CLASS ASSIGNMENT 9

Rajeev Kumar
12341700

- ## Q1 :

CODE :

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

#define BUFFER_SIZE 100

sem_t items;
pthread_mutex_t mutex;  // Mutex for mutual exclusion
int buffer[BUFFER_SIZE];
int fill_ptr = 0;
int use_ptr = 0;
int item_counter = 0;
int items_produced = 0;
int items_consumed = 0;

int produce_item() {
   return item_counter++;
}

void* producer(void* arg) {
   int id = *(int*)arg;
   for(int i = 0; i < 100000; i++) {
       int item = produce_item();
       pthread_mutex_lock(&mutex);  // Lock before accessing shared buffer
       buffer[fill_ptr] = item;
       fill_ptr = (fill_ptr + 1) % BUFFER_SIZE;
       items_produced++;
       printf("Producer %d produced: %d\n", id, item);
       pthread_mutex_unlock(&mutex);  // Unlock after accessing shared
buffer
       sem_post(&items);
```

```c
    }
    return NULL;
}

void* consumer(void* arg) {
    int id = *(int*)arg;
    for(int i = 0; i < 100000; i++) {
        sem_wait(&items);
        pthread_mutex_lock(&mutex);  // Lock before accessing shared buffer
        int item = buffer[use_ptr];
        use_ptr = (use_ptr + 1) % BUFFER_SIZE;
        items_consumed++;
        printf("Consumer %d consumed: %d\n", id, item);
        pthread_mutex_unlock(&mutex);  // Unlock after accessing shared
buffer

    }
    return NULL;
}

int main() {
    pthread_t prod_threads[2], cons_threads[2];
    int prod_ids[2] = {1, 2};
    int cons_ids[2] = {1, 2};

    sem_init(&items, 0, 0);
    pthread_mutex_init(&mutex, NULL);  // Initialize mutex

    printf("Starting Producer-Consumer (BUGGY VERSION)\n");

    for(int i = 0; i < 2; i++) {
        pthread_create(&prod_threads[i], NULL, producer, &prod_ids[i]);
        pthread_create(&cons_threads[i], NULL, consumer, &cons_ids[i]);
    }

    for(int i = 0; i < 2; i++) {
        pthread_join(prod_threads[i], NULL);
        pthread_join(cons_threads[i], NULL);
    }
```

```c
    sem_destroy(&items);
    pthread_mutex_destroy(&mutex);  // Destroy mutex

    printf("\n========== FINAL RESULTS ==========\n");
    printf("Total items produced: %d\n", items_produced);
    printf("Total items consumed: %d\n", items_consumed);
    printf("Final fill_ptr: %d\n", fill_ptr);
    printf("Final use_ptr: %d\n", use_ptr);


    return 0;
}
```

OUTPUT :

```
========== FINAL RESULTS ==========
Total items produced: 200000
Total items consumed: 200000
Final fill_ptr: 0
Final use_ptr: 0
```

EXPLANATION :

Problem : Multiple threads writing shared data together.

Fix : Protect shared data with pthread_mutex_lock() and pthread_mutex_unlock().

Result : No race condition, consistent final counts.

- **Q2 :**

CODE :

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

#define BUFFER_SIZE 100

sem_t empty;  // Counts empty slots in buffer
sem_t full;   // Counts full slots in buffer
pthread_mutex_t mutex;  // Mutex for mutual exclusion
int buffer[BUFFER_SIZE];
int fill_ptr = 0;
int use_ptr = 0;
int item_counter = 0;
int items_produced = 0;
int items_consumed = 0;

int produce_item() {
   return item_counter++;
}

void* producer(void* arg) {
   int id = *(int*)arg;
   for(int i = 0; i < 100000; i++) {
       int item = produce_item();
       sem_wait(&empty);  // Wait for an empty slot
       pthread_mutex_lock(&mutex);  // Lock before access shared buffer
       buffer[fill_ptr] = item;
       fill_ptr = (fill_ptr + 1) % BUFFER_SIZE;
       items_produced++;
       printf("Producer %d produced: %d\n", id, item);
       pthread_mutex_unlock(&mutex);  // Unlock after access shared buffer
       sem_post(&full);  // Signal slot is full

   }
   return NULL;
}

void* consumer(void* arg) {
```

```
    int id = *(int*)arg;
    for(int i = 0; i < 100000; i++) {
        sem_wait(&full);  // Wait for a full slot
        pthread_mutex_lock(&mutex);  // Lock before accessing shared buffer
        int item = buffer[use_ptr];
        use_ptr = (use_ptr + 1) % BUFFER_SIZE;
        items_consumed++;
        printf("Consumer %d consumed: %d\n", id, item);
        pthread_mutex_unlock(&mutex);  // Unlock after accessing shared
buffer
        sem_post(&empty);  // Signal that a slot is now empty

    }
    return NULL;
}

int main() {
    pthread_t prod_threads[2], cons_threads[2];
    int prod_ids[2] = {1, 2};
    int cons_ids[2] = {1, 2};

    sem_init(&empty, 0, BUFFER_SIZE);  // Initially all slots are empty
    sem_init(&full, 0, 0);              // Initially no slots are full
    pthread_mutex_init(&mutex, NULL);  // Initialize mutex

    printf("Starting Producer-Consumer with Finite Buffer\n");

    for(int i = 0; i < 2; i++) {
        pthread_create(&prod_threads[i], NULL, producer, &prod_ids[i]);
        pthread_create(&cons_threads[i], NULL, consumer, &cons_ids[i]);
    }

    for(int i = 0; i < 2; i++) {
        pthread_join(prod_threads[i], NULL);
        pthread_join(cons_threads[i], NULL);
    }

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);  // Destroy mutex
```

```c
    printf("\n========== FINAL RESULTS ==========\n");
    printf("Total items produced: %d\n", items_produced);
    printf("Total items consumed: %d\n", items_consumed);
    printf("Final fill_ptr: %d\n", fill_ptr);
    printf("Final use_ptr: %d\n", use_ptr);


    return 0;
}
```

