Native Monitors Design

author: Patricio Chilano Mateo (pchilanomate)

Introduction

The current design of native monitors uses a technique that we name "sneaky locking" to prevent possible deadlocks of the JVM during safepoints. The implementation of this technique though introduces a race when a monitor is shared between the VMThread and non-JavaThreads. Also the current design uses a lot of optimization techniques to try to improve performance making the code more complex. Maybe we can simplify the code without performance penalties. The goal of this proposal then is to fix this sneaky scheme by redesigning the implementation of native monitors in a way that also simplifies the code and keeps a comparable performance to what we have today.

Recap: Sneaky Locking

Sneaky locking prevents a deadlock scenario that can happen if, during a safepoint, the VMThread tries to acquire a monitor whose outer lock (actual lockword) has been acquired by a JavaThread that is blocked waiting for the safepoint to finish. Below is the skeleton of method Monitor::lock():

```
void Monitor::lock(Thread * Self) {
 if (TryLock()) {
   set_owner(Self);
   return;
 }
 if (Self->is_Java_thread()) {
   ThreadBlockInVM tbivm((JavaThread *) Self);
   ILock(Self);
 } else {
   ILock(Self);
 }
 set_owner(Self);
}

Here is a similar use of ThreadBlockInVM in Monitor:wait()

bool Monitor::wait() {
 set_owner(NULL);
 if (no_safepoint_check) {
   wait_status = IWait(Self, timeout);
 } else {
   ThreadBlockInVM tbivm(jt);
   wait_status = IWait(Self, timeout);
 }
 set_owner(Self);
}
```

The state transition from _thread_in_vm to _thread_blocked performed in the ThreadBlockInVM constructor is needed if we want to allow safepoints to progress because the ILock() call can potentially make blocking calls to the OS. There is also a SafepointMechanism::block_if_requested() call executed in both the constructor and destructor of the ThreadBlockInVM object. Why is it needed in the destructor? Because if there is a safepoint currently going on after the ILock() call returns the JavaThread has to be stopped and cannot keep executing VM code.
Because of this desire to check for safepoints when acquiring native monitors we now have possible deadlocking issues if a monitor is shared between JavaThreads and the VMThread. Hence the following code was added in Monitor::lock implementing "sneaky locking":

```
bool can_sneak = Self->is_VM_thread() && SafepointSynchronize::is_at_safepoint();
if (can_sneak && _owner == NULL) {
 _snuck = true;
 set_owner(Self);
}
```

As we mentioned before this code introduces a race if non-JavaThreads also try to acquire the monitor, since the non-JavaThread could have locked the monitor but not yet set itself as the owner. (Using Monitor::try_lock() has the same race). By non-JavaThreads we mean internal VM threads like ConcurrentGCThreads, WorkerThreads, WatcherThread, JfrThreadSampler, etc.
It's worth mentioning though that we could keep the current design of native monitors and modify the sneaking code part to actually identify which kind of thread has the outer lock before sneaking (we could use the second bit of the lockword for example).

Design Proposal

Platform Monitor

The proposal is based on the introduction of the new class PlatformMonitor, which serves as a wrapper for the actual synchronization primitives in each platform(mutexes and condition variables) and provides the basic API for the implementation of native monitors. Here is the definition of class PlatformMonitor (for posix OSes; for solaris and windows there is an equivalent definition):

```
// Platform specific implementation that underpins VM Monitor/Mutex class
class PlatformMonitor : public CHeapObj<mtInternal> {
 private:
   pthread_mutex_t _mutex[1]; // Native mutex for locking
   pthread_cond_t _cond[1]; // Native condition variable for blocking
 public:
   PlatformMonitor();
   void lock();
   void unlock();
   bool try_lock();
   int wait(jlong millis);
   void notify();
   void notify_all();
};
```

As an example we can see that PlatformMonitor::lock() is a wrapper for pthread_mutex_lock() and that PlatformMonitor::wait() just uses the underlying pthread_cond_timedwait() call parsing the waiting time first:

```
void os::PlatformMonitor::lock() {
 int status = pthread_mutex_lock(_mutex);
 assert_status(status == 0, status, "mutex_lock");
}
// Must already be locked
int os::PlatformMonitor::wait(jlong millis) {
 assert(millis >= 0, "negative timeout");
 if (millis > 0) {
   struct timespec abst;
   // We have to watch for overflow when converting millis to nanos,
   // but if millis is that large then we will end up limiting to
   // MAX_SECS anyway, so just do that here.
   if (millis / MILLIUNITS > MAX_SECS) {
     millis = jlong(MAX_SECS) * MILLIUNITS;
   }
   to_abstime(&abst, millis * (NANOUNITS / MILLIUNITS), false);
   int ret = OS_TIMEOUT;
   int status = pthread_cond_timedwait(_cond, _mutex, &abst);
   assert_status(status == 0 || status == ETIMEDOUT, status, "cond_timedwait");
   if (status == 0) {
     ret = OS_OK;
   }
   return ret;
 } else {
   int status = pthread_cond_wait(_cond, _mutex);
   assert_status(status == 0, status, "cond_wait");
   return OS_OK;
 }
}
```

Instead of introducing our own custom synchronization schemes on top of native primitives as we do today, we will use this new class to implement our native monitors. Most of the API calls can thus be implemented as simple wrappers around PlatformMonitor, adding more assertions and very little extra housekeeping(owner and monitor name). The exception will be the lock() and wait() calls since we have to add all the logic for handling the safepoint checking mechanism.

