/\*package Lock;

\* ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

\* Written by Doug Lea with assistance from members of JCP JSR-166

\* Expert Group and released to the public domain, as explained at

\* http://creativecommons.org/publicdomain/zero/1.0/

import java.util.concurrent.TimeUnit;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Date;

import sun.misc.Unsafe;

//\*//\*\*

\* Provides a framework for implementing blocking locks and related

\* synchronizers (semaphores, events, etc) that rely on

\* first-in-first-out (FIFO) wait queues. This class is designed to

\* be a useful basis for most kinds of synchronizers that rely on a

\* single atomic {@code int} value to represent state. Subclasses

\* must define the protected methods that change this state, and which

\* define what that state means in terms of this object being acquired

\* or released. Given these, the other methods in this class carry

\* out all queuing and blocking mechanics. Subclasses can maintain

\* other state fields, but only the atomically updated {@code int}

\* value manipulated using methods {@link #getState}, {@link

\* #setState} and {@link #compareAndSetState} is tracked with respect

\* to synchronization.

\*

\* <p>Subclasses should be defined as non-public internal helper

\* classes that are used to implement the synchronization properties

\* of their enclosing class. Class

\* {@code AbstractQueuedSynchronizer} does not implement any

\* synchronization interface. Instead it defines methods such as

\* {@link #acquireInterruptibly} that can be invoked as

\* appropriate by concrete locks and related synchronizers to

\* implement their public methods.

\*

\* <p>This class supports either or both a default <em>exclusive</em>

\* mode and a <em>shared</em> mode. When acquired in exclusive mode,

\* attempted acquires by other threads cannot succeed. Shared mode

\* acquires by multiple threads may (but need not) succeed. This class

\* does not &quot;understand&quot; these differences except in the

\* mechanical sense that when a shared mode acquire succeeds, the next

\* waiting thread (if one exists) must also determine whether it can

\* acquire as well. Threads waiting in the different modes share the

\* same FIFO queue. Usually, implementation subclasses support only

\* one of these modes, but both can come into play for example in a

\* {@link ReadWriteLock}. Subclasses that support only exclusive or

\* only shared modes need not define the methods supporting the unused mode.

\*

\* <p>This class defines a nested {@link ConditionObject} class that

\* can be used as a {@link Condition} implementation by subclasses

\* supporting exclusive mode for which method {@link

\* #isHeldExclusively} reports whether synchronization is exclusively

\* held with respect to the current thread, method {@link #release}

\* invoked with the current {@link #getState} value fully releases

\* this object, and {@link #acquire}, given this saved state value,

\* eventually restores this object to its previous acquired state. No

\* {@code AbstractQueuedSynchronizer} method otherwise creates such a

\* condition, so if this constraint cannot be met, do not use it. The

\* behavior of {@link ConditionObject} depends of course on the

\* semantics of its synchronizer implementation.

\*

\* <p>This class provides inspection, instrumentation, and monitoring

\* methods for the internal queue, as well as similar methods for

\* condition objects. These can be exported as desired into classes

\* using an {@code AbstractQueuedSynchronizer} for their

\* synchronization mechanics.

\*

\* <p>Serialization of this class stores only the underlying atomic

\* integer maintaining state, so deserialized objects have empty

\* thread queues. Typical subclasses requiring serializability will

\* define a {@code readObject} method that restores this to a known

\* initial state upon deserialization.

\*

\* <h3>Usage</h3>

\*

\* <p>To use this class as the basis of a synchronizer, redefine the

\* following methods, as applicable, by inspecting and/or modifying

\* the synchronization state using {@link #getState}, {@link

\* #setState} and/or {@link #compareAndSetState}:

\*

\* <ul>

\* <li> {@link #tryAcquire}

\* <li> {@link #tryRelease}

\* <li> {@link #tryAcquireShared}

\* <li> {@link #tryReleaseShared}

\* <li> {@link #isHeldExclusively}

\* </ul>

\*

\* Each of these methods by default throws {@link

\* UnsupportedOperationException}. Implementations of these methods

\* must be internally thread-safe, and should in general be short and

\* not block. Defining these methods is the <em>only</em> supported

\* means of using this class. All other methods are declared

\* {@code final} because they cannot be independently varied.

\*

\* <p>You may also find the inherited methods from {@link

\* AbstractOwnableSynchronizer} useful to keep track of the thread

\* owning an exclusive synchronizer. You are encouraged to use them

\* -- this enables monitoring and diagnostic tools to assist users in

\* determining which threads hold locks.

\*

\* <p>Even though this class is based on an internal FIFO queue, it

\* does not automatically enforce FIFO acquisition policies. The core

\* of exclusive synchronization takes the form:

\*

\* <pre>

\* Acquire:

\* while (!tryAcquire(arg)) {

\* <em>enqueue thread if it is not already queued</em>;

\* <em>possibly block current thread</em>;

\* }

\*

\* Release:

\* if (tryRelease(arg))

\* <em>unblock the first queued thread</em>;

\* </pre>

\*

\* (Shared mode is similar but may involve cascading signals.)

\*

\* <p id="barging">Because checks in acquire are invoked before

\* enqueuing, a newly acquiring thread may <em>barge</em> ahead of

\* others that are blocked and queued. However, you can, if desired,

\* define {@code tryAcquire} and/or {@code tryAcquireShared} to

\* disable barging by internally invoking one or more of the inspection

\* methods, thereby providing a <em>fair</em> FIFO acquisition order.

\* In particular, most fair synchronizers can define {@code tryAcquire}

\* to return {@code false} if {@link #hasQueuedPredecessors} (a method

\* specifically designed to be used by fair synchronizers) returns

\* {@code true}. Other variations are possible.

\*

\* <p>Throughput and scalability are generally highest for the

\* default barging (also known as <em>greedy</em>,

\* <em>renouncement</em>, and <em>convoy-avoidance</em>) strategy.

\* While this is not guaranteed to be fair or starvation-free, earlier

\* queued threads are allowed to recontend before later queued

\* threads, and each recontention has an unbiased chance to succeed

\* against incoming threads. Also, while acquires do not

\* &quot;spin&quot; in the usual sense, they may perform multiple

\* invocations of {@code tryAcquire} interspersed with other

\* computations before blocking. This gives most of the benefits of

\* spins when exclusive synchronization is only briefly held, without

\* most of the liabilities when it isn't. If so desired, you can

\* augment this by preceding calls to acquire methods with

\* "fast-path" checks, possibly prechecking {@link #hasContended}

\* and/or {@link #hasQueuedThreads} to only do so if the synchronizer

\* is likely not to be contended.

\*

\* <p>This class provides an efficient and scalable basis for

\* synchronization in part by specializing its range of use to

\* synchronizers that can rely on {@code int} state, acquire, and

\* release parameters, and an internal FIFO wait queue. When this does

\* not suffice, you can build synchronizers from a lower level using

\* {@link java.util.concurrent.atomic atomic} classes, your own custom

\* {@link java.util.Queue} classes, and {@link LockSupport} blocking

\* support.

\*

\* <h3>Usage Examples</h3>

\*

\* <p>Here is a non-reentrant mutual exclusion lock class that uses

\* the value zero to represent the unlocked state, and one to

\* represent the locked state. While a non-reentrant lock

\* does not strictly require recording of the current owner

\* thread, this class does so anyway to make usage easier to monitor.

\* It also supports conditions and exposes

\* one of the instrumentation methods:

\*

\* <pre> {@code

\* class Mutex implements Lock, java.io.Serializable {

\*

\* // Our internal helper class

\* private static class Sync extends AbstractQueuedSynchronizer {

\* // Reports whether in locked state

\* protected boolean isHeldExclusively() {

\* return getState() == 1;

\* }

\*

\* // Acquires the lock if state is zero

\* public boolean tryAcquire(int acquires) {

\* assert acquires == 1; // Otherwise unused

\* if (compareAndSetState(0, 1)) {

\* setExclusiveOwnerThread(Thread.currentThread());

\* return true;

\* }

\* return false;

\* }

\*

\* // Releases the lock by setting state to zero

\* protected boolean tryRelease(int releases) {

\* assert releases == 1; // Otherwise unused

\* if (getState() == 0) throw new IllegalMonitorStateException();

\* setExclusiveOwnerThread(null);

\* setState(0);

\* return true;

\* }

\*

\* // Provides a Condition

\* Condition newCondition() { return new ConditionObject(); }

\*

\* // Deserializes properly

\* private void readObject(ObjectInputStream s)

\* throws IOException, ClassNotFoundException {

\* s.defaultReadObject();

\* setState(0); // reset to unlocked state

\* }

\* }

\*

\* // The sync object does all the hard work. We just forward to it.

\* private final Sync sync = new Sync();

\*

\* public void lock() { sync.acquire(1); }

\* public boolean tryLock() { return sync.tryAcquire(1); }

\* public void unlock() { sync.release(1); }

\* public Condition newCondition() { return sync.newCondition(); }

\* public boolean isLocked() { return sync.isHeldExclusively(); }

\* public boolean hasQueuedThreads() { return sync.hasQueuedThreads(); }

\* public void lockInterruptibly() throws InterruptedException {

\* sync.acquireInterruptibly(1);

\* }

\* public boolean tryLock(long timeout, TimeUnit unit)

\* throws InterruptedException {

\* return sync.tryAcquireNanos(1, unit.toNanos(timeout));

\* }

\* }}</pre>

\*

\* <p>Here is a latch class that is like a

\* {@link java.util.concurrent.CountDownLatch CountDownLatch}

\* except that it only requires a single {@code signal} to

\* fire. Because a latch is non-exclusive, it uses the {@code shared}

\* acquire and release methods.

\*

\* <pre> {@code

\* class BooleanLatch {

\*

\* private static class Sync extends AbstractQueuedSynchronizer {

\* boolean isSignalled() { return getState() != 0; }

\*

\* protected int tryAcquireShared(int ignore) {

\* return isSignalled() ? 1 : -1;

\* }

\*

\* protected boolean tryReleaseShared(int ignore) {

\* setState(1);

\* return true;

\* }

\* }

\*

\* private final Sync sync = new Sync();

\* public boolean isSignalled() { return sync.isSignalled(); }

\* public void signal() { sync.releaseShared(1); }

\* public void await() throws InterruptedException {

\* sync.acquireSharedInterruptibly(1);

\* }

\* }}</pre>

\*

\* **@since** 1.5

\* **@author** Doug Lea

\*//\*

public abstract class AbstractQueuedSynchronizer

extends AbstractOwnableSynchronizer

implements java.io.Serializable {

private static final long serialVersionUID = 7373984972572414691L;

\*//\*\*

\* Creates a new {@code AbstractQueuedSynchronizer} instance

\* with initial synchronization state of zero.

\*//\*

protected AbstractQueuedSynchronizer() { }

\*//\*\*

\* Wait queue node class.

\*

\* <p>The wait queue is a variant of a "CLH" (Craig, Landin, and

\* Hagersten) lock queue. CLH locks are normally used for

\* spinlocks. We instead use them for blocking synchronizers, but

\* use the same basic tactic of holding some of the control

\* information about a thread in the predecessor of its node. A

\* "status" field in each node keeps track of whether a thread

\* should block. A node is signalled when its predecessor

\* releases. Each node of the queue otherwise serves as a

\* specific-notification-style monitor holding a single waiting

\* thread. The status field does NOT control whether threads are

\* granted locks etc though. A thread may try to acquire if it is

\* first in the queue. But being first does not guarantee success;

\* it only gives the right to contend. So the currently released

\* contender thread may need to rewait.

\*

\* <p>To enqueue into a CLH lock, you atomically splice it in as new

\* tail. To dequeue, you just set the head field.

\* <pre>

\* +------+ prev +-----+ +-----+

\* head | | <---- | | <---- | | tail

\* +------+ +-----+ +-----+

\* </pre>

\*

\* <p>Insertion into a CLH queue requires only a single atomic

\* operation on "tail", so there is a simple atomic point of

\* demarcation from unqueued to queued. Similarly, dequeuing

\* involves only updating the "head". However, it takes a bit

\* more work for nodes to determine who their successors are,

\* in part to deal with possible cancellation due to timeouts

\* and interrupts.

\*

\* <p>The "prev" links (not used in original CLH locks), are mainly

\* needed to handle cancellation. If a node is cancelled, its

\* successor is (normally) relinked to a non-cancelled

\* predecessor. For explanation of similar mechanics in the case

\* of spin locks, see the papers by Scott and Scherer at

\* http://www.cs.rochester.edu/u/scott/synchronization/

\*

\* <p>We also use "next" links to implement blocking mechanics.

\* The thread id for each node is kept in its own node, so a

\* predecessor signals the next node to wake up by traversing

\* next link to determine which thread it is. Determination of

\* successor must avoid races with newly queued nodes to set

\* the "next" fields of their predecessors. This is solved

\* when necessary by checking backwards from the atomically

\* updated "tail" when a node's successor appears to be null.

\* (Or, said differently, the next-links are an optimization

\* so that we don't usually need a backward scan.)

\*

\* <p>Cancellation introduces some conservatism to the basic

\* algorithms. Since we must poll for cancellation of other

\* nodes, we can miss noticing whether a cancelled node is

\* ahead or behind us. This is dealt with by always unparking

\* successors upon cancellation, allowing them to stabilize on

\* a new predecessor, unless we can identify an uncancelled

\* predecessor who will carry this responsibility.

\*

\* <p>CLH queues need a dummy header node to get started. But

\* we don't create them on construction, because it would be wasted

\* effort if there is never contention. Instead, the node

\* is constructed and head and tail pointers are set upon first

\* contention.

\*

\* <p>Threads waiting on Conditions use the same nodes, but

\* use an additional link. Conditions only need to link nodes

\* in simple (non-concurrent) linked queues because they are

\* only accessed when exclusively held. Upon await, a node is

\* inserted into a condition queue. Upon signal, the node is

\* transferred to the main queue. A special value of status

\* field is used to mark which queue a node is on.

\*

\* <p>Thanks go to Dave Dice, Mark Moir, Victor Luchangco, Bill

\* Scherer and Michael Scott, along with members of JSR-166

\* expert group, for helpful ideas, discussions, and critiques

\* on the design of this class.

\*//\*

static final class Node {

\*//\*\* Marker to indicate a node is waiting in shared mode \*//\*

static final Node SHARED = new Node();

\*//\*\* Marker to indicate a node is waiting in exclusive mode \*//\*

static final Node EXCLUSIVE = null;

\*//\*\* waitStatus value to indicate thread has cancelled \*//\*

static final int CANCELLED = 1;

\*//\*\* waitStatus value to indicate successor's thread needs unparking \*//\*

static final int SIGNAL = -1;

\*//\*\* waitStatus value to indicate thread is waiting on condition \*//\*

static final int CONDITION = -2;

\*//\*\*

\* waitStatus value to indicate the next acquireShared should

\* unconditionally propagate

\*//\*

static final int PROPAGATE = -3;

\*//\*\*

\* Status field, taking on only the values:

\* SIGNAL: The successor of this node is (or will soon be)

\* blocked (via park), so the current node must

\* unpark its successor when it releases or

\* cancels. To avoid races, acquire methods must

\* first indicate they need a signal,

\* then retry the atomic acquire, and then,

\* on failure, block.

\* CANCELLED: This node is cancelled due to timeout or interrupt.

\* Nodes never leave this state. In particular,

\* a thread with cancelled node never again blocks.

\* CONDITION: This node is currently on a condition queue.

\* It will not be used as a sync queue node

\* until transferred, at which time the status

\* will be set to 0. (Use of this value here has

\* nothing to do with the other uses of the

\* field, but simplifies mechanics.)

\* PROPAGATE: A releaseShared should be propagated to other

\* nodes. This is set (for head node only) in

\* doReleaseShared to ensure propagation

\* continues, even if other operations have

\* since intervened.

\* 0: None of the above

\*

\* The values are arranged numerically to simplify use.

\* Non-negative values mean that a node doesn't need to

\* signal. So, most code doesn't need to check for particular

\* values, just for sign.

\*

\* The field is initialized to 0 for normal sync nodes, and

\* CONDITION for condition nodes. It is modified using CAS

\* (or when possible, unconditional volatile writes).

\*//\*

volatile int waitStatus;

\*//\*\*

\* Link to predecessor node that current node/thread relies on

\* for checking waitStatus. Assigned during enqueuing, and nulled

\* out (for sake of GC) only upon dequeuing. Also, upon

\* cancellation of a predecessor, we short-circuit while

\* finding a non-cancelled one, which will always exist

\* because the head node is never cancelled: A node becomes

\* head only as a result of successful acquire. A

\* cancelled thread never succeeds in acquiring, and a thread only

\* cancels itself, not any other node.

\*//\*

volatile Node prev;

\*//\*\*

\* Link to the successor node that the current node/thread

\* unparks upon release. Assigned during enqueuing, adjusted

\* when bypassing cancelled predecessors, and nulled out (for

\* sake of GC) when dequeued. The enq operation does not

\* assign next field of a predecessor until after attachment,

\* so seeing a null next field does not necessarily mean that

\* node is at end of queue. However, if a next field appears

\* to be null, we can scan prev's from the tail to

\* double-check. The next field of cancelled nodes is set to

\* point to the node itself instead of null, to make life

\* easier for isOnSyncQueue.

