NAME: G. GANESH

REG NO: 192373008

EXERICSE-20

Construct a C program to simulate Reader-Writer problem using Semaphores.

Aim:

To construct a C program to simulate the Reader-Writer problem using semaphores to ensure synchronization between readers and writers.

Algorithm:

1. Initialize Semaphores and Shared Resource:
    - Use a semaphore rw_mutex to allow mutual exclusion for writers.
    - Use a semaphore mutex to control access to the reader count.
    - Initialize rw_mutex and mutex to 1.
2. Reader Process:
    - Wait on mutex to safely update the reader count.
    - If it is the first reader, wait on rw_mutex to block writers.
    - Signal mutex after updating the reader count.
    - Read the shared resource.
    - After reading, wait on mutex to safely update the reader count.
    - If it is the last reader, signal rw_mutex to allow writers.
    - Signal mutex after updating the reader count.
3. Writer Process:
    - Wait on rw_mutex to get exclusive access to the shared resource.
    - Write to the shared resource.
    - Signal rw_mutex to allow others.
4. Create Threads:
    - Create threads for reader and writer processes.
    - Use appropriate delays to simulate concurrent operations.
5. Terminate Execution:
    - Ensure all threads finish execution and clean up resources.

Procedure:

1. Initialize semaphores and variables.
2. Implement the reader and writer functions with synchronization logic.
3. Create reader and writer threads to simulate concurrent access.
4. Run the program to observe the synchronization behavior.
5. Print the operations performed by readers and writers to verify correctness.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t rw_mutex, mutex;
int read_count = 0, shared_data = 0;
void *reader(void *param) {
    int reader_id = *((int *)param);
    while (1) {
        sem_wait(&mutex);
        read_count++;
        if (read_count == 1) {
            sem_wait(&rw_mutex);
        }
        sem_post(&mutex);
        printf("Reader %d reads shared data: %d\n", reader_id, shared_data);
        sem_wait(&mutex);
        read_count--;
        if (read_count == 0) {
            sem_post(&rw_mutex);
        }
        sem_post(&mutex);
        sleep(rand() % 3);
```

```
    }
}
void *writer(void *param) {
    int writer_id = *((int *)param);
    while (1) {
        sem_wait(&rw_mutex);
        shared_data++;
        printf("Writer %d updates shared data to: %d\n", writer_id, shared_data);
        sem_post(&rw_mutex);
        sleep(rand() % 3);
    }
}
int main() {
    pthread_t readers[5], writers[3];
    int reader_ids[5], writer_ids[3];
    sem_init(&rw_mutex, 0, 1);
    sem_init(&mutex, 0, 1);
    for (int i = 0; i < 5; i++) {
        reader_ids[i] = i + 1;
        pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
    }
    for (int i = 0; i < 3; i++) {
        writer_ids[i] = i + 1;
        pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(readers[i], NULL);
    }
    for (int i = 0; i < 3; i++) {
        pthread_join(writers[i], NULL);
```

```
    }
    sem_destroy(&rw_mutex);
    sem_destroy(&mutex);
    return 0;
}
```

Result:

The program successfully simulates the Reader-Writer problem using semaphores, ensuring synchronization between multiple readers and writers. Readers can access the shared resource concurrently, while writers get exclusive access.

Output: