Thread Management

This section provides some explanation of how Java™ threads are scheduled and synchronized by Jikes™ RVM.

All Java threads (application threads, garbage collector threads, etc.) derive from `RVMThread`. Each `RVMThread` maps directly to one native thread, which may be implemented using whichever C/C++ threading library is in use (currently either pthreads or Harmony threads). Unless -X:forceOneCPU is used, native threads are allowed to be arbitrarily scheduled by the OS using whatever processor resources are available; Jikes™ RVM does not attempt to control the thread-processor mapping at all.

Using native threading gives Jikes™ RVM better compatibility for existing JNI code, as well as improved performance, and greater infrastructure simplicity. Scheduling is offloaded entirely to the operating system; this is both what native code would expect and what maximizes the OS scheduler's ability to optimally schedule Java™ threads. As well, the resulting VM infrastructure is both simpler and more robust, since instead of focusing on scheduling decisions it can take a "hands-off" approach except when Java threads have to be preempted for sampling, on-stack-replacement, garbage collection, Thread.suspend(), or locking. The main task of `RVMThread` and other code in `org.jikesrvm.scheduler` is thus to override OS scheduling decisions when the VM demands it.

The remainder of this section is organized as follows. The management of a thread's state is discussed in detail. Mechanisms for blocking and handshaking threads are described. The VM's internal locking mechanism, the `Monitor`, is described. Finally, the locking implementation is discussed.

Tracking the Thread State

The state of a thread is broken down into two elements:

- Should the thread yield at a safe point?
- Is the thread running Java code right now?

The first mechanism is provided by the `RVMThread.takeYieldpoint` field, which is 0 if the thread should not yield, or non-zero if it should yield at the next safe point. Negative versus positive values indicate the type of safe point to yield at (epilogue/prologue, or any, respectively).

But this alone is insufficient to manage threads, as it relies on all threads being able to reach a safe point in a timely fashion. New Java threads may be started at any time, including at the exact moment that the garbage collector is starting; a starting-but-not-yet-started thread may not reach a safe point if the thread that was starting it is already blocked. Java threads may terminate at any time; terminated threads will never again reach a safe point. Any Java thread may call into arbitrary JNI code, which is outside of the VM's control, and may run for an arbitrary amount of time without reaching a Java safe point. As well, other mechanisms of `RVMThread` may cause a thread to block, thereby making it incapable of reaching a safe point in a timely fashion. However, in each of these cases, the Java thread is "effectively safe" - it is not running Java code that would interfere with the garbage collector, on-stack-replacement, locking, or any other Java runtime mechanism. Thus, a state management system is needed that would notify these runtime services when a thread is "effectively safe" and does not need to be waited on.

`RVMThread` provides for the following thread states, which describe to other runtime services the state of a Java thread. These states are designed with extreme care to support the following features:

- Allow Java threads to either execute Java code, which periodically reaches safe points, and native code which is "effectively safe" by virtue of not having access to VM services.
- Allow other threads (either Java threads or VM threads) to asynchronously request a Java thread to block. This overlaps with the `takeYieldpoint` mechanism, but adds the following feature: a thread that is "effectively safe" does not have to block.
- Prevent race conditions on state changes. In particular, if a thread running native code transitions back to running Java code while some other thread expects it to be either "effectively safe" or blocked at a safe point, then it should block. As well, if we are waiting on some Java thread to reach a safe point but it instead escapes into running native code, then we would like to be notified that even though it is not at a safe point, it is not effectively safe, and thus, we do not have to wait for it anymore.

The states used to put these features into effect are listed below.

- NEW. This means that the thread has been created but is not started, and hence is not yet running. NEW threads are always effectively safe, provided that they do not transition to any of the other states.
- IN_JAVA. The thread is running Java code. This almost always corresponds to the OS "runnable" state - i.e. the thread has no reason to be blocked, is on the runnable queue, and if a processor becomes available it will execute, if it is not already executing. IN_JAVA thread will periodically reach safe points at which the `takeYieldpoint` field will be tested. Hence, setting this field will ensure that the thread will yield in a timely fashion, unless it transitions into one of the other states in the meantime.
- IN_NATIVE. The thread is running either native C code, or internal VM code (which, by virtue of Jikes™ RVM's metacircularity, may be written in Java). IN_NATIVE threads are "effectively safe" in that they will not do anything that interferes with runtime services, at least until they transition into some other state. The IN_NATIVE state is most often used to denote threads that are blocked, for example on a lock.
- IN_JNI. The thread has called into JNI code. This is identical to the IN_NATIVE state in all ways except one: IN_JNI threads have a `JNIEnvironment` that stores more information about the thread's execution state (stack information, etc), while IN_NATIVE threads save only the minimum set of information required for the GC to perform stack scanning.
- IN_JAVA_TO_BLOCK. This represents a thread that is running Java code, as in IN_JAVA, but has been requested to yield. In most cases, when you set `takeYieldpoint` to non-zero, you will also change the state of the thread from IN_JAVA to

