Assignment No 2:

Name: Samruddhi Devram Khilari

Branch & Year: AIML-B SY

Prn: 12420020

Roll no: 2

Sub: Operating System Lab

1. **Q] Solve Dinning philosopher problem using mutex & semaphore.**

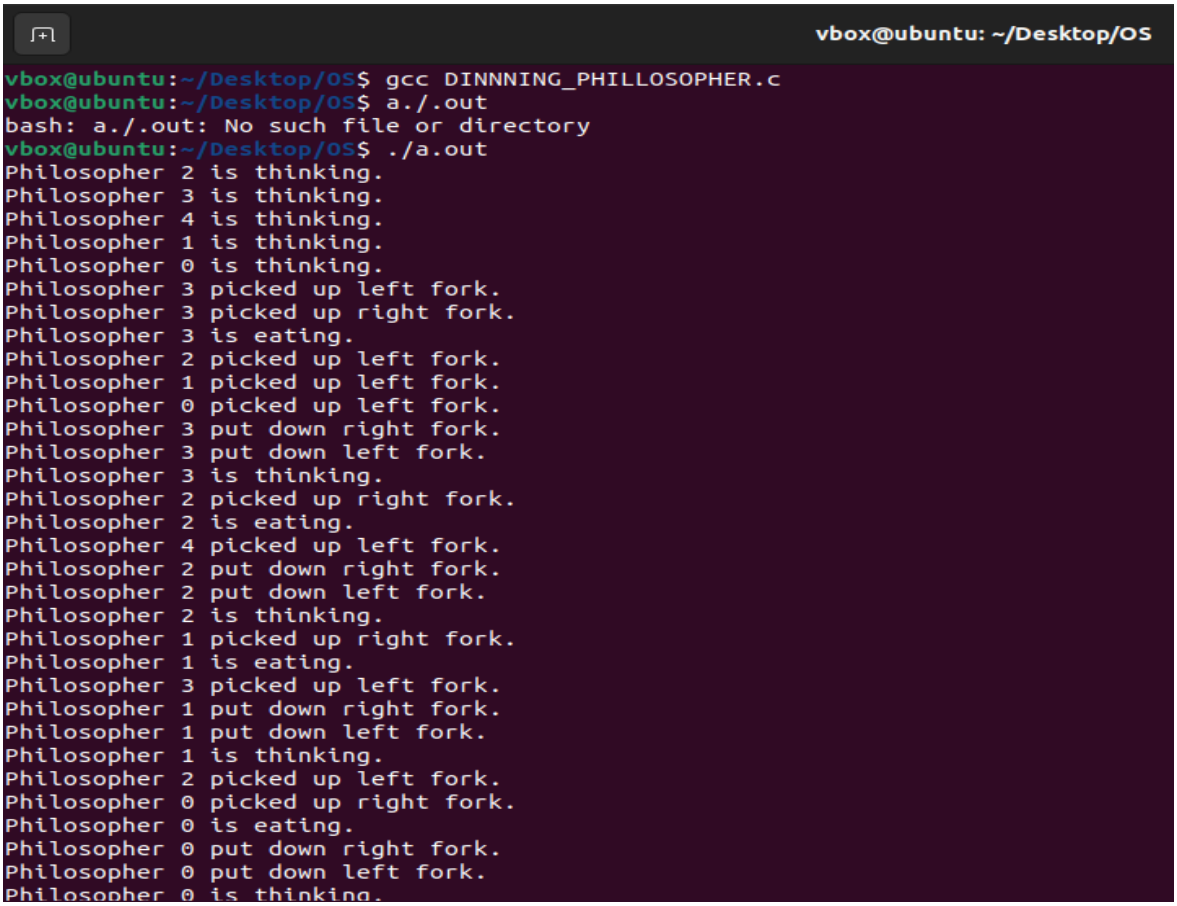
Open ▾ *DINNNING_PHILLOSOPHER.c ~/Desktop/OS

