EXPERIMENT-7

Q1. Implement the producer consumer problem using pthreads and mutex operations. Test Cases:

(a) A producer only produces if buffer is empty and consumer only consumes if some content is in the buffer.

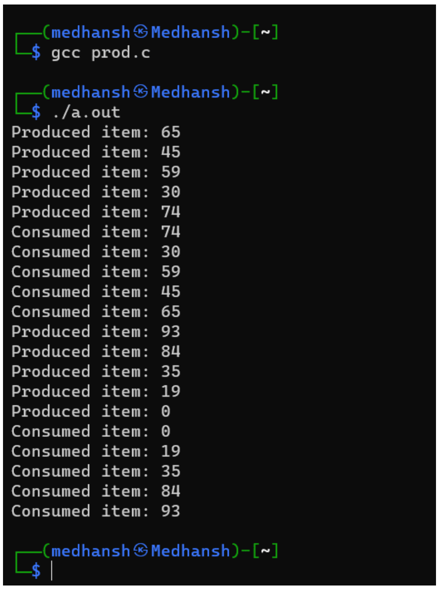
(b) A producer produces(writes) an item in the buffer and consumer consumes(deletes) the last produces item in the buffer.

(c) A producer produces(writes) on the last consumed(deleted) index of the buffer.

Code:

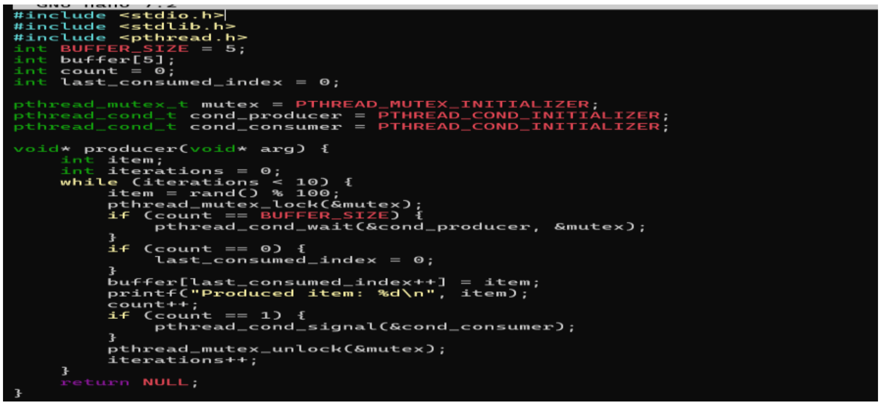
```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>



int BUFFER_SIZE = 5;

int buffer[5];

int count = 0;

int last_consumed_index = 0;


pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t cond_producer = PTHREAD_COND_INITIALIZER;

pthread_cond_t cond_consumer = PTHREAD_COND_INITIALIZER;


void* producer(void* arg) {

    int item;

    int iterations = 0;

    while (iterations < 10) { // exit after 10 iterations

        item = rand() % 100; // generate a random item

        pthread_mutex_lock(&mutex);

        if (count == BUFFER_SIZE) {

            pthread_cond_wait(&cond_producer, &mutex);

        }

        if (count == 0) {

            last_consumed_index = 0; // reset last consumed index if buffer is empty

        }

        buffer[last_consumed_index++] = item;

        printf("Produced item: %d\n", item);
        count++;
        if (count == 1) {
            pthread_cond_signal(&cond_consumer);
        }
        pthread_mutex_unlock(&mutex);
        iterations++;
    }
    return NULL;
}

void* consumer(void* arg) {
    int item;
    int iterations = 0;
    while (iterations < 10) { // exit after 10 iterations
        pthread_mutex_lock(&mutex);
        if (count == 0) {
            pthread_cond_wait(&cond_consumer, &mutex);
        }
        item = buffer[--last_consumed_index];
        printf("Consumed item: %d\n", item);
        count--;
        if (count == BUFFER_SIZE - 1) {
            pthread_cond_signal(&cond_producer);
        }
        pthread_mutex_unlock(&mutex);
        iterations++;
    }
    return NULL;
}

int main() {
    pthread_t producer_thread, consumer_thread;
    srand(time(NULL)); // initialize the random seed

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);


    pthread_join(producer_thread, NULL);

    pthread_join(consumer_thread, NULL);


    return 0;

}
```

Output:

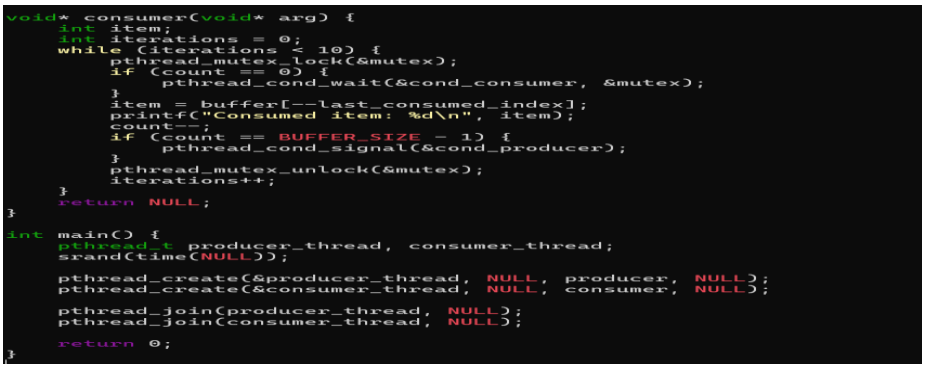


```
┌──(medhansh㉿Medhansh)-[~]
└─$ gcc prod.c

┌──(medhansh㉿Medhansh)-[~]
└─$ ./a.out
Produced item: 65
Produced item: 45
Produced item: 59
Produced item: 30
Produced item: 74
Consumed item: 74
Consumed item: 30
Consumed item: 59
Consumed item: 45
Consumed item: 65
Produced item: 93
Produced item: 84
Produced item: 35
Produced item: 19
Produced item: 0
Consumed item: 0
Consumed item: 19
Consumed item: 35
Consumed item: 84
Consumed item: 93

┌──(medhansh㉿Medhansh)-[~]
└─$ 
```

