COMP329 Robotics and Autonomous Systems

Lecture 7: Threads & Multitasking in Robots

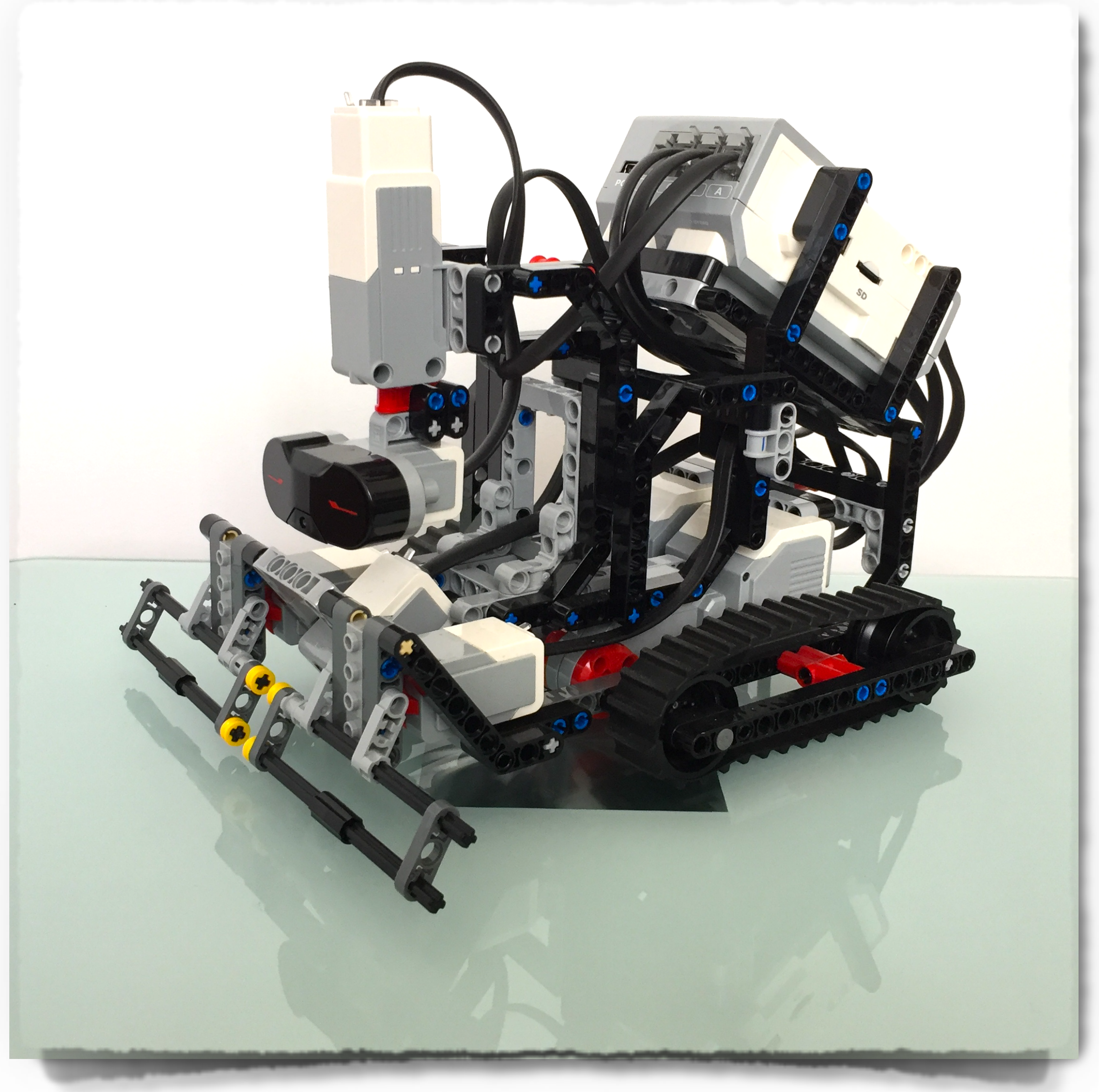
Dr Terry R. Payne

Department of Computer Science

UNIVERSITY OF LIVERPOOL

Threads & MultiTasking

- Some more programming techniques that will be helpful for the assignment.
  - The main subject will be multitasking
    - How to get the robot to do several things at once.
  - That involves using ***threads***.
    - But we'll cover some other useful programming ideas as well.
- In robotics we frequently need to deal with ***concurrency***
  - Different bits of code running more or less independently in time.
  - Once upon a time these had to be separate processes.
    - Rather heavyweight.
  - A more modern approach is that of ***threads***
    - These provide fine-grained concurrency within a process.
  - Here we discuss the basic ideas behind the use of threads in Java.



Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

What are threads

- A thread is a flow of control within a program.
  - Similar to multiple processes, but all belonging to the same program
  - Can easily share state, and coordinate behaviour

- Threads are like lectures at a university
  - Separate, independent entities that can run concurrently
  - Resources can be shared,
    - Only one entity uses a resource at the same time
    - Need coordination to manage access to resources.

- Threads also have local data that is distinct from shared resources

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Thread Object

- All execution in Java is associated with a Thread object.
  - That is what `main()` launches.
  - New threads are born when a new instance of: `java.lang.Thread` is created
- This object is what we manipulate to control and coordinate execution of the thread.

- Two ways to handle threads.
  - One way is to ***sub-class the Thread object***.
    - Create your own thread which extends the standard thread.
    - Do this by defining/over-riding the run() method.
      - (This is what is invoked when the thread starts.)
  - Second way is to ***create a Runnable object*** and execute it in an unmodified Thread
    - Many Java programmers consider that using Runnable is better style.
  - We will stick to the first.

Thread Scheduling

- In most Java implementations, threads are time-sliced.
  - Each thread runs for a while in some order
  - On other Java implementations, you might get different behavior.
  - All depends on what the VM does.
- All threads have a priority value.
  - Any time a higher priority thread becomes runnable, it preempts any lower priority threads and starts executing.
  - By default, threads with the same priority are scheduled round-robin.
- This means that once a thread begins to run it continues until:
  - It sleeps due to a sleep() or wait();
  - It waits for the lock for a synchronized method;
  - It blocks on I/O;
  - It explicitly yields control using yield(); or
  - It terminates.
- So there is no necessity for threads to be time-sliced.

Java Thread Creation

- When a Java program starts, a single thread is created
  - JVM also has own threads for garbage collection, screen updates, event handling etc.
  - New threads may be created by extending the `Thread` class
  - Again, threads may be managed directly by kernel, or implemented at user level by a library

```
class Worker1 extends Thread {
      public void run() {
            System.out.println("A Worker Thread");
      }
}
public class First {
      public static void main(String args[]) {
            Worker1 runner = new Worker1();
            runner.start();
            System.out.println("The Main Thread");
      }
}
```

- Class `Worker1` is derived from `Thread` class
  - The work of the new thread is specified in the `run()` method
  - In `main()` we create a new `Worker1` object
  - Calling the `start()` method…
    - allocates memory and initialises the new thread – causes `run()` method to be called
  - Original thread and new thread now run in parallel

Controlling Threads

- There are a few methods that allow us to control the execution of threads.
  - Some are depreciated, others we'll not look at

- We will focus on the following:
  - `start()` is used to start a thread running.
    - We will see an example in a bit.
  - `sleep()` is to pause for a short period
  - Synchronisation on shared resources
    - Coordinated using `wait()`, and `notify()`

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

sleep()

- Sometimes we need to tell a thread to take a break.
  - The method `sleep()` will do this.
    - It takes an argument that is the number of milliseconds to sleep for.
  - `sleep()` is a class method of Thread, so it can be called either using:
    - `Thread.sleep()`
    - or by calling it on a specific instance of Thread:
      - `myOwnLittleThread.sleep()`
  - Puts the current thread to sleep

sleep()

- Good practice to put a sleep in a **`try`**/ **`catch`** structure in case the thread is interrupted during its sleep.
- You often set threads to sleep precisely because you are waiting for them to be interrupted.
- A sleeping thread can be woken up by an **`InterruptedException`** so we need to specify what to do if this happens.

```
public void run(){
    while(true){
        System.out.println("One!");
        try{
            Thread.sleep(1000);
        } catch(InterruptedException e){
            // Guess we won't sleep after all
        }
    }
}
```


Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Mutual Exclusion

- Indeterminacy arises because of possible simultaneous access to a shared resource
  - The variable ‘count’ in the example opposite
- Solution is ***to allow only one thread to access*** ‘count’ at any one time; all others must be excluded
- To control access to such a shared resource we declare the section of code in which the thread/process accesses the resource to be the critical region/section
- We can then regulate access to the critical region
  - When one thread is executing in its critical region, no other thread/process is allowed to execute in its critical region
  - This is known as ***mutual exclusion***

Example

Suppose we have an object (called ‘`thing`’) which has the following method:

```
public void inc(){
    count = count + 1;
}
```

The integer `count` is private to ‘thing’, and is initially zero.