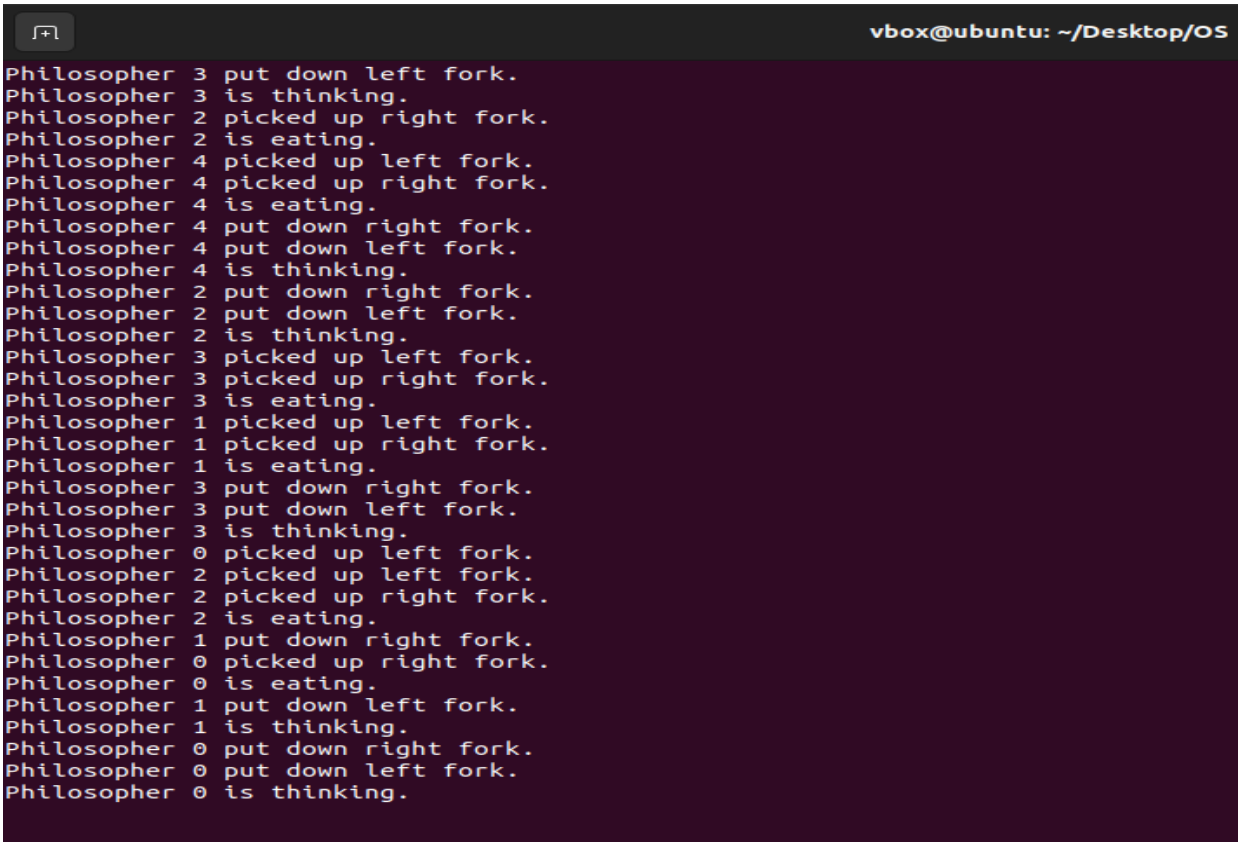
```
        // Eating
        printf("Philosopher %d is eating.\n", id);
        sleep(rand() % 2); // Simulate eating time
        // Put down right fork
        pthread_mutex_unlock(&forks[(id + 1) % NUM_PHILOSOPHERS]);
        printf("Philosopher %d put down right fork.\n", id);
        // Put down left fork
        pthread_mutex_unlock(&forks[id]);
        printf("Philosopher %d put down left fork.\n", id);
    }
}
int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopherIds[NUM_PHILOSOPHERS];
    // Initialize mutexes for each fork
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_init(&forks[i], NULL);
    }
    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopherIds[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopherIds[i]);
    }
    // Join threads (in a real program you might not do this)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }
    // Clean up
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_destroy(&forks[i]);
    }
    return 0;
}
```

