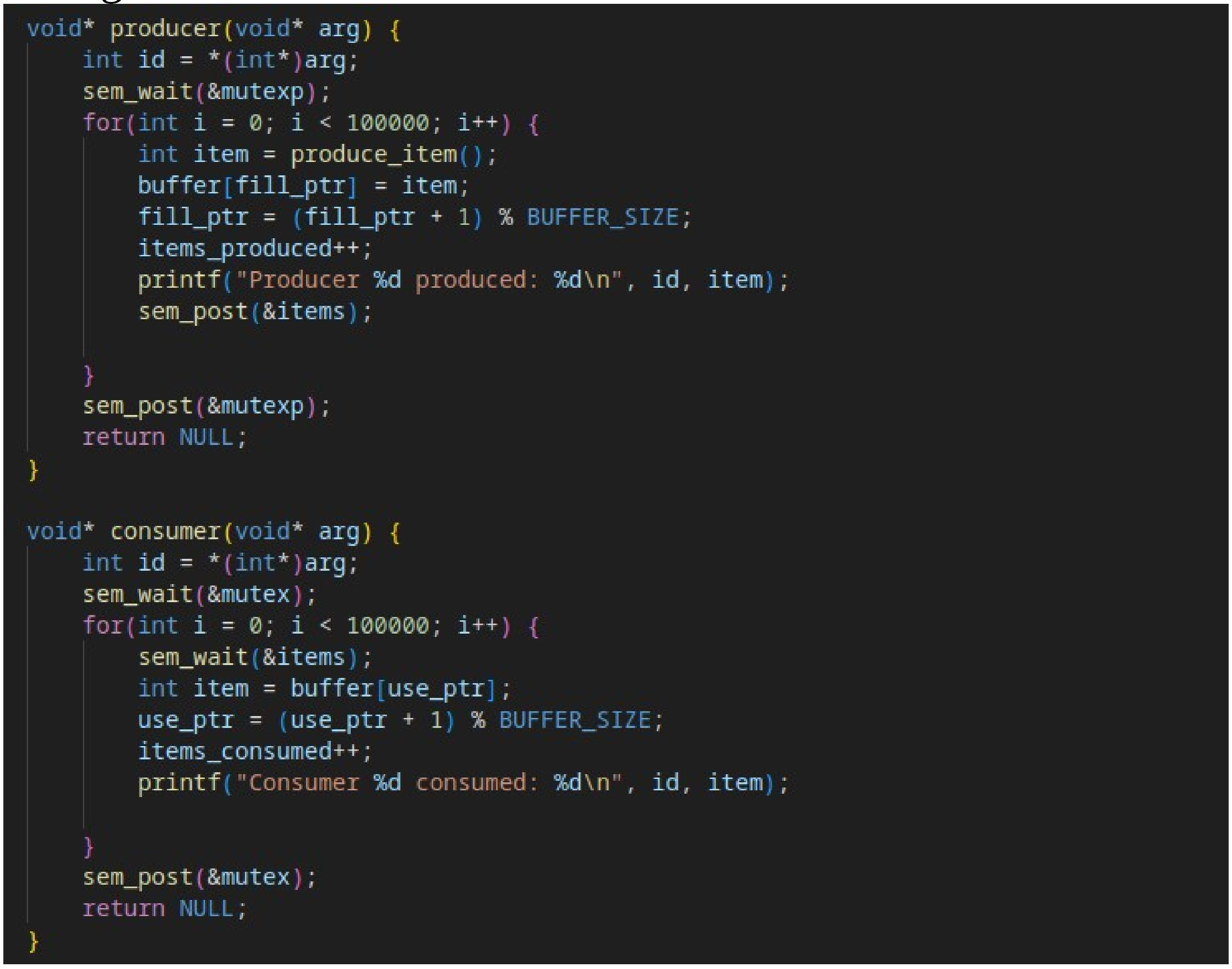
CLASS ASSIGNMENT 9
12340740

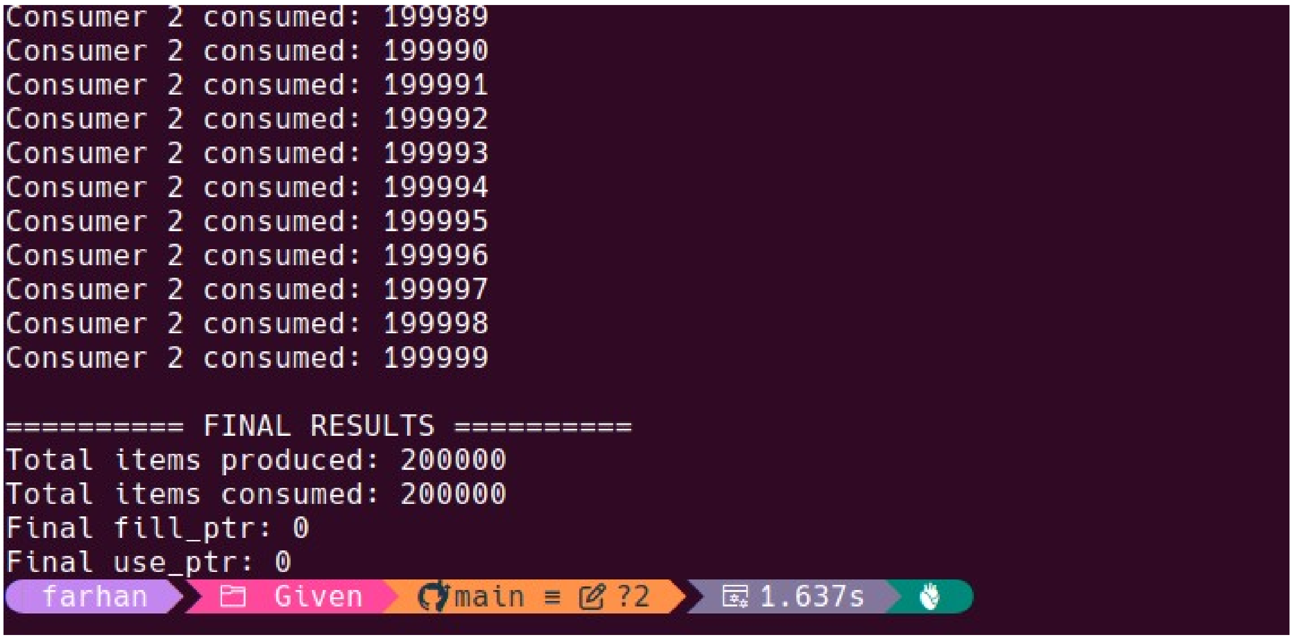
Question 1)

Modified pc_bug.c
 added mutex in both consumer and producer code before adding the mutex in both of them the consumer consumed more that the producer was producing .
Changes in code:-