IN_JAVA_TO_BLOCK. If you don't intend on waiting for the thread (for example, in the case of sampling, where you're opportunistically requesting a yield), then this step may be omitted; but in the cases of locking and garbage collection, when a thread is requested to yield using `takeYieldpoint`, its state will also be changed.
- BLOCKED_IN_NATIVE. BLOCKED_IN_NATIVE is to IN_NATIVE as IN_JAVA_TO_BLOCK is to IN_JAVA. When requesting a thread to yield, we check its state; if it's IN_NATIVE, we set it to be BLOCKED_IN_NATIVE.
- BLOCKED_IN_JNI. Same as BLOCKED_IN_NATIVE, but for IN_JNI.
- TERMINATED. The thread has died. It is "effectively safe", but will never again reach a safe point.

The states are stored in `RVMThread.execStatus`, an integer field that may be rapidly manipulated using compare-and-swap. This field uses a hybrid synchronization protocol, which includes both compare-and-swap and conventional locking (using the thread's `Monitor`, accessible via the `RVMThread.monitor()` method). The rules are as follows:

- All state changes except for IN_JAVA to IN_NATIVE or IN_JNI, and IN_NATIVE or IN_JNI back to IN_JAVA, must be done while holding the lock.
- Only the thread itself can change its own state without holding the lock.
- The only asynchronous state changes (changes to the state not done by the thread that owns it) that are allowed are IN_JAVA to IN_JAVA_TO_BLOCK, IN_NATIVE to BLOCKED_IN_NATIVE, and IN_JNI TO BLOCKED_IN_JNI.

The typical algorithm for requesting a thread to block looks as follows:

```
thread.monitor().lockNoHandshake();
if (thread is running) {
   thread.takeYieldpoint=1;

   // transitions IN_JAVA -> IN_JAVA_TO_BLOCK,
IN_NATIVE->BLOCKED_IN_NATIVE, etc.
   thread.setBlockedExecStatus();

   if (thread.isInJava()) {
      // Thread will reach safe point soon, or else notify
      // us that it left to native code.
      // In either case, since we are holding the lock,
      // the thread will effectively block on either the safe point
      // or on the attempt to go to native code, since performing
      // either state transition requires acquiring the lock,
      // which we are now holding.
   } else {
      // Thread is in native code, and thus is "effectively safe",
      // and cannot go back to running Java code so long as we hold
      // the lock, since that state transition requires
      // acquiring the lock.
   }
}
thread.monitor().unlock();
```

Most of the time, you do not have to write such code, as the cases of blocking threads are already implemented. For examples of how to utilize these mechanisms, see `RVMThread.block()`, `RVMThread.hardHandshakeSuspend()`, and `RVMThread.softHandshake()`. A discussion of how to use these methods follows in the section below.

Finally, the valid state transitions are as follows.

- NEW to IN_JAVA: occurs when the thread is actually started. At this point it is safe to expect that the thread will reach a safe point in some bounded amount of time, at which point it will have a complete execution context, and this will be able to have its stack traces by GC.
- IN_JAVA to IN_JAVA_TO_BLOCK: occurs when an asynchronous request is made, for example to stop for GC, do a mutator flush, or do an isync on PPC.
- IN_JAVA to IN_NATIVE: occurs when the code opts to run in privileged mode, without synchronizing with GC. This state transition is only performed by `Monitor`, in cases where the thread is about to go idle while waiting for notifications (such as in the case of park, wait, or sleep), and by `org.jikesrvm.runtime.FileSystem`, as an optimization to allow I/O operations to be performed without a full JNI transition.
- IN_JAVA to IN_JNI: occurs in response to a JNI downcall, or return from a JNI upcall.
- IN_JAVA_TO_BLOCK to BLOCKED_IN_NATIVE: occurs when a thread that had been asked to perform an async activity decides to go

