OS – Assignment 2

Nofar Selouk (318502721) & Noam Tarshish (207761024)

Q1. Synchronization (15 points)

a.

מצרפים צילום מסך של הקוד הכתוב –

```
1 reference
class MySemaphore
{
    int s;
    int limit;
    Mutex mutex_wait;
    Mutex mutex_release;
    Mutex atomic;

    0 references
    public MySemaphore(int starting, int max)
    {
        s = starting;
        limit = max;
        mutex_wait = new Mutex();
        mutex_release = new Mutex();
        atomic = new Mutex();
    }

    0 references
    public bool WaitOne()
    {
        mutex_wait.WaitOne();
        atomic.WaitOne();
        if (s == 0)
        {
            atomic.ReleaseMutex();
            mutex_wait.ReleaseMutex();
            return false;
        }
        s--;
        atomic.ReleaseMutex();
        mutex_wait.ReleaseMutex();
        return true;
    }
```

```
0 references
public bool Release(int num = 1)
{
    mutex_release.WaitOne();
    atomic.WaitOne();
    if (num + s > limit)
        s = limit;
    else
        s = s + num;
    atomic.ReleaseMutex();
    mutex_release.ReleaseMutex();
    return true;
}
}
```

b.

Peterson's Algorithm can be generalized for more than two processes, including three processes, but it becomes quite complex and is not typically used for more than two processes due to the complexity and inefficiency. Here's how you can generalize it for three processes using a form of the algorithm:

Generalized Peterson's Algorithm for Three Processes

Shared Variables:

- flag[i]: Boolean array, initially false, indicating if process i wants to enter the critical section.
- turn: Array of size n-1, where n is the number of processes (in this case, n = 3).

Pseudo Code:

// Process P0

Repeat

Flag[0] = true;

Turn[0] = 1;

While ((flag[1] && turn[0] == 1) || (flag[2] && turn[0] == 2)){

// busy wait

}

// critical section

…

Flag[0] = false;

```
        // remainder section
        …
Until false
// Process P1
Repeat
        Flag[1] = true;
        Turn[0] = 2;
        While ((flag[0] && turn[0] == 2) || (flag[2] && turn[1] == 2)){
                // busy wait
        }
       // critical section
        …
       Flag[1] = false;
        // remainder section
        …
Until false
// Process P2
Repeat
        Flag[2] = true;
        Turn[1] = 0;
        While ((flag[0] && turn[1] == 0) || (flag[1] && turn[1] == 0)){
                // busy wait
        }
       // critical section
        …
       Flag[2] = false;
        // remainder section
        …
Until false
```

Explanation:

1. **Initialization**: Each process sets its flag to true to indicate it wants to enter the critical section.
2. **Turn Setting**: Each process sets the turn variable to let other processes know it wants to enter the critical section.
3. **Busy Wait**: Each process waits while both other processes want to enter the critical section and have precedence according to the turn variable.
4. **Critical Section**: Once it can enter, the process executes its critical section code.
5. **Flag Reset**: The process resets its flag to false after exiting the critical section.
6. **Remainder Section**: The process executes the remainder section code.

c.

Dekker's Algorithm is one of the earliest solutions to the mutual exclusion problem in concurrent programming. It allows two processes to share a single-use resource without conflicts, ensuring mutual exclusion, progress, and bounded waiting.

Dekker's Algorithm uses two main variables:

- flag: An array of two booleans, indicating if a process wants to enter the critical section.
- turn: An integer that indicates whose turn it is to enter the critical section.

Comparison with Mutex/Semaphore Objects:

Criteria	Dekker's Algorithm	Mutex/Semaphore Objects
Number of Processes	Limited to two processes	Can handle multiple processes or threads
Hardware Support	No special hardware support required	Typically requires hardware support for atomic operations
Implementation Complexity	Simple for two processes	More complex, especially for managing multiple processes
CPU Usage	Busy waiting, can waste CPU resources	Uses blocking mechanisms, more efficient CPU usage
Scalability	Not scalable beyond two processes	Highly scalable
Risk of Deadlock	Lower risk of deadlock due to simplicity	Higher risk if not implemented correctly
Operating System Support	No special OS support required	Widely supported by modern operating systems and programming languages
Educational Value	Good for understanding basic mutual exclusion concepts	Useful for understanding advanced concurrency mechanisms

