OpenJDK / jdk8 / jdk8 / hotspot

view src/share/vm/oops/markOop.hpp @ 5820:87ee5ee27509

log
graph
tags
branches

changeset
browse

file
latest
diff
comparison
annotate
file log
raw

help

Added tag jdk8-b132 for changeset 0c94c41dcd70

author	katleman
date	Tue, 04 Mar 2014 11:51:03 -0800 (2014-03-05)
parents	da91efe96a93
children	

line source **line wrap: on**

```
/*
 * Copyright (c) 1997, 2012, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

#ifndef SHARE_VM_OOPS_MARKOOP_HPP
#define SHARE_VM_OOPS_MARKOOP_HPP

#include "oops/oop.hpp"
```

```

// The markOop describes the header of an object.
//
// Note that the mark is not a real oop but just a word.
// It is placed in the oop hierarchy for historical reasons.
//
// Bit-format of an object header (most significant first, big endian layout below):
//
//  32 bits:
//  --------
//             hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal object)
//             JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object)
//             size:32 ------------------------------------------>| (CMS free block)
//             PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted object)
//
//  64 bits:
//  --------
//  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal object)
//  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased object)
//  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promoted object)
//  size:64 ----------------------------------------------------->| (CMS free block)
//
//  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && normal object)
//  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && biased object)
//  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CMS promoted object)
//  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CMS free block)
//
//  - hash contains the identity hash value: largest value is
//    31 bits, see os::random().  Also, 64-bit vm's require
//    a hash value no bigger than 32 bits because they will not
//    properly generate a mask larger than that: see library_call.cpp
//    and c1_CodePatterns_sparc.cpp.
//
//  - the biased lock pattern is used to bias a lock toward a given
//    thread. When this pattern is set in the low three bits, the lock
//    is either biased toward a given thread or "anonymously" biased,
//    indicating that it is possible for it to be biased. When the
//    lock is biased toward a given thread, locking and unlocking can
//    be performed by that thread without using atomic operations.
//    When a lock's bias is revoked, it reverts back to the normal
//    locking scheme described below.
```

```
//
//    Note that we are overloading the meaning of the "unlocked" state
//    of the header. Because we steal a bit from the age we can
//    guarantee that the bias pattern will never be seen for a truly
//    unlocked object.
//
//    Note also that the biased state contains the age bits normally
//    contained in the object header. Large increases in scavenge
//    times were seen when these bits were absent and an arbitrary age
//    assigned to all biased objects, because they tended to consume a
//    significant fraction of the eden semispaces and were not
//    promoted promptly, causing an increase in the amount of copying
//    performed. The runtime system aligns all JavaThread* pointers to
//    a very large value (currently 128 bytes (32bVM) or 256 bytes (64bVM))
//    to make room for the age bits & the epoch bits (used in support of
//    biased locking), and for the CMS "freeness" bit in the 64bVM (+COOPs).
//
//    [JavaThread* | epoch | age | 1 | 01]       lock is biased toward given thread
//    [0           | epoch | age | 1 | 01]       lock is anonymously biased
//
//  - the two lock bits are used to describe three states: locked/unlocked and monitor.
//
//    [ptr             | 00]  locked             ptr points to real header on stack
//    [header      | 0 | 01]  unlocked           regular object header
//    [ptr             | 10]  monitor            inflated lock (header is wapped out)
//    [ptr             | 11]  marked             used by markSweep to mark an object
//                                               not valid at any other time
//
