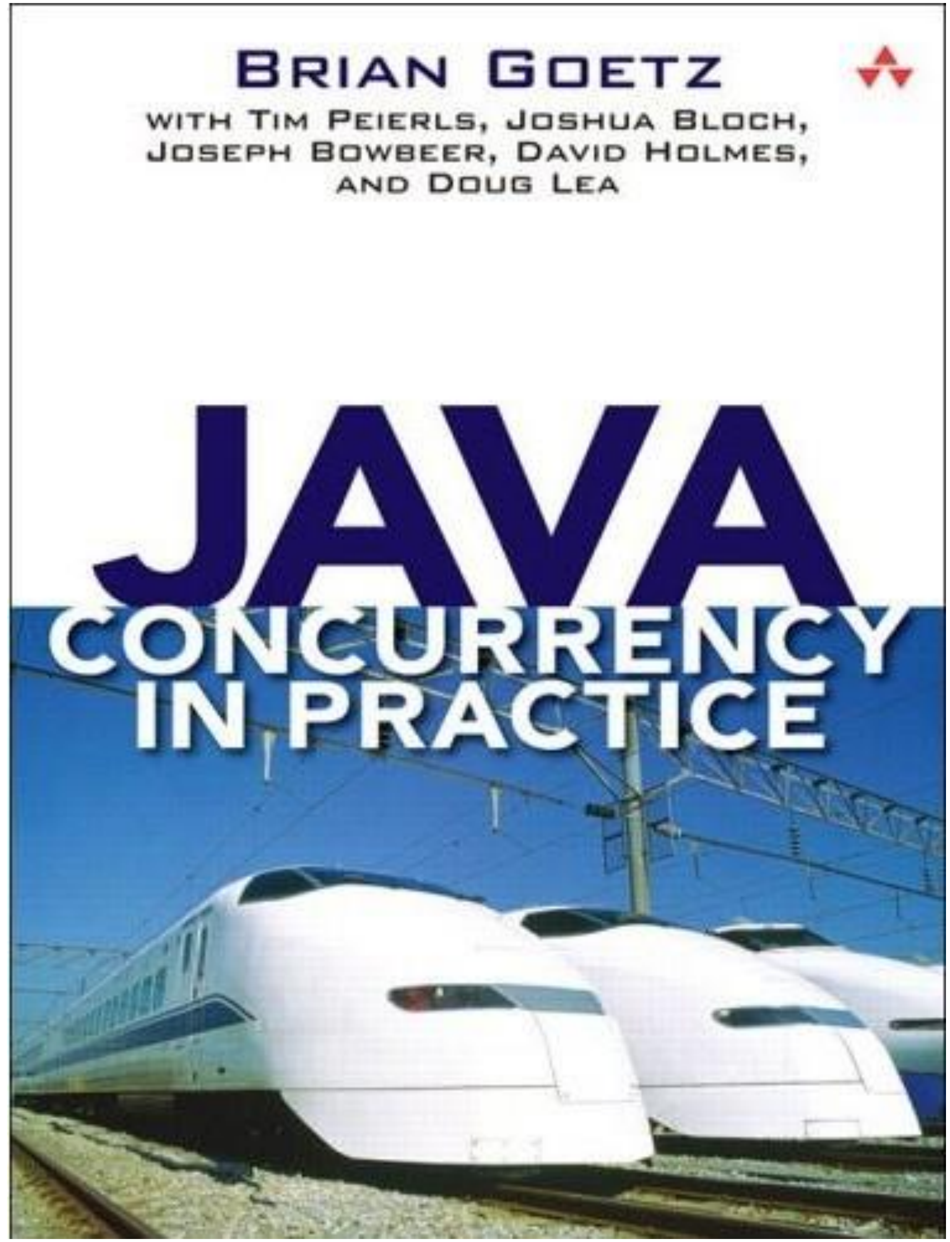
Northeastern University - Seattle

CS6650 Building Scalable Distributed Systems

Professor Ian Gorton

Building Scalable Distributed Systems

Week 2 – Introduction to Concurrent Systems http://jcip.net/



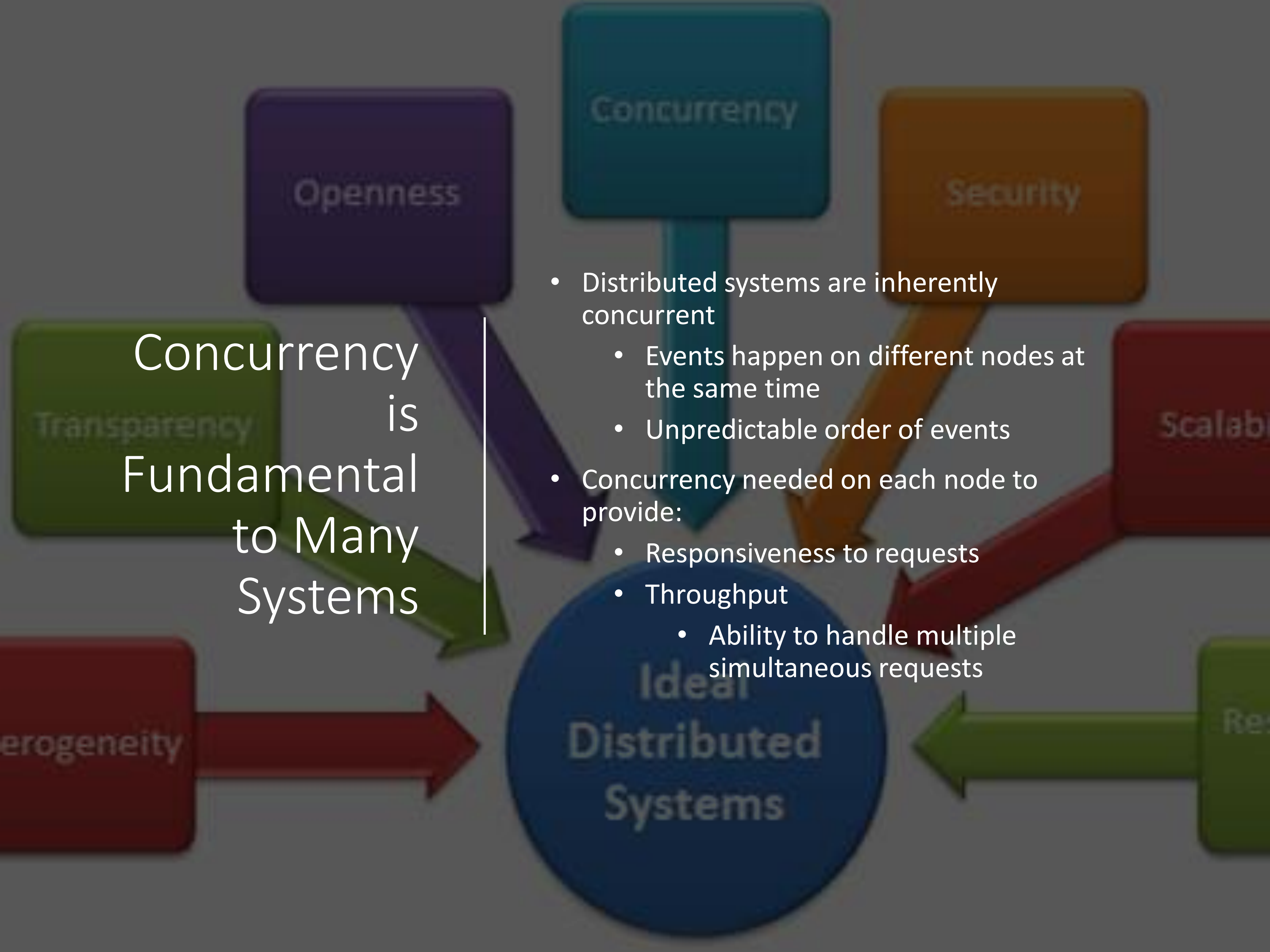
Overview

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

Learning objectives

1. Describe the difference between a thread and a process