\*//\*

volatile Node next;

\*//\*\*

\* The thread that enqueued this node. Initialized on

\* construction and nulled out after use.

\*//\*

volatile Thread thread;

\*//\*\*

\* Link to next node waiting on condition, or the special

\* value SHARED. Because condition queues are accessed only

\* when holding in exclusive mode, we just need a simple

\* linked queue to hold nodes while they are waiting on

\* conditions. They are then transferred to the queue to

\* re-acquire. And because conditions can only be exclusive,

\* we save a field by using special value to indicate shared

\* mode.

\*//\*

Node nextWaiter;

\*//\*\*

\* Returns true if node is waiting in shared mode.

\*//\*

final boolean isShared() {

return nextWaiter == SHARED;

}

\*//\*\*

\* Returns previous node, or throws NullPointerException if null.

\* Use when predecessor cannot be null. The null check could

\* be elided, but is present to help the VM.

\*

\* **@return** the predecessor of this node

\*//\*

final Node predecessor() throws NullPointerException {

Node p = prev;

if (p == null)

throw new NullPointerException();

else

return p;

}

Node() { // Used to establish initial head or SHARED marker

}

Node(Thread thread, Node mode) { // Used by addWaiter

this.nextWaiter = mode;

this.thread = thread;

}

Node(Thread thread, int waitStatus) { // Used by Condition

this.waitStatus = waitStatus;

this.thread = thread;

}

}

\*//\*\*

\* Head of the wait queue, lazily initialized. Except for

\* initialization, it is modified only via method setHead. Note:

\* If head exists, its waitStatus is guaranteed not to be

\* CANCELLED.

\*//\*

private transient volatile Node head;

\*//\*\*

\* Tail of the wait queue, lazily initialized. Modified only via

\* method enq to add new wait node.

\*//\*

private transient volatile Node tail;

\*//\*\*

\* The synchronization state.

\*//\*

private volatile int state;

\*//\*\*

\* Returns the current value of synchronization state.

\* This operation has memory semantics of a {@code volatile} read.

\* **@return** current state value

\*//\*

protected final int getState() {

return state;

}

\*//\*\*

\* Sets the value of synchronization state.

\* This operation has memory semantics of a {@code volatile} write.

\* **@param** newState the new state value

\*//\*

protected final void setState(int newState) {

state = newState;

}

\*//\*\*

\* Atomically sets synchronization state to the given updated

\* value if the current state value equals the expected value.

\* This operation has memory semantics of a {@code volatile} read

\* and write.

\*

\* **@param** expect the expected value

\* **@param** update the new value

\* **@return** {@code true} if successful. False return indicates that the actual

\* value was not equal to the expected value.

\*//\*

protected final boolean compareAndSetState(int expect, int update) {

// See below for intrinsics setup to support this

return unsafe.compareAndSwapInt(this, stateOffset, expect, update);

}

// Queuing utilities

\*//\*\*

\* The number of nanoseconds for which it is faster to spin

\* rather than to use timed park. A rough estimate suffices

\* to improve responsiveness with very short timeouts.

\*//\*

static final long spinForTimeoutThreshold = 1000L;

\*//\*\*

\* Inserts node into queue, initializing if necessary. See picture above.

\* **@param** node the node to insert

\* **@return** node's predecessor

\*//\*

private Node enq(final Node node) {

for (;;) {

Node t = tail;

if (t == null) { // Must initialize

if (compareAndSetHead(new Node()))

tail = head;

} else {

node.prev = t;

if (compareAndSetTail(t, node)) {

t.next = node;

return t;

}

}

}

}

\*//\*\*

\* Creates and enqueues node for current thread and given mode.

\*

\* **@param** mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared

\* **@return** the new node

\*//\*

private Node addWaiter(Node mode) {

Node node = new Node(Thread.currentThread(), mode);

// Try the fast path of enq; backup to full enq on failure

Node pred = tail;

if (pred != null) {

node.prev = pred;

if (compareAndSetTail(pred, node)) {

pred.next = node;

return node;

}

}

enq(node);

return node;

}

\*//\*\*

\* Sets head of queue to be node, thus dequeuing. Called only by

\* acquire methods. Also nulls out unused fields for sake of GC

\* and to suppress unnecessary signals and traversals.

\*

\* **@param** node the node

\*//\*

private void setHead(Node node) {

head = node;

node.thread = null;

node.prev = null;

}

\*//\*\*

\* Wakes up node's successor, if one exists.

\*

\* **@param** node the node

\*//\*

private void unparkSuccessor(Node node) {

\* If status is negative (i.e., possibly needing signal) try

\* to clear in anticipation of signalling. It is OK if this

\* fails or if status is changed by waiting thread.

int ws = node.waitStatus;

if (ws < 0)

compareAndSetWaitStatus(node, ws, 0);

\* Thread to unpark is held in successor, which is normally

\* just the next node. But if cancelled or apparently null,

\* traverse backwards from tail to find the actual

\* non-cancelled successor.

Node s = node.next;

if (s == null || s.waitStatus > 0) {

s = null;

for (Node t = tail; t != null && t != node; t = t.prev)

if (t.waitStatus <= 0)

s = t;

}

if (s != null)

LockSupport.unpark(s.thread);

}

\*//\*\*

\* Release action for shared mode -- signals successor and ensures

\* propagation. (Note: For exclusive mode, release just amounts

\* to calling unparkSuccessor of head if it needs signal.)

\*//\*

private void doReleaseShared() {

\* Ensure that a release propagates, even if there are other

\* in-progress acquires/releases. This proceeds in the usual

\* way of trying to unparkSuccessor of head if it needs

\* signal. But if it does not, status is set to PROPAGATE to

\* ensure that upon release, propagation continues.

\* Additionally, we must loop in case a new node is added

\* while we are doing this. Also, unlike other uses of

\* unparkSuccessor, we need to know if CAS to reset status

\* fails, if so rechecking.

for (;;) {

Node h = head;

if (h != null && h != tail) {

int ws = h.waitStatus;

if (ws == Node.SIGNAL) {

if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))

continue; // loop to recheck cases

unparkSuccessor(h);

}

else if (ws == 0 &&

!compareAndSetWaitStatus(h, 0, Node.PROPAGATE))

continue; // loop on failed CAS

}

if (h == head) // loop if head changed

break;

}

}

\*//\*\*

\* Sets head of queue, and checks if successor may be waiting

\* in shared mode, if so propagating if either propagate > 0 or

\* PROPAGATE status was set.

\*

\* **@param** node the node

\* **@param** propagate the return value from a tryAcquireShared

\*//\*

private void setHeadAndPropagate(Node node, int propagate) {

Node h = head; // Record old head for check below

setHead(node);

\* Try to signal next queued node if:

\* Propagation was indicated by caller,

\* or was recorded (as h.waitStatus either before

\* or after setHead) by a previous operation

\* (note: this uses sign-check of waitStatus because

\* PROPAGATE status may transition to SIGNAL.)

\* and

\* The next node is waiting in shared mode,

\* or we don't know, because it appears null

\*

\* The conservatism in both of these checks may cause

\* unnecessary wake-ups, but only when there are multiple

\* racing acquires/releases, so most need signals now or soon

\* anyway.

if (propagate > 0 || h == null || h.waitStatus < 0 ||

(h = head) == null || h.waitStatus < 0) {

Node s = node.next;

if (s == null || s.isShared())

doReleaseShared();

}

}

// Utilities for various versions of acquire

\*//\*\*

\* Cancels an ongoing attempt to acquire.

\*

\* **@param** node the node

\*//\*

private void cancelAcquire(Node node) {

// Ignore if node doesn't exist

if (node == null)

return;

node.thread = null;

// Skip cancelled predecessors

Node pred = node.prev;

while (pred.waitStatus > 0)

node.prev = pred = pred.prev;

// predNext is the apparent node to unsplice. CASes below will

// fail if not, in which case, we lost race vs another cancel

// or signal, so no further action is necessary.

Node predNext = pred.next;

// Can use unconditional write instead of CAS here.

// After this atomic step, other Nodes can skip past us.

// Before, we are free of interference from other threads.

node.waitStatus = Node.CANCELLED;

// If we are the tail, remove ourselves.

if (node == tail && compareAndSetTail(node, pred)) {

compareAndSetNext(pred, predNext, null);

} else {

// If successor needs signal, try to set pred's next-link

// so it will get one. Otherwise wake it up to propagate.

int ws;

if (pred != head &&

((ws = pred.waitStatus) == Node.SIGNAL ||

(ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) &&

pred.thread != null) {

Node next = node.next;

if (next != null && next.waitStatus <= 0)

compareAndSetNext(pred, predNext, next);

} else {

unparkSuccessor(node);

}

node.next = node; // help GC

}

}

\*//\*\*

\* Checks and updates status for a node that failed to acquire.

