

# Homework 3 - Operating Systems (ICS431)

Alfaifi, Ammar - 201855360

## 1 Modified Fibonccci

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define MAX_SEQUENCE 100
6 #define NUM_THREADS 2
7
8 int fib_sequence[MAX_SEQUENCE];
9 int last_idx = -1; // initialize last index to -1
10
11 pthread_mutex_t mutex;
12
13 void *fibonacci(void *arg) {
14     int n = *((int *)arg);
15     fib_sequence[0] = 0;
16     fib_sequence[1] = 1;
17
18     for (int i = 2; i < n; i++) {
19         pthread_mutex_lock(&mutex);
20         fib_sequence[i] = fib_sequence[i - 1] + fib_sequence[i - 2];
21         last_idx = i; // update last index to i
22         pthread_mutex_unlock(&mutex);
23     }
24     pthread_exit(NULL);
25 }
26
27 int main(int argc, char *argv[]) {
28     pthread_t threads[NUM_THREADS];
29
30     if (argc != 2) {
31         fprintf(stderr, "Usage: %s <Number of fib elements>\n", argv[0]);
32         exit(-1);
33     }
34     int n = atoi(argv[1]);
35
36     pthread_mutex_init(&mutex, NULL);
37
38     int rc = pthread_create(&threads[0], NULL, fibonacci, (void *)&n);
39     if (rc) {
40         printf("ERROR; return code from pthread_create() is %d\n", rc);
41         exit(-1);
42     }
43
44     while (last_idx < n - 1) { // loop until all values are updated by the child
45         pthread_mutex_lock(&mutex);
46         int idx = last_idx;
47         pthread_mutex_unlock(&mutex);
48
49         if (idx >= 0) { // print new values if there are any
50             for (int i = 0; i <= idx; i++) {
51                 printf("%d ", fib_sequence[i]);
52             }
53             printf("\n");
54         }
55     }
56
57     pthread_join(threads[0], NULL);
58
59     pthread_mutex_destroy(&mutex);
60     pthread_exit(NULL);
61 }
```

## 2 Semphaore

A list of needed variables as semaphores:

1. `patient_sem` - initialized to 1, used to ensure only one patient writes to the buffer at a time
2. `doctor_sem` - initialized to 0, used to ensure doctor waits until there is new patient information in the buffer
3. `treat_sem` - initialized to 0, used to ensure the patient waits until the doctor has written the treatment details to the buffer

A pseudocode implementation can be as following:

```
semaphore patient_sem = 1;
semaphore doctor_sem = 0;
semaphore treat_sem = 0;

void patient_process() {
    // wait in waiting room until doctor is free
    wait(doctor_sem);

    // enter doctor's office and consult doctor
    wait(patient_sem);
    consultDoctor();
    signal(patient_sem);

    // wait for doctor to treat and update buffer
    signal(doctor_sem);
    wait(treat_sem);

    // note treatment and leave doctor's office
    noteTreatment();
}

void doctor_process() {
    while (true) {
        // wait for patient to arrive and update buffer
        wait(patient_sem);
        treatPatient();
        signal(treat_sem);

        // signal patient that treatment details are in buffer
        signal(doctor_sem);
    }
}
```

A real implementation in C will be as

```
1  #include <pthread.h>
2  #include <semaphore.h>
3
4  sem_t patient_sem, doctor_sem, treat_sem;
5
6  void* patient_process(void* arg) {
7      // wait in waiting room until doctor is free
8      sem_wait(&doctor_sem);
9
10     // enter doctor's office and consult doctor
11     sem_wait(&patient_sem);
12     consultDoctor();
13     sem_post(&patient_sem);
14
15     // wait for doctor to treat and update buffer
16     sem_post(&doctor_sem);
17     sem_wait(&treat_sem);
18
19     // note treatment and leave doctor's office
20     noteTreatment();
21     pthread_exit(NULL);
22 }
23
24 void* doctor_process(void* arg) {
25     while (1) {
26         // wait for patient to arrive and update buffer
27         sem_wait(&patient_sem);
28         treatPatient();
29         sem_post(&treat_sem);
30
31         // signal patient that treatment details are in buffer
32         sem_post(&doctor_sem);
33     }
34 }
35
36 int main() {
37     // initialize semaphores
38     sem_init(&patient_sem, 0, 1);
39     sem_init(&doctor_sem, 0, 0);
40     sem_init(&treat_sem, 0, 0);
41
42     // create threads for patient and doctor processes
43     pthread_t patient_thread, doctor_thread;
44     pthread_create(&patient_thread, NULL, patient_process, NULL);
45     pthread_create(&doctor_thread, NULL, doctor_process, NULL);
46
47     // wait for threads to finish
48     pthread_join(patient_thread, NULL);
49     pthread_join(doctor_thread, NULL);
50
51     // destroy semaphores
52     sem_destroy(&patient_sem);
53     sem_destroy(&doctor_sem);
54     sem_destroy(&treat_sem);
55
56     return 0;
57 }
```

### 3 Race Conditions Problem

Here are the solution of each part:

1. The variable is `available_resources`.
2. The race condition occurs in both the `decrease_count()` and `increase_count()` functions.
3. To fix the race condition, a semaphore or mutex lock can be used to ensure mutual exclusion when accessing the `available_resources` variable. One possible implementation is as follows:

```
1  #define MAX_RESOURCES 5
2  int available_resources = MAX_RESOURCES; // Critical section
3  sem_t mutex;
4
5  int decrease_count(int count) {
6      sem_wait(&mutex);
7      if (available_resources < count) {
8          sem_post(&mutex);
9          return -1;
10     } else {
11         available_resources -= count;
12         sem_post(&mutex);
13         return 0;
14     }
15 }
16
17 void increase_count(int count) {
18     sem_wait(&mutex);
19     available_resources += count;
20     sem_post(&mutex);
21 }
```