a.

```
public class MySemaphore
{
    private readonly int maxCount;
    private int count;
    private readonly Mutex mutex;

    public MySemaphore(int initialCount, int maxCount)
    {
        if (initialCount < 0 || initialCount > maxCount)
        {
            throw new ArgumentOutOfRangeException();
        }

        this.maxCount = maxCount;
        count = initialCount;
        mutex = new Mutex();
    }

    public bool WaitOne()
    {
        bool waitSuccessful = false;

        mutex.WaitOne();

        if (count > 0)
        {
            count--;
            waitSuccessful = true;
        }

        mutex.ReleaseMutex();

        if (!waitSuccessful)
        {
            // If the count is 0, wait until signaled by another thread
            bool acquired = false;
            while (!acquired)
            {
                mutex.WaitOne();
                if (count > 0)
                {
                    count--;
                    acquired = true;
                    waitSuccessful = true;
                }
                mutex.ReleaseMutex();
            }
        }

        return waitSuccessful;
    }

    public void Release(int releaseCount=1)
    {
        mutex.WaitOne();
        count = Math.Min(count + releaseCount, maxCount);
        mutex.ReleaseMutex();
}
```

b. Create 2 semaphores which initialized to 0:

```
T1:
      //  code #1 S1;
      Signal (synch1)
T2:
      Wait(synch1)
      // code #2 S2;
      Signal(synch2)

T3:
      Wait(synch1)
      Wait(synch2)
      // code #3 S3;
```

c. Dekker and Peterson comparison:

Dekker algorithm:

•Utilizes turn-taking and flag variables to achieve mutual exclusion.

•The algorithm assumes that there are only two processes competing for the critical section.

•Each process explicitly indicates its desire to enter the critical section by setting its flag variable.

•Processes take turns based on a predefined protocol, allowing one process to enter the critical section while the other waits.

•The algorithm suffers from the problem of mutual exclusion violation when both processes try to enter the critical section simultaneously (a race condition).

Peterson’s algorithm:

•Also employs turn-taking and flag variables, similar to Dekker's algorithm.

•Designed to handle the critical section problem for any number of processes.

•Uses an additional variable called turn to decide which process should enter the critical section next.

•Each process sets its flag variable to indicate its desire to enter the critical section.

•Processes alternate between requesting and granting access to the critical section based on their turn and flag variables.

•The algorithm guarantees mutual exclusion and progress, but it may suffer from busy waiting if the other process does not release the turn.

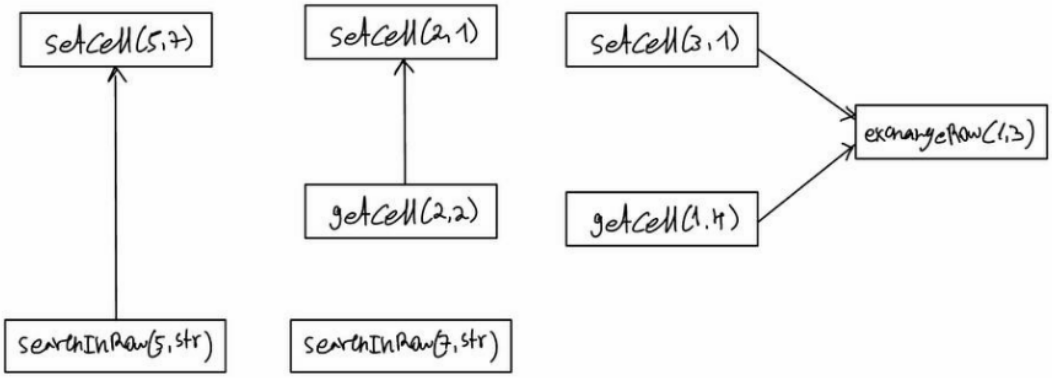
2. הסבר עיצוב: כדי להימנע מ-deadlocks ולפתור את בעיית ה-bounded buffer מימשנו את האלגוריתם של dijkstra's אשר הוצג בהרצאה ובכך התכנית מקיימת multithread לפעולות מסוימות ובנוסף thread-safety.
מצ"ב פירוט ותיעוד הפונקציות produce ו-consume מתוך האלגוריתם כפי שאנחנו מימשנו.
מימוש הפונקציות בקוד שלנו:

```
public void Produce()
{
    empty.WaitOne(); // Wait for an empty chair

    mutex.WaitOne(); //start critical section
    int customerNumber = Interlocked.Increment(ref count);
    int index = Array.FindIndex(resrestaurantBuffer, seat => seat == 0);
    if (index >= 0)
    {
        resrestaurantBuffer[index] = customerNumber;
    }

    freeChairs = freeChairs - 1; ;

    mutex.ReleaseMutex();// end critical section

    full.Release(); // Signal that a slot is filled with a value
}
```

```
public void Consume()
{
    full.WaitOne(); // Wait for a custumer that want to get out from the resturant

    mutex.WaitOne();
    int index = Array.FindIndex(resrestaurantBuffer, seat => seat != 0);
    if (index >= 0)
    {
        int customerNumber = resrestaurantBuffer[index];
        resrestaurantBuffer[index] = 0;
        int waitingTime = count - customerNumber;
        totalWaitingTime += waitingTime;
    }
    freeChairs = freeChairs + 1; ;

    mutex.ReleaseMutex();

    empty.Release(); // Signal that a slot is emptied

}
```

