Async Monitor Deflation

Table of Contents:

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: 17-for-jdk15+24.v2.15.full
>
> Inc Webrev: 17-for-jdk15+24.v2.15.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defined to be in the JVM Runtime team's area of responsibility so it is likely that the JEP (JDK-8183909) will be withdrawn and the work will proceed via the RFE (JDK-8153224).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 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.

Async Monitor Deflation is performed in two stages: stage one performs the two part protocol described in "Deflation With Interference Detection" below and moves the async deflated ObjectMonitors from an in-use list to a global wait list; the ServiceThread performs a handshake (or a safepoint) with all other JavaThreads after stage one is complete and that forces any racing threads to make forward progress; stage two moves the ObjectMonitors from the global wait list to the global free list. The special values that mark an ObjectMonitor as async deflated remain in their fields until the ObjectMonitor is moved from the global free list to a per-thread free list which is sometime after stage two has completed.

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 two 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 contentions field a large negative value with cmpxchg() forces racing threads to retry. A racing thread would would be trying to increment the contentions field.

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".

The setting of the special values that mark an ObjectMonitor as async deflated and the restoration of the object's header comprise the first stage of Async Monitor Deflation.

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 three places (deflation, ObjectMonitor::enter(), and FastHashCode). 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 "contentions" With Interference Detection

Various code paths have been updated to recognize an owner field equal to DEFLATER_MARKER or a negative contentions 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::enter() can change an idle monitor into a busy monitor. ObjectSynchronizer::deflate_monitor_using_JT() is used to asynchronously deflate an idle monitor. enter() and deflate_monitor_using_JT() can interfere with each other. The thread calling enter() (T-enter) is potentially racing with another JavaThread (T-deflate) so both threads have to check the results of the races.

Start of the Race

```
 T-enter                   ObjectMonitor              T-deflate
 ------------------------  +----------------------+   ---------------------------------------
 enter() {                 | owner=NULL           |   deflate_monitor_using_JT() {
1> atomic inc contentions  | contentions=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-enter is about to increment contentions.

Racing Threads

```
 T-enter                   ObjectMonitor              T-deflate
 ------------------------  +----------------------+  ------------------------------------------
 enter() {                 | owner=DEFLATER_MARKER |  deflate_monitor_using_JT() {
1> add_to_contentions(1)   | contentions=0         |    try_set_owner_from(NULL, DEFLATER_MARKER)
                           +----------------------+     :
                                                      1> prev = cmpxchg(&contentions, 0, -max_jint)
```

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

T-deflate Wins

```
 T-enter                           ObjectMonitor               T-deflate
 --------------------------------  +-------------------------+  ------------------------------------------
 enter() {                         | owner=DEFLATER_MARKER   |  deflate_monitor_using_JT() {
   add_to_contentions(1)           | contentions=-max_jint+1 |    try_set_owner_from(NULL, DEFLATER_MARKER)
1> if (is_being_async_deflated()) {+-------------------------+    :
     restore obj header                        ||                 prev = cmpxchg(&contentions, 0, -max_jint)
     add_to_contentions(-1)                    \/               1> if (prev == 0) {
  2> return false to force retry   +-------------------------+      restore obj header
   }                               | owner=DEFLATER_MARKER   |    2> finish the deflation
                                   | contentions=-max_jint   |    }
                                   +-------------------------+
```

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

T-enter Wins

```
 T-enter                           ObjectMonitor               T-deflate
 --------------------------------  +-------------------------+  ------------------------------------------
 enter() {                         | owner=DEFLATER_MARKER.  |  deflate_monitor_using_JT() {
   add_to_contentions(1)           | contentions=1           |    try_set_owner_from(NULL, DEFLATER_MARKER)
1> if (is_being_async_deflated()) {+-------------------------+    :
   }                                         ||                 prev = cmpxchg(&contentions, 0, -max_jint)
2> <continue contended enter>                \/               1> if (prev == 0) {
                                 +-------------------------+    } else {
                                 | owner=NULL              |      try_set_owner_from(DEFLATER_MARKER, NULL)
                                 | contentions=1           |    2> return
                                 +-------------------------+
```

- This diagram starts after "Racing Threads".
- The "1>" markers are showing where each thread is at for the ObjectMonitor box:
  - T-enter and T-deflate both observe a contentions field > 0.
- T-enter has won the race and it continues with the contended enter protocol.
- 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.
- Note: The owner == DEFLATER_MARKER and contentions < 0 values that are set by T-deflate (stage one of async deflation) remain in place until after T-deflate does a handshake (or safepoint) operation with all JavaThreads. This handshake forces T-enter to make forward progress and see that the ObjectMonitor is being async deflated before T-enter checks in for the handshake.

T-enter Wins By Cancellation Via DEFLATER_MARKER Swap

```
    T-enter                                          ObjectMonitor                T-deflate
    ---------------------------------------------  +-------------------------
+  ---------------------------------------------
    ObjectMonitor::enter() {                       | owner=DEFLATER_MARKER   |  deflate_monitor_using_JT() {
      add_to_contentions(1)                        | contentions=1           |    try_set_owner_from(NULL,
DEFLATER_MARKER)
   1> EnterI() {                                   +-------------------------+ 1> :
        if (try_set_owner_from(DEFLATER_MARKER,                ||               2> : <thread_stalls>
            Self) == DEFLATER_MARKER) {                        \/                  :
          // Add marker for cancellation                                       +-------------------------+    :
          add_to_contentions(1)                    | owner=Self/T-enter      |    :
          // EnterI is done                        | contentions=2           |    : <thread_resumes>
          return                                   +-------------------------+    prev = cmpxchg(&contentions, 0, -
max_jint)
        }                                                  ||                   if (prev == 0) {
 2>     add_to_contentions(-1)                             \/                3> } else {
    } // enter() is done                           +-------------------------+      if (try_set_owner_from
(DEFLATER_MARKER,
    : <does app work>                              | owner=Self/T-enter|NULL |           NULL) != DEFLATER_MARKER)
{
 3> :                                              | contentions=1           |        add_to_contentions(-1)
    exit() monitor                                 +-------------------------+      }
   4> owner = NULL                                             ||                  4> bailout on deflation
                                                               \/                   }
                                                   +-------------------------+
                                                   | owner=Self/T-enter|NULL |
                                                   | contentions=0           |
                                                   +-------------------------+
```

- T-deflate has set owner to DEFLATER_MARKER.
- T-enter has called ObjectMonitor::enter(), noticed that the owner is contended, increments contentions, 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() sets the owner field from DEFLATER_MARKER to Self/T-enter.
  - EnterI() increments contentions one extra time since it cancelled async deflation via a DEFLATER_MARKER swap.
  - Note: The extra increment also makes the return value from is_being_async_deflated() stable; the previous A-B-A algorithm would allow the contentions field to flicker from 0 -max_jint and back to zero. With the current algorithm, a negative contentions field value is a linearization point so once it is negative, we are committed to performing async deflation.
  - T-enter owns the monitor and returns from EnterI() (contentions still has both increments).
- 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 decrements contentions and returns from enter() (contentions still has the extra increment).
- T-enter is now ready to do work that requires the monitor to be owned.
- T-enter is doing app work (but it also could have finished and exited the monitor and it still has the extra increment).
- T-deflate resumes, tries to set the contentions field to -max_jint and fails because contentions == 1 (the extra increment comes into play!).
- 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 tries to restore the owner field from DEFLATER_MARKER to NULL:
  - If it does not succeed, then the EnterI() call managed to cancel async deflation via a DEFLATER_MARKER swap so T-deflate decrements contentions to get rid of the extra increment that EnterI() did as a marker for this type of cancellation.
  - If it does succeed, then EnterI() did not cancel async deflation via a DEFLATER_MARKER swap and we don't have an extra increment to get rid of.
  - Note: For the previous bullet, async deflation is still cancelled because the ObjectMonitor is now busy with a contended enter.
- 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-enter thread and a T-deflate thread:

Start of the Race

```
    T-enter                                          object           T-deflate
    ---------------------------------------------  +-------------
+  ---------------------------------------------
    install_displaced_markword_in_object(oop obj) {  | mark=om_ptr |  install_displaced_markword_in_object(oop
obj) {
      dmw = header()                                 +-------------+    dmw = header()
      obj->cas_set_mark(dmw, this)                                      obj->cas_set_mark(dmw,
this)                                                             }
```

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

Either Thread Wins the Race

```
    T-enter                                          object            T-deflate
    -----------------------------------------------  +-------------
+   -----------------------------------------------
    install_displaced_markword_in_object(oop obj) {  | mark=dmw    |   install_displaced_markword_in_object(oop
obj) {
      dmw = header()                                 +-------------+     dmw = header()
      obj->cas_set_mark(dmw, this)                                       obj->cas_set_mark(dmw, this)
```

- It does not matter whether T-enter 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:

- If a thread loses the cas_set_mark() race, there is no need to retry because the object's header has been restored by the other thread.

Hashcodes and Object Header Interference

There are a few races that can occur between a T-deflate thread and a thread trying to get/set a hashcode (T-hash) in an ObjectMonitor:

1. If the object has an ObjectMonitor (i.e., is inflated) and if the ObjectMonitor has a hashcode, then the hashcode value can be carefully fetched from the ObjectMonitor and returned to the caller (T-hash). If there is a race with async deflation, then we have to retry.
2. There are several reasons why we might have to inflate the ObjectMonitor in order to set the hashcode:
    a. The object is neutral, does not contain a hashcode and we (T-hash) lost the race to try an install a hashcode in the mark word.
    b. The object is stack locked and does not contain a hashcode in the mark word.
    c. The object has an ObjectMonitor and the ObjectMonitor does not have a hashcode.
    Note: In this case, the inflate() call on the common fall thru code path is almost always a no-op since the existing ObjectMonitor is not likely to be async deflated before inflate() sees that the object already has an ObjectMonitor and bails out.

The common fall thru code path (executed by T-hash) that inflates the ObjectMonitor in order to set the hashcode can race with an async deflation (T-deflate). After the hashcode has been stored in the ObjectMonitor, we (T-hash) check if the ObjectMonitor has been async deflated (by T-deflate). If it has, then we (T-hash) retry because we don't know if the hashcode was stored in the ObjectMonitor before the object's header was restored (by T-deflate). Retrying (by T-hash) will result in the hashcode being stored in either object's header or in the re-inflated ObjectMonitor's header as appropriate.

Spin-Lock Monitor List Management In Theory

Use of specialized measurement code with the CR5/v2.05/8-for-jdk13 bits revealed that the gListLock contention is responsible for much of the performance degradation observed with SPECjbb2015. Consequently the primary focus of the next round of changes is/was on switching from course grained Thread::muxAcquire(&gListLock) and Thread::muxRelease(&gListLock) pairs to spin-lock monitor list management. Of course, since the Java Monitor subsystem is full of special cases, the spin-lock list management code has to have a number of special cases which are described here.

The Spin-Lock Monitor List management code was pushed to JDK15 using the following bug id:

JDK-8235795 replace monitor list mux{Acquire,Release}(&gListLock) with spin locks

The Async Monitor Deflation project makes a few additional changes on top of what was pushed via JDK-8235795.

The Simple Case

There is one simple case of spin-lock list management with the Java Monitor subsystem so we'll start with that code as a way to introduce the spin-lock concepts:

```
L1:    while (true) {
L2:      PaddedObjectMonitor* cur = Atomic::load(&g_block_list);
L3:      Atomic::store(&new_blk[0]._next_om, cur);
L4:      if (Atomic::cmpxchg(&g_block_list, cur, new_blk) == cur) {
L5:        Atomic::add(&om_list_globals.population, _BLOCKSIZE - 1);
L6:        break;
L7:      }
L8:    }
```

What the above block of code does is:

- prepends a 'new_blk' to the front of 'g_block_list'
- increments the 'om_list_globals.population' counter to include the number of new elements

The above block of code can be called by multiple threads in parallel and must not lose track of any blocks. Of course, the "must not lose track of any blocks" part is where all the details come in:

- L2 *loads* the current 'g_block_list' value into 'cur'.

- L3 *stores* 'cur' into the 0th element's next field for 'new_blk'.
- L4 is the critical decision point for this list update. *cmpxchg* will change 'g_block_list' to 'new_blk' iff 'g_block_list' == 'cur' (publish it).
  - if the *cmpxchg* return value is 'cur', then we succeeded with the list update and we atomically update 'om_list_globals.population' to match.
  - Otherwise we loop around and do everything again from L2. This is the "spin" part of spin-lock. 🙂

At the point that *cmpxchg* has published the new 'g_block_list' value, 'new_blk' is now first block in the list and the 0th element's next field is used to find the previous first block; all of the monitor list blocks are chained together via the next field in the block's 0th element. It is the use of *cmpxchg* to update 'g_block_list' and the checking of the return value from *cmpxchg* that insures that we don't lose track of any blocks.

This example is considered to be the "simple case" because we only prepend to the list (no deletes) and we only use:

- one *load*
- one *store* and
- one *cmpxchg*

to achieve the safe update of the 'g_block_list' value; the atomic increment of the 'om_list_globals.population' counter is considered to be just accounting (pun intended).

The concepts introduced here are:

- update the new thing to refer to head of existing list
- try to update the head of the existing list to refer to the new thing
- retry as needed

Note: The above code snippet comes from ObjectSynchronizer::prepend_block_to_lists(); see that function for more complete context (and comments).

The Not So Simple Case or Taking and Prepending on the Same List Leads to A-B-A Races

Note: This subsection is talking about "Simple Take" and "Simple Prepend" in abstract terms. The purpose of this code and A-B-A example is to introduce the race concepts. The code shown here is not an exact match for the project code and the specific A-B-A example is not (currently) found in the project code.

The left hand column shows "T1" taking a node "A" from the front of the list and it shows the simple code that does that operation. The right hand column shows "T2" prepending a node "B" to the front of the list and it shows the simple code that does that operation. We have a third thread, "T3", that does a take followed by a prepend, but we don't show a column for "T3". Instead we have a column in the middle that shows the results of the interleaved operations of all three threads:

```
    T1: Simple Take:                          |                                          | T2: Simple Prepend:
    ----------------                          |  T1 and T3 see this initial list:        | --------------------
            +---+    +---+    +---+           |          +---+    +---+    +---+          |          +---+
+---+
    head -> | A | -> | X | -> | Y |           |  head -> | A | -> | X | -> | Y |          |  head -> | X | -> |
Y |
            +---+    +---+    +---+           |          +---+    +---+    +---+          |          +---+
+---+
                                              |  T3 takes "A", T2 sees this list:        |
    Take a node     ||                        |          +---+    +---+                  |  Prepend a node  ||
    from the front  ||                        |  head -> | X | -> | Y |                  |  to the front    ||
    of the list     ||                        |          +---+    +---+                  |  of the list     ||
                    \/                        |  T2 prepends "B":                        |                  \/
                                              |          +---+    +---+    +---+          |
            +---+    +---+                    |  head -> | B | -> | X | -> | Y |          |          +---+
+---+    +---+
    head -> | X | -> | Y |                    |          +---+    +---+    +---+          |  head -> | B | -> |
X | -> | Y |
            +---+    +---+                    |  T3 prepends "A":                        |          +---+
+---+    +---+
          +---+                               |          +---+    +---+    +---+    +---+  |
    cur -> | A |                              |  head -> | A | -> | B | -> | X | -> | Y |  |
           +---+                              |          +---+    +---+    +---+    +---+  |
                                              |  T1 takes "A", loses "B":                |
    // "take" a node:                         |  +---+                                   |  // "prepend" a
node:
    while (true) {                            |  | B | ----+                              |  while (true) {
      cur = head;                             |  +---+     |                              |    cur = head;
      next = cur->next;                       |            V                              |    new->next = cur;
      if (cmpxchg(next, &head, cur) == cur) { |          +---+    +---+                   |    if (cmpxchg(new,
&head, cur) == cur) {
        break;  // success changing head      |  head -> | X | -> | Y |                   |      break; //
success changing head
      }                                       |          +---+    +---+                   |    }
    }                                         |         +---+                             |  }
    return cur;                               |  cur -> | A |                             |
                                              |         +---+                             |
```

The "Simple Take" and "Simple Prepend" algorithms are just fine by themselves. The "Simple Prepend" algorithm is almost identical to the algorithm in the "The Simple Case" and just like that algorithm, it works fine if we are only doing prepend operations on the list. Similarly, the "Simple Take" algorithm works just fine if we are only doing take operations on the list; the only thing missing is an empty list check, but that would have clouded the example.

When we allow simultaneous take and prepend operations on the same list, the simple algorithms are exposed to A-B-A races. An A-B-A race is a situation where the head of the list can change from node "A" to node "B" and back to node "A" again without the simple algorithm being aware that critical state has changed. In the middle column of the above diagram, we show what happens when T3 causes the head of the list to change from node "A" to node "B" (a take operation) and back to node "A" (a prepend operation). That A-B-A race causes T1 to lose node "B" when it updates the list head to node "X" instead of node "B" because T1 was unaware that its local 'next' value was stale.

Here's the diagram again with the code in T1 and T2 lined up with the effects of the A-B-A race executed by T3:

```
    T1: Simple Take:                          |                                          | T2: Simple Prepend:
    ----------------                          |  T1 and T3 see this initial list:        | -------------------
    while (true) {                            |          +---+    +---+    +---+         |  :
      cur = head;                             |  head -> | A | -> | X | -> | Y |         |  :
      next = cur->next;                       |          +---+    +---+    +---+         |  :
      :                                       |  T3 takes "A", T2 sees this list:        |  :
      :                                       |          +---+    +---+                  |  :
      :                                       |  head -> | X | -> | Y |                  |  :
      :                                       |          +---+    +---+                  |  while (true) {
      :                                       |  T2 prepends "B":                        |    cur = head;
      :                                       |          +---+    +---+    +---+         |    new->next = cur;
      :                                       |  head -> | B | -> | X | -> | Y |         |    if (cmpxchg(new,
&head, cur) == cur) {
      :                                       |          +---+    +---+    +---+         |      break;
      :                                       |  T3 prepends "A":                        |    }
      :                                       |          +---+    +---+    +---+    +---+  |  }
      :                                       |  head -> | A | -> | B | -> | X | -> | Y |  |
      :                                       |          +---+    +---+    +---+    +---+  |
      :                                       |  T1 takes "A", loses "B":                |
      :                                       |  +---+                                   |
      :                                       |  | B | ----+                             |
      :                                       |  +---+     |                             |
      :                                       |            V                             |
      :                                       |          +---+    +---+                  |
      if (cmpxchg(next, &head, cur) == cur) {  |  head -> | X | -> | Y |                  |
      }                                       |          +---+    +---+                  |
    }                                         |         +---+                            |
    return cur;                               |  cur -> | A |                            |
                                              |         +---+                            |
```

So the simple algorithms are not sufficient when we allow simultaneous take and prepend operations.

Spin-Locking to Solve the A-B-A Race

Note: This subsection is talking about "Spin-Locking" as a solution to the A-B-A race in abstract terms. The purpose of this spin-locking code and A-B-A example is to introduce the solution concepts. The code shown here is not an exact match for the project code.

One solution to the A-B-A race is to spin-lock the next field in a node to indicate that the node is busy. Only one thread can successfully spin-lock the next field in a node at a time and other threads must loop around and retry their spin-locking operation until they succeed. Each thread that spin-locks the next field in a node must unlock the next field when it is done with the node so that other threads can proceed.

Here's the take algorithm modified with spin-locking (still ignores the empty list for clarity):

```
// "take" a node with locking:
while (true) {
  cur = head;
  if (!try_om_lock(cur)) {
    // could not lock cur so try again
    continue;
  }
  if (head != cur) {
    // head changed while locking cur so try again
    om_unlock(cur);
    continue;
  }
  next = unmarked_next(cur);
  // list head is now locked so switch it to next which also makes list head unlocked
  Atomic::store(&head, next);
  om_unlock(cur);  // unlock cur and return it
  return cur;
}
```

The modified take algorithm does not change the list head pointer until it has successfully locked the list head node. Notice that after we lock the list head node we have to verify that the list head pointer hasn't changed in the mean time. Only after we have verified that the node we locked is still the list head is it safe to modify the list head pointer. The locking of the list head prevents the take algorithm from executing in parallel with a prepend algorithm and losing a node.

Also notice that we update the list head pointer with *store* instead of with *cmpxchg*. Since we have the list head locked, we are not racing with other threads to change the list head pointer so we can use a simple *store* instead of the heavy *cmpxchg* hammer.

Here's the prepend algorithm modified with locking (ignores the empty list for clarity):

```
// "prepend" a node with locking:
while (true) {
  cur = head;
  if (!try_om_lock(cur)) {
    // could not lock cur so try again
    continue;
  }
  if (head != cur) {
    // head changed while locking cur so try again
    om_unlock(cur);
    continue;
  }
  next = unmarked_next(cur);
  // list head is now locked so switch it to 'new' which also makes list head unlocked
  Atomic::store(&head, new);
  om_unlock(cur);  // unlock the previous list head
}
```

The modified prepend algorithm does not change the list head pointer until it has successfully locked the list head node. Notice that after we lock the list head node we have to verify that the list head pointer hasn't changed in the mean time. Only after we have verified that the node we locked is still the list head is it safe to modify the list head pointer. The locking of the list head prevents the prepend algorithm from executing in parallel with the take algorithm and losing a node.

Also notice that we update the list head pointer with *store* instead of with *cmpxchg* for the same reasons as the previous algorithm.

Background: ObjectMonitor Movement Between the Lists

The purpose of this subsection is to provide background information about how ObjectMonitors move between the various lists. This project changes the way these movements are implemented, but does not change the movements themselves. For example, newly allocated blocks of ObjectMonitors are always prepending to the global free list; this is true in the baseline and is true in this project. One exception is the addition of the global wait list (see below).

ObjectMonitor Allocation Path

- ObjectMonitors are allocated by ObjectSynchronizer::om_alloc().
- Assume that the calling JavaThread has an empty free list and the global free list is also empty:
  - A block of ObjectMonitors is allocated by the calling JavaThread and prepended to the global free list.
  - ObjectMonitors are taken from the front of the global free list by the calling JavaThread and prepended to the JavaThread's free list by ObjectSynchronizer::om_release().
  - An ObjectMonitor is taken from the front of the JavaThread's free list and prepended to the JavaThread's in-use list (optimistically).

ObjectMonitor Deflation Path

- ObjectMonitors are deflated at a safepoint by:
  ObjectSynchronizer::deflate_monitor_list() calling ObjectSynchronizer::deflate_monitor()
  And when Async Monitor Deflation is enabled, they are deflated by:
  ObjectSynchronizer::deflate_monitor_list_using_JT() calling ObjectSynchronizer::deflate_monitor_using_JT()
- Idle ObjectMonitors are deflated by the ServiceThread when Async Monitor Deflation is enabled. They can also be deflated at a safepoint by the VMThread or by a task worker thread. Safepoint deflation is used when Async Monitor Deflation is disabled or when there is a special deflation request, e.g., System.gc().
- An idle ObjectMonitor is deflated and extracted from its in-use list and prepended to the global wait list. The in-use list can be either the global in-use list or a per-thread in-use list. Deflated ObjectMonitors are always prepended to the global wait list.
  - The om_list_globals.wait_list allows ObjectMonitors to be safely deflated without reuse races.
  - After a handshake/safepoint with all JavaThreads, the ObjectMonitors on the om_list_globals.wait_list are prepended to the global free list.

ObjectMonitor Flush Path

- ObjectMonitors are flushed by ObjectSynchronizer::om_flush().
- When a JavaThread exits, the ObjectMonitors on its in-use list are prepended on the global in-use list and the ObjectMonitors on its free list are prepended on the global free list.

ObjectMonitor Linkage Path

- ObjectMonitors are linked with objects by ObjectSynchronizer::inflate().
- An inflate() call by one JavaThread can race with an inflate() call by another JavaThread for the same object.
- When inflate() realizes that it failed to link an ObjectMonitor with the target object, it calls ObjectSynchronizer::om_release() to extract the ObjectMonitor from the JavaThread's in-use list and prepends it on the JavaThread's free list.
  Note: Remember that ObjectSynchronizer::om_alloc() optimistically added the newly allocated ObjectMonitor to the JavaThread's in-use list.
- When inflate() successfully links an ObjectMonitor with the target object, that ObjectMonitor stays on the JavaThread's in-use list.

The Lists and Which Threads Touch Them

- global free list:
  - prepended to by JavaThreads that allocated a new block of ObjectMonitors (malloc time)
  - prepended to by JavaThreads that are exiting (and have a non-empty per-thread free list)
  - taken from the head by JavaThreads that need to allocate ObjectMonitor(s) for their per-thread free list (reprovision)
  - prepended to by deflation done by:
    - either the VMThread or a worker thread for safepoint based
    - or the ServiceThread for async monitor deflation
- global in-use list:
  - prepended to by JavaThreads that are exiting (and have a non-empty per-thread free list)
  - extracted from by deflation done by:
    - either the VMThread or a worker thread for safepoint based
    - or the ServiceThread for async monitor deflation
- global wait list:
  - prepended by the ServiceThread during async deflation
  - entire list detached and prepended to the global free list by the ServiceThread during async deflation
  - Note: The global wait list serves the same function as Carsten's gFreeListNextSafepoint list in his prototype.
- per-thread free list:
  - prepended to by a JavaThread when it needs to allocate new ObjectMonitor(s) (reprovision)
  - taken from the head by a JavaThread when it needs to allocate a new ObjectMonitor (inflation)
  - prepended to by a JavaThread when it isn't able to link the object to the ObjectMonitor (failed inflation)
  - entire list detached and prepended to the global free list when the JavaThread is exiting
- per-thread in-use list:
  - prepended to by a JavaThread when it allocates a new ObjectMonitor (inflation, optimistically in-use)
  - extracted from by deflation done by:
    - either the VMThread or a worker thread for safepoint based
    - or the ServiceThread for async monitor deflation
  - entire list detached and prepended to the global in-use list when the JavaThread is exiting

Spin-Lock Monitor List Management In Reality

Prepending To A List That Also Allows Deletes

It is now time to switch from algorithms to real snippets from the code.

The next case to consider for spin-lock list management with the Java Monitor subsystem is prepending to a list that also allows deletes. As you might imagine, the possibility of a prepend racing with a delete makes things more complicated. The solution is to lock the next field in the ObjectMonitor at the head of the list we're trying to prepend to. A successful lock tells other prependers or deleters that the locked ObjectMonitor is busy and they will need to retry their own lock operation.