Open ▾ DINNNING_PHILLOSOPHER.c ~/Desktop/OS

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

#define NUM_PHILOSOPHERS 5

pthread_mutex_t forks[NUM_PHILOSOPHERS]; // Mutexes for each fork

// Function to simulate the eating process
void* philosopher(void* num) {
    int id = *(int*)num;

    while (1) {
        printf("Philosopher %d is thinking.\n", id);
        sleep(rand() % 2); // Simulate thinking time

        // Pick up left fork
        pthread_mutex_lock(&forks[id]);
        printf("Philosopher %d picked up left fork.\n", id);

        // Pick up right fork
        pthread_mutex_lock(&forks[(id + 1) % NUM_PHILOSOPHERS]);
        printf("Philosopher %d picked up right fork.\n", id);

        // Eating
        printf("Philosopher %d is eating.\n", id);
        sleep(rand() % 2); // Simulate eating time

        // Put down right fork
        pthread_mutex_unlock(&forks[(id + 1) % NUM_PHILOSOPHERS]);
        printf("Philosopher %d put down right fork.\n", id);
```

Outputs:


vbox@ubuntu: ~/Desktop/OS
vbox@ubuntu:~/Desktop/OS$ gcc DINNNING_PHILLOSOPHER.c
vbox@ubuntu:~/Desktop/OS$ a./.out
bash: a./.out: No such file or directory
vbox@ubuntu:~/Desktop/OS$ ./a.out
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 1 is thinking.
Philosopher 0 is thinking.
Philosopher 3 picked up left fork.
Philosopher 3 picked up right fork.
Philosopher 3 is eating.
Philosopher 2 picked up left fork.
Philosopher 1 picked up left fork.
Philosopher 0 picked up left fork.
Philosopher 3 put down right fork.
Philosopher 3 put down left fork.
Philosopher 3 is thinking.
Philosopher 2 picked up right fork.
Philosopher 2 is eating.
Philosopher 4 picked up left fork.
Philosopher 2 put down right fork.
Philosopher 2 put down left fork.
Philosopher 2 is thinking.
Philosopher 1 picked up right fork.
Philosopher 1 is eating.
Philosopher 3 picked up left fork.
Philosopher 1 put down right fork.
Philosopher 1 put down left fork.
Philosopher 1 is thinking.
Philosopher 2 picked up left fork.
Philosopher 0 picked up right fork.
Philosopher 0 is eating.
Philosopher 0 put down right fork.
Philosopher 0 put down left fork.


vbox@ubuntu: ~/Desktop/OS
Philosopher 3 put down left fork.
Philosopher 3 is thinking.
