JVM 锁 bug 导致 G1 GC 挂起问题分析和解决

原创 宋尧飞 openEuler 7月13日

收录于话题
#毕昇 JDK 技术剖析

4个

> 编者按：笔者在 AArch64 中遇到一个 G1 GC 挂起，CPU 利用率高达 300%的案例。经过分析发现问题是由 JVM 的锁机制导致，该问题根因是并发编程中没有正确理解内存序导致。本文着重介绍 JVM 中 Monitor 的基本原理，同时演示了在什么情况下会触发该问题。希望通过本文的分析，读者能够了解到内存序对性能、正确性的影响，在并发编程时更加仔细。

现象

本案例是一个典型的弱内存模型案例，大致的现象就是 AArch64 平台上，业务挂死，而进程占用 CPU 持续维持在 300%。配合 top 和 gdb，可以看到是 3 个 GC 线程在 offer_termination 处陷入了死循环：

```
$ gdb -batch -ex bt -p 4427
#0  in ParallelTaskTerminator::offer_termination(TerminatorTerminator*)
#1  in G1ParEvacuateFollowersClosure::do_void()
#2  in G1ParTask::work(unsigned int)
#3   in GangWorker::loop()
#4  in java_start(Thread*)
#5  in start_thread () from /lib64/libpthread.so.0
#6  in thread_start () from /lib64/libc.so.6

$ gdb -batch -ex bt -p 4429
#0  in ParallelTaskTerminator::offer_termination(TerminatorTerminator*)
#1  in G1ParEvacuateFollowersClosure::do_void()
#2  in G1ParTask::work(unsigned int)
#3   in GangWorker::loop()
#4  in java_start(Thread*)
#5  in start_thread () from /lib64/libpthread.so.0
#6  in thread_start () from /lib64/libc.so.6

$ gdb -batch -ex bt -p 4433
#0  in SpinPause ()
#1  in ParallelTaskTerminator::offer_termination(TerminatorTerminator*)
#2  in G1ParEvacuateFollowersClosure::do_void() ()
#3  in G1ParTask::work(unsigned int)
#4  in GangWorker::loop()
#5  in java_start(Thread*)
#6  in start_thread () from /lib64/libpthread.so.0
#7  in thread_start () from /lib64/libc.so.6
```

多个并行 GC 线程在 Minor GC 结束时调用 offer_termination，在 offer_termination 中自旋等待其他并行 GC 线程到达该位置，才说明 GC 任务完成，可以终止。（关于并行任务的中止协议问

题，可以参考相关论文，这里不做着重介绍。

简单地说，在并行任务执行时，多个任务之间可能存在任务不均衡，所以 JVM 内部设计了任务均衡机制，同时必须设计任务终止的机制来保证多个任务都能完成，这里的 offer_termination 就是尝试终止任务）。

在该案例中，部分 GC 线程完成自己的任务，等待其他的 GC 线程。此时出现挂起，很有可能是因为发生了死锁。所以问题很可能是由于那些尚未完成任务的 GC 线程上错误地使用锁。所以使用 gdb 观察了一下其他 GC 线程，发现其他 GC 线程全都阻塞在一把 JVM 的锁上：

```
$ gdb -batch -ex bt -p 4430
#0 in pthread_cond_wait()
#1 in os::PlatformEvent::park()
#2 in Monitor::ILock(Thread *)
#3 in Monitor::lock(Thread *)
#4 in G1ParGCAllocator::allocate_direct_or_new_plab
#5 in G1ParScanThreadState::copy_to_survivor_space
...
```

而这把 Monitor 中的情况如下：

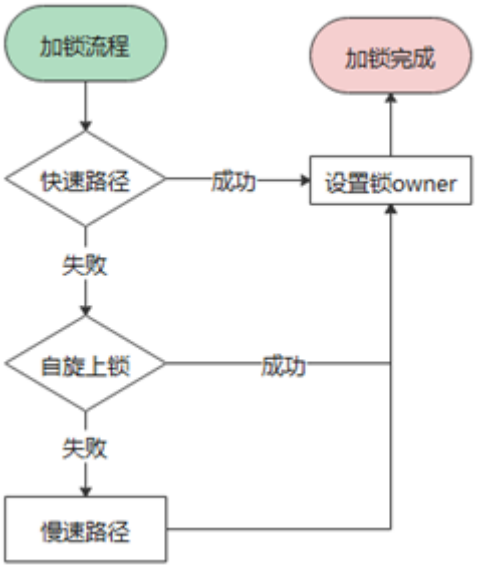
- cxq 上积累了大量 GC 线程
- OnDeck 记录的 GC 线程已经消失
- _owner 记录的锁持有者为 NULL

分析

在进一步分析前，首先普及一下 JVM 锁组件 Monitor 的基本原理，Monitor 类主要包含 4 个核心字段：

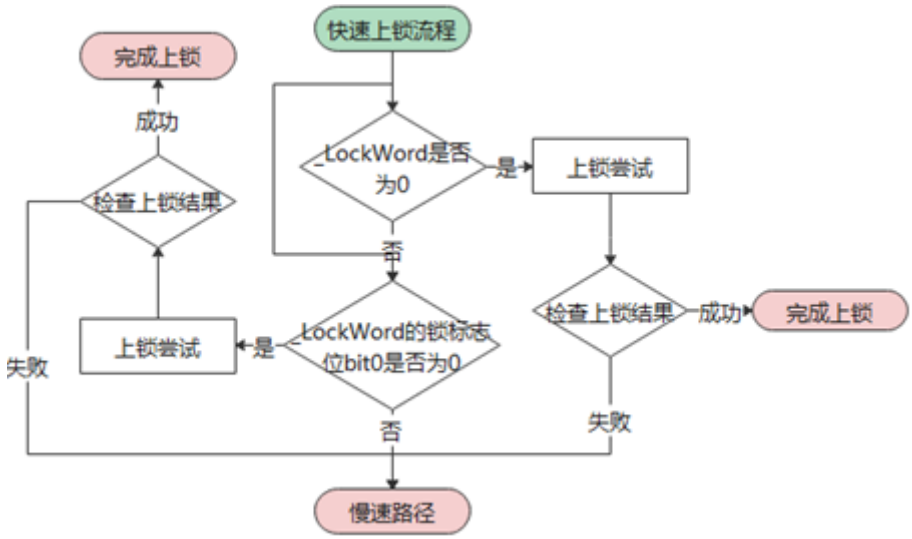
1. “Thread * volatile _owner” 字段指向这把锁的持有线程
2. “SplitWord_LockWord” 字段被设计为 1 个机器字长，目的是为了确保操作时天然的原子性，它的最低位被设计为上锁标记位，而高位区域用来存放 256 字节对齐的竞争队列(cxq)地址
3. “ParkEvent * volatile_EntryList” 字段指向一个等待队列，跟 cxq 差别不大，个人理解只是为了缓解 cxq 的竞争压力而设计
4. “ParkEvent * volatile_OnDeck” 字段指向这把锁的法定继承人，同时最低位还充当了内部锁的角色

接下来通过一组流程图来介绍加解锁的具体流程：

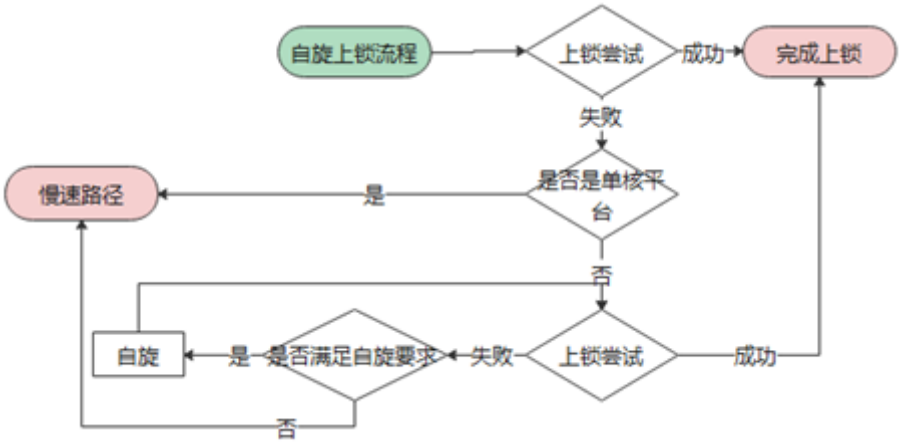


上图是加锁的一个整体流程，大致分为 3 步：

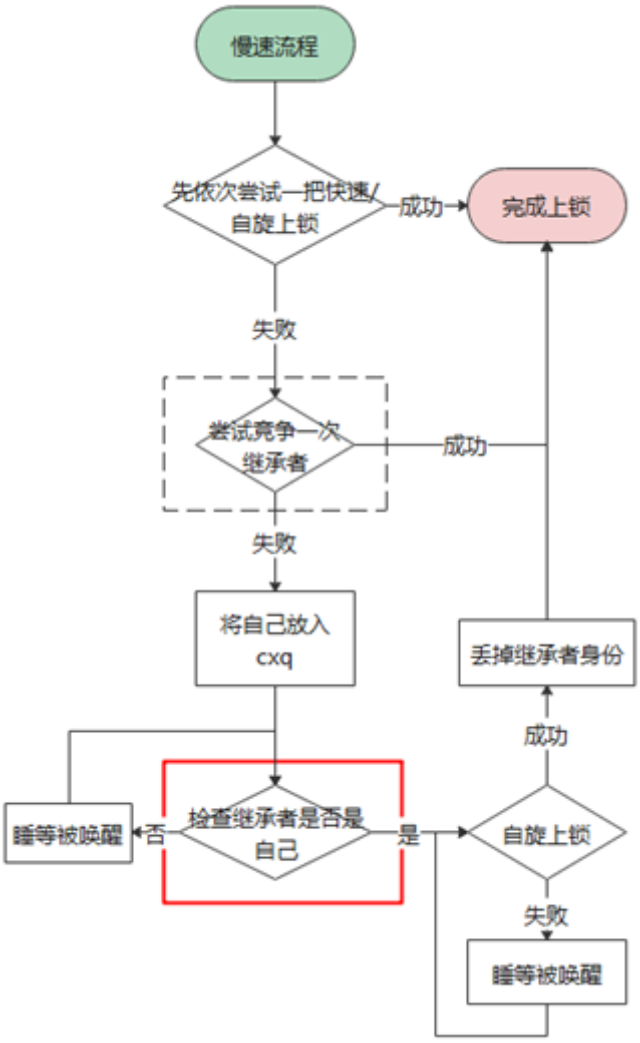
首先走快速上锁流程，主要对应锁本身无人持有的最理想情况



接着是自旋上锁流程，这是预期将在短时间内获取锁的情况



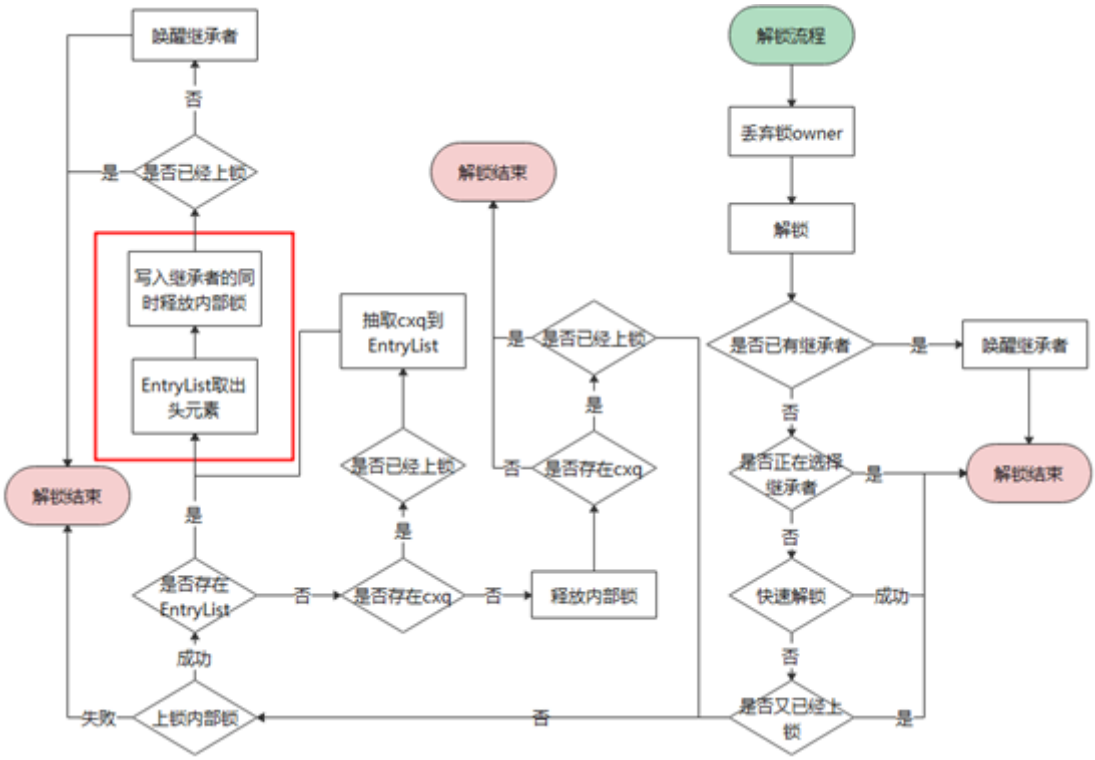
最后是慢速上锁流程，申请者将会加入等待队列(cxq)，然后进入睡眠，直到被唤醒后发现自己变成了法定继承者，于是进入自旋，直到完成上锁。



而且，基于性能考虑，整个上锁流程中的每一步几乎都做了“插队”的尝试：

```
intptr_t v = _LockWord.FullWord ;
for (;;) {
    if ((v & _LBIT) != 0) return 0 ;
    const intptr_t u = CASPTR (&_LockWord, v, v|_LBIT) ;
    if (v == u) return 1 ;
    v = u ;
}
```

如上图代码中所示，“插队”的意思就是不经过排队(cxq)，直接尝试置上锁标志位。



上图就是整个解锁流程了，显然真正的解锁操作在第二步中就已经完成了(意味着接下来时刻有“插队”现象发生)，剩下的主要就是选出继承者的过程，大致分为以下几步：

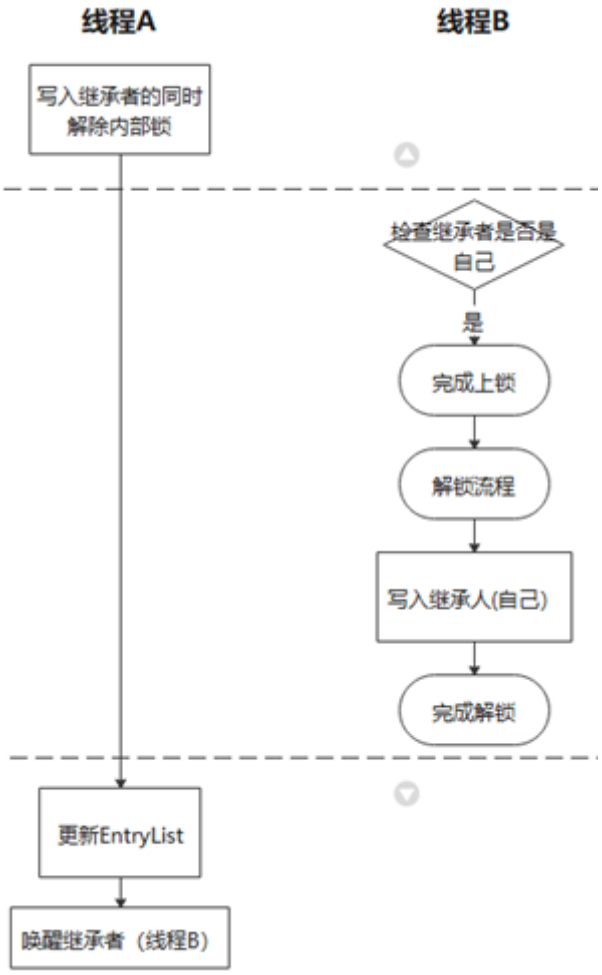
1. 解锁线程首先需要将内部锁(_OnDeck)标记上锁
2. 从竞争队列(cxq)抽取所有等待者放入等待队列(_EntryList)
3. _ EntryList 取出头一个元素，写入_OnDeck 的同时解除内部锁标记，这代表选出了继承者
4. 唤醒继承者

当然伴随着整个解锁流程每一步的，还有对“插队”行为的处理。

至此，JVM 锁组件 Monitor 的原理就介绍到这里，再回归到问题本身，一个疑问就是_OnDeck 上记录的继承者为何消失？作为继承者，既然已经消失在竞争队列和等待队列里，显然意味着它大概率已经持有锁、然后解锁走人了，所以问题很可能跟继承者选取过程有关。基于这种猜测，我们对相关代码着重进行了梳理，就发现了下图两处红框标记位置存在疑点，那就是在选继承者过程第 3 步中：

```
WakeOne:
 assert (List == _EntryList, "invariant") ;
 ParkEvent * const w = List ;
 assert (RelaxAssert || w != Thread::current()->_MutexEvent, "invariant") ;
 _EntryList = w->ListNext ;
 // as a diagnostic measure consider setting w->_ListNext = BAD
 assert (UNS(_OnDeck) == _LBIT, "invariant") ;
 _OnDeck = w ;           // pass OnDeck to w.
                         // w will clear OnDeck once it acquires the outer lock
```

写**EntryList 和写_OnDeck 之间没有 barrier 来保证执行顺序，这可能出现_OnDeck 先于** EntryList 写入的情况，一旦继承人提前持有锁，后果就可能非常糟糕...



这里贴了一张可能的问题场景：

1. 线程 A 处于解锁流程中，由于乱序，先写入了继承者同时解除内部锁
2. 线程 B 处于上锁流程，发现自己就是法定继承者后，立刻完成上锁
3. 线程 B 又迅速进入解锁流程，并从_EntryList 中取出头元素(也就是线程 B!)作为继承者写入_OnDeck，完成解锁走人
4. 线程 A 此时才更新_EntryList，然后唤醒继承者(也就是线程 B!)，完成解锁走人
5. _OnDeck 上的继承者线程 B，实际已经完成加解锁离开，后续等待线程再也无法被唤醒。

正巧在社区的高版本上找到了一个相关的修复记录，这里贴出 2 个关键的代码片段：

```
    // At any given time there is at most one ondeck thread.
    // ondeck implies not resident on cxq and not resident on EntryList
-   // Only the OnDeck thread can try to acquire -- contended for -- the lock.
+   // Only the OnDeck thread can try to acquire -- contend for -- the lock.
    // CONSIDER: use Self->OnDeck instead of m->OnDeck.
    // Deschedule Self so that others may run.
-   while (_OnDeck != ESelf) {
+   while (OrderAccess::load_ptr_acquire(&_OnDeck) != ESelf) {
      ParkCommon (ESelf, 0) ;
    }
```

上面这段代码位于慢速上锁流程，被唤醒后检查继承者是否是自己，修复后的代码在读_OnDeck 时加了 Load-Acquire 的 barrier。

```
    _EntryList = w->ListNext ;
    // as a diagnostic measure consider setting w->_ListNext = BAD
    assert (UNS( OnDeck) == _LBIT, "invariant") ;
-   _OnDeck = w ;           // pass OnDeck to w.
-                           // w will clear OnDeck once it acquires the outer lock
+
+   // Pass OnDeck role to w, ensuring that _EntryList has been set first.
+   // w will clear _OnDeck once it acquires the outer lock.
+   // Note that once we set _OnDeck that thread can acquire the mutex, proceed
+   // with its critical section and then enter this code to unlock the mutex. So
+   // you can have multiple threads active in IUnlock at the same time.
+   OrderAccess::release_store_ptr(&_OnDeck, w);
```

上面这段代码位于解锁时选继承者流程，从_ EntryList 取出头一个元素，写入_OnDeck 的同时解除内部锁标记，修复后的代码在写_OnDeck 时加了 Store-Release 的 barrier。

显然，围绕_OnDeck 添加的这对 One-way barrier 可以确保：当继承者线程被唤醒时，该线程可以“看”到_EntryList 已经被及时更新。

总结：

在 AArch64 这种弱内存模型的平台上（关于内存序更多的知识在接下来的分享中会详细介绍），一旦涉及多线程对公共内存的每一次访问，必须反复确认是否需要通过 barrier 来严格保序，而且除非存在有效的依赖关系，否则 barrier 需要在读写端成对使用。

后记

如果遇到相关技术问题（包括不限于毕昇 JDK），可以通过毕昇 JDK 社区求助（目前毕昇 JDK 最新的官网http://bishengjdk.openeuler.org已经上线，可以点击原文进入官网查找所有相关资源，包括二进制下载、代码仓库、使用教学、安装、学习资料等）。

毕昇JDK社区每双周周二举行技术例会，同时有一个技术交流群讨论GCC、LLVM、JDK和V8等相关编译技术，感兴趣的同学可以添加如下微信小助手，回复Compiler入群。

收录于话题 #毕昇 JDK 技术剖析·4个

上一篇

使用 perf 解决 JDK8 小版本升级后性能下降的问题

下一篇

Java Flight Recorder - 事件机制详解

阅读原文 文章已于2021/07/13修改