Aside: Volatile Keyword

- C, C++, Java, and C# all have the "`volatile`" keyword

- Keyword has slightly different meaning in all languages!

- More danger: you may think you know what `volatile` means but you are mistaken

Aside: History of Volatile

- C introduced `volatile` "type qualifier" keyword

- Meant to accommodate memory mapped devices

- Keyword warns that variable's value "can be changed by means outside this code"

Example, I

```
/* int foo declared mapped to hardware device register */

void read() {
    foo = 0;

    while (foo != 1)
        ;     /* spin-wait for foo to become 1 */

    ... code that assumes foo == 1 ...
}
```

Q: What's going on in the loop?

A: `foo` represents a hardware device register whose value might be changed asynchronously by the hardware

Example, II

- Optimizing compiler will emit same assembly code as for:

```
void read() {
    foo = 0;

    while (1)
        ;     /* infinite loop */

    ...
}
```

- Loops forever without ever testing `foo`'s value

Example, III

- Declaring `foo` to be `volatile` fixes the problem:

```
static volatile int foo;

void read() {
    foo = 0;

    while (foo != 1)
        ;     /* spin-wait for foo to become 1 */

    ... code that assumes foo == 1 ...
}
```

- `volatile` keyword tells compiler always to emit instructions to read/write `foo`'s memory location

Summary of Volatile

- `volatile` keyword is:

- Hard to use correctly
- Easy to use incorrectly
- Subtly different in different languages
- Unnecessary (for Java application programming)

- Summary: avoid `volatile` unless you really understand what you're doing and you're sure there is no other way to do it

Library Support for Concurrency

1. Atomic classes
2. Semaphore, read/write locks, Barrier, Countdown latch, Exchanger, SynchronousQueue
3. Collection classes, concurrent data structures
4. Task management (thread pools)
5. Concurrency patterns

Summary

- Java = base language +
HUGE library

- Slogan for conceptual cleanliness ...
"less is more"

- Slogan for Java language/library
designers??? ... "more is more"

- In their defense: why NOT accumulate
debugged classes?

- But ... library is in serious need of an
intelligent table of contents; i.e., advice for
which library classes to use, when, and how

Reminder: Wrapper Classes

- Java has 8 primitives types: `int`, `long`, `boolean`, etc.

- Recall that each primitive type has a corresponding *wrapper class*

- E.g., `Integer` for `int`, `Boolean` for `boolean`, etc.

- Allows treating primitive variable as an object

- Houses useful static methods; e.g., `Integer.parseInt`

©2014 Stevens Institute of Technology

Atomic Classes, I

- 3 of 8 primitive types (`int`, `long`, `boolean`) have corresponding **atomic classes**

- They are `AtomicInteger`, `AtomicLong`, `AtomicBoolean`

- Others too ... `AtomicReferenceArray` etc.

- These are part of `java.util.concurrent.atomic`

- Methods of atomic classes each behave atomically

©2014 Stevens Institute of Technology

Atomic Classes, II

- All atomic classes have methods `get`, `set`, `getAndSet`, `compareAndSet`

- `compareAndSet(expect, update)` sets new value (“`update`”) only if old value matches expected value (“`expect`”)

- Just like test-and-set machine instruction

- `AtomicInteger` and `AtomicLong` also have

```
incrementAndGet      ++x
getAndIncrement      x++
decrementAndGet      --x
getAndDecrement      x--
```

- Other methods as well: `addAndGet`, etc.

