Avoiding Undesired Behavior while Concurrently Accessing A Data Structure

Jeremy Khawaja

April 27, 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. We should first supply a basic definition for a concurrent data structure.

DEF. 1.1 (Concurrent Data Structure) - A *concurrent data structure* is a data structure that is concurrently accessed by multiple processes. The processes are allowed to read and write data as well as modify the structure itself.

There already exist a plethora of approaches to managing access to a shared structure, of which we will make brief mention of only a few techiques e.g. *locks*, *fine-grained locking*, *combining trees*, etc.

For example, here is a basic definition of locking a data structure.

DEF. 1.2 (Locking) - A *lock* is a device that enforces a temporary operation ownership of the concurrent data structure by a process e.g. a process in posession of the write lock will be the only process allowed to write data to the structure. Using a set of locks where each lock applies to only a substructure of the overall data structure is called: *fine-grain locking*.

In most programming languages a lock is called a *mutex* (mutual exclusion).

It is also important to specify what exactly we mean by *undesirable behavior*.

DEF. 1.3 (Undesired Behavior) - The primary potential behaviors we will be focused on avoiding in this paper are: redundancy, data races, access blocking, and a lack of any consistency guarantee.

Essentially we are trying to avoid problems with inefficiency and inconsistency. Namely, we would like to avoid overwriting values that have already been written. 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also like to avoid problems that are usually seen as a lack of linearization e.g. a value being overwritten with an out-of-date value.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 from our concurrent data structure at any time or must we always guarantee that all currently-in-progress writes have been completed? We will not focus on tradeoff choices in this paper. How a developer may choose to utilize the algorithm in this paper 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.

One of the usual goals of concurrently accessing a single data structure is: computation efficiency i.e. speedup. Sometimes it is also used to avoid having multiple copies of the same structure and being forced to use a consensus algorithm to determine which one has the "true state" of the structure (or a sub-structure) at a given moment in time.

These goals/reasons for concurrent data structure design will not be the focus of this paper. Instead, we will focus on devising an (astoudingly rather simple) algorithm that will be able to 'monitor' and maintain the state of an arbitrary concurrent data structure for us to a degree of consistency that we would like it to have.

2 Dependency DAGs

We begin our discussion around the crux of avoiding undesirable behavior: *Ordering*. In the literature: *linearization*.

In our defintion of a CDS (1.1), we discussed reading and writing data to a structure as well as changing the structure of the CDS itself. We will label all of these processes as: *access types*. More specifically: these are prcoedural types. Everytime we try to execute an access type we will call this an: *access procedure*.

We do not want to limit ourselves to some predefined set of operations such as "reads, writes, appends, removals, etc.". These do represent a sort of canonical basis for all possible access types, but we want to be flexible enough to allow for any and all possible methods of accessing a CDS.

Thus, we need to find a way to ensure the linearization, or correct ordering, of access procedures.

For example, when and how will we gurantee that one access procedure completes before another one is started? The how is simple: we can sequentially order our access procedures i.e. apply a total ordering to all access procedures across our system. But this can be accomplished by a single process. This brings us to addressing the *when* of ordering access procedures.

Not all access procedures will need to be totally-ordered i.e. a complete (mathematical) linearization of our access procedures is not a necessity. It is this lack of necessity that allows us to have the possibility of concurrency in the first place.

So, when is ordering a necessity? The only time two access procedures need an order is if one access procedure is *dependent* on the other. We do not specify the reason for the dependency, but if a dependency exists (at all) then it is easy to see that an ordering is necessary.

Since the necessity for an ordering is not always required, we see that a complete linearization of our system is not always a necessity. Rather, the access procedures which carry a dependency on other access procedures create a *partial ordering* for our system.

More specifically: all of our dependencies allows us to have a *dependency graph* for our access procedures.

All good, right?! Sort of. Sadly, this does not resolve all of our problems. We have not discussed exactly how we will order access procedures in terms of processes.

First, let's diverge quickly back into a discussion of our CDS. Let's try to formalize it:

DEF. 2.1 (Concurrent Data Structure (2)) - A *concurrent data structure* is a graph $\mathcal{G} = \{\mathsf{N}, \mathsf{E}\}$ composed of $|\mathsf{N}|$ nodes and $|\mathsf{E}|$ edges, that is used to index data of type $\mathcal{T}$, and is allowed to be accessed by some set of processes $\mathcal{P}_i$ via some set of access types $\mathcal{A}_t$.

This helps us to remember that a data structure is a spatial object. It carries a certain topology and is invariant through time unless modified. Thus, it is completely possible that while access procedures may not carry a dependency: nodes of our CDS just might! The set of all nodular-dependencies will be called the: *set of spatial dependencies*. These dependencies may be the dependencies that an access procedure "adopts" in order to consider itself dependent on another access procedure.

For example: if access procedure A is trying to write to node $\mathrm{n}_1 \in \mathsf{N}$ and access procedure B is trying to write to node $\mathrm{n}_2 \in \mathsf{N}$, but $\mathrm{n}_2 < \mathrm{n}_1$ (where $<$ is a dependency arrow), then we would want to perform B before A.

This set of spatial dependencies only covers the dependency relationships between certain nodes in our graph. But what if we wanted sub-graphs of our CDS to be dependent or more importantly *independent* of other sub-graphs of our CDS?

Let's define this:

DEF. 2.2 (Unique-Independent) - Any sub-graph $\mathcal{S} \subseteq \mathcal{G}$ that does not contain any internal spatial dependencies is called an *unqiue-independent* of our CDS.

Unique-Independents (UIs) can almost be seen as a *virtualization* of our CDS's spatial topology.

Unique-Independents can obviously have external dependencies i.e. be dependent on each other. Thus, the set of UIs of our CDS forms a dependency graph. Although, since UIs do not have to have other UIs as dependencies, we simply qualify this as a: dependencies list.

DEF. 2.3 (Dependency List) - A *dependencies list* for a CDS is the adjancency list of our disjointed graph of unique-independents.

Don't let the name "unique-independent" fool you. There is no requirement for an independent to be spatially-unique. In other words, a unique-independent can share nodes from the CDS with other UIs. The only thing that makes a unique-independent "unique" is that it is in totality unique as to the nodes it applies too.

So while, a unique-independent can be completely subsumed by another unique-independent spatially, it will never refer to the exact same set of nodes that another unique-independet is referring too. And it is this notion of "uniqueness" that allows it to be an independent.

This may seem like a unique-independent that subsumes UIs with dependencies that are other subsumed UIs conflicts with our definition (2.2) of an unique-independent.

However, what we have to notice is that a unique-independent is evaluated seperately from other UIs i.e. a unique-independent whom subsumes other UIs and their dependencies is only allowed to be addressed after all those other UIs have been addressed.

So, while an independent has "no spatial dependencies" it can still subsume UI-dependencies.

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?

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

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.