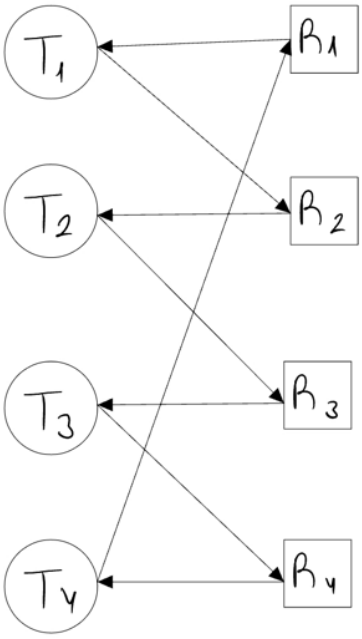
d.

Scenario: Consider a system with four threads (T1, T2, T3, T4) and four resources (R1, R2, R3, R4). Each thread requires two resources to complete its task. The allocation and request of resources happen in such a way that a circular wait condition is created, leading to a deadlock.

Deadlock Situation:

- Thread T1 holds R1 and requests R2.
- Thread T2 holds R2 and requests R3.
- Thread T3 holds R3 and requests R4.
- Thread T4 holds R4 and requests R1.

We will draw the allocation graph-



Allocation Matrix

The allocation matrix shows which resources are held by which threads and which resources are requested by which threads.

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

Request Matrix

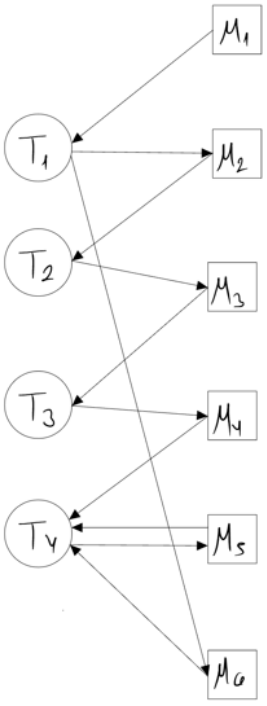
	R1	R2	R3	R4
T1	0	1	0	0
T2	0	0	1	0
T3	0	0	0	1
T4	1	0	0	0

Circular Wait: The circular wait condition is clearly visible, where each thread is waiting for a resource held by another thread, forming a cycle.
Hold and Wait: Each thread is holding at least one resource and waiting for another.
No Preemption: Resources are only released voluntarily by the threads, not preempted by the system.
Mutual Exclusion: Resources are being held exclusively by threads.
This creates a classic deadlock scenario where none of the threads can proceed because they are all waiting for a resource that is currently held by another thread in the cycle.
e.
Scenario: Consider a multi-threaded process where four threads (T1, T2, T3, T4) use six mutexes (M1, M2, M3, M4, M5, M6). The threads acquire and request mutexes in such a way that a circular wait condition is created, leading to a deadlock.

Deadlock Situation:

- Thread T1 holds M1 and requests M2.
- Thread T2 holds M2 and requests M3.
- Thread T3 holds M3 and requests M4.
- Thread T4 holds M4 and requests M5.
- Thread T1 also requests M6 to increase the complexity.
- Thread T4 holds M5 and requests M6.

We will draw the allocation graph-



Allocation Matrix:

	M1	M2	M3	M4	M5	M6
T1	1	0	0	0	0	0
T2	0	1	0	0	0	0
T3	0	0	1	0	0	0
T4	0	0	0	1	1	1

Request Matrix:

	M1	M2	M3	M4	M5	M6
T1	0	1	0	0	0	0
T2	0	0	1	0	0	0
T3	0	0	0	1	0	0
T4	0	0	0	0	1	0

Circular Wait: The circular wait condition is clearly visible, where each thread is waiting for a mutex held by another thread, forming a cycle.

Hold and Wait: Each thread is holding at least one mutex and waiting for another.

