

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;

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);

        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);

        sem_post(&items);
    }
    return NULL;
}

void* consumer(void* arg) {
    int id = *(int*)arg;
    for(int i = 0; i < 100000; i++) {
        sem_wait(&items);

        pthread_mutex_lock(&mutex);

        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);
    }
    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);

    printf("Starting Producer-Consumer (FIXED 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);
    }

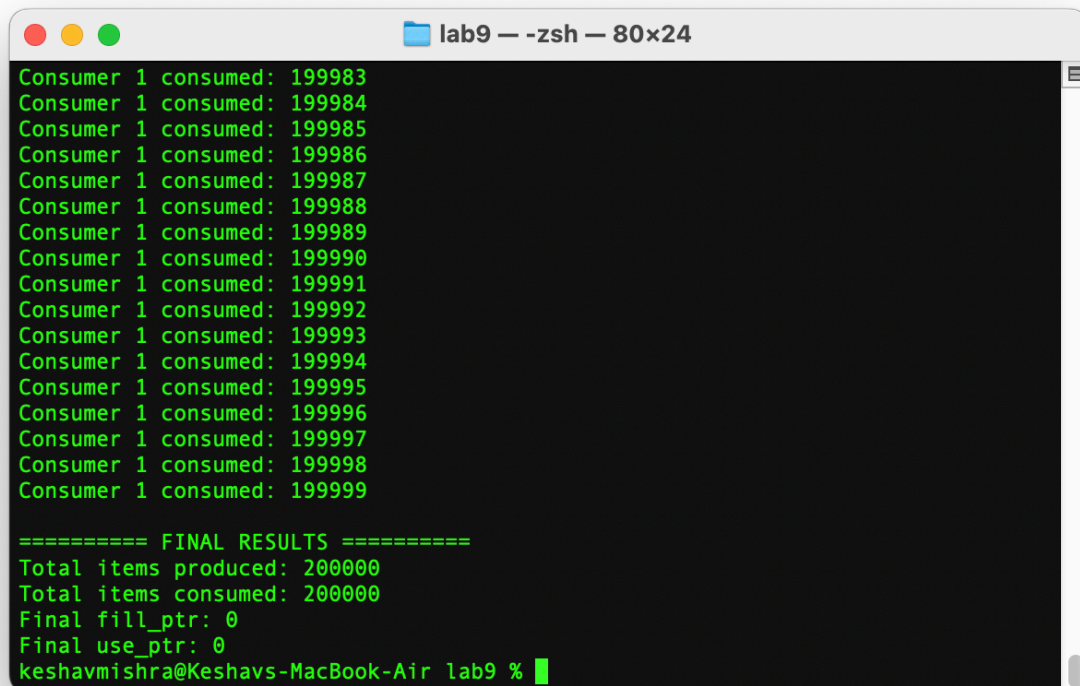
    sem_destroy(&items);
    pthread_mutex_destroy(&mutex);

    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;
}

```

Screenshot of output:



```
lab9 - -zsh - 80x24
Consumer 1 consumed: 199983
Consumer 1 consumed: 199984
Consumer 1 consumed: 199985
Consumer 1 consumed: 199986
Consumer 1 consumed: 199987
Consumer 1 consumed: 199988
Consumer 1 consumed: 199989
Consumer 1 consumed: 199990
Consumer 1 consumed: 199991
Consumer 1 consumed: 199992
Consumer 1 consumed: 199993
Consumer 1 consumed: 199994
Consumer 1 consumed: 199995
Consumer 1 consumed: 199996
Consumer 1 consumed: 199997
Consumer 1 consumed: 199998
Consumer 1 consumed: 199999

