Threading Part 2

In this lecture we discuss interthread communication and give some additional examples using threads.

wait(), notify(), and notifyAll()

The join() method we saw previously waited for a thread to finish before continuing. Sometimes you want a thread to wait for another thread to reach some point of its computation, then for the thread to continue. In other words, you might not want the thread to finish and be closed out, but instead wait/coordinate on some resource or marker point. We can do that with some additional methods, wait(), notify(), and notifyAll().

All three methods must be used inside a synchronized block.

wait

The wait() method makes the thread wait indefinitely for another thread to call notify() or notifyAll() to wake the thread up. There is an overloaded method that allows you to set a timeout and after the timeout has elapsed the thread will stop waiting.

notify

The notify() method wakes up only one thread waiting on the object. That thread can then resume whatever code comes after its call to wait(). If there is more than one thread then waiting then it is unpredictable which one will get woken up (well, you might be able to tell depending on the OS implementation, it seems to generally be the first thread that was waiting).

notifyAll

The notifyAll() method wakes up all threads waiting on the object. Note that since the wait() must be in a synchronized block, all threads won't continue at once. They will complete the synchronized block in some order of being woken up, where that order is dependent on the OS implementation.

Here is a simple example. Consider a pizza restaurant and customers, where each customer runs in a thread and the restaurant is a separate thread. This is unlikely in a real example, but it would make sense if you were running some kind of pizza restaurant simulation program, and it would be a similar model if you had mobile apps ordering online, where each mobile device corresponds to a customer thread.

First, you might want customers to wait until the restaurant is open. Here is a poorly implemented version that doesn't use wait:

```
// Customer.java
public class Customer extends Thread {
    private Restaurant restaurant = null;
    private String name;

    public Customer()
    {
        name = "Unknown";
    }
```

```
    public Customer(Restaurant r, String name)
    {
        restaurant = r;
        this.name = name;
    }

    public void run()
    {
        while (restaurant.open == false)
        {
            // Wait
            System.out.println(name + " is waiting");
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                System.out.println(e);
            }
        }
        System.out.println(name + " enters restaurant");
    }

}

// Restaurant.java
import java.util.Scanner;
public class Restaurant {
   public boolean open;

   public Restaurant()
   {
       open = false;
   }

   public static void main(String[] args)
   {
       Restaurant mockcheezmo = new Restaurant();
       Customer finn = new Customer(mockcheezmo, "Finn");
       Customer rey = new Customer(mockcheezmo, "Rey");

       finn.start();
       rey.start();

       // Open the restaurant when we type something in
       Scanner kbd = new Scanner(System.in);
       String s = kbd.next();
       mockcheezmo.open = true;

       // Wait for customer threads to finish then exit the program
       try
       {
        finn.join();
        rey.join();
       }
       catch (InterruptedException e)
       {
```

```
            System.out.println(e);
        }
        System.out.println("Closing restaurant");
    }
}
```

The main method creates two customers who then use polling to wait until the restaurant opens (controlled in main when a string is entered).

The poor part of this program is the polling that the customer has to check every second to see if the restaurant is open. This is inefficient and doesn't allow the customer to do anything while waiting. Our solution is to have the customer wait() if the restaurant is not open:

In Customer:

```
    public void run()

     {
         if (restaurant.open == false)
         {
             synchronized(restaurant)
             {
                 try {
                     restaurant.wait();
                 } catch (InterruptedException ex)
                 {
                     System.out.println(ex);
                     System.exit(0);
                 }
             }
         }
         System.out.println(name + " enters restaurant");
     }
```

In main:

```
        Scanner kbd = new Scanner(System.in);
        String s = kbd.next();
        mockcheezmo.open = true;
        synchronized (mockcheezmo)
        {
             mockcheezmo.notifyAll();
        }
```

