AbortLock for Java

David Chun (dc37875) and Grace Zhuang (gpz68)
EE382C Multicore (Prof. Vijay Garg)

Abstract

Critical section execution may not always proceed as anticipated, and a good programmer may wish to revert the global context or thread back to its pre-critical section state. This paper describes a versatile version of the standard lock, called an AbortLock, which allows threads to abort their execution and restore their state back to the point of lock acquisition. Using a combination of Java reflection and deep copying techniques, the following AbortLock gives programmers the ability to restore their thread to a pre-critical section execution state. Utilizing this implementation will allow for more functional error handling and thus more robust and higher quality code.

Introduction

The Monitor construct in Java has long been used as an intuitive, simple, and efficient method to ensure correctness among multi-threaded programs. With a simple lock object, users are able to ensure mutual exclusion when operating concurrent segments of work. For our term project, we extended this Monitor lock to include an `abort()` function, which enables users to restore the state of the thread to its original state when the thread first acquired the lock object.

In this report, we will start by detailing the background of the problem and the motivation behind the concept of AbortLock. From there, we will explain our design solution, including alternative implementations considered. Finally, we will describe our testing framework and results before wrapping up with final thoughts.

Problem Background

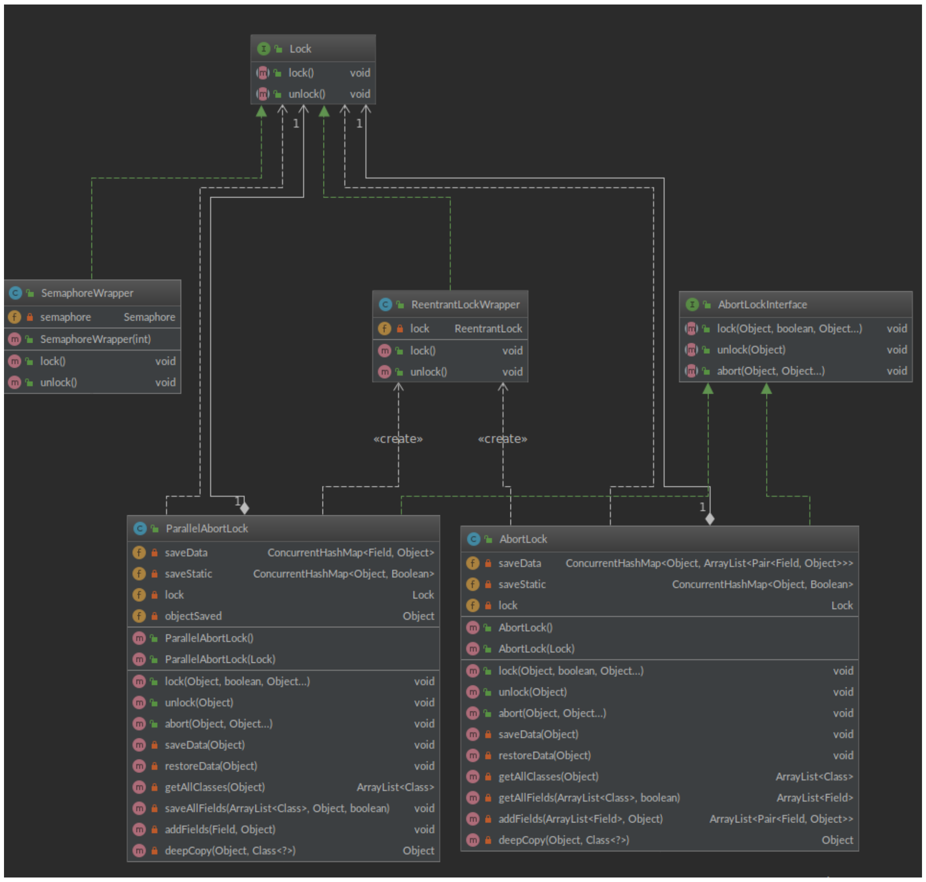
A Monitor object is created to be shared among multiple threads. This necessitates more complex frameworks within the object to ensure consistency and correctness of the state of the object. For example, locks ensure consistency by enforcing mutual exclusion on individual data fields. However, is correctness of individual data fields enough? Or will there be situations where we must consider the correctness of the object as a whole?

