Java Threads

iPDC Summer Institute

Introduction

- "Java is the first mainstream programming language to explicitly include threading within the language itself, rather than treating threading as a facility of the underlying operating system."

  https://www.ibm.com/developerworks/java/tutorials/j-threads/j-threads.html

  - This can be good (portability), or very, very bad (speed, flexibility)
  - Java threading tends to be heavy weight, and slower than the native thread interface of the OS
- *Note: Java is not widely used for parallel computing, but Java threads are good for teaching concepts*
- Java threads API is an object oriented version of a common thread API
  - E.g. Similar to POSIX threads, but Object Oriented
- A program starts with one thread
  - The *main* thread
- The main thread then creates child threads to help with computations
- The main then joins with the child threads when the computation is done
- However, the programmer must do the above steps manually
  - Unlike OpenMP, which implements the fork-join model for you

Creating a Java Threads Program

- Typically, you need the following parts:
  - Create the **main class**
    - Implement the **main method**
      - Code that creates the child threads, waits for them to complete, and then prints the answer
  - Implement a **Runnable** class
    - Implement the **run()** method
      - Contains the thread's logic
    - Implement a **constructor**
      - Typically sets the partition information for a thread

Creating a Java Threads Program

- The **structure** and **pseudocode** for a typical Java threads program look like the following:

```
class main
    method main
       // Note: main is executed by the main thread, only

       // given n partitions that you want to compute in parallel:
       create n objects that implements Runnable (in this example MyRunnable - see below),
           passing the partition size to their constructors
       for each MyRunnable object
        create an object of type Thread and pass the MyRunnable object to its constructor
        start the thread object
       end for
       wait for n threads
       print results
     end method main
end class main

class MyRunnable implements Runnable
    method run
      do the computation of my part
    end method run
    method constructor
        set my part
    end method constructor
end class MyRunnable
```

First Example (Hello World, of course)

```
1  import java.lang.*;
2
3  public class Hello {
4      public static void main(String[] args) {
5        int numThreads = 0;
6
7        // Number of threads to be executed should be passed
8        // on the command line
9
10       // If nothing was passed on the command line,
11       // then print error and exit
12       if (args.length < 1) {
13         System.err.println("usage: Hello <numthreads>");
14         System.exit(0);
15       }
16       // Convert the command line string to an integer,
17       //    exit if error
18       try {
19         numThreads = Integer.parseInt(args[0]);
20       } catch (Exception ex) {
21         System.err.println("Invalid argument");
22         System.exit(1);
23       }
24       // Spawn the number of threads passed on
25       // the command line
26       Thread[] threads = new Thread[numThreads];
27       for (int i = 0; i < numThreads; i++) {
28         threads[i] = new Thread(new HelloWorker(i));
29         threads[i].start();
30       }
```

```
31       // Wait for the threads to finish
32       for (int i = 0; i < numThreads; i++) {
33         try {
34           threads[i].join();
35         } catch (Exception ex) {
36           System.err.println("Error waiting: thread " +
37 i);
38         }
39       }
40     }
41 }
42 class HelloWorker implements Runnable {
43   public int id;
44
45   // Constructor to set the id for this thread
46   HelloWorker(int iden) {
47     id = iden;
48   }
49
50   // This method is invoked when the thread starts.
51   // It will print a friendly message.
52   public void run() {
53     synchronized(this) {
54       System.out.println("Hello World from thread "
55         + id);
60     }
61   }
62 }
```