//    We assume that stack/thread pointers have the lowest two bits cleared.

class BasicLock;
class ObjectMonitor;
class JavaThread;

class markOopDesc: public oopDesc {
 private:
  // Conversion
  uintptr_t value() const { return (uintptr_t) this; }

 public:
  // Constants
  enum { age_bits                 = 4,
         lock_bits                = 2,
         biased_lock_bits         = 1,
         max_hash_bits            = BitsPerWord - age_bits - lock_bits - biased_lock_bits,
         hash_bits                = max_hash_bits > 31 ? 31 : max_hash_bits,
         cms_bits                 = LP64_ONLY(1) NOT_LP64(0),
         epoch_bits               = 2
  };

  // The biased locking code currently requires that the age bits be
  // contiguous to the lock bits.
  enum { lock_shift               = 0,
         biased_lock_shift        = lock_bits,
         age_shift                = lock_bits + biased_lock_bits,
         cms_shift                = age_shift + age_bits,
         hash_shift               = cms_shift + cms_bits,
         epoch_shift              = hash_shift
  };

  enum { lock_mask                = right_n_bits(lock_bits),
         lock_mask_in_place       = lock_mask << lock_shift,
         biased_lock_mask         = right_n_bits(lock_bits + biased_lock_bits),
         biased_lock_mask_in_place= biased_lock_mask << lock_shift,
         biased_lock_bit_in_place = 1 << biased_lock_shift,
         age_mask                 = right_n_bits(age_bits),
         age_mask_in_place        = age_mask << age_shift,
         epoch_mask               = right_n_bits(epoch_bits),
         epoch_mask_in_place      = epoch_mask << epoch_shift,
         cms_mask                 = right_n_bits(cms_bits),
         cms_mask_in_place        = cms_mask << cms_shift
#ifndef _WIN64
         ,hash_mask               = right_n_bits(hash_bits),
         hash_mask_in_place       = (address_word)hash_mask << hash_shift
#endif
  };

  // Alignment of JavaThread pointers encoded in object header required by biased locking
  enum { biased_lock_alignment    = 2 << (epoch_shift + epoch_bits)
  };

#ifdef _WIN64
```

```
    // These values are too big for Win64
    const static uintptr_t hash_mask = right_n_bits(hash_bits);
    const static uintptr_t hash_mask_in_place  =
                            (address_word)hash_mask << hash_shift;
#endif

  enum { locked_value             = 0,
         unlocked_value           = 1,
         monitor_value            = 2,
         marked_value             = 3,
         biased_lock_pattern      = 5
  };

  enum { no_hash                  = 0 };  // no hash value assigned

  enum { no_hash_in_place         = (address_word)no_hash << hash_shift,
         no_lock_in_place         = unlocked_value
  };

  enum { max_age                  = age_mask };

  enum { max_bias_epoch           = epoch_mask };

  // Biased Locking accessors.
  // These must be checked by all code which calls into the
  // ObjectSynchronizer and other code. The biasing is not understood
  // by the lower-level CAS-based locking code, although the runtime
  // fixes up biased locks to be compatible with it when a bias is
  // revoked.
  bool has_bias_pattern() const {
    return (mask_bits(value(), biased_lock_mask_in_place) == biased_lock_pattern);
  }
  JavaThread* biased_locker() const {
    assert(has_bias_pattern(), "should not call this otherwise");
    return (JavaThread*) ((intptr_t) (mask_bits(value(), ~(biased_lock_mask_in_place | age_mask_in_place | epoch_mask_in_place))));
  }
  // Indicates that the mark has the bias bit set but that it has not
  // yet been biased toward a particular thread
  bool is_biased_anonymously() const {
    return (has_bias_pattern() && (biased_locker() == NULL));
  }
  // Indicates epoch in which this bias was acquired. If the epoch
  // changes due to too many bias revocations occurring, the biases
  // from the previous epochs are all considered invalid.
  int bias_epoch() const {
    assert(has_bias_pattern(), "should not call this otherwise");
    return (mask_bits(value(), epoch_mask_in_place) >> epoch_shift);
  }
  markOop set_bias_epoch(int epoch) {
    assert(has_bias_pattern(), "should not call this otherwise");
