Synchronization Using The ObjectMonitorTable

This text is copied and adapted from PR https://github.com/openjdk/jdk/pull/20067 by Axel Boldt-Christmas.

This text describes how the runtime implements Java synchronization. A concurrent hash table has been introduced to map Java synchronization objects to the internal native runtime implementation called ObjectMonitors. Without this table, the ObjectMonitors are stored in the first word of Java objects, in most cases. This first word is called the *markWord*. A new locking mode was introduced in JDK 22 called Lightweight locking. This locking mode does not store the stack address of a lock in the first word of the Java object. With the ObjectMonitorTable, the goal is that the first word of the Java object markWord does not contain any internal information about locking other than setting 2 bits to indicate that the object is locked and where to get more information about locking. (Probably should add a section).

See also Async Monitor Deflation for related information about how ObjectMonitor memory is reclaimed.

The page Native Monitors Design refers to locks that the JVM uses internally and is unrelated to Java synchronization.

LightweightSynchronizer

Fast Locking

CAS on locking bits in markWord.
0b00 (Fast Locked) <--> 0b01 (Unlocked)

When locking and 0b00 (Fast Locked) is observed, it may be beneficial to avoid inflating by spinning a bit.

If 0b10 (Inflated) is observed or there is to much contention or to long critical sections for spinning to be feasible, inflated locking is performed.

Fast Lock Spinning (UseObjectMonitorTable)

When a thread fails fast locking when a monitor is not yet inflated, it will spin on the markWord using a exponential backoff scheme. The thread will attempt the fast lock CAS and then SpinWait() for some time, doubling with every failed attempt, up to a maximum number of attempts. There is a diagnostic VM option LightweightFastLockingSpins which can be used to tune this value. The behavior of SpinWait() can be hardware dependent.

A future improvement may be to adapt this spinning limit to observed behavior. Which would automatically adapt to the different hardware behavior of SpinWait().

Inflated Locking

Inflated locking means that a ObjectMonitor is associated with the object and is used for locking instead of the locking bits in the markWord.

Inflated Locking without table (!UseObjectMonitorTable)

An inflating thread will create a ObjectMonitor and CAS the ObjectMonitor* into the markWord along with the 0b10 (Inflated) lock bits. If the transition of the lock bits is from 0b00 (Fast Locked) the ObjectMonitor must be published with an anonymous owner (setting _owner to ANONYMOUS_OWNER). If the transition of the lock bits is from 0b01 (Unlocked) the ObjectMonitor is published with no owner.

When encountering an ObjectMonitor with an anonymous owner the thread checks its lock stack to see if it is the owner, in which case it removes the object from its lock stack and sets itself as the owner of the ObjectMonitor along with fixing the recursion level to correspond to the number of removed lock stack entires.

Inflated Locking with table (UseObjectMonitorTable)

Because publishing the ObjectMonitor* and signaling that a object's monitor is inflated is not atomic, more care must be taken (in the presence of deflation) so that all threads agree on which ObjectMonitor* to use.

When encountering an ObjectMonitor with an anonymous owner the thread checks its lock stack to see if it is the owner, in which case it removes the object from its lock stack and sets itself as the owner of the ObjectMonitor along with fixing the recursion level to correspond to the number of removed lock stack entires.

All complications arise from deflation, or the process of disassociating an ObjectMonitor from its Java Object. So first the mechanism used for deflation is explained. Followed by retrieval and creation of ObjectMonitors.

Deflation

An ObjectMonitor can only be deflated if it has no owner, its queues are empty and no thread is in a scope where it has incremented and checked the contentions reference counter.

The interactions between deflation and wait is handled by having the owner and wait queue entry overlap to blocks out deflation; the wait queue entry is protected by a waiters reference counter which is only modified by the waiters while holding the monitor, incremented before exiting the monitor and decremented after reentering the monitor.

For enter and exit where the deflator may observe empty queues and no owner a two step mechanism is used to synchronize deflation with concurrently locking threads; deflation is synchronized using the contentions reference counter.

In the text below we refer to "holding the contentions reference counter". This means that a thread has incremented the contentions reference counter and verified that it is not negative.

```
if (Atomic::fetch_and_add(&monitor->_contentions, 1) >= 0) {
  // holding the contentions reference counter
}
Atomic::decrement(&monitor->_contentions);
```

Deflation protocol

The first step for the deflator is to try and CAS the owner from no owner to a special marker (DEFLATER_MARKER). If this is successful it blocks any entering thread from successfully installing themselves as the owner and causes compiled code to take a slow path and call into the runtime.

The second step for the deflator is to check waiters reference counter and if it is 0 try CAS the contentions reference counter from 0 to a large negative value (INT_MIN). If this succeeds the monitor is deflated.

The deflator does not have to check the entry queues because every thread on the entry queues must have either hold the contentions reference counter, or incremented the waiters reference counter, in the case they were moved from the wait queue to the entry queues by a notify. The deflator check the waiters reference counter, with the memory ordering of Waiter: { increment waiters reference counter; release owner }, Deflator: { acquire owner; check waiters reference counter }. All threads on the entry queues or wait queue invariantly holds the contentions reference counter or the waiters reference counter.

Deflation cleanup

If deflation succeeds, locking bits are then transitioned back to 0b01 (Unlocked). With UseObjectMonitorTable it is required that this is done by the deflator, or it could lead to ABA problems in the locking bits. Without the table the whole ObjectMonitor* is part of the markWord transition, with its pointer being phased out of the system with a handshake, making every value distinguishable and avoiding ABA issues.

