SAINT MARY'S UNIVERSITY SINCE 1802 | FACULTY OF SCIENCE

CSCI 3431: Operating System
Winter 2023
Project 2- Synchronization and Deadlock Detection

Date out: March 8, 2023
Due on: April 4, 2023 (11:59 p.m.)

Instructions:

- Final submission should include the code, which is compilable and runnable, a report describing the approach, results and discussion, any innovative features added, reasons for failure (if any) and References (important).
- Cite in the report if you have adapted your algorithm from any web resources/textbooks/papers. As the solutions to some of these problems are already available in the web, it is perfectly fine to refer to them and understand the context and their approach. However, I strongly encourage you to write your own code from the scratch. That is the only way you can practice and get more insights into the topic and the OS in general.
- Final submissions should be a zip file (code and report) and to be uploaded to the MS Class Teams before 11:59 pm on the due date.
- Name your file following this convention: CSCI3431-<*LastName*>
- During the evaluation, the students are expected to download the zipped file from the MS Class Teams and show the results on either their laptop or the lab machine using screen sharing. The evaluation will be done in the following two/three recitations or office hours.
- You may work in groups of two, if you wish to. If you plan to work in a group, you must inform the instructor about the team details as early as possible. Team members should divide the tasks meaningfully.
- During the evaluation, each team member will be asked to explain your role and contribution to the project.

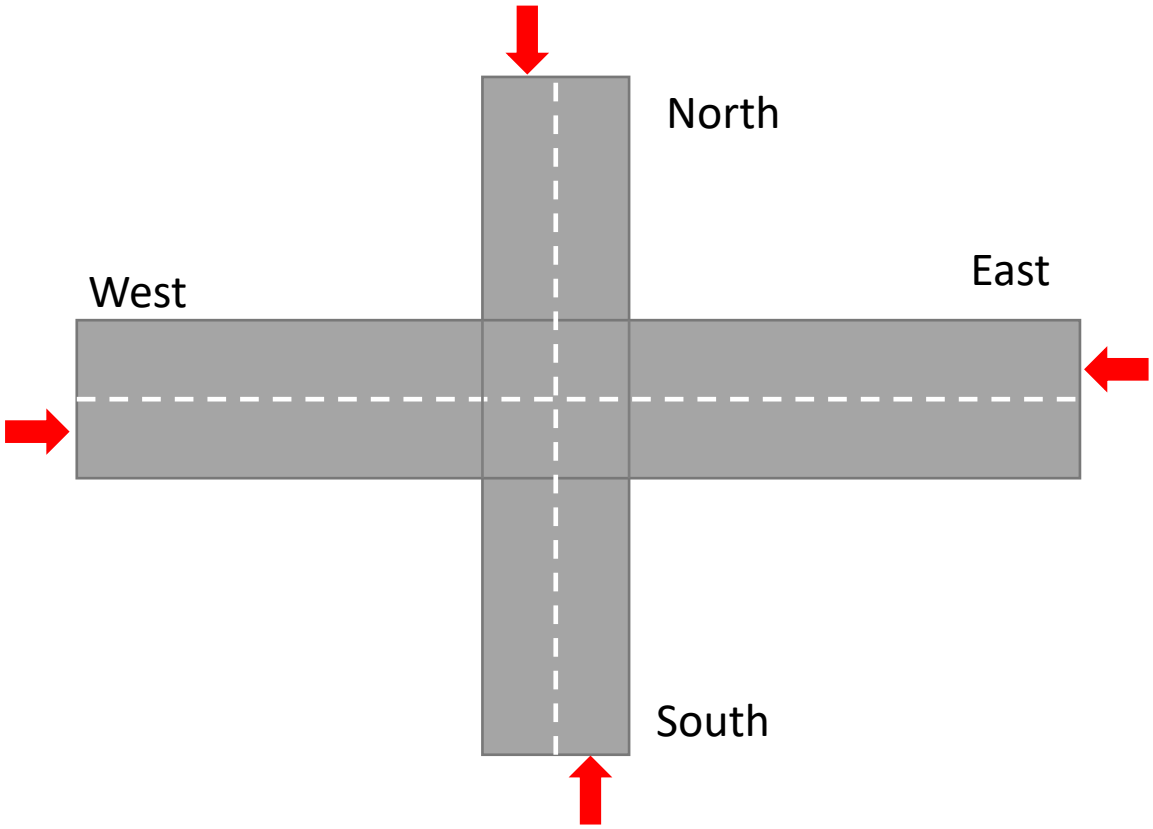
Background:

In the recent lectures, we discussed about deadlock, conditions for deadlock, deadlock detection and recovery, deadlock avoidance, and deadlock prevention. A few examples of problems involving deadlocks will be given in your next assignment (A6) for practice. This project is designed to make you more familiar with deadlock concepts and synchronization. You will learn how to design and develop synchronized multi-threading applications to detect deadlock conditions that may arise not only in the systems environment but also in the real-world scenario. Synchronization primitives such as semaphores and graph data structures such as resource allocation graphs will be utilized to accomplish the goals. The problem definition follows.

TRAFFIC MANAGER:

Problem Definition:

Assume, you have the below junction-crossing scenario. Buses may come from all the four different directions. You have to implement a system where it never happens that more than one buses crosses the ***junction*** at the same time. Every bus coming to the junction, waits if there is already a bus at the junction from **its own direction.** Each bus also gives a precedence to the bus



coming from its **right side** (for example, right of North is West) and waits for it to pass. You also have to check for deadlock condition if there is any.

Each bus is a separate process. Your task will be to create a traffic manager which creates those processes and controls the synchronisation among them.