===== FINAL RESULTS =====
Total items produced: 200000
Total items consumed: 200000
Final fill_ptr: 0
Final use_ptr: 0
keshavmishra@Keshavs-MacBook-Air lab9 %
```

Explanation:

The race condition was caused by multiple threads simultaneously modifying shared state variables like `items_produced`, `fill_ptr`, and `use_ptr` with non-atomic operations, leading to lost updates and data corruption. This was fixed by introducing a mutex(`pthread_mutex_t`) to enforce mutual exclusion, ensuring only one thread can access and modify the critical section's shared data at any given time. The semaphore only handled thread signaling (item availability), not mutual exclusion (data protection).

Q2:

Code:

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

#define BUFFER_SIZE 100

sem_t items;
sem_t spaces;
pthread_mutex_t mutex;

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(&spaces);

        pthread_mutex_lock(&mutex);

        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);

        sem_post(&items);
    }
    return NULL;
}

void* consumer(void* arg) {
    int id = *(int*)arg;
    for(int i = 0; i < 100000; i++) {
        sem_wait(&items);

        pthread_mutex_lock(&mutex);

        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);

        sem_post(&spaces);
    }
    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);
sem_init(&spaces, 0, BUFFER_SIZE);
pthread_mutex_init(&mutex, NULL);

printf("Starting Producer-Consumer (BOUNDED BUFFER FIXED 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);
}

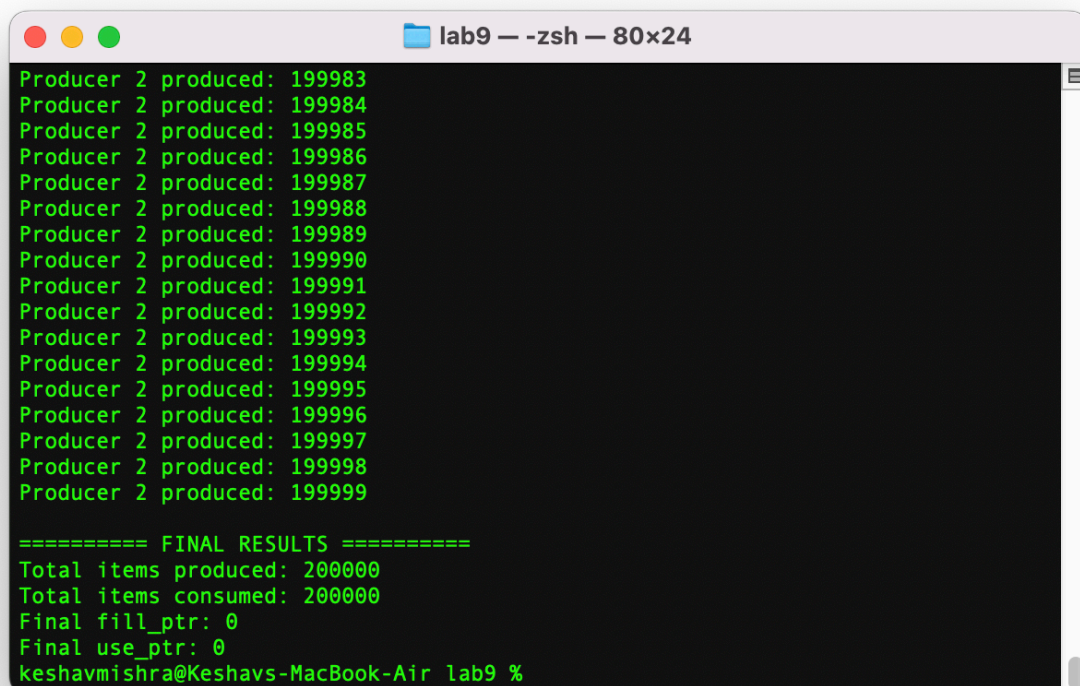
sem_destroy(&items);
sem_destroy(&spaces);
pthread_mutex_destroy(&mutex);

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;
}

```

Screenshot of output:



```

lab9 - -zsh - 80x24
Producer 2 produced: 199983
Producer 2 produced: 199984
Producer 2 produced: 199985
Producer 2 produced: 199986
Producer 2 produced: 199987
Producer 2 produced: 199988
Producer 2 produced: 199989
Producer 2 produced: 199990
Producer 2 produced: 199991
Producer 2 produced: 199992
Producer 2 produced: 199993
Producer 2 produced: 199994
Producer 2 produced: 199995
Producer 2 produced: 199996
Producer 2 produced: 199997
Producer 2 produced: 199998
Producer 2 produced: 199999

===== FINAL RESULTS =====
Total items produced: 200000
Total items consumed: 200000
Final fill_ptr: 0
Final use_ptr: 0
keshavmishra@Keshavs-MacBook-Air lab9 %

```

Explanation:

The bounded buffer solution uses the mutex to prevent data races on shared variables (mutual exclusion). It uses the items semaphore (full slots) to block consumers when the buffer is empty, and the spaces semaphore (empty slots) to block producers when the buffer is full, ensuring the finite capacity constraint.

