   * Handles lift struct and the lift
8   * function itself
9   *
10  */
11
12 #include <stdio.h>
13 #include <pthread.h>
14 #include <unistd.h>
15 #include <stdlib.h>
16
17 #include <common/request.h>
```

```
18
19  #include "lift.h"
20  #include "queue.h"
21  #include "log.h"
22
23  /**
24   * Creates a lift struct with
25   * fields initialised to import params
26   *
27   * @param queue
28   * @param lift_time
29   * @param logger
30   * @param id
31   * @return struct lift*
32   */
33  struct lift* lift_init(struct queue *queue, int lift_time, struct log
     ↪  *logger, int id)
34  {
35      struct lift *l = malloc(sizeof(struct lift));
36      l->total_movements = 0;
37      l->logger = logger;
38      l->queue = queue;
39      l->lift_time = lift_time;
40      l->id = id;
41
42      return l;
43  }
44
45  /**
46   * Frees lift struct pointed to by l
47   *
48   * @param l lift struct to be free'd
49   */
50  void lift_free(struct lift *l)
51  {
52      free(l);
53  }
54
55  /**
56   * Lift function passed to pthread_create
57   * Handles all mutual exclusion and simulates
58   * work by sleeping
59   *
60   * @param ptr points to the lift struct
61   * @return void*
62   */
63  void* lift(void *ptr)
```

```
64  {
65      struct lift *l = (struct lift*)ptr;
66      // Each lift starts at the first floor
67      int current_floor = 1;
68      int previous_floor = 1;
69      int request_no = 0;
70
71      while(1) {
72          pthread_mutex_lock(&l->queue->mutex);
73
74          if(l->queue->empty && l->queue->finished) {
75              // let everyone know that the queue is empty
76              pthread_cond_broadcast(&l->queue->cond_empty);
77              pthread_mutex_unlock(&l->queue->mutex);
78              break;
79          }
80
81          while(l->queue->empty && !l->queue->finished) {
82              pthread_cond_wait(&l->queue->cond_empty, &l->queue->mutex);
83          }
84
85          if(!l->queue->empty) {
86              request_t *r = queue_remove(l->queue);
87              // need to let scheduler thread know that there's space now in
                ↪  the queue
88              pthread_cond_signal(&l->queue->cond_full);
89              pthread_mutex_unlock(&l->queue->mutex);
90
91              request_no++;
92              previous_floor = current_floor;
93              current_floor = r->dest;
94              int r_movement = abs(previous_floor - r->src) + abs(r->src -
                ↪  current_floor);
95              l->total_movements += r_movement;
96
97              log_printf(l->logger, "Lift-%d Operation\n"
98                                    "Previous Floor: %d\n"
99                                    "Request: Floor %d to %d\n"
100                                   "Details: \n"
101                                   "\tGo from Floor %d to %d\n"
102                                   "\tGo from Floor %d to %d\n"
103                                   "\t# Movements: %d\n"
104                                   "\tRequest No: %d\n"
105                                   "\tTotal # Movement: %d\n"
106                                   "Current Position: Floor %d\n\n",
107                                   l->id, previous_floor, r->src, r->dest,
```

```
                                    previous_floor, r->src, r->src,
                                    ↪  current_floor,
                                    r_movement, request_no, l->total_movements,
                                    current_floor);

            sleep(l->lift_time);
            free(r);
            // Simulate work by putting the thread to sleep
        } else {
            pthread_mutex_unlock(&l->queue->mutex);
        }
    }

    return NULL;
}
```

A.4.2 log

```
#ifndef LOG_H
#define LOG_H
#include <stdio.h>
#include <pthread.h>

struct log {
    pthread_mutex_t mutex;
    FILE *fp;
};

struct log* log_init(FILE *fp);
void log_free(struct log *l);
void log_printf(struct log *l, const char *s, ...);

#endif
```

```
/**
 * @file log.c
 * @author Anurag Singh (18944183)
 *
 * @date 09-05-20
 *
 * Handles log struct creation
 * and mutual exclusion for logging
 * to file
 *
 */

```

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#include "log.h"

/**
 * Creates a log struct with
 * file pointer fp, initialises
 * the mutex used to protect FILE io
 *
 * @param fp
 * @return struct log*
 */
struct log* log_init(FILE *fp)
{
    struct log *l = malloc(sizeof(struct log));

    l->fp = fp;
    pthread_mutex_init(&l->mutex, NULL);

    return l;
}

/**
 * Frees the log struct, and destroys the mutex
 *
 * @param l log struct to be freed
 */
void log_free(struct log *l)
{
    // File will be closed by whoever opened it
    // eg. the main thread in this case
    pthread_mutex_destroy(&l->mutex);
    free(l);
}

/**
 * printf like function that allows printing
 * to the FILE in the log struct, while utilising
 * the mutex to protect it
 *
 * @param l
 * @param s
 * @param ...
 */
void log_printf(struct log *l, const char *s, ...)
```

```
{
    va_list args;
    va_start(args, s);

    pthread_mutex_lock(&l->mutex);
    vfprintf(l->fp, s, args);
    pthread_mutex_unlock(&l->mutex);

    va_end(args);
}
```

