锁之语

- 锁之虑
- 锁之颜
- 锁之里

锁之虑

锁，古谓之键，今谓之锁

线程安全问题

方法1：栈隔离（threadlocal）

方法2：不可变化 (final)

方法3：加锁

资源使用约束

锁之颜

悲观锁和乐观锁

独占锁和共享锁

公平锁和非公平锁

偏向锁

重入锁

自旋锁

阻塞锁

读写锁

中断锁

。。。

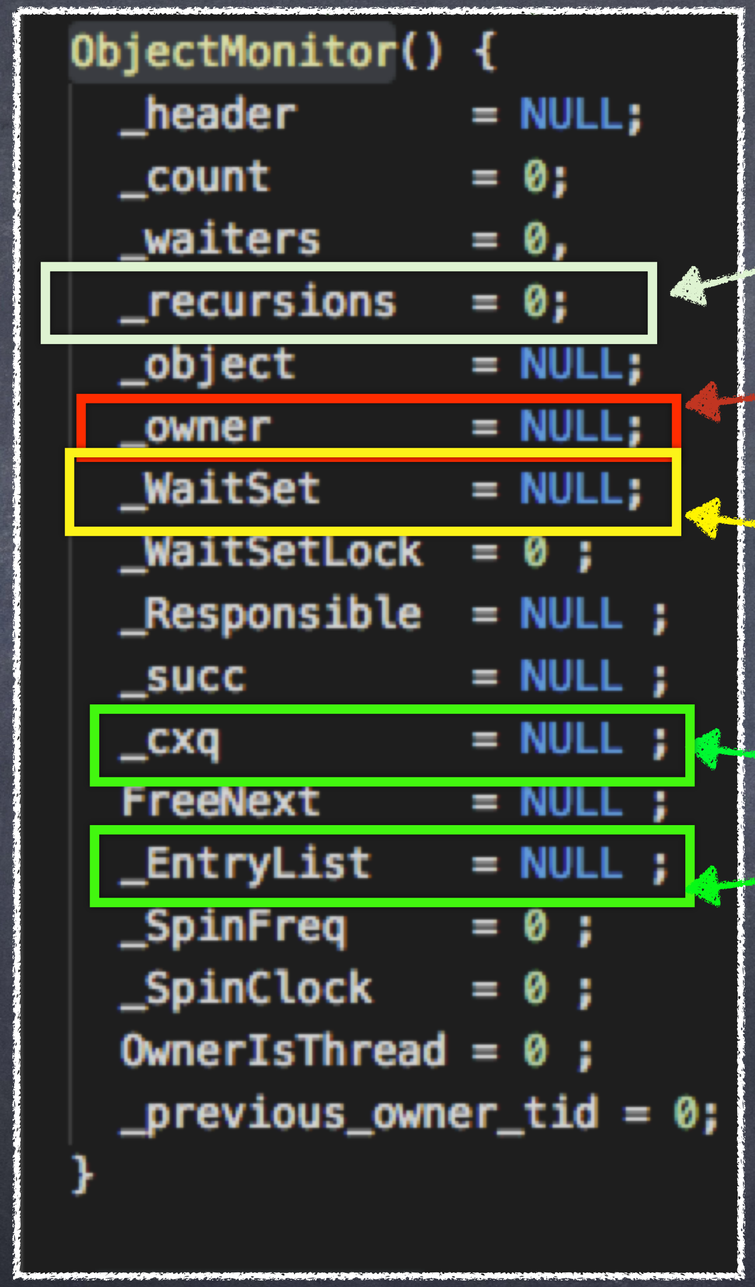
锁之里

Synchronized

```
public class SynTest {
        private int mockFlag=0;
        public  void synMock(){
            synchronized(this){
                mockFlag++;
            }
        }
}
```

```
public void synMock();
    Code:
      ..
       3: monitorenter
          4: aload_0
          5: dup
          6: getfield
          9: iconst_1
         10: iadd
         11: putfield
         14: aload_1
      15: monitorexit
         16: goto
         19: astore_2
         20: aload_1
      21: monitorexit
      ..
```