Utility of Atomic Classes

1. Simple compound operations that cannot be done with a `volatile` variable

E.g., `x++` requires a read AND a write

2. Building blocks for more complicated classes

Atomic Array Classes

- `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReferenceArray` provide atomic access to elements of array

- Other classes as well

Other Synchronization Classes

- Semaphore — we've seen this before
- Read/write locks — we've seen this before
- Barrier — we've seen this in Assignment 3
- Countdown latch — specialization of barrier
- Exchanger — even more specialized
- SynchronousQueue — not really a queue
- Locks are part of `java.util.concurrent.locks`,
rest are part of `java.util.concurrent`

Semaphore, I

- Initial value of semaphore called "number of permits:"

```
Semaphore(long permits)
Semaphore(long permits, boolean fair)
```

- "`fair`" access means FCFS — goal is to avoid starvation

Semaphore, II

- P and V called "acquire" and "release:"

```
acquire() throws InterruptedException
acquireUninterruptibly()
acquire(long permits) throws InterruptedException
acquireUninterruptibly(long permits)

release()
release(long permits)
```

- "`permits`" argument adds/deletes N to semaphore value

- There are various "`tryAcquire`" methods as well

Interruptibility

- You will see methods named “`fooInterruptibly`” and “`fooUninterruptibly`”

- Must be aware of what the default behavior is

Reminder: Moded Locks

- Recall: purpose of adding **modes** to lock is to increase concurrency

- Separate threads can simultaneously hold lock in “compatible” modes; e.g., read-read

- BY FAR most common set of modes is read & write

- There are other “fancy” mode sets

- (SIX mode is very elaborate and useful only for hierarchical data)

©2014 Stevens Institute of Technology

Read/Write Locks

Interface:

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

Class:

```
public class ReentrantReadWriteLock
             implements ReadWriteLock {
    public ReentrantReadWriteLock();
    public ReentrantReadWriteLock(boolean fair);
    public Lock readLock();
    public Lock writeLock();
}
```

Read/Write Lock Semantics, I

- Reentrant: lock can be multiply acquired by same thread

- OK to downgrade, like so:

  1. Acquire write lock
  2. Acquire read lock
  3. Drop write lock

- Upgrade NOT OK — must first drop read lock, then try to acquire write lock

©2014 Stevens Institute of Technology

Read/Write Lock Semantics, II

- Unusual API
- Uses **nested classes** of `ReentrantReadWriteLock`: `ReentrantReadWriteLock.ReadLock` and `ReentrantReadWriteLock.WriteLock`
- How to perform the 4 operations:

```
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
rwl.readLock().lock();
rwl.readLock().unlock();
rwl.writeLock().lock();
rwl.writeLock().unlock();
```

