Async Monitor Deflation

Summary

This page describes adding support for Async Monitor Deflation to OpenJDK. The primary goal of this project is to reduce the time spent in safepoint cleanup operations.

> RFE: 8153224 Monitor deflation prolong safepoints
> https://bugs.openjdk.java.net/browse/JDK-8153224
>
> Full Webrev: http://cr.openjdk.java.net/~dcubed/8153224-webrev/6-for-jdk13.full/
>
> Inc Webrev: http://cr.openjdk.java.net/~dcubed/8153224-webrev/6-for-jdk13.inc/

Background

This patch for Async Monitor Deflation is based on Carsten Varming's

> http://cr.openjdk.java.net/~cvarming/monitor_deflate_conc/0/

which has been ported to work with monitor lists. Monitor lists were optional via the '-XX:+MonitorInUseLists' option in JDK8, the option became default 'true' in JDK9, the option became deprecated in JDK10 via JDK-8180768, and the option became obsolete in JDK12 via JDK-8211384. Carsten's webrev is based on JDK10 so there was a bit of porting work needed to merge his code and/or algorithms with jdk/jdk.

Carsten also submitted a JEP back in the JDK10 time frame:

> JDK-8183909 Concurrent Monitor Deflation
>
> https://bugs.openjdk.java.net/browse/JDK-8183909

The OpenJDK JEP process has evolved a bit since JDK10 and a JEP is no longer required for a project that is well defined to be within one area of responsibility. Async Monitor Deflation is clearly defined to be in the JVM Runtime team's area of responsibility so it is likely that the JEP (JDK-8183909) will be withdrawn and the work will proceed via the RFE (JDK-8153224).

Introduction

The current idle monitor deflation mechanism executes at a safepoint during cleanup operations. Due to this execution environment, the current mechanism does not have to worry about interference from concurrently executing JavaThreads. Async Monitor Deflation uses JavaThreads and the ServiceThread to deflate idle monitors so the new mechanism has to detect interference and adapt as appropriate. In other words, data races are natural part of Async Monitor Deflation and the algorithms have to detect the races and react without data loss or corruption.

Key Parts of the Algorithm

1) Deflation With Interference Detection

ObjectSynchronizer::deflate_monitor_using_JT() is the new counterpart to ObjectSynchronizer::deflate_monitor() and does the heavy lifting of asynchronously deflating a monitor using a three part prototcol:

1. Setting a NULL owner field to DEFLATER_MARKER with cmpxchg() forces any contending thread through the slow path. A racing thread would be trying to set the owner field.
2. Making a zero ref_count field a large negative value with cmpxchg() forces racing threads to retry. A racing thread would would be trying to increment the ref_count field.
3. If the owner field is still equal to DEFLATER_MARKER, then we have won all the races and can deflate the monitor.

If we lose any of the races, the monitor cannot be deflated at this time.

Once we know it is safe to deflate the monitor (which is mostly field resetting and monitor list management), we have to restore the object's header. That's another racy operation that is described below in "Restoring the Header With Interference Detection".

2) Restoring the Header With Interference Detection

ObjectMonitor::install_displaced_markword_in_object() is the new piece of code that handles all the racy situations with restoring an object's header asynchronously. The function is called from two places (deflation and saving an ObjectMonitor* in an ObjectMonitorHandle). The restoration protocol for the object's header uses the mark bit along with the hash() value staying at zero to indicate that the object's header is being restored. Only one of the possible racing scenarios can win and the losing scenarios all adapt to the winning scenario's object header value.

3) Using "owner" or "ref_count" With Interference Detection

Various code paths have been updated to recognize an owner field equal to DEFLATER_MARKER or a negative ref_count field and those code paths will retry their operation. This is the shortest "Key Part" description, but don't be fooled. See "Gory Details" below.

An Example of ObjectMonitor Interference Detection

ObjectMonitor::save_om_ptr() is used to safely save an ObjectMonitor* in an ObjectMonitorHandle. ObjectSynchronizer::deflate_monitor_using_JT() is used to asynchronously deflate an idle monitor. save_om_ptr() and deflate_monitor_using_JT() can interfere with each other. The thread calling save_om_ptr() (T-save) is potentially racing with another JavaThread (T-deflate) so both threads have to check the results of the races.