2. Describe the two major problems that occur in multi-threaded systems
3. Write programs using threads in Java
4. Write programs using thread safe collections, queues and executors

Why Threads?

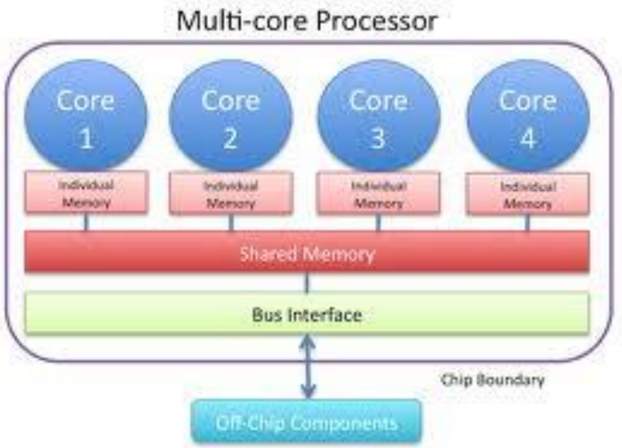


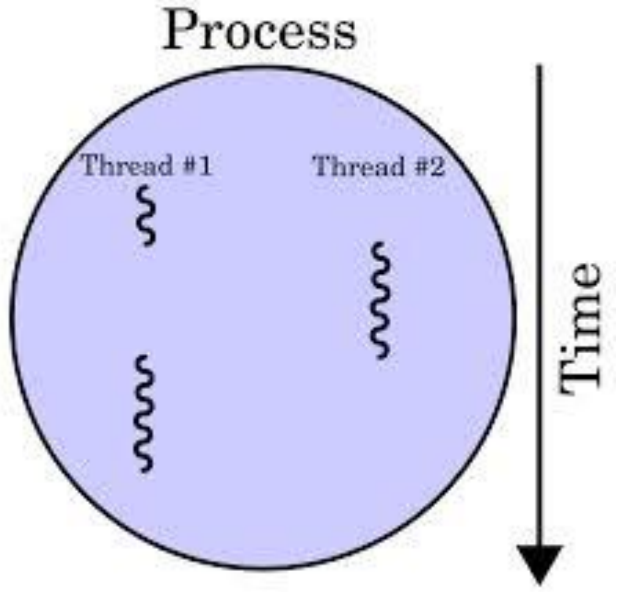
Concurrency is Fundamental to Many Systems

- 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?