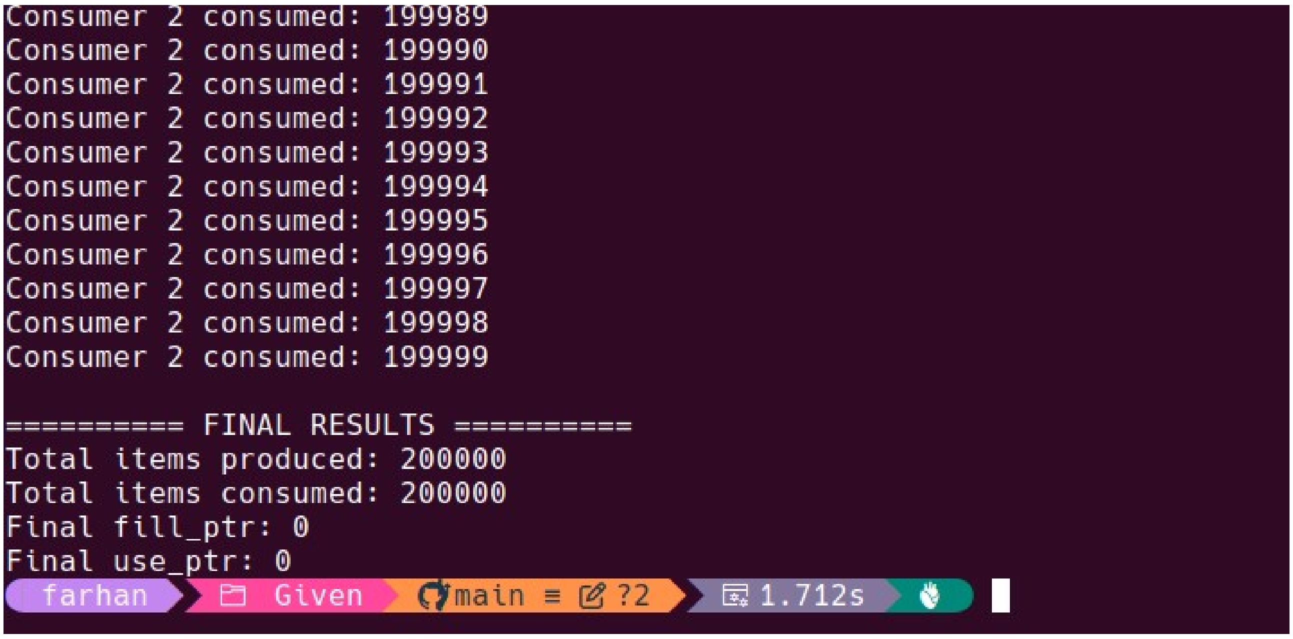


```
void* producer(void* arg) {
    int id = *(int*)arg;
    sem_wait(&mutexp);
    for(int i = 0; i < 100000; i++) {
        int item = produce_item();
        buffer[fill_ptr] = item;
        fill_ptr = (fill_ptr + 1) % BUFFER_SIZE;
        items_produced++;
        printf("Producer %d produced: %d\n", id, item);
        sem_post(&items);

    }
    sem_post(&mutexp);
    return NULL;
}

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

    }
    sem_post(&mutex);
    return NULL;
}
```

Output(after editing)

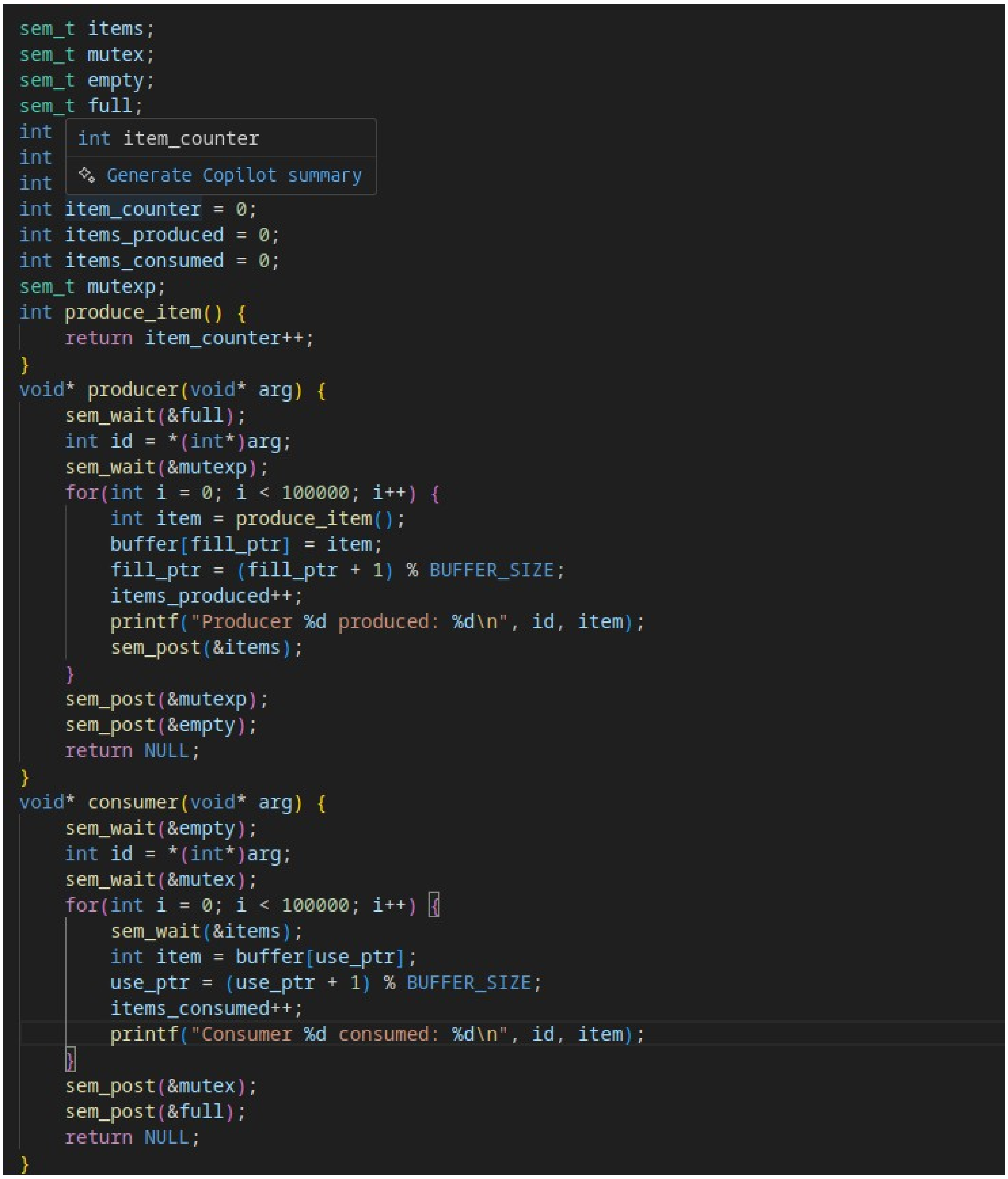


```
Consumer 2 consumed: 199989
Consumer 2 consumed: 199990
Consumer 2 consumed: 199991
Consumer 2 consumed: 199992
Consumer 2 consumed: 199993
Consumer 2 consumed: 199994
Consumer 2 consumed: 199995
Consumer 2 consumed: 199996
Consumer 2 consumed: 199997
Consumer 2 consumed: 199998
Consumer 2 consumed: 199999

========== FINAL RESULTS ==========
Total items produced: 200000
Total items consumed: 200000
Final fill_ptr: 0
Final use_ptr: 0
 farhan   Given   main ≡  ?2   1.637s
```



```
Consumer 2 consumed: 199989
Consumer 2 consumed: 199990
Consumer 2 consumed: 199991
Consumer 2 consumed: 199992
Consumer 2 consumed: 199993
Consumer 2 consumed: 199994
Consumer 2 consumed: 199995
Consumer 2 consumed: 199996
Consumer 2 consumed: 199997
Consumer 2 consumed: 199998
Consumer 2 consumed: 199999

========== FINAL RESULTS ==========
Total items produced: 200000
Total items consumed: 200000
Final fill_ptr: 0
Final use_ptr: 0
 farhan   Given   main ≡  ?2   1.712s
```

