The java.util.concurrent Synchronizer Framework

Doug Lea
SUNY Oswego
Oswego NY 13126
dl@cs.oswego.edu

ABSTRACT

Most synchronizers (locks, barriers, etc.) in the J2SE1.5 java.util.concurrent package are constructed using a small framework based on class `AbstractQueuedSynchronizer`. This framework provides common mechanics for atomically managing synchronization state, blocking and unblocking threads, and queuing. The paper describes the rationale, design, implementation, usage, and performance of this framework.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel Programming

General Terms

Algorithms, Measurement, Performance, Design.

Keywords

Synchronization, Java

1. INTRODUCTION

Java^{tm} release J2SE-1.5 introduces package java.util.concurrent, a collection of medium-level concurrency support classes created via Java Community Process (JCP) Java Specification Request (JSR) 166. Among these components are a set of *synchronizers* – abstract data type (ADT) classes that maintain an internal *synchronization state* (for example, representing whether a lock is locked or unlocked), operations to update and inspect that state, and at least one method that will cause a calling thread to block if the state requires it, resuming when some other thread changes the synchronization state to permit it. Examples include various forms of mutual exclusion locks, read-write locks, semaphores, barriers, futures, event indicators, and handoff queues.

As is well-known (see e.g., [2]) nearly any synchronizer can be used to implement nearly any other. For example, it is possible to build semaphores from reentrant locks, and vice versa. However, doing so often entails enough complexity, overhead, and inflexibility to be at best a second-rate engineering option. Further, it is conceptually unattractive. If none of these constructs are intrinsically more primitive than the others, developers should not be compelled to arbitrarily choose one of them as a basis for building others. Instead, JSR166 establishes a small framework centered on class `AbstractQueuedSynchronizer`, that provides common mechanics that are used by most of the provided synchronizers in the package, as well as other classes that users may define themselves.

The remainder of this paper discusses the requirements for this framework, the main ideas behind its design and implementation, sample usages, and some measurements showing its performance characteristics.

2. REQUIREMENTS

2.1 Functionality

Synchronizers possess two kinds of methods [7]: at least one *acquire* operation that blocks the calling thread unless/until the synchronization state allows it to proceed, and at least one *release* operation that changes synchronization state in a way that may allow one or more blocked threads to unblock.

The java.util.concurrent package does not define a single unified API for synchronizers. Some are defined via common interfaces (e.g., `Lock`), but others contain only specialized versions. So, *acquire* and *release* operations take a range of names and forms across different classes. For example, methods `Lock.lock`, `Semaphore.acquire`, `CountDownLatch.await`, and `FutureTask.get` all map to *acquire* operations in the framework. However, the package does maintain consistent conventions across classes to support a range of common usage options. When meaningful, each synchronizer supports:

- Nonblocking synchronization attempts (for example, `tryLock`) as well as blocking versions.
- Optional timeouts, so applications can give up waiting.
- Cancellability via interruption, usually separated into one version of acquire that is cancellable, and one that isn't.

Synchronizers may vary according to whether they manage only *exclusive* states – those in which only one thread at a time may continue past a possible blocking point – versus possible *shared* states in which multiple threads can at least sometimes proceed. Regular lock classes of course maintain only exclusive state, but counting semaphores, for example, may be acquired by as many threads as the count permits. To be widely useful, the framework must support both modes of operation.

The java.util.concurrent package also defines interface `Condition`, supporting monitor-style await/signal operations that may be associated with exclusive `Lock` classes, and whose implementations are intrinsically intertwined with their associated `Lock` classes.

2.2 Performance Goals

Java built-in locks (accessed using `synchronized` methods and blocks) have long been a performance concern, and there is a sizable literature on their construction (e.g., [1], [3]). However, the main focus of such work has been on minimizing space overhead (because any Java object can serve as a lock) and on minimizing time overhead when used in mostly-single-threaded

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSJP'04, July 26, 2004, St John's, Newfoundland, CA.

contexts on uniprocessors. Neither of these are especially important concerns for synchronizers: Programmers construct synchronizers only when needed, so there is no need to compact space that would otherwise be wasted, and synchronizers are used almost exclusively in multithreaded designs (increasingly often on multiprocessors) under which at least occasional contention is to be expected. So the usual JVM strategy of optimizing locks primarily for the zero-contention case, leaving other cases to less predictable "slow paths" [12] is not the right tactic for typical multithreaded server applications that rely heavily on java.util.concurrent.

Instead, the primary performance goal here is *scalability*: to predictably maintain efficiency even, or especially, when synchronizers are contended. Ideally, the overhead required to pass a synchronization point should be constant no matter how many threads are trying to do so. Among the main goals is to minimize the total amount of time during which some thread is permitted to pass a synchronization point but has not done so. However, this must be balanced against resource considerations, including total CPU time requirements, memory traffic, and thread scheduling overhead. For example, spinlocks usually provide shorter acquisition times than blocking locks, but usually waste cycles and generate memory contention, so are not often applicable.

These goals carry across two general styles of use. Most applications should maximize aggregate throughput, tolerating, at best, probabilistic guarantees about lack of starvation. However in applications such as resource control, it is far more important to maintain fairness of access across threads, tolerating poor aggregate throughput. No framework can decide between these conflicting goals on behalf of users; instead different fairness policies must be accommodated.

No matter how well-crafted they are internally, synchronizers will create performance bottlenecks in some applications. Thus, the framework must make it possible to monitor and inspect basic operations to allow users to discover and alleviate bottlenecks. This minimally (and most usefully) entails providing a way to determine how many threads are blocked.

