

OPERATING SYSTEMS

Lift Simulator - Report

Anurag Singh

18944183

May 15, 2020

Contents

1 Mutual Exclusion and Shared Resources	1
1.1 Threads	1
1.2 Processes	2
2 Testing - Processes	3
2.1 Example 1	3
2.2 Example 2	3
3 Testing - Threads	3
3.1 Example 1	3
3.2 Example 2	3
A Source Code	4
A.1 README.md	4
A.2 Common	5
A.2.1 common.h	5
A.2.2 file_io	5
A.2.3 request.h	7
A.3 Processes	7
A.3.1 cqueue	7
A.3.2 lift_main	9
A.3.3 memory	12
A.3.4 scheduler_main	15
A.3.5 lift_sim_B	17
A.4 Threads	19
A.4.1 lift	19
A.4.2 log	22
A.4.3 queue	24
A.4.4 scheduler	29
A.4.5 lift_sim_A	31

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.

```
1 ...
2 pthread_mutex_lock(&l->queue->mutex);
3 ...
4
5 while(l->queue->empty && !l->queue->finished) {
6     pthread_cond_wait(&l->queue->cond_empty, &l->queue->mutex);
7 }
8
9 if(!l->queue->empty) {
10    request_t *r = queue_remove(l->queue);
11    pthread_cond_signal(&l->queue->cond_full);
12    pthread_mutex_unlock(&l->queue->mutex);
13 }
```

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.

```
1 ...
2 pthread_mutex_lock(&s->queue->mutex);
3
4 while(s->queue->full) {
5     pthread_cond_wait(&s->queue->cond_full, &s->queue->mutex);
6 }
7
8 request_t *r = file_read_line(s->input);
9 if(r) {
10    queue_add(s->queue, r);
11    ...
12    pthread_mutex_unlock(&s->queue->mutex);
13 }
```

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

```
1 # Lift Simulator
2
3 #### Anurag Singh
4
5 ## Introduction
6
7 Lift Simulator is a solution to the Operating Systems (COMP2006) assignment
8 → in Semester 1 2020.
9
10 ## Purpose
11
12 To demonstrate multi-threading, and inter process/thread communication, and
13 → solving the critical section problem.
14
15 ## Building
16
17 Running `make` will build 2 executables, `lift_sim_A` and `lift_sim_B` which
18 → refer to threads and processes implementations respectively.
19
20 ## Usage
21
22 Both lift_sim_A, and lift_sim_B are used the same way, the program will write
23 → to the sim_out file as it runs.
24
25
26
27 $ ./lift_sim_{A,B} [max buffer size] [time per sleep] [input file]
```

A.2 Common

A.2.1 common.h

```
1 #ifndef COMMON_H
2 #define COMMON_H
3 /**
4  * Some common definitions between
5  * threads and processes implementation
6  */
7 #define OUTPUT_FILENAME          "sim_out"
8 #define TOTAL_THREADS            4
9 #define TOTAL_LIFTS              3
10 #define MAX_FLOOR                20
11 #define MIN_FLOOR                1
12 #define MIN_INPUT                50
13 #define MAX_INPUT               100
14
15 #endif
```

A.2.2 file_io

```
1 #ifndef FILE_IO_H
2 #define FILE_IO_H
3 #include <stdio.h>
4 #include <stdbool.h>
5
6 #include "request.h"
7
8 request_t* file_read_line(FILE *file);
9 bool file_validate(FILE *file);
10 #endif
```

```
1 /**
2  * @file file_io.c
3  * @author Anurag Singh (18944183)
4  *
5  * @date 24-04-20
6  *
7  */
8
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <stdbool.h>
12
13 #include "common.h"
```

```

14 #include "file_io.h"
15 #include "request.h"
16
17 /**
18 * This function reads a line
19 * from the input file, parses it
20 * in the format specified, and
21 * does a simple sanity check to
22 * ensure that its valid, before
23 * malloc'ing a struct and returning
24 * it containing the values read.
25 *
26 * @param file to read the line from
27 * @return malloc'd request_t* containing the values
28 */
29 request_t* file_read_line(FILE *file)
30 {
31     int src, dest;
32     char buffer[BUFSIZ] = {0};
33
34     if(!fgets(buffer, BUFSIZ, file)) {
35         return NULL;
36     }
37
38     int ret = sscanf(buffer, "%d %d", &src, &dest);
39     if(ret != 2) {
40         // incase 2 values weren't read from the line
41         return NULL;
42     }
43
44     // bounds check the input
45     if(src > MAX_FLOOR || dest > MAX_FLOOR || src < MIN_FLOOR || dest <
46     ↪ MIN_FLOOR) {
47         return NULL;
48     }
49
50     request_t *req = malloc(sizeof(request_t));
51     req->src = src;
52     req->dest = dest;
53
54     return req;
55 }
56
57 bool file_validate(FILE *file)
58 {
59     request_t *req;
60     int count = 0;

