Class Assignment - 8

Name - Rajeev Kumar
ID - 12341700

- ## Q1 :

1. Run the reference code and record the output

```
(base) → code gcc semaphore_sync.c -lpthread -o sem
(base) → code ./sem
Hello from A
Hello from B
```

2. Modify it by removing semaphore logic and observe the output by running it multiple times

Code :

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
void* threadA(void* arg)
{
 printf("Hello from A\n");
 return NULL;
}
void* threadB(void* arg)
{
 printf("Hello from B\n");
 return NULL;
}
int main()
{
 pthread_t tA, tB;
 pthread_create(&tA, NULL, threadA, NULL);
 pthread_create(&tB, NULL, threadB, NULL);
  pthread_join(tA, NULL);
 pthread_join(tB, NULL);
  return 0;
}
```

OUTPUT :

```
(base) ➜  code gcc q1.c -lpthread -o q1
(base) ➜  code ./q1
Hello from A
Hello from B
(base) ➜  code ./q1
Hello from A
Hello from B
(base) ➜  code ./q1
Hello from B
Hello from A
```

3. Explain why the order changes without synchronization.

ANS : Both threads are free to execute independently as soon as they are created.

4. Add one more thread "threadC" which should execute only after threadB has done its execution. Note You will need one more sem_t variable

Code :

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
sem_t semB, semC;
void* threadA(void* arg)
{
 printf("Hello from A\n");
 sem_post(&semB);
 return NULL;
}
void* threadB(void* arg)
{
 sem_wait(&semB);
 printf("Hello from B\n");
```

```c
 sem_post(&semC);
 return NULL;
}
void* threadC(void* arg)
{
 sem_wait(&semC);
 printf("Hello from C\n");
 return NULL;
}
int main()
{
 pthread_t tA, tB, tC;
  sem_init(&semB, 0, 0);
 sem_init(&semC, 0, 0);
  pthread_create(&tA, NULL, threadA, NULL);
 pthread_create(&tB, NULL, threadB, NULL);
 pthread_create(&tC, NULL, threadC, NULL);
 pthread_join(tA, NULL);
 pthread_join(tB, NULL);
 pthread_join(tC, NULL);
  sem_destroy(&semB);
 sem_destroy(&semC);
 return 0;
}
```

OUTPUT :

```
(base) ➜  code gcc q1b.c -lpthread -o q1b
(base) ➜  code ./q1b
Hello from A
Hello from B
Hello from C
```

- **Q2 :**

Code :

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
sem_t aArrived, bArrived;
void* threadA(void* arg)
{
 printf("A1 (Starting task)\n");
 sem_post(&aArrived);
 sem_wait(&bArrived);
 printf("A2 (Ending task)\n");
 return NULL;
}
void* threadB(void* arg)
{
 printf("B1 (Starting task)\n");
 sem_post(&bArrived);
 sem_wait(&aArrived);
 printf("B2 (Ending task)\n");
 return NULL;
}
int main()
{
 pthread_t tA, tB;
   sem_init(&aArrived, 0, 0);
   sem_init(&bArrived, 0, 0);
   pthread_create(&tA, NULL, threadA, NULL);
   pthread_create(&tB, NULL, threadB, NULL);
   pthread_join(tA, NULL);
   pthread_join(tB, NULL);
   sem_destroy(&aArrived);
   sem_destroy(&bArrived);
   return 0;
}
```

OUTPUT :

```
(base) ➜  code ./q2
A1 (Starting task)
B1 (Starting task)
B2 (Ending task)
A2 (Ending task)
(base) ➜  code ./q2
B1 (Starting task)
A1 (Starting task)
A2 (Ending task)
B2 (Ending task)
```

- ## Q3 :

Code :

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

long counter = 0;
pthread_mutex_t lock;
void* threadFuncNoMutex(void* arg) {
    for (long i = 0; i < 100000; i++) {
        counter++;
    }
    return NULL;
}
void* threadFuncWithMutex(void* arg) {
    for (long i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
int main() {
    pthread_t threads[10];
    counter = 0;
    for (int i = 0; i < 10; i++)
        pthread_create(&threads[i], NULL, threadFuncNoMutex, NULL);
```

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

    printf("Final counter value (no mutex): %ld (expected %ld)\n", counter,
(long)10 * (long)100000);

    counter = 0;
    pthread_mutex_init(&lock, NULL);

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

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

    printf("Final counter value (with mutex): %ld (expected %ld)\n",
counter, (long)10 * (long)100000);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

OUTPUT :

```
(base) → code gcc mutex_demo.c -lpthread -o mutex
(base) → code gcc q3.c -lpthread -o q3
(base) → code ./q3
Final counter value (no mutex): 225257 (expected 1000000)
Final counter value (with mutex): 1000000 (expected 1000000)
(base) → code ./q3
Final counter value (no mutex): 130261 (expected 1000000)
Final counter value (with mutex): 1000000 (expected 1000000)
(base) → code ./q3
Final counter value (no mutex): 231097 (expected 1000000)
Final counter value (with mutex): 1000000 (expected 1000000)
```

EXPLANATION :

Ten threads increment a shared counter. Without a mutex, simultaneous access causes race conditions and incorrect counts. With a mutex, threads access the counter one at a time, ensuring the final count is correct.

