Operating Systems (COMP2006) Assignment 2020
Lift-simulator

Nhan Dao
Student ID: 18843905

May 8, 2020

1 Problem

There are three elevator, **Lift-1**, **Lift-2**, **Lift-3**, which are servicing a 20-floor building (Floors 1 to 20). Assume that initially all lifts are in Floor 1. Each **lift** waits for lift **requests** from any floor (1 to 20) and serves one **request** at a time. One process will be created to generate lift requests from an input file, and three processes will be created to serve one request at a time.

This problem is generally known as the **bounded-buffer** (aka the **producer-consumer** problem). A **producer** process produces data in which the **consumer** process uses, since these processes runs concurrently, there must a buffer available in which the producer can place in the data and the consumer can remove data from. A **bounded buffer** assumes that there is a fixed size of the buffer, which means that the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

2 FIFO Buffer

As mentioned in section 1, a bounded buffer was required to share data between the concurrent processes. The buffer was implemented as a FIFO queue using an array with circular pointers which ensures that the each lift request was served in the correct ordering.

There are two subroutine to create the buffer. The first is the function **fifo_buf_init_malloc()** which initialise the buffer and create memory on the heap using **malloc()**. This function is used for creating memory that can only be access within the same process. The second subroutine, **fifo_buf_init_mmap()**, creates the buffer using **mmap()** which allows the data to be shared between processes.

```
1  #include <stdbool.h>
2  #include "lift.h"
3
4  #ifndef FIFO_BUF_H
5  #define FIFO_BUF_H
6
7  #define FIFO_BUF_ERR_FULLENQ -1
8  #define FIFO_BUF_ERR_EMPTYDEQ -2
9
10 typedef struct _fifo_buf_t {
```

```
11
12      LiftRequest* requests;
13      size_t head;
14      size_t tail;
15      size_t max;
16      size_t count;
17
18  } fifo_buf_t;
19
20  fifo_buf_t* fifo_buf_init_malloc(size_t size);
21  fifo_buf_t* fifo_buf_init_mmap(size_t size);
22  bool fifo_buf_full(fifo_buf_t* buf);
23  bool fifo_buf_empty(fifo_buf_t* buf);
24  void fifo_buf_destroy_malloc(fifo_buf_t* buf);
25  void fifo_buf_destroy_mmap(fifo_buf_t* buf);
26  int fifo_buf_enqueue(fifo_buf_t* buf, LiftRequest request);
27  int fifo_buf_dequeue(fifo_buf_t* buf, LiftRequest* request);
28
29  #endif
```

../fifo_buf.h

3 Implementation using threads and mutex lock

This part of the report outlines how thread synchronisation between the consumer and producer processes was achieved using the pthreads libraries.

3.1 Creating threads

Threads are created using **pthread_create()**. A total of 4 threads were created during the simulation, the first is the producer thread targeting the **request()** function, the other three threads are the consumer threads targeting the **lift()** function. Joining a thread means to block the waiting thread for the joining thread to finish, this was done in the main thread so that the main thread is blocked until all the producer/consumer threads finished their job.

```
1   int main(int argc, char **argv){
2
3       int m, i;
4       pthread_t tLiftRequest;
5       pthread_t tLift[3];
6
7
8       if (argc != 3){
9           printf("%s m t\n\n", argv[0]);
10          printf("\tm = buffer size\n");
11          printf("\tt = time per request in milliseconds\n");
12
13          exit(EXIT_FAILURE);
14
15      }
16
17      m = atoi(argv[1]);
18      t = atoi(argv[2]);
19
20      if (m <= 0 || t < 0){
```

```
21          fprintf(stderr, "Invalid arguments\n");
22
23          exit(EXIT_FAILURE);
24      }
25
26      fout = fopen(OUTPUT_FILE_PATH, "w");
27      fin = fopen(INPUT_FILE_PATH, "r");
28      buffer = fifo_buf_init_malloc(m);
29
30      /* Create threads */
31      pthread_create(&tLiftRequest, NULL, request, NULL);
32      for (i = 0; i < 3; i ++){
33          pthread_create(&tLift[i], NULL, lift, &i);
34      }
35
36
37      /* Join thread (Essentially wait for all thread to finish work)*/
38      pthread_join(tLiftRequest, NULL);
39      for (i =0; i< 3; i ++){
40          pthread_join(tLift[i], NULL);
41      }
42
43      /* Clean up  */
44
45      fifo_buf_destroy_malloc(buffer);
46
47      if (fclose(fout) != 0){
48          perror("Error closing the file\n");
49      }
50
51      if (fclose(fin) != 0){
52          perror("Error closing input file");
53      }
54
55
56      return 0;
57  }
```