3. DESIGN AND IMPLEMENTATION

The basic ideas behind a synchronizer are quite straightforward. An acquire operation proceeds as:

```
while (synchronization state does not allow acquire) {
  enqueue current thread if not already queued;
  possibly block current thread;
}
dequeue current thread if it was queued;
```

And a release operation is:

```
update synchronization state;
if (state may permit a blocked thread to acquire)
  unblock one or more queued threads;
```

Support for these operations requires the coordination of three basic components:

- Atomically managing synchronization state
- Blocking and unblocking threads
- Maintaining queues

It might be possible to create a framework that allows each of these three pieces to vary independently. However, this would neither be very efficient nor usable. For example, the information kept in queue nodes must mesh with that needed for unblocking, and the signatures of exported methods depend on the nature of synchronization state.

The central design decision in the synchronizer framework was to choose a concrete implementation of each of these three components, while still permitting a wide range of options in how they are used. This intentionally limits the range of applicability, but provides efficient enough support that there is practically never a reason not to use the framework (and instead build synchronizers from scratch) in those cases where it does apply.

3.1 Synchronization State

Class `AbstractQueuedSynchronizer` maintains synchronization state using only a single (32bit) `int`, and exports `getState`, `setState`, and `compareAndSetState` operations to access and update this state. These methods in turn rely on java.util.concurrent.atomic support providing JSR133 (Java Memory Model) compliant `volatile` semantics on reads and writes, and access to native compare-and-swap or load-linked/store-conditional instructions to implement `compareAndSetState`, that atomically sets state to a given new value only if it holds a given expected value.

Restricting synchronization state to a 32bit `int` was a pragmatic decision. While JSR166 also provides atomic operations on 64bit `long` fields, these must still be emulated using internal locks on enough platforms that the resulting synchronizers would not perform well. In the future, it seems likely that a second base class, specialized for use with 64bit state (i.e., with `long` control arguments), will be added. However, there is not now a compelling reason to include it in the package. Currently, 32 bits suffice for most applications. Only one java.util.concurrent synchronizer class, `CyclicBarrier`, would require more bits to maintain state, so instead uses locks (as do most higher-level utilities in the package).

Concrete classes based on `AbstractQueuedSynchronizer` must define methods `tryAcquire` and `tryRelease` in terms of these exported state methods in order to control the acquire and release operations. The `tryAcquire` method must return `true` if synchronization was acquired, and the `tryRelease` method must return `true` if the new synchronization state may allow future acquires. These methods accept a single `int` argument that can be used to communicate desired state; for example in a reentrant lock, to re-establish the recursion count when re-acquiring the lock after returning from a condition wait. Many synchronizers do not need such an argument, and so just ignore it.

3.2 Blocking

Until JSR166, there was no Java API available to block and unblock threads for purposes of creating synchronizers that are not based on built-in monitors. The only candidates were `Thread.suspend` and `Thread.resume`, which are unusable because they encounter an unsolvable race problem: If an unblocking thread invokes `resume` before the blocking thread has executed `suspend`, the `resume` operation will have no effect.

The java.util.concurrent.locks package includes a `LockSupport` class with methods that address this problem. Method `LockSupport.park` blocks the current thread unless or until a `LockSupport.unpark` has been issued. (Spurious wakeups are also permitted.) Calls to `unpark` are not "counted", so multiple `unparks` before a `park` only unblock a single `park`. Additionally, this applies per-thread, not per-synchronizer. A thread invoking `park` on a new synchronizer might return immediately because of a "leftover" `unpark` from a previous usage. However, in the absence of an `unpark`, its next invocation will block. While it would be possible to explicitly clear this state, it is not worth doing so. It is more efficient to invoke `park` multiple times when it happens to be necessary.

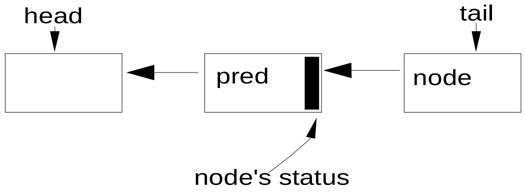
This simple mechanism is similar to those used, at some level, in the Solaris-9 thread library [11], in WIN32 "consumable events", and in the Linux NPTL thread library, and so maps efficiently to each of these on the most common platforms Java runs on. (However, the current Sun Hotspot JVM reference implementation on Solaris and Linux actually uses a pthread condvar in order to fit into the existing runtime design.) The `park` method also supports optional relative and absolute timeouts, and is integrated with JVM `Thread.interrupt` support — interrupting a thread unparks it.

3.3 Queues

The heart of the framework is maintenance of queues of blocked threads, which are restricted here to FIFO queues. Thus, the framework does not support priority-based synchronization.

These days, there is little controversy that the most appropriate choices for synchronization queues are non-blocking data structures that do not themselves need to be constructed using lower-level locks. And of these, there are two main candidates: variants of Mellor-Crummey and Scott (MCS) locks [9], and variants of Craig, Landin, and Hagersten (CLH) locks [5][8][10]. Historically, CLH locks have been used only in spinlocks. However, they appeared more amenable than MCS for use in the synchronizer framework because they are more easily adapted to handle cancellation and timeouts, so were chosen as a basis. The resulting design is far enough removed from the original CLH structure to require explanation.

A CLH queue is not very queue-like, because its enqueuing and dequeuing operations are intimately tied to its uses as a lock. It is a linked queue accessed via two atomically updatable fields, `head` and `tail`, both initially pointing to a dummy node.



