# HW3

Sahar Eitam 318283116

## q1:

### a.

```
 using System;
using System.Threading;
class SemaphoreAlternative
{
   private int currentCount;
   private int maxLimit;
   private readonly Mutex syncLock = new Mutex();

   public SemaphoreAlternative(int initialCount, int maxCount)
   {
      if (initialCount < 0 || maxCount <= 0 || initialCount > maxCount)
         throw new ArgumentException("Invalid initial or max count.");

      currentCount = initialCount;
      maxLimit = maxCount;
   }

   public bool Acquire()
   {
      syncLock.WaitOne();
      while (currentCount == 0)
      {
         syncLock.ReleaseMutex();
         Thread.Sleep(1);
         syncLock.WaitOne();
      }
      currentCount--;
      syncLock.ReleaseMutex();
      return true;
   }

   public bool Release(int increment = 1)
   {
      if (increment < 1) throw new ArgumentException("Increment must be greater than zero.");

      syncLock.WaitOne();
      if (currentCount + increment > maxLimit)
      {
         syncLock.ReleaseMutex();
```

```
            return false;
        }

        currentCount += increment;
        syncLock.ReleaseMutex();
        return true;
    }
}
```

**b.**

Yes, Peterson’s Algorithm can be generalized for three processes. Here's the pseudo-code:

```
Process P0
Enter Critical Section:
Set flag[0] = true
 turn[0] = 1
 turn[1] = 2
Wait while (flag[1] && turn[0] == 1) or (flag[2] && turn[1] == 2)
//
Critical Section: Execute code
//
Exit Critical Section: Set flag[0] = false.

Remainder Section:Execute remainder section code.

Process P1
Enter Critical Section:
Set flag[1] = true
turn[0] = 0
turn[1] = 2
Wait while (flag[0] && turn[0] == 0) or (flag[2] && turn[1] == 2)
//
Critical Section: Execute code
//
Exit Critical Section: Set flag[1] = false.

Remainder Section:Execute remainder section code

Process P2
Enter Critical Section:
Set flag[2] = true
turn[0] = 0
turn[1] = 1
Wait while (flag[0] && turn[0] == 0) or (flag[1] && turn[1] == 1)
//
Critical Section: Execute code.
//
```

Exit Critical Section: Set flag[2] = false.

Remainder Section:Execute remainder section code.

summary

- Peterson's Algorithm can be generalized for three processes.
- Each process sets its flag to true to signal entry intent.
- Turn variables are used to prioritize between processes.
- A process waits until others are not interested or it's its turn.
- After completing the critical section, the process sets its flag to false.

**c.**

**Dekker's Algorithm:**

**Advantages** of Dekker:

- **No hardware support** needed.
- **Portability:** Highly portable across different systems and architectures.
- **Educational Value:** Demonstrates fundamental principles of mutual exclusion without advanced synchronization tools.

**Disadvantages** of Dekker:

- **Restricted to Two Processes**: It can only handle two processes at a time, making it unsuitable for systems with multiple processes.
- **Inefficient CPU Usage**: It uses busy waiting, which wastes CPU cycles while waiting for access to the critical section.
- **Poor Scalability**: The algorithm doesn't easily extend to more than two processes, needing more complex synchronization techniques in larger systems.
- **Modern Hardware Issues**: Current compilers and CPUs with out-of-order execution can interfere with its functioning, requiring tools like memory barriers or volatile to ensure proper operation.

**Mutexes and Semaphores:**

**Advantages of Mutexes and Semaphores:**

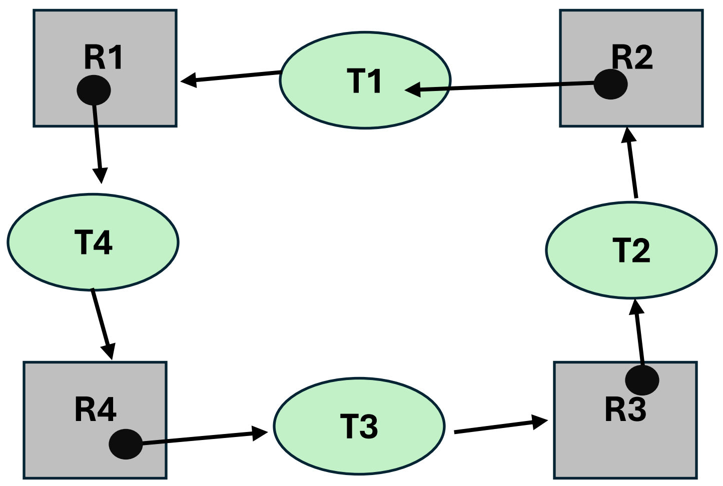
- **Mutex**: Guarantees exclusive access to a resource by allowing only one thread to use it at a time. Only the thread that locks the mutex can unlock it.
- **Semaphore**: Regulates resource access with a counter, making it useful for coordinating multiple threads and controlling how many can access a resource simultaneously.