Next let's have the restaurant produce pizzas. In our simple case the restaurant produces only one type of pizza. The number of pizzas available for consumption is held in the variable "numPizzas". We increment this in main with some pizza logic. Each time we produce a pizza we notify anyone that might be waiting for one. If nobody is waiting then nothing happens (it doesn't get matched with a future wait).

```
public class Customer extends Thread {
    private Restaurant restaurant = null;
    private String name;
```

```
    private Random rnd = new Random();
    public Customer()
    {
        name = "Unknown";
    }
    public Customer(Restaurant r, String name)
    {
        restaurant = r;
        this.name = name;
    }

    public void run()
    {
        if (restaurant.open == false)
        {
            synchronized(restaurant)
            {
                try {
                    restaurant.wait();
                } catch (InterruptedException ex)
                {
                    System.out.println(ex);
                    System.exit(0);
                }
            }
        }
        System.out.println(name + " enters restaurant");

        int total = 0;
        // Get a pizza every 4-10 seconds until we get 5 pizzas
        while (total < 5)
        {
            int ms = (rnd.nextInt(7) + 4) * 1000;
            try
            {
                Thread.sleep(ms);
            }
            catch (InterruptedException e) {}
            synchronized(restaurant)
            {
                if (restaurant.numPizzas == 0)
                    try {
                        restaurant.wait();
                    }
                    catch (InterruptedException e)
                    {
                        System.out.println(e);
                    }
                restaurant.numPizzas--;
                total++;
                System.out.println(name + " got a pizza, inventory = " +
                        restaurant.numPizzas + " " + name + " has " + total);
            }
        }
        System.out.println(name + " is done");
    }
}
```

```
public class Restaurant {
   public boolean open;
   public int numPizzas;

   public Restaurant()
   {
       open = false;
       numPizzas = 0;
   }

   public static void main(String[] args)
   {
       Restaurant mockcheezmo = new Restaurant();
       Customer finn = new Customer(mockcheezmo, "Finn");
       Customer rey = new Customer(mockcheezmo, "Rey");

       finn.start();
       rey.start();

       // Open the restaurant when we type something in
       Scanner kbd = new Scanner(System.in);
       String s = kbd.next();
       mockcheezmo.open = true;
       synchronized (mockcheezmo)
       {
            mockcheezmo.notifyAll();
       }

       // Produce pizzas
       do
       {
           s = kbd.next();
           if (s.equals("p"))
           {
               System.out.println("Produced a pizza");
               synchronized(mockcheezmo)
               {
                   mockcheezmo.numPizzas++;
                   mockcheezmo.notify();
               }
           }
       } while (!s.equals("q"));

       // Wait for customer threads to finish then exit the program
       try
       {
        finn.join();
        rey.join();
       }
       catch (InterruptedException e)
       {
           System.out.println(e);
       }
       System.out.println("Closing restaurant");
   }
}
```

If you run this version, as long as pizzas are available, the customer will grab a pizza and then wait a random time to try and get the next pizza. If a pizza is not available then the customer will wait. If both customers are waiting one of them will be woken up to get a pizza (not necessarily first come first serve!)

Question: What happens if you enter "q" in the main thread before all the customers have gotten their five pizzas?

Parallelization and Threading for Performance

If you have a multiprocessor system then threads can truly run in parallel and you have the potential for performance gains. Typical systems today have 2-4 cores. With hyperthreading, which are like virtual cores, you get the appearance of even more processors available. You can run work in threads and the OS will farm the threads to cores that may run in parallel. However, it can get complicated to split the work up to the threads and to post-process the computation made by each thread.

Here is an example that uses threads to sort an array of numbers. In this case we will use the mergesort algorithm. Exact details how mergesort works are not important for purposes of this threading discussion, but essentially it splits the arrays into smaller arrays until we get to an array of one element, which is sorted, then it merges the sorted arrays back together.

In this example we explicitly create four threads. Each thread is responsible for sorting approximately 1/4 of the array. If the thread runs in parallel we could have a speedup of 4.

