Concurrency in Java

Vivek Shah <bonii@diku.dk>
Frederik M. Madsen <fmma@diku.dk>

DIKU

Marcos Vaz Salles <vmarcos@diku.dk>
Course Responsible

Tentative Plan

Day 2: Concurrency

- Atomicity.
- Memory Consistency.
- Liveness.
- Thread Synchronization.
- High level concurrency.

Thread synchronization

- Threads do not always operate on independent resources.
- With concurrency resource contention and sharing is a problem that needs to be tackled.
- Proper access to resources must be ensured.
- A need to understand the notions of atomicity and visibility to coordinate resource sharing.
- Mutual exclusion is a mechanism to prevent thread collisions and ensure thread-safety.

Thread Interference

```
class Counter {
    protected int c = 0;

    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```

Thread interference

C = 0

Retrieve the value of c into ACC1

Add 1 to ACC1

Retrieve the value of c into ACC2

Subtract 1 from ACC2

Store the value of ACC2 to c

Store the value of ACC1 to c

Retrieve the value of c into ACC1

Add 1 to ACC1

Store the value of ACC1 to c

Retrieve the value of c into ACC2

Subtract 1 from ACC2

Store the value of ACC2 to c

ACC1 = 1 ACC2 = -1 C = 1

Thread interference

- Interference happens when different actions in separate threads on shared data interleave.
- Interleavings happen on the instructions generated by VM.
- Simple single step programming operations can be broken into multi-steps by VM leading to interleavings.
- Interleavings can happen in strange ways.
    - A thread can overwrite the others changes.
    - They can potentially interleave to produce correct results.
    - Can be very difficult to reproduce and reason for correctness.

Memory consistency

C = 0

```
public void increment() {
        c++;
}
```

C = 0 C = 1 C = 1

```
public void decrement() {
        c--;
}
```

C = -1 C = 0 C = 1 C = -1

C = 0

Understanding happens-before relationship is key to understanding visibility and avoid memory inconsistency

Memory consistency

- Errors arise due to inconsistent view of data in different threads. More frequent in a multi-processor system.
- Causes are complex and almost impossible to predict.
- Happens before is a guarantee that memory writes are visible between statements.
- Implicit happens-before in
  - Same thread
  - Thread.start
  - Thread.join
- A write to a “volatile” variable happens before subsequent reads of the same variable.

Volatile and memory consistency

```
class VolatileCounter {
    protected volatile int c = 0;

    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```

Still does not solve the interleaving issue

Using synchronized methods

- Adding “synchronized” to methods fixes both problems of interleavings (atomicity) and memory consistency.
- Two invocations of synchronized methods of same object cannot interleave.
- A synchronized method establishes “happens-before” relationship with subsequent invocation of synchronized methods on same object.

Using synchronized methods

- Provides simple design pattern for data sharing between threads. Encapsulate data in a class and synchronize ALL methods.

- Synchronized methods are built on the concept of intrinsic lock or monitor locks.

- Synchronized methods are reentrant.
    - Reentrant synchronization – Allowing the same method to re-acquire its lock.

Using synchronized methods

Does not work!!

```
class SynchronizedCounter {

    protected int c = 0;

    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```

Using synchronized methods

```
class SynchronizedCounter {

    protected int c = 0;

    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Using synchronized statements

- Synchronized statements allow to synchronize code blocks instead of methods.
- In synchronized statements one must explicitly specify the object on which an implicit lock needs to be taken.
- Provides finer granularity of synchronization, can lead to improvement of concurrency.
- Use with extreme care. You have to ensure you understand safety conditions of possible interleavings.

Using synchronized statements

```
class NewSynchronizedCounter {

    protected int c = 0;

    public void increment() {
          synchronized(this) {
              c++;
          }
    }

    public void decrement() {
          synchronized(this) {
              c--;
          }
    }

    public int value() {
        synchronized(this) {
          return c;
       }
    }
}
```

Using Locks

- Synchronized code is based on simple re-entrant locks.
- Java.util.concurrent.locks package provides more sophisticated locking patterns.
- A lock can be held by only one thread.
- A thread can check to see if a lock request can be granted and then choose to not acquire a lock instead of blocking.
  - tryLock() method. A timed version possible as well.
- Good programming requires that you use try finally where you unlock in the finally clause.