Code:



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int BUFFER_SIZE = 5;
int buffer[5];
int count = 0;
int last_consumed_index = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_producer = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_consumer = PTHREAD_COND_INITIALIZER;

void* producer(void* arg) {
    int item;
    int iterations = 0;
    while (iterations < 10) {
        item = rand() % 100;
        pthread_mutex_lock(&mutex);
        if (count == BUFFER_SIZE) {
            pthread_cond_wait(&cond_producer, &mutex);
        }
        if (count == 0) {
            last_consumed_index = 0;
        }
        buffer[last_consumed_index++] = item;
        printf("Produced item: %d\n", item);
        count++;
        if (count == 1) {
            pthread_cond_signal(&cond_consumer);
        }
        pthread_mutex_unlock(&mutex);
        iterations++;
    }
    return NULL;
}
```



```
void* consumer(void* arg) {
    int item;
    int iterations = 0;
    while (iterations < 10) {
        pthread_mutex_lock(&mutex);
        if (count == 0) {
            pthread_cond_wait(&cond_consumer, &mutex);
        }
        item = buffer[--last_consumed_index];
        printf("Consumed item: %d\n", item);
        count--;
        if (count == BUFFER_SIZE - 1) {
            pthread_cond_signal(&cond_producer);
        }
        pthread_mutex_unlock(&mutex);
        iterations++;
    }
    return NULL;
}

int main() {
    pthread_t producer_thread, consumer_thread;
    srand(time(NULL));

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    return 0;
}
```

Q2 .Implement the reader writer problem using semaphore and mutex operations to synchronize n readers active in reader section at a same time, and one writer active at a

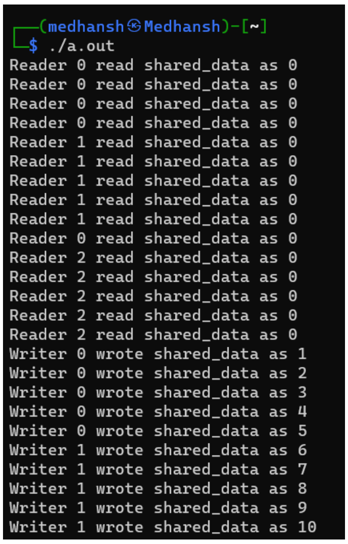
a) If n readers are active no writer is allowed to write.

(b) If one writer is writing no other writer should be allowed to read or write on the shared variable.

Code:

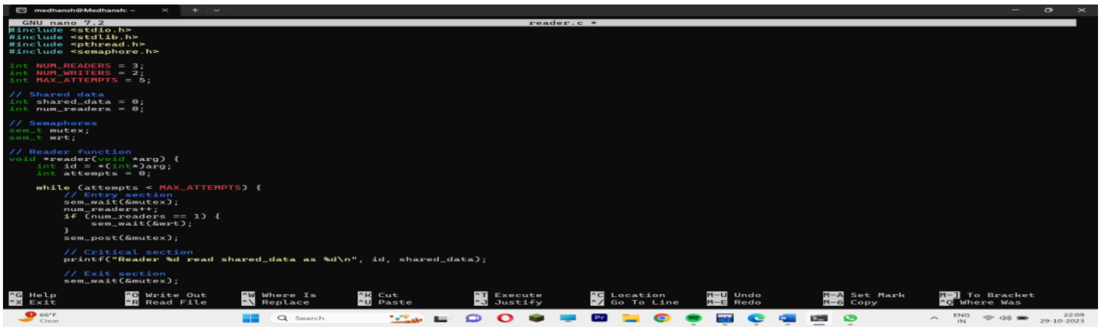
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

int NUM_READERS = 3;
int NUM_WRITERS = 2;
int MAX_ATTEMPTS = 5;

// Shared data
int shared_data = 0;
int num_readers = 0;

// Semaphores
sem_t mutex;
sem_t wrt;

// Reader function
void *reader(void *arg) {
    int id = *(int*)arg;
    int attempts = 0;

    while (attempts < MAX_ATTEMPTS) {
        // Entry section
        sem_wait(&mutex);
        num_readers++;
        if (num_readers == 1) {
            sem_wait(&wrt);
        }
        sem_post(&mutex);

        // Critical section
        printf("Reader %d read shared_data as %d\n", id, shared_data);

        // Exit section
        sem_wait(&mutex);
        num_readers--;
        if (num_readers == 0) {
            sem_post(&wrt);
        }
        sem_post(&mutex);

        attempts++;
    }

    pthread_exit(NULL);
}

// Writer function
void *writer(void *arg) {
    int id = *(int*)arg;
    int attempts = 0;

    while (attempts < MAX_ATTEMPTS) {
        // Entry section
        sem_wait(&wrt);

        // Critical section
        shared_data++;
        printf("Writer %d wrote shared_data as %d\n", id, shared_data);

        // Exit section
        sem_post(&wrt);
        attempts++;
    }

    pthread_exit(NULL);
}

int main() {
    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&wrt, 0, 1);

    // Create reader threads
    pthread_t reader_threads[NUM_READERS];
    int reader_ids[NUM_READERS];
    for (int i = 0; i < NUM_READERS; i++) {
        reader_ids[i] = i;
        pthread_create(&reader_threads[i], NULL, reader, &reader_ids[i]);
    }

    // Create writer threads
    pthread_t writer_threads[NUM_WRITERS];
    int writer_ids[NUM_WRITERS];
    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_ids[i] = i;
        pthread_create(&writer_threads[i], NULL, writer, &writer_ids[i]);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(reader_threads[i], NULL);
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writer_threads[i], NULL);
    }
    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&wrt);


    return 0;
}
```