OUTPUT :

```
========== FINAL RESULTS ==========
Total items produced: 200000
Total items consumed: 200000
Final fill_ptr: 0
Final use_ptr: 0
```

EXPLANATION :
This implements the classic producer-consumer problem with a finite buffer using two semaphores: `empty` (tracks available slots) and `full` (tracks filled slots). Producers wait if the buffer is full, and consumers wait if it is empty, ensuring no data is overwritten or lost. A mutex protects access to the shared buffer and pointers, providing mutual exclusion. This approach guarantees correct synchronization and prevents race conditions and buffer overflows.

- ## Q3 :

CODE :

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

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

#define BUFFER_SIZE 5

sem_t empty;  // Tracks empty slots
sem_t full;   // Tracks full slots
pthread_mutex_t mutex;
int buffer[BUFFER_SIZE];
int fill_ptr = 0;
int use_ptr = 0;
int item_counter = 0;

int produce_item() {
   return item_counter++;
}

void* producer(void* arg) {
   int id = *(int*)arg;
   for(int i = 0; i < 10; i++) {
       int item = produce_item();

       sem_wait(&empty);
       pthread_mutex_lock(&mutex);

       buffer[fill_ptr] = item;
       fill_ptr = (fill_ptr + 1) % BUFFER_SIZE;
       printf("Producer %d produced: %d\n", id, item);

       pthread_mutex_unlock(&mutex);
       sem_post(&full);
       usleep(100000);
   }
   return NULL;
}

void* consumer(void* arg) {
   int id = *(int*)arg;
   for(int i = 0; i < 10; i++) {
```

```
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        int item = buffer[use_ptr];
        use_ptr = (use_ptr + 1) % BUFFER_SIZE;
        printf("Consumer %d consumed: %d\n", id, item);

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        usleep(150000);
    }
    return NULL;
}

int main() {
    pthread_t prod1, cons1;
    int prod_id = 1, cons_id = 1;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    printf("Starting code - Deadlock FIXED!\n");

    pthread_create(&prod1, NULL, producer, &prod_id);
    pthread_create(&cons1, NULL, consumer, &cons_id);

    pthread_join(prod1, NULL);
    pthread_join(cons1, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
    printf("Program completed\n");
    return 0;
}
```

OUTPUT :