**Disadvantages of Mutexes and Semaphores:**

- **Risk of Deadlock**: Improper use of mutexes can result in deadlock, where threads get stuck waiting on each other indefinitely.
- **Resource Management Complexity**: Handling semaphores can be difficult, especially with many threads and resources, which can lead to issues like semaphore leaks or priority inversion.

**d.**

Deadlock Scenario:
T1 holds R1 and is waiting for R2.
T2 holds R2 and is waiting for R3.
T3 holds R3 and is waiting for R4.
T4 holds R4 and is waiting for R1.
This forms a circular chain where no thread can proceed because each one is holding a resource that another thread needs.



Allocation matrix:

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 1 | 0 | 0 | 0 |
| T2 | 0 | 1 | 0 | 0 |
| T3 | 0 | 0 | 1 | 0 |
| T4 | 0 | 0 | 0 | 1 |

## e.

Deadlock Scenario

Thread T1 locks Mutex M1 and Mutex M3.

Thread T2 locks Mutex M2 and Mutex M4.

Thread T3 locks Mutex M5 and Mutex M6.

Thread T4 locks Mutex M1.

Next, each thread attempts to lock another mutex:

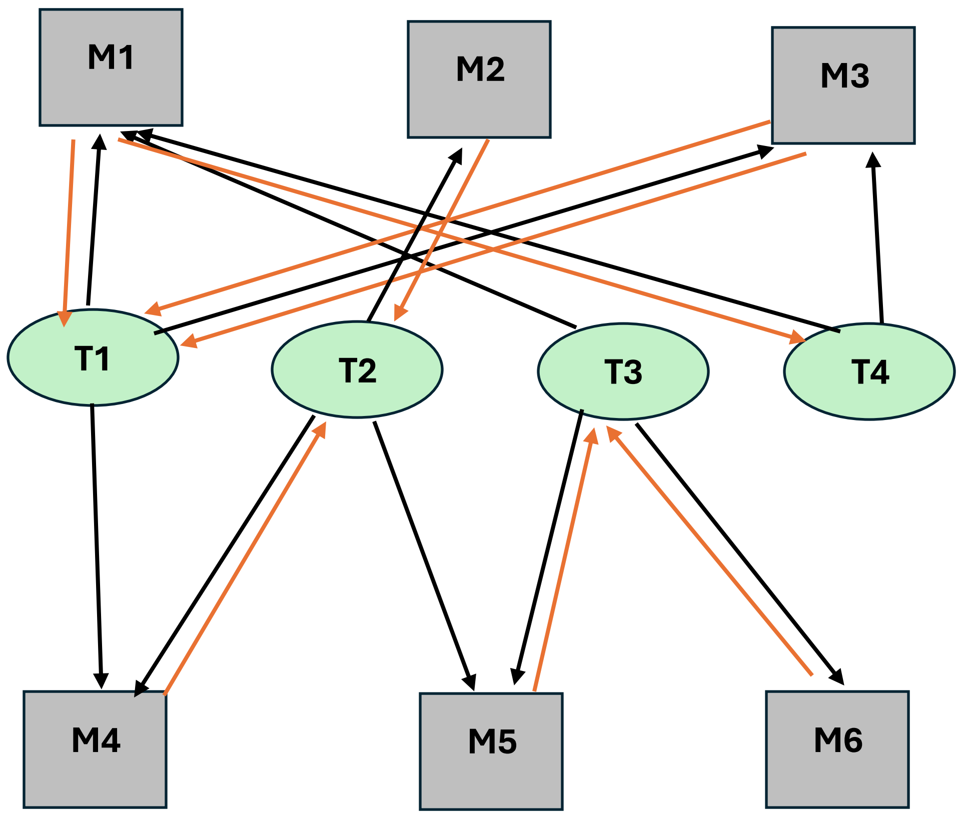
Thread T1 attempts to lock Mutex M4 (held by Thread T2).