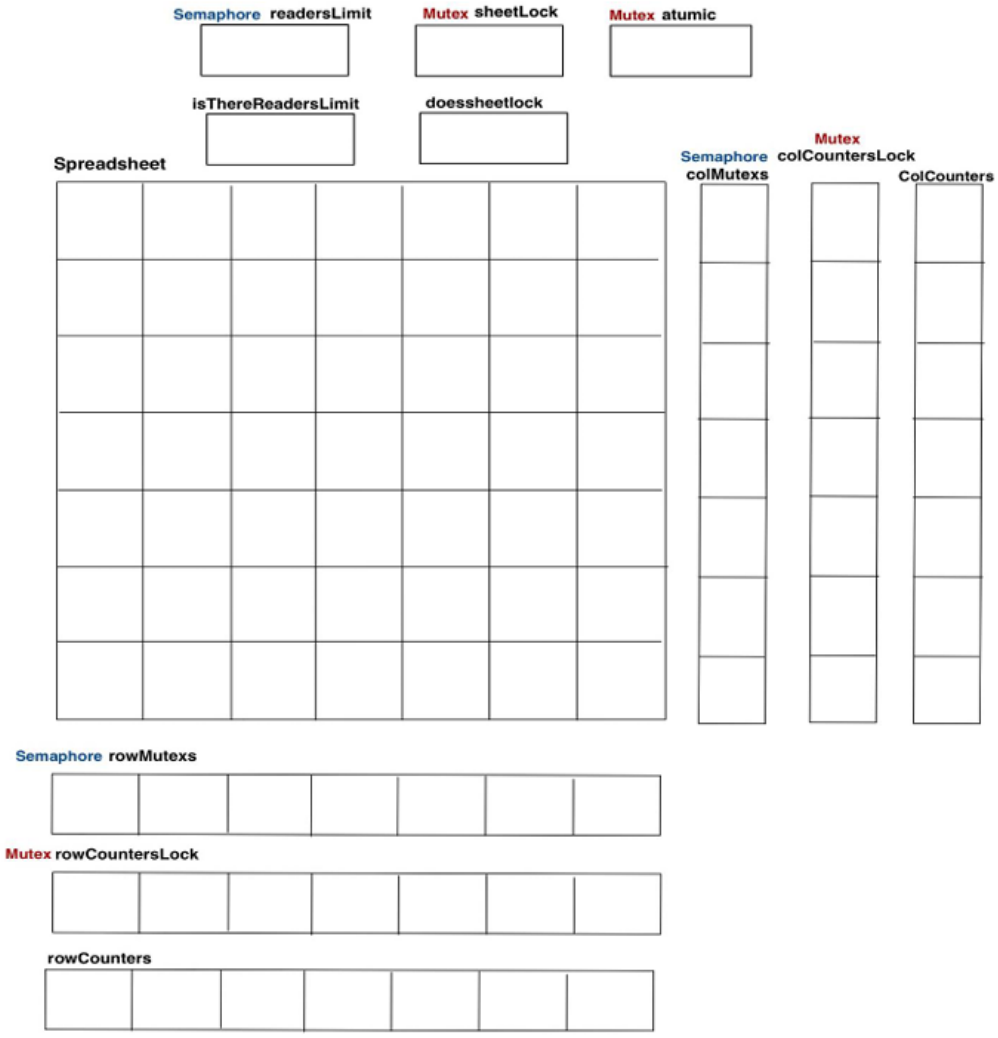
No Preemption: Mutexes are only released voluntarily by the threads, not preempted by the system.

Mutual Exclusion: Mutexes are being held exclusively by threads.

This creates a classic deadlock scenario where none of the threads can proceed because they are all waiting for a mutex that is currently held by another thread in the cycle.

Q3. Implement Sharable Spreadsheet object (50 points)

A diagram of the department locks and their deputies, and the locks related to these in the department:



Details of the department parameters:

- sheet: A list of lists that contain string names representing the spreadsheet.
- rows: A variable representing the number of rows in the table.
- columns: A variable representing the number of columns in the table.
- isThereReadersLimit: A Boolean type that will indicate whether we have a limit on the number of users allowed to perform search operations at any given moment.
- doesSheetlock: A Boolean type that will indicate whether we have an operation that activates table locking.
- rowCounters/colCounters: A list of int type that will count how many users perform operations on each row/column and adjusted in each given moment.