Question 2
For the items to be written consistently without the same items being accessed multiple times we introduce two semaphores full and empty where full tells how many spaces are left for producer to fill and empty tell how many items are there for consumers to read .Full is initialized to buffer size where as empty is initialized to 0 .The updated code is :-

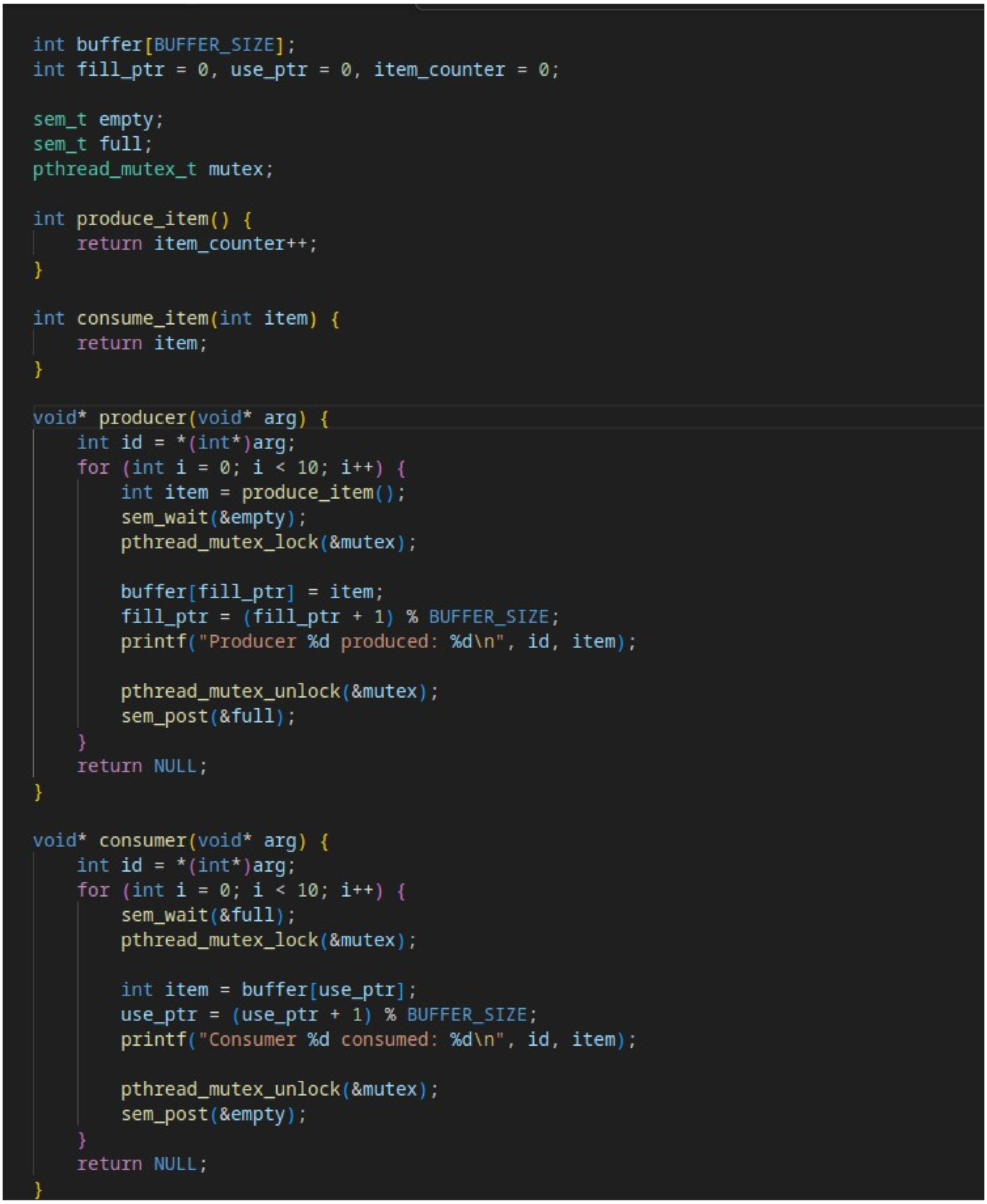


```c
sem_t items;
sem_t mutex;
sem_t empty;
sem_t full;
int
int
int
int item_counter = 0;
int items_produced = 0;
int items_consumed = 0;
sem_t mutexp;
int produce_item() {
    return item_counter++;
}
void* producer(void* arg) {
    sem_wait(&full);
    int id = *(int*)arg;
    sem_wait(&mutexp);
    for(int i = 0; i < 100000; i++) {
        int item = produce_item();
        buffer[fill_ptr] = item;
        fill_ptr = (fill_ptr + 1) % BUFFER_SIZE;
        items_produced++;
        printf("Producer %d produced: %d\n", id, item);
        sem_post(&items);
    }
    sem_post(&mutexp);
    sem_post(&empty);
    return NULL;
}
void* consumer(void* arg) {
    sem_wait(&empty);
    int id = *(int*)arg;
    sem_wait(&mutex);
    for(int i = 0; i < 100000; i++) {
        sem_wait(&items);
        int item = buffer[use_ptr];
        use_ptr = (use_ptr + 1) % BUFFER_SIZE;
        items_consumed++;
        printf("Consumer %d consumed: %d\n", id, item);
    }
    sem_post(&mutex);
    sem_post(&full);
    return NULL;
}
```

