Multi-Threaded Programming Design
CSCI 201
Principles of Software Development

Jeffrey Miller, Ph.D.
jeffrey.miller@usc.edu
Outline

• Blocking Queues

• Multi-Threaded Programming Design
Thread States Review

- **Start**: When a thread is started
- **Ready**: When the OS/JVM switches the thread into the CPU
- **Running**: When the OS/JVM switches the thread out of the CPU or the thread yields the CPU
- **Waiting**: When a thread is signaled/notified based on the resource on which it is waiting
- **Dead**: When a thread has completed execution
- **Sleeping**: When the amount of time specified for sleeping has elapsed

- **Start** → **Ready**:
  - When a thread is started

- **Ready** → **Running**:
  - When the OS/JVM switches the thread into the CPU
  - When the OS/JVM switches the thread out of the CPU or the thread yields the CPU

- **Running** → **Dead**:
  - When a thread puts itself to sleep for a certain amount of time
  - When a thread has completed execution

- **Running** → **Waiting**:
  - When a thread waits on a resource to become available

- **Waiting** → **Ready**:
  - When a thread is signaled/notified based on the resource on which it is waiting

- **Dead** → **Sleeping**:
  - When the amount of time specified for sleeping has elapsed
Producer/Consumer Problem Review

- There are two classes – **Producer** and **Consumer**
- In a shared variable (buffer in the following example), the **Producer** increases the value and the **Consumer** decreases the value
- The shared variable has a maximum capacity that the value cannot exceed (**CAPACITY** in the following example)
  - If the **Producer** tries to add a value when the buffer has reached its capacity, it must wait for the **Consumer** (with the condition **notFull** in the following example)
- The shared variable has a minimum capacity that the value cannot pass (0 in the following example)
  - If the **Consumer** tries to decrease the value when the buffer has reached its minimum capacity, it must wait for the **Producer** (with the condition **notEmpty** in the following example)
Producer/Consumer Example with Monitors

```java
import java.util.LinkedList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ProducerConsumerWithMonitors {
    private static Buffer buffer = new Buffer();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(new ProducerTask());
        executor.execute(new ConsumerTask());
        executor.shutdown();
    }

    private static class ProducerTask implements Runnable {
        public void run() {
            try {
                int i = 1;
                while (true) {
                    System.out.println("Producer writes: " + i);
                    buffer.write(i);
                    Thread.sleep((int)(Math.random() * 1000));
                }
            } catch (InterruptedException ie) {
                System.out.println("Producer IE: " + ie.getMessage());
            }
        }
    }

    private static class ConsumerTask implements Runnable {
        public void run() {
            try {
                while (true) {
                    System.out.println("Consumer reads: " + buffer.read());
                    Thread.sleep((int)(Math.random() * 1000));
                }
            } catch (InterruptedException ie) {
                System.out.println("Consumer IE: " + ie.getMessage());
            }
        }
    }

    private static class Buffer {
        private static final int CAPACITY = 1;
        private LinkedList<Integer> queue = new LinkedList<Integer>();
        private static Object notEmpty = new Object();
        private static Object notFull = new Object();
        public void write(int value) {
            synchronized(notFull) {
                synchronized(notEmpty) {
                    try {
                        while (queue.size() == CAPACITY) {
                            System.out.println("Wait for notFull condition " + value);
                            notFull.wait();
                        }
                        queue.offer(value);
                        notEmpty.notify();
                    } catch (InterruptedException ie) {
                        System.out.println("Buffer.write IE: " + ie.getMessage());
                    }
                }
            }
        }
        public int read() {
            int value = 0;
            synchronized(notFull) {
                synchronized(notEmpty) {
                    try {
                        while (queue.isEmpty()) {
                            System.out.println("Wait for notEmpty condition");
                            notEmpty.wait();
                        }
                        value = queue.remove();
                        notFull.notify();
                    } catch (InterruptedException ie) {
                        System.out.println("Buffer.read IE: " + ie.getMessage());
                    }
                }
            }
            return value;
        }
    }
}
```
import java.util.LinkedList;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ProducerConsumerWithLocks {
    private static Buffer buffer = new Buffer();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(new ProducerTask());
        executor.execute(new ConsumerTask());
        executor.shutdown();
    }

    private static class ProducerTask implements Runnable {
        public void run() {
            try {
                int i = 1;
                while (true) {
                    System.out.println("Producer tries to write: " + i);
                    buffer.write(i);
                    Thread.sleep((int)(Math.random() * 1000));
                }
            } catch (InterruptedException ie) {
                System.out.println("Producer IE: " + ie.getMessage());
            }
        }
    }