Semaphores:

You have total two types of semaphores. One for ensuring the mutual exclusion at the junction to cross and the other for synchronising the buses. There are four synchronising semaphores for buses coming from 4 directions- North, South, East and West.

Manager process (manager.c):

1. It takes as input a file sequence.txt with sequence of directions of buses coming to the junction. The input file format is the following - If it contains "NWEWS", it means 5 bus processes will be created in the order North-West-East-West-South.

2. It also creates a file matrix.txt which contains a matrix of size n X m where n is the number of buses read from sequence.txt and m is the number of synchronising semaphores used (4 in our case). Each entry (i,j) of that matrix can have three values:

- 0 => bus i has not requested for semaphore j or released semaphore j
- 1 => bus i requested for semaphore j
- 2 => bus i acquired lock of semaphore j.

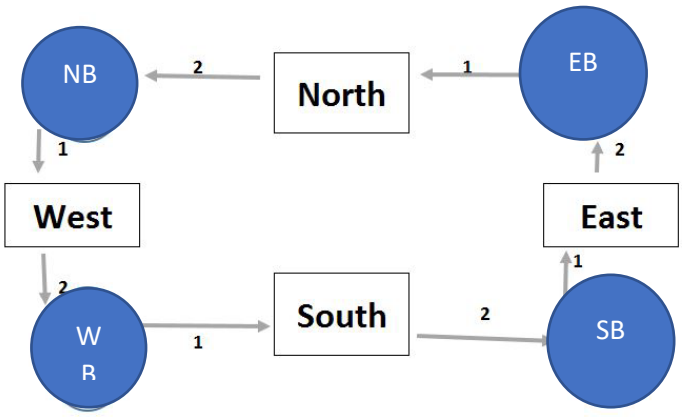
Initially all entries of the matrix are set to 0. When any bus process releases a semaphore, the corresponding matrix entry goes back from 2 to 0.

3. At each iteration with *p* probability it checks for deadlock in the system and with *1-p* probability it creates the next bus process as per the direction specified in the input file. Once all the bus processes specified in the input file are created, the manager only checks for deadlock periodically (at 1 second interval).

4. Deadlock detection: If every bus (process) waits for a bus coming from its right-side to pass, the system gets into a deadlock. Assume the semaphores to be the resources and the buses as the processes and generate the resource allocation graph (use the same procedure we discussed in the lectures). The manager process checks for cycle in the resource allocation graph built from the matrix present at **matrix.txt**. A possible deadlock scenario is the following. Assume there are four buses NB, WB, SB & EB coming from four different directions and four semaphores are East, West, Nor

	Semaphores			
	North	West	South	East
BUSES				
NB	2	1	0	0
WB	0	2	1	0
SB	0	0	2	1
EB	1	0	0	2

For the above matrix following is the Resource allocation graph. Rectangle is the resource, i.e., semaphore and circle is the bus, i.e., process. Edge weight 1 means the process has requested for the resource and edge weight 2 means the process has acquired the resource. Now we can see there is a cycle in the following graph for the above matrix, hence deadlock exists.



After detecting the deadlock, the manager prints the cycle.

Bus process (bus.c):

The steps followed by a bus coming from North are the following:
1. It waits at the "North" semaphore.
2. Then it waits at the "West" semaphore.
3. Finally it checks for mutual exclusion at the junction.
4. It crosses the junction. The time taken to cross a junction is implemented by a sleep of 2 second.
5. When the bus leaves the junction, it releases the mutual exclusion lock and the "North" semaphore.
6. Whenever the bus requests for, acquires or releases a semaphore it updates the proper (i,j) th entry in the matrix.txt. The mutual exclusion while accessing this file should be ensured (you might use one more semaphore to ensure this).

Analysis:

Vary *p* (taken as command line argument) from 0.2 to 0.7 in scale of 0.1 and print the following.

i) (bus) Process creations.
ii) The buses arriving, crossing & leaving the junction.
iii) Deadlock detection & corresponding cycle.

Sample input and output:

A sample output:

……..

Bus <pid1> North bus started

Bus <pid2>: West bus started

Bus <pid1>: requests for North-Lock

Bus <pid1>: Acquires North-Lock

Bus <pid2>: requests West-Lock

Bus <pid2>: acquires West-Lock

Bus <pid2>: requests South-Lock

Bus <pid2>: acquires South-Lock

Bus <pid1>: requests West-lock

Bus <pid2>: requests Junction-Lock

Bus <pid2>: acquires Junction-Lock; Passing Junction;

Bus <pid2>: releases junction-lock

Bus <pid2>: releases West-lock

Bus <pid1>: Acquires West-Lock

Bus <pid1>: requests Junction-Lock

Bus <pid1>: acquires Junction-Lock; Passing Junction;

Bus <pid1>: releases junction-lock

Bus <pid1>: releases North-lock

……...

System Deadlocked

<print the cycle >

(for eg. - Bus<pid10> from North is waiting for Bus <pid11> from West ------

---> Bus <pid11> from West is Waiting for Bus <pid12> from South **--------->**

Bus <pid12> from South is Waiting for Bus <pid13> from East **--------->**

Bus <pid13> from East is Waiting for Bus <pid10> from North)

Note:

"North bus started" means a bus process is created and it is coming from North.

3 sample input sequences:

i) NNSSNSS
ii) NWESNWESNWES
iii) NESWNESWNESW

Suggestions

- Start early!! If you are not comfortable with the language/concepts, it may take you a bit longer to implement.
- Backup your work frequently. It's possible (and most likely) you go try a new feature and your program crashes!
- Document your work properly.

Grading Scheme:

Program compiles, runs	25
Code quality	15
Shows the expected output	25
Report	20
Demo & Viva	15