```

```

60
61     while((req = file_read_line(file))) {
62         count++;
63         free(req);
64     }
65
66     // If we didn't reach eof
67     // it means there was an invalid line
68     if(!feof(file)) {
69         return false;
70     }
71
72     // If the overall count is less than or more than
73     // its also invalid
74     if(count > MAX_INPUT || count < MIN_INPUT) {
75         return false;
76     }
77
78     rewind(file);
79
80     return true;
81 }
```

A.2.3 request.h

```

1 #ifndef REQUEST_H
2 #define REQUEST_H
3
4 /**
5  * Each request is stored as a struct
6  */
7 typedef struct request {
8     int src;
9     int dest;
10} request_t;
11
12#endif
```

A.3 Processes

A.3.1 cqueue

```

1 #ifndef CQUEUE_H
2 #define CQUEUE_H
3 #include "memory.h"
```

```

4
5 void sm_cqueue_add(struct shared_memory *sm, request_t r);
6 request_t sm_cqueue_remove(struct shared_memory *sm);
7 #endif

```

```

1 /**
2  * @file cqueue.c
3  * @author Anurag Singh (18944183)
4  *
5  * @date 24-04-20
6  *
7  * A simple circular queue "implementation"
8  *
9 */
10
11 #include <common/request.h>
12
13 #include "memory.h"
14
15 /**
16  * Add a request to the queue stored in
17  * the shared memory param
18  *
19  * @param sm shared memory struct
20  * @param r request to add to queue
21  */
22 void sm_cqueue_add(struct shared_memory *sm, request_t r)
23 {
24     if(sm->head == -1) {
25         sm->head = 0;
26         sm->tail = 0;
27     } else {
28         if(sm->tail == sm->max - 1) {
29             sm->tail = 0;
30         } else {
31             sm->tail++;
32         }
33     }
34
35     sm->current++;
36     if(sm->current == sm->max) {
37         sm->full = true;
38     }
39     sm->empty = false;
40     sm->requests[sm->tail] = r;
41 }
42

```

```

43 /**
44 * Remove a request from the queue
45 *
46 * @param sm memory to remove from
47 * @return request_t
48 */
49 request_t sm_cqueue_remove(struct shared_memory *sm)
50 {
51     request_t r = sm->requests[sm->head];
52
53     if(sm->head == sm->tail) {
54         sm->head = -1;
55         sm->tail = -1;
56     } else {
57         if(sm->head == sm->max - 1) {
58             sm->head = 0;
59         } else {
60             sm->head++;
61         }
62     }
63
64     sm->current--;
65     if(sm->current == 0) {
66         sm->empty = true;
67     }
68     sm->full = false;
69
70     return r;
71 }
```

A.3.2 lift_main

```

1 #ifndef LIFT_MAIN_H
2 #define LIFT_MAIN_H
3 #include <stdio.h>
4 int lift_main(int lift_num, int sleep_time, FILE *output);
5
6 #endif
```

```

1 /**
2 * @file lift_main.c
3 * @author Anurag Singh (18944183)
4 *
5 * @date 24-04-20
6 *
7 * The function called when a lift process needs
```

```

8  * to be created. Effectively the "main" of the lift.
9  *
10 */
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <unistd.h>
15 #include <sys/types.h>
16 #include <sys/wait.h>
17 #include <sys/mman.h>
18 #include <sys/stat.h>
19 #include <fcntl.h>
20
21 #include <common/debug.h>
22
23 #include "memory.h"
24 #include "cqueue.h"
25 #include "lift_main.h"
26
27 int lift_main(int lift_num, int sleep_time, FILE *output)
{
28
29     int previous_floor = 1;
30     int current_floor = 1;
31     int request_num = 0;
32     D_PRINTF("lift %d : created\n", lift_num);
33     // Get access to the shared memory
34     int shm_fd = shm_open(SHARED_MEMORY_NAME, O_RDWR, 0666);
35     int req_fd = shm_open(SHARED_MEMORY_REQUESTS, O_RDWR, 0666);
36     struct shared_memory *sm = mmap(0, sizeof(struct shared_memory),
37         PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
38     sm->requests = mmap(0, sizeof(request_t) * sm->max, PROT_READ |
39         PROT_WRITE, MAP_SHARED, req_fd, 0);
40
41     while(1) {
42         sem_wait(&sm->semaphore.mutex);
43         if(sm->empty && sm->finished) {
44             D_PRINTF("lift %d : dead\n", lift_num);
45             sem_post(&sm->semaphore.full);
46             sem_post(&sm->semaphore.mutex);
47             break;
48         }
49         sem_post(&sm->semaphore.mutex);
50         // we release the mutex after checking
51         // if the process should die, because
52         // to solve the bounded buffer problem
53         // we need to wait on the full semaphore
54         // which will be posted by the producer

