

Northeastern University - Seattle



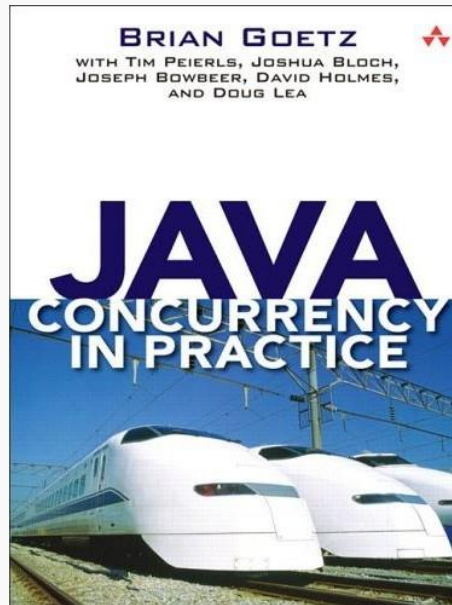
CS5510

Professor Ian Gorton

Week 9

CONCURRENCY: INTRODUCTION

<http://jcip.net/>



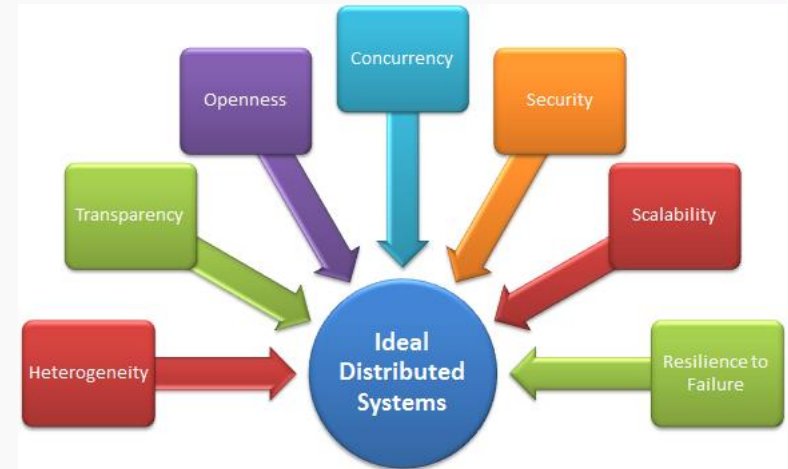
Overview

- Why threads?
- Simple threads in Java
- Problems with threading
- Synchronization primitives
- Thread coordination
- Thread states
- Thread pools
- Thread-safe collections

WHY THREADS?

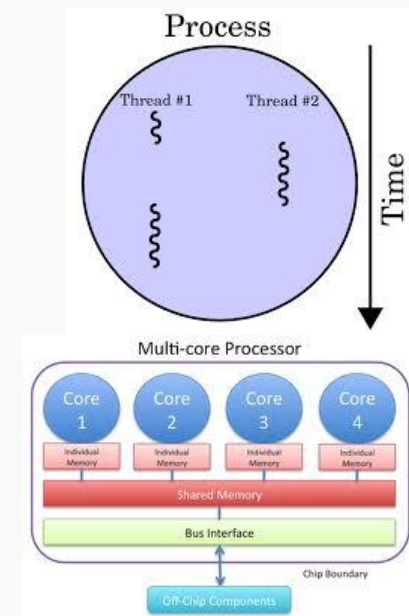
Concurrency is Fundamental to Many Systems

- Distributed systems are inherently concurrent
 - Events happen on different nodes at the same time
 - Unpredictable order of events
- Concurrency needed on each node to provide:
 - Responsiveness to requests
 - Throughput
 - Ability to handle multiple simultaneous requests



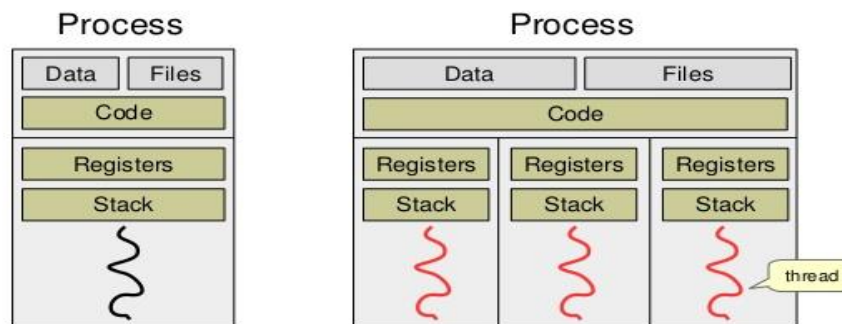
Why Concurrency?

- Concurrent execution is necessary in many systems:
 - Natural solution to many problems
 - Increase performance, e.g. do work while waiting for disk accesses
 - Necessary to exploit multicore



Address Space

What is an OS process and thread?



Threads share the same address space

- consumes less memory
 - spawning is cheaper
 - context switching is cheaper

Units of Concurrency

- Processes - different executables - comprise
 - virtual address space
 - Code
 - Security context
 - Environment variables
 - Handles to system object (e.g. sockets)
 - A main thread of execution
- A process can create multiple threads ...