Output:

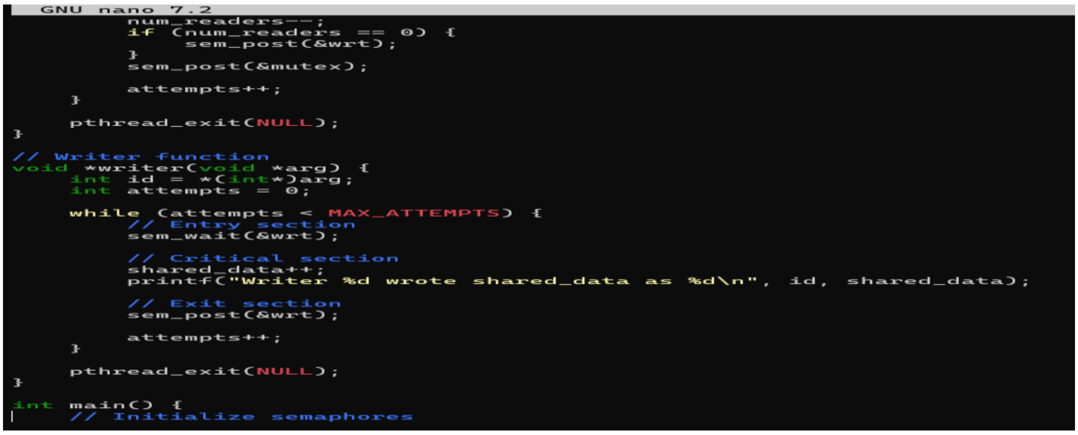


```
┌──(medhansh㉿Medhansh)-[~]
└─$ ./a.out
Reader 0 read shared_data as 0
Reader 0 read shared_data as 0
Reader 0 read shared_data as 0
Reader 0 read shared_data as 0
Reader 1 read shared_data as 0
Reader 1 read shared_data as 0
Reader 1 read shared_data as 0
Reader 1 read shared_data as 0
Reader 1 read shared_data as 0
Reader 0 read shared_data as 0
Reader 2 read shared_data as 0
Reader 2 read shared_data as 0
Reader 2 read shared_data as 0
Reader 2 read shared_data as 0
Reader 2 read shared_data as 0
Writer 0 wrote shared_data as 1
Writer 0 wrote shared_data as 2
Writer 0 wrote shared_data as 3
Writer 0 wrote shared_data as 4
Writer 0 wrote shared_data as 5
Writer 1 wrote shared_data as 6
Writer 1 wrote shared_data as 7
Writer 1 wrote shared_data as 8
Writer 1 wrote shared_data as 9
Writer 1 wrote shared_data as 10
```

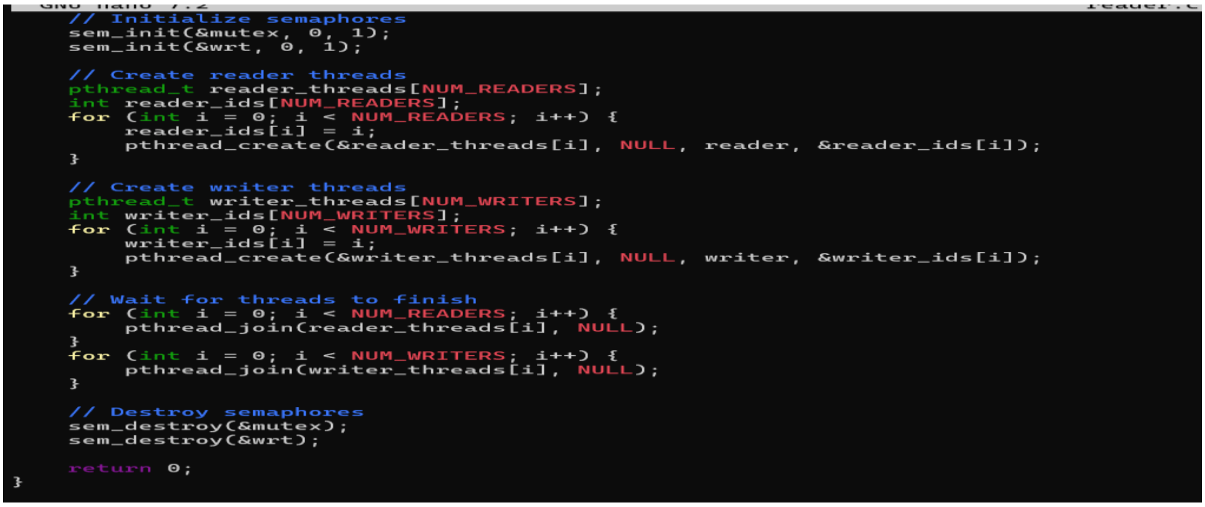
Code:



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

int NUM_READERS = 3;
int NUM_WRITERS = 2;
int MAX_ATTEMPTS = 5;

// Shared data
int shared_data = 0;
int num_readers = 0;

// Semaphores
sem_t mutex;
sem_t wrt;

// Reader function
void *reader(void *arg) {
    int id = *(int*)arg;
    int attempts = 0;

    while (attempts < MAX_ATTEMPTS) {
        // Entry section
        sem_wait(&mutex);
        num_readers++;
        if (num_readers == 1) {
            sem_wait(&wrt);
        }
        sem_post(&mutex);

        // Critical section
        printf("Reader %d read shared_data as %d\n", id, shared_data);

        // Exit section
        sem_wait(&mutex);
```



```
        num_readers--;
        if (num_readers == 0) {
            sem_post(&wrt);
        }
        sem_post(&mutex);

        attempts++;
    }

    pthread_exit(NULL);
}

// Writer function
void *writer(void *arg) {
    int id = *(int*)arg;
    int attempts = 0;

    while (attempts < MAX_ATTEMPTS) {
        // Entry section
        sem_wait(&wrt);

        // Critical section
        shared_data++;
        printf("Writer %d wrote shared_data as %d\n", id, shared_data);

        // Exit section
        sem_post(&wrt);

        attempts++;
    }

    pthread_exit(NULL);
}

int main() {
    // Initialize semaphores
```



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

    // Create reader threads
    pthread_t reader_threads[NUM_READERS];
    int reader_ids[NUM_READERS];
    for (int i = 0; i < NUM_READERS; i++) {
        reader_ids[i] = i;
        pthread_create(&reader_threads[i], NULL, reader, &reader_ids[i]);
    }

    // Create writer threads
    pthread_t writer_threads[NUM_WRITERS];
    int writer_ids[NUM_WRITERS];
    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_ids[i] = i;
        pthread_create(&writer_threads[i], NULL, writer, &writer_ids[i]);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(reader_threads[i], NULL);
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writer_threads[i], NULL);
    }

    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;
}
```

Lab Excercises

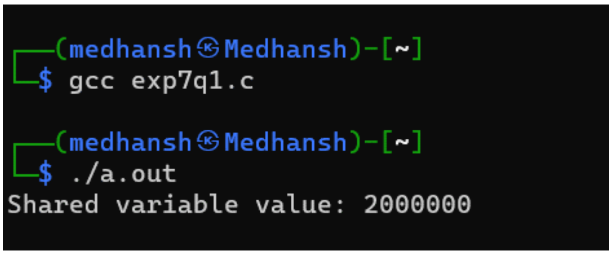
1.Write a C program to create two threads that increment a shared variable using a mutex to synchronize access to the variable

Code:

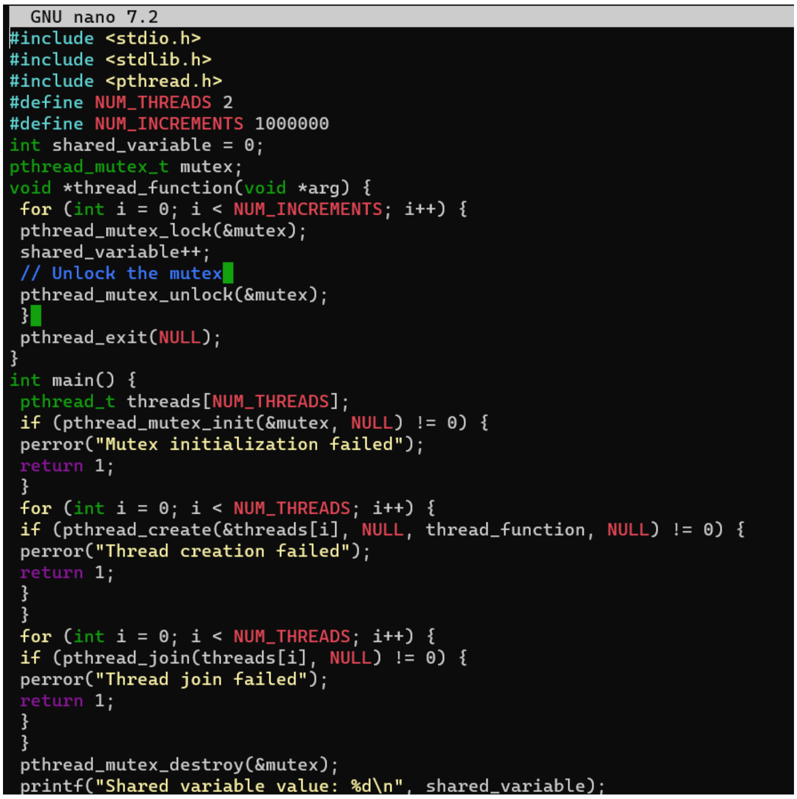
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 2
#define NUM_INCREMENTS 1000000
int shared_variable = 0;
pthread_mutex_t mutex;
void *thread_function(void *arg) {
 for (int i = 0; i < NUM_INCREMENTS; i++) {
 pthread_mutex_lock(&mutex);
 shared_variable++;
 // Unlock the mutex
 pthread_mutex_unlock(&mutex);
 }
 pthread_exit(NULL);
}
int main() {
 pthread_t threads[NUM_THREADS];
 if (pthread_mutex_init(&mutex, NULL) != 0) {
 perror("Mutex initialization failed");
 return 1;
 }
 for (int i = 0; i < NUM_THREADS; i++) {
 if (pthread_create(&threads[i], NULL, thread_function, NULL) != 0) {
 perror("Thread creation failed");
 return 1;
 }
 }
 for (int i = 0; i < NUM_THREADS; i++) {
```

