Synchronisation in Java

GURUTEAM
specialist ICT learning

Basic Data Access Synchronization

- Every Java object has a lock
  - guarantees mutual exclusion
- Basis of protecting shared data
  - synchronized method acquires lock before proceeding
  - only one thread in any synchronized method of object at a time

```
public synchronized void deposit ( int amount )
{
  int curBal = getBalance();
  curBal += amount;
  setBalance( curBal );
}
```

Synchronized Blocks

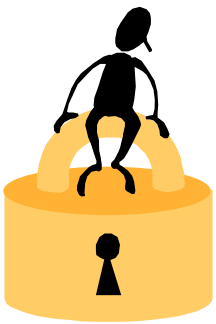
- More flexible than synchronizing entire methods
  - leads to finer grained locking
  - offers potentially higher throughput
- Can synchronise on any object

```
synchronized ( myAccount ) {
  int curBal = getBalance();
  curBal += amount;
  setBalance( curBal );
}
```

```
synchronized ( yourAccount ) {
  int curBal = getBalance();
  curBal += amount;
  setBalance( curBal );
}
```


Lock Objects

- Java objects have always had locks
  - used to implement synchronized keyword
  - one mutex-style lock per object
- Existing approach has limitations
- No timeout when waiting for a lock
  - deadlock
- No non-blocking check on lock availability
- Only one mutex per object
  - lacks flexibility
  - no shared/exclusive locks



The Lock Interface

- Aims to tackle deficiencies in built in locking
  - allow different lock strategies

```
interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit)
                          throws InterruptedException;
 void unlock();
 Condition newCondition() throws UnsupportedOperationException;
 }
```

- No automatic lock release
  - use try/finally

```
Lock l = …;
l.lock();
try {
  // access resource protected by the lock
} finally {
  l.unlock();
}
```


ReentrantLock

- Implementation of Lock interface
  - similar to built in monitor lock used in synchronized blocks
- Provides option of "fair" locking
  - lock granted to threads in order of requests
  - not supported by intrinsic locking
  - degrades performance substantially

ReadWriteLock

- Implements shared/exclusive locking
  - more efficient for read-mostly resources
  - composed of two Lock objects

```
public class ReadWriteMap {
  final Map <String, Data> m = new TreeMap<String, Data>;
  final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
  final Lock r = rwl.readLock();
  final Lock w = rwl.writeLock();

  public Data get ( String key ) {
    r.lock();
    try {
      return m.get(key);
    } finally { r.unlock(); }
  }
```

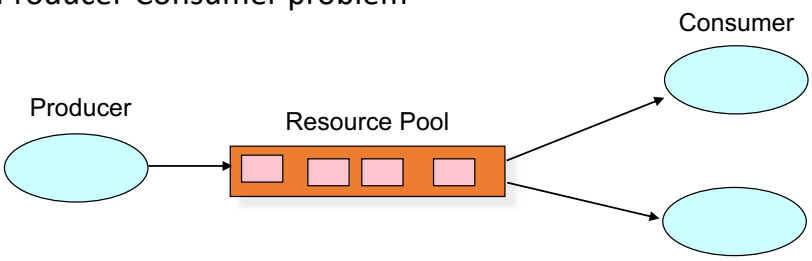

ReadWriteLock

- Implements shared/exclusive locking
  - more efficient for read-mostly resources
  - composed of two Lock objects

```
  public Data put ( String key, Data value ) {
    w.lock();
    try {
      return m.put(key,value);
    } finally { w.unlock(); }
  }
  …
}
```

Synchronizing Execution

- Producer-Consumer problem



- Cannot consume if resource pool is empty
  - cannot produce if pool is full
- Synchronization between producers and consumers is necessary

Example Producer

```
public class Producer extends Thread {
   private Letters pool;
   private final static String ALPHABET =
                                  "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

   public Producer(Letters thePool, String threadName) {
      super(threadName);    // Sets the thread's name
      this.pool = thePool;  // Reference to the shared resource
   }
```