Details of the department locks:

- readersLimit: A semaphore type lock that his maximum limit number determined by the parameter specifying the number of search users in the table (nUsers).
- sheetLock: A Mutex type lock that will prevent users from changing the value of the boolean parameter doesSheetlock.
- rowMutexes/colMutexes: A list of semaphore type locks for rows/columns, adjusted accordingly. This list will include only one user, so this lock is identical to the mutex type. The advantage of using the semaphore type over the mutex type is to allow access to many threads at the same time without the need to release them. If one user locks the read operation, and the other user calls for reading and enters the second queue, then it means they are both callers, and one of them will be the last to be called to the resource, and the first to the thread execution.
- rowCountersLock/colCountersLock: Mutex type locks to prevent operations on the row/column counters.
- To add on the atomic - a Mutex type lock that will prevent and ensure the existence of a branch of atomic operation.

Spreadsheet Design:

Design Principles: Our spreadsheet design is based on the principle of readers/writers that we learned in class. Initially, we divided the spreadsheet into individual tables for each row. Thus, whenever locks require execution of operations in the table, they will operate the rowMutex for each row. This separation allows us to maintain the critical sections associated with these operations independently.

Locks will use:

- addRow, addCol: A Mutex type lock that handles the addition of a new row/column, which increases the table size and adjusts the parameter values.
- isThereReadersLimit: A Boolean type of lock used to determine whether the table is currently locked in a search operation or not.
- doesSheetlock: A Boolean type of lock used to indicate whether the table is locked.

Operations in the Table: All the operations in the table will be divided as follows: when there's a need for a new table (intended for new operations), it will use the addRow/addCol lock. Each table will operate according to a Mutex type lock to ensure that the doesSheetlock parameter is set to true and all the operations are performed correctly. If the parameter is false, the operation will fail.

Details of the Reading/Writing Split:

Reading/Writing Content Operations:

- At the start of every search content operation in the table, we check whether our spreadsheet is under the limitation set by the isThereReadersLimit Boolean parameter.
- If yes, the operation starts by activating the readers/writers semaphore to allocate a place for the required operation.

- The next operation will limit the manager to protect the maximum number of callers in the table, ensuring that no more than the maximum number is called. This check prevents the manager from being activated beyond its capacity.
- After the first operation completes, the manager will notify the system that the search operation can now be conducted internally.
- The writing operation begins after the previous search operation completes, ensuring there is no call for multiple users. The writing operation locks the parameter until the writing completes.
- Finally, the parameter will revert to reading and release the lock.

Writing Content Operations:

- At the beginning of the writing operation, our table ensures that all the operations have stopped and no new operation can be initiated until the current one completes.
- For example, if a user wants to swap two rows in the table and we don't want to release the swap to ensure that the change happens together with the previous row. Therefore, the parameters for these operations will be treated as atomic operations.
- Each row/column work in accordance with the functional requirements and utilize an atomic lock which requires taking one request without interruption. For example, preventing a deadlock situation that can occur if two functions write different values and want to use the same row at the same time.

Table Operation Mode and Cost Waivers:

- Our **spreadsheet** can support multiple simultaneous operations. Some operations are atomic and prevent corruption, and there are other operations for which we need to execute cost waivers for what is more important to happened – a whole table lock or the changes it may change. In all queries, a mark will indicate that the content given may have changed. In all search operations, we allow multiple users to perform searches simultaneously on the table – both in rows/columns and on the same rows/columns.
- The state will be such that no changes will be performed on the content read/search operations in the rows/columns, and no changes can be made to the contents of the columns/rows accordingly.
- Locks can start checking where to save the change the user wants to make in its entirety and not start when it is simultaneous. Therefore, as more items are accumulated in the locks, we find that every item performed will be operations (writing content B-1->A) in another manager that hides this so that there will be no situation where the reading is released to one of the locks we have.
- To try and solve this problem, we decided to act as follows: start locking the columns/rows in an atomic manner. The search operation on a column/row will be locked hierarchically, each time another row. The hierarchies will be executed every time the search and release are performed. This will ensure that content is not duplicated in them and that the cost waivers are consistent with our costs.

Error Handling:

During the table operation, we constantly check that our values are accurate and not corrupted. For example, the value of my row/column number, and so on, we ensure their correctness accordingly.