UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

CSC3002F: Operating Systems Assignment 2

Social Distancing Shop Simulation

Process Synchronization

By: Zukiswa Lobola (LBLZUK002)

Lecturer: Michelle Kuttel

Department: Department of Computer Science

Date: 22 June 2020

Introduction

Running multiple programs in an operating system concurrently can be prone to errors when these processes have access to and change the same shared resources. Processes thus often need to coordinate with each other in order to gain access to these resources without interference. This coordination of access to shared resources is known as process synchronization.

In this assignment, we develop a multithreaded java simulation model of social distancing shop (amid the covid-19 pandemic) that mimics constraints of the multithreaded operating system described above. We utilize synchronization mechanisms such as semaphores and atomic variables to solve the problem of using shared resources also ensuring liveliness and protection against deadlock.

Threads running in the program

In our social distancing shop simulation, many customers visit the shop, however, only a restricted an amount are allowed in the shop at a given time. Since there are many customers who are unpredictable and wish to access the same parts of the shop such as the entrance, browsing, checkout and exits – customers thus are represented by the $customer$ threads.

In order to display the action occurring within the shop there is a threaded panel to display shop represented by the $Shopview$ threads.

In order to ensure that the restrictions are adhered to, the counters for the people waiting outside, inside, and have left the shop are necessary. These counter updates are thus represented by $counterDisplay$ threads.

The supervision of the adherence of the social distancing regulations by customers in the shop are also necessary. This inspection is done by the $inspector$ threads running in the program.

Classes shared amongst threads

- The $SocialDistancingShop$ class controls the creation, start and termination of all the threads running in the program. Hence, this class is shared with all the threads.
- The $PeopleCounter$ class keeps track of program counters such as the people that are inside, outside and have left the shop. This class is shared with the $counterDisplay$ threads.
- The $CustomerLocation$ Class that keeps track of the customer locations in the shop grid.
- The $GridBlock$ class to represent a block in the shop.
- The $ShopGrid$ represents the shop as a grid of gridblocks

Synchronization mechanisms added to designated classes

PeopleCounter Class

- A **Binary Semaphores** was added to this class as a lock to protect the *critical sections* of the code as the semaphore object named $mutex$. Critical sections refer to the segments of the code where several threads need mutually exclusive access to shared resources/data in that

region. Mutually exclusive access is necessary to avoid data inconsistencies where any alteration to these resources affects the outcome of this resource for the other threads.

These locks were used in the $personArrived()$, $personEntered()$ and $personLeft()$ methods to protect the sections which update the class counters that all threads have access to. When a customer arrives, enters and leaves the shop, it affects the $peopleOutside$ and $peopleInside$ class variables. Only one thread should update these class counters at a time to maintain data consistency and accuracy throughout the program.

- **Counting Semaphores** were added to this class to act as a buffered barrier which grants a fixed number of threads access to certain sections of the code at a time. Every time a customer enters the shop the counter for $peopleInside$ increases (and $peopleOutside$ decreases), and the space available for more customers decreases until it is full and another customer cannot enter unless another customer leaves the shop.

  The problem described is known as the producer-consumer problem, where the customers entering the shop are the producers who add data ($peopleInside$) into the buffer until it becomes full and can only be removed by the $peopleOutside$ from the buffer by consumers leaving ($peopleLeft$) who remove data (Downey, AB. 2016:55-63). The restriction on the maximum number of customers inside the shop at a given time acts as the buffer.

GridBlock Class

- A **Binary semaphore** was added to this class as a mutual exclusion lock to protect the updates to the $classCounter$ variable. This counter variable serves as the ID for each customer thread running. This section needs protection in order to prevent data inconsistencies which may occur when other threads access and update the counter at the same time, consequently resulting in customer threads with the same ID.

- **Atomic variable** – the $isOccupied$ Boolean variable was changed from a $boolean$ to an $atomicBoolean$. Each block in the shop can be occupied by a customer only if it is not occupied by another to obey the social distancing requirement. Since all customer threads can change this resource, its current value always needs to be automatically updated so that customers wishing to occupy the block have the correct status.

ShopGrid Class

- **Binary semaphores** were added to this class to protect critical sections i.e. acquire access to the entrance block when customer threads enter the shop and customer threads moving between grid blocks. These sections require mutually exclusive access to prevent errors and obey the social distancing regulations.

CustomerLocation Class

- **Atomic Variables –** there were no changes necessary for this class as atomic variables were already in place. Atomic primitives allow for automatic updates to be made on public variables that change frequently.

Liveness in the code

Liveliness is the ability of a multithreaded program to run in a timely manner where all thread requests are eventually granted. Since the liveliness of a program is dependent on the scheduling of thread execution, semaphores were used in the simulation to ensure liveliness in the program.

The semaphores added in the program allow the threads to take turns executing critical sections of the program with the use of mutual exclusion locks. While these threads have to wait for each other to run these sections of the code, liveliness is still maintained because the threads eventually get their requests granted. This demonstrates freedom from not only starvation, but deadlock-freedom as well.

Protection against deadlock

Deadlock is a major problem in concurrent programming (Gomez, E. 2010:3). Deadlock occurs when two or more threads are blocked forever, waiting for the other thread to release the resource it needs in order to release the resource it has. Deadlock can occur when all of the following conditions are true :

i) Mutual exclusion: one thread accesses a resource at a time
ii) Hold and wait: a thread holding a resource is waiting to hold another resource held by other threads
iii) No pre-emption: the held resource can only be released once it finishes the task
iv) Circular wait: a set of threads each hold a resource that another resource is wating for

In this assignment, protection against deadlock was necessary because there are scenarios that are prone to deadlock, such as customer threads occupying a grid block that another customer wants to occupy and vice versa all while holding a lock – resulting in circular wait. This was solved by allowing only one thread to acquire a lock while moving to a different block in the $move()$ method of $ShopGrid$ class. Two different locks were used to protect critical sections in the $ShopGrid$ to prevent a deadlocked state from occurring when a customer thread enters the shop and wishes to move to another block.

It was also important not to place the acquirement and release of the bounded-buffer in-between a mutex lock as it could result in a deadlock if the buffer is empty (Downey, AB 2016:61). This would result in the mutex getting blocked and the buffer because the buffer wouldn't get released.

These precautions described above ensure that at least one of the conditions required for deadlock to occur cannot hold as a preventative measure.

Conclusion

In this assignment, we have seen that process synchronization is a necessary and powerful tool for the coordination of a processes when accessing shared resources using semaphores and atomic primitive variables. We have also discussed how to prevent deadlock while maintaining liveliness in our social distancing shop simulation.

REFERENCES

Downey, Allen B. 2016. *The Little Book of Semaphores*. Second Edition (Version: 2.1.1). Available: https://cis.temple.edu/~qzeng/cis3207-spring18/files/LittleBookOfSemaphores2016.pdf [2020, June 21]

Gomez, Ernesto & Schubert, Keitth. 2010. *Algebra of Synchronization with Application to Deadlock and Semaphores.* School of Computer Science and Engineering California State University, San Bernardino California. DOI: IJNC. 1. 202-208. 10.1109/IC-NC.2010.43.

Silberschatz, A. Galvin &PB. Gagne, G. 2014. *Operating Systems Concept*. Ninth Edition. Available: https://www.academia.edu/5264253/Operating_System_Concepts_9th_Edition [2020, June 21]