- **Q4 :**

Code :

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


#define NUM_THREADS 8
#define MAX_CONCURRENT 3
sem_t multiplex;


void* worker(void* arg)
{
 int id = *(int*)arg;
 // Request to enter
 sem_wait(&multiplex); // take one token
 printf("Thread %d ENTERED critical section.\n", id);
 // Critical section (simulate some work)
 sleep(1); // simulate processing
 printf("Thread %d LEAVING critical section.\n", id);
 sem_post(&multiplex); // release token
 return NULL;
}


int main()
{
 pthread_t threads[NUM_THREADS];
 int ids[NUM_THREADS];
  // Initialize semaphore to MAX_CONCURRENT
 sem_init(&multiplex, 0, MAX_CONCURRENT);
  // Create threads
 for (int i = 0; i < NUM_THREADS; i++)
 {
   ids[i] = i + 1;
   pthread_create(&threads[i], NULL, worker, &ids[i]);
 }
  // Join threads
```

```c
for (int i = 0; i < NUM_THREADS; i++)
{
  pthread_join(threads[i], NULL);
}
 sem_destroy(&multiplex);
printf("All threads finished.\n");
return 0;
}
```

OUTPUT :

```
(base) ➜  code gcc multiplex_semaphore.c -lpthread -o multiplex
(base) ➜  code ./multiplex
Thread 2 ENTERED critical section.
Thread 1 ENTERED critical section.
Thread 3 ENTERED critical section.
Thread 2 LEAVING critical section.
Thread 1 LEAVING critical section.
Thread 4 ENTERED critical section.
Thread 5 ENTERED critical section.
Thread 3 LEAVING critical section.
Thread 6 ENTERED critical section.
Thread 6 LEAVING critical section.
Thread 4 LEAVING critical section.
Thread 5 LEAVING critical section.
Thread 7 ENTERED critical section.
Thread 8 ENTERED critical section.
Thread 8 LEAVING critical section.
Thread 7 LEAVING critical section.
All threads finished.
```

EXPLANATION :

This program uses a bounded semaphore to limit concurrent access. At most three threads can enter the critical section simultaneously. Each thread waits on the semaphore before entering and posts it after leaving, ensuring the maximum concurrency is never exceeded.

- **Q5 :**

1. **PART A : Naive Barrier Implementation**

Code :

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

#define N 5

int count = 0;
pthread_mutex_t mutex;
sem_t barrier;

void* naive_thread_func(void* arg)
{
   int id = *(int*)arg;
   printf("Thread %d reached the barrier\n", id);

   pthread_mutex_lock(&mutex);
   count++;
   if (count == N)
   {
       sem_post(&barrier);
   }
   pthread_mutex_unlock(&mutex);

   sem_wait(&barrier);
   printf("Thread %d passed the barrier\n", id);
   return NULL;
}

int main() {
   pthread_t threads[N];
   int ids[N];

   pthread_mutex_init(&mutex, NULL);
   sem_init(&barrier, 0, 0);
```

```
    for (int i = 0; i < N; i++) {
        ids[i] = i + 1;
        pthread_create(&threads[i], NULL, naive_thread_func,
&ids[i]);
    }
    for (int i = 0; i < N; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&barrier);
    return 0;
}
```

OUTPUT :

```
(base) ➜  code gcc q5a.c -lpthread -o q5a
(base) ➜  code ./q5a
Thread 1 reached the barrier
Thread 3 reached the barrier
Thread 2 reached the barrier
Thread 4 reached the barrier
Thread 5 reached the barrier
Thread 5 passed the barrier
```

EXPLANATION :

Part A may fail because only one thread signals the semaphore while all threads wait. As a result, threads that don’t receive the signal remain blocked, causing a deadlock at the barrier.

2. PART B : Reusable Barrier Solution

Code :

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

#define N 5
int count = 0;
   pthread_mutex_t mutex;
sem_t turnstile;
   sem_t turnstile2;

   void* thread_func(void* arg)
   {
      int id = *(int*)arg;
      printf("Thread %d reached the barrier\n", id);

      pthread_mutex_lock(&mutex);
      count++;
      if (count == N) {
          for (int i = 0; i < N; i++)
              sem_post(&turnstile);
      }
      pthread_mutex_unlock(&mutex);

      sem_wait(&turnstile);
      printf("Thread %d passed the barrier\n", id);

      pthread_mutex_lock(&mutex);
```

```
    count--;
    if (count == 0) {
        for (int i = 0; i < N; i++)
            sem_post(&turnstile2); // Reset barrier
    }
    pthread_mutex_unlock(&mutex);

    sem_wait(&turnstile2);
    return NULL;
}
int main() {
    pthread_t threads[N];
    int ids[N];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&turnstile, 0, 0);
    sem_init(&turnstile2, 0, 0);

    for (int i = 0; i < N; i++) {
        ids[i] = i + 1;
        pthread_create(&threads[i], NULL, thread_func, &ids[i]);
    }

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

    pthread_mutex_destroy(&mutex);
    sem_destroy(&turnstile);
    sem_destroy(&turnstile2);

    return 0;
}
```

OUTPUT :

```
(base) ➜  code gcc q5b.c -lpthread -o q5b
(base) ➜  code ./q5b
Thread 2 reached the barrier
Thread 5 reached the barrier
Thread 1 reached the barrier
Thread 3 reached the barrier
Thread 4 reached the barrier
Thread 4 passed the barrier
Thread 2 passed the barrier
Thread 3 passed the barrier
Thread 5 passed the barrier
Thread 1 passed the barrier
```

EXPLANATION :

This barrier ensures all threads wait until everyone arrives.

1. count tracks arrivals, protected by a mutex.
2. turnstile lets threads proceed together after all reach the barrier.
3. turnstile2 resets the barrier, making it reusable for future use.