- Compile: `javac Hello.java`
- Run: `java Hello 10`

First Example (Hello World, of course)

```
1  import java.lang.*;
2
3  public class Hello {
4      public static void main(String[] args) {
5        int numThreads = 0;
6
7        // Number of threads to be executed should be passed
8        // on the command line
9
10       // If nothing was passed on the command line,
11       // then print error and exit
12       if (args.length < 1) {
13         System.err.println("usage: Hello <numthreads>");
14         System.exit(0);
15       }
16       // Convert the command line string to an integer,
17       //    exit if error
18       try {
19         numThreads = Integer.parseInt(args[0]);
20       } catch (Exception ex) {
21         System.err.println("Invalid argument");
22         System.exit(1);
23       }
24       // Spawn the number of threads passed on
25       // the command line
26       Thread[] threads = new Thread[numThreads];
27       for (int i = 0; i < numThreads; i++) {
28         threads[i] = new Thread(new HelloWorker(i));
29         threads[i].start();
30       }
```

```
31       // Wait for the threads to finish
32       for (int i = 0; i < numThreads; i++) {
33         try {
34           threads[i].join();
35         } catch (Exception ex) {
36           System.err.println("Error waiting: thread " + i);
37         }
38       }
39     }
40 }
41 class HelloWorker implements Runnable {
42   public int id;
43
44   // Constructor to set the id for this thread
45   HelloWorker(int iden) {
46     id = iden;
47   }
48
49   // This method is invoked when the thread starts.
50   // It will print a friendly message.
51   public void run() {
52     synchronized(this) {
53       System.out.println("Hello World from thread "
54           + id);
55     }
60   }
61 }
62
```

Wait for threads to finish

This is how the thread gets its id

The thread's computation

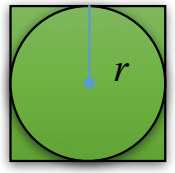
Create thread objects, passing them Runnables

Start the threads

Notes, Caveats, and Advice

- Two ways exists to create threads in Java
  - Extending Thread or implementing Runnable
- Note that a Thread object is not really a thread
  - You actually do not know when the real thread is created - that is according to the Java implementation
- If a thread needs thread specific information, such as what data it is supposed to compute:
  - Then that info should be in the Runnable object
  - It can be passed to the constructor when the Runnable is created
- It's a bad idea to start() threads from within a constructor
- A thread ends when:
  - It come to the end of its run() method
  - It throws an exception that is not caught
  - Another thread calls the deprecated stop() method (do not use)
- Note that the program will not exit until the last thread is finished (except for daemon threads)
  - But do not count on this “feature”, you should always join with your threads if you need to wait for them to finish

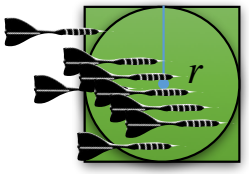
Slightly More Complex Example



- **Problem: Estimate Pi**
- Consider a circle inside of a square
- Let *p* be the ratio of the area of the circle to the area of the square, then

$$\pi = 4p \qquad p = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

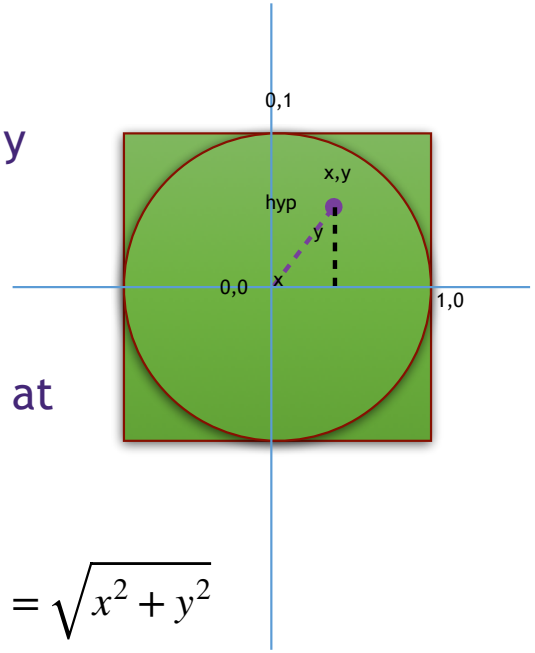
- So:
- How do we figure out *p*? The **Monte Carlo** Method