Q3:

Code:

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

#define BUFFER_SIZE 5

sem_t items;
sem_t spaces;
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(&spaces);

        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(&items);
        usleep(100000);
    }
    return NULL;
}
```

```

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

        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(&spaces);
        usleep(150000);
    }
    return NULL;
}

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

    sem_init(&items, 0, 0);
    sem_init(&spaces, 0, BUFFER_SIZE);
    pthread_mutex_init(&mutex, NULL);

    printf("Starting code - FIXED VERSION\n");

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

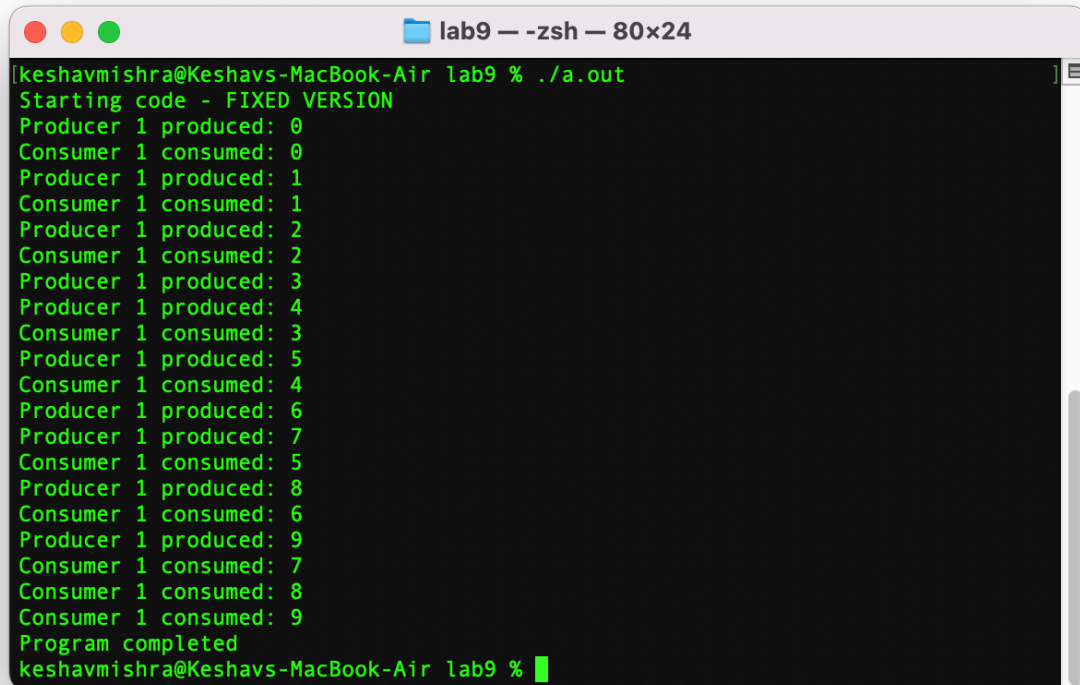
    pthread_join(prod1, NULL);
    pthread_join(cons1, NULL);

    sem_destroy(&items);
    sem_destroy(&spaces);
    pthread_mutex_destroy(&mutex);
    printf("Program completed\n");

    return 0;
}

```

Screenshot of output:



```
[keshavmishra@Keshavs-MacBook-Air lab9 % ./a.out]
Starting code - FIXED VERSION
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
keshavmishra@Keshavs-MacBook-Air lab9 %
```

Explanation:

The deadlock occurred because the producer acquired the mutex and then immediately blocked on the empty items semaphore, effectively holding the lock hostage. The consumer then tried to acquire the same mutex and blocked waiting for the producer, creating a circular dependency. The fix involves swapping the order: performing the blocking semaphore operations (`sem_wait`) outside the critical section protected by the mutex.

Q4:

Code:

```
#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;
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(&mutex);

        readers++;

        if(readers == 1) {
            sem_wait(&roomEmpty);
        }

        pthread_mutex_unlock(&mutex);

        int value = shared_data;
        read_count++;
        printf("Reader %d reads: %d (current readers=%d)\n", id, value, readers);
        usleep(100);

        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++;
        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(&mutex, NULL);

    printf("=== FIXED READERS-WRITERS VERSION ===\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++) {
        pthread_join(writer_threads[i], NULL);
    }