Philosopher 2 picked up right fork.
Philosopher 2 is eating.
Philosopher 4 picked up left fork.
Philosopher 4 picked up right fork.
Philosopher 4 is eating.
Philosopher 4 put down right fork.
Philosopher 4 put down left fork.
Philosopher 4 is thinking.
Philosopher 2 put down right fork.
Philosopher 2 put down left fork.
Philosopher 2 is thinking.
Philosopher 3 picked up left fork.
Philosopher 3 picked up right fork.
Philosopher 3 is eating.
Philosopher 1 picked up left fork.
Philosopher 1 picked up right fork.
Philosopher 1 is eating.
Philosopher 3 put down right fork.
Philosopher 3 put down left fork.
Philosopher 3 is thinking.
Philosopher 0 picked up left fork.
Philosopher 2 picked up left fork.
Philosopher 2 picked up right fork.
Philosopher 2 is eating.
Philosopher 1 put down right fork.
Philosopher 0 picked up right fork.
Philosopher 0 is eating.
Philosopher 1 put down left fork.
Philosopher 1 is thinking.
Philosopher 0 put down right fork.
Philosopher 0 put down left fork.
Philosopher 0 is thinking.

2. **Q] Solve Producer consumer problem using mutex & semaphore.**

Open

PRODUCER_CONSUMER.c
~/Desktop/OS

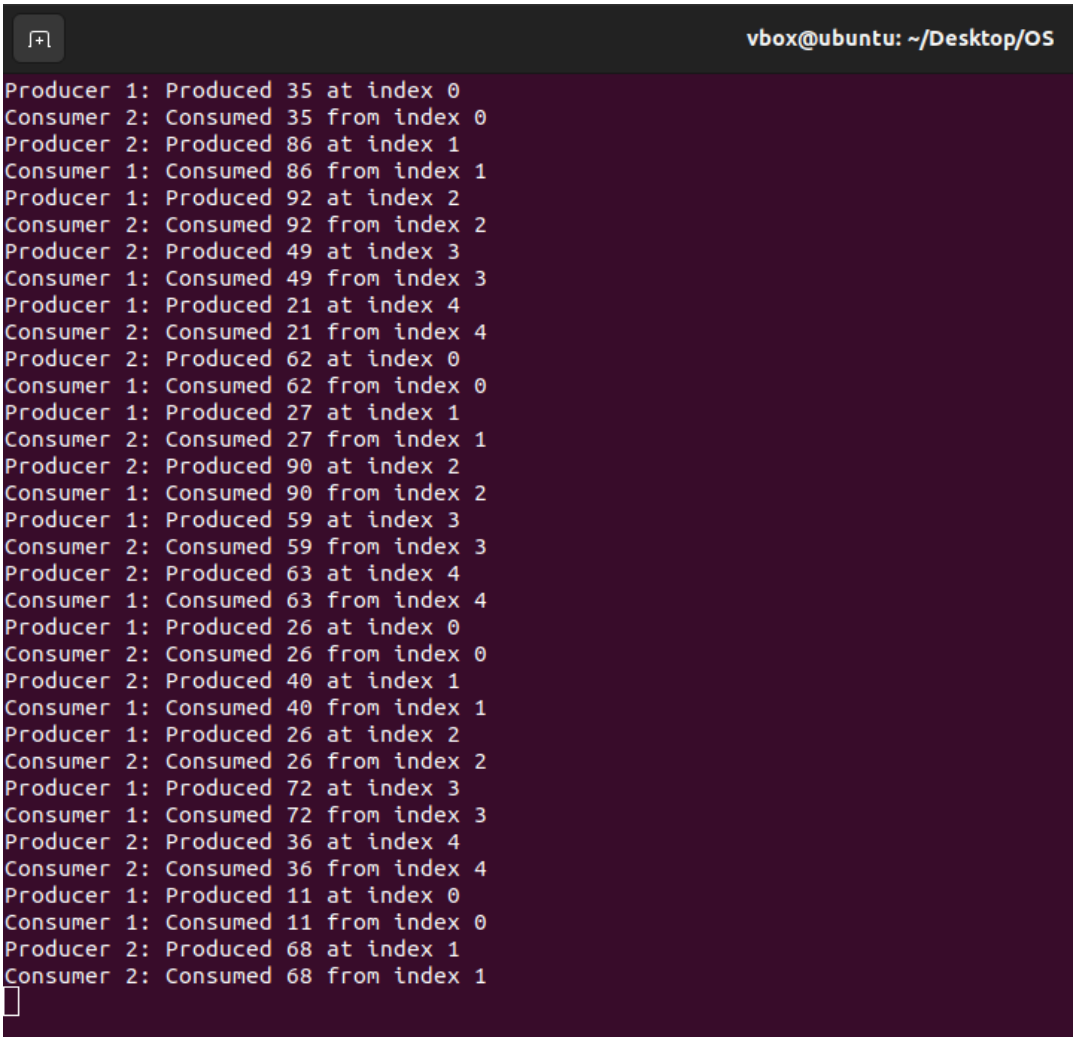
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5 // Size of the buffer
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2

int buffer[BUFFER_SIZE]; // Shared buffer
int in = 0; // Index for the next item to produce
int out = 0; // Index for the next item to consume

sem_t empty; // Semaphore to count empty slots
sem_t full;  // Semaphore to count full slots
pthread_mutex_t mutex; // Mutex for critical section

// Producer function
void *producer(void *arg) {
    int id = *(int *)arg;
    while (1) {
        // Produce an item
        int item = rand() % 100; // Random item
        sem_wait(&empty); // Wait for an empty slot
        pthread_mutex_lock(&mutex); // Lock the critical section

        // Add the item to the buffer
        buffer[in] = item;
        printf("Producer %d: Produced %d at index %d\n", id, item, in);
        in = (in + 1) % BUFFER_SIZE; // Update the index

        pthread_mutex_unlock(&mutex); // Unlock the critical section
        sem_post(&full); // Signal that a new item has been produced
```