A new node, `node`, is *enqueued* using an atomic operation:

```
  do {  pred = tail;
   } while(!tail.compareAndSet(pred, node));
```

The release status for each node is kept in its predecessor node. So, the "spin" of a spinlock looks like:

```
  while (pred.status != RELEASED) ; // spin
```

A *dequeue* operation after this spin simply entails setting the `head` field to the node that just got the lock:

```
  head = node;
```

Among the advantages of CLH locks are that enqueuing and dequeuing are fast, lock-free, and obstruction free (even under contention, one thread will always win an insertion race so will make progress); that detecting whether any threads are waiting is also fast (just check if `head` is the same as `tail`); and that release status is decentralized, avoiding some memory contention.

In the original versions of CLH locks, there were not even links connecting nodes. In a spinlock, the `pred` variable can be held as a local. However, Scott and Scherer[10] showed that by explicitly maintaining predecessor fields within nodes, CLH locks can deal with timeouts and other forms of cancellation: If a node's predecessor cancels, the node can slide up to use the previous node's status field.

The main additional modification needed to use CLH queues for blocking synchronizers is to provide an efficient way for one node to locate its successor. In spinlocks, a node need only change its status, which will be noticed on next spin by its successor, so links are unnecessary. But in a blocking synchronizer, a node needs to explicitly wake up (`unpark`) its successor.

An `AbstractQueuedSynchronizer` queue node contains a `next` link to its successor. But because there are no applicable techniques for lock-free atomic insertion of double-linked list nodes using compareAndSet, this link is not atomically set as part of insertion; it is simply assigned:

```
  pred.next = node;
```

after the insertion. This is reflected in all usages. The `next` link is treated only as an optimized path. If a node's successor does not appear to exist (or appears to be cancelled) via its `next` field, it is always possible to start at the tail of the list and traverse backwards using the `pred` field to accurately check if there really is one.

A second set of modifications is to use the status field kept in each node for purposes of controlling blocking, not spinning. In the synchronizer framework, a queued thread can only return from an acquire operation if it passes the `tryAcquire` method defined in a concrete subclass; a single "released" bit does not suffice. But control is still needed to ensure that an active thread is only allowed to invoke `tryAcquire` when it is at the head of the queue; in which case it may fail to acquire, and (re)block. This does not require a per-node status flag because permission can be determined by checking that the current node's predecessor is the `head`. And unlike the case of spinlocks, there is not enough memory contention reading `head` to warrant replication. However, cancellation status must still be present in the status field.

The queue node status field is also used to avoid needless calls to `park` and `unpark`. While these methods are relatively fast as blocking primitives go, they encounter avoidable overhead in the boundary crossing between Java and the JVM runtime and/or OS. Before invoking `park`, a thread sets a "*signal* me" bit, and then rechecks synchronization and node status once more before invoking `park`. A releasing thread clears status. This saves threads from needlessly attempting to block often enough to be worthwhile, especially for lock classes in which lost time waiting for the next eligible thread to acquire a lock accentuates other contention effects. This also avoids requiring a releasing thread to determine its successor unless the successor has set the signal bit, which in turn eliminates those cases where it must traverse multiple nodes to cope with an apparently null `next` field unless signalling occurs in conjunction with cancellation.

Perhaps the main difference between the variant of CLH locks used in the synchronizer framework and those employed in other languages is that garbage collection is relied on for managing storage reclamation of nodes, which avoids complexity and overhead. However, reliance on GC does still entail nulling of link fields when they are sure to never to be needed. This can normally be done when dequeuing. Otherwise, unused nodes would still be reachable, causing them to be uncollectable.

Some further minor tunings, including lazy initialization of the initial dummy node required by CLH queues upon first contention, are described in the source code documentation in the J2SE1.5 release.

Omitting such details, the general form of the resulting implementation of the basic acquire operation (exclusive, noninterruptible, untimed case only) is:

```
if (!tryAcquire(arg)) {
  node = create and enqueue new node;
  pred = node's effective predecessor;
  while  (pred is not head node ||  !tryAcquire(arg)) {
    if (pred's signal bit is set)
      park();
    else
      compareAndSet pred's signal bit to true;
    pred = node's effective predecessor;
  }
  head = node;
}
```

And the release operation is:

```
if (tryRelease(arg) && head node's signal bit is set) {
   compareAndSet head's signal bit to false;
   unpark head's successor, if one exists
}
```

The number of iterations of the main acquire loop depends, of course, on the nature of `tryAcquire`. Otherwise, in the absence of cancellation, each component of acquire and release is a constant-time *O(1)* operation, amortized across threads, disregarding any OS thread scheduling occuring within `park`.

Cancellation support mainly entails checking for interrupt or timeout upon each return from `park` inside the acquire loop. A cancelled thread due to timeout or interrupt sets its node status and unparks its successor so it may reset links. With cancellation, determining predecessors and successors and resetting status may include *O(n)* traversals (where *n* is the length of the queue). Because a thread never again blocks for a cancelled operation, links and status fields tend to restabilize quickly.

3.4 Condition Queues

The synchronizer framework provides a `ConditionObject` class for use by synchronizers that maintain exclusive synchronization and conform to the `Lock` interface. Any number of condition objects may be attached to a lock object, providing classic monitor-style `await`, `signal`, and `signalAll` operations, including those with timeouts, along with some inspection and monitoring methods.

The `ConditionObject` class enables conditions to be efficiently integrated with other synchronization operations, again by fixing some design decisions. This class supports only Java-style monitor access rules in which condition operations are legal only when the lock owning the condition is held by the current thread (See [4] for discussion of alternatives). Thus, a `ConditionObject` attached to a `ReentrantLock` acts in the same way as do built-in monitors (via `Object.wait` etc), differing only in method names, extra functionality, and the fact that users can declare multiple conditions per lock.

A `ConditionObject` uses the same internal queue nodes as synchronizers, but maintains them on a separate condition queue. The signal operation is implemented as a queue transfer from the condition queue to the lock queue, without necessarily waking up the signalled thread before it has re-acquired its lock.

The basic await operation is:

```
create and add new node to condition queue;
release  lock;
block until node is on lock queue;
re-acquire lock;
```

And the signal operation is:

```
transfer the first node from condition queue to lock queue;
```

Because these operations are performed only when the lock is held, they can use sequential linked queue operations (using a `nextWaiter` field in nodes) to maintain the condition queue. The transfer operation simply unlinks the first node from the condition queue, and then uses CLH insertion to attach it to the lock queue.

The main complication in implementing these operations is dealing with cancellation of condition waits due to timeouts or `Thread.interrupt`. A cancellation and signal occuring at approximately the same time encounter a race whose outcome conforms to the specifications for built-in monitors. As revised in JSR133, these require that if an interrupt occurs before a signal, then the `await` method must, after re-acquiring the lock, throw `InterruptedException`. But if it is interrupted after a signal, then the method must return without throwing an exception, but with its thread interrupt status set.

To maintain proper ordering, a bit in the queue node status records whether the node has been (or is in the process of being) transferred. Both the signalling code and the cancelling code try to compareAndSet this status. If a signal operation loses this race, it instead transfers the next node on the queue, if one exists. If a cancellation loses, it must abort the transfer, and then await lock re-acquisition. This latter case introduces a potentially unbounded spin. A cancelled wait cannot commence lock re-acquisition until the node has been successfully inserted on the lock queue, so must spin waiting for the CLH queue insertion compareAndSet being performed by the signalling thread to succeed. The need to spin here is rare, and employs a `Thread.yield` to provide a scheduling hint that some other thread, ideally the one doing the signal, should instead run. While it would be possible to implement here a helping strategy for the cancellation to insert the node, the case is much too rare to justify the added overhead that this would entail. In all other cases, the basic mechanics here and elsewhere use no spins or yields, which maintains reasonable performance on uniprocessors.

4. USAGE

Class `AbstractQueuedSynchronizer` ties together the above functionality and serves as a "template method pattern" [6] base class for synchronizers. Subclasses define only the methods that implement the state inspections and updates that control acquire and release. However, subclasses of `AbstractQueuedSynchronizer` are not themselves usable as synchronizer ADTs, because the class necessarily exports the methods needed to internally control acquire and release policies, which should not be made visible to users of these classes. All java.util.concurrent synchronizer classes declare a private inner `AbstractQueuedSynchronizer` subclass and delegate all synchronization methods to it. This also allows public methods to be given names appropriate to the synchronizer.

For example, here is a minimal `Mutex` class, that uses synchronization state zero to mean unlocked, and one to mean locked. This class does not need the value arguments supported for synchronization methods, so uses zero, and otherwise ignores them.

```
class Mutex {
  class Sync
    extends AbstractQueuedSynchronizer {
    public boolean tryAcquire(int ignore) {
      return compareAndSetState(0, 1);
    }
    public boolean tryRelease(int ignore) {
       setState(0); return true;
    }
  }
  private final Sync sync = new Sync();
  public void lock()   { sync.acquire(0); }
  public void unlock() { sync.release(0); }
}
```

A fuller version of this example, along with other usage guidance may be found in the J2SE documentation. Many variants are of course possible. For example, `tryAcquire` could employ "test-and-test-and-set" by checking the state value before trying to change it.

It may be surprising that a construct as performance-sensitive as a mutual exclusion lock is intended to be defined using a combination of delegation and virtual methods. However, these are the sorts of OO design constructions that modern dynamic compilers have long focussed on. They tend to be good at optimizing away this overhead, at least in code in which synchronizers are invoked frequently.

Class `AbstractQueuedSynchronizer` also supplies a number of methods that assist synchronizer classes in policy control. For example, it includes timeout and interruptible versions of the basic acquire method. And while discussion so far has focussed on exclusive-mode synchronizers such as locks, the `AbstractQueuedSynchronizer` class also contains a parallel set of methods (such as `acquireShared`) that differ in that the `tryAcquireShared` and `tryReleaseShared` methods can inform the framework (via their return values) that further acquires may be possible, ultimately causing it to wake up multiple threads by cascading signals.

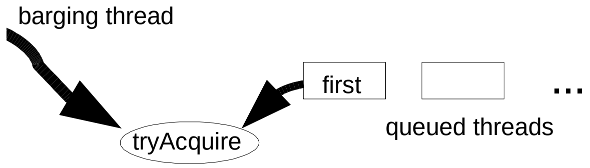
Although it is not usually sensible to serialize (persistently store or transmit) a synchronizer, these classes are often used in turn to construct other classes, such as thread-safe collections, that are commonly serialized. The `AbstractQueuedSynchronizer` and `ConditionObject` classes provide methods to serialize synchronization state, but not the underlying blocked threads or other intrinsically transient bookkeeping. Even so, most synchronizer classes merely reset synchronization state to initial values on deserialization, in keeping with the implicit policy of built-in locks of always deserializing to an unlocked state. This amounts to a no-op, but must still be explicitly supported to enable deserialization of `final` fields.

4.1 Controlling Fairness

Even though they are based on FIFO queues, synchronizers are not necessarily fair. Notice that in the basic acquire algorithm (Section 3.3), `tryAcquire` checks are performed *before* queuing. Thus a newly acquiring thread can “steal” access that is "intended" for the first thread at the head of the queue.

This *barging FIFO* strategy generally provides higher aggregate throughput than other techniques. It reduces the time during which a contended lock is available but no thread has it because the intended next thread is in the process of unblocking. At the same time, it avoids excessive, unproductive contention by only allowing one (the first) queued thread to wake up and try to acquire upon any release. Developers creating synchronizers may further accentuate barging effects in cases where synchronizers are expected to be held only briefly by defining `tryAcquire` to itself retry a few times before passing back control.