../lift_sim_A.c

3.2 Producer thread

The producer process generates lift request by reading in a file pointer, and places the data into the fixed-size buffer. When placing the data into the buffer, the producer must ensure that two condition is met:

1. Mutual exclusion. Only one thread can modify the buffer at any instant.
2. The buffer enough space for the input.

```
1  void* request(void* args){
2
3      LiftRequest request;
4      char line[512];
5      int nRequest = 1;
6
7
```

```
8       /* Read all the request from simulation file */
9       while (fgets(line, 512, fin) != NULL){ /* Scanning until EOF */
10
11          int src, dst;
12
13          if( sscanf(line, "%d %d", &src, &dst) != 2 || src < 1 || dst < 1 ){
14              fprintf(stderr, "Encountered an invalid lift request.\n");
15              continue;
16          }
17
18          request.src = src;
19          request.dst = dst;
20
21          /* acquire the lock */
22          pthread_mutex_lock(&lock);
23
24          /* attempts to add to buffer. Wait for space upon failure. */
25          while (fifo_buf_enqueue(buffer, request) == FIFO_BUF_ERR_FULLENQ){;
26              pthread_cond_wait(&hasSpace, &lock);
27          }
28
29          fprintf(fout, "-------------------------------------------\n");
30          fprintf(fout, "New Lift Request From Floor %d to Floor %d\n"\
31                        "Request No: %d\n", request.src, request.dst, nRequest);
32          fprintf(fout, "-------------------------------------------\n");
33
34          /* Signal waiting threads and release the lock */
35          pthread_cond_signal(&nonEmpty);
36          pthread_mutex_unlock(&lock);
37
38          /* Non-Critical Operations */
39          nRequest ++;
40      }
41
42      /* Signal other threads that there's no item left */
43      request.src = -1;
44      request.dst = -1;
45      pthread_mutex_lock(&lock);
46      while (fifo_buf_enqueue(buffer, request) == FIFO_BUF_ERR_FULLENQ){
47          pthread_cond_wait(&hasSpace, &lock);
48      }
49      pthread_cond_signal(&nonEmpty);
50      pthread_mutex_unlock(&lock);
51
52
53
54      return 0;
55  }
```

../lift_sim_A.c

Mutual exclusion was achieved using mutex locking. A mutex lock is synchronisation mechanism designed to enfource mutal exclusion. Before entering the critical section (line 22) the thread aquires a mutex locks, if it's able to aquire the lock then the code continues otherwise it will be blocked until the lock becomes available. The lock is released once the code exits the critical section (line 50), allowing another thread to enter its critical section.

If the producer attempts to add to a buffer that's already full it will wait until the buffer has space.

However, it must temporarily release the mutex lock so that the consumers can clear the buffer. The mechanism that enables this is **pthread_cond_wait()**, the function releases the mutex and returns on a specific signal - in this case, the consumer thread will signals it successfully remove an item from the buffer.

Once the producer thread finishes, it places a special request into the buffer to indicate to the consumer threads to not wait for new lift request.

3.3 Consumer thread

The consumer process takes each lift request and simulates the lift. Similar to the producer process, two condition must be met when it performs this operation.

1. Mutual exclusion.
2. Theres item in the buffer to consume.

```
1  void* lift(void* args){
2
3      LiftRequest request;
4      int id = *((int*) args);
5      int position = 1;
6      int nMove = 0;
7      int totalMove = 0;
8      int nRequest = 0;
9
10
11     for(;;){
12         /* Acquire the lock */
13         pthread_mutex_lock(&lock);
14
15         /* Block until theres item in the buffer */
16         while (fifo_buf_dequeue(buffer, &request) == FIFO_BUF_ERR_EMPTYDEQ){
17             pthread_cond_wait(&nonEmpty, &lock);
18         }
19
20         /* If this occurs then the producer has no more item to produce */
21         if (request.src == -1 && request.dst == -1){
22             fifo_buf_enqueue(buffer, request);
23             pthread_cond_signal(&nonEmpty);
24             pthread_mutex_unlock(&lock);
25             break;
26         }
27
28         nRequest += 1;
29         nMove = abs(position - request.src) + abs(request.src - request.dst);
30         totalMove += nMove;
31
32         /* Log Lift request */
33         fprintf(fout, "----------------------------------------------\n");
34         fprintf(fout, "Lift-%d Operation\n"\
35                       "Previous position: Floor %d\n"\
36                       "Request: Floor %d to Floor %d\n"\
37                       "Detail operations:\n"\
38                       "     Go from Floor %d to Floor %d\n"\
39                       "     Go from floor %d to Floor %d\n"\
40                       "     #movement for this request: %d\n"\
```

```
41                          "     Total #movement: %d\n"\
42                          "     Current position: Floor %d\n",
43          id, position, request.src, request.dst, position, request.src,
44          request.src, request.dst, nMove, totalMove, request.dst);
45          fprintf(fout, "----------------------------------------\n");
46
47          position = request.dst;
48
49          /* Unlock */
50          pthread_cond_signal(&hasSpace);
51          pthread_mutex_unlock(&lock);
52
53          /* Non-critical Operations */
54          usleep(t * 1000);
55      }
56
57
58      return 0;
59  }
```