Threads

- Threads are lightweight compared to processes
 - share the same address space and share data and code
 - Allocated their own stack space to support independent execution
- Context switching between threads is less expensive than between processes
- Cost of thread intercommunication is lower than process intercommunication

SIMPLE THREADS IN JAVA

Java Threads

```
public class Thread extends Object implements Runnable  
public interface Runnable {  
    void run();  
}
```

- Write a class that
 - implements Runnable,
 - overrides the run() method
- Instantiate class and call start()

Java Threads – Simple Example

```
public class NamingThread implements Runnable {  
  
    private String name;  
    public NamingThread () {  
        name = "Anon";  
    }  
    public NamingThread (String threadName) {  
        name = threadName;  
    }  
    @Override  
    public void run() {  
        System.out.println (name + " is " + Thread.currentThread());  
    }  
}
```

Java Threads – Simple Example

```
public class ThreadStartOrderExample {  
  
    public static void main(String arg[]) {  
  
        Thread th1 = new Thread (new NamingThread("Pep is a genius")) ;  
        Thread th2 = new Thread (new NamingThread("Mourinho is an idiot"));  
        Thread th3 = new Thread (new NamingThread("doh")) ;  
  
        System.out.println ("Ready to roll ...");  
        th1.start();  
        th2.start();  
        th3.start();  
  
        System.out.println ("main thread exiting " + Thread.currentThread());  
    }  
}
```

Java Threads

- Alternative is to extend java.lang.Thread class
- See [this reference](#) for a more detailed treatment of the differences

```
class MyThread extends Thread {  
  
    MyThread() {  
    }  
  
    MyThread(String threadName) {  
        super(threadName); // Initialize thread object  
        System.out.println(this);  
        start();  
    }  
  
    public void run() {  
        //Display info about this thread  
        System.out.println(Thread.currentThread().getName());  
    }  
}
```

Thread Object Example

```
public class ThreadExample2 {  
  
    public static void main(String[] args) {  
        Thread thread1 = new Thread(new MyThread(), "thread1");  
        Thread thread2 = new Thread(new MyThread(), "thread2");  
        // The next 2 threads are assigned default names  
        Thread thread3 = new MyThread();  
        Thread thread4 = new MyThread();  
        //Start the threads  
        thread1.start();  
        thread2.start();  
        thread3.start();  
        thread4.start();  
        try {  
            //sleep a one second  
            Thread.currentThread().sleep(1000);  
        } catch (InterruptedException e) {  
        }  
        //Display info about the main thread  
        System.out.println(Thread.currentThread());  
    }  
}
```

PROBLEMS WITH THREADS

Concurrency makes things 'fun'

- Problems with concurrency
 - Race conditions
 - Deadlocks
- Source of problems
 - Non-determinism
 - Interleavings



First Problem: Shared Variables

- Multiple independent threads make changes to same variable at same time
 1. read value from memory to register
 2. change value in register
 3. write register value back to memory
 - thread 1: $x = x + 6$
 - thread 2: $x = x + 1$
- The result?

Welcome to Race Conditions

Thread 1	Thread 2
Reads (x) into register	
Register value + 6	
Writes register value to (x)	
	Reads (x) into register
	Register value + 1
	Writes register value to (x)



Thread 1	Thread 2
Reads (x) into register	
Register value + 6	
	Reads (x) into register
	Register value + 1
	Writes register value to (x)
Writes register value to (x)	



Race Conditions

- Same program, different results
 - Depends on the manner in which CPU schedules execution
 - Different interleavings produce different outcomes
- Extremely hard to debug
 - Not reproducible
 - These are extremely unpleasant when they occur in production systems

Nondeterminism is Unavoidable,
but Data Races are Pure Evil

Hans-J. Boehm
HP Labs



5 November 2002

5

Root Cause: Non-Determinism

- Sequential programs exhibit deterministic behavior
- Race conditions are caused by **non-deterministic** behavior
- Two kinds of non-determinism
 - observable – program may give different result
 - non-observable – program may execute differently, but this does not affect the result
- Thread 1 { a=2; b=a+6; }
- Thread 2 { x=9; y=x-3; }
- Whatever order the scheduler runs these threads in, the result will always be the same
 - No shared variables



Thread Interleaving Example

```
public class Interleaving {  
  
    public void show() {  
        for (int i = 0; i < 5; i++) {  
            System.out.println(Thread.currentThread().getName() + " - Number: " + i);  
        }  
    }  
  
    public static void main(String[] args) {  
        final Interleaving main = new Interleaving();  
  
        Runnable runner = new Runnable() {  
            @Override  
            public void run() {  
                main.show();  
            }  
        };  
        new Thread(runner, "Thread 1").start();  
        new Thread(runner, "Thread 2").start();  
    }  
}
```

SYNCHRONIZATION PRIMITIVES

Two common causes for race conditions

- Check-then-act
- Read-modify-write
- Clone the repo at:
 - <https://github.com/xpadro/concurrency>

Check-then-act – Race Condition

```
public class UnsafeCheckThenAct {
    private int number;

    public void changeNumber() {
        if (number == 0) {
            System.out.println(Thread.currentThread().getName() + " | Changed");
            number = -1;
        }
        else {
            System.out.println(Thread.currentThread().getName() + " | Not changed");
        }
    }

    public static void main(String[] args) {
        final UnsafeCheckThenAct checkAct = new UnsafeCheckThenAct();

        for (int i = 0; i < 50; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    checkAct.changeNumber();
                }
            }, "T" + i).start();
        }
    }
}
```