Using Locks

```
class LockedCounter {

    protected int c = 0;
    protected Lock myLock = new ReentrantLock();

    public void increment() {
                mylock.lock();
                c++;
                mylock.unlock();
    }

    public void decrement() {
            myLock.lock();
            c--;
            myLock.unlock();
    }

    public int value() {
        return c;
    }
}
```

Does not work!!

Using Locks

```
class LockedCounter {

    protected int c = 0;
    protected Lock myLock = new ReentrantLock();

    public void increment() {
                myLock.lock();
                c++;
                mylock.unlock();
    }

    public void decrement() {
            myLock.lock();
            c--;
            myLock.unlock();
    }

    public int value() {
         myLock.lock();
         return c;
         myLock.unlock();
    }
}
```

Does not work!!

Using Locks

```
class LockedCounter {

    protected int c = 0;
    protected Lock myLock = new ReentrantLock();

    public void increment() {
                myLock.lock();
                c++;
                mylock.unlock();
    }

    public void decrement() {
            myLock.lock();
            c--;
            myLock.unlock();
    }

    public int value() {
         int temp;
         myLock.lock();
         temp = c;
         myLock.unlock();
         return temp;
    }
}
```

Atomic classes

- Java.util.concurrent.atomic contains classes that provide lock-free thread safe operations on single variables.
- Provide a conditional update operation of type
  - boolean compareAndSet(expectedValue, updateValue);
- Atomic classes have been built to be used as building blocks to construct non blocking data structure classes.
- Use extreme caution if you are making assumptions of atomicity.

Using Atomic Classes

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c;

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }
}
```

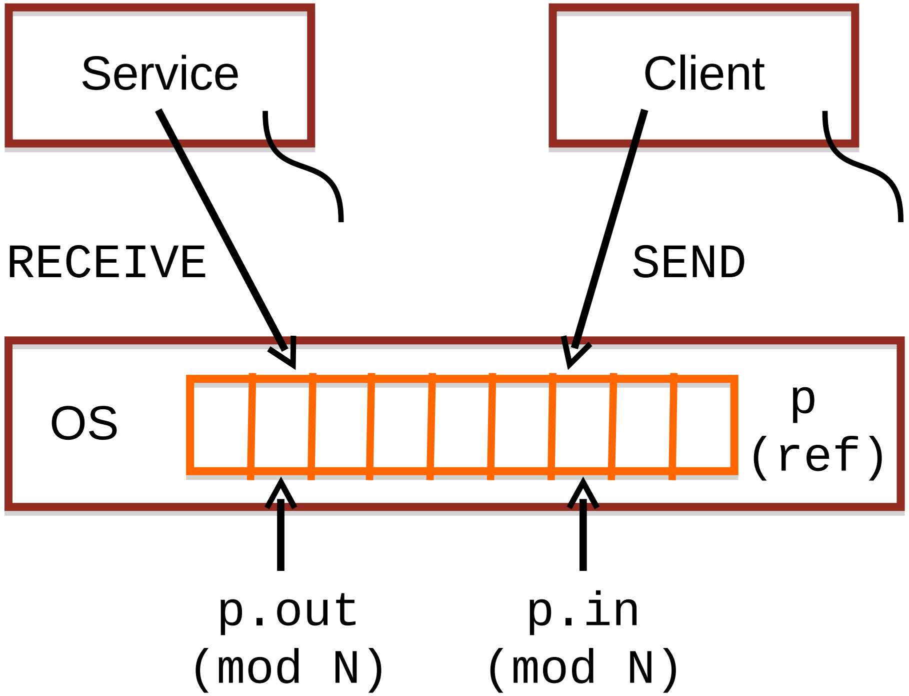
Liveness

- A concurrent application's ability to run in a timely manner.
- Challenges to liveness
    - Deadlocks
        - Threads are blocked forever, waiting for each other.
    - Starvation
        - Threads are unable to gain “regular” access to shared resource and are unable to make progress.
    - Livelock
        - Threads are busy responding to each other creating cyclic patterns without making progress.

Deadlocks

- Possible scenarios where deadlocks can occur
  - Synchronized methods or statements invoking other synchronized methods/statements.
  - Nested lock acquisitions.
  - Threads joining to each other.
- Using self-contained synchronized method provides a simple and elegant way to avoid deadlocks.
- It is often very difficult/restrictive to guarantee deadlock-free code.

Thread Synchronization: Bounded Buffer Problem



ALLOCATE_BOUNDED_BUFFER(int) → buffer

SEND(buffer, message)

RECEIVE(buffer) → message

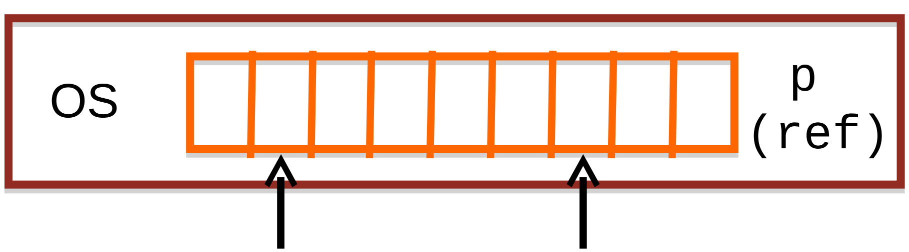
- DEALLOCATE_BOUNDED_BUFFER(buffer)

Thread Synchronization: Bounded Buffer Problem

