Concurrency Support in Java

Daniel Solano Gómez

27 January 2011

Overview

Where we are...

Runnable

```
public interface Runnable {
   void run();
}
```

- Provides the primary abstraction for a task
- Allows definition of concurrent tasks without subclassing `Thread`

Callable

```
public interface Callable<V> {
  V call() throws Exception;
}
```

A `Callable` is similar to `Runnable`, but it returns a result and can throw checked exceptions.

Future

- A `Future` represents the result of an asynchronous computation.
- Retrieve the result with the blocking `get` method.
- It is possible to cancel the task.
- Futures are normally created indirectly from `Runnable` or `Callable` objects.
- These are the same as the futures provided in Clojure 1.1.

Where we are...

Executors

```
public interface Executor {
  void execute(Runnable command);
}
```

`Executor` is the preferred abstraction for task execution in Java.

Creating executors

The utility class `Executors` provides factory methods to create preconfigured executors.

- `newCachedThreadPool()` reuses available threads, creating new threads as necessary
- `newFixedThreadPool()` uses a fixed number of threads and an unbounded queue
- `newSingleThreadedExecutor()` is guaranteed to be single threaded and uses an unbounded queue
- `newScheduledThreadPool()` is a fixed-size thread pool that supports delayed and periodic tasks, similar to `Timer`

ExecutorService

All executors provided by Java implement `ExecutorService`, an extension of `Executor` that:

- Manages termination of the executor
- Creates `Future` objects from submitted tasks
- Provides methods for invoking collections of tasks

More on executors

- Most executors created by Java are subclasses of `ThreadPoolExecutor`, which allows a large degree of tuning.
- An executor can act like a blocking queue through the use of a `ExecutorCompletionService`.

Executor demonstrations

It's demo time!

Where we are. . .

Features of atomic variables

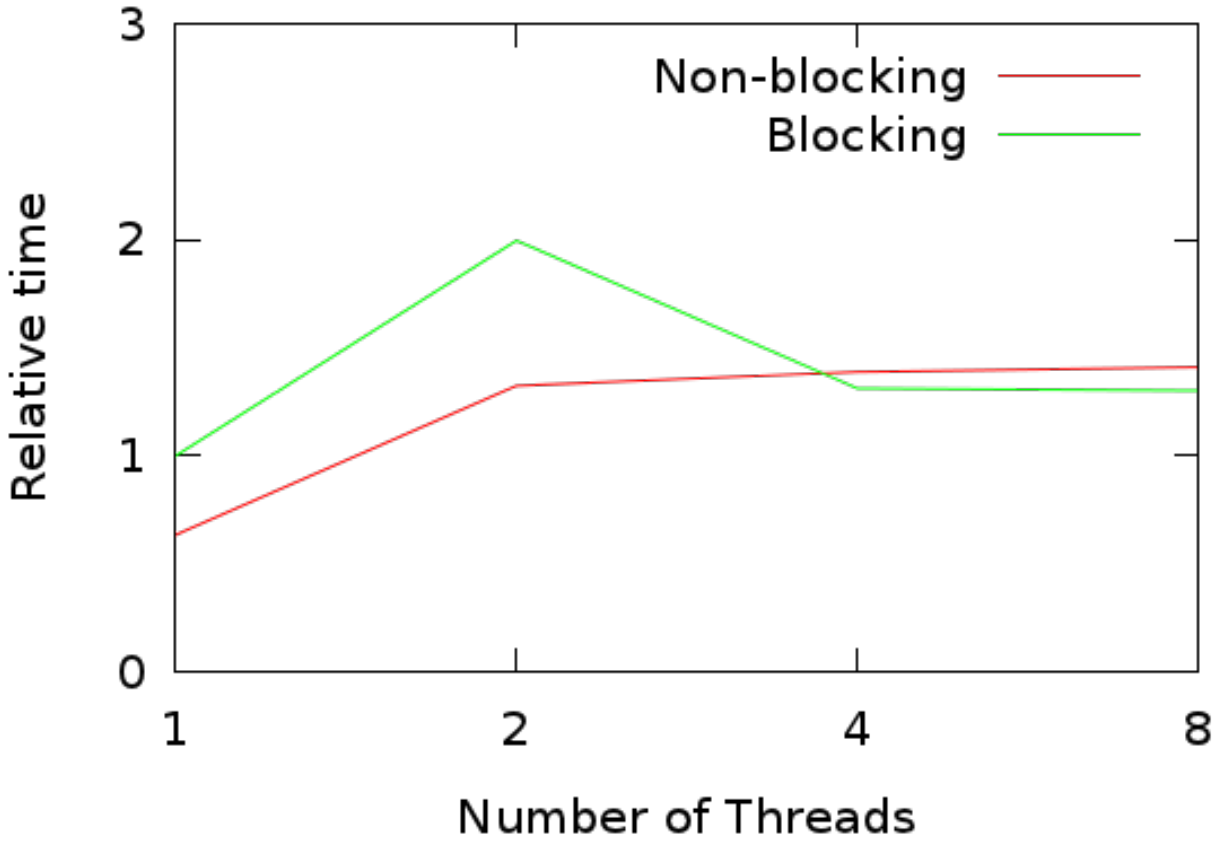
- Operations have specific memory guarantees, generally equivalent to `volatile` variables
- Allow use of `compareAndSet` to exploit instructions available on modern processors
- Provide atomic equivalents to `++` and `--` operators
- Available for boolean, integer, long, and reference types
- Available for integer, long, and reference arrays