```
if (pthread_join(threads[i], NULL) != 0) {

perror("Thread join failed");

return 1;

}

}

pthread_mutex_destroy(&mutex);

printf("Shared variable value: %d\n", shared_variable);

return 0;}
```

output:



```
┌──(medhansh㉿Medhansh)-[~]
└─$ gcc exp7q1.c

┌──(medhansh㉿Medhansh)-[~]
└─$ ./a.out
Shared variable value: 2000000
```



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 2
#define NUM_INCREMENTS 1000000
int shared_variable = 0;
pthread_mutex_t mutex;
void *thread_function(void *arg) {
 for (int i = 0; i < NUM_INCREMENTS; i++) {
 pthread_mutex_lock(&mutex);
 shared_variable++;
 // Unlock the mutex
 pthread_mutex_unlock(&mutex);
 }
 pthread_exit(NULL);
}
int main() {
 pthread_t threads[NUM_THREADS];
 if (pthread_mutex_init(&mutex, NULL) != 0) {
 perror("Mutex initialization failed");
 return 1;
 }
 for (int i = 0; i < NUM_THREADS; i++) {
 if (pthread_create(&threads[i], NULL, thread_function, NULL) != 0) {
 perror("Thread creation failed");
 return 1;
 }
 }
 for (int i = 0; i < NUM_THREADS; i++) {
 if (pthread_join(threads[i], NULL) != 0) {
 perror("Thread join failed");
 return 1;
 }
 }
 pthread_mutex_destroy(&mutex);
 printf("Shared variable value: %d\n", shared_variable);
```

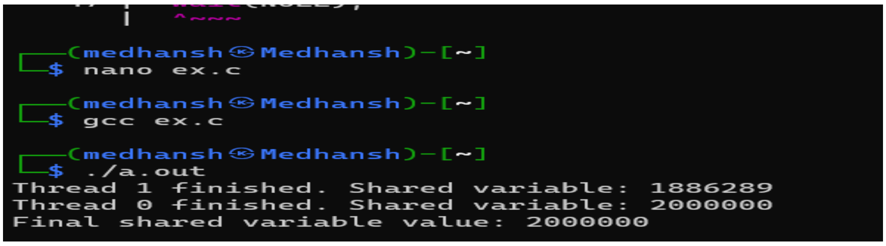
2.Write a C program to create two processes that increment a shared variable using semaphores to synchronize access to the variable.

Code:

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

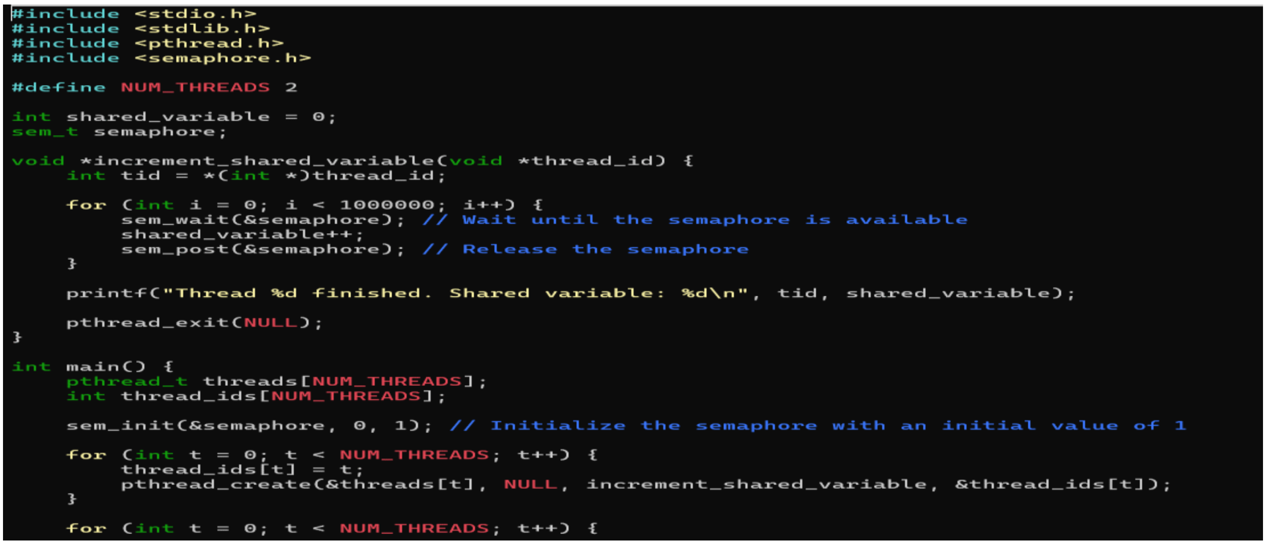
```
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define NUM_THREADS 2

int shared_variable = 0;
sem_t semaphore;

void *increment_shared_variable(void *thread_id) {
    int tid = *(int *)thread_id;

    for (int i = 0; i < 1000000; i++) {
        sem_wait(&semaphore); // Wait until the semaphore is available
        shared_variable++;
        sem_post(&semaphore); // Release the semaphore
    }

    printf("Thread %d finished. Shared variable: %d\n", tid, shared_variable);

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    sem_init(&semaphore, 0, 1); // Initialize the semaphore with an initial value of 1

    for (int t = 0; t < NUM_THREADS; t++) {
        thread_ids[t] = t;
        pthread_create(&threads[t], NULL, increment_shared_variable, &thread_ids[t]);
    }

    for (int t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }

    sem_destroy(&semaphore);

    printf("Final shared variable value: %d\n", shared_variable);

    return 0;
}
```