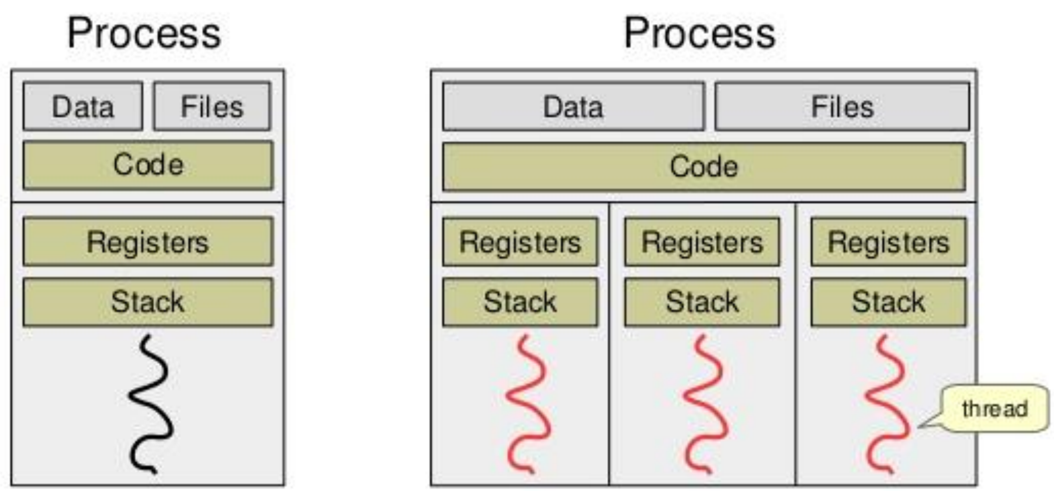
- Concurrent execution is necessary in many systems:
  - Natural solution to many problems
  - Increase performance, e.g. do work while waiting for disk accesses
  - Necessary to exploit multicore





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

```
class NamingThread implements Runnable {
      private String name;
       // default constructor
       public NamingThread() {}

        public NamingThread(String threadName) {
          System.out.println("Constructor called: " + threadName) ;
         name = threadName ;
        }

       public void setName (String threadName) {
           name = threadName;
       }
        public void run() {
          //Display info about this  thread
         System.out.println("Run called : " + name);
         System.out.println(name + " : " + Thread.currentThread());
         // and terminate silently ....
        }
}
```

Java Threads

```
public class SimpleThreadExample {
    public static void main(String[] args) {
      // create 3 threads and give them names
      NamingThread name0 = new NamingThread("thread0");
      NamingThread name1 = new NamingThread("thread1");
      NamingThread name2 = new NamingThread("thread2");
      //Create the threads
      Thread t0 = new Thread (name0);
      Thread t1 = new Thread (name1);
      Thread t2 = new Thread (name2);
      // start the threads
      t0.start();       t1.start();       t2.start();
      try {//delay the main thread for a second
         Thread.currentThread().sleep(1000);
      } catch (InterruptedException e) {
      }
      //Display info about the main thread and terminate
      System.out.println(Thread.currentThread());
    }
}
```

Java Threads

```
public class MyThread extends Thread {

    public void run(){
       System.out.println("MyThread running");
    }
  }

 // and to create and start ….
MyThread myThread = new MyThread();
  myTread.start();
```

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

Waiting for threads to terminate

```
//Create and start the threads
    Thread tids[] = new Thread[NUMTHREADS];
    for (int i=0; i < NUMTHREADS ; i++ ) {
      tids[i] = new Thread (runnables[i]);
      tids[i].start();
    }
    // wait for each one to finish
     try {
          for (int i=0; i < NUMTHREADS ; i++ ) {
               tids[i].join();
          }
     } catch (InterruptedException e) {
     }
```

Thread Interleaving

```
public class ThreadOverlap {
    public static void print() {
        for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName()  + " - iteration: " + i);
        }
    }
   public static void main(String[] args) {
                Runnable test = new Runnable() {
                @Override
                   public void run() { print(); }                    }
                 };
                new Thread(test, “Man City - Champions").start();
                new Thread(test, “Man United - Losers").start();
    }
}
```

Key Points so far

- Create threads by implementing a Runnable
- Start threads with .start() method, which returns immediately
- Threads execute independently until completion
- Order of thread execution is non-deterministic. Statements overlap.

One more thing …

- Threads and objects are orthogonal concepts
- Objects are basically state machines
  - State transitions invoked when methods called
- Threads define an execution context
  - Manipulate objects (state) they have reference to
  - The methods they call execute in the context of the calling thread

Problems with Threads

Concurreny 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
    - read value from memory to register
    - change value in register
    - write register value back to memory
  - thread 1: x=x+6
  - thread 2: x=x+1
- The result?

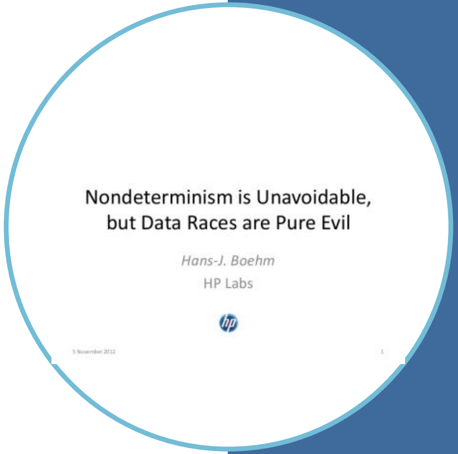
Welcome to Race Conditions

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



Root Cause: Non-Determinism

SEQUENTIAL PROGRAMS EXHIBIT DETERMINISTIC BEHAVIOR

RACE CONDITIONS ARE CAUSED BY **NON-DETERMINISTIC** BEHAVIOR

- 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

Synchronization primitives

Two common causes for race conditions

- Test and set
- Shared Variable Modification
- All code available in course repo
    - See Week 2

