UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS226 - Concurrent systems

Practical 2 Specifications - Sequential Consistency

Release date: 12-08-2024 at 06:00

Due date: 06-09-2024 at 23:59

(FF will remain open until 08-09-2024 23:59 for technical difficulties)

Total marks: 800

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually; no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a zero mark. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with Java 8**
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent), and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers to avoid any misunderstanding.** Also note that the OOP principle of code reuse does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

On completion of this practical, you will have gained experience with the following:

- Sequential Consistency
  - Identifying execution orders which are sequentially consistent.

4 Introduction

4.1 Bounded Timestamps

In concurrency control, bounded timestamps play a crucial role in ensuring fairness and avoiding deadlock. Specifically, in the context of the Bakery algorithm—a mutual exclusion algorithm designed by Leslie Lamport—timestamps are used to order the execution of processes. The algorithm assigns each process a timestamp when it seeks to enter the critical section, ensuring that processes are granted access in the order of their timestamps, much like customers taking a ticket in a bakery and waiting for their number to be called.

Bounded timestamps ensure that the time a process has to wait is limited and that timestamps do not grow indefinitely, which is vital for practical implementations. This prevents integer overflow issues and keeps the system efficient by avoiding excessive delays or resource hogging by any single process. By bounding the timestamps, the Bakery algorithm ensures that all processes have a fair chance to execute, thereby maintaining both safety and liveness in a concurrent system.

Bounded timestamps can be reasoned about using precedence graphs as seen in figure 2.12 of the textbook (*The Art of Multiprocessor Programming*).

5 Tasks

You are tasked with implementing the bounded timestamp system described in chapter 2.7 of the textbook (*The Art of Multiprocessor Programming*) for any arbitrary number of threads.

A bounded timestamp has an array of length *n-1* (where n is the number of threads expected to be used). At each index of the array, there is a ternary digit representing the node of the precedence graph to which the timestamp corresponds. For example, the array [2,0,1] indicates that the timestamp corresponds to sub-graph 2, sub-sub-graph 0, node 1. Arrays are used instead of strings for ease of casting. The textbook would use 201 instead. See Figure 2.12 in the textbook showing the precedence graphs for 1, 2 and 3 threads.

You have been given the skeleton for the `BoundedTimestamp` class. This class has two constructors. One creates a timestamp from a passed-in array while the other creates a timestamp of length n set to [0,0,0...] (up to the *n-th* 0). Recall, that the length of the timestamp is always one fewer than the maximum number of threads expected to be used.

You will implement two methods. First, the `compare()` method, which is used to compare `BoundedTimestamps`. Second, the `getNext()` method which, given an array of timestamps, returns the smallest next legal `BoundedTimestamp`.

Only your implementation of BoundedTimestamp will be assessed for marks. However, a BakeryLock which uses the BoundedTimestamp system has been provided for your understanding.

5.1 Task 1 - The `compare` method

Implement the compare method. This is a non-static method which compares `this BoundedTimestamp` to some `other BoundedTimestamp`. It returns 1 if this timestamp comes before the other (i.e. is earlier than, i.e. is pointed to by the other). It returns -1 if this timestamp comes after the other (i.e. is later than, i.e. points to the other). It returns 0 if the timestamps are equal.

Recall from figure 2.12 that 0 comes before 1. 1 comes before 2. 2 comes before 0. Where the digits closer to index 0 time the timestamp array are used to compare first. e.g. [0,1] is before [1,0] and [2,2,1] is before [2,0,2] but after [1,0,0]. Therefore:

0,1 .compare([1,0])==1

1,0 .compare([0,1])==-1

2,2,1 .compare([2,0,2])==1

2,2,1 .compare([1,0,0])==-1

2,2,1 .compare([2,2,1])==0

5.2 Task 2 - The `getNext` method

This should return the next smallest legal timestamp given an array of timestamps (in this context smallest refers to ternary value not the "compare to" order, so [0,0] is smaller than [2,2] in this context). A "legal" timestamp is a timestamp that does not cause a cycle. Note that you should ignore the timestamp at the index that is requesting a new timestamp when calculating the next biggest and that [2,2,2,2,2....2] rolls over to [0,0,0,0,0....0].

For example, given the array [[0,0],[0,0],[0,1]] with thread 1 requesting a new timestamp, the next smallest legal timestamp is [1,0]. It is not [0,2] as this would cause a cycle (since [0,0] is before [0,1], [0,1] is before [0,2] but [0,2] is before [0,1]). It is also not while [1,1] also does not cause a cycle it is not smaller than [1,0] in ternary value.

Another example: Given the array [[2,0],[1,1],[2,1]] with thread 0 requesting a new timestamp, the smallest legal timestamp is [2,2] since we need the smallest timestamp that waits for/is after both [1,1] and [2,1].

6 Upload checklist

The following files should be in the root of your archive

- `BoundedTimestamp.java`
- `Main.java` will be overridden

(BakeryLock, Marker and Paper are given for your understanding)

7 Allowed libraries

These libraries will be allowed by the automaker but you must still do the given tasks with your own code (i.e you may not use the libraries to complete the task for you)

- `import java.util.*`

8 Submission

You need to submit your source files on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. Your code must be able to be compiled with the Java 8 standard.

For this practical, you will have 20 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**