Imagine a situation where you have multiple threads utilizing a Monitor object. One thread enters the object, completes some computations, modifies some internal data fields, and then an exception is caught after one of the computations. The object is now in an inconsistent state. Half of the object's contents have been modified by this thread to the next state, and half of the contents remain in the previous state. Say the exception is handled with a simple handler routine, and the process is permitted to resume execution. Execution resumes with a different thread, which enters the Monitor object to find the object in a half-altered state. In this case, in order to revert the system back to a consistent state, we must "undo" the changes the first thread made in the object. However, in the current Monitor object, there's no way for a thread to recover the original or previous state of the object. This is where the concept of an AbortLock comes into play.

With an AbortLock, when a thread grabs a lock, the current state of the Monitor object (i.e. the state of the thread's internals) will be stored. If the thread needs to call `abort()`, the AbortLock will then restore the internals of the Monitor object to its original state before the thread began execution within the Monitor object. Sounds simple enough, but let's take a look at the implementation specifics.

Project Design

Our AbortLock is an extension of the ReentrantLock in Java. Its initialization, functions, and uses are identical to that of the ReentrantLock, with the exception of the additional `abort()` function. An object/class may create a static lock object for use among multiple threads, and threads can access critical sections by simply calling `lock()` and `unlock()` on the lock object created in class instantiation. Additionally, a thread may call `abort()` on the lock object, which will return the thread to its state before it began execution in a critical section by reverting each the contents of each field. The overall structure of the class and its internal hierarchy is extremely straightforward and mirrors what one would expect in a lock object. Figure 1 below shows a detailed class diagram, including internal methods and variable fields.



Lock
lock() void
unlock() void
1
1
SemaphoreWrapper
semaphore Semaphore
SemaphoreWrapper(int)
lock() void
unlock() void
ReentrantLockWrapper
lock ReentrantLock
lock() void
unlock() void
«create»
«create»
AbortLockInterface
lock(Object, boolean, Object...) void
unlock(Object) void
abort(Object, Object...) void
1
1
ParallelAbortLock
saveData ConcurrentHashMap<Field, Object>
saveStatic ConcurrentHashMap<Object, Boolean>
lock Lock
objectSaved Object
ParallelAbortLock()
ParallelAbortLock(Lock)
lock(Object, boolean, Object...) void
unlock(Object) void
abort(Object, Object...) void
saveData(Object) void
restoreData(Object) void
getAllClasses(Object) ArrayList<Class>
saveAllFields(ArrayList<Class>, Object, boolean) void
addFields(Field, Object) void
deepCopy(Object, Class<?>) Object
AbortLock
saveData ConcurrentHashMap<Object, ArrayList<Pair<Field, Object>>>
saveStatic ConcurrentHashMap<Object, Boolean>
lock Lock
AbortLock()
AbortLock(Lock)
lock(Object, boolean, Object...) void
unlock(Object) void
abort(Object, Object...) void
saveData(Object) void
restoreData(Object) void
getAllClasses(Object) ArrayList<Class>
getAllFields(ArrayList<Class>, boolean) ArrayList<Field>
addFields(ArrayList<Field>, Object) ArrayList<Pair<Field, Object>>
deepCopy(Object, Class<?>) Object

Figure 1: Class Diagram

One major complication in implementing an AbortLock is support for any type of object, regardless of form or internal data structure. Passing in a list of variables to save is inconvenient for the programmer and requires an understanding of the whole object stack, including opaque or even hidden third-party library implementations. To solve this issue, the AbortLock saves and restores data using Java reflection, providing the option of passing in other objects to save in addition if desired. In this manner, the programmer can efficiently save data on `lock()` and restore on `abort()`.

The way AbortLock operates internally is straightforward, allowing for easy extension and maintenance. Once the programmer passes `this` into the lock method, AbortLock iteratively finds not only the class of `this` but also all of its superclasses. Once complete, each field in all relating classes are discovered using reflection and added to a list. This list is used to operate on the `this` object to extract all values of these fields which are then saved via hashmap. Each object maps to a list of pairs containing a more detailed mapping from field to object saved. On `abort()`, the same `this` object must be passed through to properly restore the fields from the hashmap, again via reflection.

The key missing step in the previous explanation is what to do with non-primitive objects. Directly calling `field.get(save)` on an object merely copies the value of the reference, akin to what occurs when calling `println(someArray)`. This is referred to as a shallow copy. Any change to the underlying data structure will also change the value saved in the mapping and the restore will not work properly. To solve this issue, AbortLock deep copies the internal values of any complex object using Google's Gson library, ensuring a complete and separate copy for proper restoration. This is done by serializing each internal object recursively and making an entirely new copy of the object using that serialized data.

