Avoiding Undesired Behavior while Concurrently Accessing A Data Structure

Jeremy Khawaja

April 25, 2017

Abstract

The key to avoiding inefficiency and inconsitency in concurrently accessed structures is having a well-defined concurrent access algorithm. There are many approaches available to use. We devise a a method using spatial and temporal dependency graphs to manage access to arbitrary data structures.

1 Introduction

In this paper we would like to describe an algorithmic approach to avoiding undesirable behavior in systems that have multiple processes (threads) accessing a shared data structure.

There aready exist a plethora of approaches to managing access to a shared structure, of which we will make brief mention of only a few techiques.

It might be good to review: *locks*, *fine-grained locking*, and *combining trees*.

It is also important to specify what exactly we mean by *undesirable behavior*. Essentially we are trying to avoid problems with inefficiency and inconsistency. Namely, we would like to avoid overwriting values that have already been written e.g. a counter being overwritten with the same value multiple times. And we would also like to avoid problems that are usually seen as a lack of linearization e.g. a value being overwritten with an out-of-date value.

Of course, every access management implementation will require a degree of tradeoff. For example, are we allowed to read a value at any time or must we always guarantee that all writes for that value have been completed? We will not focus on tradeoff choices in this paper. We focus primarily on constructing (a rather basic) non-blocking and strictly consistent concurrency management algorithm. How a developer may choose to utilize the algorithm will depend on the developers personal choice of system requirements. But we do note that the algorithm will be rather flexible in varying the degree of consistency for concurrently accessing a structure.

We will also address the atomicity of operations and sequence of operations as defined internally to a thread.

Def. 1.1 (Concurrent Data Structure) -

2 Dependency DAGs

3 Algorithm

How are we going to assign unique-independent modifications to a thread? In other words, how will we ensure that a thread is only allowed to make modifications to a single unique-independent?

4 Formal Verification

5 Memory

Can we utilize this new model to find a better method for Memory Management of Concurrent Data Structures (in runtimes that are not using a built-in GC)?

6 Conclusion

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.