Synchronization

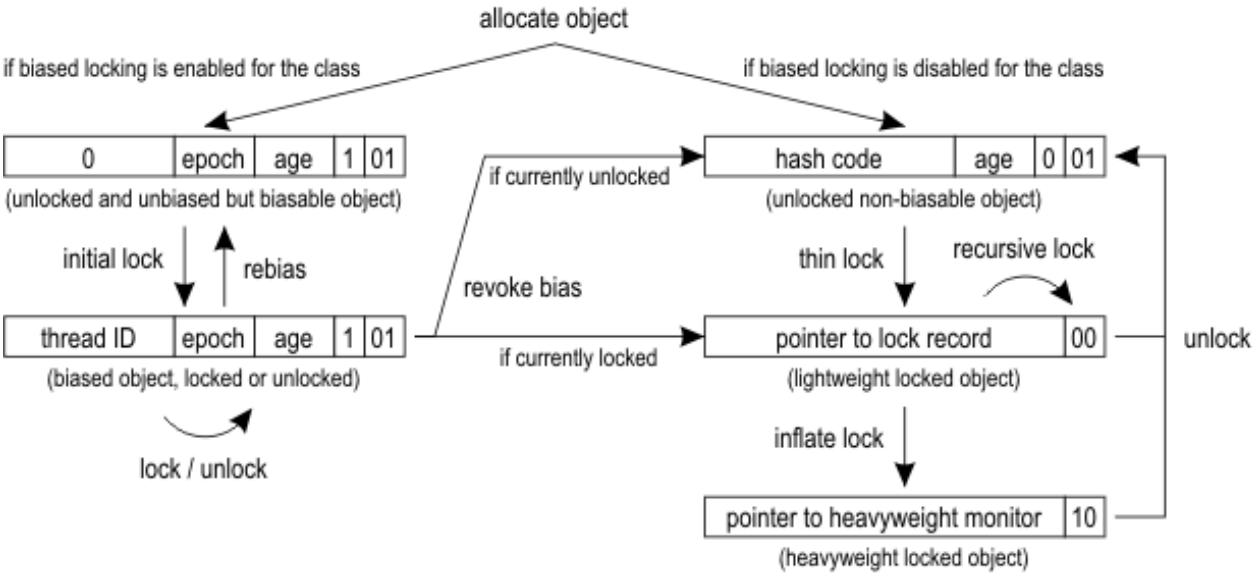
- Created by Christian Wimmer, last modified on Apr 29, 2008

Synchronization and Object Locking

Written by Thomas Kotzmann and Christian Wimmer

One of the major strengths of the Java programming language is its built-in support for multi-threaded programs. An object that is shared between multiple threads can be locked in order to synchronize its access. Java provides primitives to designate critical code regions, which act on a shared object and which may be executed only by one thread at a time. The first thread that enters the region locks the shared object. When a second thread is about to enter the same region, it must wait until the first thread has unlocked the object again.

In the Java HotSpot™ VM, every object is preceded by a class pointer and a header word. The header word, which stores the identity hash code as well as age and marking bits for generational garbage collection, is also used to implement a *thin lock scheme* [Agesen99, Bacon98]. The following figure shows the layout of the header word and the representation of different object states.



The right-hand side of the figure illustrates the standard locking process. As long as an object is unlocked, the last two bits have the value 01. When a method synchronizes on an object, the header word and a pointer to the object are stored in a lock record within the current stack frame. Then the VM attempts to install a pointer to the lock record in the object's header word via a *compare-and-swap* operation. If it succeeds, the current thread afterwards owns the lock. Since lock records are always aligned at word boundaries, the last two bits of the header word are then 00 and identify the object as being locked.

If the compare-and-swap operation fails because the object was locked before, the VM first tests whether the header word points into the method stack of the current thread. In this case, the thread already owns the object's lock and can safely continue its execution. For such a *recursively* locked object, the lock record is initialized with 0 instead of the object's header word. Only if two different threads concurrently synchronize on the same object, the thin lock must be *inflated* to a heavyweight monitor for the management of waiting threads.

Thin locks are a lot cheaper than inflated locks, but their performance suffers from the fact that every compare-and-swap operation must be executed atomically on multi-processor machines, although most objects are locked and unlocked only by one particular thread. In Java 6, this drawback is addressed by a so-called *store-free biased locking technique* [Russell06], which uses concepts similar to [Kawachiya02]. Only the first lock acquisition performs an atomic compare-and-swap to install an ID of the locking thread into the header word. The object is then said to be *biased* towards the thread. Future locking and unlocking of the object by the same thread do not require any atomic operation or an update of the header word. Even the lock record on the stack is left uninitialized as it will never be examined for a biased object.

When a thread synchronizes on an object that is biased towards another thread, the bias must be *revoked* by making the object appear as if it had been locked the regular way. The stack of the bias owner is traversed, lock records associated with the object are adjusted according to the thin lock scheme, and a pointer to the oldest of them is installed in the object's header word. All threads must be suspended for this operation. The bias is also revoked when the identity hash code of an object is accessed since the hash code bits are shared with the thread ID.

Objects that are explicitly designed to be shared between multiple threads, such as producer/consumer queues, are not suitable for biased locking. Therefore, biased locking is disabled for a class if revocations for its instances happened frequently in the past. This is called *bulk revocation*. If the locking code is invoked on an instance of a class for which biased locking was disabled, it performs the standard thin locking. Newly allocated instances of the class are marked as non-biasable.

A similar mechanism, called *bulk rebiasing*, optimizes situations in which objects of a class are locked and unlocked by different threads but never concurrently. It invalidates the bias of all instances of a class without disabling biased locking. An *epoch value* in the class acts as a timestamp that indicates the validity of the bias. This value is copied into the header word upon object allocation. Bulk rebiasing can then efficiently be implemented as an increment of the epoch in the appropriate class. The next time an instance of this class is going to be locked, the code detects a different value in the header word and rebiases the object towards the current thread.

Source Code Hints

Synchronization affects multiple parts of the JVM: The structure of the object header is defined in the classes `oopDesc` and `markOopDesc`, the code for thin locks is integrated in the interpreter and compilers, and the class `ObjectMonitor` represents inflated locks. Biased locking is centralized in the class `BiasedLocking`. It can be enabled via the flag `-XX:+UseBiasedLocking` and disabled via `-XX:-UseBiasedLocking`. It is enabled by default for Java 6 and Java 7, but activated only some seconds after the application startup. Therefore, beware of short-running micro-benchmarks. If necessary, turn off the delay using the flag `-`

`XX:BiasedLockingStartupDelay=0`.

References

[Agesen99] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, D. White: *An Efficient Meta-lock for Implementing Ubiquitous Synchronization*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 207-222. ACM Press, 1999. doi:10.1145/320384.320402

[Bacon98] D. F. Bacon, R. Konuru, C. Murthy, M. Serrano: *Thin Locks: Featherweight Synchronization for Java*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258-268. ACM Press, 1998. doi:10.1145/277650.277734

[Kawachiya02] K. Kawachiya, A. Koseki, T. Onodera: Lock Reservation: Java Locks can Mostly do without Atomic Operations. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130-141. ACM Press, 2002. doi:10.1145/582419.582433

[Russel06] K. Russell, D. Detlefs: Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 263-272. ACM Press, 2006. doi:10.1145/1167473.1167496