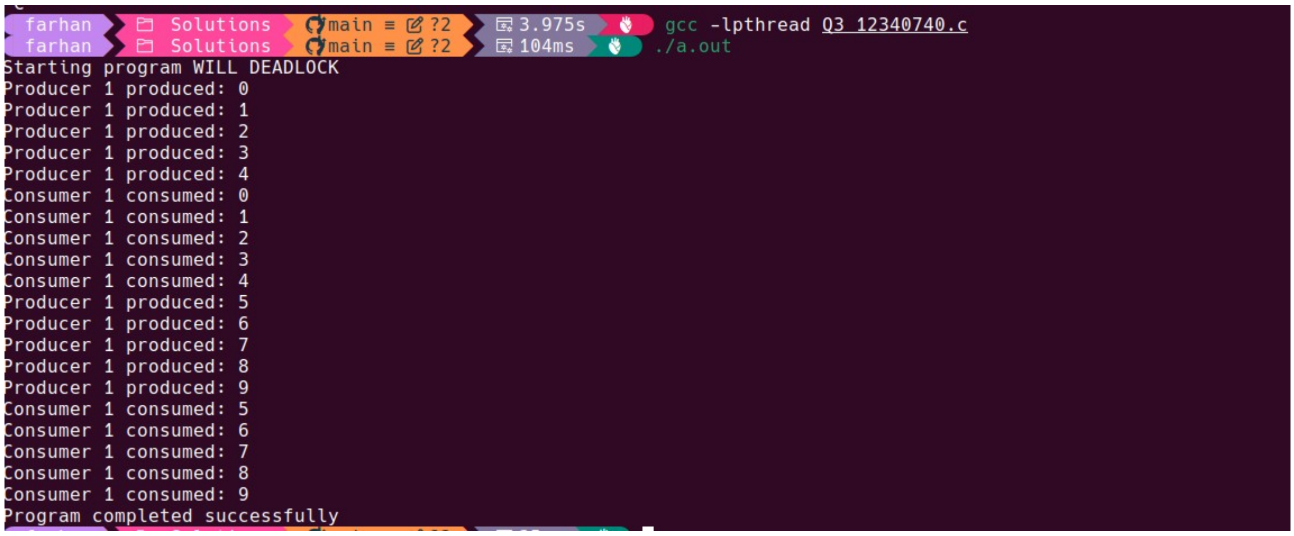
There would be no change in the output w.r.t the first question only the items accessed would differ.

Question 3

Instead of only using itmes used full and empty because use of a single semaphore was leading to deadlock .Same methodology as question no-2 has been sued code used is as follows:-

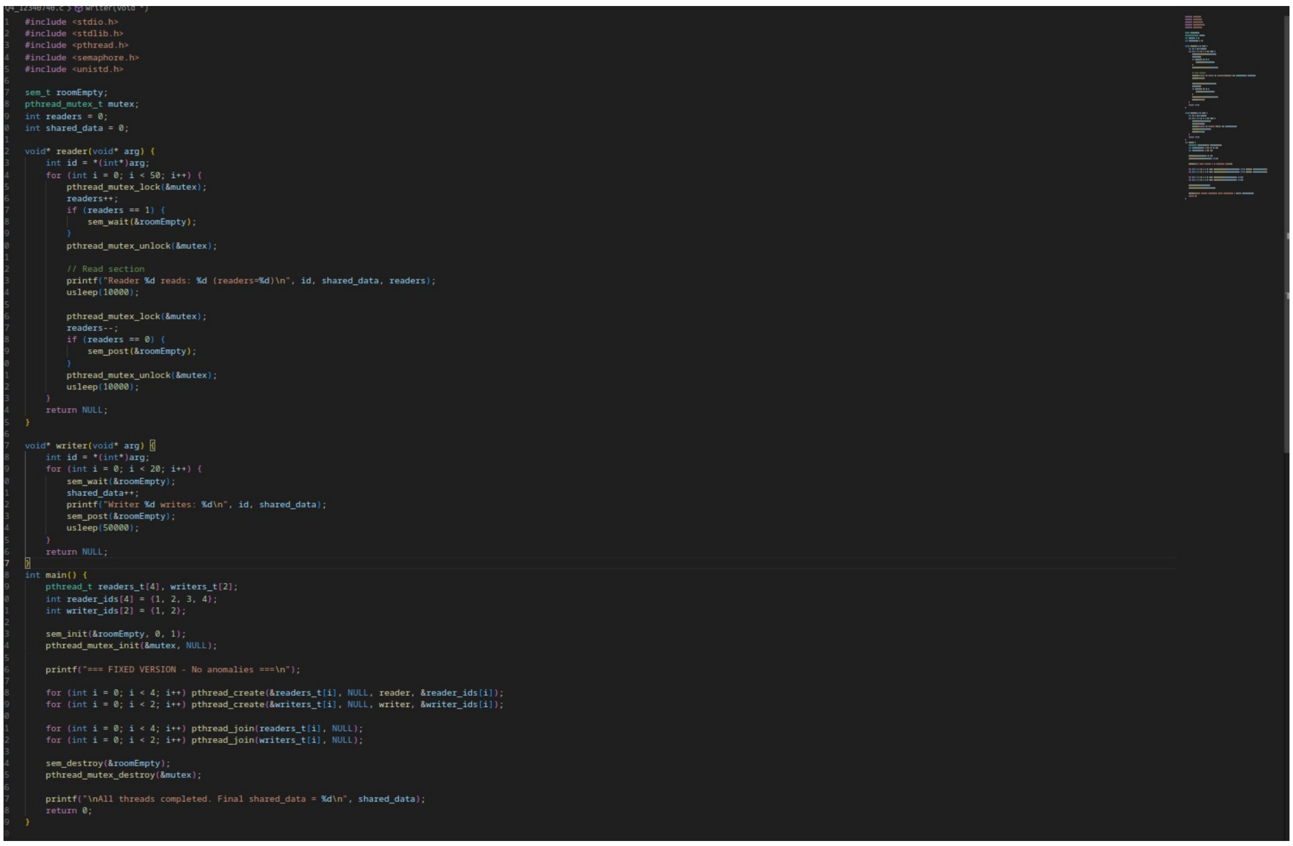

```c
int buffer[BUFFER_SIZE];
int fill_ptr = 0, use_ptr = 0, item_counter = 0;

sem_t empty;
sem_t full;
pthread_mutex_t mutex;

int produce_item() {
    return item_counter++;
}

int consume_item(int item) {
    return item;
}

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);
    }
    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);
    }
    return NULL;
}
```

Output obtained is as follows:-


```
 farhan  Solutions  main ≡ ?2  3.975s  gcc -lpthread Q3_12340740.c
 farhan  Solutions  main ≡ ?2  104ms  ./a.out
Starting program WILL DEADLOCK
Producer 1 produced: 0
Producer 1 produced: 1
Producer 1 produced: 2
Producer 1 produced: 3
Producer 1 produced: 4
Consumer 1 consumed: 0
Consumer 1 consumed: 1
Consumer 1 consumed: 2
Consumer 1 consumed: 3
Consumer 1 consumed: 4
Producer 1 produced: 5
Producer 1 produced: 6
Producer 1 produced: 7
Producer 1 produced: 8
Producer 1 produced: 9
Consumer 1 consumed: 5
Consumer 1 consumed: 6
Consumer 1 consumed: 7
Consumer 1 consumed: 8
Consumer 1 consumed: 9
Program completed successfully
```