3. הסבר עיצוב: כדי להימנע מ-deadlocks מימשנו את הפתרון reader writer problem ככה שאפשרו לכמה קוראים לגשת למידע בו זמנית ומנענו מצב שבו כמה כותבים ניגשים לאותו האובייקט בו זמנית ומבצעים כתיבה ובנוסף מנענו מצב של כותב שרוצה לגשת לאובייקט שקיים קורא עבורו. השתמשנו באובייקט readerWriterLockSlim ככה שעבור כל שורה הגדרנו מנעול מהסוג הזה. בצורה זו אפשרנו לכמה טרדים במקביל לבצע כתיבה עבור שורות שונות (במקום נניח לנעול את כל ה-spreadsheet) וייעלנו את המקביליות של הפונקציות שעובדות על ה-spreadsheet. בעצם השתמשנו במנעולים ככמות השורות בטבלה.

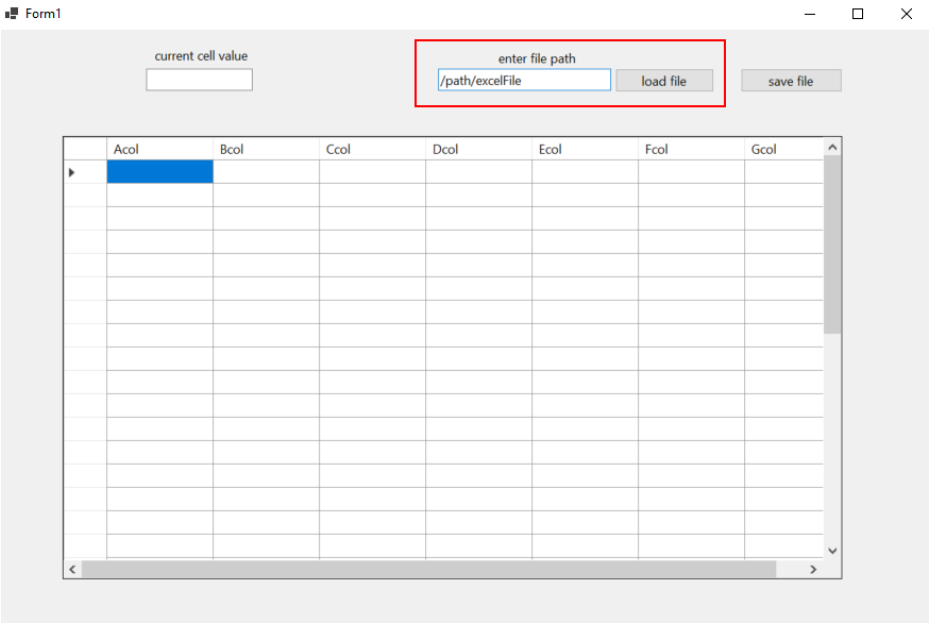
דיאגרמה: הדיאגרמה מתארת snapshot מסוים של מקביליות הפונקציות והתלויות של הפונקציות לפי המנעולים שכל פונקציה משתמשת כדי למנוע קריאה\כתיבה מלוכלכת ולהשיג מקביליות. לדוגמא הפונקציה searchInRow משתמשת במנעול read של שורות שונות והפונקציות setCell גם יכולות לרוץ במקביל כי הן משתמשות במנעול כתיבה עבור שורות שונות.