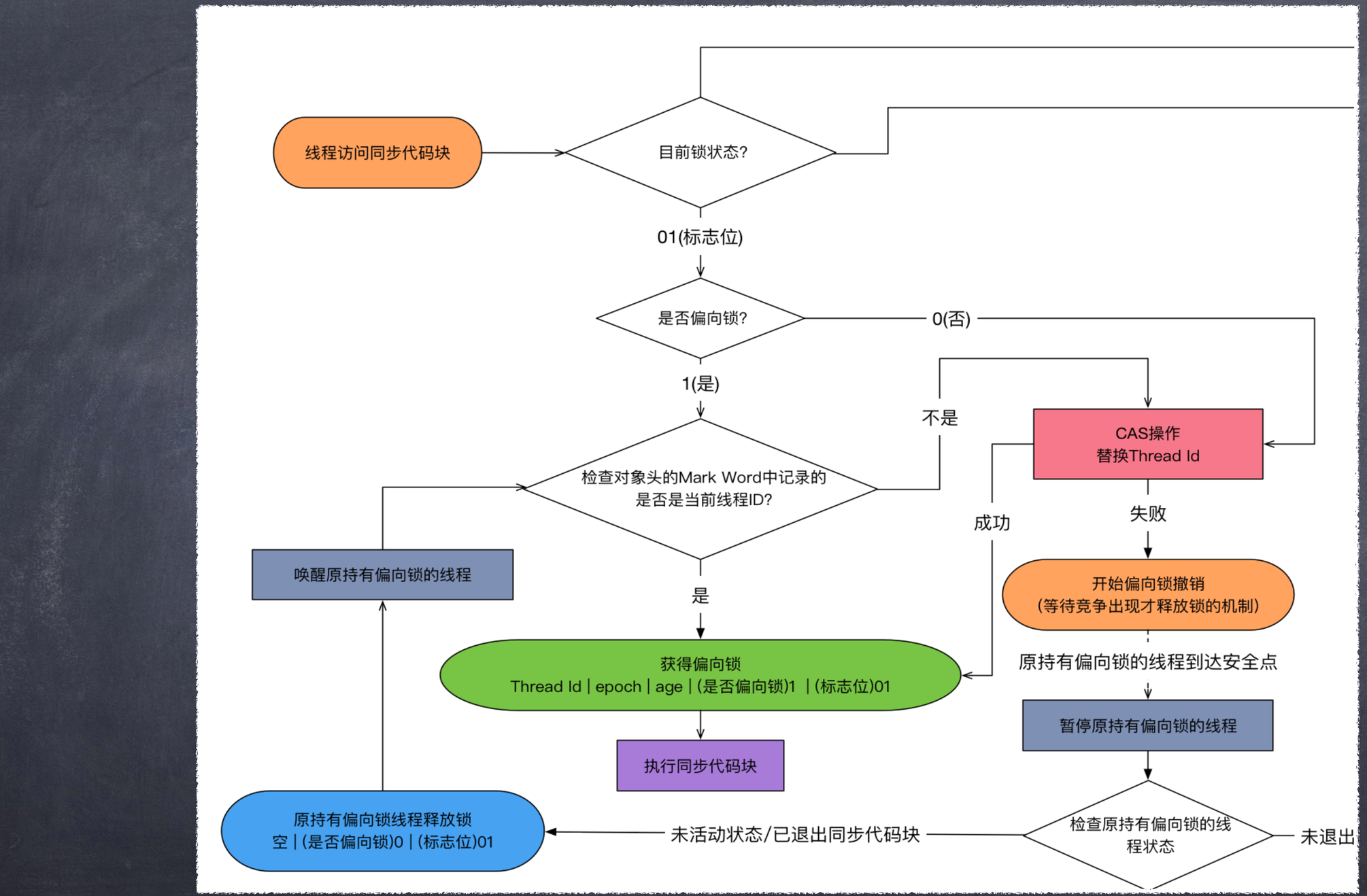
锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否偏向锁	锁标志位
无锁态	对象的hashCode		分代年龄	0	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量(重量级锁)的指针				10
GC标记	空				11
偏向锁	线程ID	Epoch	分代年龄	1	01


ObjectMonitor() {
_header = NULL;
_count = 0;
_waiters = 0,
_recursions = 0;
_object = NULL;
_owner = NULL;
_WaitSet = NULL;
_WaitSetLock = 0 ;
_Responsible = NULL ;
_succ = NULL ;
_cxq = NULL ;
FreeNext = NULL ;
_EntryList = NULL ;
_SpinFreq = 0 ;
_SpinClock = 0 ;
OwnerIsThread = 0 ;
_previous_owner_tid = 0;
}
重入次数
获取锁的线程
条件等待队列
等待队列