../lift_sim_A.c

Mutual exclusion was achieved using the same locking mechanismed discussed in section 3.2. When the consumer attempts to remove from an empty buffer, it will wait until the buffer has an request using the same mechanism as discussed in section 3.2.

4 Implementation using processes and semaphores

The second part of the assignment implements the simulation using processes and semaphores through the POXIS library. Since processes do not share memory, the parent process needs to create shared memory.

4.1 Creating processes

To create child processes, **fork()** was used, the main process forked() 4 times to create 4 child processes - one for the producer and three for the consumers (see line 42 - 56 below). The IF statements prevents child processes from creating their own processes, thus ensuring correct creation of processes.

Three semaphores were created:

1. Full. Initialised to 0.
2. Empty. Initialised to the size of the buffer.
3. Mutex. Initialised to 1.

These semaphores will be later used by the producer and consumer processes to ensure mutual exclusion and blocking of the consumer when the buffer is empty, and producer when its full. The function **waitpid()** to perform blocking of the main process until the child processes finished.

```
1  int main(int argc, char **argv){
2
3      int m, i;
```

```
4       int status = 0;
5       pid_t pid[N_LIFT + 1];
6
7
8       if (argc != 3){
9           printf("%s m t\n\n", argv[0]);
10          printf("\tm = buffer size\n");
11          printf("\tt = time per request in milliseconds\n");
12          exit(EXIT_FAILURE);
13      }
14
15      /* parse input */
16      m = atoi(argv[1]);
17      t = atoi(argv[2]);
18
19      if (m <= 0 || t < 0){
20          fprintf(stderr, "Invalid arguments\n");
21          exit(EXIT_FAILURE);
22      }
23
24
25      /* Open sim out file */
26      if((fd_out = open(OUTPUT_FILE_PATH, O_CREAT | O_RDWR | O_TRUNC | O_APPEND, 0644)) <
            0){
27          perror("error openning file");
28          exit(EXIT_FAILURE);
29      }
30
31      /* Open sim in file */
32      fin = fopen(INPUT_FILE_PATH, "r");
33
34      /* Create shared buffer */
35      buffer = fifo_buf_init_mmap(m);
36
37      /* initialise sem */
38      init_sem(&full, 0);
39      init_sem(&empty, m);
40      init_sem(&mutex, 1);
41
42      for( i = 0; i < N_LIFT + 1; i++){
43          pid[i] = fork();
44          if (pid[i] < 0){
45              perror("fork failed");
46              exit(EXIT_FAILURE);
47          }
48          else if (pid[i] == 0) {
49              if (i == 0){
50                  request(NULL);
51              }
52              else{
53                  lift(&i);
54              }
55              exit(EXIT_SUCCESS); /*prevents child from creating another child */
56          }
57      }
58
59      /* join processes */
60      for ( i = 0; i < N_LIFT + 1; i++){
61          waitpid(pid[i], &status, 0);
```

```
62      }
63
64      /* remove shared memory */
65      fifo_buf_destroy_mmap(buffer);
66
67      /* remove semaphores */
68      munmap(mutex, sizeof(sem_t));
69      munmap(full, sizeof(sem_t));
70      munmap(empty, sizeof(sem_t));
71
72      /* close output file */
73      if (close(fd_out) < 0){
74          perror("error closing file");
75      }
76
77      /* Closing the input file */
78      if (fclose(fin) != 0){
79          perror("Error closing input file");
80      }
81
82      return status;
83  }
```