Barging FIFO synchronizers have only probablistic fairness properties. An unparked thread at the head of the lock queue has an unbiased chance of winning a race with any incoming barging thread, reblocking and retrying if it loses. However, if incoming threads arrive faster than it takes an unparked thread to unblock, the first thread in the queue will only rarely win the race, so will almost always reblock, and its successors will remain blocked. With briefly-held synchronizers, it is common for multiple bargings and releases to occur on multiprocessors during the time the first thread takes to unblock. As seen below, the net effect is to maintain high rates of progress of one or more threads while still at least probabilistically avoiding starvation.



When greater fairness is required, it is a relatively simple matter to arrange it. Programmers requiring strict fairness can define `tryAcquire` to fail (return false) if the current thread is not at the head of the queue, checking for this using method `getFirstQueuedThread`, one of a handful of supplied inspection methods.

A faster, less strict variant is to also allow `tryAcquire` to succeed if the the queue is (momentarily) empty. In this case, multiple threads encountering an empty queue may race to be the first to acquire, normally without enqueuing at least one of them. This strategy is adopted in all java.util.concurrent synchronizers supporting a "fair" mode.

While they tend to be useful in practice, fairness settings have no guarantees, because the Java Language Specification does not provide scheduling guarantees. For example, even with a strictly fair synchronizer, a JVM could decide to run a set of threads purely sequentially if they never otherwise need to block waiting for each other. In practice, on a uniprocessor, such threads are likely to each run for a time quantum before being pre-emptively context-switched. If such a thread is holding an exclusive lock, it will soon be momentarily switched back, only to release the lock and block now that it is known that another thread needs the lock, thus further increasing the periods during which a synchronizer is available but not acquired. Synchronizer fairness settings tend to have even greater impact on multiprocessors, which generate more interleavings, and hence more opportunities for one thread to discover that a lock is needed by another thread.

Even though they may perform poorly under high contention when protecting briefly-held code bodies, fair locks work well, for example, when they protect relatively long code bodies and/or with relatively long inter-lock intervals, in which case barging provides little performance advantage and but greater risk of indefinite postponement. The synchronizer framework leaves such engineering decisions to its users.

4.2 Synchronizers

Here are sketches of how java.util.concurrent synchronizer classes are defined using this framework:

The `ReentrantLock` class uses synchronization state to hold the (recursive) lock count. When a lock is acquired, it also records the identity of the current thread to check recursions and detect illegal state exceptions when the wrong thread tries to unlock. The class also uses the provided `ConditionObject`, and exports other monitoring and inspection methods. The class supports an optional "fair" mode by internally declaring two different `AbstractQueuedSynchronizer` subclasses (the fair one disabling barging) and setting each `ReentrantLock` instance to use the appropriate one upon construction.

The `ReentrantReadWriteLock` class uses 16 bits of the synchronization state to hold the write lock count, and the remaining 16 bits to hold the read lock count. The `WriteLock` is otherwise structured in the same way as `ReentrantLock`. The `ReadLock` uses the `acquireShared` methods to enable multiple readers.

The `Semaphore` class (a counting semaphore) uses the synchronization state to hold the current count. It defines `acquireShared` to decrement the count or block if nonpositive, and `tryRelease` to increment the count, possibly unblocking threads if it is now positive.

The `CountDownLatch` class uses the synchronization state to represent the count. All acquires pass when it reaches zero.

The `FutureTask` class uses the synchronization state to represent the run-state of a future (initial, running, cancelled, done). Setting or cancelling a future invokes `release`, unblocking threads waiting for its computed value via `acquire`.

The `SynchronousQueue` class (a CSP-style handoff) uses internal wait-nodes that match up producers and consumers. It uses the synchronization state to allow a producer to proceed when a consumer takes the item, and vice-versa.

Users of the java.util.concurrent package may of course define their own synchronizers for custom applications. For example, among those that were considered but not adopted in the package are classes providing the semantics of various flavors of WIN32 events, binary latches, centrally managed locks, and tree-based barriers.

5. PERFORMANCE

While the synchronizer framework supports many other styles of synchronization in addition to mutual exclusion locks, lock performance is simplest to measure and compare. Even so, there are many different approaches to measurement. The experiments here are designed to reveal overhead and throughput.

In each test, each thread repeatedly updates a pseudo-random number computed using function `nextRandom(int seed)`:

```
  int t = (seed % 127773) * 16807 –
          (seed / 127773) * 2836;
  return (t > 0)? t : t + 0x7fffffff;
```

On each iteration a thread updates, with probability *S*, a shared generator under a mutual exclusion lock, else it updates its own local generator, without a lock. This results in short-duration locked regions, minimizing extraneous effects when threads are preempted while holding locks. The randomness of the function serves two purposes: it is used in deciding whether to lock or not (it is a good enough generator for current purposes), and also makes code within loops impossible to trivially optimize away.

Four kinds of locks were compared: *Builtin*, using synchronized blocks; *Mutex*, using a simple `Mutex` class like that illustrated in section 4; *Reentrant*, using `ReentrantLock;` and *Fair*, using `ReentrantLock` set in its "fair" mode. All tests used build 46 (approximately the same as beta2) of the Sun J2SE1.5 JDK in "server" mode. Test programs performed 20 uncontended runs before collecting measurements, to eliminate warm-up effects. Tests ran for ten million iterations per thread, except Fair mode tests were run only one million iterations.

