CENG328 Operating Systems
Laboratory Chapter 5

1 Process Synchronization

When working on shared resources with multiple threads, execution of these threads must be controlled/synchronized in order to prevent errors in program execution due to uncontrolled access to these resources. In order to get a better understanding of this concept, compile and try executing the program below (sync1.c) first:

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

int sum = 0;

void* addFun(void* arg) {
  int i;
  for (i = 0; i < 1000000; ++i)
    sum += 1;
  return NULL;
}

int main() {
  int i;
  pthread_t threads[10];

  for (i = 0; i < 10; ++i)
    pthread_create(&threads[i], NULL, addFun, NULL);

  for (i = 0; i < 10; ++i)
    pthread_join(threads[i], NULL);

  printf("Result is %d.\n", sum);
  return 0;
}
```

As you can see, this is a program which attempts to count up to 10.000.000. This is achieved by creating 10 threads and letting each thread to increase value of a shared variable one by one 1 million times. But whenever this program gets executed, it always generates different incorrect results:

```
Result is 2276886.
Result is 3940960.
Result is 3140907.
Result is 1860179.
Result is 2291528.
...
```

This happens because the variable that gets modified by each thread is not mutually exclusive; all threads may read and modify value of this variable arbitrarily. In order to fix this error, a "mutual exclusion" mechanism must be introduced into the code (sync2.c):

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

int sum = 0;
sem_t mutex;

void* addFun(void* arg) {
  int i;
  for (i = 0; i < 1000000; ++i) {
    sem_wait(&mutex); // lock critical region
    sum += 1;
    sem_post(&mutex); // unlock critical region
  }
  return NULL;
}

int main() {
  int i;
  pthread_t threads[10];
  sem_init(&mutex, 0, 1); // semaphore initialization

  for (i = 0; i < 10; ++i)
    pthread_create(&threads[i], NULL, addFun, NULL);

  for (i = 0; i < 10; ++i)
    pthread_join(threads[i], NULL);

  printf("Result is %d\n", sum);
  sem_destroy(&mutex); // destroy semaphore
  return 0;
}
```

"sem_t" is a special type of variable (semaphore) that may only hold the values 0 or above. **sem_init** is the function which sets the initial value of a given semaphore. **sem_post** is the function which increases value of a semaphore by one and **sem_wait** is the function that decreases value of a semaphore by one. **sem_destroy** destroys a given semaphore.

Note that sem_post may also be referred as **signal** in the lecture and the textbook.

If value of a semaphore is already 0 and sem_wait is used on this semaphore, its value can no longer be decreased and the running thread **gets blocked until** an another process or thread increases the value of this semaphore.

By setting initial value of the mutex variable to 1, we have effectively designed a critical region in this code which may be accessed by only one thread at a given time. It is still unknown which thread gets the privilege by the CPU to execute its sem_wait function but when one thread gets to execute it, all other nine threads will wait until a sem_post is issued and value of the semaphore is restored to 1. So, no matter how many times you execute the second program, you will always get the correct result now.

2 Exercises

1. Read man pages for the following library functions: sem_init, sem_post, sem_wait, sem_destroy.
2. Open the second question in Labwork 5. Solve it using only one global variable for the result with the help of a semaphore now.