Example Producer

```
    public void run() {
      char ch;
      // Add 10 letters to the pool
      for (int i = 0; i < 10; i++) {
         ch = ALPHABET.charAt((int)(Math.random() * 26));
         pool.addLetter(ch);
         // Diagnostic print
         System.out.println("Thread: " + getName()
                                        + " added " + ch );
         // Random wait before we add the next letter
         try{
            sleep((int)(Math.random() * 100));
         } catch (InterruptedException e ){}
      }
   }
}
```


Example Consumer

```
public class Consumer extends Thread {
   private Letters pool;

   public Consumer( Letters thePool, String threadName ) {
      super(threadName);     // Sets the thread's name
      this.pool = thePool;   // Reference to the shared resource
   }
```

Example Consumer

```
    public void run () {
      char ch;
      // Take 10 letters from the pool
      for(int i = 0; i < 10 ; i++) {
         ch = pool.takeLetter();
         // Diagnostic print
         System.out.println("Thread: " + getName() +
                                  " took letter : " + ch);
         // Random wait before we grab the next letter
         try {
            sleep((int)(Math.random() *  2000));
         } catch ( InterruptedException e ){}
      }
   }
}
```


Driver Program and Shared Resource

- Shared resource represented as an interface
  - allows different implementations to be tried…

```
public interface Letters {
  void addLetter(char c);
  char takeLetter();
}
```

```
public class ProducerConsumerDriver {
  private static Letters letterPool = new LettersImplClass();
  public static void main(String[] args) {
    // 2 producers and 2 consumers for now…
    new Producer(letterPool, "Producer1" ).start();
    new Producer(letterPool, "Producer2" ).start();
    new Consumer(letterPool, "Consumer1" ).start();
    new Consumer(letterPool, "Consumer2" ).start();
  }
}
```

Basic Synchronization

- wait() blocks calling thread until some condition is satisfied
  - use object to represent notification point
- notifyAll() wakes up all threads blocked in wait()
- Must hold lock on object
  - wait() atomically releases lock before blocking thread
- Threads should check condition before proceeding
  - maybe only one may proceed

```
synchronized ( obj ) {
  while ( condition == false ) {
    try {
      obj.wait();
    } catch ( InterruptedException ie ) {}
  }
}
```

```
synchronized ( obj ) {
  condition = true;
  obj.notifyAll();
}
```


Solution Using Basic Synchronization

```
public class LettersImplBasic implements Letters {

   private final static int BUFFER_CAPACITY = 6;
   private char[] buffer = new char[BUFFER_CAPACITY];
   private int next = 0;
   private boolean isFull = false;
   private boolean isEmpty = true;
```

Solution Using Basic Synchronization

```
public synchronized void addLetter(char ch) {
  // wait until pool has room for new letter
  while ( isFull ) {
     try {
        wait();
     } catch ( InterruptedException e ) {}
  }
  // add the letter to the next available spot
  buffer[next++] = ch;
  // are we full?
  if (next == BUFFER_CAPACITY) {
     isFull = true;
  }
  isEmpty = false;
  notifyAll();
}
```


Solution Using Basic Synchronization

```
public synchronized char takeLetter(){
  // wait until the pool becomes non-empty
  while (isEmpty == true) {
     try {
        wait(); // exit this when isEmpty turns false
     } catch (InterruptedException e) {}
  }
  // decrement the count, since we're going to
  // remove a letter
  next--;
  // Was this the last letter?
  if (next == 0){
     isEmpty = true;
  }
  // we know the pool can't be full
  isFull = false;
  notifyAll();
  return(buffer[next]); // return char
}
}
```

Conditions

- Generalisation of condition variable facility built in to each object
  - wait() / notify() / notifyAll()
- Obtained from Lock object
  - newCondition() method
  - ties condition variable to lock
- Supports similar operations, different method names
  - await() / signal() / signalAll()
  - possible to place timeout on wait operations

Letters Implementation Using Conditions

```
public class LettersImplLockobj implements Letters {
  private final static int BUFFER_CAPACITY = 5;
  private char[] buffer = new char[BUFFER_CAPACITY];
  private int next = 0;
  private final Lock bufferLock = new ReentrantLock();
  private final Condition notFull = bufferLock.newCondition();
  private final Condition notEmpty = bufferLock.newCondition();
  public void addLetter(char c) {
    bufferLock.lock();
    try {
      while ( next == BUFFER_CAPACITY ) {
        try {
          notFull.await();
        } catch ( InterruptedException ignore ) {  }
      }
      buffer[next++] = c;
      notEmpty.signalAll();
    } finally {
      bufferLock.unlock();
    }
  }
```