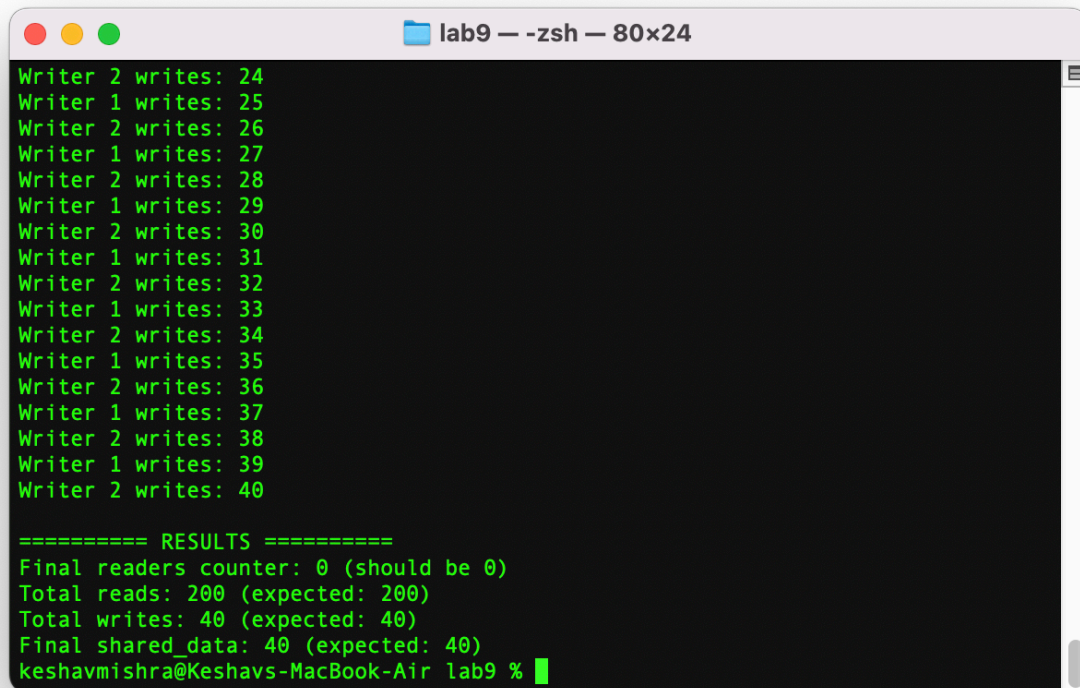
    sem_destroy(&roomEmpty);
    pthread_mutex_destroy(&mutex);

    printf("\n===== RESULTS =====\n");
    printf("Final readers counter: %d (should be 0)\n", readers);
    printf("Total reads: %d (expected: 200)\n", read_count);
    printf("Total writes: %d (expected: 40)\n", write_count);
    printf("Final shared_data: %d (expected: 40)\n", shared_data);

    return 0;
}

```

Screenshot of output:



```
lab9 - -zsh - 80x24
Writer 2 writes: 24
Writer 1 writes: 25
Writer 2 writes: 26
Writer 1 writes: 27
Writer 2 writes: 28
Writer 1 writes: 29
Writer 2 writes: 30
Writer 1 writes: 31
Writer 2 writes: 32
Writer 1 writes: 33
Writer 2 writes: 34
Writer 1 writes: 35
Writer 2 writes: 36
Writer 1 writes: 37
Writer 2 writes: 38
Writer 1 writes: 39
Writer 2 writes: 40

===== RESULTS =====
Final readers counter: 0 (should be 0)
Total reads: 200 (expected: 200)
Total writes: 40 (expected: 40)
Final shared_data: 40 (expected: 40)
keshavmishra@Keshavs-MacBook-Air lab9 %
```

Explanation:

The race condition was on the shared counter readers, where multiple readers could perform non-atomic Read-Modify-Write operations, leading to lost increments and a corrupted count. The fix was adding a mutex to protect the counter's access and modification, ensuring that the critical logic (checking if readers equals 1 or 0) correctly signals the roomEmpty semaphore.

Q5:

Code:

```
#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) {
    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;
}