Start of the Race

```
 T-save                   ObjectMonitor              T-deflate
 ----------------------   +-----------------------+  ----------------------------------------
 save_om_ptr() {          | owner=NULL            |  deflate_monitor_using_JT() {
1> atomic inc ref_count   | ref_count=0           | 1> cmpxchg(DEFLATER_MARKER, &owner, NULL)
                          +-----------------------+
```

- The data fields are at their starting values.
- The "1>" markers are showing where each thread is at for the ObjectMonitor box:
  - T-deflate is about to execute cmpxchg().
  - T-save is about to increment the ref_count.

Racing Threads

```
 T-save                   ObjectMonitor              T-deflate
 ----------------------   +-----------------------+  ------------------------------------------
 save_om_ptr() {          | owner=DEFLATER_MARKER |  deflate_monitor_using_JT() {
1> atomic inc ref_count   | ref_count=0           |    cmpxchg(DEFLATER_MARKER, &owner, NULL)
                          +-----------------------+    :
                                                    1> prev = cmpxchg(-max_jint, &ref_count, 0)
```

- T-deflate has executed cmpxchg() and set owner to DEFLATE_MARKER.
- T-save still hasn't done anything yet
- The "1>" markers are showing where each thread is at for the ObjectMonitor box:
  - T-save and T-deflate are racing to update the ref_count field.

T-deflate Wins

```
 T-save                              ObjectMonitor              T-deflate
 ---------------------------------   +-----------------------+  ------------------------------------------
 save_om_ptr() {                     | owner=DEFLATER_MARKER |  deflate_monitor_using_JT() {
   atomic inc ref_count              | ref_count=-max_jint+1 |    cmpxchg(DEFLATER_MARKER, &owner, NULL)
1> if (owner == DEFLATER_MARKER &&   +-----------------------+    :
       ref_count <= 0) {                        ||                prev = cmpxchg(-max_jint, &ref_count, 0)
     restore obj header                         \/             1> if (prev == 0 &&
     atomic dec ref_count            +-----------------------+        owner == DEFLATER_MARKER) {
  2> return false to force retry     | owner=DEFLATER_MARKER |      restore obj header
   }                                 | ref_count=-max_jint   |   2> finish the deflation
                                     +-----------------------+    }
```

- This diagram starts after "Racing Threads".
- The "1>" markers are showing where each thread is at for that ObjectMonitor box:
  - T-save and T-deflate both observe owner == DEFLATER_MARKER and a negative ref_count field.
- T-save has lost the race: it restores the obj header (not shown) and decrements the ref_count.
- T-deflate restores the obj header (not shown).
- The "2>" markers are showing where each thread is at for that ObjectMonitor box.
- T-save returns false to cause the caller to retry.
- T-deflate finishes the deflation.

T-save Wins

```
 T-save                              ObjectMonitor              T-deflate
 ---------------------------------   +-----------------------+  ------------------------------------------
 save_om_ptr() {                     | owner=DEFLATER_MARKER |  deflate_monitor_using_JT() {
   atomic inc ref_count              | ref_count=1           |    cmpxchg(DEFLATER_MARKER, &owner, NULL)
1> if (owner == DEFLATER_MARKER &&   +-----------------------+    :
       ref_count <= 0) {                        ||                prev = cmpxchg(-max_jint, &ref_count, 0)
 } else {                                       \/              1> if (prev == 0 &&
   save om_ptr in the                +-----------------------+          owner == DEFLATER_MARKER) {
     ObjectMonitorHandle             | owner=NULL            |    } else {
2> return true                       | ref_count=1           |      cmpxchg(NULL, &owner, DEFLATER_MARKER)
                                     +-----------------------+   2> return
```

- This diagram starts after "Racing Threads".
- The "1>" markers are showing where each thread is at for the ObjectMonitor box:
  - T-save and T-deflate both observe a ref_count field > 0.
- T-save has won the race and it saves the ObjectMonitor* in the ObjectMonitorHandle (not shown).
- T-deflate detects that it has lost the race (prev != 0) and bails out on deflating the ObjectMonitor:
  - Before bailing out T-deflate tries to restore the owner field to NULL if it is still DEFLATER_MARKER.
- The "2>" markers are showing where each thread is at for that ObjectMonitor box.