Tests were performed on four x86-based machines and four UltraSparc-based machines. All x86 machines were running Linux using a RedHat NPTL-based 2.4 kernel and libraries. All UltraSparc machines were running Solaris-9. All systems were at most lightly loaded while testing. The nature of the tests did not demand that they be otherwise completely idle. The "4P" name reflects the fact a dual hyperthreaded (HT) Xeon acts more like a 4-way than a 2-way machine. No attempt was made to normalize across the differences here. As seen below, the relative costs of synchronization do not bear a simple relationship to numbers of processors, their types, or speeds.

Table 1 Test Platforms

Name	*Processors*	*Type*	*Speed (Mhz)*
1P	1	Pentium3	900
2P	2	Pentium3	1400
2A	2	Athlon	2000
4P	2 HT	Pentium4/Xeon	2400
1U	1	UltraSparc2	650
4U	4	UltraSparc2	450
8U	8	UltraSparc3	750
24U	24	UltraSparc3	750

5.1 Overhead

Uncontended overhead was measured by running only one thread, subtracting the time per iteration taken with a version setting S=0 (zero probability of accessing shared random) from a run with S=1. Table 2 displays these estimates of the per-lock overhead of synchronized code over unsynchronized code, in nanoseconds. The Mutex class comes closest to testing the basic cost of the framework. The additional overhead for Reentrant locks indicates the cost of recording the current owner thread and of error-checking, and for Fair locks the additional cost of first checking whether the queue is empty.

Table 2 also shows the cost of `tryAcquire` versus the "fast path" of a built-in lock. Differences here mostly reflect the costs of using different atomic instructions and memory barriers across locks and machines. On multiprocessors, these instructions tend to completely overwhelm all others. The main differences between Builtin and synchronizer classes are apparently due to Hotspot locks using a compareAndSet for both locking and unlocking, while these synchronizers use a compareAndSet for acquire and a `volatile` write (i.e., with a memory barrier on multiprocessors, and reordering constraints on all processors) on release. The absolute and relative costs of each vary across machines.

At the other extreme, Table 3 shows per-lock overheads with S=1 and running 256 concurrent threads, creating massive lock contention. Under complete saturation, barging-FIFO locks have about an order of magnitude less overhead (and equivalently greater throughput) than Builtin locks, and often two orders of magnitude less than Fair locks. This demonstrates the effectiveness of the barging-FIFO policy in maintaining thread progress even under extreme contention.

Table 2 Uncontended Per-Lock Overhead in Nanoseconds

Machine	*Builtin*	*Mutex*	*Reentrant*	*Fair*
1P	18	9	31	37
2P	58	71	77	81
2A	13	21	31	30
4P	116	95	109	117
1U	90	40	58	67
4U	122	82	100	115
8U	160	83	103	123
24U	161	84	108	119

Table 3 Saturated Per-Lock Overhead in Nanoseconds

Machine	*Builtin*	*Mutex*	*Reentrant*	*Fair*
1P	521	46	67	8327
2P	930	108	132	14967
2A	748	79	84	33910
4P	1146	188	247	15328
1U	879	153	177	41394
4U	2590	347	368	30004
8U	1274	157	174	31084
24U	1983	160	182	32291

Table 3 also illustrates that even with low internal overhead, context switching time completely determines performance for Fair locks. The listed times are roughly proportional to those for blocking and unblocking threads on the various platforms.

Additionally, a follow-up experiment (using machine 4P only) shows that with the very briefly held locks used here, fairness settings had only a small impact on overall variance. Differences in termination times of threads were recorded as a coarse-grained measure of variability. Times on machine 4P had standard deviation of 0.7% of mean for Fair, and 6.0% for Reentrant. As a contrast, to simulate long-held locks, a version of the test was run in which each thread computed 16K random numbers while holding each lock. Here, total run times were nearly identical (9.79s for Fair, 9.72s for Reentrant). Fair mode variability remained small, with standard deviation of 0.1% of mean, while Reentrant rose to 29.5% of mean.

5.2Throughput

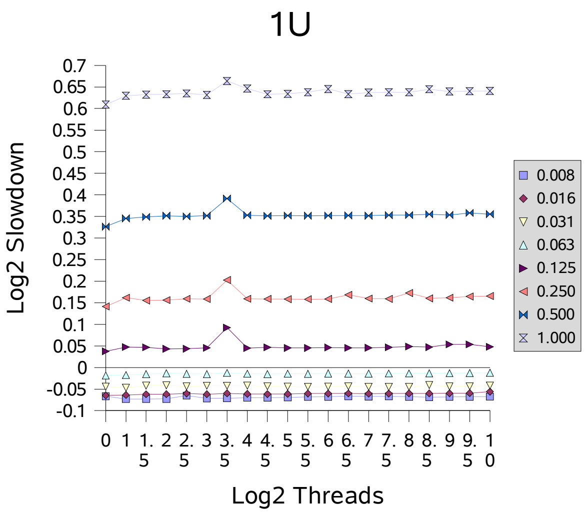
Usage of most synchronizers will range between the extremes of no contention and saturation. This can be experimentally examined along two dimensions, by altering the contention probability of a fixed set of threads, and/or by adding more threads to a set with a fixed contention probability. To illustrate these effects, tests were run with different contention probablilities and numbers of threads, all using Reentrant locks. The accompanying figures use a *slowdown* metric:

$$slowdown = \frac{t}{S \cdot b \cdot n + (1 - S) \cdot b \cdot max(1, \frac{n}{p})}$$

