Operating Systems

Lift Simulator - Report

Anurag Singh
18944183

May 15, 2020

Contents

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

2 Testing - Processes

2.1 Example 1

2.2 Example 2

3 Testing - Threads

3.1 Example 1

```
./lift_sim_A 3 1 test/inputs/small.txt
```

Input found at `test/inputs/small.txt`

Output found at `examples/threads/small.out`

The input files has a total of 6 requests, and the output records going through 6 requests, since this is a smaller text file, the output file can be inspected easily to determine if the program is working or not.

3.2 Example 2

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

Input found at `test/inputs/test1.txt`

Output found at `examples/threads/test1.out`

The input files has a total of 50 requests, and the output records going through 50 requests.

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 outputs
↪ available in the `examples` directory.
```

A.2 Common

A.2.1 common.h

```c
#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

```c
#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
```

```c
/**
 * @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;
}

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

```c
#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

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

```c

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

```c
/**
 * @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;
}

```

```c
/**
 * 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

```c
#ifndef LIFT_MAIN_H
#define LIFT_MAIN_H
#include <stdio.h>
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 <common/debug.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;
    D_PRINTF("lift %d : created\n", lift_num);
    // 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) {
            D_PRINTF("lift %d : dead\n", lift_num);
            sem_post(&sm->semaphore.full);
            sem_post(&sm->semaphore.mutex);
            break;
        }
        sem_post(&sm->semaphore.mutex);
        // we release the mutex after checking
        // if the process should die, because
        // to solve the bounded buffer problem
        // we need to wait on the full semaphore
        // which will be posted by the producer
        // waiting while keeping holding the
        // "mutex" will prevent progress, and likely deadlock


        D_PRINTF("lift %d : waiting on mutex\n", lift_num);
        sem_wait(&sm->semaphore.full);
        sem_wait(&sm->semaphore.mutex);
        D_PRINTF("lift %d : mutex grabbed\n", lift_num);

        if(!sm->empty) {
            request_t r = sm_cqueue_remove(sm);
            D_PRINTF("lift %d : moving from %d to %d\n", lift_num, r.src,
            ↪  r.dest);

            sem_wait(&sm->semaphore.file);
            request_num++;
            previous_floor = current_floor;
            current_floor = r.dest;
            int r_movement = abs(previous_floor - r.src) + abs(r.src -
            ↪  current_floor);
            sm->lift_movements[lift_num - 1] += r_movement;

            fprintf(output, "Lift-%d Operation\n"
                            "Previous Floor: %d\n"
                            "Request: Floor %d to %d\n"
                            "Details: \n"
                            "\tGo from Floor %d to %d\n"
                            "\tGo from Floor %d to %d\n"
                            "\t# Movements: %d\n"
                            "\tRequest No: %d\n"
                            "\tTotal # Movement: %d\n"
                            "Current Position: Floor %d\n\n",
                            lift_num, previous_floor, r.src, r.dest,
                            previous_floor, r.src, r.src, current_floor,
                            r_movement, request_num,
                            ↪  sm->lift_movements[lift_num - 1],
                            current_floor);
            fflush(output);

            sem_post(&sm->semaphore.file);
            sem_post(&sm->semaphore.mutex);
            sem_post(&sm->semaphore.empty);
            D_PRINTF("lift %d : mutex released\n", lift_num);

            // release the mutex since the critical section
            // doesn't involve the work we do (in this case)
            // pretend to work
```

```c
            sleep(sleep_time);
        } else {
            sem_post(&sm->semaphore.mutex);
            sem_post(&sm->semaphore.empty);
        }
    }


    fclose(output);
    return 0;
}
```