Two threads, T1 and T2, both execute the following code:

```
thing.inc();
```


Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Synchronisation

- Indeterminacy arises because of possible simultaneous access to a shared resource
  - The variable ‘count’ in the example
- Solution is to ***allow only one thread*** to access `count` at any one time
  - all others must be excluded
- To control access to such a shared resource we declare the section of code in which the thread/process accesses the resource to be the ***critical region/section***
- We can then regulate access to the critical region
  - When one thread is executing in its critical region, no other thread/process is allowed to execute in its critical region
  - This is known as ***mutual exclusion***
- A key part of synchronisation is ensuring that no job is left waiting indefinitely

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Synchronisation

- In Java, mutual exclusion is achieved by ensuring synchronisation when calling methods that access shared resources
  - Declare the method as `synchronized`
- Only one thread at a time is allowed to execute any `synchronized` method of an object.
  - i.e. when called, the object becomes locked
  - Other threads are blocked until they can acquire the lock on the object
- Note that locks are reentrant, so a thread does not block itself.
  - The `synchronized` function can call itself recursively, and it can call other `synchronized` methods of the same object.

```
public synchronized void myFunction() {
    ...
}
```


Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

wait() and notify()

- wait() and notify() provide more direct synchronization of threads.
  - When a thread executes a synchronized method that contains a wait(), it gives up its hold on the block and goes to sleep.
  - The idea is that the thread is waiting for some necessary event to take place.
  - Later on, when it wakes up, it will start to try to get the lock for the synchronized object.
    - When it gets the lock, it will continue from where it left off.
- What wakes the thread up from waiting is a call to notify() on the same synchronized object.

```
class Buffer {
      private int v;
      private volatile boolean empty=true;
      public synchronized void insert(int x) {
            while (!empty) {
                  try {
                        wait();
                  } catch (InterruptedException e) {}
            }
            empty = false;
            v = x;
            notify();
      }
}
```

- The wait() call - releases the lock and moves the calling thread to the 'wait set'
- The notify() call - moves an arbitrary thread from the wait set back to the entry set
- Can use notifyAll() to move all waiting threads back to entry set
- We use volatile to guarantee that a shared variable is updated

SimpleRobot

- Let's define a Java class to represent our robot.
  - Two touch sensors
  - One infrared sensor
  - One colour sensor
  - Two drive motors
  - (I'm ignoring the motor pointing the infrared sensor).
- Create an instance as part of a control program.
  - Provide a data element for each element of the robot

```
public class SimpleRobot {

   private EV3TouchSensor leftBump, rightBump;
   private EV3IRSensor irSensor;
   private EV3ColorSensor cSensor;
   private SampleProvider leftSP, rightSP, distSP, colourSP;
   private float[] leftSample, rightSample, distSample, colourSample;
   private EV3LargeRegulatedMotor motorL, motorR;
   private EV3MediumRegulatedMotor motorS;

   …
```

SimpleRobot Accessor Methods

- Each of these private members would need appropriate "get" and/or "set" functions.
  - Get sensor values.
  - Set motor values.
  - Get motor values, for example `isLeftMotorOn()`

```
public boolean isLeftBumpPressed() {}

public boolean isRightBumpPressed() {}

public float getDistance() {}

public float[] getColour() {}

public void startMotors(){}

public void reverseMotors(){}

public void turnMotors(boolean clockwise){}

public void stopMotors(){}

public boolean isRightMotorOn() {}

public boolean isLeftMotorOn() {}
```

SimpleRobot Constructor

- Constructor sets up the data members to talk to the relevant bits of the hardware.
- Good Java practice/style to set up the robot like this.
  - Independent of using threads.
- Also include a `closeRobot()` method to ensure ports are closed

```
public SimpleRobot() {
      Brick myEV3 = BrickFinder.getDefault();

      leftBump = new EV3TouchSensor(myEV3.getPort("S2"));
      rightBump = new EV3TouchSensor(myEV3.getPort("S1"));
      irSensor = new EV3IRSensor(myEV3.getPort("S3"));
      cSensor = new EV3ColorSensor(myEV3.getPort("S4"));

      leftSP = leftBump.getTouchMode();
      rightSP = rightBump.getTouchMode();
      distSP = irSensor.getDistanceMode();
      colourSP = cSensor.getRGBMode();

      leftSample = new float[leftSP.sampleSize()];
      rightSample = new float[rightSP.sampleSize()];
      distSample = new float[distSP.sampleSize()];
      colourSample = new float[colourSP.sampleSize()];

      motorL = new EV3LargeRegulatedMotor(myEV3.getPort("B"));
      motorR = new EV3LargeRegulatedMotor(myEV3.getPort("C"));
      motorS = new EV3MediumRegulatedMotor(myEV3.getPort("A"));
   }

   public void closeRobot() {
      leftBump.close();
      rightBump.close();
      irSensor.close();
      cSensor.close();
   }
```

Robot Monitor Thread

- Now we'll use a thread to set up a robot monitor.
  - Thread that observes what the robot is doing
  - Uses the `SimpleRobot` object to do this.

