SEMAPHORES AND THREADS IN ALGORITHMS

By Mirak Bumnanpol 3320409. October 2021.

# Introduction

This assignment was to solve three problems by using semaphores and threads to design algorithms that prevented deadlocks and were starvation free. This report reviews how the programs were designed to enforce mutual exclusion and ensure the algorithm was deadlock and starvation free. This report also discusses how the algorithm was tested in order to ensure these aspects.

# ENFORCing MUTUAL EXCLUSION

Mutual exclusion is a program object that prevents simultaneous access to a shared resource. For Problem 1, a Semaphore was implemented into the algorithm that allowed only one WAR to cross the intersection at a time, thus enforcing mutual exclusion. Likewise for Problem 3, a Semaphore was utilised and limited to 3 as there are only 3 printer heads. However, this implementation did not work as the output goes up to 9 printer heads instead of limiting to 3.

# DEADLOCK AND STARVATION FREE

A deadlock is a situation in which two or more processes are unable to proceed because each is waiting for the other to do something. Starvation is a situation in which a runnable process is overlooked indefinitely by the scheduler; although it can proceed, it is never chosen.

For Problem 1, the use of Semaphores and Threads ensured that the waiting WARs remained locked and did not enter the intersection while there was an active WAR running. The release() method allowed for the lock on the WAR to be released and then locate the next WAR (by ID) to start going through the intersection. All of this ensured that the algorithm remained deadlock and starvation free.

Similarly, Problem 3 also utilised Semaphores and Threads to locate the next job to be printed and then print accordingly. Although it was technically deadlock and starvation free, the program was unable to use 3 printer heads and instead used 9, deeming the classification of deadlock and starvation free to be unreliable.

Problem 2 did not use any Semaphores and hence, unable to ensure that it was deadlock free. Several different methods were used but I was not able to implement any methods that worked in time. However, Threads were used for this problem and implementation of the wait() method ensured that the algorithm was starvation free.

# TESTING MUTUAL EXCLUSION, DEADLOCK AND STARVATION FREE

I tested these aspects by using the built-in debugger in the IntelliJ IDE. Testing was done often to ensure minimal bugs to the algorithm. There was nothing incredibly notable during the testing phase to note of however, testing Problem 3 was more difficult than testing Problem 1. This was probably due to the use of 3 Semaphores instead of 1.

# Conclusion

In summary, Problem 1 was the easiest out of the 3 problems to both enforce mutual exclusion and ensure the algorithm was deadlock and starvation free. Problem 3 was the most difficult due to the use of 3 Semaphores. Problem 2 was difficult to ensure it was deadlock free and enforced mutual exclusion due to the lack of Semaphores.