Eradicating Race Conditions

- Use locks to impose ordering constraints
 - Lock shared variables so they can be accessed only by a single thread at once
 - Serialized access to shared resources
 - Implements non-observable non-determinism
 - Locks sometimes known as semaphores

Locks

- Locks serialize access to shared variables
 - Each thread wishing to access a variable:
 - takes the lock
 - changes the variable
 - releases the lock
 - If the lock is set, all other threads wait for it to be released
 - Which thread proceeds next?
 - Think about solving the race condition on the previous slide
-

Check-then-act – No Race Condition

```
public class UnsafeCheckThenAct {
    private int number;

    public synchronized void changeNumber() {
        if (number == 0) {
            System.out.println(Thread.currentThread().getName() + " | Changed");
            number = -1;
        }
        else {
            System.out.println(Thread.currentThread().getName() + " | Not changed");
        }
    }

    public static void main(String[] args) {
        final UnsafeCheckThenAct checkAct = new UnsafeCheckThenAct();

        for (int i = 0; i < 50; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    checkAct.changeNumber();
                }
            }, "T" + i).start();
        }
    }
}
```

Java synchronized methods

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment() { c++; }  
    public synchronized void decrement() { c--; }  
    public synchronized int value() { return c; }  
}
```

- **Known as critical section**
 - it is not possible for two invocations of any synchronized methods on the same object to interleave
 - less error-prone as release is automatic

Definition - Atomicity

Operations A and B are atomic with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has.

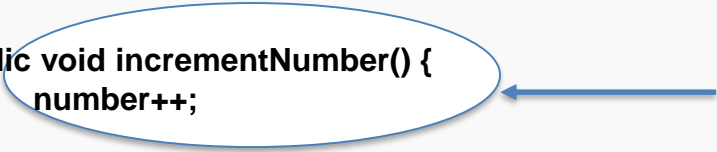
An atomic operation is one that is atomic with respect to all operations that operate on the same state

Monitor Locks

- Synchronization is implemented using monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock.
- Only one thread at a time may hold a lock on a monitor.
- Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor.
- A thread *t* may lock a particular monitor multiple times; each unlock reverses the effect of one lock operation.

Read-Modify-Write – Race Condition

```
public class UnsafeReadModifyWrite {  
    private int number;  
    C  
    public void incrementNumber() {  
        number++;  
    }  
    public int getNumber() {  
        return this.number;  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        final UnsafeReadModifyWrite rmw = new UnsafeReadModifyWrite();  
  
        for (int i = 0; i < 1000; i++) {  
            new Thread(new Runnable() {  
                @Override  
                public void run() {  
                    rmw.incrementNumber();  
                }  
            }, "T" + i).start();  
        }  
  
        Thread.sleep(6000);  
        System.out.println("Final number (should be 1000): " + rmw.getNumber());  
    }  
}
```



Compound operation –
synchronized needed

Reproducing the error

- Run UnsafeReadModifyWrite.java
- Can you see the race condition?



UnsafeReadModifyWriteWithLatch

```
public static void test() throws InterruptedException {
    final UnsafeReadModifyWriteWithLatch rmw = new UnsafeReadModifyWriteWithLatch();

    for (int i = 0; i < NUM_THREADS; i++) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    rmw.startSignal.await();
                    rmw.incrementNumber();
                } catch (InterruptedException e) {}

                } finally {
                    rmw.endSignal.countDown();
                }
            }
        }, "T" + i).start();

        Thread.sleep(2000);
        rmw.startSignal.countDown();
        rmw.endSignal.await();
        System.out.println("Final number (should be 1_000): " + rmw.getNumber());
    }
}
```

What if we remove this?

Another Solution – Atomic Variables

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

public class SafeReadModifyWriteAtomic {
    private final AtomicInteger number = new AtomicInteger();

    public void incrementNumber() {
        number.getAndIncrement();
    }

    public int getNumber() {return this.number.get();}

    // rest is same as previous
```

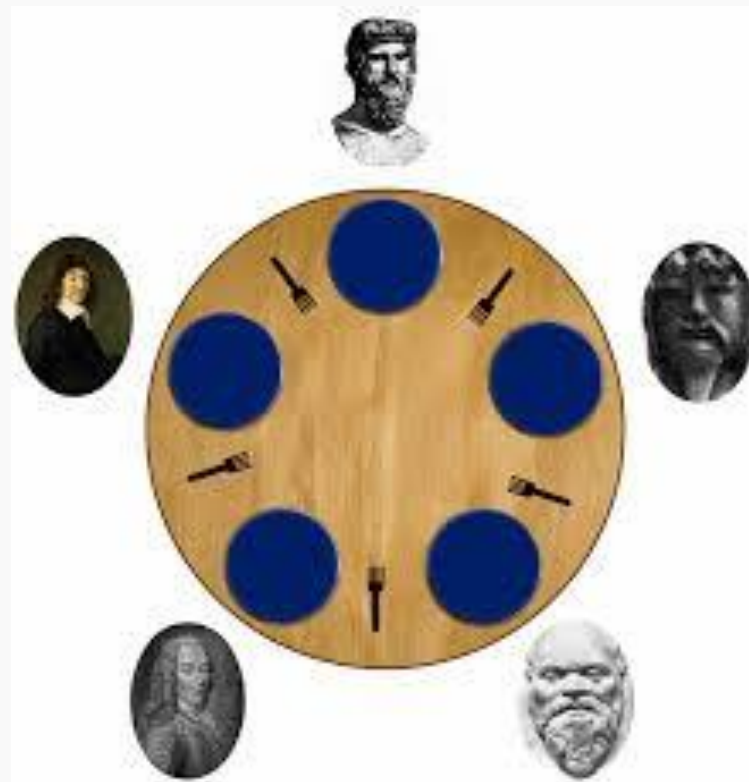
More Atomic types explained at:

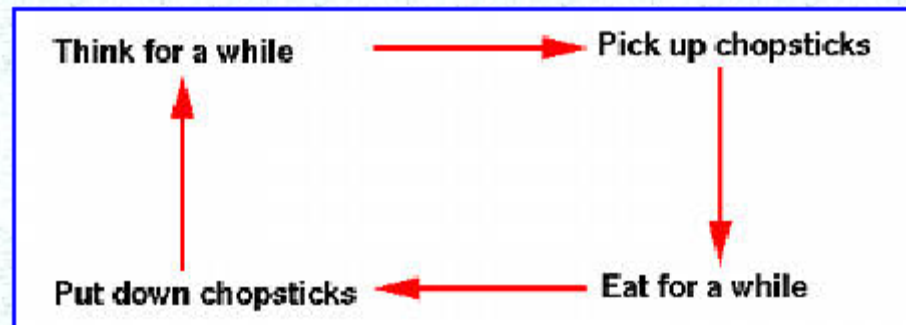
<http://tutorials.jenkov.com/java-util-concurrent/index.html>

Sharing Structures

- Consider a linked list with explicit size variable
 1. read size variable
 2. add new element to the list at end
 3. increment and write back size variable
- size variable and list elements must be synchronized
- concurrent access of non thread-safe structures is dangerous
 - none of `java.util.*` are thread-safe

The Dining Philosophers Problem





Pseudo-Code for a Philosopher

```
while(true) {  
    // Initially, thinking about life, universe, and everything  
    think();  
  
    // Take a break from thinking, hungry now  
    pick_up_left_fork();  
    pick_up_right_fork();  
    eat();  
    put_down_right_fork();  
    put_down_left_fork();  
  
    // Not hungry anymore. Back to thinking!  
}
```

**Let's test this out
With
Real Philosophers
(and food!)**

```

public class Philosopher implements Runnable {

    private final Object leftFork;
    private final Object rightFork;

    Philosopher(Object left, Object right) {
        this.leftFork = left;
        this.rightFork = right;
    }

    private void doAction(String action) throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + " " + action);
        Thread.sleep(((int) (Math.random() * 100)));
    }

    public void run() {
        try {
            while (true) {
                doAction(System.nanoTime() + ": Thinking"); // thinking
                synchronized (leftFork) {
                    doAction(System.nanoTime() + ": Picked up left fork");
                    synchronized (rightFork) {
                        doAction(System.nanoTime() + ": Picked up right fork - eating"); // eating
                        doAction(System.nanoTime() + ": Put down right fork");
                    }
                    doAction(System.nanoTime() + ": Put down left fork. Returning to thinking");
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