A.4.3 queue

```
#ifndef QUEUE_H
#define QUEUE_H
#include <stdbool.h>
#include <pthread.h>

#include <common/request.h>

struct node {
    struct node *next;
    request_t *data;
};

/// Queue backed by a List
struct queue {
    pthread_mutex_t mutex;
    pthread_cond_t cond_empty;
    pthread_cond_t cond_full;

    struct node *head;
    struct node *tail;

    bool full;
    bool empty;

    bool finished;

    int max;
    int count;
};

struct queue* queue_init(int m);
void queue_add(struct queue *queue, request_t *node);
request_t* queue_remove(struct queue *queue);
```

```
34 void queue_free(struct queue *queue);
35
36 #endif
```

```
1  /**
2   * @file queue.c
3   * @author Anurag Singh (18944183)
4   *
5   * @date 09-05-20
6   *
7   * Queue implemenation, backed by a list
8   *
9   */
10
11 #include <stdlib.h>
12
13 #include <common/request.h>
14 #include "queue.h"
15
16 /**
17  * Initialises a queue struct with
18  * a maximum size of m
19  *
20  * @param m max size
21  * @return struct queue*
22  */
23 struct queue* queue_init(int m)
24 {
25     struct queue *queue = malloc(sizeof(struct queue));
26     queue->max = m;
27     queue->count = 0;
28     queue->head = NULL;
29     queue->tail = NULL;
30     queue->empty = true;
31     queue->full = false;
32     queue->finished = false;
33
34     pthread_mutex_init(&queue->mutex, NULL);
35     pthread_cond_init(&queue->cond_empty, NULL);
36     pthread_cond_init(&queue->cond_full, NULL);
37
38     return queue;
39 }
40
41 /**
42  * Creates a node containing the data
43  *
```

```
44  * @param data data to put in the node
45  * @return struct node*
46  */
47 static struct node* node_init(request_t *data)
48 {
49     struct node *node = malloc(sizeof(struct node));
50     node->next = NULL;
51     node->data = data;
52 
53     return node;
54 }
55 
56 /**
57  * Adds to the end of the list
58  * to allow FIFO behaviour
59  *
60  * @param queue queue to add to
61  * @param ptr data to add
62  */
63 void queue_add(struct queue *queue, request_t *ptr)
64 {
65     if(queue->count == queue->max) {
66         return;
67     }
68 
69     struct node *node = node_init(ptr);
70 
71     if(queue->tail == NULL) {
72         queue->head = node;
73         queue->tail = node;
74     } else {
75         queue->tail->next = node;
76         queue->tail = node;
77     }
78 
79     queue->count++;
80     if(queue->count == queue->max) {
81         queue->full = true;
82     }
83 
84     queue->empty = false;
85 }
86 
87 /**
88  * Removes from the front of the queue
89  *
90  * @param queue
```

```
 * @return request_t*
 */
request_t* queue_remove(struct queue *queue)
{
    struct node *node = queue->head;

    if(!node) {
        return NULL;
    }

    void *data = queue->head->data;

    if(queue->head && queue->head->next) {
        queue->head = queue->head->next;
    } else if(!queue->head->next) {
        queue->head = NULL;
        queue->tail = NULL;
        queue->empty = true;
    }

    queue->full = false;
    queue->count--;
    free(node);

    return data;
}

/**
 * Frees a node
 *
 * @param node
 */
static void node_free(struct node *node)
{
    free(node->data);
    free(node);
}

/**
 * Frees the entire queue including
 * the struct and any mutexes
 *
 * @param queue
 */
void queue_free(struct queue *queue)
{
    struct node *current = queue->head;
```

```
138
139     while(current) {
140         struct node *next = current->next;
141
142         node_free(current);
143
144         current = next;
145     }
146
147     pthread_mutex_destroy(&queue->mutex);
148     pthread_cond_destroy(&queue->cond_empty);
149     pthread_cond_destroy(&queue->cond_full);
150     free(queue);
151 }
```