output:

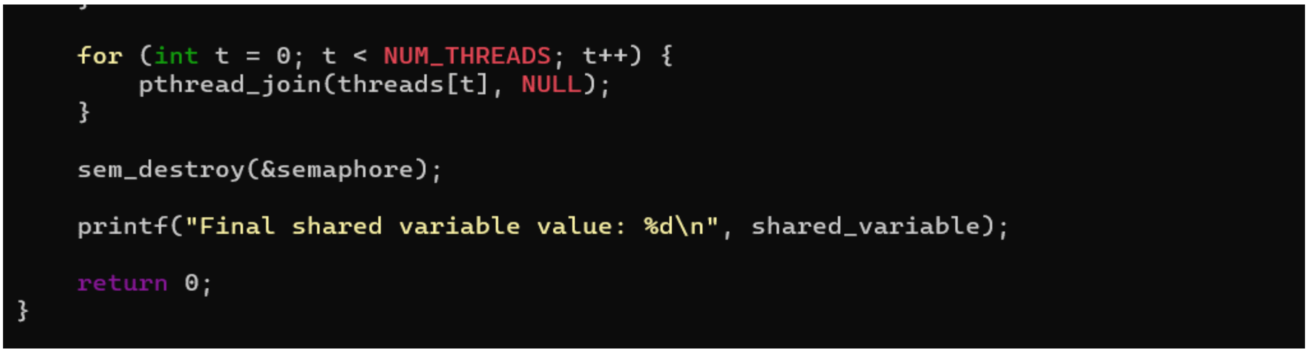


```
      |    ^~~~

┌──(medhansh㉿Medhansh)-[~]
└─$ nano ex.c

┌──(medhansh㉿Medhansh)-[~]
└─$ gcc ex.c

┌──(medhansh㉿Medhansh)-[~]
└─$ ./a.out
Thread 1 finished. Shared variable: 1886289
Thread 0 finished. Shared variable: 2000000
Final shared variable value: 2000000
```

Code:



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

#define NUM_THREADS 2

int shared_variable = 0;
sem_t semaphore;

void *increment_shared_variable(void *thread_id) {
    int tid = *(int *)thread_id;

    for (int i = 0; i < 1000000; i++) {
        sem_wait(&semaphore); // Wait until the semaphore is available
        shared_variable++;
        sem_post(&semaphore); // Release the semaphore
    }

    printf("Thread %d finished. Shared variable: %d\n", tid, shared_variable);

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    sem_init(&semaphore, 0, 1); // Initialize the semaphore with an initial value of 1

    for (int t = 0; t < NUM_THREADS; t++) {
        thread_ids[t] = t;
        pthread_create(&threads[t], NULL, increment_shared_variable, &thread_ids[t]);
    }

    for (int t = 0; t < NUM_THREADS; t++) {
```



```
    for (int t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }

    sem_destroy(&semaphore);

    printf("Final shared variable value: %d\n", shared_variable);