For UseObjectMonitorTable the deflated monitor is also removed from the table. This is done after transitioning the markWord to allow concurrently entering threads to fast lock on the object while the monitor is being removed from the hash table.

If deflation fails after the marker (DEFLATER_MARKER) has been CASed into the owner field the owner must be restored. From the deflation threads point of view it is as simple as CASing from the marker to no owner. However to not have all threads depend on the deflation thread making progress here we allow any thread to CAS from the marker if that thread has both incremented and checked the contentions counter. This thread has now effectively canceled the deflation, but it is important that the deflator observes this fact, we do this by forgetting to decrement the contentions counter. The effect is that the contentions CAS will fail, which will force the deflator to try and restore the owner, but this will also fail because it got canceled. So the deflator decrements the contentions counter instead on behalf of the canceling thread to balance the reference counting. (Currently this is implemented by doing a +1 +1 -1 reference count on the locking thread, but a simple only +1 would suffice).

Retrieve ObjectMonitor

HashTable

Maintains a mapping between Java Objects and ObjectMonitors. Lookups are done via the objects identity_hash. If the hash table contains an ObjectMonitor for a specific object then that ObjectMonitor is used for locking unless it is being deflated.

Only deflation removes (not dead) entries inside the HashTable.

ThreadLocal Cache (UseObjectMonitorTable)

The most recently locked ObjectMonitors by a thread are cached in that thread's local storage. These are used to elide hash table lookups. These caches uses raw oops to make cache lookups trivial. However this requires special handling of the cache at safepoints. The caches are cleared when a safepoint is triggered (instead of letting the gc visit them), this to avoid keeping cache entries as gc roots.

These cache entires may become deflated, but locking on such a monitor still participates in the normal deflation protocol. Because these entries are cleared during a safepoint, the handshake performed by monitor deflation to phase out ObjectMonitor* from the system will also phase these out.

StackLocal Cache

Each monitorenter has a corresponding BasicLock entry on the stack. Each successful inflated monitorenter saves the ObjectMonitor* inside this BasicLock entry and retrieves it when performing the corresponding monitorexit.

This means it is important that the BasicLock entry is always initialized to a known state (nullptr is used).

The RAII object class CacheSetter is used to ensure that the BasicLock gets initialized before leaving the runtime code, and that both caches gets updated correctly. (Only once, with the same locked ObjectMonitor).

The cache entries are set when a monitor is entered and never used again after a that monitored has been exited. So there are no interactions with deflation here. Similarly these caches does not track the associated oop, but rely on the fact that the same BasicLock data created for a monitorenter is used when executing the corresponding monitorexit.

Creating ObjectMonitor

If retrieval of the ObjectMonitor fails, because there is no ObjectMonitor, either because this is the first time inflating or the ObjectMonitor has been deflated a new ObjectMonitor must be created and associated with the object.

The inflating thread will then attempt to insert a newly created ObjectMonitor in the hash table. The important invariant is that any ObjectMonitor inserted must have an anonymous owner (setting _owner to ANONYMOUS_OWNER).

This solves the issue of not being able to atomically inserting the ObjectMonitor in the hash table, and transitioning the markWord to 0b10 (Inflated). We instead have all inflating threads insert an identical anonymously owned ObjectMonitor in the table and then decide ownership based on how the markWord is transitioned to 0b10 (Inflated). Note: Only one ObjectMonitor can be inserted.

This also has the effect of blocking deflation on a newly inserted ObjectMonitor, until the contentions reference counter can be incremented. The contentions reference counter is held while transitioning the markWord to block out deflation.

- If a thread observes 0b10 (Inflated)
    - If the current thread is the thread that fast locked, take ownership.
      Update ObjectMonitor _recursions based on fast locked recursions.
      Call ObjectMonitor::enter(current);
    - Otherwise Some other thread is the owner, and will claim ownership.
      Call ObjectMonitor::enter(current);
- If a thread succeeds with the CAS to 0b10 (Inflated)
    - From 0b00 (Fast Locked)
        - If the current thread is the thread that fast locked, take ownership.
          Update ObjectMonitor _recursions based on fast locked recursions.
          Call ObjectMonitor::enter(current);
        - Otherwise Some other thread is the owner, and will claim ownership.
          Call ObjectMonitor::enter(current);
    - From 0b01 (Unlocked)
        - Claim ownership, no ObjectMonitor::enter is required.
- If a thread fails the CAS reload markWord and retry

Un-contended Inflated Locking

CAS on _owner field in ObjectMonitor.
JavaThread* (Locked By Thread) <--> nullptr (Unlocked)

Contended Inflated Locking

Blocks out deflation.

Spin CAS on _owner field in ObjectMonitor.
JavaThread* (Locked By Thread) <--> nullptr (Unlocked)

Details in ObjectMonitor.hpp

HashTable Resizing and Cleanup

Resizing is currently handled with the similar logic to what the string and symbol table uses. And is delegated to the ServiceThread.

The goal is to eventually this to deflation thread, to allow for better interactions with the deflation cycles, making it possible to also shrink the table. But this will be done incrementally as a separate enhancement. The ServiceThread is currently used to deal with the fact that we currently allow the deflation thread to be turned off via JVM options.

Cleanup is mostly handled by the the deflator which actively removes deflated monitors, which includes monitors for dead objects. However we allow any thread to remove dead objects' ObjectMonitor* associations. But actual memory reclamation of the ObjectMonitor is always handled by the deflator.

The table is currently initialized before `init_globals`, as such the max size of the table which is based on `MaxHeapSize` may be incorrect because it is not yet finalize