```

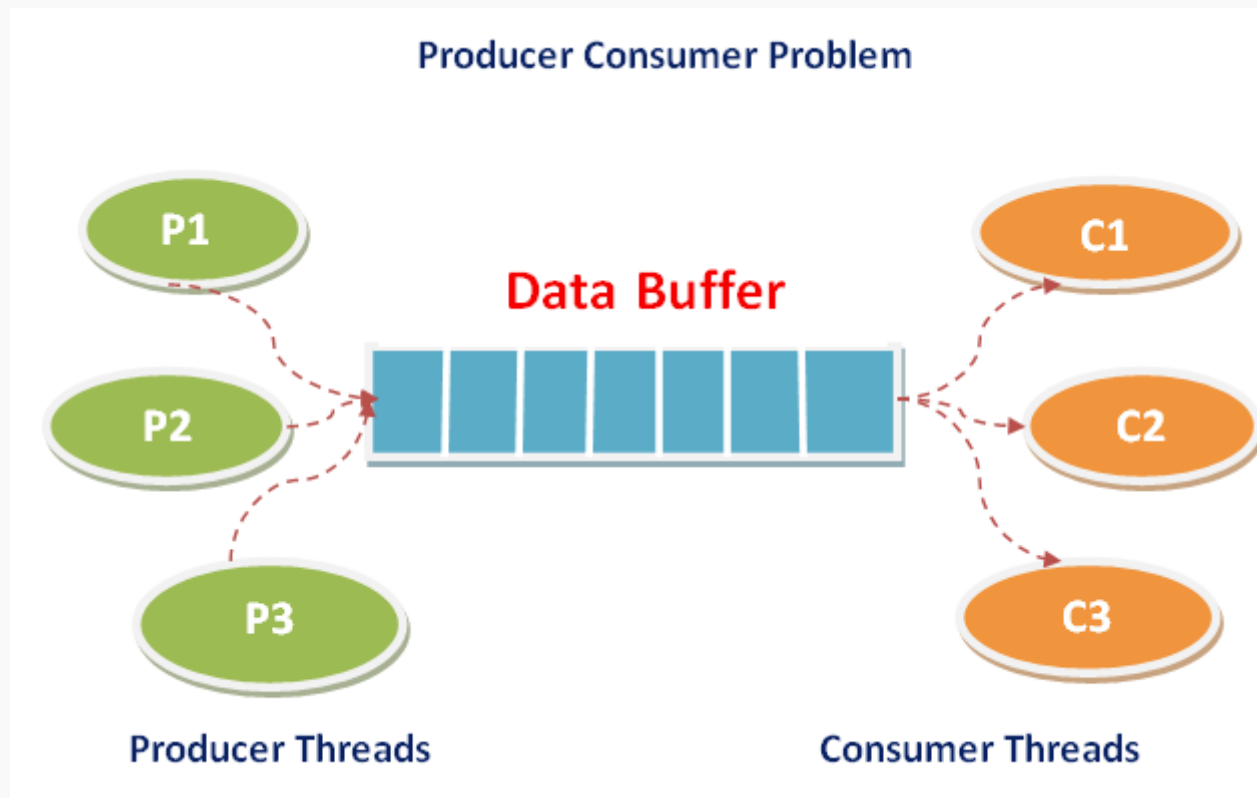
Deadlock

- 2 threads sharing access to 2 shared variables via locks
 1. thread 1: takes lock a
 2. thread 2: takes lock b
 3. thread 1: blocks on b
 4. thread 2: blocks on lock a
- Deadlock!!
 - Neither thread can proceed
 - *This violates 'liveness'* – something good eventually happens

This is why concurrency is hard

- Too few ordering constraints => race conditions
- Too many ordering constraints => deadlocks
- Hard/impossible to reason about based on modularity
 - If an object is shared by multiple threads, need to think about what all threads could do
- Thorough testing is impossible
 - Non-determinism leads to an infinite number of possible interleavings
 - Controlled by the scheduler and events, not the program

THREAD COORDINATION



Producer Consumer – First cut

```
int itemCount = 0;
```

```
procedure producer()
```

```
{  
    while (true)  
    {  
        item = produceItem();  
  
        if (itemCount == BUFFER_SIZE)  
        {  
            sleep();  
        }  
  
        putItemIntoBuffer(item);  
        itemCount = itemCount + 1;  
  
        if (itemCount == 1)  
        {  
            wakeup(consumer);  
        }  
    }  
}
```

```
procedure consumer()
```

```
{  
    while (true)  
    {  
        if (itemCount == 0)  
        {  
            sleep();  
        }  
  
        item = removeItemFromBuffer();  
        itemCount = itemCount - 1;  
  
        if (itemCount == BUFFER_SIZE - 1)  
        {  
            wakeup(producer);  
        }  
  
        consumeItem(item);  
    }  
}
```

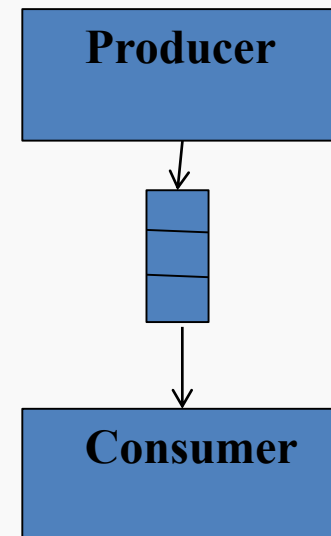
Deadlock

- Consumer reads itemCount, notices it's zero and moves inside the if block.
- Before calling sleep, the consumer is interrupted
- Producer creates an item, puts it into the buffer, and increases itemCount.
- Because the buffer was empty, the producer tries to wake up the consumer.
- Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost.
- The producer will loop until the buffer is full, after which it will also go to sleep.

DEADLOCK

Guards

- Producer-Consumer style examples require 'guards'
- Producer stores message in a shared buffer
 - Except when full
- Consumers retrieve messages from buffer
 - Wait when empty



Java Guards (aka Monitors)

- Wait() and notify() statements
- Wait and notify provide thread inter-communication that synchronizes on the same object.
 - final void wait(long timeout) throws InterruptedException
 - final void wait() throws InterruptedException
 - final void notify()
 - final void notifyAll()
- Let's work through an example ...
 - see ProducerConsumerExample

```
public class Buffer {  
    // Message buffer between producer to consumer.  
    // private String message;  
    // True if consumer must wait for producer to send  
    // message,  
    // false if producer must wait for consumer to retrieve  
    message.  
    private boolean empty = true;
```

```
    public synchronized String retrieve() {  
        // Wait until message is available.  
        while (empty) {  
            try {  
                System.out.println("Waiting for a message");  
                wait();  
            } catch (InterruptedException e) {}  
        }  
        // Toggle status.  
        empty = true;  
        // Notify producer that buffer is empty  
        notifyAll();  
        return message;  
    }  
}
```

```
    public synchronized void put(String message) {  
        // Wait until message has been retrieved.  
        while (!empty) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
        // Toggle status.  
        empty = false;  
        // Store message.  
        this.message = message;  
        // Notify consumer that message is  
        // available  
        notifyAll();  
    }  
}
```

Class Exercise

Look at the BoundedBufferExample in the repo you cloned and make sure you understand how it works.

- What happens if you start more than producer thread?
- More than one consumer thread?