T-enter Wins By A-B-A

```
    T-enter                                       ObjectMonitor            T-deflate
    -------------------------------------------   +-------------------------
+  -----------------------------------------
    ObjectMonitor::enter() {                      | owner=DEFLATER_MARKER   |  deflate_monitor_using_JT() {
      <owner is contended>                        | ref_count=1             |    cmpxchg(DEFLATER_MARKER, &owner,
NULL)
   1> EnterI() {                                  +-------------------------+ 1> :
        if (owner == DEFLATER_MARKER &&                       ||              2> : <thread_stalls>
            cmpxchg(Self, &owner,                             \/                 :
                    DEFLATER_MARKER)              +-------------------------+    :
                    == DEFLATER_MARKER) {         | owner=Self/T-enter      |    :
          // EnterI is done                       | ref_count=0             |    : <thread_resumes>
          return                                  +-------------------------+    prev = cmpxchg(-max_jint,
&ref_count, 0)
        }                                                     ||                 if (prev == 0 &&
    } // enter() is done                                      \/                3>   owner == DEFLATER_MARKER) {
    ~OMH: atomic dec ref_count                    +-------------------------+    } else {
 2> : <does app work>                             | owner=Self/T-enter|NULL |      cmpxchg(NULL, &owner,
DEFLATER_MARKER)
 3> :                                             | ref_count=-max_jint     |      atomic add max_jint to
ref_count
    exit() monitor                                +-------------------------+   4> bailout on deflation
   4> owner = NULL                                            ||                 }
                                                              \/
                                                  +-------------------------+
                                                  | owner=Self/T-enter|NULL |
                                                  | ref_count=0             |
                                                  +-------------------------+
```

- T-deflate has executed cmpxchg() and set owner to DEFLATE_MARKER.
- T-enter has called ObjectMonitor::enter() with "ref_count == 1", noticed that the owner is contended and is about to call ObjectMonitor::EnterI().
- The first ObjectMonitor box is showing the fields at this point and the "1>" markers are showing where each thread is at for that ObjectMonitor box.
- T-deflate stalls after setting the owner field to DEFLATER_MARKER.
- T-enter calls EnterI() to do the contended enter work:
  - EnterI() observes owner == DEFLATER_MARKER and uses cmpxchg() to set the owner field to Self/T-enter.
  - T-enter owns the monitor, returns from EnterI(), and returns from enter().
  - The ObjectMonitorHandle destructor decrements the ref_count.
- T-enter is now ready to do work that requires the monitor to be owned.
- The second ObjectMonitor box is showing the fields at this point and the "2>" markers are showing where each thread is at for that ObjectMonitor box.
- T-enter is doing app work (but it also could have finished and exited the monitor).
- T-deflate resumes, calls cmpxchg() to set the ref_count field to -max_jint, and passes the first part of the bailout expression because "prev == 0".
- The third ObjectMonitor box is showing the fields at this point and the "3>" markers are showing where each thread is at for that ObjectMonitor box.
- T-deflate performs the A-B-A check which observes that "owner != DEFLATE_MARKER" and bails out on deflation:
  - Depending on when T-deflate resumes after the stall, it will see "owner == T-enter" or "owner == NULL".
  - Both of those values will cause deflation to bailout so we have to conditionally undo work:
    - restore the owner field to NULL if it is still DEFLATER_MARKER (it's not DEFLATER_MARKER)
    - undo setting ref_count to -max_jint by atomically adding max_jint to ref_count which will restore ref_count to its proper value.
  - If the T-enter thread has managed to enter but not exit the monitor during the T-deflate stall, then our owner field A-B-A transition is:
    - NULL DEFLATE_MARKER Self/T-enter
  - so we really have A1-B-A2, but the A-B-A principal still holds.
  - If the T-enter thread has managed to enter and exit the monitor during the T-deflate stall, then our owner field A-B-A transition is:
    - NULL DEFLATE_MARKER Self/T-enter   NULL
  - so we really have A-B1-B2-A, but the A-B-A principal still holds.
- T-enter finished doing app work and is about to exit the monitor (or it has already exited the monitor).
- The fourth ObjectMonitor box is showing the fields at this point and the "4>" markers are showing where each thread is at for that ObjectMonitor box.

An Example of Object Header Interference

After T-deflate has won the race for deflating an ObjectMonitor it has to restore the header in the associated object. Of course another thread can be trying to do something to the object's header at the same time. Isn't asynchronous work exciting?!?!

