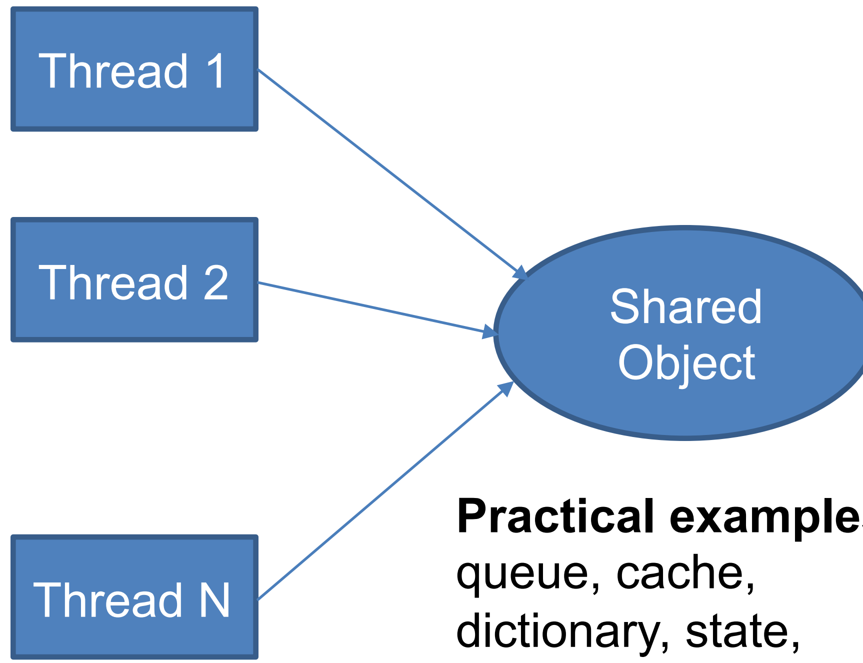
# Wait for your fortune without blocking!

# Why concurrency?

- Key motivators
  - **Performance**
  - **Scalability**

- Unless you need both, don't bother with concurrency:
  - Write single-threaded
  - Scale by running multiple copies of code

Share nothing and sleep well




**Practical examples:** queue, cache, dictionary, state, statistics, log, etc

What is **blocking**?

What is **non-blocking** *algorithm*?

# Blocking (aka locking)

- *Semi*-formally

> An algorithm is called **non-blocking (lock-free)**
> if suspension of any thread cannot
> cause suspension of another thread

- In Java *practice* **non-blocking** algorithms
  - just read/write **volatile** variable and/or use
  - **j.u.c.a.AtomicXXX** classes with **compareAndSet** and other methods

- **Blocking** algorithms (with **locks**) use
  - **synchronized (…)** which produces monitorEnter/monitorExit instrs
  - **j.u.c.l.Lock** lock/unlock methods
  - **NOTE:** *You can code blocking without realizing it*

# Toy problem solved with locks

```
// sort of a queue, but does not actually queue
public class DataHolder<T> {
1   private T value; // shared state!

    // updates current value
2   public synchronized void updateValue(T newValue) {
        value = newValue;
    }

    // removes current value to publish it somewhere
3   public synchronized T removeValue() {
        T oldValue = value;
        value = null;
        return oldValue;
    }
```

**Locks** are *the easiest* way to make your object *linearizable* (aka **thread-safe**)

Just protect **all** operations on a **shared** state with the same lock (or monitor)

What is **waiting** [for condition]?

What is **waiting** *operation*?

sometimes aka “blocking”, too ☹

# Waiting for condition

- Formally

**Partial function**
from *object state set* **X** to *result set* **Y**
is defined only on a subset of **X'** of **X**.
Method invocation can complete only
when object state is in **X'**
(when condition is satisfied).

- For example, let's implement *partial* **takeValue** operation that is defined only when there is **value** != **null** in **DataHolder**
- *Waiting is orthogonal to blocking/non-blocking*

# Waiting is easy with monitors

```
// updates current value
public synchronized void updateValue(T newValue) {
    value = newValue;
    notifyAll();
}
```

1

If in doubt, always use **notifyAll** instead of **notify** (but can use **notify** *here*)

```
// takes current value, waiting until it is updated
public synchronized T takeValue() throws InterruptedException {
    while (value == null) wait();
    T oldValue = value;
    value = null;
    return oldValue;
}
```

2

This is **waiting** code (**partial function**): it is only defined when **value** != **null**

This is code with **locks** (**synchronized**): suspension of one thread on any of these lines causes suspension of all other threads that attempt to do any operation