Race Condition

```
public class TestAndSetBroken {
       private int val =0;
        final static private int NUMTHREADS =10;
       public void testVal() {
               if (val == 0) {
                 System.out.println(Thread.currentThread().getName() + " - Value Set");
                 val = 1;
               }
               else {
                 System.out.println(Thread.currentThread().getName() + " - Value not modified");
               }
             }
   public static void main(String[] args) {
           final TestAndSetBroken tSet = new TestAndSetBroken();
           for (int i = 0; i < NUMTHREADS; i++) {
            new Thread(new Runnable() {
                 @Override
                   public void run() {          tSet.testVal();  }
                }, "Thread" + i).start();
            }
      }
}
```

Eradicating Race Conditions - Locks

- 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
- 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?
  - Locks sometimes known as semaphores

Test and Set – thread safe

```
public synchronized void testVal() {
  if (val == 0) {
      System.out.println(Thread.currentThread().getName() + " - Value Set");
       val = 1;
  } else {
     System.out.println(Thread.currentThread().getName() + " - Value not modified");
  }
}
```

Java synchronized methods

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

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.
  - synchronized (this);
  - synchronized(Object);
- Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor.

Shared Variable Modification – Race Condition

```
public class SharedVariableBroken {
  final static private int NUMTHREADS = 10000
  private int val = 0;

  public void inc() {
    val++;
  }
  public int getVal() {
    return this.val;
  }
  public static void main(String[] args) throws InterruptedException {
    final SharedVariableBroken rmw = new SharedVariableBroken();

    for (int i = 0; i < NUMTHREADS; i++) {
        // lambda runnable creation - interface only has a single method so lambda works fine
        Runnable thread =  () -> { rmw.inc(); };
          new Thread(thread).start();
    }
    Thread.sleep(5000);
    System.out.println("Value should be equal to " + NUMTHREADS + " It is: " + rmw.getVal());
  }
}
```

Compound operation – synchronized needed

Reproducing the error

- Lets run SharedVariableBroken.java
- Can you see the race condition?


Threads are EVIL!

Shared Variable Modification – Race Condition

```
public class SharedVariableBroken {
  final static private int NUMTHREADS = 10000
  private int val = 0;

  public void inc() {
    val++;
  }
  public int getVal() {
    return this.val;
  }
  public static void main(String[] args) throws InterruptedException {
    final SharedVariableBroken rmw = new SharedVariableBroken();

    for (int i = 0; i < NUMTHREADS; i++) {
        // lambda runnable creation - interface only has a single method so lambda works fine
        Runnable thread =  () -> { rmw.inc(); };
          new Thread(thread).start();
    }
    Thread.sleep(5000);
    System.out.println("Value should be equal to " + NUMTHREADS + " It is: " + rmw.getVal());
  }
}
```

Compound operation – synchronized needed

Another Solution – Atomic Variables

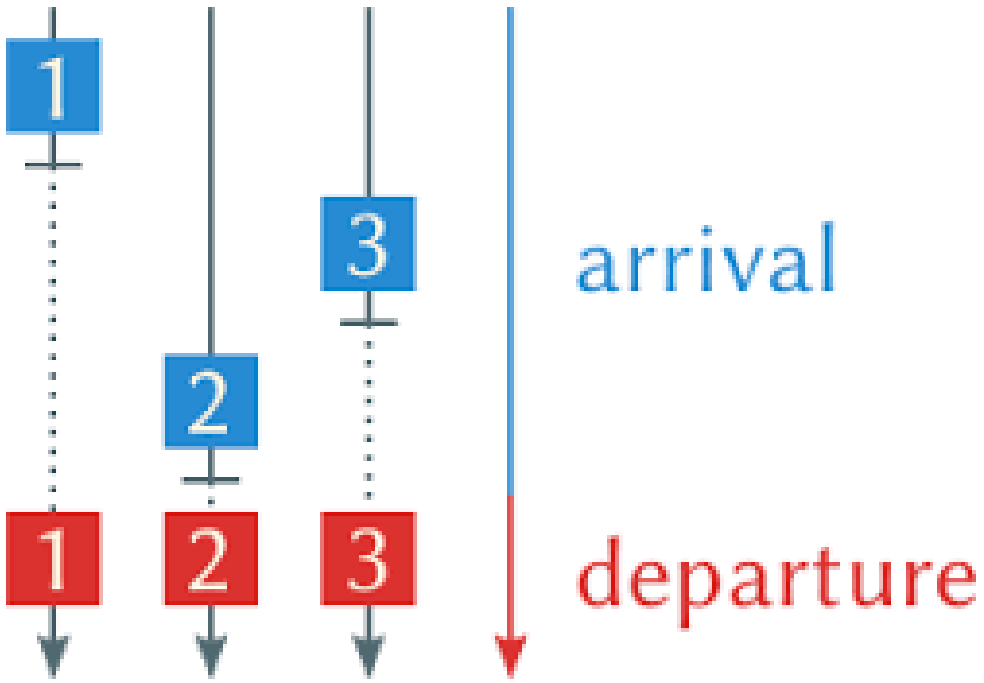
```
public class SharedVariable {
        private final AtomicInteger val= new AtomicInteger();

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

        public int getNumber() { val.get();}

        // rest is same as previous
```

Barrier Synchronization



CountDown Latch

- Implements a barrier
- Initialized with a given count
- await() blocks until count is zero
- countdown() method decrements value
- When count is zero, all threads resume
  - await() returns
- This is a one-shot phenomenon -- the count cannot be reset.
- If you need to reset the count, use a CyclicBarrier.