A.3.3 memory

```c
#ifndef MEMORY_H
#define MEMORY_H
#include <stdbool.h>
#include <semaphore.h>

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

#define SHARED_MEMORY_NAME          "shared_mem"
#define SHARED_MEMORY_REQUESTS      "requests_list"

/**
 * kinda like the stack of
 * memory that'll be shared
 * across processes, also contains
 * the shared semaphores in a separate struct
 */
struct shared_memory {
    request_t *requests;
    int max;
    int current;
    int head;
    int tail;

    int total_requests;
    int lift_movements[TOTAL_LIFTS];

    bool finished;
    bool empty;
    bool full;

    struct semaphore {
        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
```

```c
/**
 * @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)
{
    int shm_fd = shm_open(SHARED_MEMORY_NAME, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, sizeof(struct shared_memory));
    struct shared_memory *ptr = mmap(0, sizeof(struct shared_memory),
     ↪  PROT_WRITE, MAP_SHARED, shm_fd, 0);

    int req_list_fd = shm_open(SHARED_MEMORY_REQUESTS, O_CREAT | O_RDWR,
     ↪  0666);
    ftruncate(req_list_fd, sizeof(request_t) * m);
    ptr->requests = mmap(0, sizeof(request_t) * m, PROT_WRITE, MAP_SHARED,
     ↪  req_list_fd, 0);

    // i think the memory is zero'd out
    // but just incase, we need sane defaults
    ptr->current = 0;
    ptr->max = m;
    ptr->finished = false;
    ptr->empty = true;
    ptr->full = false;
    ptr->head = -1;
    ptr->tail = -1;
    ptr->total_requests = 0;
    for(int i = 0; i < TOTAL_LIFTS; i++) {
        ptr->lift_movements[i] = 0;
    }

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

    return ptr;
}

/**
 * Unlinks and unmaps all of the shared memory
 *
 * @param sm shared memory block
 * @param m buffer size (munmap needs this)
 */
void shared_memory_destroy(struct shared_memory *sm, int m)
{
    sem_destroy(&sm->semaphore.mutex);
    sem_destroy(&sm->semaphore.empty);
    sem_destroy(&sm->semaphore.full);
    sem_destroy(&sm->semaphore.file);

    munmap(sm->requests, sizeof(request_t) * m);
    munmap(sm, sizeof(struct shared_memory));

    shm_unlink(SHARED_MEMORY_REQUESTS);
    shm_unlink(SHARED_MEMORY_NAME);
}
```

A.3.4 scheduler_main

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

```
/**
 * @file scheduler_main.c
 * @author Anurag Singh (18944183)
 *
 * @date 24-04-20
 *
 * the function called after fork to create a scheduler child
 *
 */

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

#include "memory.h"
#include "cqueue.h"
#include "scheduler_main.h"

int scheduler_main(const char *filename, FILE *output)
{
    bool finished = false;
    D_PRINTF("sched %d : created\n", getpid());
    int shm_fd = shm_open(SHARED_MEMORY_NAME, O_RDWR, 0666);
    int req_fd = shm_open(SHARED_MEMORY_REQUESTS, O_RDWR, 0666);
```

```c
    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);

    // Make sure we can actually open the input file
    // Else we mark as finished, and post the full mutex
    // to allow each process to wake up and die
    FILE *fp = fopen(filename, "r");
    if(!fp) {
        sem_wait(&sm->semaphore.mutex);
        sm->finished = true;
        sm->empty = true;
        sem_post(&sm->semaphore.mutex);
        sem_post(&sm->semaphore.full);
        perror("input file");
        fclose(output);
        return 0;
    }

    while(!finished) {
        request_t *r = file_read_line(fp);
        if(r) {
            D_PRINTF("sched %d : waiting on mutex\n", getpid());
            sem_wait(&sm->semaphore.empty);
            sem_wait(&sm->semaphore.mutex);
            D_PRINTF("sched %d : mutex grabbed\n", getpid());
            D_PRINTF("sched : read: %d %d\n", r->src, r->dest);
            sm->total_requests++;
            sm_cqueue_add(sm, *r);

            sem_wait(&sm->semaphore.file);
            fprintf(output, "------------------------------\n"
                            "New Lift Request from %d to %d\n"
                            "Request No: %d\n"
                            "------------------------------\n\n", r->src,
                             ↪  r->dest, sm->total_requests);
            fflush(output);
            sem_post(&sm->semaphore.file);

            sem_post(&sm->semaphore.mutex);
            sem_post(&sm->semaphore.full);
            D_PRINTF("sched %d : mutex released\n", getpid());

            free(r);
        } else {
            sm->finished = true;
            finished = true;
            // wake up anything sleeping before i die
            sem_post(&sm->semaphore.full);
        }

    }

    D_PRINTF("sched %d : dead\n", getpid());
    fclose(fp);
    fclose(output);

    return 0;
}
```

A.3.5 lift_sim_B

```c
/**
 * @file lift_sim_B.c
 * @author Anurag Singh (18944183)
 *
 * @date 24-04-20
 *
 * The actual main of the process part
 * Loads up shared memory, and setups up
 * to fork and create child processes,
 * while also waiting for them to
 * die so we can clean up afterwards
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

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

#include "memory.h"
#include "lift_main.h"
#include "scheduler_main.h"

```

```c
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 >= 1, time > 0\n");
        return EXIT_FAILURE;
    }

    FILE *output = fopen(OUTPUT_FILENAME, "w");
    if(!output) {
        perror("output file");
        return EXIT_FAILURE;
    }

    // Shared Memory
    struct shared_memory *sm = shared_mem_create(m);

    pid_t lifts[TOTAL_LIFTS];
    pid_t scheduler;

    // TODO: Add error checking to fork()
    for(int i = 0; i < TOTAL_LIFTS; i++) {
        if((lifts[i] = fork()) == 0) {
            // Child
            // Each child will immediately exit main by returning
            return lift_main(i + 1, lift_time, output);
        }
    }

    // Create scheduler
    if((scheduler = fork()) == 0) {
        // Child
        return scheduler_main(argv[3], output);
    }

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


    shared_memory_destroy(sm, m);

    fclose(output);

    return 0;
}
```

A.4 Threads

A.4.1 lift

```
#ifndef LIFT_H
#define LIFT_H

#include "queue.h"
#include "log.h"

struct lift {
    int id;
    int lift_time;
    int total_movements;
    struct queue *queue;
    struct log *logger;
};

struct lift* lift_init(struct queue *queue, int lift_time, struct log
↪  *logger, int id);
void lift_free(struct lift *l);
void *lift(void *ptr);
#endif
```

```
/**
 * @file lift.c
 * @author Anurag Singh (18944183)
 *
 * @date 09-05-20
```

```c
 *
 * Handles lift struct and the lift
 * function itself
 *
 */

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

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

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

/**
 * Creates a lift struct with
 * fields initialised to import params
 *
 * @param queue
 * @param lift_time
 * @param logger
 * @param id
 * @return struct lift*
 */
struct lift* lift_init(struct queue *queue, int lift_time, struct log
↪  *logger, int id)
{
    struct lift *l = malloc(sizeof(struct lift));
    l->total_movements = 0;
    l->logger = logger;
    l->queue = queue;
    l->lift_time = lift_time;
    l->id = id;

    return l;
}

/**
 * Frees lift struct pointed to by l
 *
 * @param l lift struct to be free'd
 */
void lift_free(struct lift *l)
```

```c
{
    free(l);
}

/**
 * Lift function passed to pthread_create
 * Handles all mutual exclusion and simulates
 * work by sleeping
 *
 * @param ptr points to the lift struct
 * @return void*
 */
void* lift(void *ptr)
{
    struct lift *l = (struct lift*)ptr;
    // Each lift starts at the first floor
    int current_floor = 1;
    int previous_floor = 1;
    int request_no = 0;

#ifdef DEBUG
    pthread_t t = pthread_self();
    D_PRINTF("consumer created: %d\n", l->id);
#endif

    while(1) {
        pthread_mutex_lock(&l->queue->mutex);

        if(l->queue->empty && l->queue->finished) {
            // let everyone know that the queue is empty
            pthread_cond_broadcast(&l->queue->cond_empty);
            pthread_mutex_unlock(&l->queue->mutex);
            break;
        }

        while(l->queue->empty && !l->queue->finished) {
            D_PRINTF("consumer waiting: %d\n", l->id);
            pthread_cond_wait(&l->queue->cond_empty, &l->queue->mutex);
        }

        if(!l->queue->empty) {
            D_PRINTF("consumer accessing queue: %d\n", l->id);
            request_t *r = queue_remove(l->queue);
            // need to let scheduler thread know that there's space now in
             ↪ the queue
            pthread_cond_signal(&l->queue->cond_full);
            pthread_mutex_unlock(&l->queue->mutex);

            D_PRINTF("consumer working: %d %d sleeping for %d\n", r->src,
             ↪  r->dest, l->lift_time);

            request_no++;
            previous_floor = current_floor;
            current_floor = r->dest;
            int r_movement = abs(previous_floor - r->src) + abs(r->src -
             ↪  current_floor);
            l->total_movements += r_movement;

            log_printf(l->logger, "Lift-%d Operation\n"
                                  "Previous Floor: %d\n"
                                  "Request: Floor %d to %d\n"
                                  "Details: \n"
                                  "\tGo from Floor %d to %d\n"
                                  "\tGo from Floor %d to %d\n"
                                  "\t# Movements: %d\n"
                                  "\tRequest No: %d\n"
                                  "\tTotal # Movement: %d\n"
                                  "Current Position: Floor %d\n\n",
                                  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);
        }
    }

    D_PRINTF("thread -> %ld has died\n", t);

    return NULL;
}
```