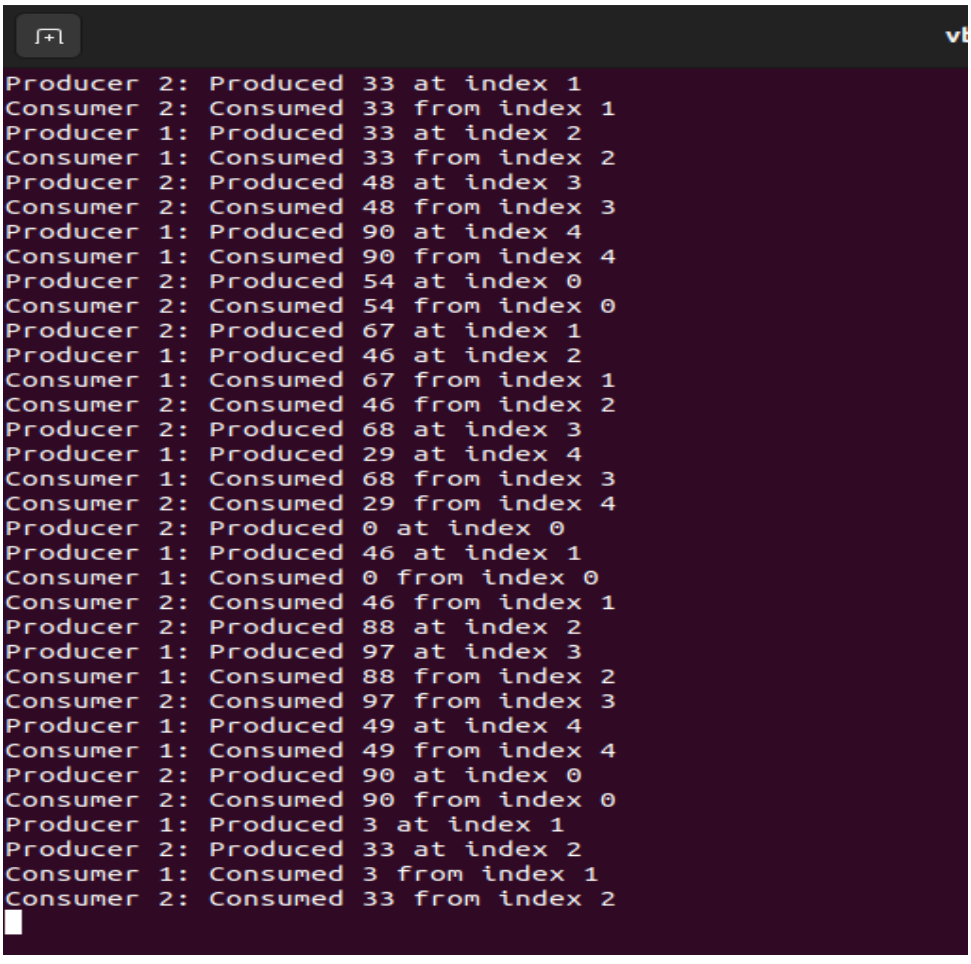
Open

*PRODUCER_CONSUMER.c
~/Desktop/OS

```c
        sleep(1); // Simulate time taken to consume
    }
}
int main() {
    pthread_t producers[NUM_PRODUCERS], consumers[NUM_CONSUMERS];
    int producerIds[NUM_PRODUCERS], consumerIds[NUM_CONSUMERS];
    // Initialize semaphores and mutex
    sem_init(&empty, 0, BUFFER_SIZE); // Initially, all slots are empty
    sem_init(&full, 0, 0); // Initially, no slots are full
    pthread_mutex_init(&mutex, NULL);
    // Create producer threads
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        producerIds[i] = i + 1;
        pthread_create(&producers[i], NULL, producer, &producerIds[i]);
    }
    // Create consumer threads
    for (int i = 0; i < NUM_CONSUMERS; i++) {
        consumerIds[i] = i + 1;
        pthread_create(&consumers[i], NULL, consumer, &consumerIds[i]);
    }
    // Join threads (in a real program you might not do this)
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_join(producers[i], NULL);
    }
    for (int i = 0; i < NUM_CONSUMERS; i++) {
        pthread_join(consumers[i], NULL);
    }
    // Clean up
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Outputs:


vbox@ubuntu: ~/Desktop/OS
Producer 1: Produced 35 at index 0
Consumer 2: Consumed 35 from index 0
Producer 2: Produced 86 at index 1
Consumer 1: Consumed 86 from index 1
Producer 1: Produced 92 at index 2
Consumer 2: Consumed 92 from index 2
Producer 2: Produced 49 at index 3
Consumer 1: Consumed 49 from index 3
Producer 1: Produced 21 at index 4
Consumer 2: Consumed 21 from index 4
Producer 2: Produced 62 at index 0
Consumer 1: Consumed 62 from index 0
Producer 1: Produced 27 at index 1
Consumer 2: Consumed 27 from index 1
Producer 2: Produced 90 at index 2
Consumer 1: Consumed 90 from index 2
Producer 1: Produced 59 at index 3
Consumer 2: Consumed 59 from index 3
Producer 2: Produced 63 at index 4
Consumer 1: Consumed 63 from index 4
Producer 1: Produced 26 at index 0
Consumer 2: Consumed 26 from index 0
Producer 2: Produced 40 at index 1
Consumer 1: Consumed 40 from index 1
Producer 1: Produced 26 at index 2
Consumer 2: Consumed 26 from index 2
Producer 1: Produced 72 at index 3
Consumer 1: Consumed 72 from index 3
Producer 2: Produced 36 at index 4
Consumer 2: Consumed 36 from index 4