Barrier Synchronization - Latch

```
public static void main(String[] args) throws InterruptedException {
      final SharedVariable rmw = new SharedVariable();
      for (int i = 0; i < NUMTHREADS; i++) {
        Runnable thread =  () -> {
          try {
            rmw.startSignal.await();
            rmw.inc();
          } catch (InterruptedException e) {
          } finally {
             rmw.endSignal.countDown();
          }
        };
        new Thread(thread).start();
       } // end for loop

      rmw.startSignal.countDown();
      rmw.endSignal.await();
      System.out.println("Value should be " + NUMTHREADS + " - It is " + rmw.getVal());
   }
```

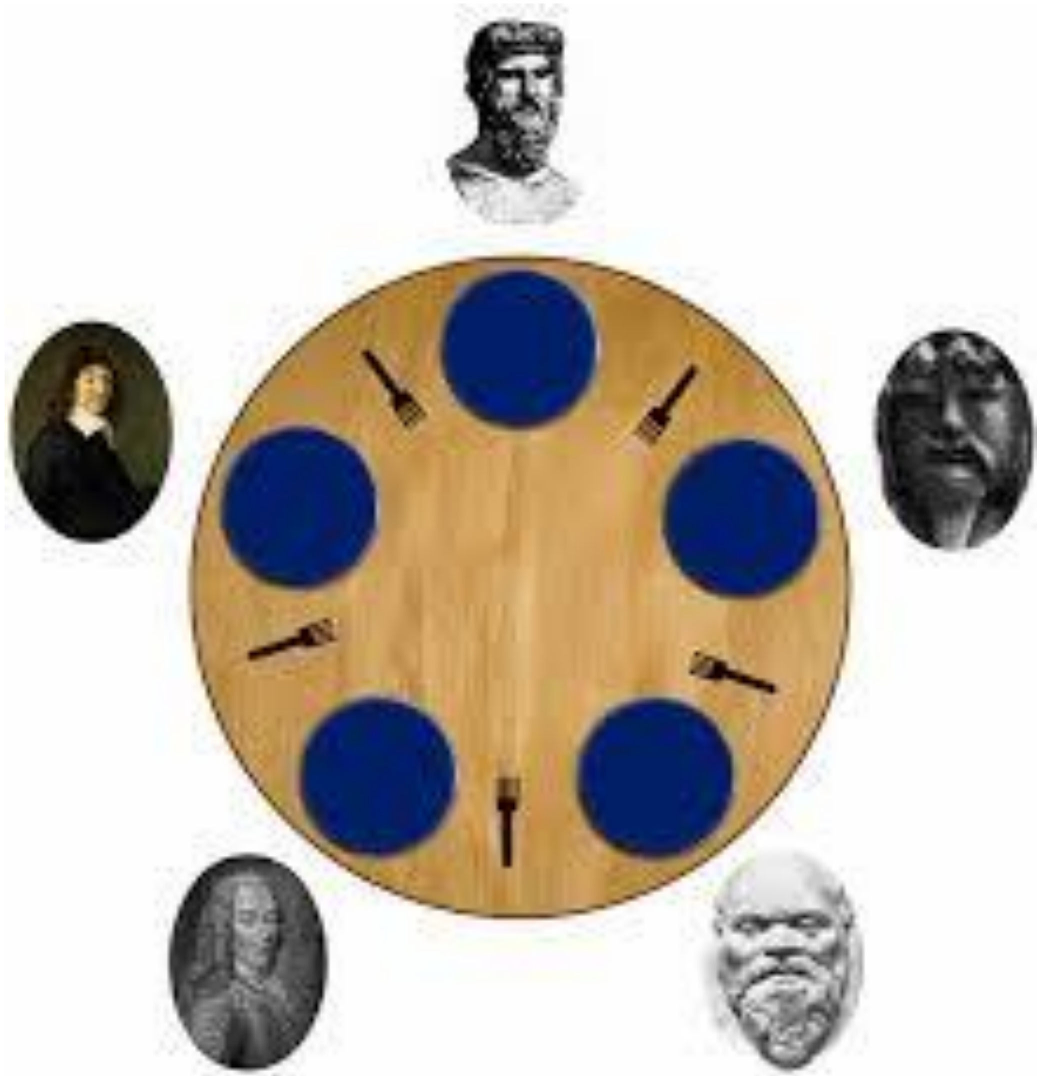
Sharing Structures

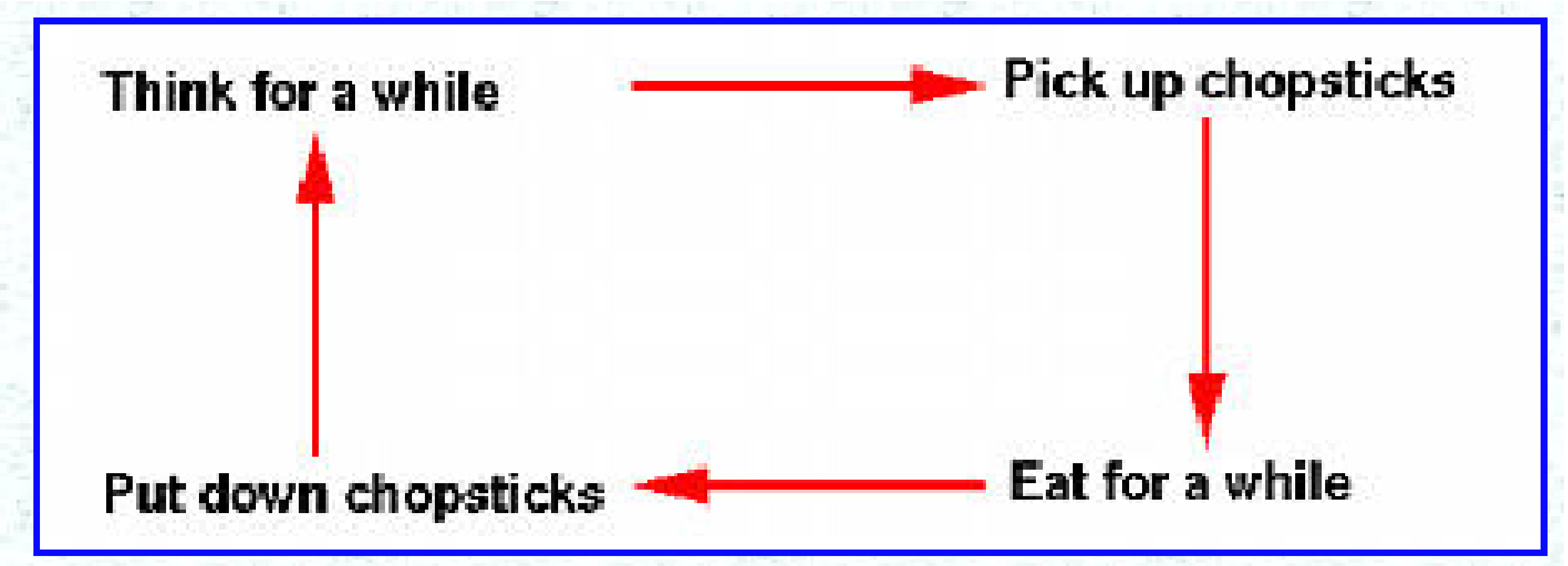
- Consider a linked list with explicit size variable
  - read size variable
  - add new element to the list at end
  - increment and write back size variable
- size variable and list elements must be synchronized
- concurrent access of non *thread-safe* structures is dangerous
  - none of *java.util.**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

Lab 2

Let’s play with servlets

The Dining Philosophers Problem




Think for a while
Pick up chopsticks
Eat for a while