# Why go non-blocking (aka lock-free)?

- **Performance**
  - Locking is expensive when contended
  - Actually, *context switches* are expensive
- **Dead-lock avoidance**
  - Too much locking can get you into trouble
  - Sometimes it is just easier to get rid of locks

DEVEXPERTS

# Let's go lock-free

1

```java
private final AtomicReference<T> valueRef =
        new AtomicReference<>();
```

2

```java
// updates current value
public void updateValue(T newValue) {
    valueRef.set(newValue);
}
```

3, 4

```java
// removes current value to publish it somewhere
public T removeValue() {
    while (true) {
        T oldValue = valueRef.get();
        if (oldValue == null) return null;
        if (valueRef.compareAndSet(oldValue, null))
            return oldValue;
    }
}
```

# Lock-free partial operations (waiting aka parking)

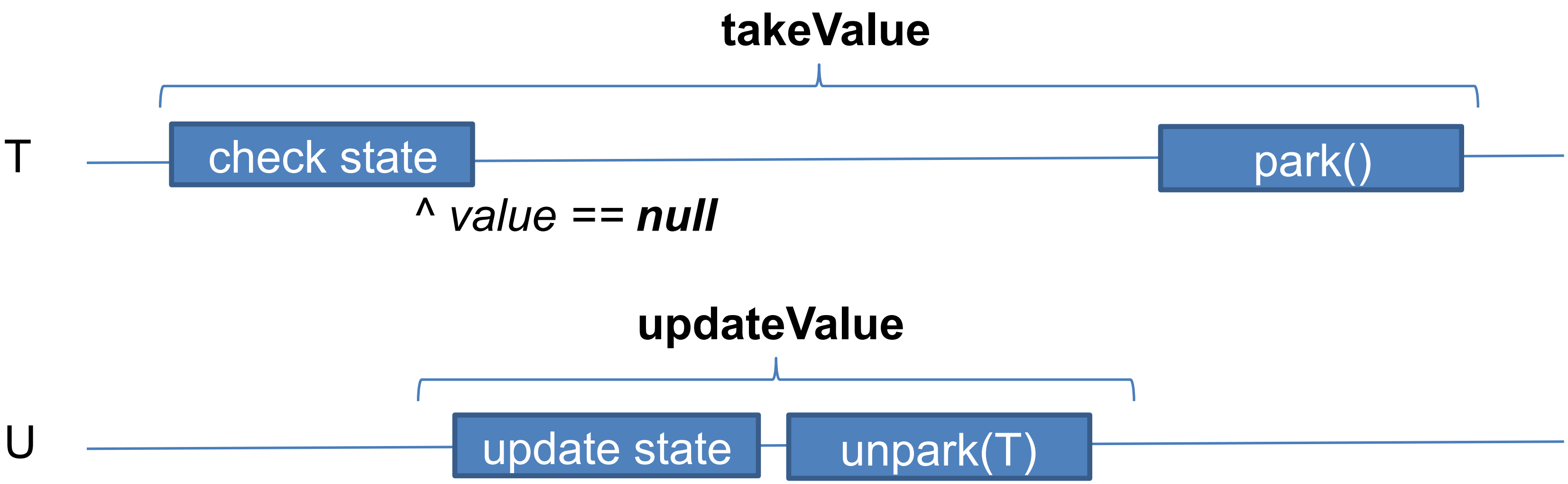
```java
// Let's start with a single taker thread (uses takeValue)
public class TakerThread<T> extends Thread {
    // takes current value, waiting until it is updated
    private T takeValue() throws InterruptedException {
        assert Thread.currentThread() == this;
        while (true) {
            T oldValue = valueRef.get();
            if (oldValue == null) {
                LockSupport.park(); // This is lock-free waiting
                // ... with an appropriate interrupted idiom
                if (interrupted())
                    throw new InterruptedException();
                continue;
            }
            if (valueRef.compareAndSet(oldValue, null))
                return oldValue;
        }
}
```

# Lock-free wakeup (aka unparking)

```java
// updates current value (can be called from any thread)
public void updateValue(T newValue) {
    valueRef.set(newValue);
    LockSupport.unpark(this);
}
```

- **Note:** in lock-free code order is important (first update, then unpark)
- Updaters are 100% wait-free (never locked out by other threads)
- Taker (**takeValue**) can get *starved* in CAS loop, but still non-blocking (formally, *lock-free*)