\* Returns true if thread should block. This is the main signal

\* control in all acquire loops. Requires that pred == node.prev.

\*

\* **@param** pred node's predecessor holding status

\* **@param** node the node

\* **@return** {@code true} if thread should block

\*//\*

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {

int ws = pred.waitStatus;

if (ws == Node.SIGNAL)

\* This node has already set status asking a release

\* to signal it, so it can safely park.

return true;

if (ws > 0) {

\* Predecessor was cancelled. Skip over predecessors and

\* indicate retry.

do {

node.prev = pred = pred.prev;

} while (pred.waitStatus > 0);

pred.next = node;

} else {

\* waitStatus must be 0 or PROPAGATE. Indicate that we

\* need a signal, but don't park yet. Caller will need to

\* retry to make sure it cannot acquire before parking.

compareAndSetWaitStatus(pred, ws, Node.SIGNAL);

}

return false;

}

\*//\*\*

\* Convenience method to interrupt current thread.

\*//\*

static void selfInterrupt() {

Thread.currentThread().interrupt();

}

\*//\*\*

\* Convenience method to park and then check if interrupted

\*

\* **@return** {@code true} if interrupted

\*//\*

private final boolean parkAndCheckInterrupt() {

LockSupport.park(this);

return Thread.interrupted();

}

\* Various flavors of acquire, varying in exclusive/shared and

\* control modes. Each is mostly the same, but annoyingly

\* different. Only a little bit of factoring is possible due to

\* interactions of exception mechanics (including ensuring that we

\* cancel if tryAcquire throws exception) and other control, at

\* least not without hurting performance too much.

\*//\*\*

\* Acquires in exclusive uninterruptible mode for thread already in

\* queue. Used by condition wait methods as well as acquire.

\*

\* **@param** node the node

\* **@param** arg the acquire argument

\* **@return** {@code true} if interrupted while waiting

\*//\*

final boolean acquireQueued(final Node node, int arg) {

boolean failed = true;

try {

boolean interrupted = false;

for (;;) {

final Node p = node.predecessor();

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return interrupted;

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

}

}

\*//\*\*

\* Acquires in exclusive interruptible mode.

\* **@param** arg the acquire argument

\*//\*

private void doAcquireInterruptibly(int arg)

throws InterruptedException {

final Node node = addWaiter(Node.EXCLUSIVE);

boolean failed = true;

try {

for (;;) {

final Node p = node.predecessor();

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return;

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

throw new InterruptedException();

}

} finally {

if (failed)

cancelAcquire(node);

}

}

\*//\*\*

\* Acquires in exclusive timed mode.

\*

\* **@param** arg the acquire argument

\* **@param** nanosTimeout max wait time

\* **@return** {@code true} if acquired

\*//\*

private boolean doAcquireNanos(int arg, long nanosTimeout)

throws InterruptedException {

if (nanosTimeout <= 0L)

return false;

final long deadline = System.nanoTime() + nanosTimeout;

final Node node = addWaiter(Node.EXCLUSIVE);

boolean failed = true;

try {

for (;;) {

final Node p = node.predecessor();

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return true;

}

nanosTimeout = deadline - System.nanoTime();

if (nanosTimeout <= 0L)

return false;

if (shouldParkAfterFailedAcquire(p, node) &&

nanosTimeout > spinForTimeoutThreshold)

LockSupport.parkNanos(this, nanosTimeout);

if (Thread.interrupted())

throw new InterruptedException();

}

} finally {

if (failed)

cancelAcquire(node);

}

}

\*//\*\*

\* Acquires in shared uninterruptible mode.

\* **@param** arg the acquire argument

\*//\*

private void doAcquireShared(int arg) {

final Node node = addWaiter(Node.SHARED);

boolean failed = true;

try {

boolean interrupted = false;

for (;;) {

final Node p = node.predecessor();

if (p == head) {

int r = tryAcquireShared(arg);

if (r >= 0) {

setHeadAndPropagate(node, r);

p.next = null; // help GC

if (interrupted)

selfInterrupt();

failed = false;

return;

}

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

}

}

\*//\*\*

\* Acquires in shared interruptible mode.

\* **@param** arg the acquire argument

\*//\*

private void doAcquireSharedInterruptibly(int arg)

throws InterruptedException {

final Node node = addWaiter(Node.SHARED);

boolean failed = true;

try {

for (;;) {

final Node p = node.predecessor();

if (p == head) {

int r = tryAcquireShared(arg);

if (r >= 0) {

setHeadAndPropagate(node, r);

p.next = null; // help GC

failed = false;

return;

}

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

throw new InterruptedException();

}

} finally {

if (failed)

cancelAcquire(node);

}

}

\*//\*\*

\* Acquires in shared timed mode.

\*

\* **@param** arg the acquire argument

\* **@param** nanosTimeout max wait time

\* **@return** {@code true} if acquired

\*//\*

private boolean doAcquireSharedNanos(int arg, long nanosTimeout)

throws InterruptedException {

if (nanosTimeout <= 0L)

return false;

final long deadline = System.nanoTime() + nanosTimeout;

final Node node = addWaiter(Node.SHARED);

boolean failed = true;

try {

for (;;) {

final Node p = node.predecessor();

if (p == head) {

int r = tryAcquireShared(arg);

if (r >= 0) {

setHeadAndPropagate(node, r);

p.next = null; // help GC

failed = false;

return true;

}

}

nanosTimeout = deadline - System.nanoTime();

if (nanosTimeout <= 0L)

return false;

if (shouldParkAfterFailedAcquire(p, node) &&

nanosTimeout > spinForTimeoutThreshold)

LockSupport.parkNanos(this, nanosTimeout);

if (Thread.interrupted())

throw new InterruptedException();

}

} finally {

if (failed)

cancelAcquire(node);

}

}

// Main exported methods

\*//\*\*

\* Attempts to acquire in exclusive mode. This method should query

\* if the state of the object permits it to be acquired in the

\* exclusive mode, and if so to acquire it.

\*

\* <p>This method is always invoked by the thread performing

\* acquire. If this method reports failure, the acquire method

\* may queue the thread, if it is not already queued, until it is

\* signalled by a release from some other thread. This can be used

\* to implement method {@link Lock#tryLock()}.

\*

\* <p>The default

\* implementation throws {@link UnsupportedOperationException}.

\*

\* **@param** arg the acquire argument. This value is always the one

\* passed to an acquire method, or is the value saved on entry

\* to a condition wait. The value is otherwise uninterpreted

\* and can represent anything you like.

\* **@return** {@code true} if successful. Upon success, this object has

\* been acquired.

\* **@throws** IllegalMonitorStateException if acquiring would place this

\* synchronizer in an illegal state. This exception must be

\* thrown in a consistent fashion for synchronization to work

\* correctly.

\* **@throws** UnsupportedOperationException if exclusive mode is not supported

\*//\*

protected boolean tryAcquire(int arg) {

throw new UnsupportedOperationException();

}

\*//\*\*

\* Attempts to set the state to reflect a release in exclusive

\* mode.

\*

\* <p>This method is always invoked by the thread performing release.

\*

\* <p>The default implementation throws

\* {@link UnsupportedOperationException}.

\*

\* **@param** arg the release argument. This value is always the one

\* passed to a release method, or the current state value upon

\* entry to a condition wait. The value is otherwise

\* uninterpreted and can represent anything you like.

\* **@return** {@code true} if this object is now in a fully released

\* state, so that any waiting threads may attempt to acquire;

\* and {@code false} otherwise.

\* **@throws** IllegalMonitorStateException if releasing would place this

\* synchronizer in an illegal state. This exception must be

\* thrown in a consistent fashion for synchronization to work

\* correctly.

\* **@throws** UnsupportedOperationException if exclusive mode is not supported

\*//\*

protected boolean tryRelease(int arg) {

throw new UnsupportedOperationException();

}

\*//\*\*

\* Attempts to acquire in shared mode. This method should query if

\* the state of the object permits it to be acquired in the shared

\* mode, and if so to acquire it.

\*

\* <p>This method is always invoked by the thread performing

\* acquire. If this method reports failure, the acquire method

\* may queue the thread, if it is not already queued, until it is

\* signalled by a release from some other thread.

\*

\* <p>The default implementation throws {@link

\* UnsupportedOperationException}.

\*

\* **@param** arg the acquire argument. This value is always the one

\* passed to an acquire method, or is the value saved on entry

\* to a condition wait. The value is otherwise uninterpreted

\* and can represent anything you like.