偏向锁
轻量级锁
重量级锁

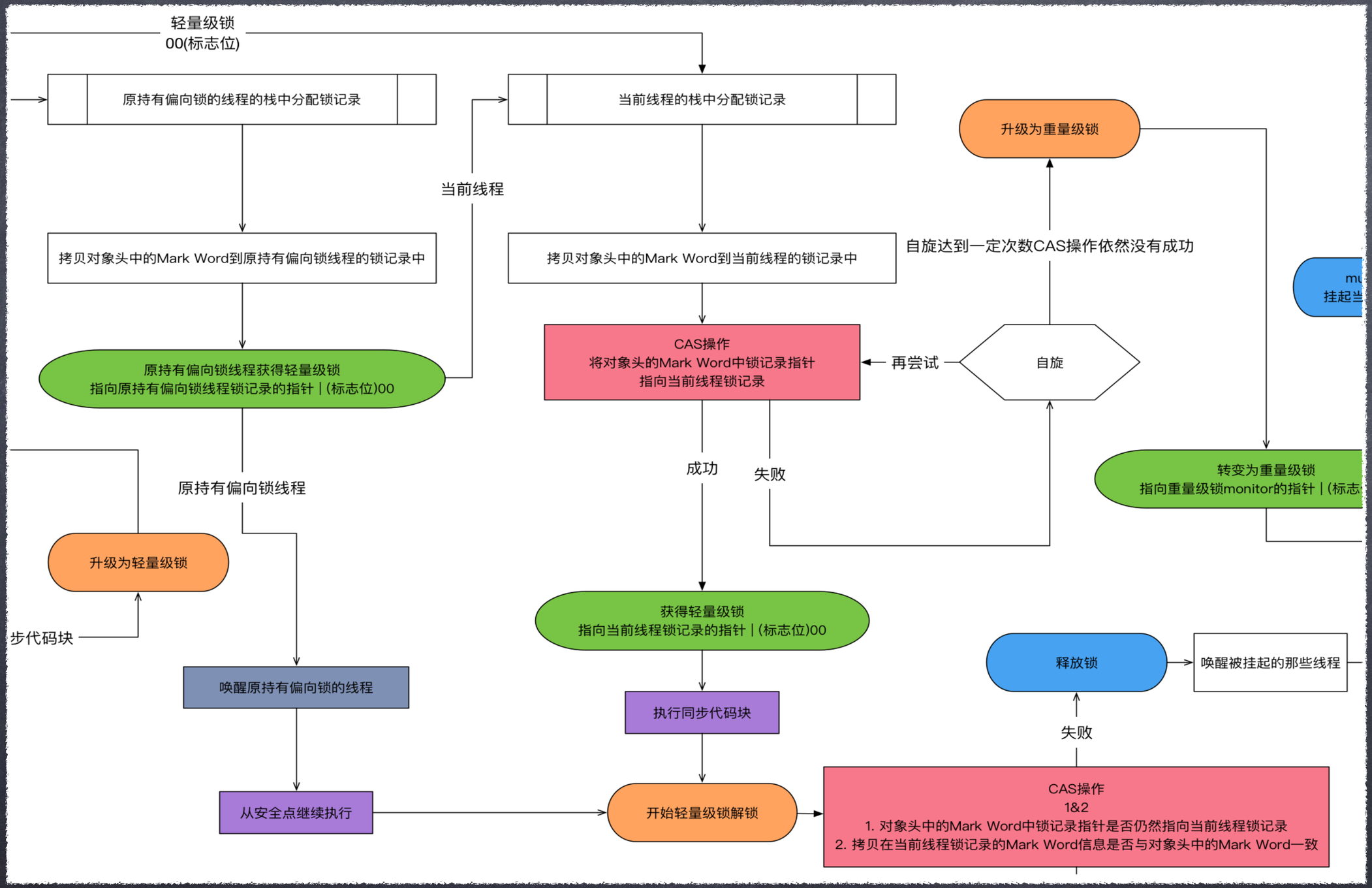
```
IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread* thread, BasicObjectLock* elem))
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
  if (PrintBiasedLockingStatistics) {
    Atomic::inc(BiasedLocking::slow_path_entry_count_addr());
  }
  Handle h_obj(thread, elem->obj());
  assert(Universe::heap()->is_in_reserved_or_null(h_obj()),
         "must be NULL or an object");
  if (UseBiasedLocking) {
    // Retry fast entry if bias is revoked to avoid unnecessary inflation
    ObjectSynchronizer::fast_enter(h_obj, elem->lock(), true, CHECK);
  } else {
    ObjectSynchronizer::slow_enter(h_obj, elem->lock(), CHECK);
  }
  assert(Universe::heap()->is_in_reserved_or_null(elem->obj()),
         "must be NULL or an object");
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
IRT_END
```

```
void ObjectSynchronizer::fast_enter(Handle obj, BasicLock* lock, bool attempt_rebias, TRAPS) {
 if (UseBiasedLocking) {
    if (!SafepointSynchronize::is_at_safepoint()) {
      BiasedLocking::Condition cond = BiasedLocking::revoke_and_rebias(obj, attempt_rebias, THREAD);
      if (cond == BiasedLocking::BIAS_REVOKED_AND_REBIASED) {
        return;
      }
    } else {
      assert(!attempt_rebias, "can not rebias toward VM thread");
      BiasedLocking::revoke_at_safepoint(obj);
    }
    assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
 }

 slow_enter (obj, lock, THREAD) ;
}
```


线程访问同步代码块
目前锁状态?
01(标志位)
是否偏向锁?
0(否)
1(是)
不是
检查对象头的Mark Word中记录的
是否是当前线程ID?
CAS操作
替换Thread Id
成功
失败
唤醒原持有偏向锁的线程
开始偏向锁撤销
(等待竞争出现才释放锁的机制)
是
获得偏向锁
Thread Id | epoch | age | (是否偏向锁)1 | (标志位)01
原持有偏向锁的线程到达安全点
暂停原持有偏向锁的线程
执行同步代码块
原持有偏向锁线程释放锁
空 | (是否偏向锁)0 | (标志位)01
未活动状态/已退出同步代码块
检查原持有偏向锁的线
程状态
未退出