A.4.4 scheduler

```
1  #ifndef SCHEDULER_H
2  #define SCHEDULER_H
3
4  #include <stdio.h>
5  #include <pthread.h>
6
7  #include "queue.h"
8  #include "log.h"
9
10 struct scheduler {
11     int total_requests;
12     FILE *input;
13     struct queue *queue;
14     struct log *logger;
15 };
16
17 struct scheduler* scheduler_init(FILE *input, struct queue *queue, struct log
   ↪  *logger);
18 void scheduler_free(struct scheduler *s);
19 void* scheduler(void *ptr);
20
21 #endif
```

```
1  /**
2   * @file scheduler.c
3   * @author Anurag Singh (18944183)
4   *
5   * @date 09-05-20
6   *
```

```
 * Handles all scheduler based actions
 * including allocating the structure
 * and the function passed to pthread_create
 *
 */

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

#include <common/file_io.h>
#include <common/request.h>

#include "scheduler.h"

/**
 * Create a scheduler struct, initialising struct
 * fields to imported variables
 *
 * @param input
 * @param queue
 * @param logger
 * @return struct scheduler*
 */
struct scheduler* scheduler_init(FILE *input, struct queue *queue, struct log
    *logger)
{
    struct scheduler *s = malloc(sizeof(struct scheduler));
    s->total_requests = 0;
    s->input = input;
    s->logger = logger;
    s->queue = queue;

    return s;
}

/**
 * Frees a scheduler struct
 *
 * @param s
 */
void scheduler_free(struct scheduler *s)
{
    free(s);
}

/**
```

```
 * Function passed to pthread_create
 * in order to create a scheduler struct
 * Contains all mutual exclusion required
 *
 * @param ptr scheduler struct
 * @return void*
 */
void* scheduler(void *ptr)
{
    struct scheduler *s = (struct scheduler *)ptr;
    bool finished = false;

    while(!finished) {
        pthread_mutex_lock(&s->queue->mutex);

        while(s->queue->full) {
            // if queue at max capacity, wait for someone to dequeue
            pthread_cond_wait(&s->queue->cond_full, &s->queue->mutex);
        }

        request_t *r = file_read_line(s->input);
        if(r) {
            queue_add(s->queue, r);
            s->total_requests++;
            log_printf(s->logger, "------------------------------\n"
                                  "New Lift Request from %d to %d\n"
                                  "Request No: %d\n"
                                  "------------------------------\n\n",
                                   ↪  r->src, r->dest, s->total_requests);
            pthread_cond_signal(&s->queue->cond_empty);
        } else {
            // let everyone know we got nothing to put in
            // for anyone waiting for something to be added
            pthread_cond_broadcast(&s->queue->cond_empty);
            s->queue->finished = true;
            finished = true;
        }

        pthread_mutex_unlock(&s->queue->mutex);
    }

    return NULL;
}
```

A.4.5 lift_sim_A

```c
/**
 * @file lift_sim_A.c
 * @author Anurag Singh (18944183)
 *
 * @date 09-05-20
 *
 * The main of the threads part, handles
 * logging, queue and creation and
 * destruction of threads.
 *
 */

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

#include <common/request.h>
#include <common/file_io.h>
#include <common/common.h>

#include "queue.h"
#include "scheduler.h"
#include "lift.h"
#include "log.h"

int main(int argc, const char *argv[])
{
    if(argc < 4) {
        fprintf(stderr, "usage: %s [buffer size] [time] [filename]\n",
        ↪  argv[0]);
        return EXIT_FAILURE;
    }

    int m = atoi(argv[1]);
    int lift_time = atoi(argv[2]);

    if(m <= 0 && lift_time < 0) {
        fprintf(stderr, "buffer size > 0, time >= 0\n");
        return EXIT_FAILURE;
    }

    FILE *input = fopen(argv[3], "r");
    if(!input) {
        perror("input file");
        return EXIT_FAILURE;
    }
```

```
45
46      FILE *output = fopen(OUTPUT_FILENAME, "w");
47      if(!output) {
48          fclose(input);
49          perror("output file");
50          return EXIT_FAILURE;
51      }
52
53      bool valid = file_validate(input);
54      if(!valid) {
55          fclose(input);
56          fclose(output);
57
58          fprintf(stderr, "invalid input file: %s\n", argv[3]);
59          return EXIT_FAILURE;
60      }
61
62      pthread_t threads[TOTAL_THREADS];
63      struct lift *lifts[TOTAL_LIFTS];
64
65      struct queue *queue = queue_init(m);
66      struct log *logger = log_init(output);
67
68      // thread 0 is task scheduler
69      // thread 1 - TOTAL_THREADS are lift threads
70      for(int i = 0; i < TOTAL_LIFTS; i++) {
71          lifts[i] = lift_init(queue, lift_time, logger, i + 1);
72          pthread_create(&threads[i + 1], NULL, lift, lifts[i]);
73      }
74
75      struct scheduler *s = scheduler_init(input, queue, logger);
76      pthread_create(&threads[0], NULL, scheduler, s);
77
78      // Clean up
79      // join all threads
80      for(int i = 0; i < TOTAL_THREADS; i++) {
81          pthread_join(threads[i], NULL);
82      }
83
84      // Log the final stuff once the threads are dead
85      log_printf(logger, "Total Number of Requests: %d\n"
86                         "Total Number of Movements: %d\n",
87                         s->total_requests, lifts[0]->total_movements
88                                          + lifts[1]->total_movements
89                                          + lifts[2]->total_movements);
90
91      for(int i = 0; i < TOTAL_LIFTS; i++) {
```

```
92          lift_free(lifts[i]);
93      }
94
95      fclose(input);
96      fclose(output);
97
98      log_free(logger);
99      queue_free(queue);
100     scheduler_free(s);
101
102     return EXIT_SUCCESS;
103 }
```