- Throw darts at the square. Lots and lots of darts.
- Count the number of darts that land inside the circle
- Divide the number of darts that land in the circle by total number thrown to estimate *p* !!!
- Multiply by 4, and we have an estimate for π

Some Unresolved Questions

- How does a computer throw darts? By generating random x,y coordinates for where the dart would land
- Given an (x,y), how can the computer tell if it landed in the circle
  - Make it simple, use the unit circle, and only throw darts at the upper right quadrant
  - Calculate the distance from 0,0
    - Just calculate the hypotenuse of the triangle $hyp = \sqrt{x^2 + y^2}$
  - If *hyp* < 1, then the point falls within the unit circle!
- Use `ThreadLocalRandom` instead of `Random` to get random numbers
  - Its thread safe



Algorithm

```
class Pi
    function main
       get number of threads from the command line
            argument as numThreads
       create four objects of class Monte
          passing numThreads / 4 to each of their constructors
       for each Runnable object
                create an object of class Thread and pass the Runnable
                to its constructor
                start the thread object
       end for
       wait for 4 threads
       sum answer from each of the four Monte objects into result
       print result
     end function main
end class main
```

```
class Monte implements Runnable
    has integer numIterations
    has double answer

    function run
               create random number generator
               set numInside to 0
               loop numIterations times
                 set x to new random number
                 set y to new random number
                 calculate hyp = square root of x^2+y^2
                 if hyp < 1.0
              add 1 to numInside
          end if
       end loop
       set answer to numInside / numIterations
    end function run

    function constructor(iters)
        set numIterations to iters
    end function constructor

end class MyRunnable
```

The Code

```java
import java.lang.*;
import java.lang.Math;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

public class Pi {
    public static void main(String[] iters) {
     int numIter = 0;
     if (iters.length < 1) {
         System.err.println("usage: Pi <iterations>");
         System.exit(0);
     }
     try {
         numIter = Integer.parseInt(iters[0]);
     } catch (Exception ex) {
         System.err.println("Bad argument");
         System.exit(1);
     }
     Runnable[] runnables = new Runnable[4];
     Thread[] threads = new Thread[4];
     for (int i = 0; i < 4; i++) {
         runnables[i] = new Monte(numIter/4);
         threads[i] = new Thread(runnables[i]);
         threads[i].start();
     }

     double answer = 0;
     try {
         for (int i = 0; i < 4; i++) {
          threads[i].join();
          answer += ((Monte) runnables[i]).getRatio();
         }
     } catch (Exception ex) {
         System.err.println("Thread interrupted");
         System.exit(2);
     }

     System.out.println("Ratio is: " + answer);
    }
}
```

```java
class Monte implements Runnable {

    private double ratio;
    private int iters;

    public void run() {
     ratio = findRatio(iters);
    }

    public Monte(int iterations) {
     iters = iterations;
    }

    public double getRatio() {
     return ratio;
    }

    private double findRatio(int iterations) {
     ThreadLocalRandom rand = ThreadLocalRandom.current();
     int numIn = 0;
     int numOut = 0;
     for (int i = 0; i < iterations; i++) {
         // get random number from 0 to 1
         double x = rand.nextDouble();
         double y = rand.nextDouble();
         double hyp = Math.sqrt(x*x + y*y);
         if (hyp < 1.0) {
          numIn++;
         } else {
          numOut++;
         }
     }
     return ((numIn + 0.0) / (numIn+numOut));
     }

}
```

Working with Arrays

- Many parallel computations happen on arrays
- For data decomposition, you must partition the array among the child threads
- Typically, partitioning is done by assigning a number of items per each thread
  - And then having the thread compute its starting and ending index based on its thread id

