Q-Write a program to implement Dekker's algorithm using Semaphore.

```
#include <iostream>
#include <pthread.h>
using namespace std;
#define N 2 // Number of processes
bool flag[N] = {false, false}; // Flags for each process
int turn = 0; // Turn variable
pthread_mutex_t lock; // Mutex for critical section
sem_t sem[N]; // Semaphores for each process
void* process(void* arg) {
    int id = *((int*)arg);
    int other = 1 - id;
        for (int i = 0; i < 5; ++i) {
        // Entry section
        flag[id] = true;
        while (flag[other]) {
            if (turn != id) {
                flag[id] = false;
                while (turn != id) {} // Wait
                flag[id] = true;
            }
        }

        // Critical section
        pthread_mutex_lock(&lock);
        cout << "Process " << id << " is in critical section" << endl;
        pthread_mutex_unlock(&lock);
         // Exit section
        turn = other;
        flag[id] = false;
    }
```

```
  return nullptr;
}
int main() {
    pthread_t threads[N];
    int ids[N] = {0, 1};

    pthread_mutex_init(&lock, NULL);
    for (int i = 0; i < N; ++i) {
        sem_init(&sem[i], 0, 0);
    }
 for (int i = 0; i < N; ++i) {
        pthread_create(&threads[i], NULL, process, &ids[i]);
    }
for (int i = 0; i < N; ++i) {
        pthread_join(threads[i], NULL);
    }
  pthread_mutex_destroy(&lock);
    for (int i = 0; i < N; ++i) {
        sem_destroy(&sem[i]);
    }
  return 0;
}
```

Q-Write a program to implement Reader and Writer Problem using Semaphore..

```
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
using namespace std;
// Data shared between readers and writers
int data = 0;
int readers_count = 0;
// Semaphores
```

```
sem_t mutex, wrt;
void* reader(void* arg) {
    int id = *((int*)arg);
 while (true) {
        sem_wait(&mutex); // Lock mutex to protect readers_count
        readers_count++;
        if (readers_count == 1) {
            sem_wait(&wrt); // Lock wrt to block writers
        }
        sem_post(&mutex); // Unlock mutex

        // Read data
        cout << "Reader " << id << " reads: " << data << endl;

        sem_wait(&mutex); // Lock mutex to protect readers_count
        readers_count--;
        if (readers_count == 0) {
            sem_post(&wrt); // Unlock wrt to allow writers
        }
   sem_post(&mutex); // Unlock mutex
      // Sleep for a random time
        usleep(rand() % 1000000);
    }
return NULL;
}

void* writer(void* arg) {
    int id = *((int*)arg);
 while (true) {
        sem_wait(&wrt); // Lock wrt to ensure mutual exclusion between writers
        data++; // Write to data
        cout << "Writer " << id << " writes: " << data << endl;
        sem_post(&wrt); // Unlock wrt

        // Sleep for a random time
```

```
        usleep(rand() % 1000000);
    }

    return NULL;
}
int main() {
    const int NUM_READERS = 3;
    const int NUM_WRITERS = 2;
pthread_t reader_threads[NUM_READERS], writer_threads[NUM_WRITERS];
    int reader_ids[NUM_READERS], writer_ids[NUM_WRITERS];
 // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&wrt, 0, 1);
 // Create reader threads
    for (int i = 0; i < NUM_READERS; ++i) {
        reader_ids[i] = i + 1;
        pthread_create(&reader_threads[i], NULL, reader, &reader_ids[i]);
    }

    // Create writer threads
    for (int i = 0; i < NUM_WRITERS; ++i) {
        writer_ids[i] = i + 1;
        pthread_create(&writer_threads[i], NULL, writer, &writer_ids[i]);
    }

    // Join reader threads
    for (int i = 0; i < NUM_READERS; ++i) {
        pthread_join(reader_threads[i], NULL);
    }

    // Join writer threads
    for (int i = 0; i < NUM_WRITERS; ++i) {
        pthread_join(writer_threads[i], NULL);
    }
```

```
    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;
}
```