Letters Implementation Using Conditions

```
  public char takeLetter() {
    bufferLock.lock();
    try {
      while ( next == 0 ) {
        try {
          notEmpty.await();
        } catch ( InterruptedException ignore ) {  }
      }

      next--;
      notFull.signalAll();
      return(buffer[next]);
    } finally {
      bufferLock.unlock();
    }
  }
}
```


Semaphores

- Synchronization class
  - implements Dijkstra counting semaphore
  - resource counter
- Sometimes used as a lock
  - only 1 resource available

```
public class LettersImplSema4 implements Letters {

  private static final int BUFFER_CAPACITY = 5;
  private char [] buffer = new char[BUFFER_CAPACITY];
  private int next = 0;

  private final Semaphore letters = new Semaphore(0);
  private final Semaphore spaces =
                                new Semaphore(BUFFER_CAPACITY);
```

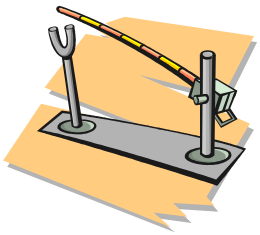
Semaphores

```
public void addLetter(char c) {
    try {
      spaces.acquire();
    } catch ( InterruptedException ignore ) { }
    synchronized ( this ) { buffer[next++] = c;  }
    letters.release();
  }

  public char takeLetter() {
    char charToReturn;
    try {
      letters.acquire();
    } catch ( InterruptedException ignore ) { }
    synchronized ( this ) {
      charToReturn = buffer[--next];
      buffer[next] = ' ';
    }
    spaces.release();
    return charToReturn;
  }
}
```


Latches

- Used to control the progress of threads
- Latch has initial state, and associated value
  - number of events that have to occur before latch opens
- Threads wait on latch
  - await() method
- Latch opens when it reaches its terminal state
  - value becomes 0
  - no more threads to wait for
- Latch cannot be reset from terminal state



CountDownLatch Example

```
public class LatchWorker extends Thread {

  // What to wait for before starting
  private CountDownLatch startGate;
  // What to signal when finished
  private CountDownLatch endGate;

  public LatchWorker( int n, CountDownLatch start,
                             CountDownLatch end ) {
    super("LatchWorker" + n);
    startGate = start;
    endGate = end;
  }
```


CountDownLatch Example

```
  public void run() {
    try {
      // Wait for latch to open
      startGate.await();
      Thread.sleep((int)(Math.random()*1000));  // Do something…
    } catch ( InterruptedException ie ) {
      // Whatever…
    } finally {
      // Signal finished
      endGate.countDown();
    }
  }
}
```

CountDownLatch Example

```
public class LatchExample {
  public static void main(String[] args) {
    int nThreads = Integer.parseInt(args[0]);
    CountDownLatch readyToGo = new CountDownLatch(1);
    CountDownLatch allDone = new CountDownLatch(nThreads);

    // Create and set up the worker threads…
    for ( int i=0; i < nThreads; i++ ) {
      new LatchWorker(i, readyToGo, allDone).start();
    }

    // Delay to allow threads to be created
    try {
      Thread.sleep(1000);
    } catch (InterruptedException ignore ) { }
```