```

```

53 // waiting while keeping holding the
54 // "mutex" will prevent progress, and likely deadlock
55
56
57 D_PRINTF("lift %d : waiting on mutex\n", lift_num);
58 sem_wait(&sm->semaphore.full);
59 sem_wait(&sm->semaphore.mutex);
60 D_PRINTF("lift %d : mutex grabbed\n", lift_num);
61
62 if(!sm->empty) {
63     request_t r = sm_cqueue_remove(sm);
64     D_PRINTF("lift %d : moving from %d to %d\n", lift_num, r.src,
65             ↪ r.dest);
66
67     sem_wait(&sm->semaphore.file);
68     request_num++;
69     previous_floor = current_floor;
70     current_floor = r.dest;
71     int r_movement = abs(previous_floor - r.src) + abs(r.src -
72             ↪ current_floor);
73     sm->lift_movements[lift_num - 1] += r_movement;
74
75     fprintf(output, "Lift-%d Operation\n"
76             "Previous Floor: %d\n"
77             "Request: Floor %d to %d\n"
78             "Details: \n"
79             "\tGo from Floor %d to %d\n"
80             "\tGo from Floor %d to %d\n"
81             "\t# Movements: %d\n"
82             "\tRequest No: %d\n"
83             "\tTotal # Movement: %d\n"
84             "Current Position: Floor %d\n",
85             lift_num, previous_floor, r.src, r.dest,
86             previous_floor, r.src, r.src, current_floor,
87             r_movement, request_num,
88             ↪ sm->lift_movements[lift_num - 1],
89             current_floor);
90
91     fflush(output);
92
93     sem_post(&sm->semaphore.file);
94     sem_post(&sm->semaphore.mutex);
95     sem_post(&sm->semaphore.empty);
96     D_PRINTF("lift %d : mutex released\n", lift_num);
97
98     // release the mutex since the critical section
99     // doesn't involve the work we do (in this case)
100    // pretend to work

```

```

97         sleep(sleep_time);
98     } else {
99         sem_post(&sm->semaphore.mutex);
100        sem_post(&sm->semaphore.empty);
101    }
102}
103
104
105 fclose(output);
106 return 0;
107}

```

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
32     struct semaphore {

```

```

33         sem_t full;
34         sem_t empty;
35         sem_t mutex;
36         sem_t file;
37     } semaphore;
38 }
39
40 struct shared_memory* shared_mem_create(int m);
41 void shared_memory_destroy(struct shared_memory *sm, int m);
42
43 #endif

```

```

1 /**
2  * @file memory.c
3  * @author Anurag Singh (18944183)
4  *
5  * @date 24-04-20
6  *
7  * Handles allocating and freeing shared memory
8  *
9  */
10
11 // ftruncate
12 #define _POSIX_C_SOURCE 200112L
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/request.h>
21 #include <common/common.h>
22
23 #include "memory.h"
24
25 /**
26  * Create the shared memory
27  * initialising all fields to sane defaults
28  * Uses the POSIX shared memory API
29  *
30  * @param m buffer size
31  * @return struct shared_memory*
32  */
33 struct shared_memory* shared_mem_create(int m)
34 {
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),
38         → PROT_WRITE, MAP_SHARED, shm_fd, 0);
39
39     int req_list_fd = shm_open(SHARED_MEMORY_REQUESTS, O_CREAT | O_RDWR,
40         → 0666);
40     ftruncate(req_list_fd, sizeof(request_t) * m);
41     ptr->requests = mmap(0, sizeof(request_t) * m, PROT_WRITE, MAP_SHARED,
42         → req_list_fd, 0);
43
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
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
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 #include <common/debug.h>
22
23 #include "memory.h"
24 #include "cqueue.h"
25 #include "scheduler_main.h"
26
27 int scheduler_main(const char *filename, FILE *output)
28 {
29     bool finished = false;
30     DPRINTF("sched %d : created\n", getpid());
31     int shm_fd = shm_open(SHARED_MEMORY_NAME, O_RDWR, 0666);
32     int req_fd = shm_open(SHARED_MEMORY_REQUESTS, O_RDWR, 0666);
```