For convenience sake, AbortLock currently allows three configurable options in its method signature. Firstly, the programmer may pass in any lock implementing the Lock interface, allowing for flexible integration. Second, depending on execution pattern, point of abort, or lock type, the programmer might not want to restore static variables to the objects and thus can pass a `false` into the `saveStatic` argument of the constructor. Lastly, both `save()` and `abort()` are `varargs` methods used to pass in additional objects to save and restore as needed, to save method variables for example. These options give programmers the proper tools to cover for more complex program structures.

Alternatives

Many of the alternate implementations for AbortLock relate to the method of deep cloning objects. While a copy constructor, or similar such options (including the cloneable interface) are possible options, these should not be relied upon, not only because they are not guaranteed to be implemented across all objects (e.g. third-party libraries), but also because they are not guaranteed to be implemented correctly. A better alternative is serializing all the object variable data and making a completely new instance as a deep copy. To abstract this functionality, we explored a couple options and Google Gson emerged as the only viable option. Unlike other alternatives, it does not require objects to implement Serializable and does not need a default constructor. Using Gson to deep copy objects meets all the requirements, allowing for flexible and intuitive implementation for the programmer using AbortLock.

One other area we explored for potential alternative implementations was the method of saving and restoring data upon a `lock()` or `abort()` call. The time required to save and restore internal data fields in a thread object could prove costly as object size scales. To solve this issue, we implemented two versions of AbortLock: a sequential implementation and a parallel alternative based on Java streams. While the underlying mechanisms are identical for both as described above, the parallel AbortLock saves and restores all values in parallel via the `parallelStream()` method call, resulting in better, more consistent performance. We will examine precise performance results of these two variations in the following section.

Test Analysis

Testing the AbortLock involves structuring the executed thread code such that it returns a test object with internal values to verify in an assertion rather than relying on singular static variable returns. Switching to the Java Callable rather than simple Thread-based execution allows for a return object storing all data post-execution. This can then be used to check for expected values. The callable AbortLockCallable extends SuperAbortLockCallable to cover superclass variable restoration. Additionally, it contains static and non-static variables as well as complex object FancyObject. This FancyObject class extends SuperFancyObject to cover nested superclass variables while also holding most of the primitive types, an inner complex object SubFancyObject, and an array of primitives. These test for proper cloning of reference based objects as well as covering for a nested object edge case. All these variables are wrapped up at the end of execution into a TestObject wrapper holding the modified or restored thread data and is compared for equality to an expected TestObject created in the AbortLockTest test cases. Figure 2 below details the class diagram (minus class internals) for our testing framework, showing the hierarchical structure described above.