to privileged mode instead. This state always corresponds to a notification being sent to other threads, letting them know that this thread is idle. When the thread is idle, any asynchronous requests (such as mutator flushes) can instead be performed on behalf of this thread by other threads, since this thread is guaranteed not to be running any user Java code, and will not be able to return to running Java code without first blocking, and waiting to be unblocked (see BLOCKED_IN_NATIVE to IN_JAVA transition.
- IN_JAVA_TO_BLOCK to BLOCKED_IN_JNI: occurs when a thread that had been asked to perform an async activity decides to make a JNI downcall, or return from a JNI upcall, instead. In all other regards, this is identical to the IN_JAVA_TO_BLOCK to BLOCKED_IN_NATIVE transition.
- IN_NATIVE to IN_JAVA: occurs when a thread returns from idling or running privileged code to running Java code.
- BLOCKED_IN_NATIVE to IN_JAVA: occurs when a thread that had been asked to perform an async activity while running privileged code or idling decides to go back to running Java code. The actual transition is preceded by the thread first performing any requested actions (such as mutator flushes) and waiting for a notification that it is safe to continue running (for example, the thread may wait until GC is finished).
- IN_JNI to IN_JAVA: occurs when a thread returns from a JNI downcall, or makes a JNI upcall.
- BLOCKED_IN_JNI to IN_JAVA: same as BLOCKED_IN_NATIVE to IN_JAVA, except that this occurs in response to a return from a JNI downcall, or as the thread makes a JNI upcall.
- IN_JAVA to TERMINATED: the thread has terminated, and will never reach any more safe points, and thus will not be able to respond to any more requests for async activities.

Blocking and Handshaking

Various VM services, such as the garbage collector and locking, may wish to request a thread to block. In some cases, we want to block all threads except for the thread that makes the request. As well, some VM services may only wish for a "soft handshake", where we wait for each non-collector thread to perform some action exactly once and then continue (in this case, the only thread that blocks is the thread requesting the soft handshake, but all other non-collector threads must "yield" in order to perform the requested action; in most cases that action is non-blocking). A unified facility for performing all of these requests is provided by `RVMThread`.

Four types of thread blocking and handshaking are supported:

- `RVMThread.block()`. This is a low-level facility for requesting that a particular thread blocks. It is inherently unsafe to use this facility directly - for example, if thread A calls B.block() while thread B calls A.block(), the two threads may mutually deadlock.
- `RVMThread.beginPairHandshake()`. This implements a safe pair-handshaking mechanism, in which two threads become bound to each other for a short time. The thread requesting the pair handshake waits until the other thread is at a safe point or else is "effectively safe", and prevents it from going back to executing Java code. Note that at this point, neither thread will respond to any other handshake requests until `RVMThread.endPairHandshake()` is called. This is useful for implementing biased locking, but it has general utility anytime one thread needs to manipulate something another thread's execution state.
- `RVMThread.softHandshake()`. This implements soft handshakes. In a soft handshake, the requesting thread waits for all non-collector threads to perform some action exactly once, and then returns. If any of those threads are effectively safe, then the requesting thread performs the action on their behalf. `softHandshake()` is invoked with a `SoftHandshakeVisitor` that determines which threads are to be affected, and what the requested action is. An example of how this is used is found in `org.jikesrvm.mm.mmtk.Collection` and `org.jikesrvm.compilers.opt.runtimesupport.OptCompiledMethod`.
- `RVMThread.hardHandshakeSuspend()`. This stops all threads except for the garbage collector threads and the thread making the request. It returns once all Java threads are stopped. This is used by the garbage collector itself, but may be of utility elsewhere (for example, dynamic software updating). To resume all stopped threads, call `RVMThread.hardHandshakeResume()`. Note that this mechanism is carefully designed so that even after the world is stopped, it is safe to request a garbage collection (in that case, the garbage collector will itself call a variant of `hardHandshakeSuspend()`, but it will only affect the one remaining running Java thread).

The Monitor API

The VM internally uses an OS-based locking implementation, augmented with support for safe lock recursion and awareness of handshakes. The `Monitor` API provides locking and notification, similar to a Java lock, and may be implemented using either a `pthread_mutex` and a `pthread_cond`, or using Harmony's monitor API.

Acquiring a `Monitor` lock, or awaiting notification, may cause the calling `RVMThread` to block. This prevents the calling thread from acknowledging handshakes until the blocking call returns. In some cases, this is desirable. For example:

- In the implementation of handshakes, the code already takes special care to use the `RVMThread` state machine to notify other threads that the caller may block. As such, acquiring a lock or waiting for a notification is safe.
- If acquiring a lock that may only be held for a short, guaranteed-bounded length of time, the fact that the thread will ignore handshake requests while blocking is safe - the lock acquisition request will return in bounded time, allowing the thread to acknowledge any pending handshake requests.