# Park/unpark magic




**LockSupport.unpark(T)**: “Makes available the permit for the given thread, if it was not already available. If the thread was blocked on park then it will unblock. *Otherwise, its next call to park is guaranteed not to block.*”

# Lock-free waiting from different/multiple threads

- Must maintain **wait queue** of threads in a lock-free way
  - This is a non-trivial

- **j.u.c.l.AbstractQueuedSynchronizer** is a good place to start
- It is used to implement a number of **j.u.c.*** classes:
  - **ReentrantLock**
  - **ReentrantReadWriteLock**
  - **Semaphore**
  - **CountDownLatch**

- You can use it to for your own needs, too

# Anatomy of AbstractQueuedSynchronizer

1 private state

```
int state; // optionally use for state
wait queue <Node>; // nodes reference threads
```

almost separate aspects

2 state access

```
int getState()
void setState(int newState)
boolean compareAndSetState(int expect, int update)
```

3 override

```
boolean tryAcquire(int arg)
boolean tryRelease(int arg)
int tryAcquireShared(int arg)
boolean tryReleaseShared(int arg)
```

4 use

```
void acquire(int arg)
void acquireInterruptibly(int arg)
boolean tryAcquireNanos(int arg, long nanos)
boolean release(int arg)
void acquireShared(int arg)
// and others
```

# Anatomy of AbstractQueuedSynchronizer (2)

1

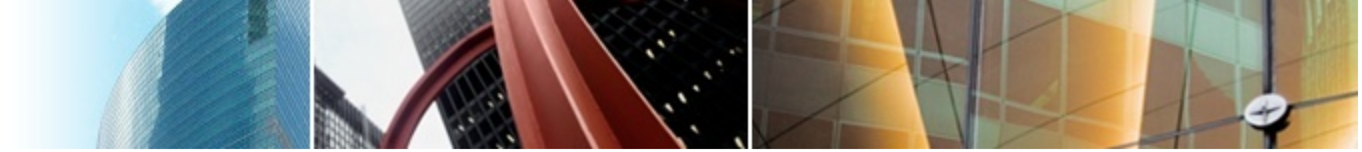
```java
public final void acquireInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (!tryAcquire(arg))
        doAcquireInterruptibly(arg);
}
```

2 adds to wait queue

3

```java
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

4 unlinks from wait queue



# Our own synchronizer

```
private class Sync extends AbstractQueuedSynchronizer {
    @Override
    protected boolean tryAcquire(int arg) {
1       T oldValue = valueRef.get();
        if (oldValue == null)
            return false;
        if (!valueRef.compareAndSet(oldValue, null))
            return false;
        // This is a kludge to return result from this method
2       results[arg] = oldValue;
        return true;
    }

    @Override
    protected boolean tryRelease(int arg) {
3       return true; // object is always "released", wake up next
    }
}
```

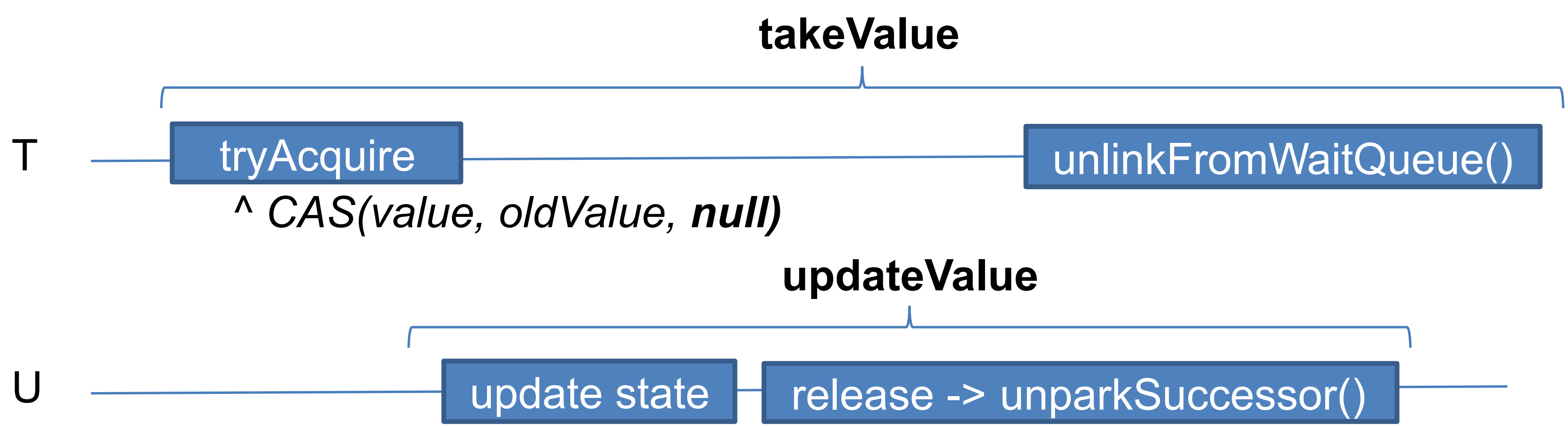
# Use synchronizer to implement notify/wait

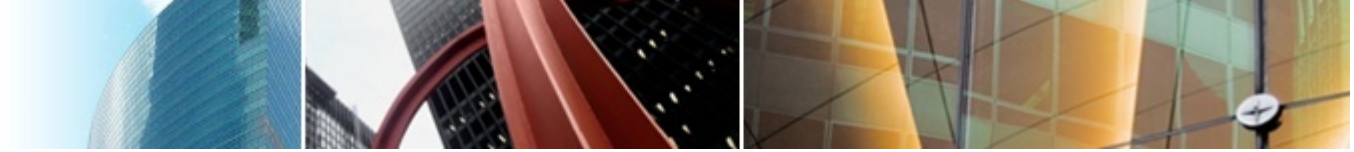
```
private final Sync sync = new Sync();