Array Example

- Simple example to print an array using four threads
- In this example, we will not spawn multiple runnable, but we will spawn on only one
  - The threads will share a single array in the runnable
- Algorithm:

```
1  class PrintArray
2      method main
3         get number of items from the command line
4              argument as numItems
5         create one object of class Printer
6            passing numItems its constructor
7         for i from 1 to 4
8                  create an object of class Thread and pass the
9              Printer object to its constructor
10                 start the thread object
11        end for
12        wait for 4 threads
13     end method main
14 end class main
```

```
15 class Printer
16   has integer array intArray
17   has integer nextId initialized to 0
18
19   method constructor()
20      fill intArray with random integers
21   end method constructor
22
23   method run()
24     // Note that the next two lines must be atomic
25     //   How do we do this in Java?
26     set myId to nextId
27     increment nextId
28
29     set howBig to length of intArray devided by 4
30     set myStart to myId * howBig
31     // Why do the following?
32     if this is the last thread (has myId 3)
33       set myEnd to length of intArray minus 1
34     else
35       set my End to myStart + howBig - 1
36     end if
37
38     loop i from myStart to myEnd
39        print item at the ith index of intArray
40     end loop
41   end method run
   end class Printer
```

The Code

```
1  import java.lang.*;
2  import java.util.Random;
3
4  public class PrintArray {
5      public static void main(String[] args) {
6       int numItems = 0;
7       if (args.length < 1) {
8           System.err.println("usage: Hello <numItems>");
9           System.exit(0);
10      }
11      try {
12          numItems = Integer.parseInt(args[0]);
13      } catch (Exception ex) {
14          System.err.println("Bad argument");
15          System.exit(1);
16      }
17
18      Runnable runnable = new Printer(numItems);
19      Thread[] threads = new Thread[4];
20      for (int i = 0; i < 4; i++) {
21          threads[i] = new Thread(runnable);
22          threads[i].start();
23      }
24      for (int i = 0; i < 4; i++) {
25          try {
26           threads[i].join();
27          } catch (Exception ex) {
28           System.err.println("Error waiting for " + i);
29          }
30      }
31     }
32 }
```

```
33 class Printer implements Runnable {
34     public int[] intArray;
35     private int nextId = 0;
36
37     Printer(int numItems) {
38      intArray = new int[numItems];
39      Random rand = new Random();
40      for (int i = 0; i < numItems; i++) {
41          intArray[i] = rand.nextInt(50);
42      }
43     }
44
45     public void run() {
46      int myId = 0;
47      synchronized(this) {
48          myId = nextId;
49          nextId++;
50      }
51      int howBig = intArray.length / 4;
52      int myStart = myId * howBig;
53      int myEnd = myStart + howBig-1;
54      if (myId == 3) {
55          myEnd = intArray.length-1;
56      }
57
58      for (int i = myStart; i <= myEnd; i++) {
59          synchronized(this) {
60           System.out.println("Thread " + myId +
61                    " index " + i + " in (" + myStart +
62                    ".." + myEnd + "): " + intArray[i]);
63          }
64      }
65     }
}
```

Counting Primes in an Array

- Implement a Java program to count primes
- Your program should accept a count *n* of numbers from the command line
- The program will generate *n* numbers randomly from [1..50) in an array, and then test each number to see if it is prime
- Use a naive algorithm for testing for primeness (for example, loop from 2 to the square root of the number and attempt to divide it by the loop counter)
- The result of the program should be the number of primes found in the array
- Divide the array evenly among four threads

Algorithm

```
class Primes
    method main
       get number of items from the command line
            argument as numItems
       create one object of class Counter
          passing numItems its constructor
       for i from 1 to 4
                create an object of class Thread and pass the
           Counter object to its constructor
                start the thread object
       end for
       wait for 4 threads
       set answer to sum of answers from the threads
       print answer
     end method main
end class main
```

```
class Counter
  has integer array intArray
  has integer nextId initialized to 0
  Has integer array answers

  method constructor()
     fill intArray with random integers
     Fill answers with all 0s
  end method constructor

  method run()
    // Note that the next two lines must be atomic
    set myId to nextId
    increment nextId

    set howBig to length of intArray devided by 4
    set myStart to myId * howBig
    // Why do the following?
    if this is the last thread (has myId 3)
      set myEnd to length of intArray minus 1
    else
      set my End to myStart + howBig - 1
    end if

    loop i from myStart to myEnd
       if isPrime(value of intArray at index i)
         Increment value of answers at index myId by 1
       end if
    end loop
  end method run

  method isPrime(num)
    // you can figure this out
  end method isPrime
end class Printer
```