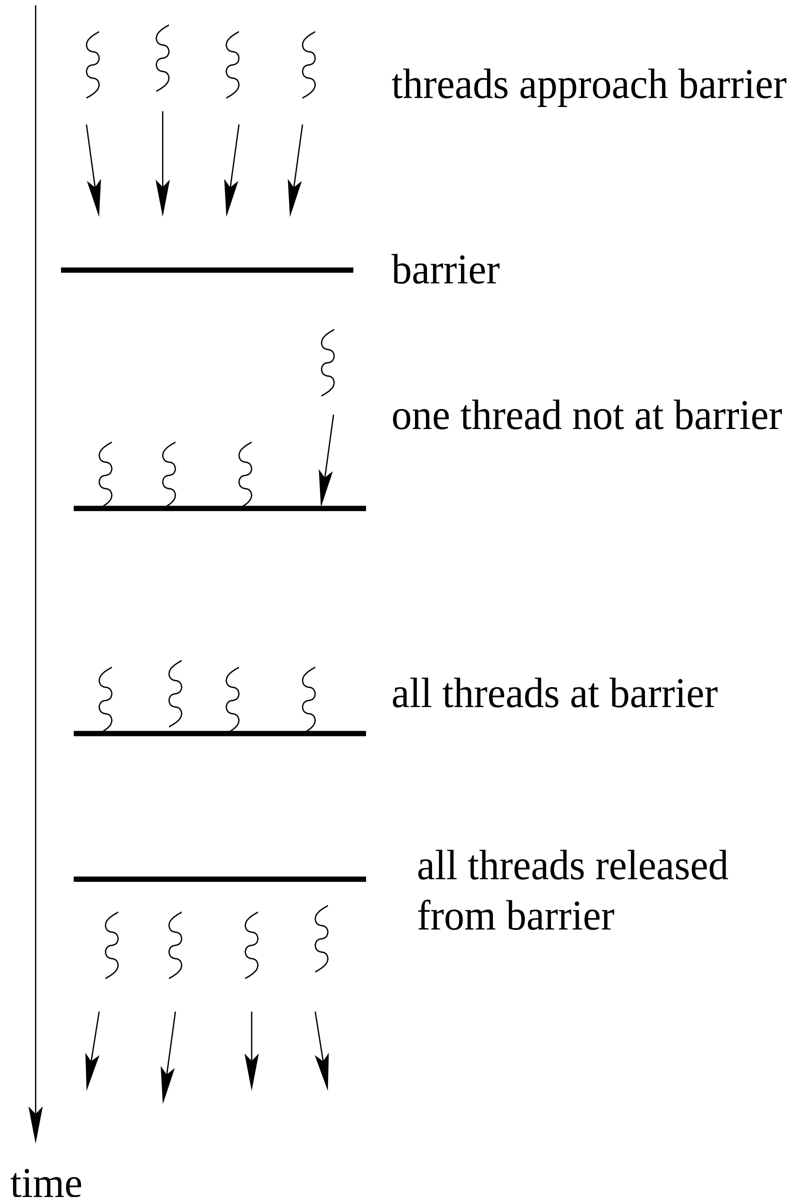
Barrier

- **Barrier** is common synchronization primitive

- ALL threads must reach barrier before ANY thread may continue past barrier

- Most often used in scientific computation; e.g., relaxation computation

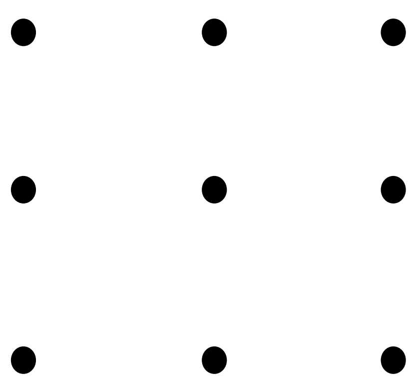
Depiction



threads approach barrier
barrier
one thread not at barrier
all threads at barrier
all threads released
from barrier
time

Aside: Relaxation, I

- Common pattern in scientific computation: simulate 2D or 3D real-world phenomenon by computing the physics at each point on a grid:

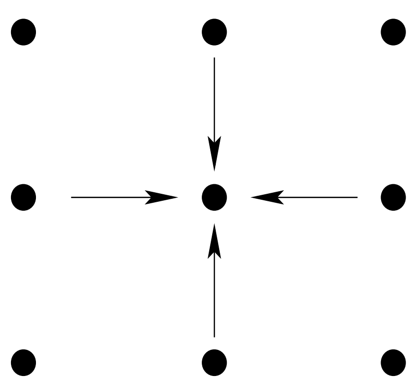


- The denser the grid points, the more faithful the simulation

Aside: Relaxation, II

- Values at adjacent grid points affect the computation

- Simulation runs in stages: compute result at each point, exchange results, compute again, etc.



- End of each stage is a barrier synchronization point

Barrier API

```
public class CyclicBarrier {
  public CyclicBarrier(int parties);
  public CyclicBarrier(int parties,
                       Runnable barrierAction);
  public int await() throws InterruptedException,
                            BrokenBarrierException;
  public boolean isBroken();
  public void reset();
  public int getParties();
  public int getNumberWaiting();
}
```

Barrier Use, I

1. Each thread calls `await()` and blocks

2. Once LAST thread calls `await()`, ALL threads return

- Barrier can be reused in this fashion — that's why it's called "*cyclic*"

- The number of threads given as "`parties`" constructor argument

Barrier Use, II

- Possible to specify action to take after all threads reach barrier but before they are released:

```
public CyclicBarrier(int parties,
                     Runnable barrierAction);
```

- "`barrierAction`" is an object that implements `Runnable` — meaning it has a "`run`" method

- `run()` is executed by last thread to reach barrier (NOT in a separate thread)