```
SEND(buffer ref p, message m):
        while p.in – p.out = N do nothing
        p.message[p.in mod N] ← m
        p.in ← p.in + 1

RECEIVE(buffer ref p):
        while p.in = p.out do nothing
        m ← p.message[p.out mod N]
        p.out ← p.out + 1
        return m
```

Does this work with multiple sender threads? Why?



Busy waiting

- Guarded blocks are the most common co-ordination pattern.
- Share a common variable between threads.
- Keep polling the variable to evaluate if a condition is true.
- Wasteful of machine cycles since it continuously executes while waiting.

BoundedBuffer using busy/wait (let's look at the code)

Wait and Notify

- When a thread enters a wait(), the thread is suspended and its locks are released (different to sleep,yield).

- The wait() does not return until another thread issues a notification in the form of notify() or notifyAll() or the timer runs out (for the timed version).

- The only place you can call wait( ), notify( ) or notifyAll( ) is within a synchronized method or block.
    - Will compile, but generate Runtime exception.

Wait and notify

- Wait() can get spurious wake-ups. Always invoke wait() inside a loop where you test for condition.
- Wait() is used when a thread needs to wait for certain conditions to change outside the control of the current thread.
- If you cannot change the world, go to sleep and hope to be woken up when it actually changes!!

BoundedBuffer using wait/notify (let's look at the code)

Condition objects

- Conditions provide a way for threads to wait for “conditions” and to notify “conditions”.
- Condition interface in java.util.concurrent.locks package.
- Can control fine granule waiting and notifying.
- wait → await(), notify → signal(), notifyAll() → signalAll()
- Uses lock objects.
- Optimized syntactic sugared version of wait/notify.

BoundedBuffer using condition objects (let's look at the code)

And we just finished implementing
java.util.concurrent.BlockingQueue :-)

Reader Writer interface

```
public interface ReaderWriter {
    public void acquireExclusive();
    public void releaseExclusive();
    public void acquireShared();
    public void releaseShared();
}
```

Reader Writer implementation. Lets look at the code

Lots of higher level concurrency aids in java.util.concurrent

- ThreadLocal.
- BlockingQueue.
- ReadWriteLocks.
- ConcurrentMap.
- Semaphores.
- Mutexes.
- Cyclic barriers.
- Countdown Latch.
- If you are looking to build concurrent code, look at available components in java.util.concurrent before building.