```
L01:    while (true) {
L02:      om_lock(m);  // Lock m so we can safely update its next field.
L03:      ObjectMonitor* cur = NULL;
L04:      // Lock the list head to guard against A-B-A race:
L05:      if ((cur = get_list_head_locked(list_p)) != NULL) {
L06:        // List head is now locked so we can safely switch it.
L07:        m->set_next_om(cur);  // m now points to cur (and unlocks m)
L08:        Atomic::store(list_p, m);  // Switch list head to unlocked m.
L09:        om_unlock(cur);
L10:        break;
L11:      }
L12:      // The list is empty so try to set the list head.
L13:      assert(cur == NULL, "cur must be NULL: cur=" INTPTR_FORMAT, p2i(cur));
L14:      m->set_next_om(cur);  // m now points to NULL (and unlocks m)
L15:      if (Atomic::cmpxchg(list_p, cur, m) == cur) {
L16:        // List head is now unlocked m.
L17:        break;
L18:      }
L19:      // Implied else: try it all again
L20:    }
L21:    Atomic::inc(count_p);
```

What the above block of code does is:

- prepends an ObjectMonitor 'm' to the front of the list referred to by list_p
  - lock 'm'
  - lock the list head
  - update 'm' to refer to the list head
  - update 'list_p' to refer to 'm'
  - unlock the previous list head
- increments the counter referred to by 'count_p' by one

The above block of code can be called by multiple prependers in parallel or with deleters running in parallel and must not lose track of any ObjectMonitor. Of course, the "must not lose track of any ObjectMonitor" part is where all the details come in:

- L02 locks 'm'; internally we have to loop because another thread (T2) might have 'm' locked and we try again until we have locked it. You might be asking yourself: why does T2 have 'm' locked?

        - Before T1 was trying to prepend 'm' to an in-use list, T1 and T2 were racing to take an ObjectMonitor off the free list.
        - T1 won the race, locked 'm', removed 'm' from the free list and unlocked 'm'; T2 stalled before trying to lock 'm'.
        - T2 resumed and locked 'm', realized that 'm' was no longer the head of the free list, unlocked 'm' and tried it all again.
        - If our thread (T1) does not lock 'm' before it tries to prepend it to an in-use list, then T2's unlocking of 'm' could erase the next value that T1 wants to put in 'm'.
    - L05 tries to lock the list head 'list_p'; if get_list_head_locked() returns non-NULL, we have the list head locked and can safely update it:
        - L07: Update 'm's next field to point to the current list head (which unlocks 'm').
        - L08: *store* 'm' into 'list_p' which switches the list head to an unlocked 'm'.
        - L09: We unlock the previous list head.
    - If get_list_head_locked() returned NULL, we have an empty list:
        - L14: Update 'm's next field to NULL (which unlocks 'm').
        - L15: Try to *cmpxchg* 'list_p' to 'm':
            - if *cmpxchg* works, then we're done.
            - Otherwise, another prepender won the race to update the list head so we have to try again.
    - The counter referred to by 'count_p' is incremented by one.

ObjectMonitor 'm' is safely on the list at the point that we have updated 'list_p' to refer to 'm'. In this subsection's block of code, we also called three new functions: om_lock(), get_list_head_locked() and set_next_om(), that are explained in the next few subsections about helper functions.

Note: The above code snippet comes from prepend_to_common(); see that function for more context and a few more comments.

try_om_lock(), mark_om_ptr(), and set_next_om() Helper Functions

Managing spin-locks on ObjectMonitors has been abstracted into a few helper functions. try_om_lock() is the first interesting one:

```
L1:  static bool try_om_lock(ObjectMonitor* om) {
L2:    // Get current next field without any OM_LOCK_BIT value.
L3:    ObjectMonitor* next = unmarked_next(om);
L4:    if (om->try_set_next_om(next, mark_om_ptr(next)) != next) {
L5:      return false;  // Cannot lock the ObjectMonitor.
L6:    }
L7:    return true;
L8:  }
```

The above function tries to lock the ObjectMonitor:

- If locking is successful, then true is returned.
- Otherwise, false is returned.

The function can be called by multiple threads at the same time and only one thread will succeed in the locking operation (return == true) and all other threads will get return == false. Of course, the "only one thread will succeed" part is where all the details come in:

- L3 *loads* the ObjectMonitor's next field and strips the locking bit:
    - The unlocked value is saved in 'next'.
    - We need the unlocked next value in order to properly detect if the next field was already locked.
- L4 tries to *cmpxchg* a locked 'next' value into the ObjectMonitor's next field:
    - if *cmpxchg* does not work, then we return false:
        - The *cmpxchg* will not work if the next field changes after we *loaded* the value on L3.
        - The *cmpxchg* will not work if the next field is already locked.
    - Otherwise, we return true.

The try_om_lock() function calls another helper function, mark_om_ptr(), that needs a quick explanation:

```
L1:  static ObjectMonitor* mark_om_ptr(ObjectMonitor* om) {
L2:    return (ObjectMonitor*)((intptr_t)om | OM_LOCK_BIT);
L3:  }
```

This function encapsulates the setting of the locking bit in an ObjectMonitor* for the purpose of hiding the details and making the calling code easier to read:

- L2 casts the ObjectMonitor* into a type that will allow the '|' operator to be used.
- We use the 0x1 (OM_LOCK_BIT) bit as our locking value because ObjectMonitors are aligned on a cache line so the low order bit is not used by the normal addressing of an ObjectMonitor*.

set_next_om() is the next interesting function and it also only needs a quick explanation:

```
L1:  inline void ObjectMonitor::set_next_om(ObjectMonitor* value) {
L2:    Atomic::store(&_next_om, value);
L3:  }
```

This function encapsulates the setting of the next field in an ObjectMonitor for the purpose of hiding the details and making the calling code easier to read:

- This function is simply a wrapper around a *store* of an ObjectMonitor* into the next field in an ObjectMonitor.
- The typical "cur->set_next_om(next)" call sequence is easier to read than "OrderAccess::release_store(&cur_next_om, next)".

om_lock() Helper Function

om_lock() is the next interesting helper function:

```
L1:  static void om_lock(ObjectMonitor* om) {
L2:    while (true) {
L3:      if (try_om_lock(om)) {
L4:        return;
L5:      }
L6:    }
L7:  }
```

The above function loops until it locks the target ObjectMonitor. There is nothing particularly special about this function so we don't need any line specific annotations.

Debugging Tip: If there's a bug where an ObjectMonitor's next field is not properly unlocked, then this function will loop forever and the caller will be stuck.

get_list_head_locked() Helper Function

get_list_head_locked() is the next interesting helper function:

```
L01:  static ObjectMonitor* get_list_head_locked(ObjectMonitor** list_p) {
L02:    while (true) {
L03:      ObjectMonitor* mid = Atomic::load(list_p);
L04:      if (mid == NULL) {
L05:        return NULL;  // The list is empty.
L06:      }
L07:      if (try_om_lock(mid)) {
L08:        if (Atomic::load(list_p) != mid) {
L09:          // The list head changed so we have to retry.
L10:          om_unlock(mid);
L11:          continue;
L12:        }
L13:        return mid;
L14:      }
L15:    }
L16:  }
```

The above function tries to lock the list head's ObjectMonitor:

- If the list is empty, NULL is returned.
- Otherwise, the list head's ObjectMonitor* is returned.

The function can be called by more than one thread on the same 'list_p' at a time. False is only returned when 'list_p' refers to an empty list. Otherwise only one thread will return an ObjectMonitor* at a time. Since the ObjectMonitor is locked, any parallel callers to get_list_head_locked() will loop until the list head's ObjectMonitor is no longer locked. That typically happens when the list head's ObjectMonitor is taken off the list and 'list_p' is advanced to the next ObjectMonitor on the list. Of course, making sure that "only one thread will return true at a time" is where all the details come in:

- L03 *loads* the current 'list_p' value into 'mid'.
- L0[45] is the empty list check and the only time that NULL is returned by this function.
- L07 tries to lock 'mid':
  - If locking is not successful, we loop around to try it all again (the "spin" part of spin-lock).
  - L08 *loads* the current 'list_p' value to see if it still matches 'mid':
    - If the list head has changed, then we unlock mid on L10 and try it all again.
    - Otherwise, 'mid' is returned.

When this function returns a non-NULL ObjectMonitor*, the ObjectMonitor is locked and any parallel callers of get_list_head_locked() on the same list will be looping until the list head's ObjectMonitor is no longer locked. The caller that just got the ObjectMonintor* needs to finish up its work quickly.

Debugging Tip: If there's a bug where the list head ObjectMonitor is not properly unlocked, then this function will loop forever and the caller will be stuck.

Taking From The Start Of A List

The next case to consider for spin-lock list management with the Java Monitor subsystem is taking an ObjectMonitor from the start of a list. Taking an ObjectMonitor from the start of a list is a specialized form of delete that is guaranteed to interact with a thread that is prepending to the same list at the same time. Again, the core of the solution is to lock the ObjectMonitor at the head of the list we're trying to take the ObjectMonitor from, but we use slightly different code because we have less linkages to make than a prepend.

```
L01:  static ObjectMonitor* take_from_start_of_common(ObjectMonitor** list_p,
L02:                                                 int* count_p) {
L03:    ObjectMonitor* take = NULL;
L04:    // Lock the list head to guard against A-B-A race:
L05:    if ((take = get_list_head_locked(list_p)) == NULL) {
L06:      return NULL;  // None are available.
L07:    }
L08:    ObjectMonitor* next = unmarked_next(take);
L09:    // Switch locked list head to next (which unlocks the list head, but
L10:    // leaves take locked):
L11:    Atomic::store(list_p, next);
L12:    Atomic::dec(count_p);
L13:    // Unlock take, but leave the next value for any lagging list
L14:    // walkers. It will get cleaned up when take is prepended to
L15:    // the in-use list:
L16:    om_unlock(take);
L17:    return take;
L18:  }
```

What the above function does is:

- Tries to lock the ObjectMonitor at the head of the list:
  - Locking will only fail if the list is empty so that NULL can be returned.
  - Otherwise get_list_head_locked() will loop until the ObjectMonitor at the list head has been locked.
- Get the next pointer from the taken ObjectMonitor.
- Updates the list head to refer to the next ObjectMonitor.
- Decrements the counter referred to by 'count_p'.
- Unlocks the taken ObjectMonitor.

The function can be called by more than one thread at a time and each thread will take a unique ObjectMonitor from the start of the list (if one is available) without losing any other ObjectMonitors on the list. Of course, the "take a unique ObjectMonitor" and "without losing any other ObjectMonitors" parts are where all the details come in:

- L05 tries to lock the list head:
  - get_list_head_locked() returns NULL if the list is empty so we return NULL on L06.
  - Otherwise, 'take' is a pointer to the locked list head.
- L08 gets the next pointer from take.
- L11 *stores* 'next' into 'list_p'.
  You might be asking yourself: Why *store* instead of *cmpxchg*?
  - get_list_head_locked() only returns to the caller when it has locked the ObjectMonitor at the head of the list.
  - Because of that guarantee, any prepender or deleter thread that is running in parallel must loop until we have *stored* 'next' into 'list_p' which unlocks the list head.
- L12 decrements the counter referred to by 'count_p'.
- L16 unlocks 'take':
  - Keeping the 'next' value in take's next field allows any lagging list walker to get to the next ObjectMonitor on that list.
  - take's next field will get cleaned up when take is prepended to its target in-use list.
- L17 returns 'take' to the caller.

lock_next_for_traversal() Helper Function

This last helper function exists for making life easier for list walker code. List walker code calls get_list_head_locked() to get the locked list head and then walks the list applying its particular logic to elements in the list. In order to safely walk to the 'next' ObjectMonitor in a list, the list walker code must lock the 'next' ObjectMonitor before unlocking the 'current' ObjectMonitor that it has locked. If a list walker unlocks 'current' before locking 'next', then there is race where 'current' could be modified to refer to something other than the 'next' value that was in place when 'current' was locked. By locking 'next' first and then unlocking 'current', the list walker can safely advance to 'next'.