A.4.2 log

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

```c

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, ...);

#define FILE_LOG(l, s, ...)                 \
    do {                                    \
        pthread_mutex_lock(&l->mutex);      \
        fprintf(l->fp, s, __VA_ARGS__);     \
        pthread_mutex_unlock(&l->mutex);    \
    } while(0)

#endif
```

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

#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

```c
#ifndef QUEUE_H
#define QUEUE_H
```

```c
#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);
void queue_free(struct queue *queue);

#endif
```

```c
/**
 * @file queue.c
 * @author Anurag Singh (18944183)
 *
 * @date 09-05-20
 *
 * Queue implemenation, backed by a list
 *
 */

#include <stdlib.h>

```

```c
#include <common/request.h>
#include "queue.h"

/**
 * Initialises a queue struct with
 * a maximum size of m
 *
 * @param m max size
 * @return struct queue*
 */
struct queue* queue_init(int m)
{
    struct queue *queue = malloc(sizeof(struct queue));
    queue->max = m;
    queue->count = 0;
    queue->head = NULL;
    queue->tail = NULL;
    queue->empty = true;
    queue->full = false;
    queue->finished = false;

    pthread_mutex_init(&queue->mutex, NULL);
    pthread_cond_init(&queue->cond_empty, NULL);
    pthread_cond_init(&queue->cond_full, NULL);

    return queue;
}

/**
 * Creates a node containing the data
 *
 * @param data data to put in the node
 * @return struct node*
 */
static struct node* node_init(request_t *data)
{
    struct node *node = malloc(sizeof(struct node));
    node->next = NULL;
    node->data = data;

    return node;
}

/**
 * Adds to the end of the list
 * to allow FIFO behaviour
 *
 * @param queue queue to add to
 * @param ptr data to add
 */
void queue_add(struct queue *queue, request_t *ptr)
{
    if(queue->count == queue->max) {
        return;
    }

    struct node *node = node_init(ptr);

    if(queue->tail == NULL) {
        queue->head = node;
        queue->tail = node;
    } else {
        queue->tail->next = node;
        queue->tail = node;
    }

    queue->count++;
    if(queue->count == queue->max) {
        queue->full = true;
    }

    queue->empty = false;
}

/**
 * Removes from the front of the queue
 *
 * @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;
```

```c
        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;

    while(current) {
        struct node *next = current->next;

        node_free(current);

        current = next;
    }

    pthread_mutex_destroy(&queue->mutex);
    pthread_cond_destroy(&queue->cond_empty);
    pthread_cond_destroy(&queue->cond_full);
    free(queue);
}
```