    assert((epoch & (~epoch_mask)) == 0, "epoch overflow");
    return markOop(mask_bits(value(), ~epoch_mask_in_place) | (epoch << epoch_shift));
  }
  markOop incr_bias_epoch() {
    return set_bias_epoch((1 + bias_epoch()) & epoch_mask);
  }
  // Prototype mark for initialization
  static markOop biased_locking_prototype() {
    return markOop( biased_lock_pattern );
  }

  // lock accessors (note that these assume lock_shift == 0)
  bool is_locked()   const {
    return (mask_bits(value(), lock_mask_in_place) != unlocked_value);
  }
  bool is_unlocked() const {
    return (mask_bits(value(), biased_lock_mask_in_place) == unlocked_value);
  }
  bool is_marked()   const {
    return (mask_bits(value(), lock_mask_in_place) == marked_value);
  }
  bool is_neutral()  const { return (mask_bits(value(), biased_lock_mask_in_place) == unlocked_value); }

  // Special temporary state of the markOop while being inflated.
  // Code that looks at mark outside a lock need to take this into account.
  bool is_being_inflated() const { return (value() == 0); }

  // Distinguished markword value - used when inflating over
  // an existing stacklock.  0 indicates the markword is "BUSY".
  // Lockword mutators that use a LD...CAS idiom should always
```

```
// check for and avoid overwriting a 0 value installed by some
// other thread.  (They should spin or block instead.  The 0 value
// is transient and *should* be short-lived).
static markOop INFLATING() { return (markOop) 0; }    // inflate-in-progress

// Should this header be preserved during GC?
inline bool must_be_preserved(oop obj_containing_mark) const;
inline bool must_be_preserved_with_bias(oop obj_containing_mark) const;

// Should this header (including its age bits) be preserved in the
// case of a promotion failure during scavenge?
// Note that we special case this situation. We want to avoid
// calling BiasedLocking::preserve_marks()/restore_marks() (which
// decrease the number of mark words that need to be preserved
// during GC) during each scavenge. During scavenges in which there
// is no promotion failure, we actually don't need to call the above
// routines at all, since we don't mutate and re-initialize the
// marks of promoted objects using init_mark(). However, during
// scavenges which result in promotion failure, we do re-initialize
// the mark words of objects, meaning that we should have called
// these mark word preservation routines. Currently there's no good
// place in which to call them in any of the scavengers (although
// guarded by appropriate locks we could make one), but the
// observation is that promotion failures are quite rare and
// reducing the number of mark words preserved during them isn't a
// high priority.
inline bool must_be_preserved_for_promotion_failure(oop obj_containing_mark) const;
inline bool must_be_preserved_with_bias_for_promotion_failure(oop obj_containing_mark) const;

// Should this header be preserved during a scavenge where CMS is
// the old generation?
// (This is basically the same body as must_be_preserved_for_promotion_failure(),
// but takes the Klass* as argument instead)
inline bool must_be_preserved_for_cms_scavenge(Klass* klass_of_obj_containing_mark) const;
inline bool must_be_preserved_with_bias_for_cms_scavenge(Klass* klass_of_obj_containing_mark) const;

// WARNING: The following routines are used EXCLUSIVELY by
// synchronization functions. They are not really gc safe.
// They must get updated if markOop layout get changed.
markOop set_unlocked() const {
  return markOop(value() | unlocked_value);
}
bool has_locker() const {
  return ((value() & lock_mask_in_place) == locked_value);
}
BasicLock* locker() const {
  assert(has_locker(), "check");
  return (BasicLock*) value();
}
bool has_monitor() const {
  return ((value() & monitor_value) != 0);
}
ObjectMonitor* monitor() const {
  assert(has_monitor(), "check");
  // Use xor instead of &~ to provide one extra tag-bit check.
  return (ObjectMonitor*) (value() ^ monitor_value);
}
bool has_displaced_mark_helper() const {
  return ((value() & unlocked_value) == 0);
}
markOop displaced_mark_helper() const {
  assert(has_displaced_mark_helper(), "check");
  intptr_t ptr = (value() & ~monitor_value);
  return *(markOop*)ptr;