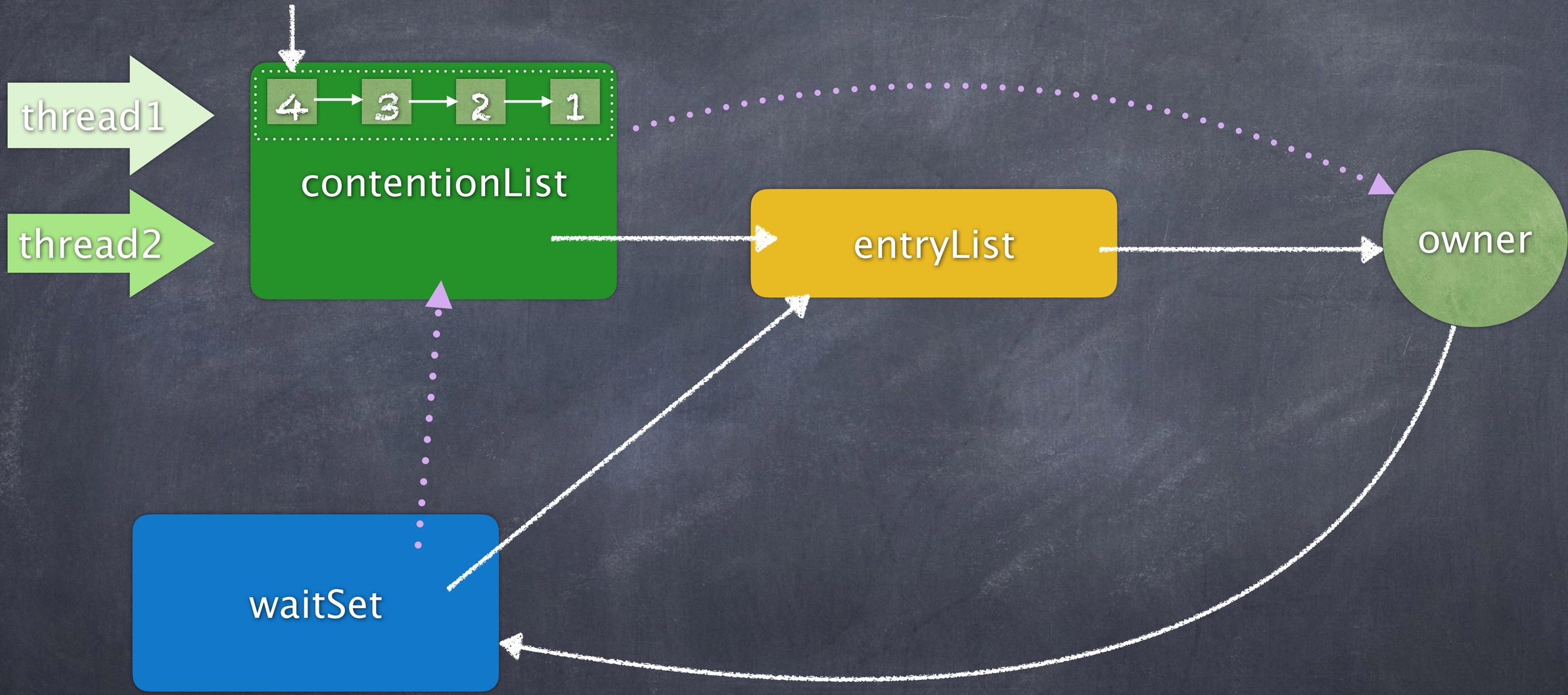
```
void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {
  markOop mark = obj->mark();
  assert(!mark->has_bias_pattern(), "should not see bias pattern here");

  if (mark->is_neutral()) {
    // Anticipate successful CAS -- the ST of the displaced mark must
    // be visible <= the ST performed by the CAS.
    lock->set_displaced_header(mark);
    if (mark == (markOop) Atomic::cmpxchg_ptr(lock, obj()->mark_addr(), mark)) {
      TEVENT (slow_enter: release stacklock) ;
      return ;
    }
    // Fall through to inflate() ...
  } else
  if (mark->has_locker() && THREAD->is_lock_owned((address)mark->locker())) {
    assert(lock != mark->locker(), "must not re-lock the same lock");
    assert(lock != (BasicLock*)obj->mark(), "don't relock with same BasicLock");
    lock->set_displaced_header(NULL);
    return;
  }
#if 0
#endif
  lock->set_displaced_header(markOopDesc::unused_mark());
  ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
}
```

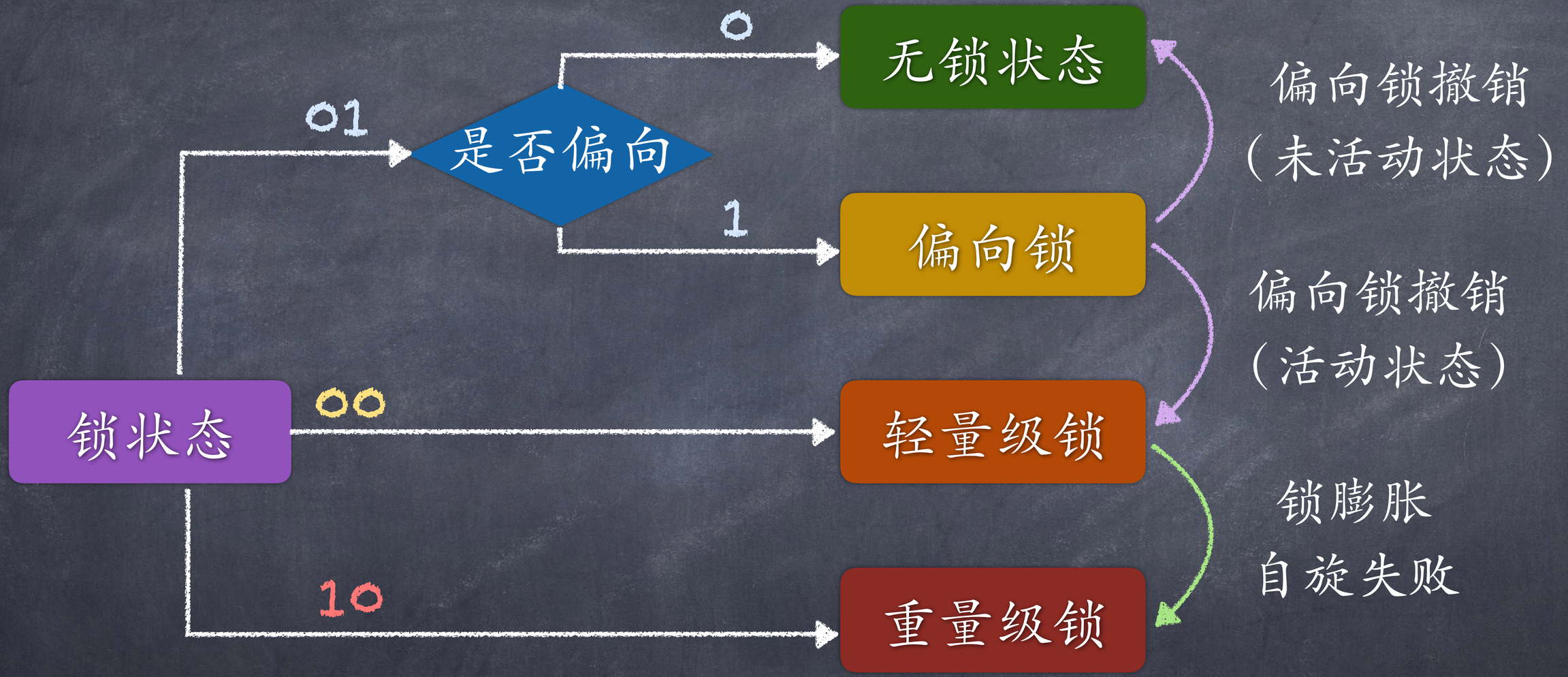
```
for (;;) {
    const markOop mark = object->mark() ;
    assert (!mark->has_bias_pattern(), "invariant") ;

    if (mark->has_monitor()) {
        ObjectMonitor * inf = mark->monitor() ;
        ...
        return inf ;
    }
    if (mark == markOopDesc::INFLATING()) {
       TEVENT (Inflate: spin while INFLATING) ;
       ReadStableMark(object) ;
       continue ;
    }
    if (mark->has_locker()) {
        ObjectMonitor * m = omAlloc (Self) ;
        ...
        markOop cmp = (markOop) Atomic::cmpxchg_ptr (markOopDesc::INFLATING(), object->mark_addr(), mark) ;
        ...
        markOop dmw = mark->displaced_mark_helper() ;
        assert (dmw->is_neutral(), "invariant") ;

        m->set_header(dmw) ;
        m->set_owner(mark->locker());
        m->set_object(object);
        guarantee (object->mark() == markOopDesc::INFLATING(), "invariant") ;
        ...

        return m ;
    }
```


轻量级锁
00(标志位)
原持有偏向锁的线程的栈中分配锁记录
当前线程的栈中分配锁记录
当前线程
升级为重量级锁
拷贝对象头中的Mark Word到原持有偏向锁线程的锁记录中
拷贝对象头中的Mark Word到当前线程的锁记录中
自旋达到一定次数CAS操作依然没有成功
CAS操作
将对象头的Mark Word中锁记录指针
指向当前线程锁记录
再尝试
自旋
原持有偏向锁线程获得轻量级锁
指向原持有偏向锁线程锁记录的指针 | (标志位)00
转变为重量级锁
指向重量级锁monitor的指针 | (标志
成功
失败
原持有偏向锁线程
升级为轻量级锁
步代码块
获得轻量级锁
指向当前线程锁记录的指针 | (标志位)00
释放锁
唤醒被挂起的那些线程
唤醒原持有偏向锁的线程
执行同步代码块
失败
从安全点继续执行
开始轻量级锁解锁
CAS操作
1&2
1. 对象头中的Mark Word中锁记录指针是否仍然指向当前线程锁记录
2. 拷贝在当前线程锁记录的Mark Word信息是否与对象头中的Mark Word一致

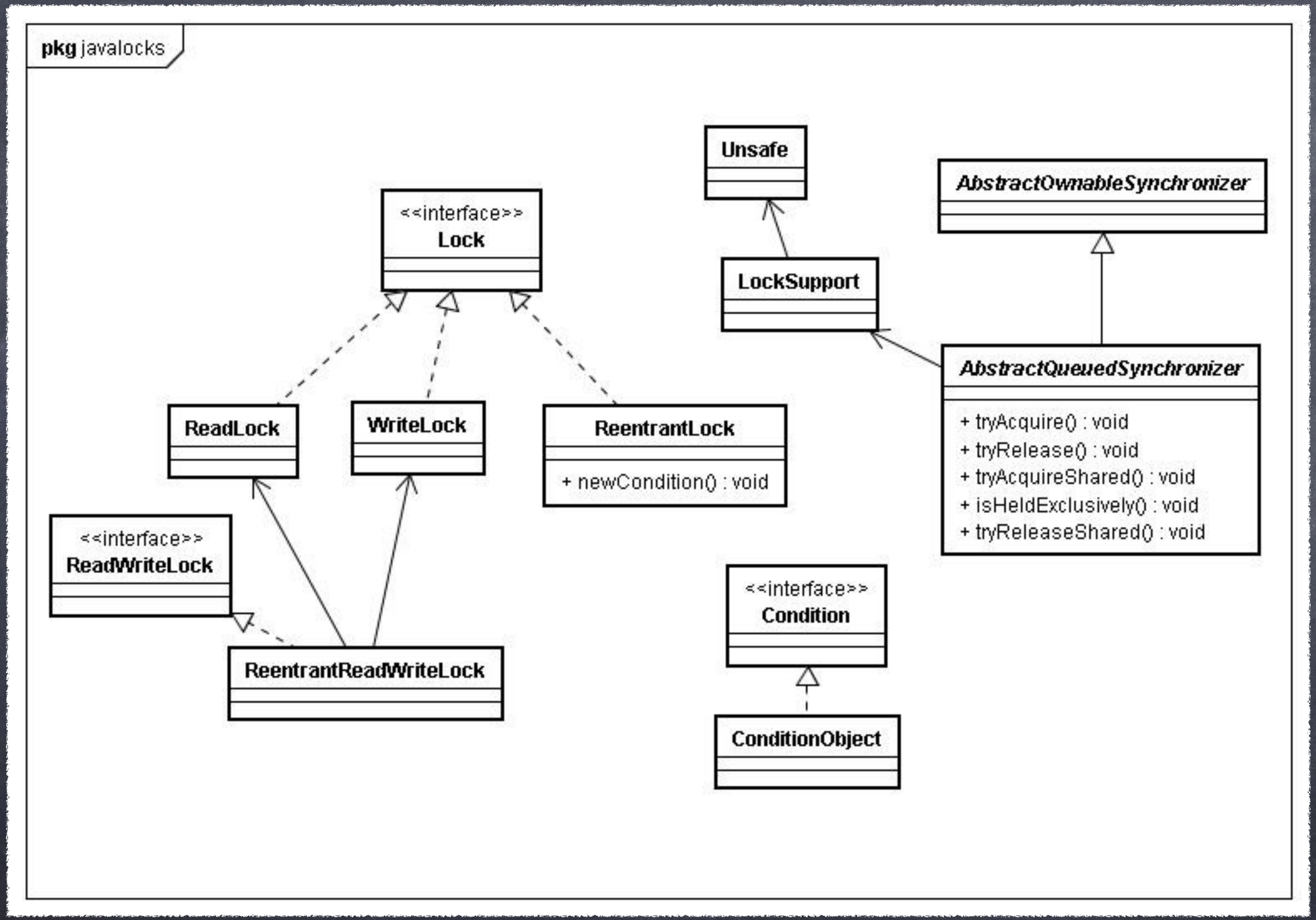
```
void ATTR ObjectMonitor::enter(TRAPS) {
  // The following code is ordered to check the most common cases first
  // and to reduce RTS->RTO cache line upgrades on SPARC and IA32 processors.
  Thread * const Self = THREAD ;
  void * cur ;

  cur = Atomic::cmpxchg_ptr (Self, &_owner, NULL) ;
  if (cur == NULL) {
     // Either ASSERT _recursions == 0 or explicitly set _recursions = 0.
     assert (_recursions == 0    , "invariant") ;
     assert (_owner      == Self, "invariant") ;
     // CONSIDER: set or assert OwnerIsThread == 1
     return ;
  }

  if (cur == Self) {
     // TODO-FIXME: check for integer overflow!  BUGID 6557169.
     _recursions ++ ;
     return ;
  }

  if (Self->is_lock_owned ((address)cur)) {
    assert (_recursions == 0, "internal state error");
    _recursions = 1 ;
    // Commute owner from a thread-specific on-stack BasicLockObject address to
    // a full-fledged "Thread *".
    _owner = Self ;
    OwnerIsThread = 1 ;
```

```
ObjectWaiter node(Self) ;
Self->_ParkEvent->reset() ;
node._prev   = (ObjectWaiter *) 0xBAD ;
node.TState  = ObjectWaiter::TS_CXQ ;

// Push "Self" onto the front of the _cxq.
// Once on cxq/EntryList, Self stays on-queue until it acquires the lock.
// Note that spinning tends to reduce the rate at which threads
// enqueue and dequeue on EntryList|cxq.
ObjectWaiter * nxt ;
for (;;) {
    node._next = nxt = _cxq ;
    if (Atomic::cmpxchg_ptr (&node, &_cxq, nxt) == nxt) break ;

    // Interference - the CAS failed because _cxq changed.  Just retry.
    // As an optional optimization we retry the lock.
    if (TryLock (Self) > 0) {
        assert (_succ != Self         , "invariant") ;
        assert (_owner == Self        , "invariant") ;
        assert (_Responsible != Self  , "invariant") ;
        return ;
    }
}
```

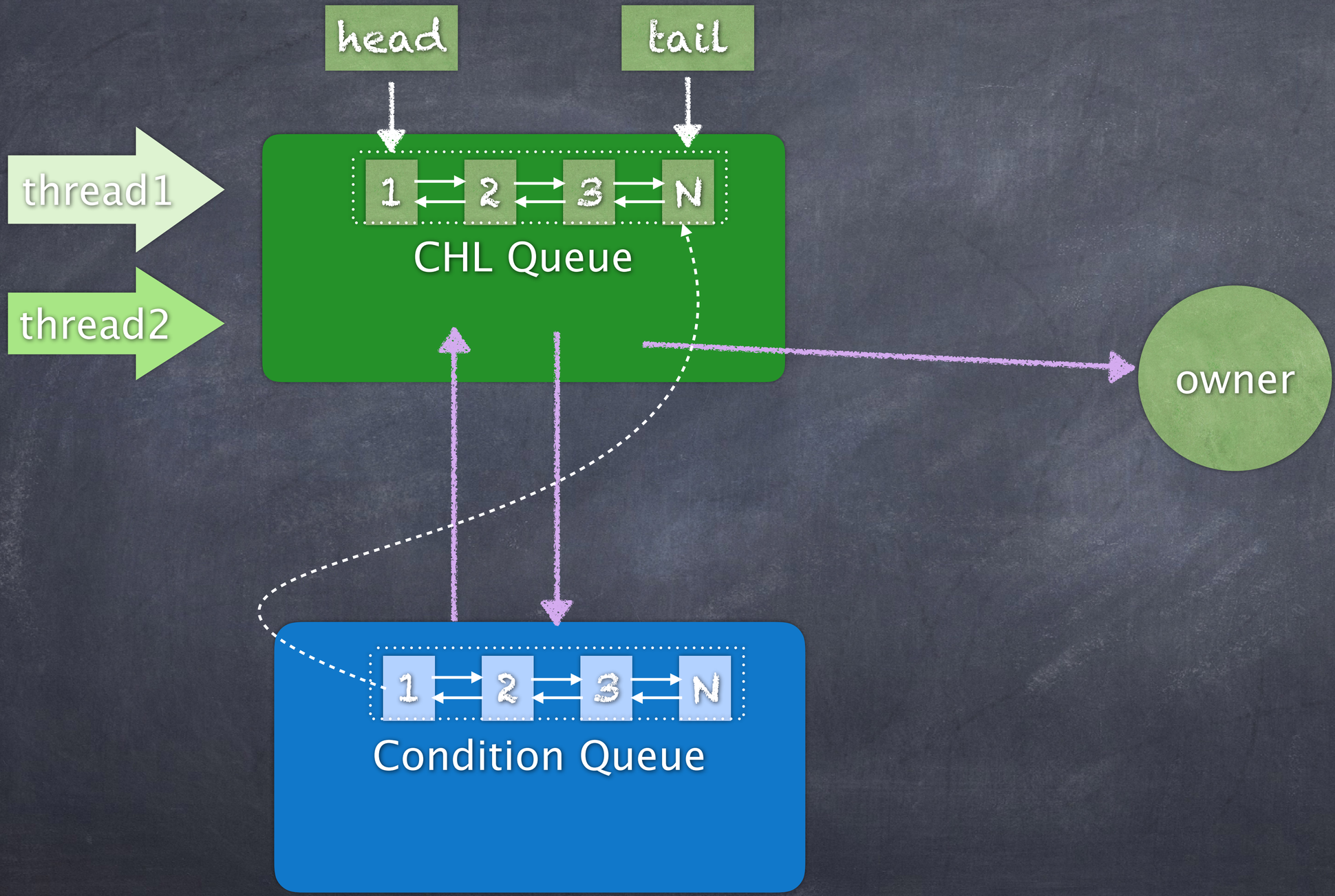

thread1
thread2
4
3
2
1
contentionList
entryList
owner
waitSet


0
01
是否偏向
1
无锁状态
偏向锁
轻量级锁
重量级锁
锁状态
00
10
偏向锁撤销
（未活动状态）
偏向锁撤销
（活动状态）
锁膨胀
自旋失败

Lock


pkg javalocks
Unsafe
AbstractOwnableSynchronizer
<<interface>>
Lock
LockSupport
AbstractQueuedSynchronizer
+ tryAcquire() : void
+ tryRelease() : void
+ tryAcquireShared() : void
+ isHeldExclusively() : void
+ tryReleaseShared() : void
ReadLock
WriteLock
ReentrantLock
+ newCondition() : void
<<interface>>
ReadWriteLock
<<interface>>
Condition
ReentrantReadWriteLock
ConditionObject

AbstractQueuedSynchronizer


head
tail
thread1
thread2
1
2
3
N
CHL Queue
owner
1
2
3
N
Condition Queue

isHeldExclusively

tryAcquire(int)

tryRelease(int)

tryAcquireShared(int)

tryReleaseShared(int)

ConditionObject

await()

awaitNanos(long)

signal()

signalAll()

LockSupport

```
class Parker : public os::PlatformParker {
private:
  volatile int _counter ;
  ...
public:
  void park(bool isAbsolute, jlong time);
  void unpark();
  ...
}
class PlatformParker : public CHeapObj<mtInternal> {
  protected:
    pthread_mutex_t _mutex [1] ;
    pthread_cond_t  _cond  [1] ;
    ...
}
```

```
void Parker::park(bool isAbsolute, jlong time) {

  if (Atomic::xchg(0, &_counter) > 0) return;

  Thread* thread = Thread::current();
  assert(thread->is_Java_thread(), "Must be JavaThread");
  JavaThread *jt = (JavaThread *)thread;

  if (Thread::is_interrupted(thread, false)) {
    return;
  }

  timespec absTime;
  if (time < 0 || (isAbsolute && time == 0) ) { // don't wait at all
    return;
  }
  if (time > 0) {
    unpackTime(&absTime, isAbsolute, time);
  }

  ThreadBlockInVM tbivm(jt);
  if (Thread::is_interrupted(thread, false) || pthread_mutex_trylock(_mutex) != 0) {
    return;
  }
  int status ;
  if (_counter > 0)  { // no wait needed
    _counter = 0;
    status = pthread_mutex_unlock(_mutex);
    assert (status == 0, "invariant") ;
    OrderAccess::fence();
    return;
  }
```

```
void Parker::unpark() {
  int s, status ;
  status = pthread_mutex_lock(_mutex);
  assert (status == 0, "invariant") ;
  s = _counter;
  _counter = 1;
  if (s < 1) {
    // thread might be parked
    if (_cur_index != -1) {
      // thread is definitely parked
      if (WorkAroundNPTLTimedWaitHang) {
        status = pthread_cond_signal (&_cond[_cur_index]);
        assert (status == 0, "invariant");
        status = pthread_mutex_unlock(_mutex);
        assert (status == 0, "invariant");
      } else {
        // must capture correct index before unlocking
        int index = _cur_index;
        status = pthread_mutex_unlock(_mutex);
        assert (status == 0, "invariant");
        status = pthread_cond_signal (&_cond[index]);
        assert (status == 0, "invariant");
      }
    } else {
      pthread_mutex_unlock(_mutex);
      assert (status == 0, "invariant") ;
    }
  } else {
    pthread_mutex_unlock(_mutex);
    assert (status == 0, "invariant") ;
  }
}
```

pthread_cond_wait

pthread_cond_timedwait

pthread_mutex_lock

pthread_mutex_unlock

pthread_cond_signal

CAS(Unsafe.java)

```
inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*     dest, jint     c
  // alternative for InterlockedCompareExchange
  int mp = os::is_MP();
  __asm {
    mov edx, dest
    mov ecx, exchange_value
    mov eax, compare_value
    LOCK_IF_MP(mp)
    cmpxchg dword ptr [edx], ecx
  }
}
```

ABA

循环时间长开销大

只能保证一个共享变量的原子操作

减少锁持有时间

减小锁粒度

锁分离

锁粗化

Q&A

Thank you