    private static class ConsumerTask implements Runnable {
        public void run() {
            try {
                while (true) {
                    System.out.println("Consumer reads: " + buffer.read());
                    Thread.sleep((int)(Math.random() * 1000));
                }
            } catch (InterruptedException ie) {
                System.out.println("Consumer IE: " + ie.getMessage());
            }
        }
    }

    private static class Buffer {
        private static final int CAPACITY = 1;
        private LinkedList<Integer> queue = new LinkedList<Integer>();
        private static Lock lock = new ReentrantLock();
        private static Condition notEmpty = lock.newCondition();
        private static Condition notFull = lock.newCondition();
        public void write(int value) {
            lock.lock();
            try {
                while (queue.size() == CAPACITY) {
                    System.out.println("Wait for notFull condition " + value);
                    notFull.await();
                }
                queue.offer(value);
                notEmpty.signal();
            } catch (InterruptedException ie) {
                System.out.println("Buffer.write IE: " + ie.getMessage());
            } finally {
                lock.unlock();
            }
        }

        public int read() {
            int value = 0;
            lock.lock();
            try {
                while (queue.isEmpty()) {
                    System.out.println("Wait for notEmpty condition");
                    notEmpty.await();
                }
                value = queue.remove();
                notFull.signal();
            } catch (InterruptedException ie) {
                System.out.println("Buffer.read IE: " + ie.getMessage());
            } finally {
                lock.unlock();
            }
            return value;
        }
    }
}

private static class Buffer {
    private static final int CAPACITY = 1;
    private LinkedList<Integer> queue = new LinkedList<Integer>();
    private static Lock lock = new ReentrantLock();
    private static Condition notEmpty = lock.newCondition();
    private static Condition notFull = lock.newCondition();
    public void write(int value) {
        lock.lock();
        try {
            while (queue.size() == CAPACITY) {
                System.out.println("Wait for notFull condition " + value);
                notFull.await();
            }
            queue.offer(value);
            notEmpty.signal();
        } catch (InterruptedException ie) {
            System.out.println("Buffer.write IE: " + ie.getMessage());
        } finally {
            lock.unlock();
        }
    }

    public int read() {
        int value = 0;
        lock.lock();
        try {
            while (queue.isEmpty()) {
                System.out.println("Wait for notEmpty condition");
                notEmpty.await();
            }
            value = queue.remove();
            notFull.signal();
        } catch (InterruptedException ie) {
            System.out.println("Buffer.read IE: " + ie.getMessage());
        } finally {
            lock.unlock();
        }
        return value;
    }
} // ends class Buffer