Using atomic variables

- Useful for implementing non-blocking data structures
- Only for state confined to a single variable
- Not a general replacement for locking, best for little or no contention

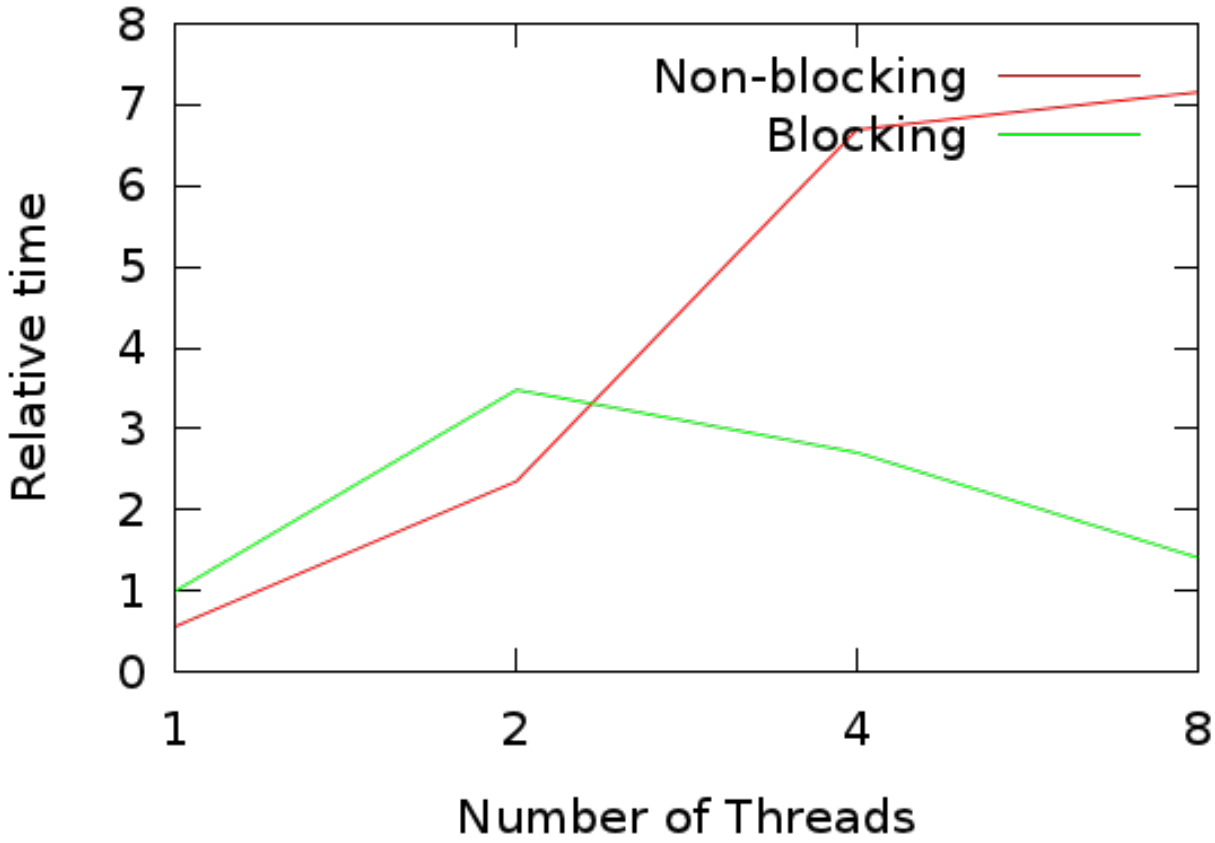
Xorshift PRNG performance

Dual core



Xorshift PRNG performance

Quad core



Where we are. . .