However, when the threads are done the whole array isn't sorted. Instead we have four chunks of the array that are sorted. To get the final sorted array, we have to merge subarrays 1 and 2, merge subarrays 3 and 4, which results in two sorted subarrays, then finally merge the two sorted subarrays to get one sorted total array.

Here is code for a small test array:

```
public class Sorter extends Thread {
    private int[] arr, tmp;
    private int startIndex, endIndex;

    public Sorter()
    {
        arr = null;
        tmp = null;
        startIndex = 0;
        endIndex = 0;
    }

    public Sorter(int[] arr, int start, int end)
    {
        this.arr = arr;
        tmp = new int[arr.length];
        this.startIndex  = start;
        this.endIndex = end;
    }
```

```
    private void mergeSort(int left, int right)
    {
        if( left < right )
      {
            int center = (left + right) / 2;
            mergeSort(left, center);
            mergeSort(center + 1, right);
            merge(left, center + 1, right);
      }
    }

    private void merge(int left, int right, int rightEnd)
    {
        int leftEnd = right - 1;
        int k = left;
        int num = rightEnd - left + 1;

        while(left <= leftEnd && right <= rightEnd)
            if(arr[left] < (arr[right]))
                tmp[k++] = arr[left++];
            else
                tmp[k++] = arr[right++];

        while(left <= leftEnd)    // Copy rest of first half
            tmp[k++] = arr[left++];

        while(right <= rightEnd)  // Copy rest of right half
            tmp[k++] = arr[right++];

        // Copy tmp back
        for(int i = 0; i < num; i++, rightEnd--)
            arr[rightEnd] = tmp[rightEnd];
    }

    public void run()
    {
        mergeSort(startIndex, endIndex);
    }
}
```

```
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

public class SorterMain {
    private static int[] arr, tmp;
    private static Random rnd = new Random();

    // This is here so we can compare with sorting in one thread
    private static void mergeSort(int left, int right)
    {
        if( left < right )
      {
            int center = (left + right) / 2;
            mergeSort(left, center);
```

```
        mergeSort(center + 1, right);
        merge(left, center + 1, right);
  }
}

private static void merge(int left, int right, int rightEnd)
{
    int leftEnd = right - 1;
    int k = left;
    int num = rightEnd - left + 1;

    while(left <= leftEnd && right <= rightEnd)
        if(arr[left] < (arr[right]))
            tmp[k++] = arr[left++];
        else
            tmp[k++] = arr[right++];

    while(left <= leftEnd)    // Copy rest of first half
        tmp[k++] = arr[left++];

    while(right <= rightEnd)  // Copy rest of right half
        tmp[k++] = arr[right++];

    // Copy tmp back
    for(int i = 0; i < num; i++, rightEnd--)
        arr[rightEnd] = tmp[rightEnd];
}

public static void main(String[] args) {
    arr = new int[16];
    tmp = new int[16];
    for (int i = 0; i < arr.length; i++)
    {
        arr[i] = rnd.nextInt(1000);
    }

    Sorter s1 = new Sorter(arr,0,3);
    Sorter s2 = new Sorter(arr,4,7);
    Sorter s3 = new Sorter(arr,8,11);
    Sorter s4 = new Sorter(arr,12,15);
    s1.start(); s2.start(); s3.start(); s4.start();

    try {
        s1.join();
        s2.join();
        s3.join();
        s4.join();
    } catch (InterruptedException ex) {
        System.out.println(ex);
    }

    // Merge subarrays together
    merge(0,4,7);
    merge(8,12,15);
    merge(0,8,15);

    for (int i = 0; i < arr.length; i++)
```

```
            System.out.println(arr[i]);
    }
}
```

To do: Create an extremely large array and see if there is any performance gain using threads over running in a single thread. Netbeans has a profiler that can tell us how long it takes a method to run.

If we were to create more threads then we start having overhead issues where it takes more time to manage the threads and for the threads to communicate. At some point the overhead may not be worth the performance benefit. In GPU programming there is very little overhead to create a thread and we can make thousands or millions of them that run in parallel on the video card.