} // ends class ProducerConsumerWithLocks
Producer/Consumer Output

Producer writes: 1  
Consumer reads: 1
Wait for notEmpty condition

Producer writes: 2  
Consumer reads: 2
Wait for notEmpty condition

Producer writes: 3  
Consumer reads: 3

Producer writes: 4  
Consumer reads: 4

Producer writes: 5  
Consumer reads: 5
Wait for notEmpty condition

Producer writes: 6  
Consumer reads: 6
Wait for notEmpty condition

Producer writes: 7  
Consumer reads: 7
Wait for notEmpty condition

Producer writes: 8  
Consumer reads: 8
Wait for notEmpty condition

Producer writes: 9  
Consumer reads: 9
Wait for notEmpty condition

Producer writes: 10  
Consumer reads: 10
Wait for notEmpty condition

Producer writes: 1  
Wait for notEmpty condition
Consumer reads: 1

Producer writes: 2  
Consumer reads: 2
Wait for notEmpty condition

Producer writes: 3  
Consumer reads: 3
Wait for notEmpty condition

Producer writes: 4  
Consumer reads: 4
Wait for notEmpty condition

Producer writes: 5  
Consumer reads: 5
Wait for notEmpty condition

Producer writes: 6  
Consumer reads: 6
Wait for notEmpty condition

Producer writes: 7  
Consumer reads: 7
Wait for notEmpty condition

Producer writes: 8  
Consumer reads: 8
Wait for notEmpty condition

Producer writes: 9
Blocking Queues

- A blocking queue causes a thread to block (i.e. move to the waiting state) when you try to add an element to a full queue or to remove an element from an empty queue
  - It will remain there until the queue is no longer full or no longer empty
  - There are three blocking queues in Java: `ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`
import java.util.concurrent.*;
public class ProducerConsumer {
    private static ArrayBlockingQueue<Integer> buffer = new ArrayBlockingQueue<Integer>(2);
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(new ProducerTask());
        executor.execute(new ConsumerTask());
        executor.shutdown();
    }
    private static class ProducerTask implements Runnable {
        public void run() {
            try {
                int i = 1;
                while (true) {
                    System.out.println("Producer writes: "+i);
                    buffer.put(i++);
                    Thread.sleep((int)(Math.random() * 10000));
                }
            } catch (InterruptedException ie) {
                System.out.println("Producer IE: " + ie.getMessage());
            }
        }
    }
    private static class ConsumerTask implements Runnable {
        public void run() {
            try {
                while (true) {
                    System.out.println("\t\t\tConsumer reads: " + buffer.take());
                    Thread.sleep((int)(Math.random() * 10000));
                }
            } catch (InterruptedException ie) {
                System.out.println("Consumer IE: " + ie.getMessage());
            }
        }
    }
}
Outline

• Blocking Queues
• Multi-Threaded Programming Design
Avoiding Deadlock

- Deadlock can occur when two threads are both waiting on locks the other thread has.

- This can be avoided if locks are obtained in the same order every time:
  - If the lock on `object1` is always obtained before the lock on `object2`, deadlock will be avoided in the above example.
  - NOTE: If Thread1 waits on `object1` inside the `object2` synchronization, deadlock can still occur (which is what we did in the ProducerConsumer example with monitors).
Java Collections Synchronization

- The classes in the Java Collections framework are not thread-safe
  - Vector, Stack, and Hashtable are thread-safe, but they are older objects (since version 1.0)
  - They have been replaced by ArrayList, LinkedList, and Map (since version 1.2)
- There are methods in the Collections class that can be used for obtaining thread-safe versions of any of the Collection objects
  - A synchronized collection object has synchronized versions of all the methods that access and update the original collection
  - Note: There are many more methods in the Collections class

```java
java.util.Collections
+synchronizedCollection(c: Collection): Collection
+synchronizedList(list: List): List
+synchronizedMap(map: Map): Map
+synchronizedSet(set: Set): Set
+synchronizedSortedMap(map: SortedMap): SortedMap
+synchronizedSortedSet(set: SortedSet): SortedSet

Returns a synchronized collection.
Returns a synchronized list from the specified list.
Returns a synchronized map from the specified map.
Returns a synchronized set from the specified set.
Returns a synchronized sorted map from the specified sorted map.
Returns a synchronized sorted set.
```
Iterators and Synchronized Collections

- Even though we can get a synchronized Collections object, the iterator is fail-fast (not synchronized)
  - If the collection being iterated over is modified by another thread, the iterator will throw a java.util.ConcurrentModificationException
  - We can avoid this by obtaining a lock on the object over which we are iterating before we begin iterating
Synchronized Collections Example #1

```java
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class CollectionsTest {
    public static void main(String[] args) {
        for (int i=0; i < 100; i++) {
            MyThread mt = new MyThread(i);
            mt.start();
        }
    }
}

class MyThread extends Thread {
    private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
    private int num;
    public MyThread(int num) {
        this.num = num;
        hashSet.add(num);
    }
    public void run() {
        System.out.print("thread "+num+" ");
        Iterator<Integer> iterator = hashSet.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println();
    }
}
```

Exception in thread "Thread-6" Exception in thread "Thread-0"
```java
import java.util.*;
import java.util.concurrent.locks.*;

public class CollectionsTest {
    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            MyThread mt = new MyThread(i);
            mt.start();
        }
    }
}

class MyThread extends Thread {
    private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
    private static Lock lock = new ReentrantLock();
    private int num;
    public MyThread(int num) {
        this.num = num;
        hashSet.add(num);
    }
    public void run() {
        lock.lock();
        try {
            System.out.print("thread "+num +": ");
            Iterator<Integer> iterator = hashSet.iterator();
            while (iterator.hasNext()) {
                System.out.print(iterator.next()+" ");
            }
            System.out.println();
        } finally {
            lock.unlock();
        }
    }
}
```

```
thread 0: 0 1 2 3 4 5 6 7 8 9 10 11 12
thread 2: 0 1 2 3 4 5 6 7 8 9 10 11 12
thread 5: 0 Exception in thread "Thread-5" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
    at java.util.HashMap$KeyIterator.next(Unknown Source)
    at MyThread.run(Test.java:34)
thread 7: 0 Exception in thread "Thread-7" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
    at java.util.HashMap$KeyIterator.next(Unknown Source)
    at MyThread.run(Test.java:34)
thread 6: 0 1 2 3 Exception in thread "Thread-6" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
    at java.util.HashMap$KeyIterator.next(Unknown Source)
    at MyThread.run(Test.java:34)
thread 8: 0 1 2 Exception in thread "Thread-8" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
    at java.util.HashMap$KeyIterator.next(Unknown Source)
    at MyThread.run(Test.java:34)
thread 1: 0 Exception in thread "Thread-1" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
    at java.util.HashMap$KeyIterator.next(Unknown Source)
    at MyThread.run(Test.java:34)
```