Java collections and thread safety

- Original Java `Hashtable` and `Vector` are synchronized
- Collections introduced in Java 1.2 are generally not thread-safe
- `Collections.synchronizedXxx` methods can make collections thread-safe
- Compound operations must be externally synchronized
- Beware of `ConcurrentModificationException`

Concurrent collections

- Java SE 5 introduced new thread-safe collections
- Offer specific memory model guarantees
- Performance and consistency guarantees may differ from unsafe and synchronized collections
- Iterators are weakly-consistent and do not throw `ConcurrentModificationException`
- Bulk operations are not atomic
- Do not support `null` elements
- `size` and `isEmpty` may be approximations

ConcurrentHashMap

- Implements `ConcurrentMap`, an extension of `Map` with compare-and-set-like functions
- Drop-in replacement for `Hashtable` for thread-safety
- Reads require no locking
- Adjustable write concurrency
- Impossible to lock entire table

ConcurrentSkipListMap

- Implements `ConcurrentNavigableMap`, a combination of `ConcurrentMap` and `NavigableMap`
- Returned `Map.Entry` objects are snapshots and do not support `setValue`
- `size` is not constant-time

ConcurrentSkipListSet

- Implements NavigableSet
- size method is not constant-time

ConcurrentLinkedQueue

- Implements unbounded `Queue` based on linked nodes
- `size` method is not constant-time

Where we are. . .

COW Collections

- `CopyOnWriteArrayList` and `CopyOnWriteArraySet`
- Similar behaviour to Clojure's persistent data structures
- Reads are always consistent
- However, writes are expensive and require full copies

Blocking queues

- Facilitate consumer-produce pattern
- Producers may block if queue is full
- Consumers may block if queue is empty
- Useful in combination with thread pools

Basic blocking queues

`LinkedBlockingQueue` and `ArrayBlockingQueue`

- FIFO queues analogous to `ArrayList` and `LinkedList`
- Thread-safe and better performance than a synchronized `List`
- `ArrayBlockingQueue` is bounded and can support fairness
- `LinkedBlockingQueue` is optionally bounded

More blocking queues

- `LinkedBlockingDeque`, a thread-safe `Deque` backed by a linked list
- `PriorityBlockingQueue`, a thread-safe counterpart to `PriorityQueue`
- `DelayQueue`, a blocking queue of `Delayed` elements, like a priority queue sorted by expiration
- `SynchronousQueue`, a zero-capacity queue used to coordinate handoffs, with optional fairness

Where we are. . .

CountDownLatch

- General purpose latch is initialised with a count
- `countDown` method decrements count
- `await` method will block until count is zero
- Not reusable
- Good for waiting for one-time events or waiting for multiple parties to be ready to proceed

Semaphore

- A counting semaphore with a number of virtual permits
- Binary semaphore is a mutex
- Permits are not tied to objects or threads
- Ideal for implementing resource pools

CyclicBarrier

- Coordinates a fixed number of threads
- All threads come together at the barrier at the same time
- Reusable
- Good for simulations, ensuring that one step of the simulation is complete before proceeding to next step

Exchanger

- Like a two-party barrier, but includes swapping objects
- Example use is swapping of buffers between a producer and consumer

Where we are. . .

The `Lock` interface

- Allows alternatives to intrinsic locking
- Implementations must guarantee same memory-visibility semantics
- Offers choice of unconditional, polled, timed, and interruptible lock acquisition
- Allows non-block-structured blocking, requires explicit `lock` and `unlock`
- Allows multiple wait conditions

Intrinsic locking example

```
Object lock = new Object();
...
synchronized(lock) {
  // do something
}
```

Lock object example

```
Object lock = new ReentrantLock();
...
lock.lock();
try {
   // do something
} finally {
   lock.unlock();
}
```

ReentrantLock

- Reentrant
- Offers same memory semantics as a synchronized block
- Allows fair lock acquisition

Read-write locks

`ReadWriteLock` interface

- Allows either multiple readers or a single writer
- Separate read and write locks
- Allows implementation flexibility

`ReentrantReadWriteLock` implementation

- Reentrant for both locks
- Allows fairness
- Allows downgrades from writer to reader
- Best when locks are held for long times

Where we are. . .

Expected in JDK 7

Subject to change

- Fork/Join framework for recursive breakdown of tasks
- `TransferQueue` allows producer to wait for consumers
- `Phaser`, a flexible combination of `CyclicBarrier` with `CountDownLatch`
- `ThreadLocalRandom`

Where we are. . .

Further reading

- *Java Concurrency in Practice*, Brian Goetz et al.
- Java SE 6.0 SDK Documentation
- The Java™ Tutorials: Concurrency