```
● (base) → os ./q3
Starting code - Deadlock FIXED!
Producer 1 produced: 0
Consumer 1 consumed: 0
Producer 1 produced: 1
Consumer 1 consumed: 1
Producer 1 produced: 2
Consumer 1 consumed: 2
Producer 1 produced: 3
Producer 1 produced: 4
Consumer 1 consumed: 3
Producer 1 produced: 5
Consumer 1 consumed: 4
Producer 1 produced: 6
Producer 1 produced: 7
Consumer 1 consumed: 5
Producer 1 produced: 8
Consumer 1 consumed: 6
Producer 1 produced: 9
Consumer 1 consumed: 7
Consumer 1 consumed: 8
Consumer 1 consumed: 9
Program completed
```

EXPLANATION :

This code demonstrates the producer-consumer problem with a bounded buffer of size 5, using semaphores and a mutex for synchronization. The `empty` and `full` semaphores ensure that producers wait when the buffer is full and consumers wait when it is empty. The mutex protects access to the shared buffer and pointers, preventing race conditions. This approach guarantees safe and deadlock-free concurrent access to the buffer.

- **Q4 :**

CODE :

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

sem_t roomEmpty;
pthread_mutex_t readers_mutex;  // Mutex to protect readers counter
int readers = 0;
int shared_data = 0;
int read_count = 0;
int write_count = 0;

void* reader(void* arg) {
   int id = *(int*)arg;
   for(int i = 0; i < 50; i++) {

       pthread_mutex_lock(&readers_mutex);
       readers++;
       if(readers == 1) {
           sem_wait(&roomEmpty);
       }
       pthread_mutex_unlock(&readers_mutex);

       int value = shared_data;
       read_count++;
       printf("Reader %d reads: %d (readers=%d)\n", id, value, readers);

       pthread_mutex_lock(&readers_mutex);
       readers--;
       if(readers == 0) {
           sem_post(&roomEmpty);
       }
       pthread_mutex_unlock(&readers_mutex);

       usleep(10000);
   }
   return NULL;
}
```

```c
void* writer(void* arg) {
   int id = *(int*)arg;
   for(int i = 0; i < 20; i++) {
       sem_wait(&roomEmpty);

       shared_data++;
       write_count++;
       printf("Writer %d writes: %d\n", id, shared_data);
       usleep(50000);

       sem_post(&roomEmpty);
       usleep(100000);
   }
   return NULL;
}

int main() {
   pthread_t reader_threads[4], writer_threads[2];
   int reader_ids[4] = {1, 2, 3, 4};
   int writer_ids[2] = {1, 2};

   sem_init(&roomEmpty, 0, 1);
   pthread_mutex_init(&readers_mutex, NULL);

   printf("=== FIXED VERSION - Proper Reader-Writer Synchronization
===\n");
   printf("Expected behavior: Writers should NEVER run while
readers>0\n\n");

   for(int i = 0; i < 4; i++) {
       pthread_create(&reader_threads[i], NULL, reader, &reader_ids[i]);
   }
   for(int i = 0; i < 2; i++) {
       pthread_create(&writer_threads[i], NULL, writer, &writer_ids[i]);
   }

   for(int i = 0; i < 4; i++) {
       pthread_join(reader_threads[i], NULL);
   }
   for(int i = 0; i < 2; i++) {
```

```c
        pthread_join(writer_threads[i], NULL);
    }

    sem_destroy(&roomEmpty);
    pthread_mutex_destroy(&readers_mutex);

    printf("\n========== RESULTS ==========\n");
    printf("Final readers counter: %d (should be 0)\n", readers);
    printf("Total reads: %d\n", read_count);
    printf("Total writes: %d (expected: 40)\n", write_count);
    printf("Final shared_data: %d (expected: 40)\n", shared_data);

    if(readers != 0) {
        printf("\nBUG DETECTED: readers counter is corrupted!\n");
    }
    if(write_count != 40 || shared_data != 40) {
        printf("BUG DETECTED: Data corruption occurred!\n");
    }

    return 0;
}
```

OUTPUT :

```
========== RESULTS ==========
Final readers counter: 0 (should be 0)
Total reads: 200
Total writes: 40 (expected: 40)
Final shared_data: 40 (expected: 40)
```

EXPLANATION :

Added a mutex to protect the readers count, ensuring correct synchronization so multiple readers don't corrupt the counter or violate writer exclusivity.

- **Q5 :**

CODE :

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


typedef struct {
   int counter;
   sem_t mutex;
} Lightswitch;

Lightswitch readSwitch;
sem_t roomEmpty;


void lightswitch_init(Lightswitch *ls) {
   ls->counter = 0;
   sem_init(&ls->mutex, 0, 1);
}


void lightswitch_lock(Lightswitch *ls, sem_t *semaphore) {
   sem_wait(&ls->mutex);
   ls->counter++;
   if (ls->counter == 1) {
         sem_wait(semaphore);
   }
   sem_post(&ls->mutex);
}

void lightswitch_unlock(Lightswitch *ls, sem_t *semaphore) {
   sem_wait(&ls->mutex);
   ls->counter--;
   if (ls->counter == 0) {
        sem_post(semaphore);
   }
   sem_post(&ls->mutex);
}