- Reports the robot state on the screen.
  - Useful debug tool.

```
public class RobotMonitor extends Thread {
   private int delay;
   public SimpleRobot robot;

    GraphicsLCD lcd = LocalEV3.get().getGraphicsLCD();

    // Make the monitor a daemon and set
    // the robot it monitors and the delay
    public RobotMonitor(SimpleRobot r, int d){
      this.setDaemon(true);
      delay = d;
      robot = r;
    }
```

Daemons

- Daemons are threads providing “services” for other threads in the program.
  - They run as background processes
  - They serve basic functionalities upon which other threads build
- If a thread is declared Daemon, its existence does not prevent the JVM from exiting (unlike other threads).
  - Useful methods in `java.lang.Thread`:
  - `boolean isDaemon()`
    - Flags whether thread is daemon
  - `void setDaemon (Boolean on)`
    - Sets the thread to be a daemon. Can only be used before the thread is created.

Robot Monitor

- We can now report on the status of the robot
  - Note that the infrared sensor returns one value (distance)
  - The colour sensor returns three values (RGB)
    - We've used a `DecimalFormat` object to round the values to three significant digits

```
public void run(){
  // The decimalformat here is used to round the number to three significant digits
  DecimalFormat df = new DecimalFormat("####0.000");

  while(true){
     lcd.clear();
     lcd.setFont(Font.getDefaultFont());
     lcd.drawString("Robot Monitor", lcd.getWidth()/2, 0, GraphicsLCD.HCENTER);
     lcd.setFont(Font.getSmallFont());

     lcd.drawString("LBump: "+robot.isLeftBumpPressed(), 0, 20, 0);
     lcd.drawString("RBump: "+robot.isRightBumpPressed(), 0, 30, 0);
     lcd.drawString("Dist: "+robot.getDistance(), 0, 40, 0);
     lcd.drawString("Colour: ["+
           df.format(robot.getColour()[0]) +" "+
           df.format(robot.getColour()[1]) +" "+
           df.format(robot.getColour()[2]) +"]", 0, 50, 0);
     lcd.drawString("Lmotor: "+robot.isLeftMotorOn(), 0, 60, 0);
     lcd.drawString("Rmotor: "+robot.isRightMotorOn(), 0, 70, 0);
     try{
        sleep(delay);
     }
     catch(Exception e){}
   }
}
```

Run Monitor

- Finally we connect the monitor and an instance of Simple Robot

- Clearly, we could use the same style to build more complex robot controllers.
  - Threads controlling different aspects of the robot:
    - Moving around
    - Avoiding obstacles
    - Preventing collisions
  - All talking to the SimpleRobot object to operate the hardware.
  - All together determining what the robot does.

```
public class RunMonitor {

   public static void main(String[] args) throws Exception{
      SimpleRobot me = new SimpleRobot();

      RobotMonitor myMonitor = new RobotMonitor(me, 400);

      myMonitor.start();

      // Do stuff...
      me.closeRobot();
   }
}
```

Listeners, Events and Behaviours

- In the NXT API Listeners allowed us to monitor sensors and keys.
  - No longer needed to keep a busy watch on the hardware
    - Instead, have the hardware tell us when some thing changes.
  - Exactly the same kind of event-driven programming that we have in GUIs.
    - Pressing a button typically leads to an action.

- In EV3, the listener model has been depreciated
  - Problematic with different types of sensor
  - Some listeners still exist, e.g. for `MoveListener` or `NavigationListener`
- Behaviours now allow us to “listen” for specific events using the `takeControl()` method
  - Thus events determine which Behaviour fires in our robot

Summary

- This lecture looked at multi-tasking, which is handy for many robotics tasks.
  - First we looked at threads, which provide a lightweight approach to multi-tasking.
  - Then we looked at how threads can be used in LeJOS.
- Our example also showed how to use LeJOS in a more object-oriented way.
- In the next lecture, we will look at maps and mapping, and in particular:
  - Occupancy Grids!