Overall Design Strategy

As mentioned before, the idea for implementing native monitors is to mostly rely on the simple underlying PlatformMonitor class. However, to be able to remove the lock sneaking code and at the same time avoid deadlocking scenarios, we will have to be able to solve these two problematic scenarios:

- In Monitor::lock() and Monitor::wait(), blocking for a safepoint in the ThreadBlockInVM destructor could lead to a deadlock, since the monitor could be shared with the VMThread.

-In Monitor::wait(), blocking for a safepoint in the ThreadBlockInVM constructor could lead to a deadlock, since we already own the monitor and as before the monitor could be shared with the VMThread. Doing a plain release of the lock prior to entering the constructor is not an option since we could miss a notify() call.

To handle the first scenario we will make the JavaThread that has just acquired the lock but detects there is a safepoint request in the ThreadBlockInVM destructor release the lock before blocking at the safepoint. After resuming from it, the JavaThread will have to acquire the lock again as it has just called Monitor::lock(). For the second scenario we will change the ThreadBlockInVM constructor to allow for a possible safepoint request to make progress but without letting the JavaThread block for it. In order to avoid adding extra conditionals to the current ThreadBlockInVM jacket to implement this functionality, we will actually create and use a new jacket instead, ThreadBlockInVMWithDeadlockCheck.

Implementation of Monitor::lock()

Here is the implementation of Monitor::lock() (without extra checks and assertions):

```
void Monitor::lock(Thread * self) {
 Monitor* in_flight_monitor = NULL;
 while (!_lock.try_lock()) {
   // The lock is contended
   if (self->is_Java_thread()) {
     ThreadBlockInVMWithDeadlockCheck tbivmdc((JavaThread *) self, &in_flight_monitor);
     in_flight_monitor = this;
     _lock.lock();
   }
   if (in_flight_monitor != NULL) {
     break;
   } else {
     _lock.lock();
    break;
   }
 }
 set_owner(self);
}
```

The locking path for non-JavaThreads and the VMThread is straightforward. For JavaThreads whenever we return from the TBIVWDC jacket without having stopped at a safepoint we will just set ourselves as the owners and return. If we did stop at a safepoint, then the in_flight_monitor local variable will be NULL, so we will loop again trying to acquire the monitor. The code that sets in_flight_monitor to NULL and releases the monitor is shown below. It is called from the TBIVWDC destructor in case we need to block for a safepoint or handshake:

```
void release_monitor() {
 Monitor* in_flight_monitor = *_in_flight_monitor_adr;
 if (in_flight_monitor != NULL) {
   in_flight_monitor->release_for_safepoint();
   *_in_flight_monitor_adr = NULL;
 }
}
```

where release_for_safepoint() just releases the lock.

Implementation of Monitor::wait()

Here is the implementation of Monitor::wait() (without extra checks and assertions):

```
bool Monitor::wait(bool no_safepoint_check, long timeout, bool as_suspend_equivalent) {
 int wait_status;
 set_owner(NULL);
 if (no_safepoint_check) {
   wait_status = _lock.wait(timeout);
   set_owner(self);
 } else {
   JavaThread *jt = (JavaThread *)self;
   Monitor* in_flight_monitor = NULL;
   {
     ThreadBlockInVMWithDeadlockCheck tbivmdc(jt, &in_flight_monitor);
     OSThreadWaitState osts(self->osthread(), false /* not Object.wait() */);
     if (as_suspend_equivalent) {
       jt->set_suspend_equivalent();
     }
     wait_status = _lock.wait(timeout);
     in_flight_monitor = this;
     if (as_suspend_equivalent && jt->handle_special_suspend_equivalent_condition()) {
       _lock.unlock();
       jt->java_suspend_self();
       _lock.lock();
     }
   }
   if (in_flight_monitor != NULL) {
     set_owner(self);
   } else {
     lock(self);
   }
 }
 return wait_status != 0; // return true IFF timeout
}
```

The use of the in_flight_monitor local variable is analogous to the lock() case. In the TBIVWDC constructor we will just callback if there is a pending safepoint but the JavaThread will not block. To do that we added a new bool parameter to SS::block, block_in_safepoint_check, which defaults to true. If this value is true we execute as we do now in Safepoint::block(). If we call SS::block with a false value we allow for the safepoint to make progress but never try to grab the Threads_lock. Here is the abbreviated logic of Safepoint::block():

```
Safepoint_lock->lock_without_safepoint_check();
if (is_synchronizing()) {
 //make changes to allow the safepoint to progress
}
if(block_in_safepoint_check) {
 // current logic
 thread->set_thread_state(_thread_blocked);
 Safepoint_lock->unlock();
 Threads_lock->lock_without_safepoint_check();

 thread->set_thread_state(state);
 Threads_lock->unlock();
}
else {
 // We choose not to block
 thread->set_thread_state(_thread_blocked);
 Safepoint_lock->unlock();
}
```