Put down chopsticks

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 with real philosphers (and food!!)

```java
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
  - thread 1: takes lock a
  - thread 2: takes lock b
  - thread 1: blocks on b
  - thread 2: blocks on lock a
- *Deadlock!!*
  - *Neither thread can proceed*
  - *This violates 'liveness' – something good eventually happens*
  - *Circular wait*

Solution

```
for (int i = 0; i < philosophers.length; i++) {
        Object leftFork = forks[i];
        Object rightFork = forks[(i + 1) % forks.length];

        if (i == philosophers.length - 1) {

            // The last philosopher picks up the right fork first
            philosophers[i] = new Philosopher(rightFork, leftFork);
        } else {
            philosophers[i] = new Philosopher(leftFork, rightFork);
        }

        Thread t
          = new Thread(philosophers[i], "Philosopher " + (i + 1));
        t.start();
    }
```

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 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?

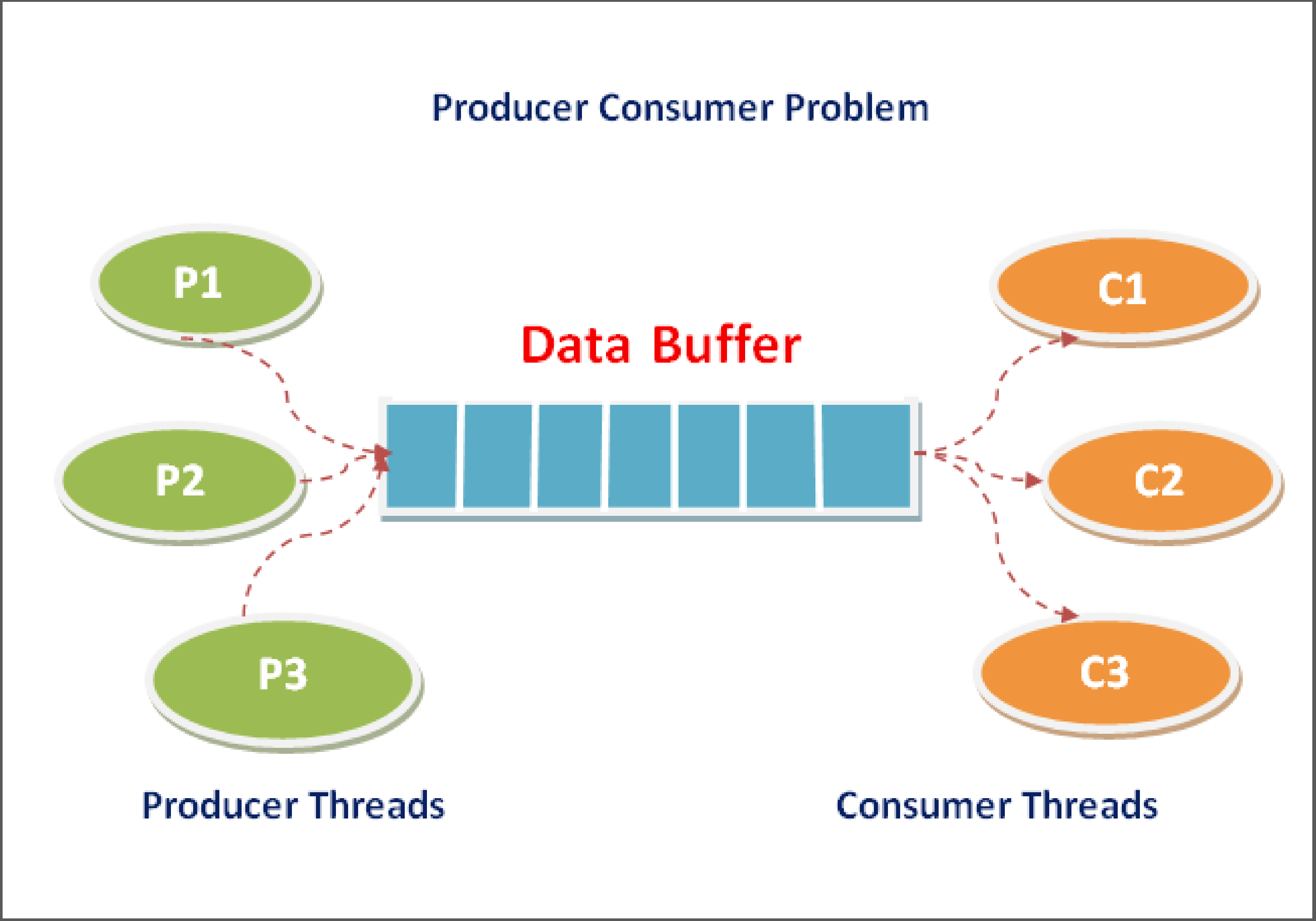
```
public class hipsterBaseClass  {
        public synchronized void doHipsterStuff() {
        // random hipster behaviour
        }
}