    return 0;
}
```

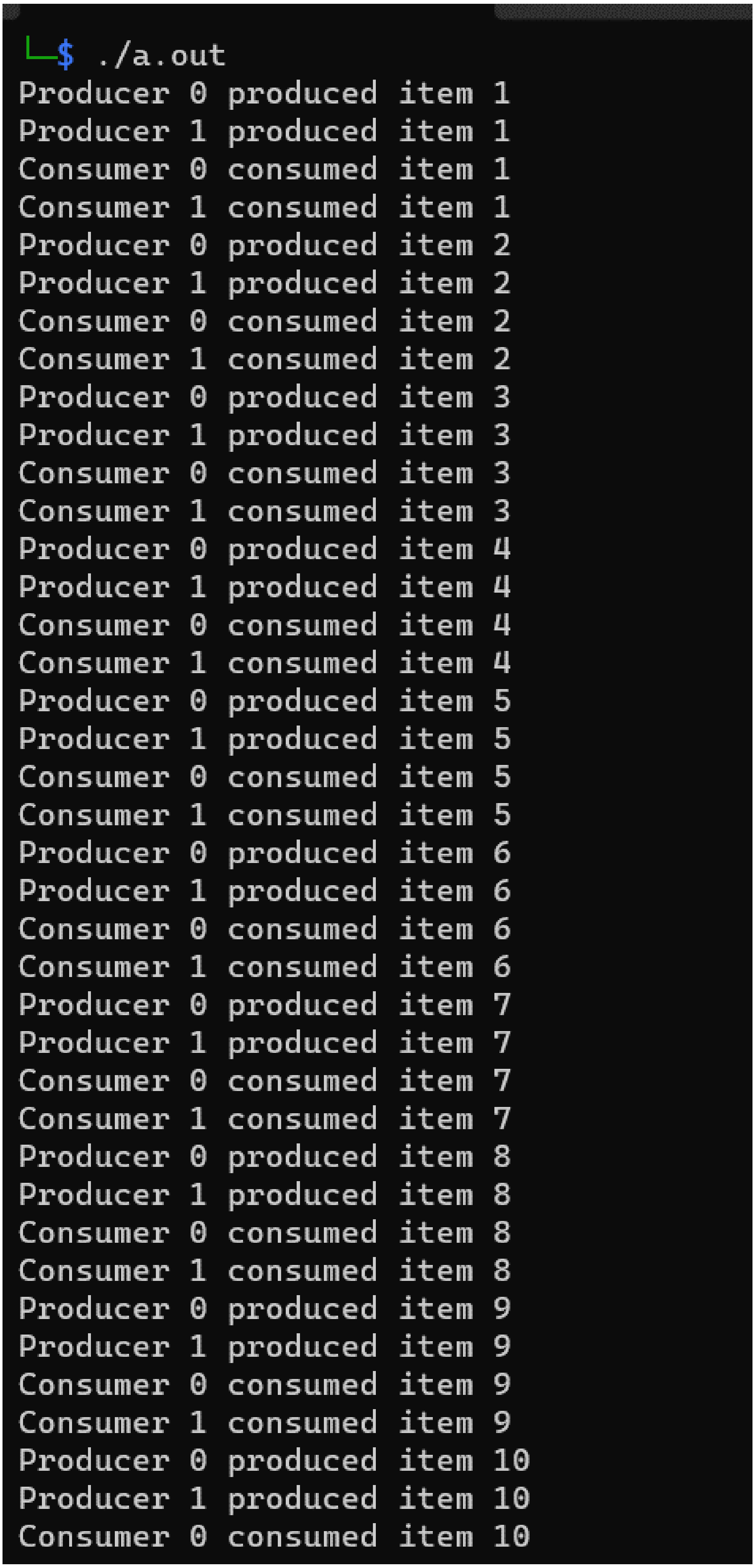
3. Write a C program to create two processes that implement a producer-consumer model using semaphores to synchronize access to the shared buffer.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


#define BUFFER_SIZE 5

#define NUM_PRODUCERS 2

#define NUM_CONSUMERS 2


int buffer[BUFFER_SIZE];

sem_t empty, full;

pthread_mutex_t mutex;


void *producer(void *thread_id) {

  int tid = *(int *)thread_id;


  for (int i = 0; i < 10; i++) {

    int item = i + 1;


    sem_wait(&empty);

    pthread_mutex_lock(&mutex);


    buffer[i % BUFFER_SIZE] = item;

    printf("Producer %d produced item %d\n", tid, item);


    pthread_mutex_unlock(&mutex);

    sem_post(&full);


    sleep(1);

  }
```