ObjectMonitor::install_displaced_markword_in_object() is called from two places so we can have a race between a T-save thread and a T-deflate thread:

Start of the Race

```
T-save                                       object           T-deflate
-------------------------------------------  +-------------+  -------------------------------------------
install_displaced_markword_in_object() {     | mark=om_ptr |  install_displaced_markword_in_object() {
  dmw = header()                             +-------------+    dmw = header()
  if (!dmw->is_marked() &&                                      if (!dmw->is_marked() &&
      dmw->hash() == 0) {                                           dmw->hash() == 0) {
    create marked_dmw                                                create marked_dmw
    dmw = cmpxchg(marked_dmw, &header, dmw)                          dmw = cmpxchg(marked_dmw, &header, dmw)
  }                                                                }
```

- The data field (mark) is at its starting value.
- 'dmw' and 'marked_dmw' are local copies in each thread.
- T-save and T-deflate are both calling install_displaced_markword_in_object() at the same time.
- Both threads are poised to call cmpxchg() at the same time.

T-deflate Wins First Race

```
T-save                                       object            T-deflate
-------------------------------------------  +-------------+   -------------------------------------------
install_displaced_markword_in_object() {     | mark=om_ptr |   install_displaced_markword_in_object() {
  dmw = header()                             +-------------+     dmw = header()
  if (!dmw->is_marked() &&                                       if (!dmw->is_marked() &&
      dmw->hash() == 0) {                                            dmw->hash() == 0) {
    create marked_dmw                                                create marked_dmw
    dmw = cmpxchg(marked_dmw, &header, dmw)                          dmw = cmpxchg(marked_dmw, &header, dmw)
  }                                                              }
  // dmw == marked_dmw here                                      // dmw == original dmw here
  if (dmw->is_marked())                                          if (dmw->is_marked())
    unmark dmw                                                     unmark dmw
  obj = object()                                                 obj = object()
  obj->cas_set_mark(dmw, this)                                   obj->cas_set_mark(dmw, this)
```

- The return value from cmpxchg() in each thread will be different.
- Since T-deflate won the race, its 'dmw' variable contains the header/dmw from the ObjectMonitor.
- Since T-save lost the race, its 'dmw' variable contains the 'marked_dmw' set by T-deflate.
  - T-save will unmark its 'dmw' variable.
- Both threads are poised to call cas_set_mark() at the same time.

T-save Wins First Race

```
T-save                                       object            T-deflate
-------------------------------------------  +-------------+   -------------------------------------------
install_displaced_markword_in_object() {     | mark=om_ptr |   install_displaced_markword_in_object() {
  dmw = header()                             +-------------+     dmw = header()
  if (!dmw->is_marked() &&                                       if (!dmw->is_marked() &&
      dmw->hash() == 0) {                                            dmw->hash() == 0) {
    create marked_dmw                                                create marked_dmw
    dmw = cmpxchg(marked_dmw, &header, dmw)                          dmw = cmpxchg(marked_dmw, &header, dmw)
  }                                                              }
  // dmw == original dmw here                                    // dmw == marked_dmw here
  if (dmw->is_marked())                                          if (dmw->is_marked())
    unmark dmw                                                     unmark dmw
  obj = object()                                                 obj = object()
  obj->cas_set_mark(dmw, this)                                   obj->cas_set_mark(dmw, this)
```

- This diagram is the same as "T-deflate Wins First Race" except we've swapped the post cmpxchg() comments.
- Since T-save won the race, its 'dmw' variable contains the header/dmw from the ObjectMonitor.
- Since T-deflate lost the race, its 'dmw' variable contains the 'marked_dmw' set by T-save.
  - T-deflate will unmark its 'dmw' variable.
- Both threads are poised to call cas_set_mark() at the same time.