But in all other cases, the calling thread must ensure that the handshake mechanism is notified that thread will block. Hence, all blocking `Monitor` methods have both a "NoHandshake" and "WithHandshake" version. Consider the following code:

```
someMonitor.lockNoHandshake();
// perform fast, bounded-time critical section
someMonitor.unlock(); // non-blocking
```

In this code, lock acquisition is done without notifying handshakes. This makes the acquisition faster. In this case, it is safe because the critical section is bounded-time. As well, we require that in this case, any other critical sections protected by `someMonitor` are bounded-time as well. If, on the other hand, the critical section was not bounded-time, we would do:

```
someMonitor.lockWithHandshake();
// perform potentially long critical section
someMonitor.unlock();
```

In this case, the `lockWithHandshake()` operation will transition the calling thread to the IN_NATIVE state before acquiring the lock, and then transition it back to IN_JAVA once the lock is acquired. This may cause the thread to block, if a handshake is in progress. As an added safety provision, if the `lockWithHandshake()` operation blocks due to a handshake, it will ensure that it does so without holding the `someMonitor` lock.

A special `Monitor` is provided with each thread. This monitor is of the type `NoYieldpointsMonitor` and will also ensure that yieldpoints (safe points) are disabled while the lock is held. This is necessary because any safe point may release the `Monitor` lock by waiting on it, thereby breaking atomicity of the critical section. The `NoYieldpointsMonitor` for any `RVMThread` may be accessed using the `RVMThread.monitor()` method.

Additional information about how to use this API is found in the following section, which discusses the implementation of Java locking.

Thin and Biased Locking

Jikes™ RVM uses a hybrid thin/biased locking implementation that is designed for very high performance under any of the following loads:

- Locks only ever acquired by one thread. In this case, biased locking is used, an no atomic operations (like compare-and-swap) need to be used to acquire and release locks.
- Locks acquired by multiple threads but rarely under contention. In this case, thin locking is used; acquiring and releasing the lock involves a fast inlined compare-and-swap operation. It is not as fast as biased locking on most architectures.
- Contended locks. Under sustained contention, the lock is "inflated" - the lock will now consist of data structures used to implement a fast barging FIFO mutex. A barging FIFO mutex allows threads to immediately acquire the lock as soon as it is available, or otherwise enqueue themselves on a FIFO and await its availability.

Thin locking has a relatively simple implementation; roughly 20 bits in the object header are used to represent the current lock state, and compare-and-swap is used to manipulate it. Biased locking and contended locking are more complicated, and are described below.

Biased locking makes the optimistic assumption that only one thread will ever want to acquire the lock. So long as this assumption holds, acquisition of the lock is a simple non-atomic increment/decrement. However, if the assumption is violated (a thread other than the one to which the lock is biased attempts to acquire the lock), a fallback mechanism is used to turn the lock into either a thin or contended lock. This works by using `RVMThread.beginPairHandshake()` to bring both the thread that is requesting the lock and the thread to which the lock is biased to a safe point. No other threads are affected; hence this system is very scalable. Once the pair handshake begins, the thread requesting the lock changes the lock into either a thin or contended lock, and then ends the pair handshake, allowing the thread to which the lock was biased to resume execution, while the thread requesting the lock may now contend on it using normal thin/contended mechanisms.

Contended locks, or "fat locks", consist of three mechanisms:

- A spin lock to protect the data structures.
- A queue of threads blocked on the lock.
- A mechanism for blocked threads to go to sleep until awoken by being dequeued.

The spin lock is a `org.jikesrvm.scheduler.SpinLock`. The queue is implemented in `org.jikesrvm.scheduler.ThreadQueue`. And the blocking/unblocking mechanism leverages `org.jikesrvm.scheduler.Monitor`; in particular, it uses the `Monitor` that is attached to each thread, accessible via `RVMThread.monitor()`. The basic algorithm for lock acquisition is:

```
spinLock.lock();
while (true) {
   if (lock available) {
      acquire the lock;
      break;
   } else {
      queue.enqueue(me);
      spinLock.unlock();

      me.monitor().lockNoHandshake();
      while (queue.isQueued(me)) {
         // put this thread to sleep waiting to be dequeued,
         // and do so while the thread is IN_NATIVE to ensure
         // that other threads don't wait on this one for
         // handshakes while we're blocked.
         me.monitor().waitWithHandshake();
      }
      me.monitor().unlock();
      spinLock.lock();
   }
}
spinLock.unlock();
```

The algorithm for unlocking dequeues the thread at the head of the queue (if there is one) and notifies its `Monitor` using the `lockedBroadcastNoHandshake()` method. Note that these algorithms span multiple methods in `org.jikesrvm.scheduler.ThinLock` and `org.jikesrvm.scheduler.Lock`; in particular, `lockHeavy()`, `lockHeavyLocked()`, `unlockHeavy()`, `lock()`, and `unlock()`.