import java.util.*;
import java.util.concurrent.locks.*;

public class CollectionsTest {
    public static void main(String [] args) {
        for (int i=0; i < 100; i++) {
            MyThread mt = new MyThread(i);
            mt.start();
        }
    }
}

class MyThread extends Thread {
    private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
    private static Lock lock = new ReentrantLock();
    private int num;

    public MyThread(int num) {
        this.num = num;
        lock.lock();
        try {
            hashSet.add(num);
        } finally {
            lock.unlock();
        }
    }

    public void run() {
        lock.lock();
        try {
            System.out.print("thread "+num+": ");
            Iterator<Integer> iterator = hashSet.iterator();
            System.out.print(iterator.next() + " ");
            System.out.println();
        } finally {
            lock.unlock();
        }
    }
}

thread 0: 0 1 2
thread 1: 0 1 2
thread 2: 0 1 2 3
thread 3: 0 1 2 3 4 5
thread 4: 0 1 2 3 4 5 6
thread 5: 0 1 2 3 4 5 6 7 8
thread 6: 0 1 2 3 4 5 6 7 8
thread 7: 0 1 2 3 4 5 6 7 8 9
thread 8: 0 1 2 3 4 5 6 7 8 9 10
thread 9: 0 1 2 3 4 5 6 7 8 9 10
thread 10: 0 1 2 3 4 5 6 7 8 9 10
thread 11: 0 1 2 3 4 5 6 7 8 9 10 11
thread 12: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
thread 13: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
thread 14: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1. import java.util.*;
2.
3. public class CollectionsTest {
4.   public static void main(String [] args) {
5.       for (int i=0; i < 100; i++) {
6.           MyThread mt = new MyThread(i);
7.           mt.start();
8.       }
9.   }
10. }
11. class MyThread extends Thread {
12.   private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
13.   private int num;
14.   public MyThread(int num) {
15.       this.num = num;
16.       synchronized(hashSet) {
17.           hashSet.add(num);
18.       }
19.   }
20.   public void run() {
21.       synchronized(hashSet) {
22.           System.out.print("thread " + num + ": ");
23.           Iterator<Integer> iterator = hashSet.iterator();
24.           while (iterator.hasNext()) {
25.               System.out.print(iterator.next() + " ");
26.           }
27.           System.out.println();
28.       }
29.   }
30. }

thread 0: 0 1 2
thread 1: 0 1 2
thread 2: 0 1 2 3
thread 3: 0 1 2 3 4 5
thread 4: 0 1 2 3 4 5 6
thread 5: 0 1 2 3 4 5 6 7 8
thread 6: 0 1 2 3 4 5 6 7 8
thread 7: 0 1 2 3 4 5 6 7 8 9
thread 8: 0 1 2 3 4 5 6 7 8 9 10
thread 9: 0 1 2 3 4 5 6 7 8 9 10
thread 10: 0 1 2 3 4 5 6 7 8 9 10
thread 11: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
import java.util.*;

public class CollectionsTest {
    public static void main(String [] args) {
        for (int i=0; i < 100; i++) {
            MyThread mt = new MyThread(i);
            mt.start();
        }
    }
}

class MyThread extends Thread {
    private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
    private int num;
    public MyThread(int num) {
        this.num = num;
        hashSet.add(num);
    }
    public void run() {
        synchronized (hashSet) {
            System.out.print("thread "+num+": ");
            Iterator<Integer> iterator = hashSet.iterator();
            while (iterator.hasNext()) {
                System.out.print(iterator.next()+" ");
            }
            System.out.println();
        }
    }
}
Multi-Threaded Programming Rules

▪ Use one lock for one resource
▪ Always acquire locks in the same order in different threads
▪ Always release locks in the opposite order they were acquired
  › If acquiring multiple locks, never release the outer lock without releasing the inner lock first
▪ Always synchronize iterating through a data structure if the variable is shared across multiple threads
▪ Use the Collections framework for creating thread-safe data structures instead of Vector, Stack, and Hashtable