Either Wins the Second Race

```
T-save                                       object            T-deflate
-------------------------------------------  +-------------+   -------------------------------------------
install_displaced_markword_in_object() {     | mark=dmw    |   install_displaced_markword_in_object() {
  dmw = header()                             +-------------+     dmw = header()
  if (!dmw->is_marked() &&                                       if (!dmw->is_marked() &&
      dmw->hash() == 0) {                                            dmw->hash() == 0) {
    create marked_dmw                                                create marked_dmw
    dmw = cmpxchg(marked_dmw, &header, dmw)                          dmw = cmpxchg(marked_dmw, &header, dmw)
  }                                                              }
  // dmw == ...                                                  // dmw == ...
  if (dmw->is_marked())                                          if (dmw->is_marked())
    unmark dmw                                                     unmark dmw
  obj = object()                                                 obj = object()
  obj->cas_set_mark(dmw, this)                                   obj->cas_set_mark(dmw, this)
```

- It does not matter whether T-save or T-deflate won the cmpxchg() call so the comment does not say who won.

- It does not matter whether T-save or T-deflate won the cas_set_mark() call; in this scenario both were trying to restore the same value.
- The object's mark field has changed from 'om_ptr' 'dmw'.

Please notice that install_displaced_markword_in_object() does not do any retries on any code path:

- Instead the code adapts to being the loser in a cmpxchg() by unmarking its copy of the dmw.
- In the second race, if a thread loses the cas_set_mark() race, there is also no need to retry because the object's header has been restored by the other thread.

Hashcodes and Object Header Interference

If we have a race between a T-deflate thread and a thread trying to get/set a hashcode (T-hash), then the race is between the ObjectMonitorHandle. save_om_ptr(obj, mark) call in T-hash and deflation protocol in T-deflate.

Start of the Race

```
 T-hash                  ObjectMonitor              T-deflate
 ----------------------  +-----------------------+  ----------------------------------------
 save_om_ptr() {         | owner=NULL            |  deflate_monitor_using_JT() {
   :                     | ref_count=0           | 1> cmpxchg(DEFLATER_MARKER, &owner, NULL)
1> atomic inc ref_count  +-----------------------+
```

- The data fields are at their starting values.
- T-deflate is about to execute cmpxchg().
- T-hash is about to increment ref_count.
- The "1>" markers are showing where each thread is at for the ObjectMonitor box.

Racing Threads

```
 T-hash                  ObjectMonitor              T-deflate
 ----------------------  +-----------------------+  ------------------------------------------
 save_om_ptr() {         | owner=DEFLATER_MARKER |  deflate_monitor_using_JT() {
   :                     | ref_count=0           |    cmpxchg(DEFLATER_MARKER, &owner, NULL)
1> atomic inc ref_count  +-----------------------+    if (contentions != 0 || waiters != 0) {
                                                      }
                                                 1> prev = cmpxchg(-max_jint, &ref_count, 0)
```

- T-deflate has set the owner field to DEFLATER_MARKER.
- The "1>" markers are showing where each thread is at for the ObjectMonitor box:
  - T-deflate is about to execute cmpxchg().
  - T-save is about to increment the ref_count.

T-deflate Wins

If T-deflate wins the race, then T-hash will have to retry at most once.

```
 T-hash                      ObjectMonitor              T-deflate
 -------------------------   +-----------------------+  ------------------------------------------
 save_om_ptr() {             | owner=DEFLATER_MARKER |  deflate_monitor_using_JT() {
1> atomic inc ref_count      | ref_count=-max_jint   |    cmpxchg(DEFLATER_MARKER, &owner, NULL)
   if (owner ==              +-----------------------+    if (contentions != 0 || waiters != 0) {
       DEFLATER_MARKER &&               ||                }
       ref_count <= 0) {                \/                prev = cmpxchg(-max_jint, &ref_count, 0)
     restore obj header      +-----------------------+ 1> if (prev == 0 &&
     atomic dec ref_count    | owner=DEFLATER_MARKER |        owner == DEFLATER_MARKER) {
  2> return false to         | ref_count=-max_jint   |      restore obj header
       cause a retry         +-----------------------+   2> finish the deflation
   }                                                      }
```

- T-deflate made it past the cmpxchg() of ref_count before T-hash incremented it.
- T-deflate set the ref_count field to -max_jint and is about to make the last of the protocol checks.
- The first ObjectMonitor box is showing the fields at this point and the "1>" markers are showing where each thread is at for that ObjectMonitor box.
- T-deflate sees "prev == 0 && owner == DEFLATER_MARKER" so it knows that it has won the race.
- T-deflate restores obj header (not shown).
- T-hash increments the ref_count.
- T-hash observes "owner == DEFLATER_MARKER && ref_count <= 0" so it restores obj header (not shown) and decrements ref_count.
- The second ObjectMonitor box is showing the fields at this point and the "2>" markers are showing where each thread is at for that ObjectMonitor box.
- T-deflate finishes the deflation work.
- T-hash returns false to cause a retry and when T-hash retries:
  - it observes the restored object header (done by T-hash or T-deflate):
    - if the object's header does not have a hash, then generate a hash and merge it with the object's header.
    - Otherwise, extract the hash from the object's header and return it.

