CSL301
12340220

Assignment: CA9

Question 1. Identify Race Condition and Fix with Mutex

```c
#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 shared data

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++;
        pthread_mutex_unlock(&mutex);

        sem_post(&items);
        printf("Producer %d produced: %d\n", id, item);
    }
    return NULL;
}
```

```c
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++;
        pthread_mutex_unlock(&mutex);

        printf("Consumer %d consumed: %d\n", id, item);
    }
    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;
```

```
}
```

Output:

Buggy and Fixed

```
========== FINAL RESULTS ==========
Total items produced: 199994
Total items consumed: 199999
Final fill_ptr: 94
Final use_ptr: 99
```

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

Explanation:

In the original code, shared variables like buffer, fill_ptr, use_ptr, items_produced and items_consumed were accessed by multiple threads simultaneously. Since there was no mutual exclusion, producers and consumers could modify these values at the same time, leading to data corruption and inconsistent results (Race Condition).

Question 2. Add finite buffer constant

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

#define BUFFER_SIZE 100

sem_t empty, full;
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 put(int item) {
    buffer[fill_ptr] = item;
    fill_ptr = (fill_ptr + 1) % BUFFER_SIZE;
    items_produced++;
}
```

```c
int get() {
    int item = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % BUFFER_SIZE;
    items_consumed++;
    return item;
}

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);
        put(item);
        pthread_mutex_unlock(&mutex);

        sem_post(&full);        // signal one more full slot
    }
    return NULL;
}

void* consumer(void* arg) {
    int id = *(int*)arg;
    for(int i = 0; i < 100000; i++) {
        sem_wait(&full);        // wait until an item exists

        pthread_mutex_lock(&mutex);
        int item = get();
        pthread_mutex_unlock(&mutex);

        sem_post(&empty);       // signal one more empty slot
    }
    return NULL;
}

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

    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty, 0, BUFFER_SIZE); // initially all slots empty
    sem_init(&full, 0, 0);            // initially no items

    printf("Starting Producer-Consumer (BOUNDED BUFFER VERSION)\n");
```

```c
    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);
    }

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

    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:

```
Starting Producer-Consumer (BOUNDED BUFFER VERSION)

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

Explanation:

The previous version treated the buffer as unbounded. Producers could overwrite items that were not yet consumed when the circular buffer wrapped around. This could cause lost or corrupted data.
Two semaphores were introduced to enforce bounded buffer behavior:

- empty → counts free slots in buffer
- full → counts items available to consume

Question 3. Identify and fix deadlock

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

```c
#define BUFFER_SIZE 5

sem_t full, empty;
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();

        // Wait for empty space
        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);
        // Signal item available
        sem_post(&full);

        usleep(100000);
    }
    return NULL;
}

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

        // Wait for item
        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);
        // Signal space available
        sem_post(&empty);

        usleep(150000);
    }
    return NULL;
}

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

    sem_init(&full, 0, 0);
    sem_init(&empty, 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(&full);
    sem_destroy(&empty);
    pthread_mutex_destroy(&mutex);

    printf("Program completed successfully\n");
    return 0;
}
```

Output:

```
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 successfully
```

Explanation:

The deadlock occurred because both the producer and consumer threads locked the mutex first and then called sem_wait(&items). When the semaphore value became zero, the thread executing sem_wait() blocked while still holding the mutex. The other thread could not proceed because it also required the mutex to update the buffer and post the semaphore. As a result, both threads waited forever and the program deadlocked.

Question 4. Readers-Writers Deadlock error

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

sem_t roomEmpty;
int readers = 0;
int shared_data = 0;
int read_count = 0;
int write_count = 0;
pthread_mutex_t readMutex;

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

        pthread_mutex_lock(&readMutex);
        readers++;
        if(readers == 1)
```

```
            sem_wait(&roomEmpty);  // block writers when first reader enters
        pthread_mutex_unlock(&readMutex);

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

        pthread_mutex_lock(&readMutex);
        readers--;
        if(readers == 0)
            sem_post(&roomEmpty);  // last reader leaves, allow writers
        pthread_mutex_unlock(&readMutex);

        usleep(10000);
    }
    return NULL;
}


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

        // Writing
        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(&readMutex, NULL);

    printf("=== BUGGY VERSION - Watch for anomalies ===\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++) {
        pthread_join(writer_threads[i], NULL);
    }

    sem_destroy(&roomEmpty);
    pthread_mutex_destroy(&readMutex);

    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:

Deadlock occurs because both producer and consumer lock the mutex before calling sem_wait(). If the buffer is empty or full, sem_wait() blocks, but since the mutex is still locked, the other thread cannot proceed, causing both threads to

wait forever. The fix is to use separate semaphores for tracking empty and full buffer spaces, and make sure sem_wait() is done outside the mutex lock so a thread never blocks while holding the mutex.

Question 5. Lightswitch

```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) {
```

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

    // Enter critical section for readers
    lightswitch_lock(&readSwitch, &roomEmpty);

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

    // Exit critical section for readers
    lightswitch_unlock(&readSwitch, &roomEmpty);

    return NULL;
}

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

    // Writer requires exclusive access
    sem_wait(&roomEmpty);

    printf("Writer %d is writing...\n", id);
    sleep(2);  // Simulate write operation
    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:

```
Reader 1 is reading...
Reader 2 is reading...
Reader 2 finished reading.
Reader 1 finished reading.
Writer 1 is writing...
Writer 1 finished writing.
Writer 2 is writing...
Writer 2 finished writing.
```

Explanation:

This solution uses the Lightswitch pattern to allow multiple readers to access the shared resource concurrently while ensuring that writers always get exclusive access. The Lightswitch ensures that only the first reader locks out writers by acquiring the roomEmpty semaphore, and only the last reader releases it, allowing writers to proceed. Writers simply wait on roomEmpty, guaranteeing mutual exclusion during write operations. This prevents data corruption and ensures correct reader–writer synchronization.

Question 5. Dining Philosophers

```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...\n", id);

        // Lower ID first strategy to prevent circular wait
        if (left_fork < right_fork) {
            sem_wait(&forks[left_fork]);
            printf("  Philosopher %d picked left fork %d\n", id, left_fork);
            sem_wait(&forks[right_fork]);
            printf("  Philosopher %d picked right fork %d\n", id, right_fork);
        } else {
            sem_wait(&forks[right_fork]);
            printf("  Philosopher %d picked right fork %d\n", id, right_fork);
            sem_wait(&forks[left_fork]);
            printf("  Philosopher %d picked left fork %d\n", id, left_fork);
        }

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

        // Release forks (order doesn't matter)
        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 - DEADLOCK VERSION ===\n");
    printf("Number of philosophers: %d\n", N);
```

```
    printf("Watch for deadlock...\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:

```
  Philosopher 1 put down both forks

  Philosopher 0 picked right fork 1
Philosopher 0 is EATING (meal #3)
  Philosopher 0 put down both forks

  Philosopher 4 picked right fork 0
  Philosopher 4 picked 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)
```

Explanation:

The original code deadlocks because every philosopher picks their left fork first. This creates a circular wait, where each one holds a left fork and waits forever for the right fork.
By always picking the fork with the lower ID first, we impose a strict ordering, the circular wait is broken, and deadlock is avoided.