```
L01:  static ObjectMonitor* lock_next_for_traversal(ObjectMonitor* cur) {
L02:    assert(is_locked(cur), "cur=" INTPTR_FORMAT " must be locked", p2i(cur));
L03:    ObjectMonitor* next = unmarked_next(cur);
L04:    if (next == NULL) {  // Reached the end of the list.
L05:      om_unlock(cur);
L06:      return NULL;
L07:    }
L08:    om_lock(next);   // Lock next before unlocking current to keep
L09:    om_unlock(cur);  // from being by-passed by another thread.
L10:    return next;
L11:  }
```

This function is pretty straight forward so there are no detailed notes for it.

Using The New Spin-Lock Monitor List Functions

ObjectSynchronizer::om_alloc(Thread* self, ...)

ObjectSynchronizer::om_alloc() is responsible for allocating an ObjectMonitor and returning it to the caller. It has a three step algorithm:

1) Try to allocate from self's local free-list:

- take_from_start_of_om_free_list(self) takes an ObjectMonitor from self's free list (if possible).
- prepend_to_om_in_use_list(self, m) prepends the newly taken ObjectMonitor to self's in-use list.

2) Try to allocate from the global free list (up to selfom_free_provision times):

- take_from_start_of_global_free_list() takes an ObjectMonitor from the global free list (if possible).
- om_release(self, take, false) prepends the newly taken ObjectMonitor to self's free list.
- Retry the allocation from step 1.

3) Allocate a block of new ObjectMonitors:

- prepend_block_to_lists() prepends the newly allocated block to 'g_block_list' and to the global free list.
- Retry the allocation from step 1.

ObjectSynchronizer::om_release(Thread* self, ObjectMonitor* m, bool from_per_thread_alloc)

ObjectSynchronizer::om_release() is responsible for putting an ObjectMonitor on self's free list. If 'from_per_thread_alloc' is true, then om_release() is also responsible for extracting the ObjectMonitor from self's in-use list. The extraction from self's in-use list must happen first:

```
L01:    if (from_per_thread_alloc) {
L02:      if ((mid = get_list_head_locked(&self->om_in_use_list)) == NULL) {
L03:        fatal("thread=" INTPTR_FORMAT " in-use list must not be empty.", p2i(self));
L04:      }
L05:      next = unmarked_next(mid);
L06:      if (m == mid) {
L07:        Atomic::store(&self->om_in_use_list, next);
L08:      } else if (m == next) {
L09:        mid = next;
L10:        om_lock(mid);
L11:        next = unmarked_next(mid);
L12:        self->om_in_use_list->set_next_om(next);
L13:      } else {
L14:        ObjectMonitor* anchor = next;
L15:        om_lock(anchor);
L16:        om_unlock(mid);
L17:        while ((mid = unmarked_next(anchor)) != NULL) {
L18:          if (m == mid) {
L19:            next = unmarked_next(mid);
L20:            anchor->set_next_om(next);
L21:            break;
L22:          } else {
L23:            om_lock(mid);
L24:            om_unlock(anchor);
L25:            anchor = mid;
L26:          }
L27:        }
L28:      }
L29:      Atomic::dec(&self->om_in_use_count);
L30:      om_unlock(mid);
L31:    }
L32:    prepend_to_om_free_list(self, m);
```

Most of the above code block extracts 'm' from self's in-use list; it is not an exact quote from om_release(), but it is the highlights:

- L02 is used to lock self's in-use list head:
  - 'mid' is self's in-use list head and it is locked.
- L05 'next' is the unmarked next field from 'mid'.
- L06 L07: handle first special case where the target ObjectMonitor 'm' matches the list head.
- L08 L12: handle second special case where the target ObjectMonitor 'm' matches next after the list head.
- L14 L30: self's in-use list is traversed looking for the target ObjectMonitor 'm':
  - L18: if the current 'mid' matches 'm':
    - L19: get the next after 'm'
    - L20: update the anchor to refer to the next after 'm'
    - L21: break out since we found a match
  - else
    - L23: lock the current 'mid'
    - L24-5: unlock the current anchor and advance to the new anchor
    - loop around and try again
- L29 L30: we've successfully extracted 'm' from self's in-use list so we decrement self's in-use counter, unlock 'mid' and we're done.

The last line of the code block (L32) prepends 'm' to self's free list.

ObjectSynchronizer::om_flush(Thread* self)

ObjectSynchronizer::om_flush() is reponsible for flushing self's in-use list to the global in-use list and self's free list to the global free list during self's thread exit processing. om_flush() starts with self's in-use list:

```
L01:    if ((in_use_list = get_list_head_locked(&self->om_in_use_list)) != NULL) {
L02:      in_use_tail = in_use_list;
L03:      in_use_count++;
L04:      for (ObjectMonitor* cur_om = unmarked_next(in_use_list); cur_om != NULL;) {
L05:        if (is_locked(cur_om)) {
L06:          while (is_locked(cur_om)) {
L07:            os::naked_short_sleep(1);
L08:          }
L09:          cur_om = unmarked_next(in_use_tail);
L10:          continue;
L11:        }
L12:        if (cur_om->is_free()) {
L13:          cur_om = unmarked_next(in_use_tail);
L14:          continue;
L15:        }
L16:        in_use_tail = cur_om;
L17:        in_use_count++;
L18:        cur_om = unmarked_next(cur_om);
L19:      }
L20:      guarantee(in_use_tail != NULL, "invariant");
L21:      int l_om_in_use_count = Atomic::load(&self->om_in_use_count);
L22:      ADIM_guarantee(l_om_in_use_count == in_use_count, "in-use counts don't match: "
L23:                     "l_om_in_use_count=%d, in_use_count=%d", l_om_in_use_count, in_use_count);
L24:      Atomic::store(&self->om_in_use_count, 0);
L25:      Atomic::store(&self->om_in_use_list, (ObjectMonitor*)NULL);
L26:      om_unlock(in_use_list);
L27:    }
```

The above is not an exact copy of the code block from om_flush(), but it is the highlights. What the above code block needs to do is pretty simple:

- Count the number of ObjectMonitors on the in-use list using 'in_use_count'.
- Point to the last ObjectMonitor on the in-use list using 'in_use_tail'.
- Set self's in-use count to zero.
- Set self's in-use llist to NULL.

However, in this case, there are a lot of details:

- L01 locks the in-use list head (if it is not empty):
  - The in-use list head is kept locked to prevent an async deflation thread from entering the list behind this thread.
    Note: An async deflation thread does check to see if the target thread is exiting, but if it has made it past that check before this thread started exiting, then we're racing.
- L04-L19: loops over the in-use list counting and advancing 'in_use_tail'.
  - L05-L10: 'cur_om' is locked so there must be an async deflater thread or a list walker thread ahead of us so we delay to give it a chance to finish and refetch 'in_use_tail's (possibly changed) next field and try again.
  - L12-L14: 'cur_om' was deflated and its allocation state was changed to Free while it was locked. We just happened to be lucky enough to see it just after it was unlocked (and added to the free list). We refetch 'in_use_tail's (possibly changed) next field and try again.
  - L1[67]: finally 'cur_om' has been completely vetted so we can update 'in_use_tail' and increment 'in_use_count'.
  - L18: advance 'cur_om' to the next ObjectMonitor and do it all again.
- L24: *store* self's in-use count to zero.
  Note: We clear self's in-use count before unlocking self's in-use list head to avoid races.
- L25: *store* self's in-use list head to NULL.
- L26: unlock the disconnected list head.
  Note: Yes, self's in-use list head was kept locked for the whole loop to keep any racing async deflater thread or list walker thread out of the in-use list. After L26, the racing async deflater thread will loop around and see self's in-use list is empty and bail out. Similarly, a racing list walker thread will retry and see self's in-use list is empty and bail out.

The code to process self's free list is much, much simpler because we don't have any races with an async deflater thread like self's in-use list. The only interesting bits:

- *load* self's free list head.
- *store* self's free list count to zero.
- *store* self's free list head to NULL.

The last interesting bits for this function are prepending the local lists to the right global places:

- prepend_list_to_global_free_list(free_list, free_tail, free_count);
- prepend_list_to_global_om_in_use_list(in_use_list, in_use_tail, in_use_count);

ObjectSynchronizer::deflate_monitor_list(ObjectMonitor* volatile * list_p, int volatile * count_p, ObjectMonitor** free_head_p, ObjectMonitor** free_tail_p)

ObjectSynchronizer::deflate_monitor_list() is responsible for deflating idle ObjectMonitors at a safepoint. This function can use the simpler lock-mid-as-we-go protocol since there can be no parallel list deletions due to the safepoint:

```
L01:  int ObjectSynchronizer::deflate_monitor_list(ObjectMonitor** list_p,
L02:                                               int* count_p,
L03:                                               ObjectMonitor** free_head_p,
L04:                                               ObjectMonitor** free_tail_p) {
L05:    ObjectMonitor* cur_mid_in_use = NULL;
L06:    ObjectMonitor* mid = NULL;
L07:    ObjectMonitor* next = NULL;
L08:    int deflated_count = 0;
L09:    if ((mid = get_list_head_locked(list_p)) == NULL) {
L10:      return 0;  // The list is empty so nothing to deflate.
L11:    }
L12:    next = unmarked_next(mid);
L13:    while (true) {
L14:      oop obj = (oop) mid->object();
L15:      if (obj != NULL && deflate_monitor(mid, obj, free_head_p, free_tail_p)) {
L16:        if (cur_mid_in_use == NULL) {
L17:          Atomic::store(list_p, next);
L18:        } else {
L19:          cur_mid_in_use->set_next_om(next);
L20:        }
L21:        deflated_count++;
L22:        Atomic::dec(count_p);
L23:        mid->set_next_om(NULL);
L24:      } else {
L25:        om_unlock(mid);
L26:        cur_mid_in_use = mid;
L27:      }
L28:      mid = next;
L29:      if (mid == NULL) {
L30:        break;  // Reached end of the list so nothing more to deflate.
L31:      }
L32:      om_lock(mid);
L33:      next = unmarked_next(mid);
L34:    }
L35:    return deflated_count;
L36:  }
```

Note: The above version of deflate_monitor_list() uses locking, but those changes were dropped during the code review cycle for JDK-8235795. The locking is only needed when additional calls to audit_and_print_stats() are used during debugging so it was decided that the pushed version would be simpler.

The above is not an exact copy of the code block from deflate_monitor_list(), but it is the highlights. What the above code block needs to do is pretty simple:

- Walk the list and deflate any idle ObjectMonitor that is associated with an object.
- 'free_head_p' and 'free_tail_p' track the list of deflated ObjectMonitors.
- 'deflated_count' is the number of deflated ObjectMonitors on the 'free_head_p' list.

Since we're using the simpler mark-mid-as-we-go protocol, there are not too many details:

- L09: locks the 'list_p' head (if it is not empty)
- L12: 'next' is the unmarked next field from 'mid'.
- L13-L33: We walk each 'mid' in the list and determine if it can be deflated:
  - L15: if 'mid' is associated with an object and can be deflated:
    - L16: if cur_mid_in_use is NULL, we're still processing the head of the in-use list so...
      - L17: we store the list head to 'next'.
    - else
      - L19: we set cur_mid_in_use's next field to 'next'.
    - L21 L23: we've successfully extracted 'mid' from 'list_p's list so we increment 'deflated_count', decrement the counter referred to by 'count_p', set 'mid's next field to NULL and we're done.
      Note: 'mid' is the current tail in the 'free_head_p' list so we have to NULL terminate it (which also unlocks it).
  - L2[4-6]: else 'mid' can't be deflated so unlock mid and advance 'cur_mid_in_use'.
  - L28: advance 'mid'.
  - L29 L30: we reached the end of the list so break out of the loop.
  - L32: lock the new 'mid'
  - L33: update 'next'; loop around and do it all again.
- L35: all done so return 'deflated_count'.

ObjectSynchronizer::deflate_monitor_list_using_JT(ObjectMonitor* volatile * list_p, int volatile * count_p, ObjectMonitor** free_head_p, ObjectMonitor** free_tail_p, ObjectMonitor** saved_mid_p)

ObjectSynchronizer::deflate_monitor_list_using_JT() is responsible for asynchronously deflating idle ObjectMonitors using a JavaThread. This function uses the more complicated lock-cur_mid_in_use-and-mid-as-we-go protocol because om_release() can do list deletions in parallel. We also lock-next-next-as-we-go to prevent an om_flush() that is behind this thread from passing us. Because this function can asynchronously interact with so many other functions, this is the largest clip of code:

```
L01: int ObjectSynchronizer::deflate_monitor_list_using_JT(ObjectMonitor** list_p,
L02:                                                       int* count_p,
L03:                                                       ObjectMonitor** free_head_p,
L04:                                                       ObjectMonitor** free_tail_p,
L05:                                                       ObjectMonitor** saved_mid_in_use_p) {
L06:   JavaThread* self = JavaThread::current();
L07:   ObjectMonitor* cur_mid_in_use = NULL;
L08:   ObjectMonitor* mid = NULL;
L09:   ObjectMonitor* next = NULL;
L10:   ObjectMonitor* next_next = NULL;
L11:   int deflated_count = 0;
L12:   NoSafepointVerifier nsv;
L13:   if (*saved_mid_in_use_p == NULL) {
L14:     if ((mid = get_list_head_locked(list_p)) == NULL) {
L15:       return 0;  // The list is empty so nothing to deflate.
L16:     }
L17:     next = unmarked_next(mid);
L18:   } else {
L19:     cur_mid_in_use = *saved_mid_in_use_p;
L20:     om_lock(cur_mid_in_use);
L21:     mid = unmarked_next(cur_mid_in_use);
L22:     if (mid == NULL) {
L23:       om_unlock(cur_mid_in_use);
L24:       *saved_mid_in_use_p = NULL;
L25:       return 0;  // The remainder is empty so nothing more to deflate.
L26:     }
L27:     om_lock(mid);
L28:     next = unmarked_next(mid);
L29:   }
L30:   while (true) {
L31:     if (next != NULL) {
L32:       om_lock(next);
L33:       next_next = unmarked_next(next);
L34:     }
L35:     if (mid->object() != NULL && mid->is_old() &&
L36:         deflate_monitor_using_JT(mid, free_head_p, free_tail_p)) {
L37:       if (cur_mid_in_use == NULL) {
L38:         Atomic::store(list_p, next);
L39:       } else {
L40:         ObjectMonitor* locked_next = mark_om_ptr(next);
L41:         cur_mid_in_use->set_next_om(locked_next);
L42:       }
L43:       deflated_count++;
L44:       Atomic::dec(count_p);
L45:       mid->set_next_om(NULL);
L46:       mid = next;  // mid keeps non-NULL next's locked state
L47:       next = next_next;
L48:     } else {
L49:       if (cur_mid_in_use != NULL) {
L50:         om_unlock(cur_mid_in_use);
L51:       }
L52:       cur_mid_in_use = mid;
L53:       mid = next;  // mid keeps non-NULL next's locked state
L54:       next = next_next;
L55:       if (SafepointMechanism::should_block(self) &&
L56:           cur_mid_in_use != Atomic::load(list_p) && cur_mid_in_use->is_old()) {
L57:         *saved_mid_in_use_p = cur_mid_in_use;
L58:         om_unlock(cur_mid_in_use);
L59:         if (mid != NULL) {
L60:           om_unlock(mid);
L61:         }
L62:         return deflated_count;
L63:       }
L64:     }
L65:     if (mid == NULL) {
L66:       if (cur_mid_in_use != NULL) {
L67:         om_unlock(cur_mid_in_use);
L68:       }
L69:       break;  // Reached end of the list so nothing more to deflate.
L70:     }
L71:   }
L72:   *saved_mid_in_use_p = NULL;
L73:   return deflated_count;
L74: }
```

The above is not an exact copy of the code block from deflate_monitor_list_using_JT(), but it is the highlights. What the above code block needs to do is pretty simple:

- Walk the list and deflate any idle ObjectMonitor that is associated with an object.
- 'free_head_p' and 'free_tail_p' track the list of deflated ObjectMonitors.
- 'deflated_count' is the number of deflated ObjectMonitors on the 'free_head_p' list.

Since we're using the more complicated lock-cur_mid_in_use-and-mid-as-we-go protocol and also the lock-next-next-as-we-go protocol, there is a mind numbing amount of detail:

- L1[3-7]: Handle the initial setup if we are not resuming after a safepoint or a handshake:
  - L14: locks the 'list_p' head (if it is not empty):
  - L17: 'next' is the unmarked next field from 'mid'.
- L18-L28: Handle the initial setup if we are resuming after a safepoint or a handshake:
  - L20: lock 'cur_mid_in_use'
  - L21: update 'mid'
  - L22-L25: If 'mid' == NULL, then we've resumed context at the end of the list so we're done.
  - L27: lock 'mid'
  - L28: update 'next'
- L30-L71: We walk each 'mid' in the list and determine if it can be deflated:
  - L3[1-3]: if next != NULL, then lock 'next' and update 'next_next'
  - L35-L47: if 'mid' is associated with an object, 'mid' is old, and can be deflated:
    - L37: if cur_mid_in_use is NULL, we're still processing the head of the in-use list so...
      - L38: we *store* the list head to 'next'.
    - else
      - L40: make a locked copy of 'next'
      - L41: we set cur_mid_in_use's next field to 'locked_next'.
    - L43 L45: we've successfully extracted 'mid' from 'list_p's list so we increment 'deflated_count', decrement the counter referred to by 'count_p', set 'mid's next field to NULL and we're done.
      Note: 'mid' is the current tail in the 'free_head_p' list so we have to NULL terminate it (which also unlocks it).
    - L46: advance 'mid' to 'next'.
      Note: 'mid' keeps non-NULL 'next's locked state
    - L47: advance 'next' to 'next_next'.
  - L48-L63: 'mid' can't be deflated so we have to carefully advance the list pointers:
    - L49,50: if cur_mid_in_use != NULL, then unlock 'cur_mid_in_use'.
    - L52: advance 'cur_mid_in_use' to 'mid'.
      Note: 'mid' is still locked and 'cur_mid_in_use' keeps that state.
    - L53: advance 'mid' to 'next'.
      Note: A non-NULL 'next' is still locked and 'mid' keeps that state.
    - L54: advance 'next' to 'next_next'.
    - L55-L62: Handle a safepoint or a handshake if one has started and it is safe to do so.
  - L65-L69: we reached the end of the list:
    - L6[67]: if cur_mid_in_use != NULL, then unlock 'cur_mid_in_use'.
    - L69: break out of the loop because we are done
- L72: not pausing for a safepoint or handshake so clear saved state.
- L73: all done so return 'deflated_count'.

ObjectSynchronizer::deflate_idle_monitors(...)

ObjectSynchronizer::deflate_idle_monitors() handles deflating idle monitors at a safepoint from the global in-use list using ObjectSynchronizer::deflate_monitor_list(). There are only a few things that are worth mentioning:

- Atomic::load(&om_list_globals.in_use_list) is used to get the latest global in-use list.
- Atomic::load(&om_list_globals.in_use_count) is used to get the latest global in-use count.
- prepend_list_to_global_free_list(free_head_p, free_tail_p, deflated_count) is used to prepend the deflated ObjectMonitors on the global free list.

ObjectSynchronizer::deflate_common_idle_monitors_using_JT(bool is_global, JavaThread* target)

ObjectSynchronizer::deflate_common_idle_monitors_using_JT() handles asynchronously deflating idle monitors from either the global in-use list or a per-thread in-use list using ObjectSynchronizer::deflate_monitor_list_using_JT(). There are only a few things that are worth mentioning:

- Atomic::load(&om_list_globals.in_use_count) is used to get the latest global in-use count.
- Atomic::load(&targetom_in_use_count) is used to get the latest per-thread in-use count.
- prepend_list_to_global_free_list(free_head_p, free_tail_p, local_deflated_count) is used to prepend the deflated ObjectMonitors on the global free list.

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 the ServiceThread which offloads the safepoint cleanup mechanism.
  - Having the ServiceThread 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.
- New diagnostic option '-XX:AsyncDeflationInterval' that is default 250 millis; this this option controls how frequently we async default idle monitors when MonitorUsedDeflationThreshold is exceeded.

- Everything else is just monitor list management, infrastructure, logging, debugging and the like. :-)

Monitor Deflation Invocation Details

- The existing safepoint deflation mechanism is still invoked at safepoint "cleanup" time when '-XX:AsyncDeflateIdleMonitors' is false or when a special cleanup request is made.
- SafepointSynchronize::do_cleanup_tasks() calls:
  - ObjectSynchronizer::prepare_deflate_idle_monitors()
  - A ParallelSPCleanupTask is used to perform the tasks (possibly using parallel tasks):
    - A ParallelSPCleanupThreadClosure is used to perform the per-thread tasks:
      - ObjectSynchronizer::deflate_thread_local_monitors() to deflate per-thread idle monitors
    - ObjectSynchronizer::deflate_idle_monitors() to deflate global idle monitors
  - ObjectSynchronizer::finish_deflate_idle_monitors()

- If MonitorUsedDeflationThreshold is exceeded (default is 90%, 0 means off), then the ServiceThread will invoke a cleanup safepoint when '-XX:AsyncDeflateIdleMonitors' is false. When '-XX:AsyncDeflateIdleMonitors' is true, the ServiceThread will call ObjectSynchronizer::deflate_idle_monitors_using_JT().
  - This experimental flag was added in JDK10 via:

    JDK-8181859 Monitor deflation is not checked in cleanup path

  - For this option, exceeded means:

    ((om_list_globals.population - om_list_globals.free_count) / om_list_globals.population) > NN%

- Changes to the safepoint deflation mechanism by the Async Monitor Deflation project (when async deflation is enabled):
  - If System.gc() is called, then a special deflation request is made which invokes the safepoint deflation mechanism.
  - Added the AsyncDeflationInterval diagnostic option (default 250 millis, 0 means off) to prevent MonitorUsedDeflationThreshold requests from swamping the ServiceThread.
    - Description: Async deflate idle monitors every so many milliseconds when MonitorUsedDeflationThreshold is exceeded (0 is off).
    - A special deflation request can cause an async deflation to happen sooner than AsyncDeflationInterval.
  - SafepointSynchronize::is_cleanup_needed() now calls:
    - ObjectSynchronizer::is_safepoint_deflation_needed() instead of ObjectSynchronizer::is_cleanup_needed().
    - is_safepoint_deflation_needed() returns true only if a special deflation request is made (see above).
  - SafepointSynchronize::do_cleanup_tasks() now (indirectly) calls:
    - ObjectSynchronizer::do_safepoint_work() instead of ObjectSynchronizer::deflate_idle_monitors().
    - do_cleanup_tasks() can be called for non deflation related cleanup reasons and that will still result in a call to do_safepoint_work().
  - ObjectSynchronizer::do_safepoint_work() only does the safepoint cleanup tasks if there is a special deflation request. Otherwise it just sets the is_async_deflation_requested flag and notifies the ServiceThread.
  - ObjectSynchronizer::deflate_idle_monitors() and ObjectSynchronizer::deflate_thread_local_monitors() do nothing unless there is a special deflation request.

- Changes to the ServiceThread mechanism by the Async Monitor Deflation project (when async deflation is enabled):
  - The ServiceThread will wake up every GuaranteedSafepointInterval to check for cleanup tasks.
    - This allows is_async_deflation_needed() to be checked at the same interval.
  - The ServiceThread handles deflating global idle monitors and deflating the per-thread idle monitors by calling ObjectSynchronizer::deflate_idle_monitors_using_JT().

- Other invocation changes by the Async Monitor Deflation project (when async deflation is enabled):
  - VM_Exit::doit_prologue() will request a special cleanup to reduce the noise in 'monitorinflation' logging at VM exit time.
  - Before the final safepoint in a non-System.exit() end to the VM, we will request a special cleanup to reduce the noise in 'monitorinflation' logging at VM exit time.
  - The following whitebox test functions will request a special cleanup:
    - WB_G1StartMarkCycle()
    - WB_FullGC()
    - WB_ForceSafepoint()

Gory Details

- Counterpart function mapping for those that know the existing code:
  - ObjectSynchronizer class:
    - deflate_idle_monitors() has deflate_idle_monitors_using_JT(), 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:
    - clear() has clear_using_JT()
- These functions recognize the Async Monitor Deflation protocol and adapt their operations:
  - ObjectMonitor::enter()
  - ObjectMonitor::EnterI()
  - ObjectSynchronizer::quick_enter()
  - ObjectSynchronizer::deflate_monitor()
  - Note: These changes include handling the lingering owner == DEFLATER_MARKER value.

- Also these functions had to adapt and retry their operations:
  - ObjectSynchronizer::FastHashCode()
  - 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.
- 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. :-)