Producer 1: Produced 11 at index 0
Consumer 1: Consumed 11 from index 0
Producer 2: Produced 68 at index 1
Consumer 2: Consumed 68 from index 1


Producer 2: Produced 33 at index 1
Consumer 2: Consumed 33 from index 1
Producer 1: Produced 33 at index 2
Consumer 1: Consumed 33 from index 2
Producer 2: Produced 48 at index 3
Consumer 2: Consumed 48 from index 3
Producer 1: Produced 90 at index 4
Consumer 1: Consumed 90 from index 4
Producer 2: Produced 54 at index 0
Consumer 2: Consumed 54 from index 0
Producer 2: Produced 67 at index 1
Producer 1: Produced 46 at index 2
Consumer 1: Consumed 67 from index 1
Consumer 2: Consumed 46 from index 2
Producer 2: Produced 68 at index 3
Producer 1: Produced 29 at index 4
Consumer 1: Consumed 68 from index 3
Consumer 2: Consumed 29 from index 4
Producer 2: Produced 0 at index 0
Producer 1: Produced 46 at index 1
Consumer 1: Consumed 0 from index 0
Consumer 2: Consumed 46 from index 1
Producer 2: Produced 88 at index 2
Producer 1: Produced 97 at index 3
Consumer 1: Consumed 88 from index 2
Consumer 2: Consumed 97 from index 3
Producer 1: Produced 49 at index 4
Consumer 1: Consumed 49 from index 4
Producer 2: Produced 90 at index 0
Consumer 2: Consumed 90 from index 0
Producer 1: Produced 3 at index 1
Producer 2: Produced 33 at index 2
Consumer 1: Consumed 3 from index 1
Consumer 2: Consumed 33 from index 2

3. **Q] Solve Reader writer problem using mutex & semaphore.**

Open

READER_WRITER.c
~/Desktop/OS

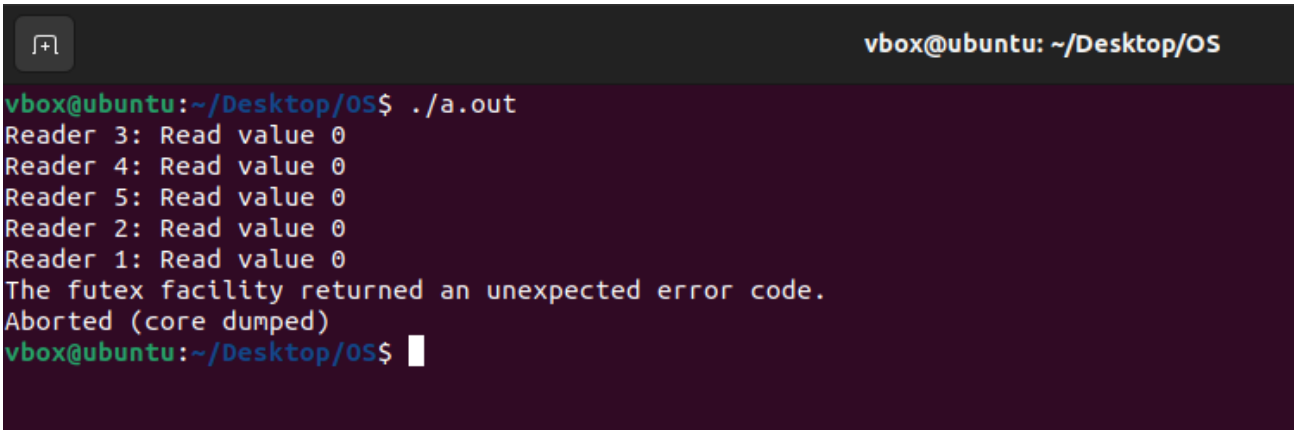
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_READERS 5
#define NUM_WRITERS 2

sem_t readSemaphore;
pthread_mutex_t writeMutex;
int readCount = 0; // Count of current readers
int sharedResource = 0; // Shared resource

// Reader function
void *reader(void *arg) {
    int id = *(int *)arg;
    while (1) {
        // Start reading
        sem_wait(&readSemaphore);
        pthread_mutex_lock(&writeMutex);
        readCount++;
        if (readCount == 1) {
            // First reader locks the resource
            sem_wait(&writeMutex);
        }
        pthread_mutex_unlock(&writeMutex);
        sem_post(&readSemaphore);
        // Reading
        printf("Reader %d: Read value %d\n", id, sharedResource);
        sleep(1); // Simulate reading time

        // Done reading
        pthread_mutex_lock(&writeMutex);
```

