CS 131 Homework 3 Report

Siddharth Joshi

Abstract

This report evaluates concurrency in Java using the various constructs the language provides to deal with the issue of race conditions and the undefined behaviour that may occur as a result. A simple simulation is used to benchmark and compare the performance of synchronized (those that assure sequential consistency) and unsynchronized classes (those that don't) in terms of their runtime and reliability. The principle idea is to analyze the trade offs of inserting sequential checks to achieve greater reliability with the objective of faster runtime. This report also justifies the choices made while implementing a more efficient synchronized class by comparing a few of the packages that provide support for ensuring sequential consistency in Java. The study concludes with a summary comparing the various implemenations of the simulations in terms of being DRF (Data Race Free) and suggests which implementation may be ideal for the corporation GDI.

1 Introduction

Concurrency is a critical part of every modern program and thus is one of the most meaningful criteria with which to evaluate the effectiveness of a programming language. The fundamental challenge in applications relying on concurrency is maintaining correctness or more formally sequential consistency i.e. the concurrency mustn't break the necessary ordering of operations and result in unpredictable behavior. The reason this concern arises is due to the asynchronous nature of the parallel computing where threads can and do execute and differing and unknown speeds, thus it is impossible to predict where another thread may be while one is executing. In general, in keeping with the field's fixation with worst-case analysis, computer scientists generally assume the worst possible interleaving of operations and attempt to guarantee sequential consistency in the case of a such an event as well. The focus of this report is rather different from this notion of an absolute guarantee of correctness. In this report, the aim is to evaluate various degrees of ensuring sequential consistency and analyze the performance tradeoffs this results in.

2 Empirical Testing

2.1 Runtime v/s Number of Threads

Model	Time per Transition			DRF
	8 Threads	16 Threads	32 Threads	
Null	0 ns	0 ns	0 ns	Yes
Synchronized	0 ns	0 ns	0 ns	Yes
Unsynchronized	0 ns	0 ns	0 ns	No
GetNSet	0 ns	0 ns	0 ns	No
BetterSafe	0 ns	0 ns	0 ns	Yes

2.2 Runtime v/s Number of Threads

Model	Time per Transition			DRF
	100 Elements	500 Elements	1000 Elements	
Null	0 ns	0 ns	0 ns	Yes
Synchronized	0 ns	0 ns	0 ns	Yes
Unsynchronized	0 ns	0 ns	0 ns	No
GetNSet	0 ns	0 ns	0 ns	No
BetterSafe	0 ns	0 ns	0 ns	Yes

2.3 Runtime v/s Number of Threads

Model	Time per Transition			DRF
	8 Threads	16 Threads	32 Threads	
Null	0 ns	0 ns	0 ns	Yes
Synchronized	0 ns	0 ns	0 ns	Yes
Unsynchronized	0 ns	0 ns	0 ns	No
GetNSet	0 ns	0 ns	0 ns	No
BetterSafe	0 ns	0 ns	0 ns	Yes

3 Analysis of Packages with Support for Ensuring Sequential Consistency for the BetterSafe Class' Implementation

`java.util.concurrent`

The concurrent package provides a variety of functionality such as classes for sequentially consistent versions of classic data structures like Hash Maps, Skip Lists alongside rather advanced functionality such as *DelayQueue* and *ForkjoinPool* that gives the programmer far more control over the the behavior of the threads - not merely controlling them through simple data locks but actually programming their behavior when they are unable to execute or face some other concurrency challenge. While this functionality would undoubtedly be invaluable to any production level endeavor in the industry, it is definitely overkill for the simple simulation that is being run here and it is unclear if it contains any tools relevant for resolving the task at hand without adding unnecessary complexity. Thus the tools of this would not be well suited to the problem that the BetterSafe class attempts to resolve.

`java.util.concurrent.atomic`

atomics

`java.util.concurrent.locks`

`java.lang.invoke.VarHandle`

4 Challenges Faced

5 DRF Class Analysis