THREAD STATES

Thread States

- New Thread state (Ready-to-run state)
 - Created but not started
- Runnable state (Running state)
 - Started and either running or waiting to run
- Not Runnable state
- Dead state
 - Stop() called or run() terminates

Not Runnable

- A thread is Not Runnable if one of the following occurs:
 - When **sleep()** is invoked
 - `Thread.currentThread().sleep(1000);`
 - When **suspend()** is invoked
 - When **the wait()** method is invoked
 - waits for notification of a free resource
 - waits for completion of another thread
 - waits to acquire a lock of an object.
 - The thread is blocking on an I/O request

Thread Resumption

- If a thread is asleep:
 - the sleep period must elapse or interrupt() method called
- If a thread is suspended:
 - its resume() method must be called
- If a thread is waiting on a condition variable,
 - an object owning the variable must relinquish it by calling either notify() or notifyAll().
- If a thread is waiting on I/O, then I/O must complete

Thread Priority

- In Java every thread has a priority
 - Higher priority threads get scheduled more frequently than lower priority threads
- A Java thread inherits its priority from its parent
 - MIN_PRIORITY (0) Lowest Priority
 - NORM_PRIORITY (5) Default Priority
 - MAX_PRIORITY (10) Highest Priority

Thread Scheduling

- The thread scheduler chooses the Runnable thread with the highest priority for execution.
- When multiple threads to choose from, scheduler chooses one in a round-robin fashion. The chosen thread will run until:
 - a higher priority thread becomes Runnable. (Pre-emptive)
 - it yields, or its run() method exits
 - its time allotment has expired (time-slicing)

Reentrancy

- Every Java object has a lock associated with it
 - Known as the intrinsic lock
 - Aka monitor or mutex locks
- Synchronized methods exploit this intrinsic lock
 - Lock acquired by executing thread before entering a synchronized block
 - Lock released automatically when the thread exits the synchronized block

Does this Deadlock?

```
public class hipsterBaseClass {
    public synchronized void doHipsterStuff() {
        // random hipster behaviour
    }
}

public class capitolHillBar extends hipsterBaseClass
{
    public synchronized void orderDrinks() {
        // get drinks order
        super.doHipsterStuff();
    }
}
```

Reentrancy

- Intrinsic locks are reentrant
 - If a thread tries to acquire a lock it already holds, it succeeds
 - Each lock has an acquisition count and owning thread
 - Count can only be incremented above 1 by same owning thread
- Reentrancy facilitates encapsulation of locking behavior, and simplifies OO concurrent code

Performance and Scalability Issues with Threads

- Thread safety requires the internal state of an object to be protected from concurrent updates
 - Updates must be atomic and serialized
- What if an object has no state that persists between calls?
- Or cannot be modified by a calling thread?
- Is this thread-safe?

Stateless Servlet (jcip p13)

```
public class StatelessFactorizer extends GenericServlet implements  
Servlet {
```

```
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        encodeIntoResponse(resp, factors);  
    }
```

**Stateless and Immutable
objects
are
always
thread-safe**

THREADS POOLS

Java.util.concurrent

- The java.util.concurrent package contains a range of utilities to simplify multithreaded programs
 - Executor framework
 - Thread-safe collections

The Executor Framework

public interface Executor

An object that executes submitted Runnable tasks.

For example, rather than invoking `new Thread(new RunnableTask()).start()` for each of a set of tasks, you might use:

```
Executor executor = anExecutor;  
executor.execute(new RunnableTask1());  
executor.execute(new RunnableTask2());  
...
```

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Executor.html#method_summary