Broken Barrier Concept

- Barrier may be "broken:"

```
public int await() throws InterruptedException,
                          BrokenBarrierException;
public boolean isBroken();
```

- Ways to break a barrier:

- `await`-ing thread is interrupted
- `barrierAction` throws exception

- Broken barrier must be "`reset`" before next use:

```
reset()
```

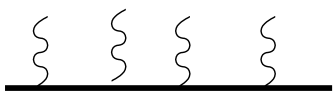
Problems With Barriers

1. Must know number of threads when barrier is created

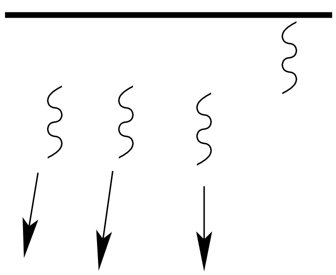
2. Scheduling anomaly can lead to barrier getting stuck

Scheduling Anomaly

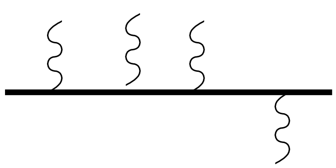
Barrier often occurs inside loop; e.g., relaxation problem



all threads at barrier



all threads released
from barrier, but
one doesn’t get
scheduled promptly



at next occurrence
of barrier, all threads
are "at the barrier"
but one thread did
not perform its
computation during
the previous stage

©2014 Stevens Institute of Technology

Solution

- Need a **sense reversing barrier**

- Java's `CyclicBarrier` is sense reversing – that's what "cyclic" means

Non-Cyclic Barrier

- Non-cyclic barrier's internal logic executed by each thread:

```
LOCK(barrier.lock);
if (barrier.count == 0) {
    // first thread executes here
    barrier.release = 0;
}
local_count = ++barrier.count;
UNLOCK(barrier.lock);
if (local_count == N) {
    // last thread executes here
    barrier.count = 0;
    barrier.release = 1;
} else {
    while (barrier.release == 0)
        ;
}
```

- Unprotected access to `barrier` inside if-clause is OK because thread knows others are waiting (spinning on `barrier.release`)

- Unprotected access to `barrier.release` assumes it is atomic (i.e., one word)

Problem With Non-Cyclic Barrier

- One “fast” thread ...

  - Releases from first barrier
  - Is first thread to get to second barrier
  - Resets `barrier.release` to 0

All this happens before “slow” thread releases from first barrier

- Slow thread is still executing

```
while (barrier.release == 0)
    ;
```

- Slow thread won’t release from first barrier so count at second barrier will never go beyond N-1

Cyclic Barrier

Cyclic barrier's internal logic executed by each thread:

```
local_sense = !local_sense;
LOCK(barrier.lock);
local_count = barrier.count++;
UNLOCK(barrier.lock);

if (local_count == N) {
    barrier.count = 0;
    barrier.release = local_sense;
} else {
    while (barrier.release != local_sense)
        ;
}
```

Countdown Latch

- Similar to barrier

- `CountDownLatch` starts with a positive `int` value

- Each thread to call `await()` blocks if count>0

- When count reaches 0, all threads released

Q: How does count go down?

A: call "`countDown()`"

Exchanger

- Think of it as 2-thread barrier plus object exchange:

```
public class Exchanger<V> {
    public Exchanger();
    public V exchange(V x) throws
                            InterruptedException;
}
```

- A *generic* type — note type parameter "`V`"

1. First thread to call "`exchange`" blocks

2. When second thread calls `exchange`, both threads return AND each returns the other's argument

©2014 Stevens Institute of Technology

SynchronousQueue

- Think of it as 1-item exchanger

- Implements what textbooks call a "rendezvous"

- First thread calls `put` and blocks

- Second thread later calls `take` – it now has the item and the first thread is unblocked

- (If `take` is called first then `put` unblocks the taker)

- Despite the name, it's NOT a queue ... `size` always returns 0

Does This Stuff Really Get Used?

Semaphore — yes

Read/write locks — yes

Barrier — yes

Countdown latch — no

Exchanger — no

SynchronousQueue — no