The Code

```java
import java.lang.*;
import java.lang.Math;
import java.util.Random;

public class Primes {
    public static void main(String[] args) {

     int numItems = 0;
     if (args.length < 1) {
         System.err.println("usage: Hello <numItems>");
         System.exit(0);
     }
     try {
         numItems = Integer.parseInt(args[0]);
     } catch (Exception ex) {
         System.err.println("Bad argument");
         System.exit(1);
     }

     Runnable runnable = new Counter(numItems);
     Thread[] threads = new Thread[4];
     for (int i = 0; i < 4; i++) {
         threads[i] = new Thread(runnable);
         threads[i].start();
     }
     int answer = 0;
     for (int i = 0; i < 4; i++) {
         try {
          threads[i].join();
          answer += ((Counter) runnable).getAnswer(i);
         } catch (Exception ex) {
          System.err.println("Error waiting for " + i);
         }
     }
     System.out.println("The number of primes is: " +
                            answer);
    }
}

class Counter implements Runnable {
    public int[] intArray;
    private int nextId = 0;
    private int[] answers;

    Counter(int numItems) {
      intArray = new int[numItems];
         Random rand = new Random();
         for (int i = 0; i < numItems; i++) {
             intArray[i] = rand.nextInt(50);
         }
      answers = new int[] {0,0,0,0};
    }

    public void run() {
         int myId = 0;
         synchronized(this) {
             myId = nextId;
             nextId++;
         }
         int howBig = intArray.length / 4;
         int myStart = myId * howBig;
         int myEnd = myStart + howBig-1;
         if (myId == 3) {
             myEnd = intArray.length-1;
         }

         for (int i = myStart; i <= myEnd; i++) {
          if (isPrime(intArray[i])) {
           answers[myId]++;
          }
         }
    }

    boolean isPrime(int num) {
      if (num == 2) return true;

      for (int i = 3; i < Math.sqrt(num); i++) {
          if ((num % i) == 0) {
           return false;
          }
      }

      return true;
    }

    Int getAnswer(int idx) {
      return answers[idx];
    }
}
```

Your Challenge

- Write a Java program to find the smallest integer in an array.
- Your program should evenly partition the work among four threads
- Note that the master thread should wait for the other thread to find the smallest integer in their partition,
  - and then choose the smallest from their results
- Your program should generate the initial array by filling it with random numbers
- The size of the array should be given on the command line

Synchronization
Dining Philosophers

- The dining philosophers problem
- Classical CS problem
- A video is worth a thousand words

Algorithm from William Stallings OS Book

- This trick is that this code guarantees that at least one person will get two forks
- You can use Java ReentrantLock and Condition to implement a Monitor
- Use try..finally to lock and unlock

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};   /* availability status of each fork */

void get_forks(int pid)     /* pid is the philosopher id number */
{
   int left = pid;
   int right = (pid++) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);        /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);        /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (pid++) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])     /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])    /*no one is waiting for this fork */
      fork(right) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

