Assignment 4 33239 M10

1) 4A

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

int *buffer; // Dynamic buffer
int buffer_size;
int in = 0, out = 0;
sem_t empty;
sem_t full;
pthread_mutex_t mutex;

void print_buffer() {
    printf("Buffer: [ ");
    for (int i = 0; i < buffer_size; i++) {
        if (i == in && i == out) {
            printf("I&O ");
        } else if (i == in) {
            printf("I ");
        } else if (i == out) {
            printf("O ");
        } else if (i < in && i >= out) {
            printf("%d ", buffer[i]);
        } else if (in < out && (i >= out || i < in)) {
            printf("%d ", buffer[i]);
        } else {
            printf("E ");
        }
    }
    printf("]\n");
}

void *producer(void *arg) {
    int item_count = *((int *)arg);
    int item;
    for (int i = 0; i < item_count; i++) {
        item = rand() % 100; // Produce a random item
        sem_wait(&empty); // Wait for an empty slot
        pthread_mutex_lock(&mutex); // Lock the buffer
        // Add item to the buffer
        buffer[in] = item;
        in = (in + 1) % buffer_size;
        printf("Producer produced: %d\n", item);
        print_buffer();
        pthread_mutex_unlock(&mutex); // Unlock the buffer
        sem_post(&full); // Signal that there is a full slot
        sleep(1); // Simulate time taken to produce an item
    }
```

```
    pthread_exit(NULL);
}

void *consumer(void *arg) {
    int item_count = *((int *)arg);
    int item;
    for (int i = 0; i < item_count; i++) {
        sem_wait(&full); // Wait for a full slot
        pthread_mutex_lock(&mutex); // Lock the buffer
        // Remove item from the buffer
        item = buffer[out];
        buffer[out] = 0; // Optional: Clear the consumed slot
        out = (out + 1) % buffer_size;
        printf("Consumer consumed: %d\n", item);
        print_buffer();
        pthread_mutex_unlock(&mutex); // Unlock the buffer
        sem_post(&empty); // Signal that there is an empty slot
        sleep(1); // Simulate time taken to consume an item
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t prod_thread, cons_thread;
    int produce_count, consume_count;

    // Get user input for buffer size, produce count, and consume count
    printf("Enter the buffer size: ");
    scanf("%d", &buffer_size);
    printf("Enter the number of items to produce: ");
    scanf("%d", &produce_count);
    printf("Enter the number of items to consume: ");
    scanf("%d", &consume_count);

    // Allocate buffer dynamically
    buffer = (int *)malloc(buffer_size * sizeof(int));

    // Initialize the buffer with empty slots
    for (int i = 0; i < buffer_size; i++) {
        buffer[i] = 0;
    }

    // Initialize the semaphores and mutex
    sem_init(&empty, 0, buffer_size);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    // Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, &produce_count);
    pthread_create(&cons_thread, NULL, consumer, &consume_count);
```

```
    // Wait for threads to finish
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    // Destroy the semaphores and mutex
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    // Free the dynamically allocated buffer
    free(buffer);
    return 0;
}
```

OUTPUT:

```
pritam@PritamPC:/mnt/d/TE/SEM V/33239_OSL$ g++ temp.
Enter the buffer size: 5
Enter the number of items to produce: 7
Enter the number of items to consume: 7
Producer produced: 83
Buffer: [ O I E E E ]
Consumer consumed: 83
Buffer: [ E I&O E E E ]
Producer produced: 86
Buffer: [ E O I E E ]
Consumer consumed: 86
Buffer: [ E E I&O E E ]
Producer produced: 77
Buffer: [ E E O I E ]
Consumer consumed: 77
Buffer: [ E E E I&O E ]
Producer produced: 15
Buffer: [ E E E O I ]
Consumer consumed: 15
Buffer: [ E E E E I&O ]
Producer produced: 93
Buffer: [ I E E E O ]
Consumer consumed: 93
Buffer: [ I&O E E E E ]
Producer produced: 35
```

2) 4B

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

sem_t rw_mutex; // Controls access to the shared resource
sem_t mutex; // Controls access to the reader count
int read_count = 0; // Number of active readers

void* reader(void* arg) {
    int reader_id = *((int*)arg);
    int iterations = *((int*)(arg + sizeof(int))); // Get the number of iterations
from the passed arguments

    for (int i = 0; i < iterations; i++) {
        sem_wait(&mutex); // Request access to modify read_count
        read_count++;
        if (read_count == 1) {
            sem_wait(&rw_mutex); // First reader locks the shared resource
        }
        sem_post(&mutex); // Release access to read_count

        // Reading section
        printf("Reader %d is reading\n", reader_id);
        fflush(stdout);
        sleep(1); // Simulate reading time

        sem_wait(&mutex); // Request access to modify read_count
        read_count--;
        if (read_count == 0) {
            sem_post(&rw_mutex); // Last reader unlocks the shared resource
        }
        sem_post(&mutex); // Release access to read_count

        sleep(1); // Simulate time between reading attempts
    }

    printf("Reader %d has finished reading\n", reader_id);
    fflush(stdout);
    return NULL;
}

void* writer(void* arg) {
    int writer_id = *((int*)arg);
    int iterations = *((int*)(arg + sizeof(int))); // Get the number of iterations
from the passed arguments

    for (int i = 0; i < iterations; i++) {
        sem_wait(&rw_mutex); // Request exclusive access to the shared resource
```

```
        // Writing section
        printf("Writer %d is writing\n", writer_id);
        fflush(stdout);
        sleep(1); // Simulate writing time

        sem_post(&rw_mutex); // Release exclusive access to the shared resource
        sleep(2); // Simulate time between writing attempts
    }

    printf("Writer %d has finished writing\n", writer_id);
    fflush(stdout);
    return NULL;
}

int main() {
    int num_readers, num_writers;

    // Ask the user for the number of readers and writers
    printf("Enter the number of readers: ");
    fflush(stdout);
    scanf("%d", &num_readers);

    printf("Enter the number of writers: ");
    fflush(stdout);
    scanf("%d", &num_writers);

    pthread_t readers[num_readers], writers[num_writers];
    int reader_ids[num_readers], writer_ids[num_writers];
    int read_iterations[num_readers], write_iterations[num_writers];

    // Collect reader and writer iterations
    for (int i = 0; i < num_readers; i++) {
        reader_ids[i] = i + 1;
        printf("Enter the number of read operations for Reader %d: ",
reader_ids[i]);
        fflush(stdout);
        scanf("%d", &read_iterations[i]);
    }

    for (int i = 0; i < num_writers; i++) {
        writer_ids[i] = i + 1;
        printf("Enter the number of write operations for Writer %d: ",
writer_ids[i]);
        fflush(stdout);
        scanf("%d", &write_iterations[i]);
    }

    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&rw_mutex, 0, 1);
```

```
    // Create reader threads
    for (int i = 0; i < num_readers; i++) {
        // Pass both ID and iterations to the thread
        int* args = malloc(2 * sizeof(int));
        args[0] = reader_ids[i];
        args[1] = read_iterations[i];
        pthread_create(&readers[i], NULL, reader, args);
    }

    // Create writer threads
    for (int i = 0; i < num_writers; i++) {
        // Pass both ID and iterations to the thread
        int* args = malloc(2 * sizeof(int));
        args[0] = writer_ids[i];
        args[1] = write_iterations[i];
        pthread_create(&writers[i], NULL, writer, args);
    }

    // Wait for all threads to finish
    for (int i = 0; i < num_readers; i++) {
        pthread_join(readers[i], NULL);
    }

    for (int i = 0; i < num_writers; i++) {
        pthread_join(writers[i], NULL);
    }

    // Destroy the semaphores
    sem_destroy(&mutex);
    sem_destroy(&rw_mutex);

    return 0;
}
```