\* **@return** a negative value on failure; zero if acquisition in shared

\* mode succeeded but no subsequent shared-mode acquire can

\* succeed; and a positive value if acquisition in shared

\* mode succeeded and subsequent shared-mode acquires might

\* also succeed, in which case a subsequent waiting thread

\* must check availability. (Support for three different

\* return values enables this method to be used in contexts

\* where acquires only sometimes act exclusively.) Upon

\* success, this object has been acquired.

\* **@throws** IllegalMonitorStateException if acquiring would place this

\* synchronizer in an illegal state. This exception must be

\* thrown in a consistent fashion for synchronization to work

\* correctly.

\* **@throws** UnsupportedOperationException if shared mode is not supported

\*//\*

protected int tryAcquireShared(int arg) {

throw new UnsupportedOperationException();

}

\*//\*\*

\* Attempts to set the state to reflect a release in shared mode.

\*

\* <p>This method is always invoked by the thread performing release.

\*

\* <p>The default implementation throws

\* {@link UnsupportedOperationException}.

\*

\* **@param** arg the release argument. This value is always the one

\* passed to a release method, or the current state value upon

\* entry to a condition wait. The value is otherwise

\* uninterpreted and can represent anything you like.

\* **@return** {@code true} if this release of shared mode may permit a

\* waiting acquire (shared or exclusive) to succeed; and

\* {@code false} otherwise

\* **@throws** IllegalMonitorStateException if releasing would place this

\* synchronizer in an illegal state. This exception must be

\* thrown in a consistent fashion for synchronization to work

\* correctly.

\* **@throws** UnsupportedOperationException if shared mode is not supported

\*//\*

protected boolean tryReleaseShared(int arg) {

throw new UnsupportedOperationException();

}

\*//\*\*

\* Returns {@code true} if synchronization is held exclusively with

\* respect to the current (calling) thread. This method is invoked

\* upon each call to a non-waiting {@link ConditionObject} method.

\* (Waiting methods instead invoke {@link #release}.)

\*

\* <p>The default implementation throws {@link

\* UnsupportedOperationException}. This method is invoked

\* internally only within {@link ConditionObject} methods, so need

\* not be defined if conditions are not used.

\*

\* **@return** {@code true} if synchronization is held exclusively;

\* {@code false} otherwise

\* **@throws** UnsupportedOperationException if conditions are not supported

\*//\*

protected boolean isHeldExclusively() {

throw new UnsupportedOperationException();

}

\*//\*\*

\* Acquires in exclusive mode, ignoring interrupts. Implemented

\* by invoking at least once {@link #tryAcquire},

\* returning on success. Otherwise the thread is queued, possibly

\* repeatedly blocking and unblocking, invoking {@link

\* #tryAcquire} until success. This method can be used

\* to implement method {@link Lock#lock}.

\*

\* **@param** arg the acquire argument. This value is conveyed to

\* {@link #tryAcquire} but is otherwise uninterpreted and

\* can represent anything you like.

\*//\*

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

\*//\*\*

\* Acquires in exclusive mode, aborting if interrupted.

\* Implemented by first checking interrupt status, then invoking

\* at least once {@link #tryAcquire}, returning on

\* success. Otherwise the thread is queued, possibly repeatedly

\* blocking and unblocking, invoking {@link #tryAcquire}

\* until success or the thread is interrupted. This method can be

\* used to implement method {@link Lock#lockInterruptibly}.

\*

\* **@param** arg the acquire argument. This value is conveyed to

\* {@link #tryAcquire} but is otherwise uninterpreted and

\* can represent anything you like.

\* **@throws** InterruptedException if the current thread is interrupted

\*//\*

public final void acquireInterruptibly(int arg)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

if (!tryAcquire(arg))

doAcquireInterruptibly(arg);

}

\*//\*\*

\* Attempts to acquire in exclusive mode, aborting if interrupted,

\* and failing if the given timeout elapses. Implemented by first

\* checking interrupt status, then invoking at least once {@link

\* #tryAcquire}, returning on success. Otherwise, the thread is

\* queued, possibly repeatedly blocking and unblocking, invoking

\* {@link #tryAcquire} until success or the thread is interrupted

\* or the timeout elapses. This method can be used to implement

\* method {@link Lock#tryLock(long, TimeUnit)}.

\*

\* **@param** arg the acquire argument. This value is conveyed to

\* {@link #tryAcquire} but is otherwise uninterpreted and

\* can represent anything you like.

\* **@param** nanosTimeout the maximum number of nanoseconds to wait

\* **@return** {@code true} if acquired; {@code false} if timed out

\* **@throws** InterruptedException if the current thread is interrupted

\*//\*

public final boolean tryAcquireNanos(int arg, long nanosTimeout)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

return tryAcquire(arg) ||

doAcquireNanos(arg, nanosTimeout);

}

\*//\*\*

\* Releases in exclusive mode. Implemented by unblocking one or

\* more threads if {@link #tryRelease} returns true.

\* This method can be used to implement method {@link Lock#unlock}.

\*

\* **@param** arg the release argument. This value is conveyed to

\* {@link #tryRelease} but is otherwise uninterpreted and

\* can represent anything you like.

\* **@return** the value returned from {@link #tryRelease}

\*//\*

public final boolean release(int arg) {

if (tryRelease(arg)) {

Node h = head;

if (h != null && h.waitStatus != 0)

unparkSuccessor(h);

return true;

}

return false;

}

\*//\*\*

\* Acquires in shared mode, ignoring interrupts. Implemented by

\* first invoking at least once {@link #tryAcquireShared},

\* returning on success. Otherwise the thread is queued, possibly

\* repeatedly blocking and unblocking, invoking {@link

\* #tryAcquireShared} until success.

\*

\* **@param** arg the acquire argument. This value is conveyed to

\* {@link #tryAcquireShared} but is otherwise uninterpreted

\* and can represent anything you like.

\*//\*

public final void acquireShared(int arg) {

if (tryAcquireShared(arg) < 0)

doAcquireShared(arg);

}

\*//\*\*

\* Acquires in shared mode, aborting if interrupted. Implemented

\* by first checking interrupt status, then invoking at least once

\* {@link #tryAcquireShared}, returning on success. Otherwise the

\* thread is queued, possibly repeatedly blocking and unblocking,

\* invoking {@link #tryAcquireShared} until success or the thread

\* is interrupted.

\* **@param** arg the acquire argument.

\* This value is conveyed to {@link #tryAcquireShared} but is

\* otherwise uninterpreted and can represent anything

\* you like.

\* **@throws** InterruptedException if the current thread is interrupted

\*//\*

public final void acquireSharedInterruptibly(int arg)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

if (tryAcquireShared(arg) < 0)

doAcquireSharedInterruptibly(arg);

}

\*//\*\*

\* Attempts to acquire in shared mode, aborting if interrupted, and

\* failing if the given timeout elapses. Implemented by first

\* checking interrupt status, then invoking at least once {@link

\* #tryAcquireShared}, returning on success. Otherwise, the

\* thread is queued, possibly repeatedly blocking and unblocking,

\* invoking {@link #tryAcquireShared} until success or the thread

\* is interrupted or the timeout elapses.

\*

\* **@param** arg the acquire argument. This value is conveyed to

\* {@link #tryAcquireShared} but is otherwise uninterpreted

\* and can represent anything you like.

\* **@param** nanosTimeout the maximum number of nanoseconds to wait

\* **@return** {@code true} if acquired; {@code false} if timed out

\* **@throws** InterruptedException if the current thread is interrupted

\*//\*

public final boolean tryAcquireSharedNanos(int arg, long nanosTimeout)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

return tryAcquireShared(arg) >= 0 ||

doAcquireSharedNanos(arg, nanosTimeout);

}

\*//\*\*

\* Releases in shared mode. Implemented by unblocking one or more

\* threads if {@link #tryReleaseShared} returns true.

\*

\* **@param** arg the release argument. This value is conveyed to

\* {@link #tryReleaseShared} but is otherwise uninterpreted

\* and can represent anything you like.

\* **@return** the value returned from {@link #tryReleaseShared}

\*//\*

public final boolean releaseShared(int arg) {

if (tryReleaseShared(arg)) {

doReleaseShared();

return true;

}

return false;

}

// Queue inspection methods

\*//\*\*

\* Queries whether any threads are waiting to acquire. Note that

\* because cancellations due to interrupts and timeouts may occur

\* at any time, a {@code true} return does not guarantee that any

\* other thread will ever acquire.

\*

\* <p>In this implementation, this operation returns in

\* constant time.

\*

\* **@return** {@code true} if there may be other threads waiting to acquire

\*//\*

public final boolean hasQueuedThreads() {

return head != tail;

}

\*//\*\*

\* Queries whether any threads have ever contended to acquire this

\* synchronizer; that is if an acquire method has ever blocked.