```

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

```

```

77         finished = true;
78         // wake up anything sleeping before i die
79         sem_post(&sm->semaphore.full);
80     }
81
82 }
83
84 D_PRINTF("sched %d : dead\n", getpid());
85 fclose(fp);
86 fclose(output);
87
88 return 0;
89 }
```

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 <sys/types.h>
20 #include <sys/wait.h>
21
22 #include <common/request.h>
23 #include <common/file_io.h>
24 #include <common/debug.h>
25 #include <common/common.h>
26
27 #include "memory.h"
28 #include "lift_main.h"
29 #include "scheduler_main.h"
```

```

31 int main(int argc, const char *argv[])
32 {
33     if(argc < 4) {
34         fprintf(stderr, "usage: %s [buffer size] [time] [filename]\n",
35             argv[0]);
36         return EXIT_FAILURE;
37     }
38
39     int m = atoi(argv[1]);
40     int lift_time = atoi(argv[2]);
41
42     if(m <= 0 && lift_time < 0) {
43         fprintf(stderr, "buffer size >= 1, time > 0\n");
44         return EXIT_FAILURE;
45     }
46
47     FILE *output = fopen(OUTPUT_FILENAME, "w");
48     if(!output) {
49         perror("output file");
50         return EXIT_FAILURE;
51     }
52
53     // Shared Memory
54     struct shared_memory *sm = shared_mem_create(m);
55
56     pid_t lifts[TOTAL_LIFTS];
57     pid_t scheduler;
58
59     // TODO: Add error checking to fork()
60     for(int i = 0; i < TOTAL_LIFTS; i++) {
61         if((lifts[i] = fork()) == 0) {
62             // Child
63             // Each child will immediately exit main by returning
64             return lift_main(i + 1, lift_time, output);
65         }
66     }
67
68     // Create scheduler
69     if((scheduler = fork()) == 0) {
70         // Child
71         return scheduler_main(argv[3], output);
72     }
73
74     // Parent
75     for(int i = 0; i < TOTAL_LIFTS; i++) {
76         waitpid(lifts[i], NULL, 0);
77     }

```

```

77
78     waitpid(scheduler, NULL, 0);
79
80     fprintf(output, "Total Number of Requests: %d\n"
81             "Total Number of Movements: %d\n",
82             sm->total_requests, sm->lift_movements[0]
83                     + sm->lift_movements[1]
84                     + sm->lift_movements[2]);
85
86
87     shared_memory_destroy(sm, m);
88
89     fclose(output);
90
91     return 0;
92 }
```

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
16                         *logger, int id);
17 void lift_free(struct lift *l);
18 void *lift(void *ptr);
19 #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 #include <common/debug.h>
19
20 #include "lift.h"
21 #include "queue.h"
22 #include "log.h"
23
24 /**
25  * Creates a lift struct with
26  * fields initialised to import params
27  *
28  * @param queue
29  * @param lift_time
30  * @param logger
31  * @param id
32  * @return struct lift*
33  */
34 struct lift* lift_init(struct queue *queue, int lift_time, struct log
35  ↵ *logger, int id)
36 {
37     struct lift *l = malloc(sizeof(struct lift));
38     l->total_movements = 0;
39     l->logger = logger;
40     l->queue = queue;
41     l->lift_time = lift_time;
42     l->id = id;
43
44     return l;
45 }
46
47 /**
48  * Frees lift struct pointed to by l
49  *
50  * @param l lift struct to be free'd
51 */
void lift_free(struct lift *l)

```