OUTPUT:

```
pritam@PritamPC:/mnt/d/TE/SEM V/33239_OSL$ g++ temp.cpp -o te
Enter the number of readers: 3
Enter the number of writers: 2
Enter the number of read operations for Reader 1: 3
Enter the number of read operations for Reader 2: 5
Enter the number of read operations for Reader 3: 2
Enter the number of write operations for Writer 1: 3
Enter the number of write operations for Writer 2: 4
Reader 1 is reading
Reader 2 is reading
Reader 3 is reading
Writer 1 is writing
Writer 2 is writing
Reader 2 is reading
Reader 3 is reading
Reader 1 is reading
Writer 1 is writing
Reader 3 has finished reading
Writer 2 is writing
Reader 2 is reading
Reader 1 is reading
Writer 1 is writing
Reader 1 has finished reading
Reader 2 is reading
Writer 2 is writing
Writer 1 has finished writing
Reader 2 is reading
Writer 2 is writing
Reader 2 has finished reading
Writer 2 has finished writing
```

Assignment 7B 33239 M10

- Server Code

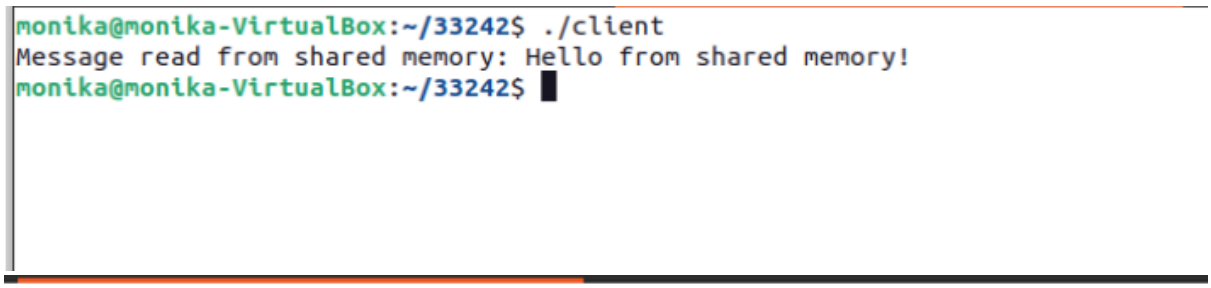
```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <stdlib.h>

#define SHM_SIZE 1024 // Size of the shared memory

int main() {
    // Generate a unique key
    key_t key = ftok("shmfile", 65);

    // Create shared memory
    int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);

    // Attach to the shared memory
    char *shared_memory = (char*) shmat(shmid, NULL, 0);

    // Write a message to the shared memory
    printf("Write a message to shared memory: ");
    fgets(shared_memory, SHM_SIZE, stdin); // Get input from user
    printf("Message written to shared memory: %s", shared_memory);

    // Detach from shared memory
    shmdt(shared_memory);

    return 0;
}
```

- Client Code

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>

#define SHM_SIZE 1024 // Size of the shared memory

int main() {
    // Generate a unique key
    key_t key = ftok("shmfile", 65);

    // Get the shared memory ID
    int shmid = shmget(key, SHM_SIZE, 0666);

    // Attach to the shared memory
    char *shared_memory = (char*) shmat(shmid, NULL, 0);

    // Read the message from the shared memory
    printf("Message read from shared memory: %s", shared_memory);
```

```
    // Detach from shared memory
    shmdt(shared_memory);

    // Destroy the shared memory (cleanup)
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

OUTPUT:




```
monika@monika-VirtualBox:~/33242$ gcc server.c -o server
monika@monika-VirtualBox:~/33242$ gcc client.c -o client
monika@monika-VirtualBox:~/33242$ ./server
Write a message to shared memory: Hello from shared memory!
Message written to shared memory: Hello from shared memory!
monika@monika-VirtualBox:~/33242$
```