\*

\* <p>In this implementation, this operation returns in

\* constant time.

\*

\* **@return** {@code true} if there has ever been contention

\*//\*

public final boolean hasContended() {

return head != null;

}

\*//\*\*

\* Returns the first (longest-waiting) thread in the queue, or

\* {@code null} if no threads are currently queued.

\*

\* <p>In this implementation, this operation normally returns in

\* constant time, but may iterate upon contention if other threads are

\* concurrently modifying the queue.

\*

\* **@return** the first (longest-waiting) thread in the queue, or

\* {@code null} if no threads are currently queued

\*//\*

public final Thread getFirstQueuedThread() {

// handle only fast path, else relay

return (head == tail) ? null : fullGetFirstQueuedThread();

}

\*//\*\*

\* Version of getFirstQueuedThread called when fastpath fails

\*//\*

private Thread fullGetFirstQueuedThread() {

\* The first node is normally head.next. Try to get its

\* thread field, ensuring consistent reads: If thread

\* field is nulled out or s.prev is no longer head, then

\* some other thread(s) concurrently performed setHead in

\* between some of our reads. We try this twice before

\* resorting to traversal.

Node h, s;

Thread st;

if (((h = head) != null && (s = h.next) != null &&

s.prev == head && (st = s.thread) != null) ||

((h = head) != null && (s = h.next) != null &&

s.prev == head && (st = s.thread) != null))

return st;

\* Head's next field might not have been set yet, or may have

\* been unset after setHead. So we must check to see if tail

\* is actually first node. If not, we continue on, safely

\* traversing from tail back to head to find first,

\* guaranteeing termination.

Node t = tail;

Thread firstThread = null;

while (t != null && t != head) {

Thread tt = t.thread;

if (tt != null)

firstThread = tt;

t = t.prev;

}

return firstThread;

}

\*//\*\*

\* Returns true if the given thread is currently queued.

\*

\* <p>This implementation traverses the queue to determine

\* presence of the given thread.

\*

\* **@param** thread the thread

\* **@return** {@code true} if the given thread is on the queue

\* **@throws** NullPointerException if the thread is null

\*//\*

public final boolean isQueued(Thread thread) {

if (thread == null)

throw new NullPointerException();

for (Node p = tail; p != null; p = p.prev)

if (p.thread == thread)

return true;

return false;

}

\*//\*\*

\* Returns {@code true} if the apparent first queued thread, if one

\* exists, is waiting in exclusive mode. If this method returns

\* {@code true}, and the current thread is attempting to acquire in

\* shared mode (that is, this method is invoked from {@link

\* #tryAcquireShared}) then it is guaranteed that the current thread

\* is not the first queued thread. Used only as a heuristic in

\* ReentrantReadWriteLock.

\*//\*

final boolean apparentlyFirstQueuedIsExclusive() {

Node h, s;

return (h = head) != null &&

(s = h.next) != null &&

!s.isShared() &&

s.thread != null;

}

\*//\*\*

\* Queries whether any threads have been waiting to acquire longer

\* than the current thread.

\*

\* <p>An invocation of this method is equivalent to (but may be

\* more efficient than):

\* <pre> {@code

\* getFirstQueuedThread() != Thread.currentThread() &&

\* hasQueuedThreads()}</pre>

\*

\* <p>Note that because cancellations due to interrupts and

\* timeouts may occur at any time, a {@code true} return does not

\* guarantee that some other thread will acquire before the current

\* thread. Likewise, it is possible for another thread to win a

\* race to enqueue after this method has returned {@code false},

\* due to the queue being empty.

\*

\* <p>This method is designed to be used by a fair synchronizer to

\* avoid <a href="AbstractQueuedSynchronizer#barging">barging</a>.

\* Such a synchronizer's {@link #tryAcquire} method should return

\* {@code false}, and its {@link #tryAcquireShared} method should

\* return a negative value, if this method returns {@code true}

\* (unless this is a reentrant acquire). For example, the {@code

\* tryAcquire} method for a fair, reentrant, exclusive mode

\* synchronizer might look like this:

\*

\* <pre> {@code

\* protected boolean tryAcquire(int arg) {

\* if (isHeldExclusively()) {

\* // A reentrant acquire; increment hold count

\* return true;

\* } else if (hasQueuedPredecessors()) {

\* return false;

\* } else {

\* // try to acquire normally

\* }

\* }}</pre>

\*

\* **@return** {@code true} if there is a queued thread preceding the

\* current thread, and {@code false} if the current thread

\* is at the head of the queue or the queue is empty

\* **@since** 1.7

\*//\*

public final boolean hasQueuedPredecessors() {

// The correctness of this depends on head being initialized

// before tail and on head.next being accurate if the current

// thread is first in queue.

Node t = tail; // Read fields in reverse initialization order

Node h = head;

Node s;

return h != t &&

((s = h.next) == null || s.thread != Thread.currentThread());

}

// Instrumentation and monitoring methods

\*//\*\*

\* Returns an estimate of the number of threads waiting to

\* acquire. The value is only an estimate because the number of

\* threads may change dynamically while this method traverses

\* internal data structures. This method is designed for use in

\* monitoring system state, not for synchronization

\* control.

\*

\* **@return** the estimated number of threads waiting to acquire

\*//\*

public final int getQueueLength() {

int n = 0;

for (Node p = tail; p != null; p = p.prev) {

if (p.thread != null)

++n;

}

return n;

}

\*//\*\*

\* Returns a collection containing threads that may be waiting to

\* acquire. Because the actual set of threads may change

\* dynamically while constructing this result, the returned

\* collection is only a best-effort estimate. The elements of the

\* returned collection are in no particular order. This method is

\* designed to facilitate construction of subclasses that provide

\* more extensive monitoring facilities.

\*

\* **@return** the collection of threads

\*//\*

public final Collection<Thread> getQueuedThreads() {

ArrayList<Thread> list = new ArrayList<Thread>();

for (Node p = tail; p != null; p = p.prev) {

Thread t = p.thread;

if (t != null)

list.add(t);

}

return list;

}

\*//\*\*

\* Returns a collection containing threads that may be waiting to

\* acquire in exclusive mode. This has the same properties

\* as {@link #getQueuedThreads} except that it only returns

\* those threads waiting due to an exclusive acquire.

\*

\* **@return** the collection of threads

\*//\*

public final Collection<Thread> getExclusiveQueuedThreads() {

ArrayList<Thread> list = new ArrayList<Thread>();

for (Node p = tail; p != null; p = p.prev) {

if (!p.isShared()) {

Thread t = p.thread;

if (t != null)

list.add(t);

}

}

return list;

}

\*//\*\*

\* Returns a collection containing threads that may be waiting to

\* acquire in shared mode. This has the same properties

\* as {@link #getQueuedThreads} except that it only returns

\* those threads waiting due to a shared acquire.

\*

\* **@return** the collection of threads

\*//\*

public final Collection<Thread> getSharedQueuedThreads() {

ArrayList<Thread> list = new ArrayList<Thread>();

for (Node p = tail; p != null; p = p.prev) {

if (p.isShared()) {

Thread t = p.thread;

if (t != null)

list.add(t);

}

}

return list;

}

\*//\*\*

\* Returns a string identifying this synchronizer, as well as its state.

\* The state, in brackets, includes the String {@code "State ="}

\* followed by the current value of {@link #getState}, and either

\* {@code "nonempty"} or {@code "empty"} depending on whether the

\* queue is empty.

\*

\* **@return** a string identifying this synchronizer, as well as its state

\*//\*

public String toString() {

int s = getState();

String q = hasQueuedThreads() ? "non" : "";

return super.toString() +

"[State = " + s + ", " + q + "empty queue]";

}

// Internal support methods for Conditions

\*//\*\*

\* Returns true if a node, always one that was initially placed on

\* a condition queue, is now waiting to reacquire on sync queue.

\* **@param** node the node

\* **@return** true if is reacquiring

\*//\*

final boolean isOnSyncQueue(Node node) {

if (node.waitStatus == Node.CONDITION || node.prev == null)

return false;

if (node.next != null) // If has successor, it must be on queue

return true;

\* node.prev can be non-null, but not yet on queue because

\* the CAS to place it on queue can fail. So we have to

\* traverse from tail to make sure it actually made it. It

\* will always be near the tail in calls to this method, and

\* unless the CAS failed (which is unlikely), it will be

\* there, so we hardly ever traverse much.

return findNodeFromTail(node);

}

\*//\*\*

\* Returns true if node is on sync queue by searching backwards from tail.

\* Called only when needed by isOnSyncQueue.