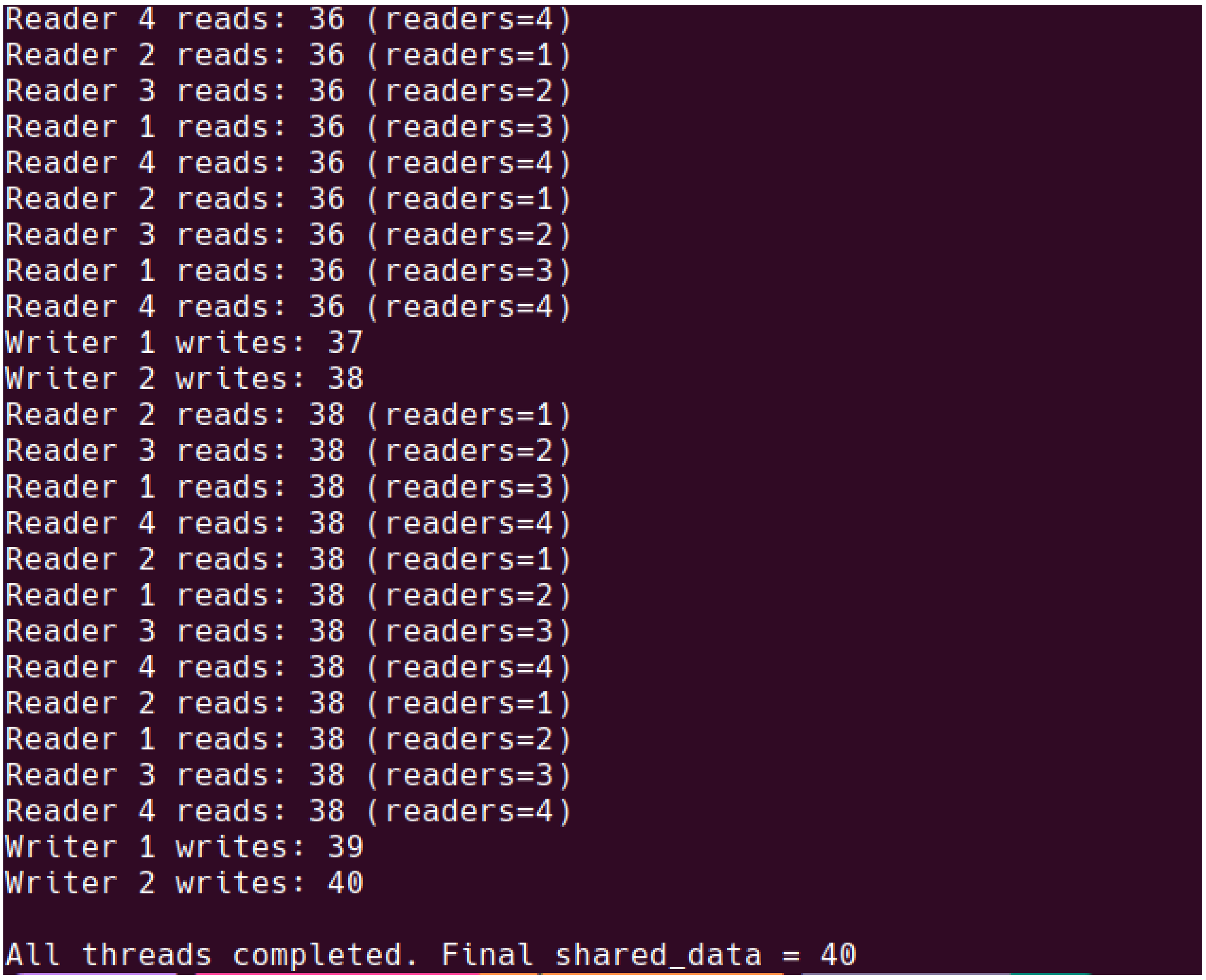
Question 4

The buggy readers–writers code failed because the `readers` counter was updated unsafely by multiple threads at once, causing race conditions. This was fixed by adding a `pthread_mutex_t` to protect all increments and decrements of `readers`. Additionally, only the **first reader** now locks the `roomEmpty` semaphore to block writers, and the **last reader** releases it, preventing concurrent `sem_wait()` calls. Using the same mutex for both increment and decrement ensures atomic updates, while the combination of the semaphore and mutex guarantees proper synchronization between readers and writers — avoiding data corruption and ensuring mutual exclusion.
The code used is as follows:-



```c
Q4_12340740.c > writer(void *)
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t roomEmpty;
pthread_mutex_t mutex;
int readers = 0;
int shared_data = 0;

void* reader(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 50; i++) {
        pthread_mutex_lock(&mutex);
        readers++;
        if (readers == 1) {
            sem_wait(&roomEmpty);
        }
        pthread_mutex_unlock(&mutex);

        // Read section
        printf("Reader %d reads: %d (readers=%d)\n", id, shared_data, readers);
        usleep(10000);

        pthread_mutex_lock(&mutex);
        readers--;
        if (readers == 0) {
            sem_post(&roomEmpty);
        }
        pthread_mutex_unlock(&mutex);
        usleep(10000);
    }
    return NULL;
}

void* writer(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 20; i++) {
        sem_wait(&roomEmpty);
        shared_data++;
        printf("Writer %d writes: %d\n", id, shared_data);
        sem_post(&roomEmpty);
        usleep(50000);
    }
    return NULL;
}
int main() {
    pthread_t readers_t[4], writers_t[2];
    int reader_ids[4] = {1, 2, 3, 4};
    int writer_ids[2] = {1, 2};

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

    printf("=== FIXED VERSION - No anomalies ===\n");

    for (int i = 0; i < 4; i++) pthread_create(&readers_t[i], NULL, reader, &reader_ids[i]);
    for (int i = 0; i < 2; i++) pthread_create(&writers_t[i], NULL, writer, &writer_ids[i]);

    for (int i = 0; i < 4; i++) pthread_join(readers_t[i], NULL);
    for (int i = 0; i < 2; i++) pthread_join(writers_t[i], NULL);

    sem_destroy(&roomEmpty);
    pthread_mutex_destroy(&mutex);

    printf("\nAll threads completed. Final shared_data = %d\n", shared_data);
    return 0;
}
```

