Homework 3 - Operating Systems (ICS431)

Alfaifi, Ammar - 201855360

1 Modified Fibonacci

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

#define MAX_SEQUENCE 100
#define NUM_THREADS 2

int fib_sequence[MAX_SEQUENCE];
int last_idx = -1; // initialize last index to -1

pthread_mutex_t mutex;

void *fibonacci(void *arg) {
  int n = *((int *)arg);
  fib_sequence[0] = 0;
  fib_sequence[1] = 1;

  for (int i = 2; i < n; i++) {
    pthread_mutex_lock(&mutex);
    fib_sequence[i] = fib_sequence[i - 1] + fib_sequence[i - 2];
    last_idx = i; // update last index to i
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <Number of fib elements>\n", argv[0]);
    exit(-1);
  }
  int n = atoi(argv[1]);

  pthread_mutex_init(&mutex, NULL);

  int rc = pthread_create(&threads[0], NULL, fibonacci, (void *)&n);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
  }

  while (last_idx < n - 1) { // loop until all values are updated by the child
    pthread_mutex_lock(&mutex);
    int idx = last_idx;
    pthread_mutex_unlock(&mutex);

    if (idx >= 0) { // print new values if there are any
      for (int i = 0; i <= idx; i++) {
        printf("%d ", fib_sequence[i]);
      }
      printf("\n");
    }
  }

  pthread_join(threads[0], NULL);

  pthread_mutex_destroy(&mutex);
  pthread_exit(NULL);
}
```

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

```c
#include <pthread.h>
#include <semaphore.h>

sem_t patient_sem, doctor_sem, treat_sem;

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

    // enter doctor's office and consult doctor
    sem_wait(&patient_sem);
    consultDoctor();
    sem_post(&patient_sem);

    // wait for doctor to treat and update buffer
    sem_post(&doctor_sem);
    sem_wait(&treat_sem);

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

void* doctor_process(void* arg) {
    while (1) {
        // wait for patient to arrive and update buffer
        sem_wait(&patient_sem);
        treatPatient();
        sem_post(&treat_sem);

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

int main() {
    // initialize semaphores
    sem_init(&patient_sem, 0, 1);
    sem_init(&doctor_sem, 0, 0);
    sem_init(&treat_sem, 0, 0);

    // create threads for patient and doctor processes
    pthread_t patient_thread, doctor_thread;
    pthread_create(&patient_thread, NULL, patient_process, NULL);
    pthread_create(&doctor_thread, NULL, doctor_process, NULL);

    // wait for threads to finish
    pthread_join(patient_thread, NULL);
    pthread_join(doctor_thread, NULL);

    // destroy semaphores
    sem_destroy(&patient_sem);
    sem_destroy(&doctor_sem);
    sem_destroy(&treat_sem);

    return 0;
}
```

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:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES; // Critical section
sem_t mutex;

int decrease_count(int count) {
    sem_wait(&mutex);
    if (available_resources < count) {
        sem_post(&mutex);
        return -1;
    } else {
        available_resources -= count;
        sem_post(&mutex);
        return 0;
    }
}

void increase_count(int count) {
    sem_wait(&mutex);
    available_resources += count;
    sem_post(&mutex);
}
```

4 2-Thread Matrix

The implementation of two matrices of size 40x40 with random values of 0 or 1 with two separate threads is as following:

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

#define ROWS 40
#define COLS 40

int matrixA[ROWS][COLS];
int matrixB[ROWS][COLS];
int sumMatrix[ROWS][COLS];
int diffMatrix[ROWS][COLS];

void *computeSum(void *arg) {
  for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
      sumMatrix[i][j] = matrixA[i][j] + matrixB[i][j];
    }
  }
  pthread_exit(NULL);
}

void *computeDiff(void *arg) {
  for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
      diffMatrix[i][j] = matrixA[i][j] - matrixB[i][j];
    }
  }
  pthread_exit(NULL);
}

void printMatrix(int matrix[ROWS][COLS]) {
  for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
      printf("%d ", matrix[i][j]);
    }
    printf("\n");
  }
}

int main() {
  pthread_t threadSum, threadDiff;

  // Initialize matrices with binary inputs
  for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
      matrixA[i][j] = rand() % 2;
      matrixB[i][j] = rand() % 2;
    }
  }

  // Create threads for computing sum and difference
  pthread_create(&threadSum, NULL, computeSum, NULL);
  pthread_create(&threadDiff, NULL, computeDiff, NULL);

  // Wait for threads to complete
  pthread_join(threadSum, NULL);
  pthread_join(threadDiff, NULL);

  printf("\nSum Matrix:\n");
  printMatrix(sumMatrix);

  printf("\nDifference Matrix:\n");
  printMatrix(diffMatrix);

  return 0;
}
```

Theoretical Part

1 Scheduling Policies

(a) Using FCFS scheduling algorithm and completing the table, we have the following:

Process	Arrival T.	Burst T.	Turnaround T.
P1	0.0	8	8
P2	0.4	4	11.6
P3	1.0	1	12

Then the average turn around time can be found as

$$\bar{T_t} = \frac{8 + 11.6 + 12}{3} = 10.5$$

Where $\bar{T_t}$ is the average turnaround time.

(b) Doing the same but with SJF algorithm we get

Process	Arrival T.	Burst T.	Turnaround T.
P1	0.0	8	8
P3	1.0	1	8
P2	0.4	4	12.6

So we have

$$\bar{T_t} = \frac{2(8) + 12.6}{3} = 9.5$$

(c) After waiting for 1 time unit, we get the following

Process	Arrival T.	Burst T.	Turnaround T.
P3	1.0	1	1
P2	0.4	4	4.6
P1	0.0	8	13

And we get

$$\bar{T_t} = \frac{1 + 4.6 + 13}{3} = 6.2$$

2 Dining-Philosopher' Problem

One solution is if there is only one chopstick left and no other philosopher has two chopsticks, do not grant the request of a philosopher asking for the first chopstick.

3 Banker's Algorithm

(a) The need matrix will be, Need$[i][j]$ = Max$[i][j]$ − Allocation$[i][j]$,

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix} \tag{1}$$

(b) Yes.

(c) Yes, there's enough available resource instances.

4 Violation of Mutual Exclusion

executing atomically means that a set of operations is performed as a single, indivisible unit, such that it appears to the outside world as if it were a single operation. In other words, atomicity guarantees that all the steps of the operation are executed as a single, uninterrupted action, and any other process or thread executing on the same resource cannot interfere with it or observe it in a partially completed state.

Assuming that there are two processes A and B with a shared resource and using semaphores to enforce mutual exclusion, let's consider the following scenario:

1. Process A executes the 'wait()' operation on the semaphore, which decreases the value of the semaphore by 1.
2. Before process A executes the 'signal()' operation, the processor switches to process B.
3. Process B executes the 'wait()' operation on the semaphore, which also decreases its value by 1.
4. Now the semaphore value is 0, which means that no process can access the shared resource.
5. The processor switches back to process A, which now executes the 'signal()' operation, increasing the semaphore value by 1.
6. The semaphore value is now 1, but process B has not yet executed the 'signal()' operation to release the semaphore.
7. Process A can now access the shared resource, violating mutual exclusion.

This scenario shows that if the 'wait()' and 'signal()' semaphore operations are not executed atomically, i.e., without interruption from other processes or threads, then mutual exclusion may be violated. To prevent this, the 'wait()' and 'signal()' operations must be executed atomically, either by the operating system or by using special hardware instructions.