// updates current value
public void updateValue(T newValue) {
    valueRef.set(newValue);
1   sync.release(0); // we don't use arg here
}

// takes current value, waiting until it is updated
private T takeValue() throws InterruptedException {
    int arg = reserveResultsSlot(); // kludge needed
2   sync.acquireInterruptibly(arg); // ... to return result
    if (valueRef.get() != null) // must double check
3       sync.release(0); // ... or else might loose unpark
    return releaseResultsSlot(arg);
}
```

# Why double check? (more internals)

*Simplified code*

```
void doAcquireXXX(int arg) {
    addToWaitQueue();
    for (;;) {
        if (isFirstInQueue() && tryAcquire(arg)) {
            unlinkFromWaitQueue(); return;
        }
        doPark();
    }
}
```





DEVEXPERTS

# Naïve “performance improvement”

```
// updates current value
public void updateValue(T newValue) {
1       T oldValue = valueRef.getAndSet(newValue);
        if (oldValue == null)
2           sync.release(0); // we don't use arg here
}
```

- The idea is to unpark just *one* thread when setting value for the first time only (and avoid unparking on subsequent updates)
- **DOES NOT WORK *SUBTLY*: updateValue** may cause concurrent **tryAcquire** to fail on CAS and park, but we don’t call **release** in this case anymore, so it will never unpark

.

# Corrected Sync.tryAcquire method

```
protected boolean tryAcquire(int arg) {
    while (true) {
1       T oldValue = valueRef.get();
        if (oldValue == null)
            return false;
2       if (!valueRef.compareAndSet(oldValue, null))
            continue; // retry CAS (not fail!)
        // This is a kludge to return result from this method
        results[arg] = oldValue;
        return true;
    }
}
```

- Use CAS-loop idiom to retry in the case of contention
- Optimal version in terms of context switching

# This is optimal, but not fair!

- Let's take a closer look at **AQS.acquireXXX**

```
public final void acquireInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (!tryAcquire(arg))
        doAcquireInterruptibly(arg);
}
```

- Thread might jump ahead of the queue
  - Good or bad? – depends on the problem being solved

# Make it fair (if needed)

```java
protected boolean tryAcquire(int arg) {
    while (true) {
        T oldValue = valueRef.get();
        if (oldValue == null)
            return false;
        if (hasQueuedPredecessors())
            return false; // be fair!
        if (!valueRef.compareAndSet(oldValue, null))
            continue; // retry CAS (not fail!)
        // This is a kludge to return result from this method
        results[arg] = oldValue;
        return true;
    }
}
```

DEVEXPERTS

# Conclusion

- Waiting can be implemented in a **non-blocking** way
  - **Recap non-blocking:** suspension of any thread (*on any line of code*) cannot cause suspension of another thread
  - **Bonus:** context switch only when really need to wait & wakeup
  - **Fairness:** is an optional aspect of waiting

- **AbstractQueuedSynchronizer**
  - is designed for writing custom lock-like classes
  - but can be repurposed as a ready wait-queue impl for other cases

Lock-free programming is extremely bug-prone

# Thank you

**Any questions?**

Slides are available at **elizarov.livejournal.com**
Twitter at @relizarov