T-hash Wins

If T-hash wins the race, then the ref_count will cause T-deflate to bail out on deflating the monitor.

Note: header is not mentioned in any of the previous sections for simplicity.

```
    T-hash                      ObjectMonitor             T-deflate
    -------------------------   +-----------------------+  -----------------------------------------
    save_om_ptr() {             | header=dmw_no_hash    |  deflate_monitor_using_JT() {
      atomic inc ref_count      | owner=DEFLATER_MARKER |    cmpxchg(DEFLATER_MARKER, &owner, NULL)
   1> if (owner ==              | ref_count=1           |    if (contentions != 0 || waiters != 0) {
          DEFLATER_MARKER &&    +-----------------------+    }
          ref_count <= 0) {                ||             1> prev = cmpxchg(-max_jint, &ref_count, 0)
      } else {                             \/                if (prev == 0 &&
     2> save om_ptr in the      +-----------------------+        owner == DEFLATER_MARKER) {
          ObjectMonitorHandle   | header=dmw_no_hash    |    } else {
        return true             | owner=NULL            |      cmpxchg(NULL, &owner, DEFLATER_MARKER)
      }                         | ref_count=1           |   2> bailout on deflation
    }                           +-----------------------+    }
    if save_om_ptr() {                     ||
      if no hash                           \/
        gen hash & merge        +-----------------------+
      hash = hash(header)       | header=dmw_hash       |
    }                           | owner=NULL            |
 3> atomic dec ref_count        | ref_count=1           |
    return hash                 +-----------------------+
```

- T-deflate has set the owner field to DEFLATER_MARKER.
- T-hash has incremented ref_count before T-deflate made it to cmpxchg().
- The first ObjectMonitor box is showing the fields at this point and the "1>" markers are showing where each thread is at for that ObjectMonitor box.
- T-deflate bails out on deflation, but first it tries to restore the owner field:
    - The return value of cmpxchg() is not checked here.
    - If T-deflate cannot restore the owner field to NULL, then another thread has managed to enter the monitor (or enter and exit the monitor) and we don't want to overwrite that information.
- T-hash observes:
    - "owner == DEFLATER_MARKER && ref_count > 0" or
    - "owner == NULL && ref_count > 0" so it gets ready to save the ObjectMonitor*.
- The second ObjectMonitor box is showing the fields at this point and the "2>" markers are showing where each thread is at for that ObjectMonitor box.
- T-hash saves the ObjectMonitor* in the ObjectMonitorHandle (not shown) and returns to the caller.
- save_om_ptr() returns true since the ObjectMonitor is safe:
    - if ObjectMonitor's 'header/dmw' field does not have a hash, then generate a hash and merge it with the 'header/dmw' field.
    - Otherwise, extract the hash from the ObjectMonitor's 'header/dmw' field.
- The third ObjectMonitor box is showing the fields at this point and the "3>" marker is showing where T-hash is at for that ObjectMonitor box.
- T-hash decrements the ref_count field.
- T-hash returns the hash value.

Please note that in Carsten's original prototype, there was another race in ObjectSynchronizer::FastHashCode() when the object's monitor had to be inflated. The setting of the hashcode in the ObjectMonitor's header/dmw could race with T-deflate. That race is resolved in this version by the use of an ObjectMonitorHandle in the call to ObjectSynchronizer::inflate(). The ObjectMonitor* returned by ObjectMonitorHandle.om_ptr() has a non-zero ref_count so no additional races with T-deflate are possible.

Housekeeping Parts of the Algorithm

The devil is in the details! Housekeeping or administrative stuff are usually detailed, but necessary.