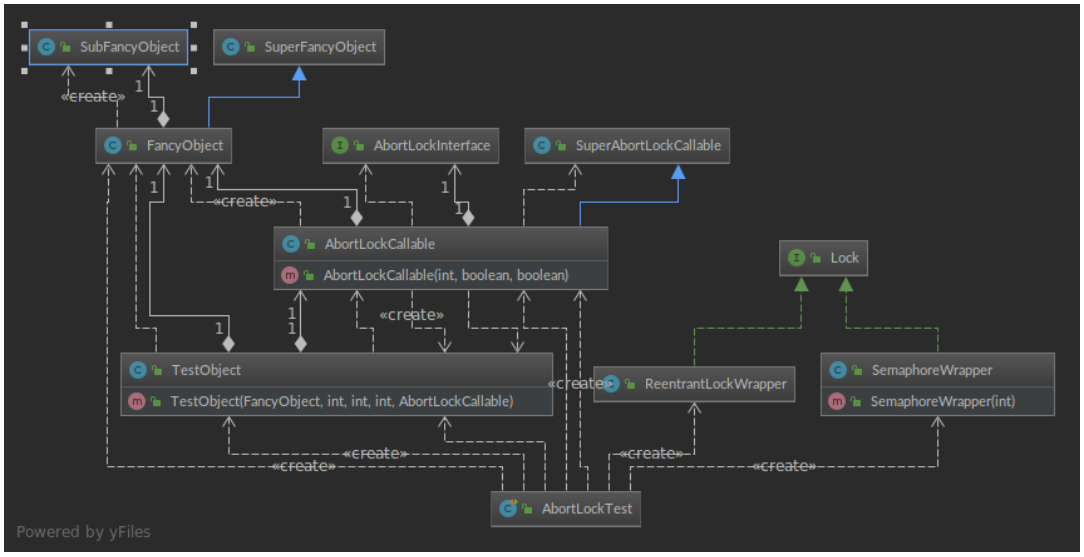


Figure 2: Testing Framework Class Diagram

The final round of tests examine performance as opposed to functionality. They compare runtimes and efficiency of sequential vs. parallel execution in AbortLock. While the previous tests ensure `abort()` works correctly, the tests with “parallel” or “variable” in their method names look at performance as total thread to aborted thread count parameters vary. The results confirm better, more consistent performance with the parallel implementation, although the advantages are more significant when aborted thread counts are low. Below, Figure 3 shows how execution time varies as the number of aborted threads increases, comparing the sequential and parallel versions to each other.

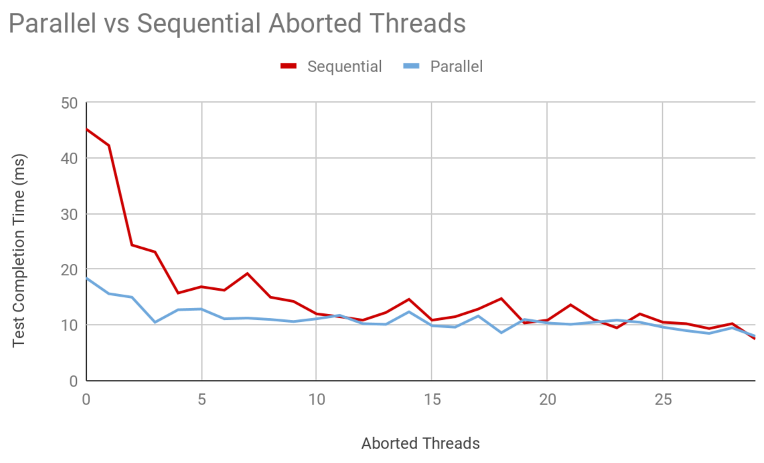


Figure 3: Graph of Sequential vs Parallel Performance as Aborted Thread Number Varies

Additionally, Figure 4 shows a more general workload testing `abort()` and `lock()` each with 1 thread executing the AbortLockCallable 10 times to show how execution time varies based on sequential vs. parallel implementation of `abort()` and `lock()`.

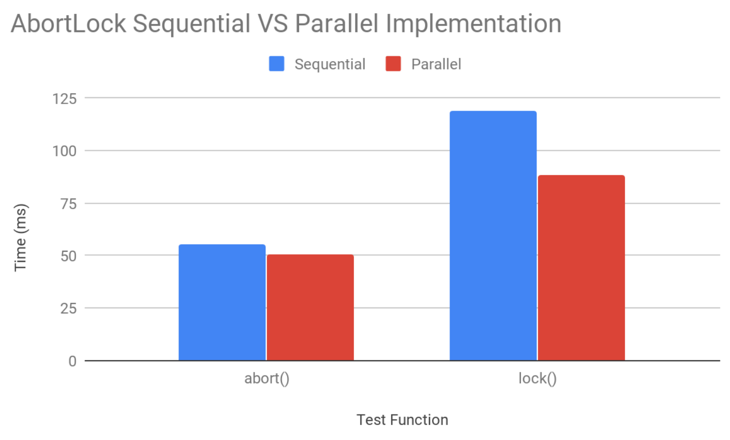


Figure 4: Graph of Overall Sequential vs Parallel AbortLock Execution

Conclusion

State restoration during lock-based execution can be solved via AbortLock. A combination of Java reflection and deep copy implementation allow programmers to halt and return thread state to safe conditions. In our testing, aborting and restoring all object data minimally affected overhead compared to the normal lock/unlock cycle. By utilizing AbortLock, programmers can produce more robust, efficient, and scalable code for handling exceptions and other events.

References

1. Google, “google/gson,” *GitHub*, 26-Oct-2018. [Online]. Available: https://github.com/google/gson. [Accessed: 28-Nov-2018].

2. “Package java.lang.reflect,” *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, 09-Oct-2018. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html. [Accessed: 28-Nov-2018].