\* **@return** true if present

\*//\*

private boolean findNodeFromTail(Node node) {

Node t = tail;

for (;;) {

if (t == node)

return true;

if (t == null)

return false;

t = t.prev;

}

}

\*//\*\*

\* Transfers a node from a condition queue onto sync queue.

\* Returns true if successful.

\* **@param** node the node

\* **@return** true if successfully transferred (else the node was

\* cancelled before signal)

\*//\*

final boolean transferForSignal(Node node) {

\* If cannot change waitStatus, the node has been cancelled.

if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))

return false;

\* Splice onto queue and try to set waitStatus of predecessor to

\* indicate that thread is (probably) waiting. If cancelled or

\* attempt to set waitStatus fails, wake up to resync (in which

\* case the waitStatus can be transiently and harmlessly wrong).

Node p = enq(node);

int ws = p.waitStatus;

if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))

LockSupport.unpark(node.thread);

return true;

}

\*//\*\*

\* Transfers node, if necessary, to sync queue after a cancelled wait.

\* Returns true if thread was cancelled before being signalled.

\*

\* **@param** node the node

\* **@return** true if cancelled before the node was signalled

\*//\*

final boolean transferAfterCancelledWait(Node node) {

if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) {

enq(node);

return true;

}

\* If we lost out to a signal(), then we can't proceed

\* until it finishes its enq(). Cancelling during an

\* incomplete transfer is both rare and transient, so just

\* spin.

while (!isOnSyncQueue(node))

Thread.yield();

return false;

}

\*//\*\*

\* Invokes release with current state value; returns saved state.

\* Cancels node and throws exception on failure.

\* **@param** node the condition node for this wait

\* **@return** previous sync state

\*//\*

final int fullyRelease(Node node) {

boolean failed = true;

try {

int savedState = getState();

if (release(savedState)) {

failed = false;

return savedState;

} else {

throw new IllegalMonitorStateException();

}

} finally {

if (failed)

node.waitStatus = Node.CANCELLED;

}

}

// Instrumentation methods for conditions

\*//\*\*

\* Queries whether the given ConditionObject

\* uses this synchronizer as its lock.

\*

\* **@param** condition the condition

\* **@return** {@code true} if owned

\* **@throws** NullPointerException if the condition is null

\*//\*

public final boolean owns(ConditionObject condition) {

return condition.isOwnedBy(this);

}

\*//\*\*

\* Queries whether any threads are waiting on the given condition

\* associated with this synchronizer. Note that because timeouts

\* and interrupts may occur at any time, a {@code true} return

\* does not guarantee that a future {@code signal} will awaken

\* any threads. This method is designed primarily for use in

\* monitoring of the system state.

\*

\* **@param** condition the condition

\* **@return** {@code true} if there are any waiting threads

\* **@throws** IllegalMonitorStateException if exclusive synchronization

\* is not held

\* **@throws** IllegalArgumentException if the given condition is

\* not associated with this synchronizer

\* **@throws** NullPointerException if the condition is null

\*//\*

public final boolean hasWaiters(ConditionObject condition) {

if (!owns(condition))

throw new IllegalArgumentException("Not owner");

return condition.hasWaiters();

}

\*//\*\*

\* Returns an estimate of the number of threads waiting on the

\* given condition associated with this synchronizer. Note that

\* because timeouts and interrupts may occur at any time, the

\* estimate serves only as an upper bound on the actual number of

\* waiters. This method is designed for use in monitoring of the

\* system state, not for synchronization control.

\*

\* **@param** condition the condition

\* **@return** the estimated number of waiting threads

\* **@throws** IllegalMonitorStateException if exclusive synchronization

\* is not held

\* **@throws** IllegalArgumentException if the given condition is

\* not associated with this synchronizer

\* **@throws** NullPointerException if the condition is null

\*//\*

public final int getWaitQueueLength(ConditionObject condition) {

if (!owns(condition))

throw new IllegalArgumentException("Not owner");

return condition.getWaitQueueLength();

}

\*//\*\*

\* Returns a collection containing those threads that may be

\* waiting on the given condition associated with this

\* synchronizer. Because the actual set of threads may change

\* dynamically while constructing this result, the returned

\* collection is only a best-effort estimate. The elements of the

\* returned collection are in no particular order.

\*

\* **@param** condition the condition

\* **@return** the collection of threads

\* **@throws** IllegalMonitorStateException if exclusive synchronization

\* is not held

\* **@throws** IllegalArgumentException if the given condition is

\* not associated with this synchronizer

\* **@throws** NullPointerException if the condition is null

\*//\*

public final Collection<Thread> getWaitingThreads(ConditionObject condition) {

if (!owns(condition))

throw new IllegalArgumentException("Not owner");

return condition.getWaitingThreads();

}

\*//\*\*

\* Condition implementation for a {@link

\* AbstractQueuedSynchronizer} serving as the basis of a {@link

\* Lock} implementation.

\*

\* <p>Method documentation for this class describes mechanics,

\* not behavioral specifications from the point of view of Lock

\* and Condition users. Exported versions of this class will in

\* general need to be accompanied by documentation describing

\* condition semantics that rely on those of the associated

\* {@code AbstractQueuedSynchronizer}.

\*

\* <p>This class is Serializable, but all fields are transient,

\* so deserialized conditions have no waiters.

\*//\*

public class ConditionObject implements Condition, java.io.Serializable {

private static final long serialVersionUID = 1173984872572414699L;

\*//\*\* First node of condition queue. \*//\*

private transient Node firstWaiter;

\*//\*\* Last node of condition queue. \*//\*

private transient Node lastWaiter;

\*//\*\*

\* Creates a new {@code ConditionObject} instance.

\*//\*

public ConditionObject() { }

// Internal methods

\*//\*\*

\* Adds a new waiter to wait queue.

\* **@return** its new wait node

\*//\*

private Node addConditionWaiter() {

Node t = lastWaiter;

// If lastWaiter is cancelled, clean out.

if (t != null && t.waitStatus != Node.CONDITION) {

unlinkCancelledWaiters();

t = lastWaiter;

}

Node node = new Node(Thread.currentThread(), Node.CONDITION);

if (t == null)

firstWaiter = node;

else

t.nextWaiter = node;

lastWaiter = node;

return node;

}

\*//\*\*

\* Removes and transfers nodes until hit non-cancelled one or

\* null. Split out from signal in part to encourage compilers

\* to inline the case of no waiters.

\* **@param** first (non-null) the first node on condition queue

\*//\*

private void doSignal(Node first) {

do {

if ( (firstWaiter = first.nextWaiter) == null)

lastWaiter = null;

first.nextWaiter = null;

} while (!transferForSignal(first) &&

(first = firstWaiter) != null);

}

\*//\*\*

\* Removes and transfers all nodes.

\* **@param** first (non-null) the first node on condition queue

\*//\*

private void doSignalAll(Node first) {

lastWaiter = firstWaiter = null;

do {

Node next = first.nextWaiter;

first.nextWaiter = null;

transferForSignal(first);

first = next;

} while (first != null);

}

\*//\*\*

\* Unlinks cancelled waiter nodes from condition queue.

\* Called only while holding lock. This is called when

\* cancellation occurred during condition wait, and upon

\* insertion of a new waiter when lastWaiter is seen to have

\* been cancelled. This method is needed to avoid garbage

\* retention in the absence of signals. So even though it may

\* require a full traversal, it comes into play only when

\* timeouts or cancellations occur in the absence of

\* signals. It traverses all nodes rather than stopping at a

\* particular target to unlink all pointers to garbage nodes

\* without requiring many re-traversals during cancellation

\* storms.

\*//\*

private void unlinkCancelledWaiters() {

Node t = firstWaiter;

Node trail = null;

while (t != null) {

Node next = t.nextWaiter;

if (t.waitStatus != Node.CONDITION) {

t.nextWaiter = null;

if (trail == null)

firstWaiter = next;

else

trail.nextWaiter = next;

if (next == null)

lastWaiter = trail;

}

else

trail = t;

t = next;

}

}

// public methods

\*//\*\*

\* Moves the longest-waiting thread, if one exists, from the

\* wait queue for this condition to the wait queue for the

\* owning lock.

\*

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*//\*

public final void signal() {

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

Node first = firstWaiter;

if (first != null)

doSignal(first);

}

\*//\*\*

\* Moves all threads from the wait queue for this condition to

\* the wait queue for the owning lock.

\*

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*//\*

public final void signalAll() {

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

Node first = firstWaiter;

if (first != null)

doSignalAll(first);

}

\*//\*\*

\* Implements uninterruptible condition wait.