```

52     {
53         free(l);
54     }
55
56 /**
57 * Lift function passed to pthread_create
58 * Handles all mutual exclusion and simulates
59 * work by sleeping
60 *
61 * @param ptr points to the lift struct
62 * @return void*
63 */
64 void* lift(void *ptr)
65 {
66     struct lift *l = (struct lift*)ptr;
67     // Each lift starts at the first floor
68     int current_floor = 1;
69     int previous_floor = 1;
70     int request_no = 0;
71
72 #ifdef DEBUG
73     pthread_t t = pthread_self();
74     D_PRINTF("consumer created: %d\n", l->id);
75 #endif
76
77     while(1) {
78         pthread_mutex_lock(&l->queue->mutex);
79
80         if(l->queue->empty && l->queue->finished) {
81             // let everyone know that the queue is empty
82             pthread_cond_broadcast(&l->queue->cond_empty);
83             pthread_mutex_unlock(&l->queue->mutex);
84             break;
85         }
86
87         while(l->queue->empty && !l->queue->finished) {
88             D_PRINTF("consumer waiting: %d\n", l->id);
89             pthread_cond_wait(&l->queue->cond_empty, &l->queue->mutex);
90         }
91
92         if(!l->queue->empty) {
93             D_PRINTF("consumer accessing queue: %d\n", l->id);
94             request_t *r = queue_remove(l->queue);
95             // need to let scheduler thread know that there's space now in
96             // the queue
97             pthread_cond_signal(&l->queue->cond_full);
98             pthread_mutex_unlock(&l->queue->mutex);

```

```

98
99     D_PRINTF("consumer working: %d %d sleeping for %d\n", r->src,
100    ↵   r->dest, l->lift_time);
101
102    request_no++;
103    previous_floor = current_floor;
104    current_floor = r->dest;
105    int r_movement = abs(previous_floor - r->src) + abs(r->src -
106    ↵   current_floor);
107    l->total_movements += r_movement;
108
109    log_printf(l->logger, "Lift-%d Operation\n"
110               "Previous Floor: %d\n"
111               "Request: Floor %d to %d\n"
112               "Details: \n"
113               "\tGo from Floor %d to %d\n"
114               "\tGo from Floor %d to %d\n"
115               "\t# Movements: %d\n"
116               "\tRequest No: %d\n"
117               "\tTotal # Movement: %d\n"
118               "Current Position: Floor %d\n\n",
119               l->id, previous_floor, r->src, r->dest,
120               previous_floor, r->src, r->src,
121               ↵   current_floor,
122               r_movement, request_no, l->total_movements,
123               current_floor);
124
125    sleep(l->lift_time);
126    free(r);
127    // Simulate work by putting the thread to sleep
128 } else {
129     pthread_mutex_unlock(&l->queue->mutex);
130 }
131
132 D_PRINTF("thread -> %ld has died\n", t);
133
134 return NULL;
}

```

A.4.2 log

```

1 #ifndef LOG_H
2 #define LOG_H
3 #include <stdio.h>
4 #include <pthread.h>

```

```

5
6 struct log {
7     pthread_mutex_t mutex;
8     FILE *fp;
9 };
10
11 struct log* log_init(FILE *fp);
12 void log_free(struct log *l);
13 void log_printf(struct log *l, const char *s, ...);
14
15 #define FILE_LOG(l, s, ...) \
16     do { \
17         pthread_mutex_lock(&l->mutex); \
18         fprintf(l->fp, s, __VA_ARGS__); \
19         pthread_mutex_unlock(&l->mutex); \
20     } while(0)
21
22 #endif

```

```

1 /**
2  * @file log.c
3  * @author Anurag Singh (18944183)
4  *
5  * @date 09-05-20
6  *
7  * Handles log struct creation
8  * and mutual exclusion for logging
9  * to file
10 *
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <stdarg.h>
16
17 #include "log.h"
18
19 /**
20  * Creates a log struct with
21  * file pointer fp, initialises
22  * the mutex used to protect FILE io
23  *
24  * @param fp
25  * @return struct log*
26  */
27 struct log* log_init(FILE *fp)
28 {

```