Thread T2 attempts to lock Mutex M5 (held by Thread T3).

Thread T3 attempts to lock Mutex M1 (held by Thread T1).

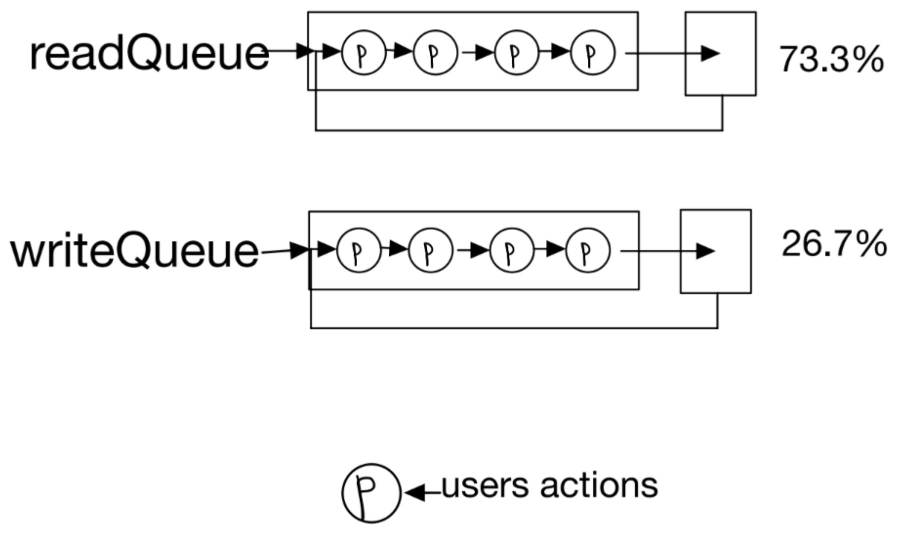
Thread T4 attempts to lock Mutex M3 (held by Thread T1).

This situation creates a deadlock because all four threads are waiting on mutexes that are already locked by others, leading to a circular dependency where no thread can proceed.



| Process | Holds | Wants |
|---|---|---|
| T1 | M1, M3 | M4 |
| T2 | M2, M4 | M5 |
| T3 | M5, M6 | M1 |
| T4 | M1 | M3 |

## Q3.



If there is write thread wait, else open read lock

If there is read threads wait, else open write lock

In this task, I created a schedule by 2 queues, that each read operations go to the read queue and each write operation go to the write queue:

| | |
|---|---|
| readQueue | SearchInRange<br>FindAll<br>Print<br>SearchString<br>GetCell<br>SearchInRow<br>SearchInCol<br>GetSize |
| writeQueue | AddRow<br>AddCol<br>SetAll<br>SetCell<br>ExchangeRows<br>ExchangeCols |

For each method I did method that enter the user action to the right queue!

In the provided implementation of the **SharableSpreadsheet** class, several synchronization mechanisms are used to ensure thread safety and prevent race conditions, while maintaining efficiency in a concurrent environment.

**1. Types of Locks Used:**

- **ReaderWriterLockSlim**: This lock is used to allow multiple concurrent read operations while ensuring exclusive access for write operations. It improves performance by permitting more than one thread to read the spreadsheet simultaneously, but only one thread can modify it at a time.
- **SemaphoreSlim**: This semaphore limits the number of concurrent users allowed to perform operations (when the number of users is specified in the constructor). It provides a mechanism to control how many threads can concurrently search or modify the spreadsheet.

**2. Number of Locks:**

- **ReaderWriterLockSlim**: This lock is applied across the entire spreadsheet, controlling read and write access to the shared data.

- **SemaphoreSlim**: Used to limit the number of concurrent users performing operations. If a maximum number of users is specified, this semaphore restricts access accordingly.

**3. Design Strategy:**

- **Queue-based Scheduler**: The implementation uses two **concurrent queues** (one for read operations and one for write operations) and a background thread that processes these queues. This design ensures that operations are executed in order and appropriately managed by the locks.
- **Efficient Read/Write Operations**: The design optimizes concurrent read operations, allowing multiple threads to read data simultaneously while still ensuring exclusive access for write operations.

In the diagram, the spreadsheet cells are protected by **ReaderWriterLockSlim**. Each thread performing a read operation can acquire a read lock, while threads that perform write operations must acquire a write lock. The **SemaphoreSlim** controls the total number of concurrent users allowed to access the spreadsheet.