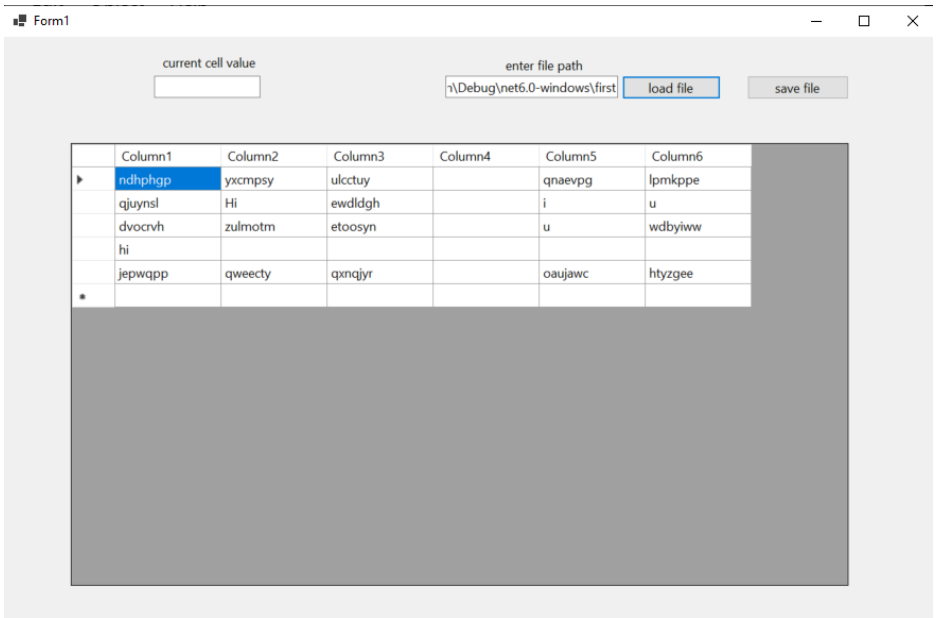


5. ה-GUI תומך בפעולות הבאות:

1. העלאה של קובץ xlsx והצגה שלו באפליקציה כ-datagridview

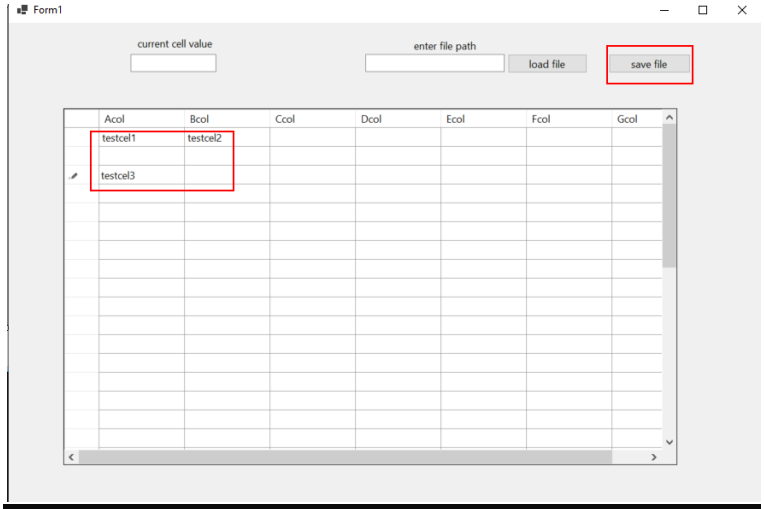
- כתיבת מיקום מלא של הקובץ
- לחיצה על load

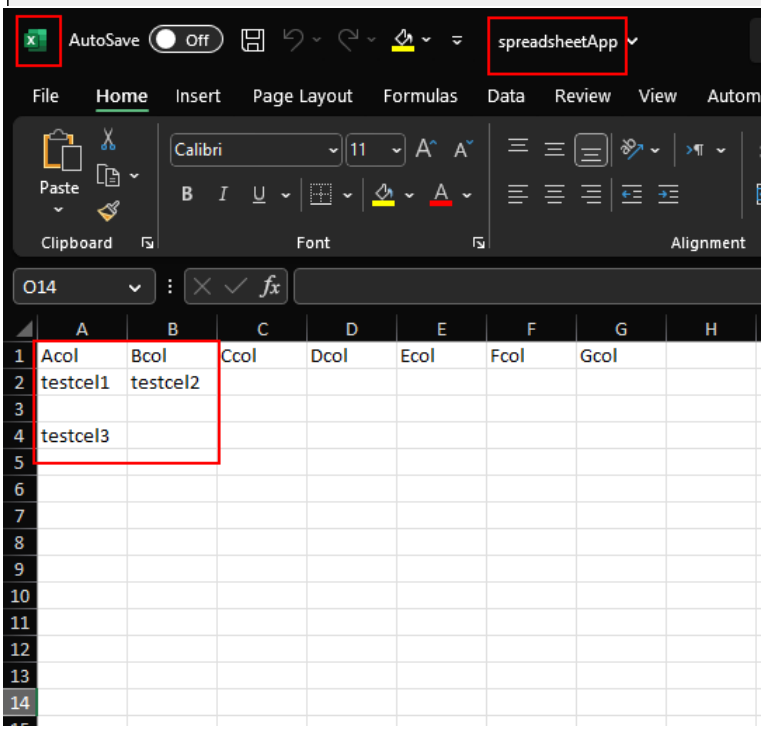




2. שמירה ה-datagridview הנוכחי באפליקציה כקובץ excel.

- לחיצה על save





3. הצגה של ערך התא הנוכחי:
- לעמוד עם העכבר על התא