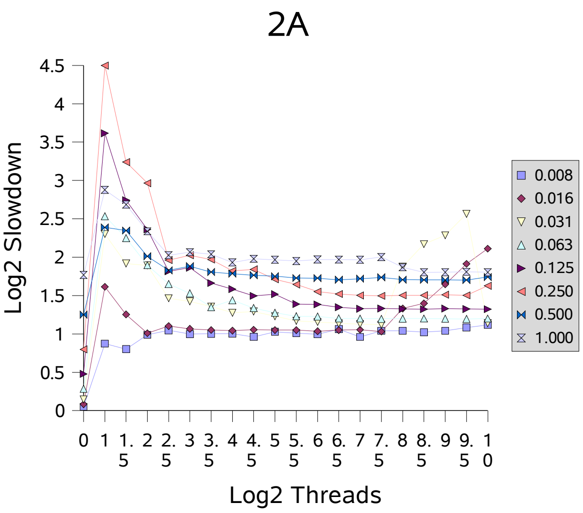
Here, *t* is the total observed execution time, *b* is the baseline time for one thread with no contention or synchronization, *n* is the number of threads, *p* is the number of processors, and S remains the proportion of shared accesses. This value is the ratio of observed time to the (generally unattainable) ideal execution time as computed using Amdahl's law for a mix of sequential and parallel tasks. The ideal time models an execution in which, without any synchronization overhead, no thread blocks due to conflicts with any other. Even so, under very low contention, a few test results displayed very small speedups compared to this ideal, presumably due to slight differences in optimization, pipelining, etc., across baseline versus test runs.

The figures use a base 2 log scale. For example, a value of 1.0 means that a measured time was twice as long as ideally possible, and a value of 4.0 means 16 times slower. Use of logs ameliorates reliance on an arbitrary base time (here, the time to compute random numbers), so results with different base computations should show similar trends. The tests used contention probabilities from 1/128 (labelled as "0.008") to 1, stepping in powers of 2, and numbers of threads from 1 to 1024, stepping in half-powers of 2.

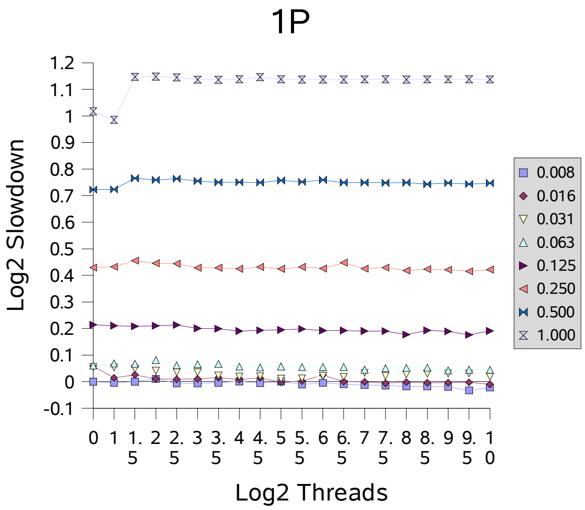
On uniprocessors (1P and 1U) performance degrades with increasing contention, but generally not with increasing numbers of threads. Multiprocessors generally encounter much worse slowdowns under contention. The graphs for multiprocessors show an early peak in which contention involving only a few threads usually produces the worst relative performance. This reflects a transitional region of performance, in which barging and signalled threads are about equally likely to obtain locks, thus frequently forcing each other to block. In most cases, this is followed by a smoother region, as the locks are almost never available, causing access to resemble the near-sequential pattern of uniprocessors; approaching this sooner on machines with more processors. Notice for example that the values for full contention (labelled "1.000") exhibit relatively worse slowdowns on machines with fewer processors.