void *reader(void *arg) {
```

```c
    int id = *(int *)arg;


    lightswitch_lock(&readSwitch, &roomEmpty);

    printf("Reader %d is reading...\n", id);
    sleep(1);
    printf("Reader %d finished reading.\n", id);

    lightswitch_unlock(&readSwitch, &roomEmpty);

    return NULL;
}

void *writer(void *arg) {
    int id = *(int *)arg;

    sem_wait(&roomEmpty);

    printf("Writer %d is writing...\n", id);
    sleep(2);
    printf("Writer %d finished writing.\n", id);

    sem_post(&roomEmpty);

    return NULL;
}

int main() {
    pthread_t r1, r2, w1, w2;
    int rID1 = 1, rID2 = 2;
    int wID1 = 1, wID2 = 2;

     lightswitch_init(&readSwitch);
    sem_init(&roomEmpty, 0, 1);

    pthread_create(&r1, NULL, reader, &rID1);
    pthread_create(&r2, NULL, reader, &rID2);
    pthread_create(&w1, NULL, writer, &wID1);
    pthread_create(&w2, NULL, writer, &wID2);
```

```
    pthread_join(r1, NULL);
    pthread_join(r2, NULL);
    pthread_join(w1, NULL);
    pthread_join(w2, NULL);

    sem_destroy(&roomEmpty);
    sem_destroy(&readSwitch.mutex);

    return 0;
}
```

OUTPUT :

```
(base) ➜  os ./q5
Reader 1 is reading...
Reader 2 is reading...
Reader 1 finished reading.
Reader 2 finished reading.
Writer 1 is writing...
Writer 1 finished writing.
Writer 2 is writing...
Writer 2 finished writing.
```

EXPLANATION :

Lightswitch ensures multiple readers can read together while writers get exclusive access safely.

- **Q6 :**

CODE :

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

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

#define N 5

sem_t forks[N];
int eat_count[N] = {0};

void* philosopher(void* arg) {
   int id = *(int*)arg;
   int left_fork = id;
   int right_fork = (id + 1) % N;

   // Lower ID First Strategy: always pick lower-numbered fork first
   int first_fork = (left_fork < right_fork) ? left_fork : right_fork;
   int second_fork = (left_fork < right_fork) ? right_fork : left_fork;

   for(int i = 0; i < 3; i++) {

       printf("Philosopher %d is thinking...\n", id);
       usleep(100000);


       printf("Philosopher %d is hungry, reaching for forks %d and %d\n",
              id, left_fork, right_fork);

       // Pick up lower-numbered fork first
       sem_wait(&forks[first_fork]);
       printf("  Philosopher %d picked up fork %d (first)\n", id,
first_fork);
       usleep(50000);

       // Then pick up higher-numbered fork
       sem_wait(&forks[second_fork]);
       printf("  Philosopher %d picked up fork %d (second)\n", id,
second_fork);
```

```c
        printf("Philosopher %d is EATING (meal #%d)\n", id, eat_count[id] +
1);
        eat_count[id]++;
        usleep(200000);


        sem_post(&forks[first_fork]);
        sem_post(&forks[second_fork]);
        printf("  Philosopher %d put down both forks\n\n", id);
    }

    return NULL;
}

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


    for(int i = 0; i < N; i++) {
        sem_init(&forks[i], 0, 1);
        ids[i] = i;
    }

    printf("=== DINING PHILOSOPHERS - DEADLOCK FIXED (Lower ID First)
===\n");
    printf("Number of philosophers: %d\n", N);
    printf("Strategy: Always pick up lower-numbered fork first\n\n");


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


    for(int i = 0; i < N; i++) {
        pthread_join(philosophers[i], NULL);
    }
```

```
    for(int i = 0; i < N; i++) {
        sem_destroy(&forks[i]);
    }

    printf("\n========== MEAL COUNT ==========\n");
    for(int i = 0; i < N; i++) {
        printf("Philosopher %d ate %d times (expected: 3)\n", i,
eat_count[i]);
    }

    return 0;
}
```

OUTPUT :

```
========== MEAL COUNT ==========
Philosopher 0 ate 3 times (expected: 3)
Philosopher 1 ate 3 times (expected: 3)
Philosopher 2 ate 3 times (expected: 3)
Philosopher 3 ate 3 times (expected: 3)
Philosopher 4 ate 3 times (expected: 3)
```

EXPLANATION :

Deadlock is avoided by making philosophers pick forks in a consistent (lower ID first) order, breaking circular wait.