The output is as follows:-

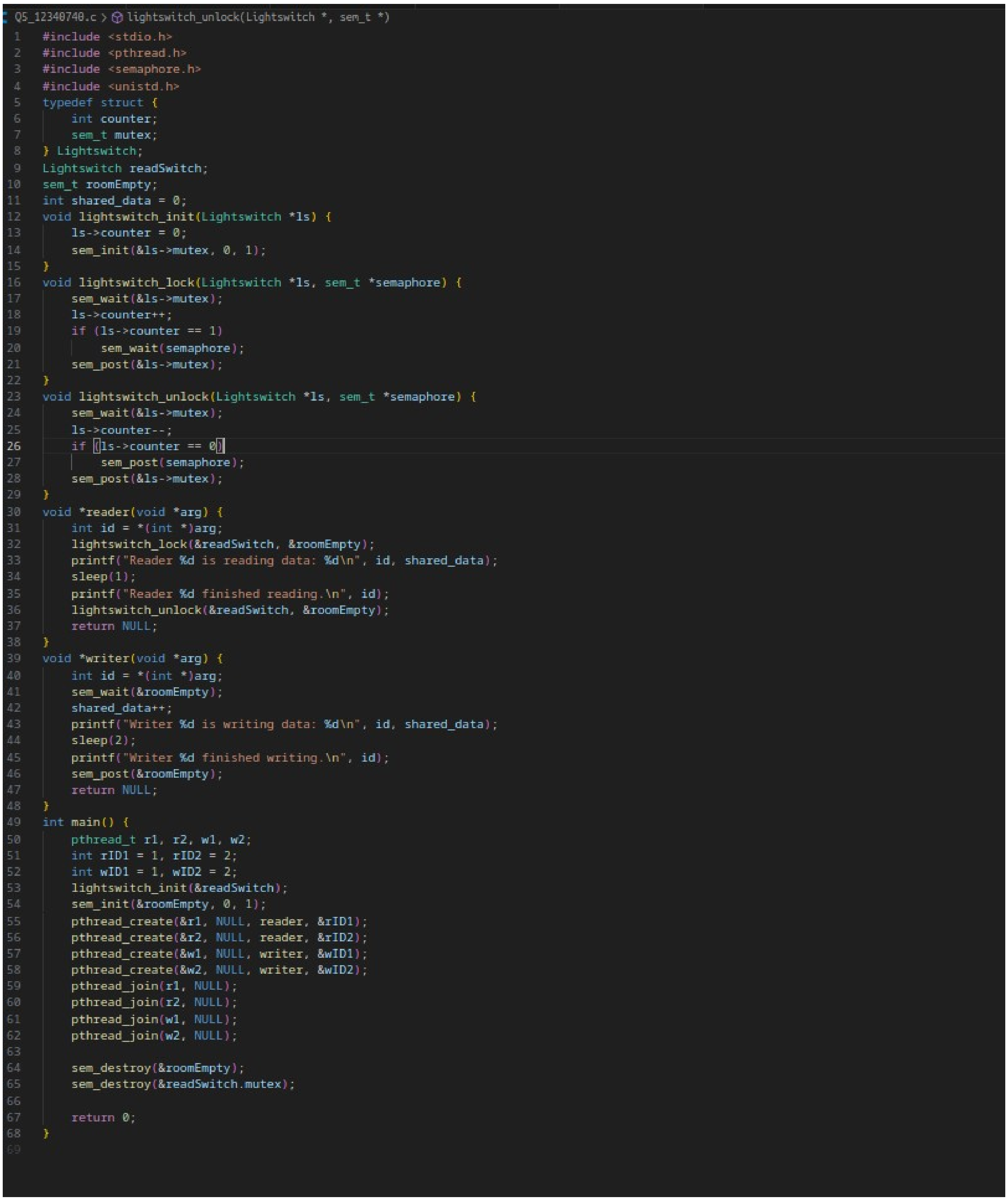


```
Reader 4 reads: 36 (readers=4)
Reader 2 reads: 36 (readers=1)
Reader 3 reads: 36 (readers=2)
Reader 1 reads: 36 (readers=3)
Reader 4 reads: 36 (readers=4)
Reader 2 reads: 36 (readers=1)
Reader 3 reads: 36 (readers=2)
Reader 1 reads: 36 (readers=3)
Reader 4 reads: 36 (readers=4)
Writer 1 writes: 37
Writer 2 writes: 38
Reader 2 reads: 38 (readers=1)
Reader 3 reads: 38 (readers=2)
Reader 1 reads: 38 (readers=3)
Reader 4 reads: 38 (readers=4)
Reader 2 reads: 38 (readers=1)
Reader 1 reads: 38 (readers=2)
Reader 3 reads: 38 (readers=3)
Reader 4 reads: 38 (readers=4)
Reader 2 reads: 38 (readers=1)
Reader 1 reads: 38 (readers=2)
Reader 3 reads: 38 (readers=3)
Reader 4 reads: 38 (readers=4)
Writer 1 writes: 39
Writer 2 writes: 40

All threads completed. Final shared_data = 40
```

Question 5)

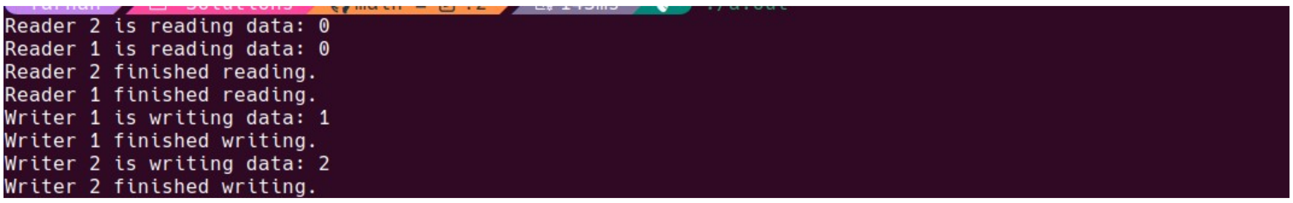
This program demonstrates a **correct implementation of the Readers–Writers problem** using the **Lightswitch pattern** and a **semaphore** for mutual exclusion. The `Lightswitch` structure allows multiple readers to access the shared resource concurrently, while writers still require exclusive access. When the **first reader** enters, it locks the `roomEmpty` semaphore, preventing writers from entering. Subsequent readers can enter freely without blocking one another. When the **last reader** finishes, it releases the semaphore, allowing writers to proceed. Writers, on the other hand, always perform a `sem_wait(&roomEmpty)` before writing, ensuring that no reader or writer is active at the same time. This approach avoids race conditions, ensures data consistency, and provides efficient synchronization between multiple readers and writers without causing deadlocks.