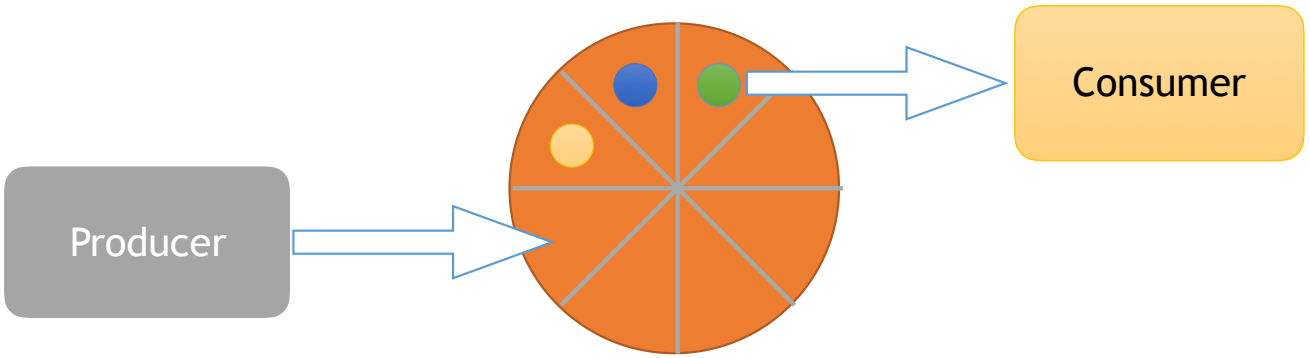
```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true)
   {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```


Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor

```java
import java.lang.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
import java.util.Random;

class DiningPhil {
    public static void main(String[] args) {

        Runnable phils = new Philosophers();
        Thread[] threads;
        threads = new Thread[5];
        for (int i = 0; i < 5; i++) {
            threads[i] = new Thread(phils);
            threads[i].start();
        }
    }
}

class Philosophers implements Runnable {
    private Lock lock;
    private int nextId = 0;
    private boolean[] fork;
    private Condition[] forkReady;

    Philosophers() {
        lock = new ReentrantLock();
        fork = new boolean[5];
        forkReady = new Condition[5];
        for (int i = 0; i < 5; i++) {
            forkReady[i] = lock.newCondition();
            fork[i] = true;
        }
    }

    private void getForks(int pid) {
        lock.lock();

        try {

            int left = pid;
            int right = (pid+1) % 5;

            System.out.println("Grabbing forks (" + pid + ")...");
            while (!fork[left]) {
                System.out.println("Left not available, waiting (" + pid + ")...");
                try {
                    forkReady[left].await();
                } catch (Exception ex) {
                    System.out.println("Await error");
                    ex.printStackTrace();
                    System.exit(1);
                }
            }
            System.out.println("Got left fork (" + pid + ")");
            fork[left] = false;

            while (!fork[right]) {
                System.out.println("Right not available, waiting (" + pid +
                                   ")...");
                try {
                    forkReady[right].await();
                } catch (Exception ex) {
                    System.out.println("Await error");
                    ex.printStackTrace();
                    System.exit(1);
                }
            }
            System.out.println("Got right fork (" + pid + ")");
            fork[right] = false;
        } finally {
            lock.unlock();
        }
    }

    private void releaseForks(int pid) {

        lock.lock();

        try {
            int left = pid;
            int right = (pid+1) % 5;
            System.out.println("Releasing forks (" + pid + ")...");

            fork[left] = true;
            forkReady[left].signal();
            fork[right] = true;
            forkReady[right].signal();
        } finally {
            lock.unlock();
        }

    }

    public void run() {
        int myId = 0;

        lock.lock();
        try {
            myId = nextId;
            nextId++;
        } finally {
            lock.unlock();
        }

        Random rand = new Random();
        try {
            while (true) {
                System.out.println("In deep thought (" + myId + ")...");
                Thread.sleep(rand.nextInt(2000));
                getForks(myId);
                System.out.println("Eating, yum (" + myId + ")...");
                Thread.sleep(rand.nextInt(2000));
                releaseForks(myId);
            }
        } catch (Exception ex) {
            System.out.println("Error in run");
            ex.printStackTrace();
        }
    }
}
```

Task Parallelism
Producer-Consumer

- Another classic: Producer-Consumer
- Common pattern in PDC
- Not data parallel, but task parallel
  - Threads run different methods
- However, must adhere to the data dependency
  - A consumer cannot consume until a slot has an item
  - A producer cannot produce unless the queue has an empty slot



Algorithm from William Stallings OS Book

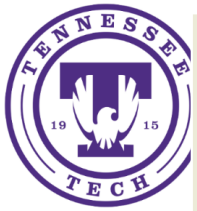
- Again, you can use Java ReentrantLock and Condition to implement a Monitor
- Have the producer generate random numbers, and then sleep to pretend to "produce"
- Have the consumer also sleep to pretend to "consume"