\* <ol>

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* </ol>

\*//\*

public final void awaitUninterruptibly() {

Node node = addConditionWaiter();

int savedState = fullyRelease(node);

boolean interrupted = false;

while (!isOnSyncQueue(node)) {

LockSupport.park(this);

if (Thread.interrupted())

interrupted = true;

}

if (acquireQueued(node, savedState) || interrupted)

selfInterrupt();

}

\* For interruptible waits, we need to track whether to throw

\* InterruptedException, if interrupted while blocked on

\* condition, versus reinterrupt current thread, if

\* interrupted while blocked waiting to re-acquire.

\*//\*\* Mode meaning to reinterrupt on exit from wait \*//\*

private static final int REINTERRUPT = 1;

\*//\*\* Mode meaning to throw InterruptedException on exit from wait \*//\*

private static final int THROW\_IE = -1;

\*//\*\*

\* Checks for interrupt, returning THROW\_IE if interrupted

\* before signalled, REINTERRUPT if after signalled, or

\* 0 if not interrupted.

\*//\*

private int checkInterruptWhileWaiting(Node node) {

return Thread.interrupted() ?

(transferAfterCancelledWait(node) ? THROW\_IE : REINTERRUPT) :

0;

}

\*//\*\*

\* Throws InterruptedException, reinterrupts current thread, or

\* does nothing, depending on mode.

\*//\*

private void reportInterruptAfterWait(int interruptMode)

throws InterruptedException {

if (interruptMode == THROW\_IE)

throw new InterruptedException();

else if (interruptMode == REINTERRUPT)

selfInterrupt();

}

\*//\*\*

\* Implements interruptible condition wait.

\* <ol>

\* <li> If current thread is interrupted, throw InterruptedException.

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled or interrupted.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* <li> If interrupted while blocked in step 4, throw InterruptedException.

\* </ol>

\*//\*

public final void await() throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

Node node = addConditionWaiter();

int savedState = fullyRelease(node);

int interruptMode = 0;

while (!isOnSyncQueue(node)) {

LockSupport.park(this);

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

}

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

if (node.nextWaiter != null) // clean up if cancelled

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

}

\*//\*\*

\* Implements timed condition wait.

\* <ol>

\* <li> If current thread is interrupted, throw InterruptedException.

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled, interrupted, or timed out.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* <li> If interrupted while blocked in step 4, throw InterruptedException.

\* </ol>

\*//\*

public final long awaitNanos(long nanosTimeout)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

Node node = addConditionWaiter();

int savedState = fullyRelease(node);

final long deadline = System.nanoTime() + nanosTimeout;

int interruptMode = 0;

while (!isOnSyncQueue(node)) {

if (nanosTimeout <= 0L) {

transferAfterCancelledWait(node);

break;

}

if (nanosTimeout >= spinForTimeoutThreshold)

LockSupport.parkNanos(this, nanosTimeout);

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

nanosTimeout = deadline - System.nanoTime();

}

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

if (node.nextWaiter != null)

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

return deadline - System.nanoTime();

}

\*//\*\*

\* Implements absolute timed condition wait.

\* <ol>

\* <li> If current thread is interrupted, throw InterruptedException.

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled, interrupted, or timed out.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* <li> If interrupted while blocked in step 4, throw InterruptedException.

\* <li> If timed out while blocked in step 4, return false, else true.

\* </ol>

\*//\*

public final boolean awaitUntil(Date deadline)

throws InterruptedException {

long abstime = deadline.getTime();

if (Thread.interrupted())

throw new InterruptedException();

Node node = addConditionWaiter();

int savedState = fullyRelease(node);

boolean timedout = false;

int interruptMode = 0;

while (!isOnSyncQueue(node)) {

if (System.currentTimeMillis() > abstime) {

timedout = transferAfterCancelledWait(node);

break;

}

LockSupport.parkUntil(this, abstime);

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

}

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

if (node.nextWaiter != null)

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

return !timedout;

}

\*//\*\*

\* Implements timed condition wait.

\* <ol>

\* <li> If current thread is interrupted, throw InterruptedException.

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled, interrupted, or timed out.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* <li> If interrupted while blocked in step 4, throw InterruptedException.

\* <li> If timed out while blocked in step 4, return false, else true.

\* </ol>

\*//\*

public final boolean await(long time, TimeUnit unit)

throws InterruptedException {

long nanosTimeout = unit.toNanos(time);

if (Thread.interrupted())

throw new InterruptedException();

Node node = addConditionWaiter();

int savedState = fullyRelease(node);

final long deadline = System.nanoTime() + nanosTimeout;

boolean timedout = false;

int interruptMode = 0;

while (!isOnSyncQueue(node)) {

if (nanosTimeout <= 0L) {

timedout = transferAfterCancelledWait(node);

break;

}

if (nanosTimeout >= spinForTimeoutThreshold)

LockSupport.parkNanos(this, nanosTimeout);

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

nanosTimeout = deadline - System.nanoTime();

}

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

if (node.nextWaiter != null)

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

return !timedout;

}

// support for instrumentation

\*//\*\*

\* Returns true if this condition was created by the given

\* synchronization object.

\*

\* **@return** {@code true} if owned

\*//\*

final boolean isOwnedBy(AbstractQueuedSynchronizer sync) {

return sync == AbstractQueuedSynchronizer.this;

}

\*//\*\*

\* Queries whether any threads are waiting on this condition.

\* Implements {@link AbstractQueuedSynchronizer#hasWaiters(ConditionObject)}.

\*

\* **@return** {@code true} if there are any waiting threads

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*//\*

protected final boolean hasWaiters() {

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

for (Node w = firstWaiter; w != null; w = w.nextWaiter) {

if (w.waitStatus == Node.CONDITION)

return true;

}

return false;

}

\*//\*\*

\* Returns an estimate of the number of threads waiting on

\* this condition.

\* Implements {@link AbstractQueuedSynchronizer#getWaitQueueLength(ConditionObject)}.

\*

\* **@return** the estimated number of waiting threads

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*//\*

protected final int getWaitQueueLength() {

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

int n = 0;

for (Node w = firstWaiter; w != null; w = w.nextWaiter) {

if (w.waitStatus == Node.CONDITION)

++n;

}

return n;

}

\*//\*\*

\* Returns a collection containing those threads that may be

\* waiting on this Condition.

\* Implements {@link AbstractQueuedSynchronizer#getWaitingThreads(ConditionObject)}.

\*

\* **@return** the collection of threads

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*//\*

protected final Collection<Thread> getWaitingThreads() {

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

ArrayList<Thread> list = new ArrayList<Thread>();

for (Node w = firstWaiter; w != null; w = w.nextWaiter) {

if (w.waitStatus == Node.CONDITION) {

Thread t = w.thread;

if (t != null)

list.add(t);

}

}

return list;

}

}

\*//\*\*

\* Setup to support compareAndSet. We need to natively implement

\* this here: For the sake of permitting future enhancements, we

\* cannot explicitly subclass AtomicInteger, which would be

\* efficient and useful otherwise. So, as the lesser of evils, we

\* natively implement using hotspot intrinsics API. And while we

\* are at it, we do the same for other CASable fields (which could

\* otherwise be done with atomic field updaters).

\*//\*

private static final Unsafe unsafe = Unsafe.getUnsafe();

private static final long stateOffset;

private static final long headOffset;

private static final long tailOffset;

private static final long waitStatusOffset;

private static final long nextOffset;

static {

try {

stateOffset = unsafe.objectFieldOffset

(AbstractQueuedSynchronizer.class.getDeclaredField("state"));

headOffset = unsafe.objectFieldOffset

(AbstractQueuedSynchronizer.class.getDeclaredField("head"));

tailOffset = unsafe.objectFieldOffset

(AbstractQueuedSynchronizer.class.getDeclaredField("tail"));

waitStatusOffset = unsafe.objectFieldOffset

(Node.class.getDeclaredField("waitStatus"));

nextOffset = unsafe.objectFieldOffset

(Node.class.getDeclaredField("next"));

} catch (Exception ex) { throw new Error(ex); }

}

\*//\*\*

\* CAS head field. Used only by enq.

\*//\*

private final boolean compareAndSetHead(Node update) {

return unsafe.compareAndSwapObject(this, headOffset, null, update);

}

\*//\*\*

\* CAS tail field. Used only by enq.

\*//\*

private final boolean compareAndSetTail(Node expect, Node update) {

return unsafe.compareAndSwapObject(this, tailOffset, expect, update);

}

\*//\*\*

\* CAS waitStatus field of a node.

\*//\*

private static final boolean compareAndSetWaitStatus(Node node,

int expect,

int update) {

return unsafe.compareAndSwapInt(node, waitStatusOffset,

expect, update);

}

\*//\*\*

\* CAS next field of a node.

\*//\*

private static final boolean compareAndSetNext(Node node,

Node expect,

Node update) {

return unsafe.compareAndSwapObject(node, nextOffset, expect, update);

}

}

\*/