The code is as follows:-



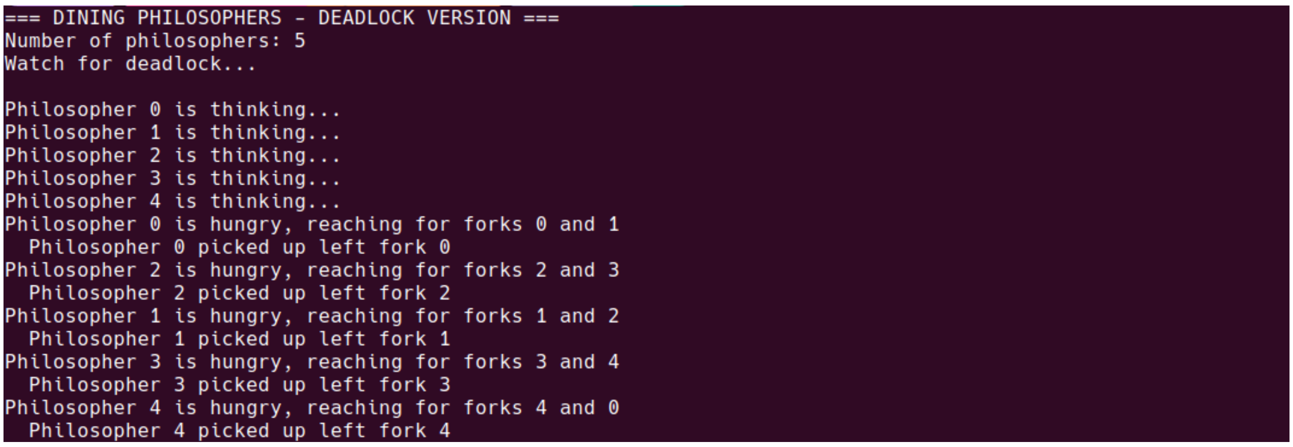
```
Q5_12340740.c > lightswitch_unlock(Lightswitch *, sem_t *)
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
typedef struct {
    int counter;
    sem_t mutex;
} Lightswitch;
Lightswitch readSwitch;
sem_t roomEmpty;
int shared_data = 0;
void lightswitch_init(Lightswitch *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) {
    int id = *(int *)arg;
    lightswitch_lock(&readSwitch, &roomEmpty);
    printf("Reader %d is reading data: %d\n", id, shared_data);
    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);
    shared_data++;
    printf("Writer %d is writing data: %d\n", id, shared_data);
    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;
}
```

The output is as follows:-



```
Reader 2 is reading data: 0
Reader 1 is reading data: 0
Reader 2 finished reading.
Reader 1 finished reading.
Writer 1 is writing data: 1
Writer 1 finished writing.
Writer 2 is writing data: 2
Writer 2 finished writing.
```

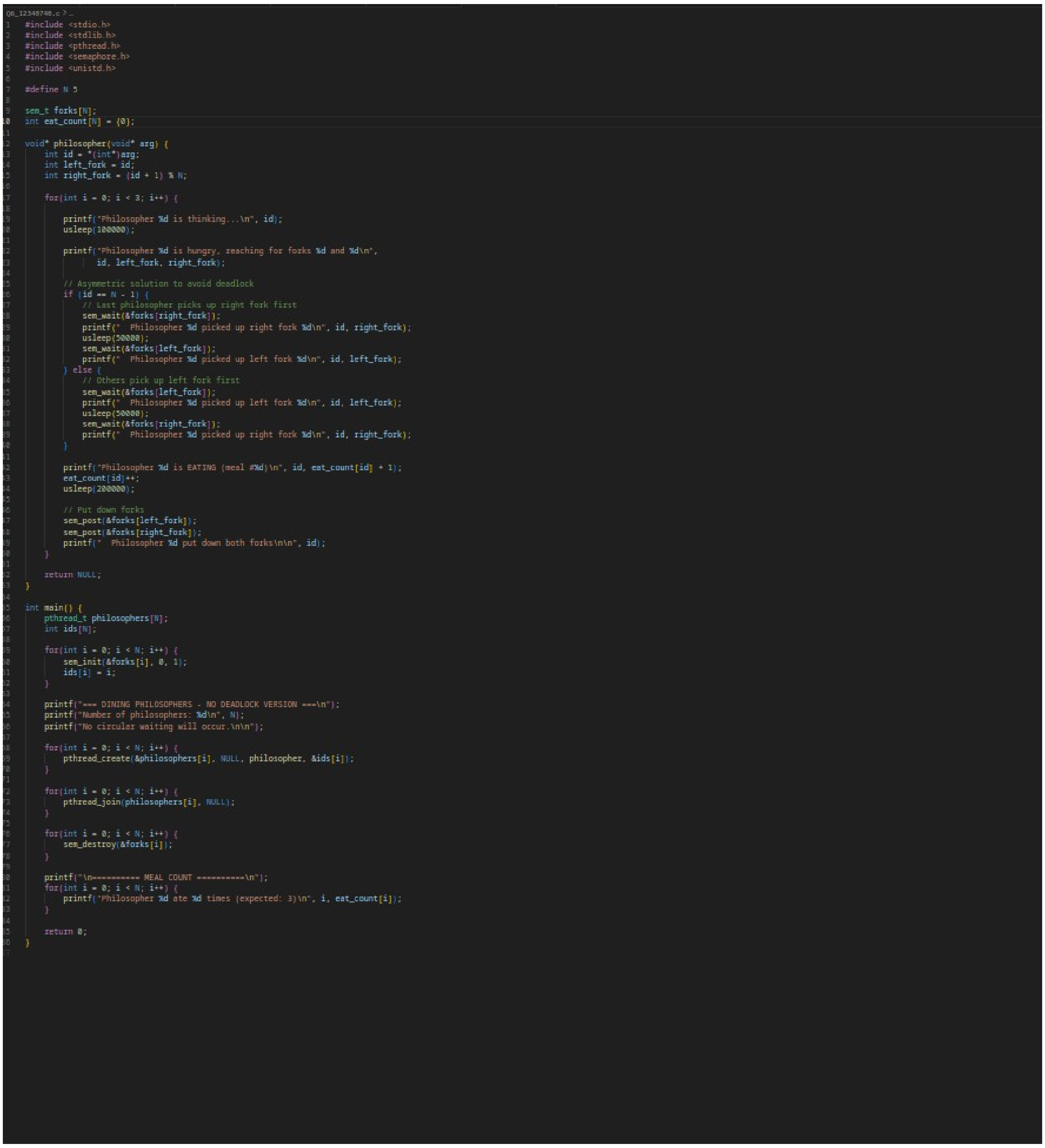
Qusetion 6
 given code dp_deadlock.c
output



```
=== DINING PHILOSOPHERS - DEADLOCK VERSION ===
Number of philosophers: 5
Watch for deadlock...

Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 4 is thinking...
Philosopher 0 is hungry, reaching for forks 0 and 1
  Philosopher 0 picked up left fork 0
Philosopher 2 is hungry, reaching for forks 2 and 3
  Philosopher 2 picked up left fork 2
Philosopher 1 is hungry, reaching for forks 1 and 2
  Philosopher 1 picked up left fork 1
Philosopher 3 is hungry, reaching for forks 3 and 4
  Philosopher 3 picked up left fork 3
Philosopher 4 is hungry, reaching for forks 4 and 0
  Philosopher 4 picked up left fork 4
```