}
void set_displaced_mark_helper(markOop m) const {
  assert(has_displaced_mark_helper(), "check");
  intptr_t ptr = (value() & ~monitor_value);
  *(markOop*)ptr = m;
}
markOop copy_set_hash(intptr_t hash) const {
  intptr_t tmp = value() & (~hash_mask_in_place);
  tmp |= ((hash & hash_mask) << hash_shift);
  return (markOop)tmp;
}
// it is only used to be stored into BasicLock as the
// indicator that the lock is using heavyweight monitor
static markOop unused_mark() {
  return (markOop) marked_value;
}
// the following two functions create the markOop to be
// stored into object header, it encodes monitor info
static markOop encode(BasicLock* lock) {
  return (markOop) lock;
}
static markOop encode(ObjectMonitor* monitor) {
  intptr_t tmp = (intptr_t) monitor;
  return (markOop) (tmp | monitor_value);
}
static markOop encode(JavaThread* thread, uint age, int bias_epoch) {
  intptr_t tmp = (intptr_t) thread;
  assert(UseBiasedLocking && ((tmp & (epoch_mask_in_place | age_mask_in_place | biased_lock_mask_in_place)) == 0), "misaligned JavaThread pointer");
  assert(age <= max_age, "age too large");
  assert(bias_epoch <= max_bias_epoch, "bias epoch too large");
  return (markOop) (tmp | (bias_epoch << epoch_shift) | (age << age_shift) | biased_lock_pattern);
}

// used to encode pointers during GC
markOop clear_lock_bits() { return markOop(value() & ~lock_mask_in_place); }

// age operations
markOop set_marked()   { return markOop((value() & ~lock_mask_in_place) | marked_value); }
markOop set_unmarked() { return markOop((value() & ~lock_mask_in_place) | unlocked_value); }

uint    age()               const { return mask_bits(value() >> age_shift, age_mask); }
markOop set_age(uint v) const {
  assert((v & ~age_mask) == 0, "shouldn't overflow age field");
  return markOop((value() & ~age_mask_in_place) | (((uintptr_t)v & age_mask) << age_shift));
}
markOop incr_age()          const { return age() == max_age ? markOop(this) : set_age(age() + 1); }

// hash operations
intptr_t hash() const {
  return mask_bits(value() >> hash_shift, hash_mask);
}

bool has_no_hash() const {
  return hash() == no_hash;
}

// Prototype mark for initialization
static markOop prototype() {
  return markOop( no_hash_in_place | no_lock_in_place );
}

// Helper function for restoration of unmarked mark oops during GC
static inline markOop prototype_for_object(oop obj);

// Debugging
void print_on(outputStream* st) const;

// Prepare address of oop for placement into mark
inline static markOop encode_pointer_as_mark(void* p) { return markOop(p)->set_marked(); }

// Recover address of oop from encoded form used in mark
inline void* decode_pointer() { if (UseBiasedLocking && has_bias_pattern()) return NULL; return clear_lock_bits(); }

// These markOops indicate cms free chunk blocks and not objects.
// In 64 bit, the markOop is set to distinguish them from oops.
// These are defined in 32 bit mode for vmStructs.
const static uintptr_t cms_free_chunk_pattern  = 0x1;

// Constants for the size field.
enum { size_shift                = cms_shift + cms_bits,
       size_bits                 = 35    // need for compressed oops 32G
     };
// These values are too big for Win64
const static uintptr_t size_mask = LP64_ONLY(right_n_bits(size_bits))
                                   NOT_LP64(0);
const static uintptr_t size_mask_in_place =
                                   (address_word)size_mask << size_shift;

#ifdef _LP64
static markOop cms_free_prototype() {
  return markOop(((intptr_t)prototype() & ~cms_mask_in_place) |
                 ((cms_free_chunk_pattern & cms_mask) << cms_shift));
}
uintptr_t cms_encoding() const {
  return mask_bits(value() >> cms_shift, cms_mask);
}
bool is_cms_free_chunk() const {
  return is_neutral() &&
         (cms_encoding() & cms_free_chunk_pattern) == cms_free_chunk_pattern;
}

size_t get_size() const       { return (size_t)(value() >> size_shift); }
static markOop set_size_and_free(size_t size) {
  assert((size & ~size_mask) == 0, "shouldn't overflow size field");
  return markOop(((intptr_t)cms_free_prototype() & ~size_mask_in_place) |
                 (((intptr_t)size & size_mask) << size_shift));
}
#endif // _LP64
};

#endif // SHARE_VM_OOPS_MARKOOP_HPP
```