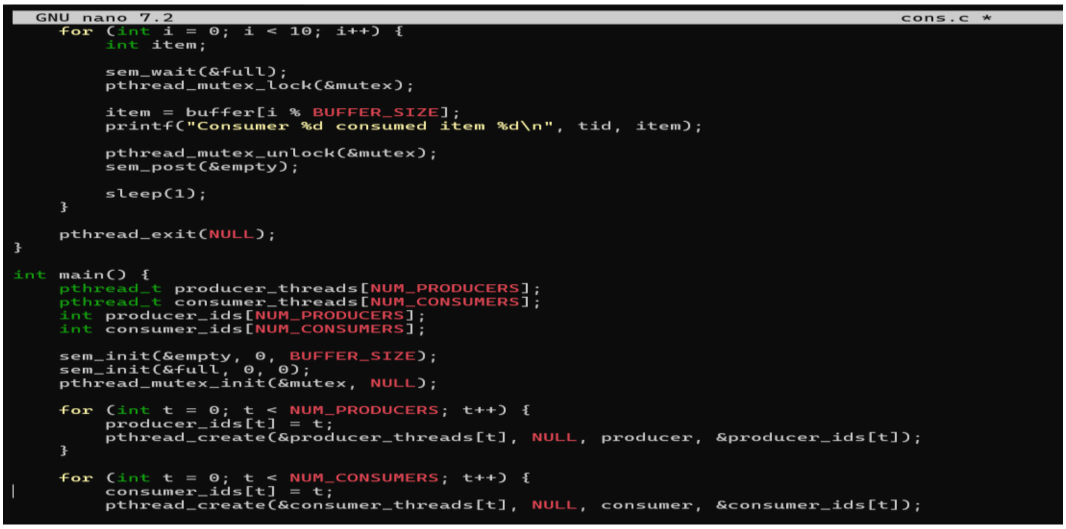
```
    pthread_exit(NULL);
}

void *consumer(void *thread_id) {
    int tid = *(int *)thread_id;

    for (int i = 0; i < 10; i++) {
        int item;

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

        item = buffer[i % BUFFER_SIZE];
        printf("Consumer %d consumed item %d\n", tid, item);

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);

        sleep(1);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t producer_threads[NUM_PRODUCERS];
    pthread_t consumer_threads[NUM_CONSUMERS];
    int producer_ids[NUM_PRODUCERS];
    int consumer_ids[NUM_CONSUMERS];

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

    for (int t = 0; t < NUM_PRODUCERS; t++) {
        producer_ids[t] = t;
        pthread_create(&producer_threads[t], NULL, producer, &producer_ids[t]);
    }

    for (int t = 0; t < NUM_CONSUMERS; t++) {
        consumer_ids[t] = t;
        pthread_create(&consumer_threads[t], NULL, consumer, &consumer_ids[t]);
    }

    for (int t = 0; t < NUM_PRODUCERS; t++) {
        pthread_join(producer_threads[t], NULL);
    }

    for (int t = 0; t < NUM_CONSUMERS; t++) {
        pthread_join(consumer_threads[t], NULL);
    }

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Output:

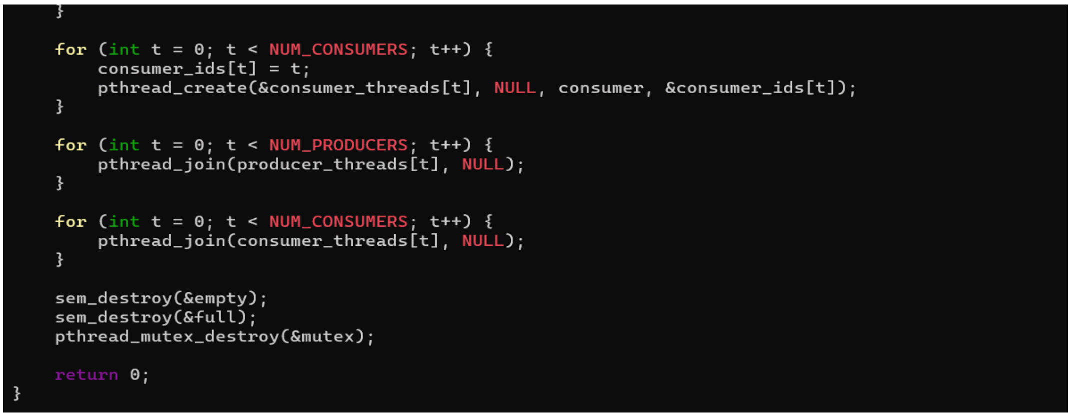


```
└─$ ./a.out
Producer 0 produced item 1
Producer 1 produced item 1
Consumer 0 consumed item 1
Consumer 1 consumed item 1
Producer 0 produced item 2
Producer 1 produced item 2
Consumer 0 consumed item 2
Consumer 1 consumed item 2
Producer 0 produced item 3
Producer 1 produced item 3
Consumer 0 consumed item 3
Consumer 1 consumed item 3
Producer 0 produced item 4
Producer 1 produced item 4
Consumer 0 consumed item 4
Consumer 1 consumed item 4
Producer 0 produced item 5
Producer 1 produced item 5
Consumer 0 consumed item 5
Consumer 1 consumed item 5
Producer 0 produced item 6
Producer 1 produced item 6
Consumer 0 consumed item 6
Consumer 1 consumed item 6
Producer 0 produced item 7
Producer 1 produced item 7
Consumer 0 consumed item 7
Consumer 1 consumed item 7
Producer 0 produced item 8
Producer 1 produced item 8
Consumer 0 consumed item 8
Consumer 1 consumed item 8
Producer 0 produced item 9
Producer 1 produced item 9
Consumer 0 consumed item 9
Consumer 1 consumed item 9
Producer 0 produced item 10
Producer 1 produced item 10
Consumer 0 consumed item 10
```

Code:



```
GNU nano 7.2                                                                  cons.c *
    for (int i = 0; i < 10; i++) {
        int item;

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

        item = buffer[i % BUFFER_SIZE];
        printf("Consumer %d consumed item %d\n", tid, item);

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);

        sleep(1);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t producer_threads[NUM_PRODUCERS];
    pthread_t consumer_threads[NUM_CONSUMERS];
    int producer_ids[NUM_PRODUCERS];
    int consumer_ids[NUM_CONSUMERS];

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

    for (int t = 0; t < NUM_PRODUCERS; t++) {
        producer_ids[t] = t;
        pthread_create(&producer_threads[t], NULL, producer, &producer_ids[t]);
    }

    for (int t = 0; t < NUM_CONSUMERS; t++) {
        consumer_ids[t] = t;
        pthread_create(&consumer_threads[t], NULL, consumer, &consumer_ids[t]);
```



```
    }

    for (int t = 0; t < NUM_CONSUMERS; t++) {
        consumer_ids[t] = t;
        pthread_create(&consumer_threads[t], NULL, consumer, &consumer_ids[t]);
    }

    for (int t = 0; t < NUM_PRODUCERS; t++) {
        pthread_join(producer_threads[t], NULL);
    }

    for (int t = 0; t < NUM_CONSUMERS; t++) {
        pthread_join(consumer_threads[t], NULL);
    }

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Submitted By:

Medhansh Alok

Varsha

Submitted to

Ashish sir