```

29     struct log *l = malloc(sizeof(struct log));
30
31     l->fp = fp;
32     pthread_mutex_init(&l->mutex, NULL);
33
34     return l;
35 }
36
37 /**
38 * Frees the log struct, and destroys the mutex
39 *
40 * @param l log struct to be freed
41 */
42 void log_free(struct log *l)
43 {
44     // File will be closed by whoever opened it
45     // eg. the main thread in this case
46     pthread_mutex_destroy(&l->mutex);
47     free(l);
48 }
49
50 /**
51 * printf like function that allows printing
52 * to the FILE in the log struct, while utilising
53 * the mutex to protect it
54 *
55 * @param l
56 * @param s
57 * @param ...
58 */
59 void log_printf(struct log *l, const char *s, ...)
60 {
61     va_list args;
62     va_start(args, s);
63
64     pthread_mutex_lock(&l->mutex);
65     vfprintf(l->fp, s, args);
66     pthread_mutex_unlock(&l->mutex);
67
68     va_end(args);
69 }
```

A.4.3 queue

```

1 #ifndef QUEUE_H
2 #define QUEUE_H
```

```

3 #include <stdbool.h>
4 #include <pthread.h>
5
6 #include <common/request.h>
7
8 struct node {
9     struct node *next;
10    request_t *data;
11};
12
13 /// Queue backed by a List
14 struct queue {
15     pthread_mutex_t mutex;
16     pthread_cond_t cond_empty;
17     pthread_cond_t cond_full;
18
19     struct node *head;
20     struct node *tail;
21
22     bool full;
23     bool empty;
24
25     bool finished;
26
27     int max;
28     int count;
29};
30
31 struct queue* queue_init(int m);
32 void queue_add(struct queue *queue, request_t *node);
33 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 implementation, backed by a list
8  *
9  */
10
11 #include <stdlib.h>

```

```

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
91 * @return request_t*
92 */
93 request_t* queue_remove(struct queue *queue)
94 {
95     struct node *node = queue->head;
96
97     if(!node) {
98         return NULL;
99     }
100
101    void *data = queue->head->data;
102
103    if(queue->head && queue->head->next) {
104        queue->head = queue->head->next;
105    } else if(!queue->head->next) {
106        queue->head = NULL;

```

```

107         queue->tail = NULL;
108         queue->empty = true;
109     }
110
111     queue->full = false;
112     queue->count--;
113     free(node);
114
115     return data;
116 }
117
118 /**
119 * Frees a node
120 *
121 * @param node
122 */
123 static void node_free(struct node *node)
124 {
125     free(node->data);
126     free(node);
127 }
128
129 /**
130 * Frees the entire queue including
131 * the struct and any mutexes
132 *
133 * @param queue
134 */
135 void queue_free(struct queue *queue)
136 {
137     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
18 → *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 *
7 * Handles all scheduler based actions
8 * including allocating the structure
9 * and the function passed to pthread_create
10 *
11 */
12
13 #include <stdlib.h>
14 #include <string.h>
15 #include <pthread.h>
16
17 #include <common/file_io.h>
18 #include <common/request.h>
19 #include <common/debug.h>
20
21 #include "scheduler.h"
```

```

23 // TODO: Figure out why printing is sometimes 3 tasks added
24
25 /**
26 * Create a scheduler struct, initialising struct
27 * fields to imported variables
28 *
29 * @param input
30 * @param queue
31 * @param logger
32 * @return struct scheduler*
33 */
34 struct scheduler* scheduler_init(FILE *input, struct queue *queue, struct log
35   ↵ *logger)
36 {
37     struct scheduler *s = malloc(sizeof(struct scheduler));
38     s->total_requests = 0;
39     s->input = input;
40     s->logger = logger;
41     s->queue = queue;
42
43     return s;
44 }
45
46 /**
47 * Frees a scheduler struct
48 *
49 * @param s
50 */
51 void scheduler_free(struct scheduler *s)
52 {
53     free(s);
54 }
55
56 /**
57 * Function passed to pthread_create
58 * in order to create a scheduler struct
59 * Contains all mutual exclusion required
60 *
61 * @param ptr scheduler struct
62 * @return void*
63 */
64 void* scheduler(void *ptr)
65 {
66     struct scheduler *s = (struct scheduler *)ptr;
67     bool finished = false;
68
69 #ifdef DEBUG

```