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

Thread coordination

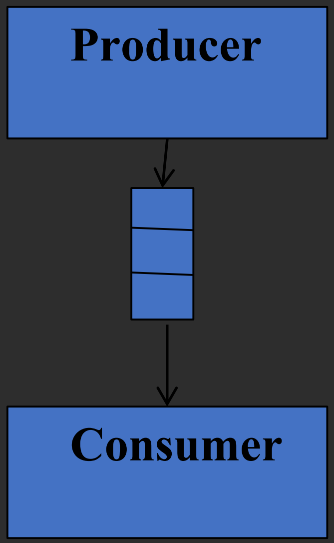

Producer Consumer Problem
P1
P2
P3
Data Buffer
C1
C2
C3
Producer Threads
Consumer Threads

Producer Consumer

- Need to ensure
  - Consumer waits if buffer empty
  - Consumer can remove item when buffer not empty
  - Producer waits if buffer full
  - Producer can add item if buffer has free space

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

- 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 one producer thread?
- More than one consumer thread?

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

- 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.

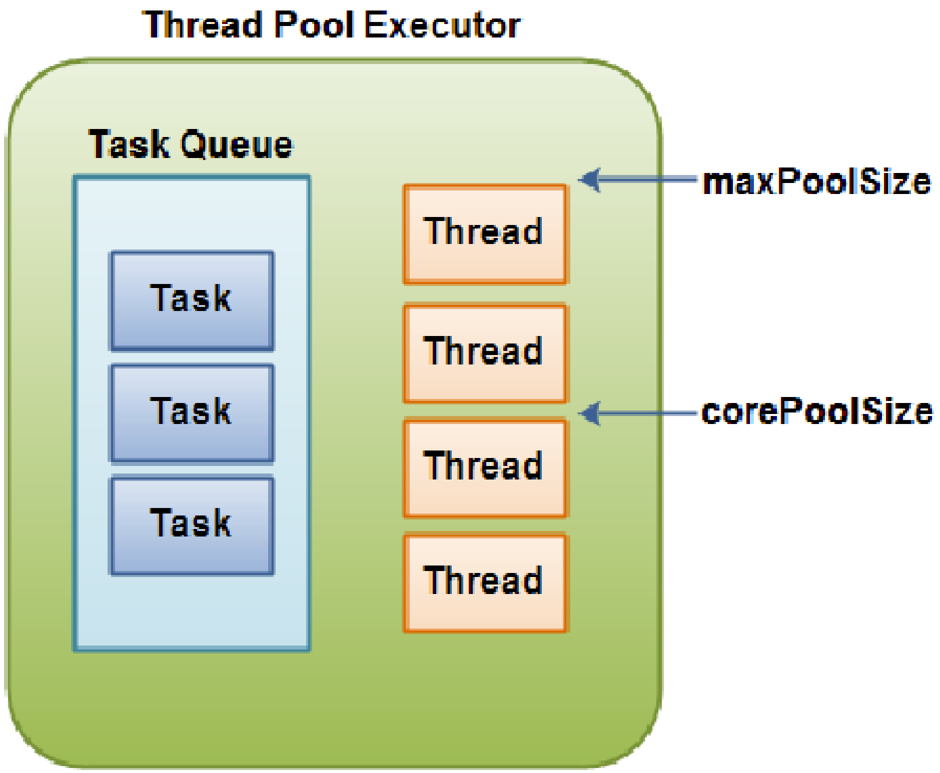
Executor – Fixed Size thread Pool

```
ExecutorService executorService =
            Executors.newFixedThreadPool(SIZE);