```
void producer()
char x;
{
     while (true)
     {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true)
     {
       take(x);
       consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

```
void append (char x)
{
     while(count == N)
            cwait(notfull);              /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;                                  /* one more item in buffer */
     cnotify(notempty);                     /* notify any waiting consumer */
}

void take (char x)
{
     while(count == 0)
     cwait(notempty);                  /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                  /* one fewer item in buffer */
     cnotify(notfull);                    /* notify any waiting producer */
}
```

Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor



```java
import java.lang.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
import java.util.Random;

public class ProdConDriver {
    public static void main(String[] args) {

        ProdConBuffer pcbuf = new ProdConBuffer();
        Runnable producer = new Producer(pcbuf);
        Runnable consumer = new Consumer(pcbuf);
        Thread prodThread = new Thread(producer);
        Thread conThread = new Thread(consumer);
        prodThread.start();
        conThread.start();
    }

}

class Producer implements Runnable {
    private ProdConBuffer pcbuffer;

    Producer(ProdConBuffer pcbuf) {
      pcbuffer = pcbuf;
    }

    public void run() {
      Random rand = new Random();
      while (true) {
          int x = rand.nextInt(1000);
          try {
            Thread.sleep(x);
          } catch (Exception ex) {
            System.out.println("Error while sleeping");
            System.exit(1);
          }
          System.out.println("+" + x);
          pcbuffer.append(x);
      }
    }
}

class Consumer implements Runnable {
    private ProdConBuffer pcbuffer;

    Consumer(ProdConBuffer pcbuf) {
      pcbuffer = pcbuf;
    }

    public void run() {
      Random rand = new Random();
      while (true) {
          int x = pcbuffer.take();
          System.out.flush();
          int workTime = rand.nextInt(2000);
          try {
            Thread.sleep(workTime);
          }
          catch (Exception ex) {
            System.out.println("Error while sleeping"););
                System.exit(1);
            }
            System.out.println("-" + x);
        }
    }
}

class ProdConBuffer {
    private int[] buffer;
    private int nextIn = 0, nextOut = 0, count = 0;
    private Condition notFull, notEmpty;

    private Lock lock;

    ProdConBuffer() {
        buffer = new int[10];
        lock = new ReentrantLock();
        notFull = lock.newCondition();
        notEmpty = lock.newCondition();
    }

    void append(int x) {
        lock.lock();
        try {
            while (count == 10) {
                try {
                    notFull.await();
                } catch (Exception ex) {
                    System.out.println("append: Error waiting");
                    System.exit(2);
                }
            }
            buffer[nextIn] = x;
            nextIn = (nextIn + 1) % 10;
            count++;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    int take() {
        lock.lock();
        int x;
        try {
            while (count == 0) {
                try {
                    notEmpty.await();
                } catch (Exception ex) {
                    System.out.println("take: Error waiting");
                    System.exit(2);
                }
            }
            x = buffer[nextOut];
            nextOut = (nextOut + 1) % 10;
            count--;
            notFull.signal();
        } finally {
            lock.unlock();
        }
        return x;
    }
}
```

Your Challenge

- Implement parallel Conway's Game of Life
- Neat description and example is here
- Some things you will have to figure out
  - The algorithm happened in steps
    - Threads will have to synchronize after each step
      - See Java CyclicBarrier
  - Threads must use their neighbor's borders to calculate their first and last row
    - At each step
- Take a look at the Java algorithm here for a serial version.