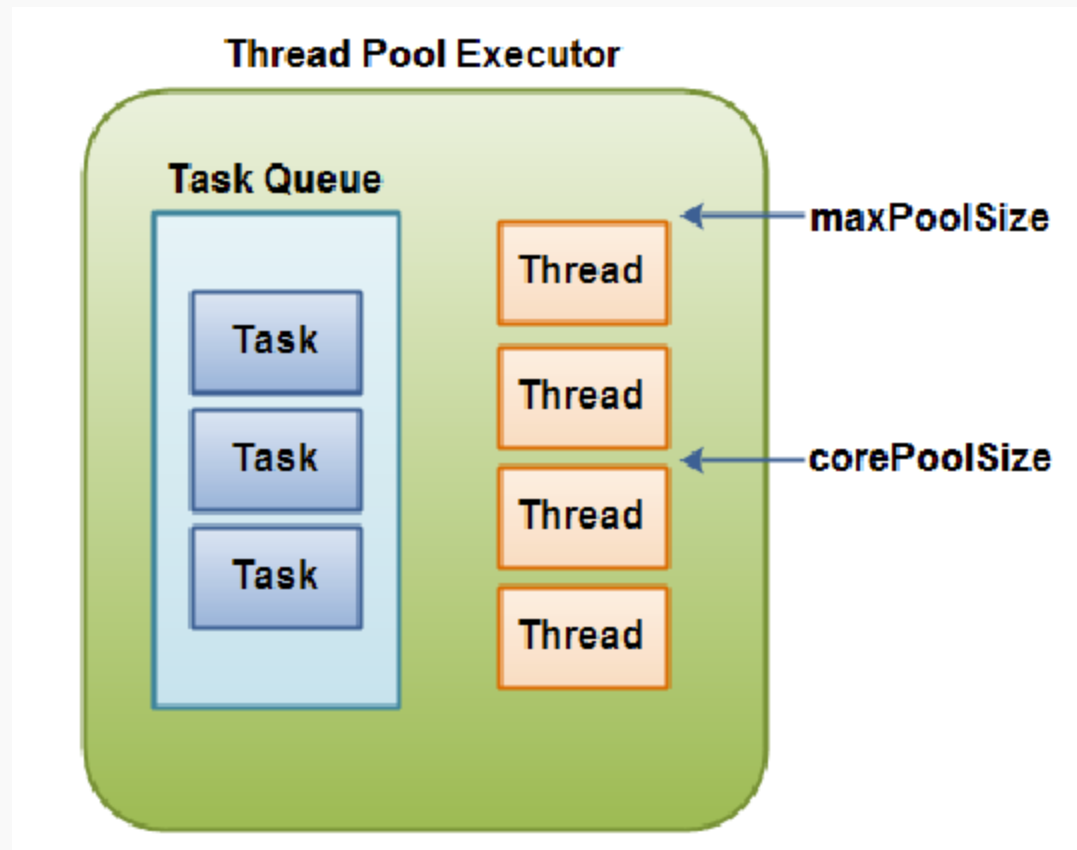
Executor

- Supports asynchronous task execution
- Decouples task submission from task executions
 - Supports different task execution policies
 - Provides task lifecycle support
 - Has hooks for adding statistics, management, monitoring
- [Executors](#) provide a factory method to create an Executor with desired policies.

Executor – Fixed Size thread Pool

```
ExecutorService executorService =  
    Executors.newFixedThreadPool(SIZE);  
  
executorService.execute(new Runnable() {  
    public void run() {  
        System.out.println("Asynchronous task");  
    }  
});  
  
executorService.shutdown();
```

Thread Pools



Execution Policies

- Executors decouple the submission of a request from the execution policy used
 - Makes it easy to change policies to suit deployment hardware – just choose a different Executors interface
 - Policies specific things like:
 - How many concurrent threads?
 - How many queued requests?
 - What to do if server overloaded?
 - Execution priorities/order (LIFO, FIFO), etc ...
-

More on Executors

- no way to obtain the result of a Runnable
 - if necessary.
- Or find out when threads have completed
- You will have to use a Callable or Future

submit(runnable)

```
Future future = executorService.submit(new Runnable() {  
    public void run() {  
        System.out.println("Asynchronous task");  
    }  
});
```

`future.get();` //returns null if the task has finished correctly.

submit(Callable)

```
Future future = executorService.submit(new Callable(){  
    public Object call() throws Exception {  
        System.out.println("Asynchronous Callable");  
        return "Callable Result";  
    }  
});
```

```
System.out.println("future.get() = " + future.get());
```

ExecutorService Shutdown

- Must shutdown an executor
 - `executorService.shutdown();`
- Stops accepting new requests but does not shutdown immediately
- Must wait for all threads to complete
 - `executorService.awaitTermination();`