../lift_sim_B.c

4.2 Producer process

To achieve mutual exclusion within the critical section, the mutex semaphore was used, and to ensure that the producer process wait for the buffer to have enough space, the empty semaphore was used. When the producer process finshes, it appends an EOF request to signal the consumer processes not to wait for any new requests.

```
1   void* request(void* arg){
2
3       LiftRequest request;
4       char line[512];
5       int nRequest = 1;
6
7
8       /* Read all the request from simulation file */
9       while (fgets(line, 512, fin) != NULL){ /* Scanning until EOF */
10
11          int src, dst;
12          if( sscanf(line, "%d %d", &src, &dst) != 2 || src < 1 || dst < 1 ){
13              fprintf(stderr, "Encountered an invalid lift request, choosing to ignore\n"
                    );
14              continue;
15          }
16
17          /* Critical section */
18          sem_wait(empty);
19          sem_wait(mutex); /* for a 1-producer v. N-consumer, you do not need mutex */
20
21          request.src = src;
22          request.dst = dst;
23          fifo_buf_enqueue(buffer, request);
```

```
24
25          /* Write to file */
26          sprintf(line,                "-----------------------------------------\n");
27          sprintf(line + strlen(line), "New Lift Request From Floor %d to Floor %d\n"\
28                                       "Request No: %d\n", request.src, request.dst,
                                             nRequest);
29          sprintf(line + strlen(line), "-----------------------------------------\n");
30
31          /* write to fd_out */
32          lseek(fd_out, 0, SEEK_END);
33          write(fd_out, line, strlen(line));
34
35          sem_post(mutex);
36          sem_post(full);
37
38          /* Non-Critical Operations */
39          nRequest ++;
40      }
41
42      /* Append an EOF to the buffer to signify that the request process is finished */
43      sem_wait(empty);
44      sem_wait(mutex);
45      request.src = EOF;
46      request.dst = EOF;
47      fifo_buf_enqueue(buffer, request);
48      sem_post(mutex);
49      sem_post(full);
50
51      return 0;
52  }
```

../lift_sim_B.c

It is important to call **sem_wait(empty)** before **sem_wait(mutex)** in order to avoid **deadlock**, and it is the same reason why we call **sem_post(mutex)** before calling **sem_post(full)**.

4.3 Consumer process

To achive mutual exclusion wthin the critical sectin, the mutex semaphore was used, and to ensure that the consumer wait for the buffer to be non-empty, the full semaphore was used. When the consumer thread receive a request indicating EOF, it put that request back into the buffer exits the loop to finish.

```
1   void* lift(void* args){
2
3       LiftRequest request;
4       int id = *((int*) args);
5       int position = 1;
6       int totalMove = 0;
7       int nMove = 0;
8       int nRequest = 0;
9       char line[512];
10
11
12      for(;;){
13
```

```
14          /* Critical section */
15          sem_wait(full);
16          sem_wait(mutex);
17
18          fifo_buf_dequeue(buffer, &request);
19
20          /* if this request siginifes EOF then exits */
21          if (request.src == EOF && request.dst == EOF){
22              fifo_buf_enqueue(buffer, request);
23              sem_post(mutex);
24              sem_post(full);
25              break;
26          }
27
28          nRequest += 1;
29          nMove = abs(position - request.src) + abs(request.src - request.dst);
30          totalMove += nMove;
31
32          /* Log lift request */
33          sprintf(line,                   "-------------------------------------------\n")
                ;
34          sprintf(line + strlen(line),    "Lift-%d Operation\n"\
35                                          "Previous position: Floor %d\n"\
36                                          "Request: Floor %d to Floor %d\n"\
37                                          "Detail operations:\n"\
38                                          "     Go from Floor %d to Floor %d\n"\
39                                          "     Go from floor %d to Floor %d\n"\
40                                          "     #movement for this request: %d\n"\
41                                          "     Total #movement: %d\n"\
42                                          "     Current position: Floor %d\n",
43          id, position, request.src, request.dst, position, request.src,
44          request.src, request.dst, nMove, totalMove, request.dst);
45          sprintf(line + strlen(line) , "-------------------------------------------\n")
                ;
46
47          /* Write to end of the file */
48          lseek(fd_out, 0, SEEK_END);
49          write(fd_out, line, strlen(line));
50
51          position = request.dst; /* Updates the position */
52
53          sem_post(mutex);
54          sem_post(empty);
55
56          /* Non-critical section */
57          usleep(t * 1000);
58      }
59
60      return 0;
61  }
```

../lift_sim_B.c

Similar to the producer process, the order in which we call sem_wait() and sem_post() is important in order to prevent deadlock.