Open

READER_WRITER.c
~/Desktop/OS

```c
        sleep(1); // Simulate idle time before next write
    }
}
int main() {
    pthread_t readers[NUM_READERS], writers[NUM_WRITERS];
    int readerIds[NUM_READERS], writerIds[NUM_WRITERS];
    // Initialize semaphores and mutex
    sem_init(&readSemaphore, 0, 1);
    sem_init(&writeMutex, 0, 1);
    pthread_mutex_init(&writeMutex, NULL);
    // Create reader threads
    for (int i = 0; i < NUM_READERS; i++) {
        readerIds[i] = i + 1;
        pthread_create(&readers[i], NULL, reader, &readerIds[i]);
    }
    // Create writer threads
    for (int i = 0; i < NUM_WRITERS; i++) {
        writerIds[i] = i + 1;
        pthread_create(&writers[i], NULL, writer, &writerIds[i]);
    }
    // Join threads (in a real program you might not do this)
    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);
    }
    // Clean up
    sem_destroy(&readSemaphore);
    sem_destroy(&writeMutex);
    pthread_mutex_destroy(&writeMutex);
    return 0;
}
```

Outputs:


vbox@ubuntu: ~/Desktop/OS
vbox@ubuntu:~/Desktop/OS$ ./a.out
Reader 3: Read value 0
Reader 4: Read value 0
Reader 5: Read value 0
Reader 2: Read value 0
Reader 1: Read value 0
The futex facility returned an unexpected error code.
Aborted (core dumped)
vbox@ubuntu:~/Desktop/OS$