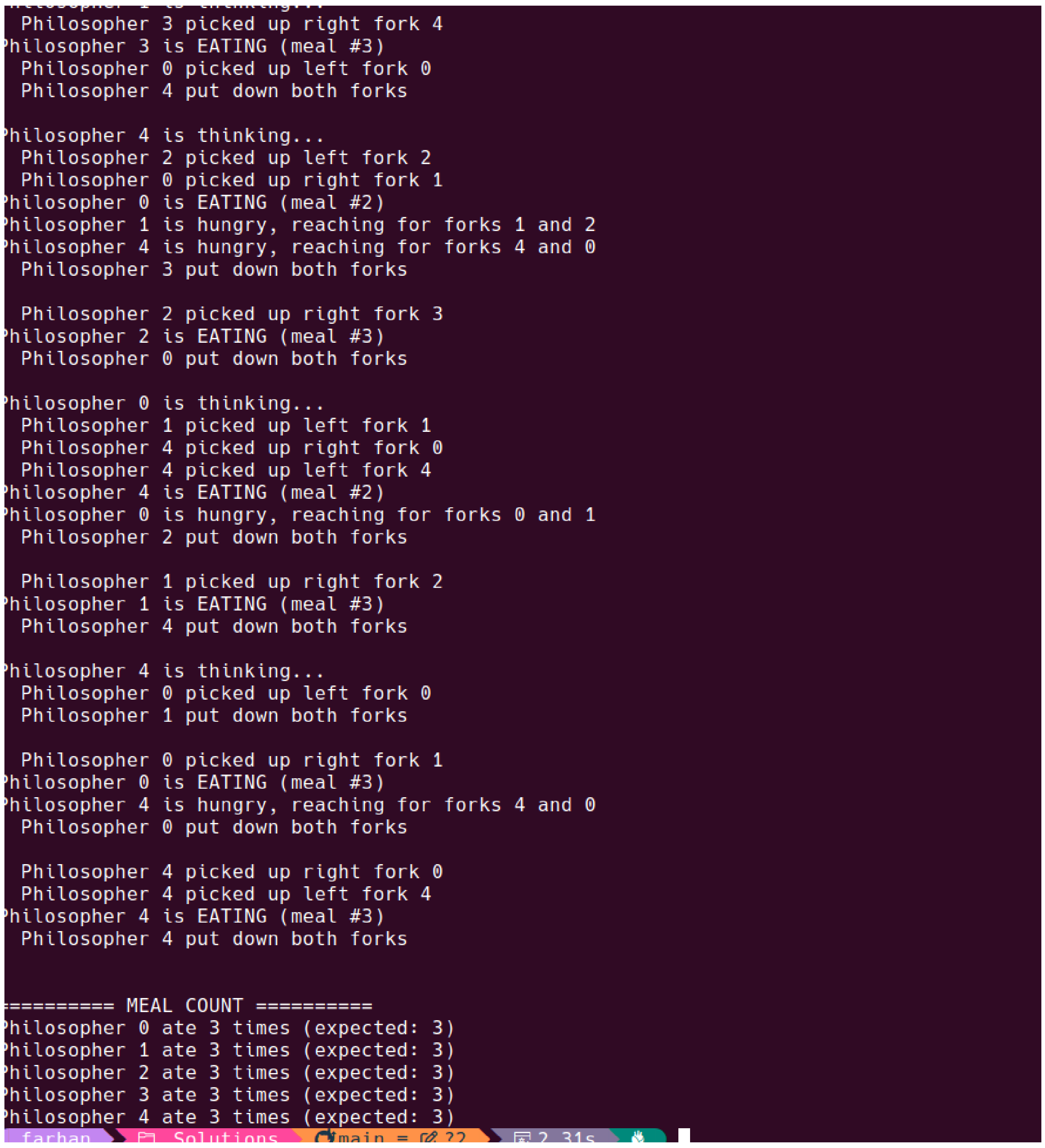
In the modified version of the Dining Philosophers program, the main change made to prevent deadlock is the introduction of an **asymmetric fork-picking strategy**. In the original version, all philosophers attempted to pick up their left fork first and then the right fork, which could lead to a circular wait — a situation where each philosopher holds one fork and waits indefinitely for the other. To break this circular dependency, the updated version makes one philosopher (specifically the last one) pick up the **right fork first** and then the left fork, while all others continue to pick up the left fork first. This simple change removes the possibility of a circular wait, thereby preventing deadlock while maintaining fairness in eating. The rest of the program logic, including initialization, eating behavior, and

cleanup, remains the same.
The code used is as follows:-



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

    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);

        // Asymmetric solution to avoid deadlock
        if (id == N - 1) {
            // Last philosopher picks up right fork first
            sem_wait(&forks[right_fork]);
            printf("  Philosopher %d picked up right fork %d\n", id, right_fork);
            usleep(50000);
            sem_wait(&forks[left_fork]);
            printf("  Philosopher %d picked up left fork %d\n", id, left_fork);
        } else {
            // Others pick up left fork first
            sem_wait(&forks[left_fork]);
            printf("  Philosopher %d picked up left fork %d\n", id, left_fork);
            usleep(50000);
            sem_wait(&forks[right_fork]);
            printf("  Philosopher %d picked up right fork %d\n", id, right_fork);
        }

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

        // Put down forks
        sem_post(&forks[left_fork]);
        sem_post(&forks[right_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 - NO DEADLOCK VERSION ===\n");
    printf("Number of philosophers: %d\n", N);
    printf("No circular waiting will occur.\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;
}
```

the output is as follows:-



```
Philosopher 1 is thinking...
  Philosopher 3 picked up right fork 4
Philosopher 3 is EATING (meal #3)
  Philosopher 0 picked up left fork 0
  Philosopher 4 put down both forks

Philosopher 4 is thinking...
  Philosopher 2 picked up left fork 2
  Philosopher 0 picked up right fork 1
Philosopher 0 is EATING (meal #2)
Philosopher 1 is hungry, reaching for forks 1 and 2
Philosopher 4 is hungry, reaching for forks 4 and 0
  Philosopher 3 put down both forks

  Philosopher 2 picked up right fork 3
Philosopher 2 is EATING (meal #3)
  Philosopher 0 put down both forks

Philosopher 0 is thinking...
  Philosopher 1 picked up left fork 1
  Philosopher 4 picked up right fork 0
  Philosopher 4 picked up left fork 4
Philosopher 4 is EATING (meal #2)
Philosopher 0 is hungry, reaching for forks 0 and 1
  Philosopher 2 put down both forks

  Philosopher 1 picked up right fork 2
Philosopher 1 is EATING (meal #3)
  Philosopher 4 put down both forks

Philosopher 4 is thinking...
  Philosopher 0 picked up left fork 0
  Philosopher 1 put down both forks

  Philosopher 0 picked up right fork 1
Philosopher 0 is EATING (meal #3)
Philosopher 4 is hungry, reaching for forks 4 and 0
  Philosopher 0 put down both forks

  Philosopher 4 picked up right fork 0
  Philosopher 4 picked up left fork 4
Philosopher 4 is EATING (meal #3)
  Philosopher 4 put down both forks


========== 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)
  farhan    Solutions    main = ?2    2 31s
```