executorService.execute(new Runnable() {
   public void run() {
      System.out.println(“new  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

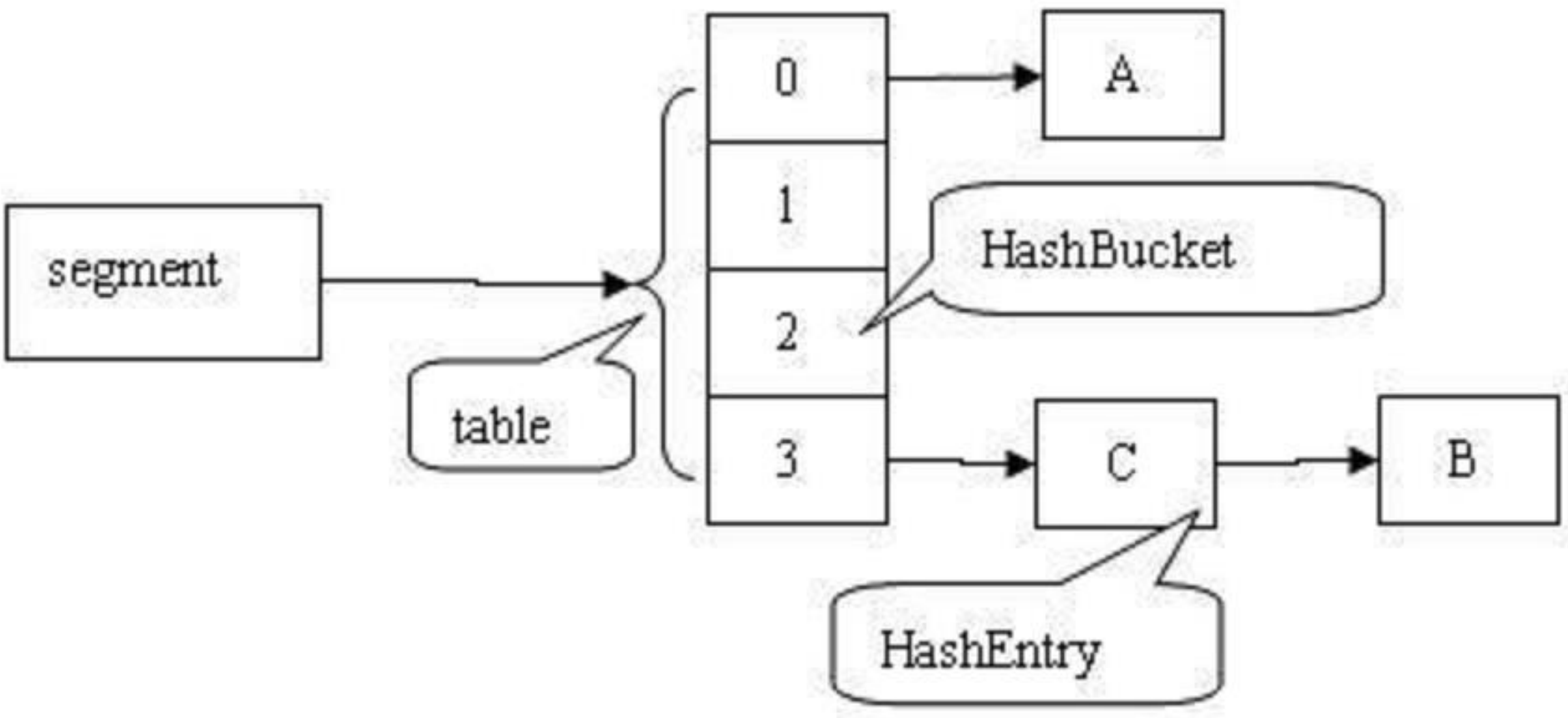
Thread-safe collection classes

- 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

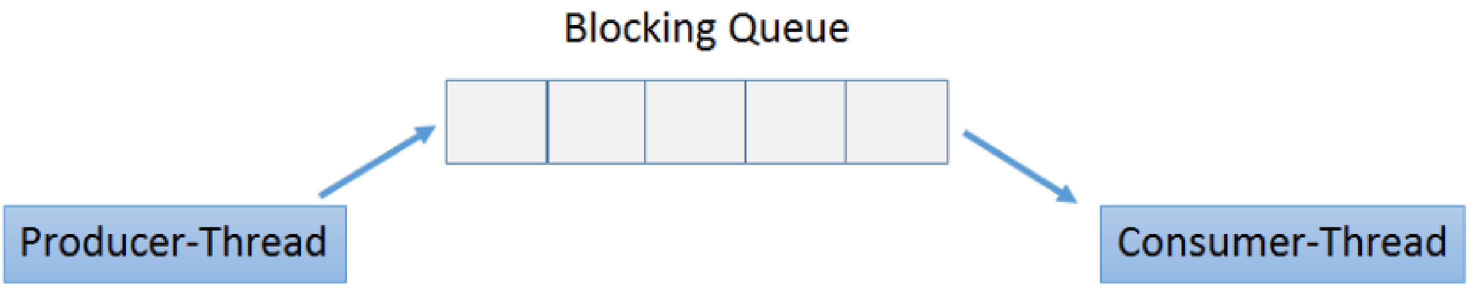
ConcurrentHashMap



ConcurrentHashMap

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

BlockingQueue



Put element in the queue and wait till space is available if queue is full.

Retrieve element from the queue and wait till element is available if queue is empty.

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

- http://manikandanmv.wordpress.com/tag/extends-thread-vs-implements-runnable/
- http://www.javamex.com/tutorials/threads/how_threads_work.shtml
- https://www.baeldung.com/java-dining-philoshophers


I ASK QUESTIONS.