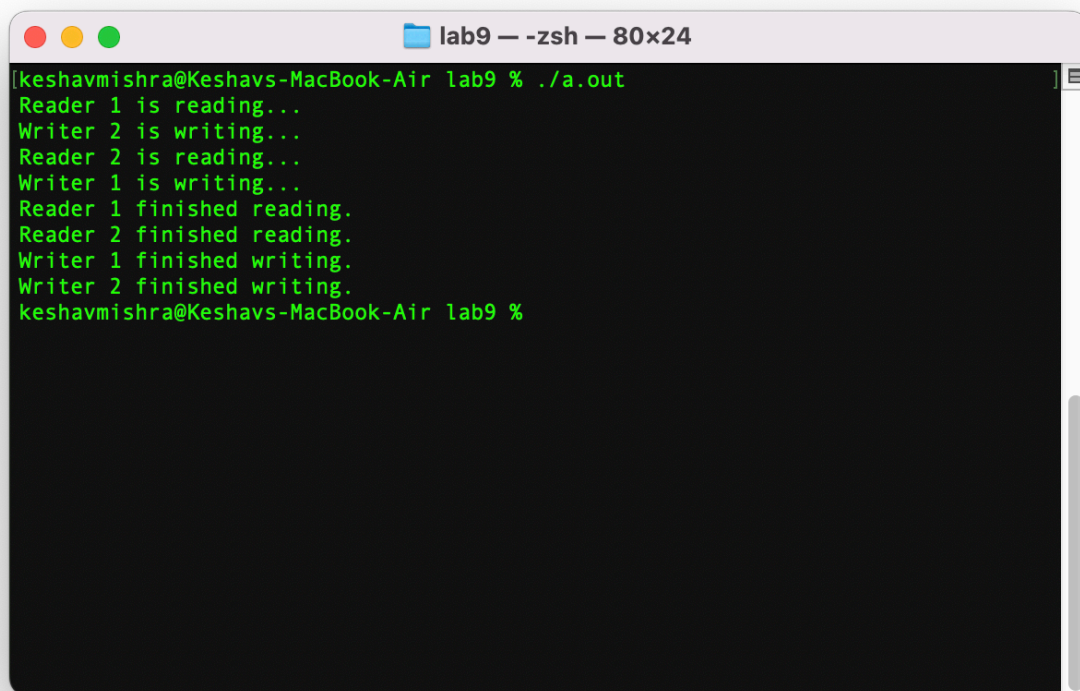
void *writer(void *arg) {
    int id = *(int *)arg;
    sem_wait(&roomEmpty);
    printf("Writer %d is writing...\n", id);
    sleep(2);
    printf("Writer %d finished writing.\n", id);
    sem_post(&roomEmpty);
    return NULL;
}

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

Screenshot of output:



```
lab9 - -zsh - 80x24
[keshavmishra@Keshavs-MacBook-Air lab9 % ./a.out
Reader 1 is reading...
Writer 2 is writing...
Reader 2 is reading...
Writer 1 is writing...
Reader 1 finished reading.
Reader 2 finished reading.
Writer 1 finished writing.
Writer 2 finished writing.
keshavmishra@Keshavs-MacBook-Air lab9 %
```

Explanation:

The program correctly implements the Readers-Writers problem using the Lightswitch pattern, allowing multiple readers to access the shared resource concurrently while ensuring writers have exclusive access. The first reader locks out writers, and the last reader releases them.

Q6:

Code:

```
#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);

        if (left_fork < right_fork) {
            sem_wait(&forks[left_fork]);
            printf("  Philosopher %d picked up fork %d\n", id, left_fork);
            sem_wait(&forks[right_fork]);
            printf("  Philosopher %d picked up fork %d\n", id, right_fork);
        } else {
            sem_wait(&forks[right_fork]);
            printf("  Philosopher %d picked up fork %d\n", id, right_fork);
            sem_wait(&forks[left_fork]);
            printf("  Philosopher %d picked up fork %d\n", id, left_fork);
        }

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

        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 - FIXED (LOWER ID 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;
}

```

Screenshot of output:

```
Philosopher 3 is hungry, reaching for forks 3 and 4
Philosopher 3 picked up fork 3
Philosopher 3 picked up fork 4
Philosopher 3 is EATING (meal #2)
Philosopher 2 picked up fork 3
Philosopher 2 is EATING (meal #2)
Philosopher 4 is hungry, reaching for forks 4 and 0
Philosopher 4 picked up fork 0
Philosopher 4 picked up fork 4
Philosopher 4 is EATING (meal #2)
Philosopher 0 put down both forks

Philosopher 0 is thinking...
Philosopher 1 put down both forks

Philosopher 1 is thinking...
Philosopher 3 put down both forks

Philosopher 3 is thinking...
Philosopher 2 put down both forks

Philosopher 2 is thinking...
Philosopher 4 put down both forks

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

Philosopher 1 put down both forks

Philosopher 0 put down both forks

Philosopher 2 put down both forks

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)
```

Explanation:

Deadlock occurs because all philosophers try to pick up their left fork first, causing circular wait. The fix uses the Lower ID First strategy, where each philosopher picks up the fork with the smaller ID first, breaking the circular wait and preventing deadlock.