A.4.4 scheduler

```c
#ifndef SCHEDULER_H
#define SCHEDULER_H

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

#include "queue.h"
#include "log.h"

struct scheduler {
    int total_requests;
    FILE *input;
    struct queue *queue;
    struct log *logger;
};

struct scheduler* scheduler_init(FILE *input, struct queue *queue, struct log
↪  *logger);
void scheduler_free(struct scheduler *s);
void* scheduler(void *ptr);

#endif
```

```c
/**
 * @file scheduler.c
 * @author Anurag Singh (18944183)
 *
 * @date 09-05-20
 *
 * 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 <common/debug.h>

#include "scheduler.h"

```

```c
// TODO: Figure out why printing is sometimes 3 tasks added

/**
 * 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;

#ifdef DEBUG
```

```c
    pthread_t t = pthread_self();
    D_PRINTF("prod created: %ld\n", t);
#endif

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

        D_PRINTF("prod: q rn: %d\n", s->queue->count);

        while(s->queue->full) {
            // if queue at max capacity, wait for someone to dequeue
            D_PRINTF("prod: q at max: %d\n", s->queue->count);
            pthread_cond_wait(&s->queue->cond_full, &s->queue->mutex);
        }

        request_t *r = file_read_line(s->input);
        if(r) {
            D_PRINTF("prod: added : %d %d\n", r->src, r->dest);
            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);
    }

    D_PRINTF("prod has died %s\n",  "lol");

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

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

// TODO: Report
// TODO: Commenting
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 >= 1, time > 0\n");
        return EXIT_FAILURE;
    }

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

    FILE *output = fopen(OUTPUT_FILENAME, "w");
    if(!output) {
        if(input) {
            fclose(input);
        }
        perror("output file");
        return EXIT_FAILURE;
    }

    bool valid = file_validate(input);
    if(!valid) {
        if(input) {
            fclose(input);
        }

        if(output) {
            fclose(output);
        }

        fprintf(stderr, "invalid input file: %s\n", argv[3]);

        return EXIT_FAILURE;
    }

    pthread_t threads[TOTAL_THREADS];
    struct lift *lifts[TOTAL_LIFTS];

    struct queue *queue = queue_init(m);
    struct log *logger = log_init(output);

    // thread 0 is task scheduler
    // thread 1 - TOTAL_THREADS are lift threads
    for(int i = 0; i < TOTAL_LIFTS; i++) {
        lifts[i] = lift_init(queue, lift_time, logger, i + 1);
        pthread_create(&threads[i + 1], NULL, lift, lifts[i]);
    }

    struct scheduler *s = scheduler_init(input, queue, logger);
    pthread_create(&threads[0], NULL, scheduler, s);

    // Clean up
    // join all threads
    for(int i = 0; i < TOTAL_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    D_PRINTF("final : %d %d %d %d\n", s->total_requests,
```

```c
                lifts[0]->total_movements,
                lifts[1]->total_movements,
                lifts[2]->total_movements);

    // Log the final stuff once the threads are dead
    log_printf(logger, "Total Number of Requests: %d\n"
                       "Total Number of Movements: %d\n",
                       s->total_requests, lifts[0]->total_movements
                                        + lifts[1]->total_movements
                                        + lifts[2]->total_movements);

    for(int i = 0; i < TOTAL_LIFTS; i++) {
        lift_free(lifts[i]);
    }

    fclose(input);
    fclose(output);

    log_free(logger);
    queue_free(queue);
    scheduler_free(s);

    return EXIT_SUCCESS;
}
```