```

69     pthread_t t = pthread_self();
70     D_PRINTF("prod created: %ld\n", t);
71 #endif
72
73     while(!finished) {
74         pthread_mutex_lock(&s->queue->mutex);
75
76         D_PRINTF("prod: q rn: %d\n", s->queue->count);
77
78         while(s->queue->full) {
79             // if queue at max capacity, wait for someone to dequeue
80             D_PRINTF("prod: q at max: %d\n", s->queue->count);
81             pthread_cond_wait(&s->queue->cond_full, &s->queue->mutex);
82         }
83
84         request_t *r = file_read_line(s->input);
85         if(r) {
86             D_PRINTF("prod: added : %d %d\n", r->src, r->dest);
87             queue_add(s->queue, r);
88             s->total_requests++;
89             log_printf(s->logger, "-----\n"
90                         "New Lift Request from %d to %d\n"
91                         "Request No: %d\n"
92                         "-----\n\n",
93                         r->src, r->dest, s->total_requests);
94             pthread_cond_signal(&s->queue->cond_empty);
95         } else {
96             // let everyone know we got nothing to put in
97             // for anyone waiting for something to be added
98             pthread_cond_broadcast(&s->queue->cond_empty);
99             s->queue->finished = true;
100            finished = true;
101        }
102
103        pthread_mutex_unlock(&s->queue->mutex);
104    }
105
106    D_PRINTF("prod has died %s\n", "lol");
107
108    return NULL;
}

```

A.4.5 lift_sim_A

```

1 /**
2  * @file lift_sim_A.c

```

```

3  * @author Anurag Singh (18944183)
4  *
5  * @date 09-05-20
6  *
7  * The main of the threads part, handles
8  * logging, queue and creation and
9  * destruction of threads.
10 *
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15
16 #include <common/request.h>
17 #include <common/file_io.h>
18 #include <common/debug.h>
19 #include <common/common.h>
20
21 #include "queue.h"
22 #include "scheduler.h"
23 #include "lift.h"
24 #include "log.h"
25
26 // TODO: Report
27 // TODO: Commenting
28 int main(int argc, const char *argv[])
29 {
30     if(argc < 4) {
31         fprintf(stderr, "usage: %s [buffer size] [time] [filename]\n",
32             argv[0]);
33         return EXIT_FAILURE;
34     }
35
36     int m = atoi(argv[1]);
37     int lift_time = atoi(argv[2]);
38
39     if(m <= 0 && lift_time < 0) {
40         fprintf(stderr, "buffer size >= 1, time > 0\n");
41         return EXIT_FAILURE;
42     }
43
44     FILE *input = fopen(argv[3], "r");
45     if(!input) {
46         perror("input file");
47         return EXIT_FAILURE;
48     }

```

```

49     FILE *output = fopen(OUTPUT_FILENAME, "w");
50     if(!output) {
51         if(input) {
52             fclose(input);
53         }
54         perror("output file");
55         return EXIT_FAILURE;
56     }
57
58     bool valid = file_validate(input);
59     if(!valid) {
60         if(input) {
61             fclose(input);
62         }
63
64         if(output) {
65             fclose(output);
66         }
67
68         fprintf(stderr, "invalid input file: %s\n", argv[3]);
69
70         return EXIT_FAILURE;
71     }
72
73     pthread_t threads[TOTAL_THREADS];
74     struct lift *lifts[TOTAL_LIFTS];
75
76     struct queue *queue = queue_init(m);
77     struct log *logger = log_init(output);
78
79     // thread 0 is task scheduler
80     // thread 1 - TOTAL_THREADS are lift threads
81     for(int i = 0; i < TOTAL_LIFTS; i++) {
82         lifts[i] = lift_init(queue, lift_time, logger, i + 1);
83         pthread_create(&threads[i + 1], NULL, lift, lifts[i]);
84     }
85
86     struct scheduler *s = scheduler_init(input, queue, logger);
87     pthread_create(&threads[0], NULL, scheduler, s);
88
89     // Clean up
90     // join all threads
91     for(int i = 0; i < TOTAL_THREADS; i++) {
92         pthread_join(threads[i], NULL);
93     }
94
95     D_PRINTF("final : %d %d %d %d\n", s->total_requests,

```

```

96         lifts[0]->total_movements,
97         lifts[1]->total_movements,
98         lifts[2]->total_movements);
99
100    // Log the final stuff once the threads are dead
101    log_printf(logger, "Total Number of Requests: %d\n"
102                "Total Number of Movements: %d\n",
103                s->total_requests, lifts[0]->total_movements
104                + lifts[1]->total_movements
105                + lifts[2]->total_movements);
106
107    for(int i = 0; i < TOTAL_LIFTS; i++) {
108        lift_free(lifts[i]);
109    }
110
111    fclose(input);
112    fclose(output);
113
114    log_free(logger);
115    queue_free(queue);
116    scheduler_free(s);
117
118    return EXIT_SUCCESS;
119}

```