- New diagnostic option '-XX:AsyncDeflateIdleMonitors' that is default 'true' so that the new mechanism is used by default, but it can be disabled for potential failure diagnosis.
- ObjectMonitor deflation is still initiated or signaled as needed at a safepoint. When Async Monitor Deflation is in use, flags are set so that the work is done by JavaThreads and the ServiceThread which offloads the safepoint cleanup mechanism.
- ObjectSynchronizer::omAlloc() is modified to call (as needed) ObjectSynchronizer::deflate_per_thread_idle_monitors_using_JT(). Having the JavaThread cleanup its own per-thread monitor list permits this work to happen without any per-thread list locking or critical sections.
    - Having a JavaThread deflate a potentially long list of in-use monitors could potentially delay the start of a safepoint. This is detected in ObjectSynchronizer::deflate_monitor_list_using_JT() which will save the current state when it is safe to do so and return to its caller to drop locks as needed before honoring the safepoint request.

- ObjectSynchronizer::inflate() has to be careful how omAlloc() is called. If the inflation cause is inflate_cause_vm_internal, then it is not safe to deflate monitors on the per-thread lists so we skip that. When monitor deflation is done, inflate() has to do the oop refresh dance that is common to any code that can go to a safepoint while holding a naked oop. And, no you can't use a Handle here either. :-)
- Everything else is just monitor list management, infrastructure, logging, debugging and the like. :-)

Gory Details

- Counterpart function mapping for those that know the existing code:
    - ObjectSynchronizer class:
        - deflate_idle_monitors() has deflate_global_idle_monitors_using_JT(), deflate_per_thread_idle_monitors_using_JT(), and deflate _common_idle_monitors_using_JT().

        - deflate_monitor_list() has deflate_monitor_list_using_JT()
        - deflate_monitor() has deflate_monitor_using_JT()
    - ObjectMonitor class:
        - is_busy() has is_busy_async()
        - clear() has clear_using_JT()
- These functions recognize the Async Monitor Deflation protocol and adapt their operations:
    - ObjectMonitor::EnterI()
    - ObjectMonitor::ReenterI()
- Also these functions had to adapt and retry their operations:
    - ObjectSynchronizer::quick_enter()
    - ObjectSynchronizer::FastHashCode()
    - ObjectSynchronizer::current_thread_holds_lock()
    - ObjectSynchronizer::query_lock_ownership()
    - ObjectSynchronizer::get_lock_owner()
    - ObjectSynchronizer::monitors_iterate()
    - ObjectSynchronizer::inflate_helper()
    - ObjectSynchronizer::inflate()
- Various assertions had to be modified to pass without their real check when AsyncDeflateIdleMonitors is true; this is due to the change in semantics for the ObjectMonitor owner field.
- ObjectMonitor has a new allocation_state field that supports three states: 'Free', 'New', 'Old'. Async Monitor Deflation is only applied to ObjectMonitors that have reached the 'Old' state.
    - Note: Prior to CR1/v2.01/4-for-jdk13, the allocation state was transitioned from 'New' to 'Old' in deflate_monitor_via_JT(). This meant that deflate_monitor_via_JT() had to see an ObjectMonitor twice before deflating it. This policy was intended to prevent oscillation from 'New' 'Old' and back again.
    - In CR1/v2.01/4-for-jdk13, the allocation state is transitioned from 'New' -> "Old" in inflate(). This makes ObjectMonitors available for deflation earlier. So far there has been no signs of oscillation from 'New' 'Old' and back again.
- ObjectMonitor has a new ref_count field that is used as part of the async deflation protocol and to indicate that an ObjectMonitor* is in use so the ObjectMonitor should not be deflated; this is needed for operations on non-busy monitors so that ObjectMonitor values don't change while they are being queried. There is a new ObjectMonitorHandle helper to manage the ref_count.
- The ObjectMonitor::owner() accessor detects DEFLATER_MARKER and returns NULL in that case to minimize the places that need to understand the new DEFLATER_MARKER value.
- System.gc()/JVM_GC() causes a special monitor list cleanup request which uses the safepoint based monitor list mechanism. So even if AsyncDeflateIdleMonitors is enabled, the safepoint based mechanism is still used by this special case.
    - This is necessary for those tests that do something to cause an object's monitor to be inflated, clear the only reference to the object and then expect that enough System.gc() calls will eventually cause the object to be GC'ed even when the thread never inflates another object's monitor. Yes, we have several tests like that. :-)