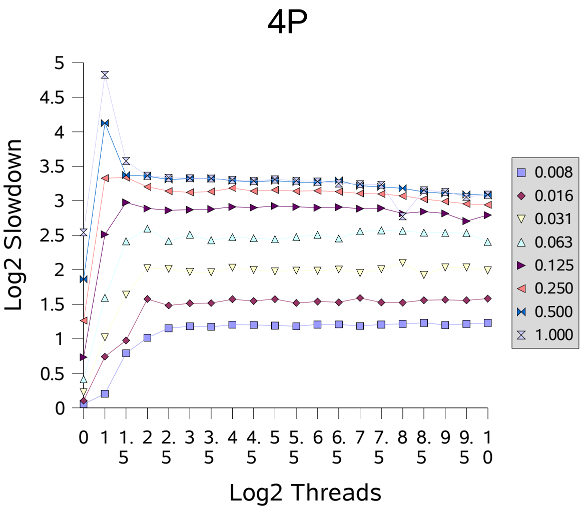
1U



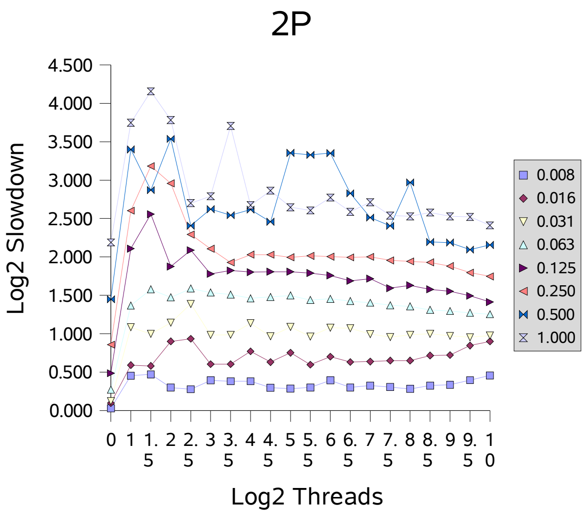
2A



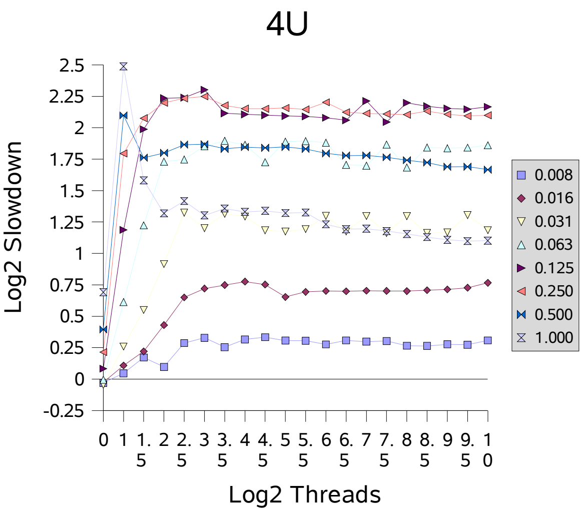
1P



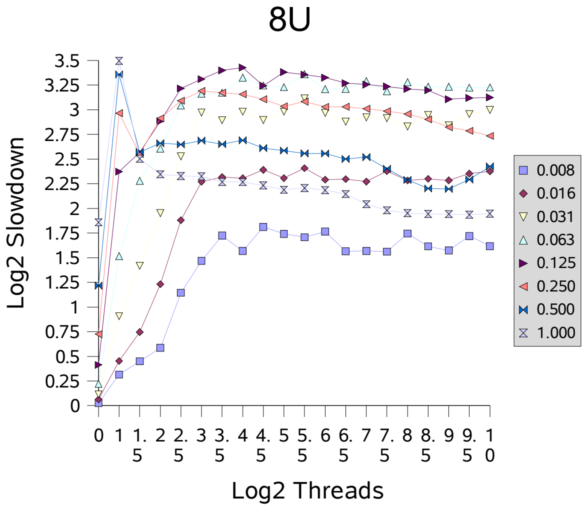
4P

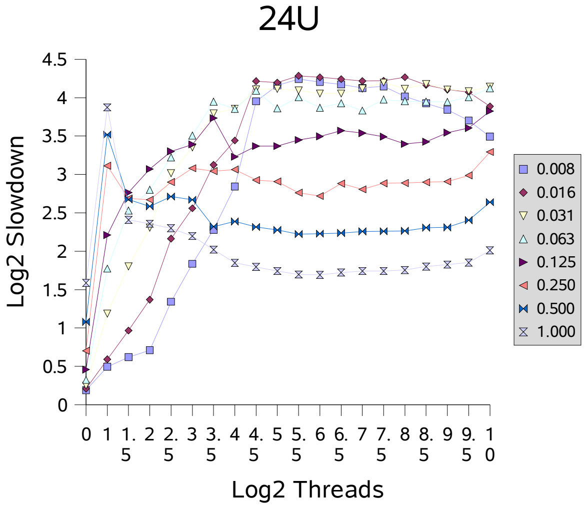


2P



4U





On the basis of these results, it appears likely that further tuning of blocking (`park`/`unpark`) support to reduce context switching and related overhead could provide small but noticeable improvements in this framework. Additionally, it may pay off for synchronizer classes to employ some form of adaptive spinning for briefly-held highly-contended locks on multiprocessors, to avoid some of the flailing seen here. While adaptive spins tend to be very difficult to make work well across different contexts, it is possible to build custom forms of locks using this framework, targetted for specific applications that encounter these kinds of usage profiles.

6. CONCLUSIONS

As of this writing, the java.util.concurrent synchronizer framework is too new to evaluate in practice. It is unlikely to see widespread usage until well after final release of J2SE1.5, and there will surely be unexpected consequences of its design, API, implementation, and performance. However, at this point, the framework appears successful in meeting the goals of providing an efficient basis for creating new synchronizers.

7. ACKNOWLEDGMENTS

Thanks to Dave Dice for countless ideas and advice during the development of this framework, to Mark Moir and Michael Scott for urging consideration of CLH queues, to David Holmes for critiquing early versions of the code and API, to Victor Luchangco and Bill Scherer for reviewing previous incarnations of the source code, and to the other members of the JSR166 Expert Group (Joe Bowbeer, Josh Bloch, Brian Goetz, David Holmes, and Tim Peierls) as well as Bill Pugh, for helping with design and specifications and commenting on drafts of this paper. Portions of this work were made possible by a DARPA PCES grant, NSF grant EIA-0080206 (for access to the 24way Sparc) and a Sun Collaborative Research Grant.

8. REFERENCES

[1] Agesen, O., D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. *ACM OOPSLA Proceedings,* 1999.

[2] Andrews, G. *Concurrent Programming*. Wiley, 1991.

[3] Bacon, D. Thin Locks: Featherweight Synchronization for Java. *ACM PLDI Proceedings*, 1998.

[4] Buhr, P. M. Fortier, and M. Coffin. Monitor Classification, *ACM Computing Surveys*, March 1995.

[5] Craig, T. S. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, Feb. 1993.

[6] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*, Addison Wesley, 1996.

[7] Holmes, D. *Synchronisation Rings*, PhD Thesis, Macquarie University, 1999.

[8] Magnussen, P., A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. *8th Intl. Parallel Processing Symposium*, Cancun, Mexico, Apr. 1994.

[9] Mellor-Crummey, J.M., and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Computer Systems*, February 1991

[10] M. L. Scott and W N. Scherer III. Scalable Queue-Based Spin Locks with Timeout. *8th ACM Symp. on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.

[11] Sun Microsystems. *Multithreading in the Solaris Operating Environment*. White paper available at http://wwws.sun.com/software/solaris/whitepapers.html 2002.

[12] Zhang, H., S. Liang, and L. Bak. *Monitor Conversion in a Multithreaded Computer System*. United States Patent 6,691,304. 2004.