https://rosettacode.org/wiki/Conway's_Game_of_Life#Stretch

OMP4J

- You have practiced parallel programming with OpenMP
  - It makes your life easier (in most cases)
- Java threads requires more code
  - You have to micro-manage thread creation, synchronization (fork-join), etc.
- OMP4J is a tool created to mimic the features of OpenMP, but for Java
- Note: it is not ready for prime-time
  - Poor error reporting
  - Some bugs
  - Some poor implementation (looking at you, parallel FOR)
  - Missing features
- However, it may be good to use as a teaching tool for simple examples
  - And it should continue to mature

Directives

- Beware parallel FOR (it performs poorly)

Directive	Usage	Behavior
`// omp parallel`	Before `{...}` statement	The statement will be invoked in parallel (as many threads as possible).
`// omp parallel for`	Before for-loop	The for-loop will be iterated in parallel.
`// omp sections`	Before `{...}` statement	Wrapper for `// omp section`s directives. It may not contain any other code or directives.
`// omp section`	Before `{...}` statement	The statement will be invoked together with other sections in parallel.
`// omp critical`	Before `{...}` statement	At most one thread will access the statement at any particular time.
`// omp atomic`	Before `{...}` statement	**Deprecated** atomic operation. Internally translated into critical.
`// omp barrier`	Before empty `{}` statement	All threads stop here until the for the last one.
`// omp master`	Before `{...}` statement	Only master thread will execute the statement.
`// omp single`	Before `{...}` statement	Only one thread will execute the statement, no matter which one.

Attributes and Macros

Attribute	Behavior
`threadNum(N)`	The directive will be invoked with `N` threads. Default value is set to number of CPUs.
`schedule(dynamic|static)`	The directive will use dynamic or static executor. Default value is set to dynamic.
`public(a,b)`	Variables `a` and `b` are shared among all threads.
`private(a,b)`	Variables `a` and `b` are created (via parameter-less constructor) for each thread separately.
`firstprivate(a,b)`	Variables `a` and `b` are created (via copy-constructor) for each thread separately.

Macro	Meaning
`OMP4J_THREAD_NUM`	Integer representing current thread ID
`OMP4J_NUM_THREADS`	Integer representing current number of threads

Examples

- Hello world, of course

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3     // omp parallel
4     System.out.println("Hello world from " + OMP4J_THREAD_NUM + "/" + OMP4J_NUM_THREADS);
5     }
6 }
```

Examples

- Estimating Pi with the Monte Carlo Method
- OMP4J does not have reduction!
- This code is slooooooow
  - Because of the parallel FOR

```
1  import java.lang.*;
2  import java.lang.Math;
3  import java.util.Random;
4  import java.util.concurrent.ThreadLocalRandom;
5
6  public class Pi {
7      public static void main(String[] iters) {
8       int numIter = 0;
9       if (iters.length < 1) {
10          System.err.println("usage: Pi <iterations>");
11          System.exit(0);
12      }
13      try {
14          numIter = Integer.parseInt(iters[0]);
15      } catch (Exception ex) {
16          System.err.println(“Bad argument on command line");
17          System.exit(1);
18      }
19
20      int[] numIn = new int[4];
21      ThreadLocalRandom rand = ThreadLocalRandom.current();
22
23      // omp parallel for threadNum(4)
24      for (int i = 0; i < numIter; i++) {
25          // get random number from 0 to 1
26          double x = rand.nextDouble();
27          double y = rand.nextDouble();
28          double hyp = Math.sqrt(x*x + y*y);
29          if (hyp < 1.0) {
30           numIn[OMP4J_THREAD_NUM]++;
31          }
32      }
33
34      double answer = 0;
35      for (int i = 0; i < 4; i++) {
36           answer += numIn[i];
37      }
38
39      System.out.println("Ratio is: " + 4 * (answer / numIter));
40
41  }
42 }
```