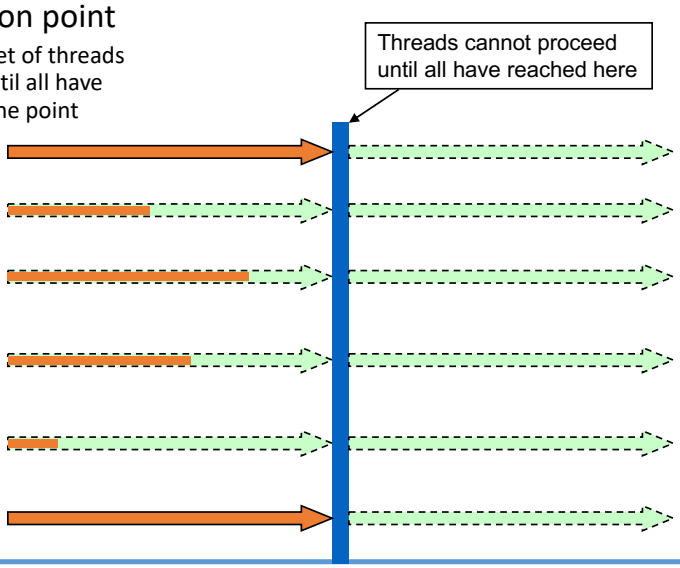

CountDownLatch Example

```
    // Now we assume all threads are ready to go,
    // open the latch…
    readyToGo.countDown();

    // Wait for all the threads to complete
    try {
      allDone.await();
    } catch ( InterruptedException ie ) {
      System.out.println("Interrupted");
    }
    System.out.println("Done!");
  }
}
```

The Cyclic Barrier

- Synchronization point
  - allows a set of threads to wait until all have reached the point



CyclicBarrier Example

```
public class Worker extends Thread {
  private CyclicBarrier theBarrier;
  public Worker ( int i, CyclicBarrier b ) {
    super("Worker"+i); theBarrier = b;
  }
  public void run() {
    try {
      // Wait until all Worker threads are ready
      theBarrier.await();
      // Do some work…
      Thread.sleep((int) (Math.random() * 1000));
      // Wait until all Worker threads are finished
      theBarrier.await();
    } catch ( InterruptedException ignore ) {
    } catch ( BrokenBarrierException bbe ) {
      System.out.println("Barrier broken!");
    }
  }
}
```

Optional timeout on `await()` call has expired before all threads have reached the barrier

CyclicBarrier Example

```
public class CyclicBarrierDriver {

  public static void main(String[] args) {
    int nThreads = Integer.parseInt(args[0]);

    CyclicBarrier barrier = new CyclicBarrier( nThreads,
                    new Runnable() {
                      public void run() {
                       System.out.println("Barrier reached");
                      }
                    } );

    for ( int i = 0; i < nThreads; i++ ) {
      new Worker(i, barrier).start();
    }
  }
}
```

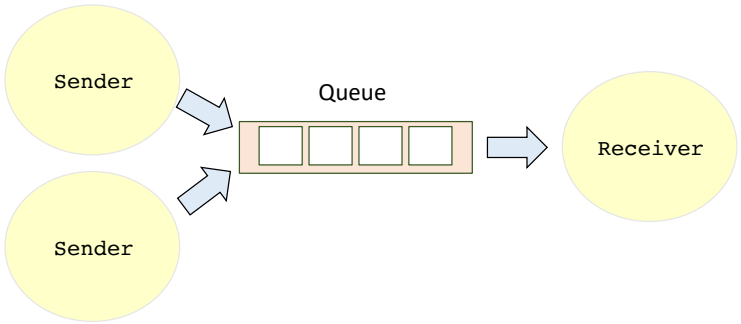
Code executed when the barrier is opened

`CyclicBarrier` Example

```
$ java CyclicBarrierDriver 3
Thread: Worker0 is ready
Thread: Worker1 is ready
Thread: Worker2 is ready
Barrier reached
Thread: Worker2 is working
Thread: Worker0 is working
Thread: Worker1 is working
Thread: Worker1 is finished
Thread: Worker2 is finished
Thread: Worker0 is finished
Barrier reached
```

Inter-thread Communication

- Threads regularly pass information to each other
  - e.g. submission of tasks to Executor
- Queue data structure used for this
  - maintains order of messages
  - may block if queue is empty (or full)



Blocking Queues

- Extends Queue interface with blocking operations
  - take() - remove element, wait until this is possible
  - put() - add element, wait until this is possible
- ArrayBlockingQueue
  - ordered FIFO backed by array, bounded
- LinkedBlockingQueue
  - ordered FIFO, may be bounded
- PriorityBlockingQueue
  - blocking version of PriorityQueue
- SynchronousQueue
  - rendezvous channel, each put() must be matched by a take() and vice versa