THREAD-SAFE COLLECTIONS

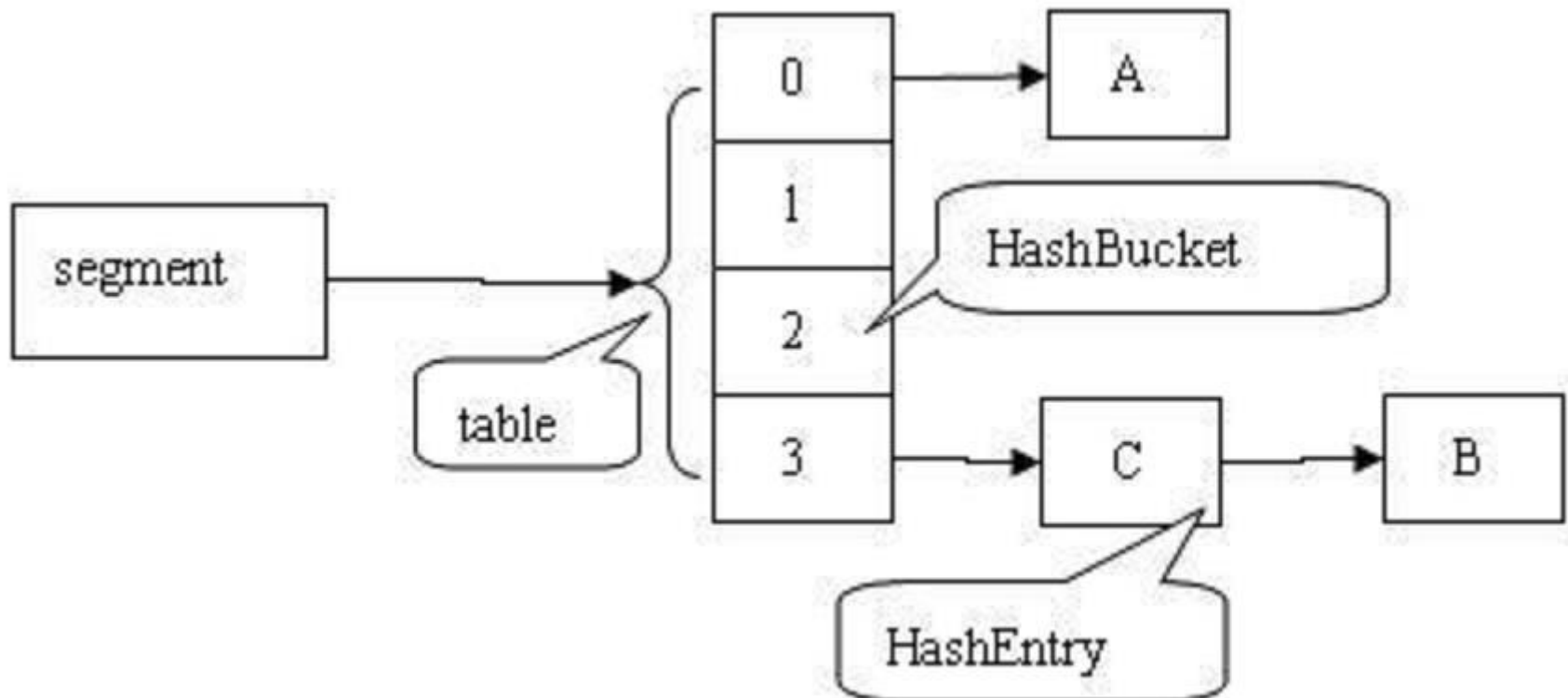
Thread-safe collection classes

- Standard collection classes are NOT thread-safe
- `java.util.concurrent` package includes additions to the Java Collections Framework.
 - **BlockingQueue**: FIFO that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
 - **ConcurrentMap** is a subinterface of `java.util.Map` that defines useful atomic operations. Also **ConcurrentHashMap**, which is a concurrent analog of `HashMap`.
 - **ConcurrentNavigableMap** is a subinterface of `ConcurrentMap` that supports approximate matches. Also **ConcurrentSkipListMap**, which is a concurrent analog of `TreeMap`.

ConcurrentHashMap

- HashMap divided into buckets
 - 16 by default
- A lock is applied at the bucket level
 - Allows safe concurrent modification

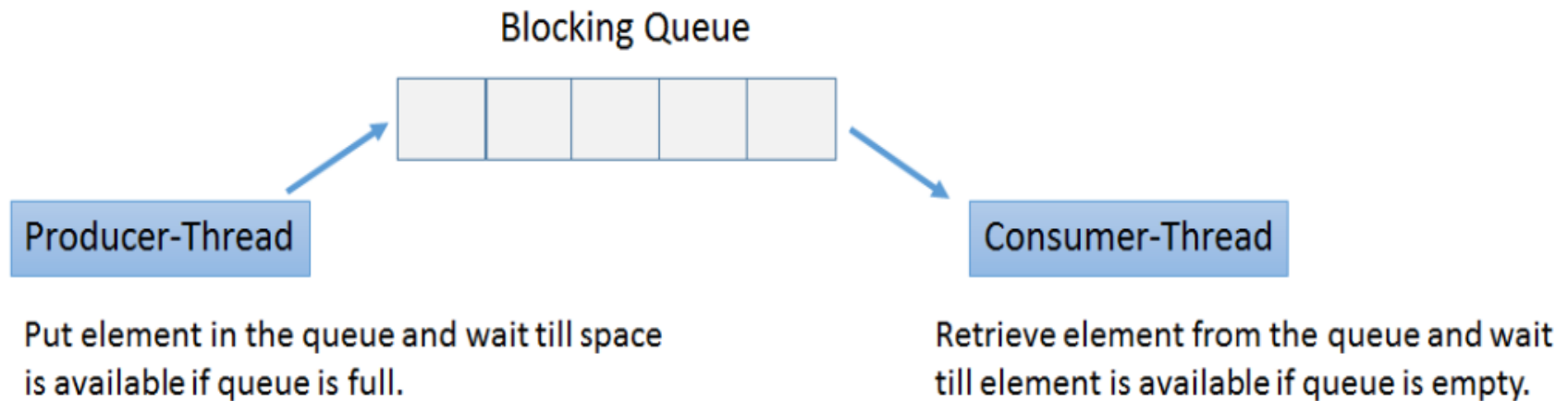
ConcurrentHashMap



ConcurrentHashMap

- Atomic operations:
 - putIfAbsent()
 - remove()
 - replace()
- Trade-offs - relaxed consistency for
 - Map.size()
 - Map.isEmpty()
 - iterators

BlockingQueue



BlockingQueue Example

```
class Producer implements Runnable {  
    private final BlockingQueue queue;  
    Producer(BlockingQueue q) { queue = q; }  
    public void run() {  
        try {  
            while (true) { queue.put(produce()); }  
        } catch (InterruptedException ex) { ... handle ... }  
    }  
    Object produce() { ... }  
}
```

```
class Consumer implements Runnable {  
    private final BlockingQueue queue;  
    Consumer(BlockingQueue q) { queue = q; }  
    public void run() {  
        try {  
            while (true) { consume(queue.take()); }  
        } catch (InterruptedException ex) { ... handle ... }  
    }  
    void consume(Object x) { ... }  
}
```

```
class Setup {  
    void main() {  
        BlockingQueue q = new  
            LinkedBlockingQueue();  
        Producer p = new Producer(q);  
        Consumer c1 = new Consumer(q);  
        Consumer c2 = new Consumer(q);  
        new Thread(p).start();  
        new Thread(c1).start();  
        new Thread(c2).start();  
    }  
}
```

And hot(-ish) off the presses

- Java 9 immutable collections
- <https://docs.oracle.com/javase/9/docs/api/java/util/Collections.html>

Summary

- Concurrency is fundamental to software systems
- Introduces problems of race conditions and deadlocks
- Synchronization required as a solution
- Threads move through various states during their lifetime
- Scheduler makes decisions on which thread to run based on their state and priority
- Executors and concurrent utility classes simplify threaded programs

References

1. <http://manikandanmv.wordpress.com/tag/extends-thread-vs-implements-runnable/>
2. http://www.javamex.com/tutorials/threads/how_threads_work.shtml

