COMP31212: Concurrency
A Review of Java Concurrency

Giles Reger
What are Java Threads?

- In Java, concurrency is achieved by Threads
- A Java Thread object is just an object on the heap, like any other object
- So when we say ‘Java Thread’ we mean something else
- Let’s consider the Big Picture
The Big Picture
The Big Picture

Machine Level
The Big Picture

Machine Level
The Big Picture

Machine Level
The Big Picture
The Big Picture

Machine Level
The Big Picture
The Big Picture

Memory Hierarchy

Machine Level

Hardware Threads
The Big Picture
The Big Picture

Memory Hierarchy

OS Level

Machine Level

Network

Hardware Threads
The Big Picture

OS Level

Machine Level

Process

Memory Hierarchy

Network

Hardware Threads
The Big Picture

OS Level

Memory Hierarchy

Network

Hardware Threads
The Big Picture

OS Level

- Software Threads
- Process
- Software Threads
- Process
- Software Threads
- Process

Machine Level

- Memory Hierarchy
- Network

- Hardware Threads
The Big Picture

OS Level

Machine Level

Software Threads

Process

Scheduling

Memory Hierarchy

Network

Hardware Threads
The Big Picture

OS Level

Memory Hierarchy

Machine Level

- Software Threads
  - Process
  - Scheduling
- Software Threads
  - Process
  - Resource Management
- Software Threads
  - Process

Network

Hardware Threads
The Big Picture

Java Level

OS Level

Machine Level

JVM Threads

Scheduling  Resource Management

Software Threads  Software Threads  Software Threads

Process  Process  Process

Memory Hierarchy

Network

Hardware Threads
The Big Picture

Java Level

OS Level

Machine Level

Memory Hierarchy

Network

Hardware Threads

Process

Scheduling

Resource Management
The Big Picture

Java Level

OS Level

Machine Level

Memory Hierarchy

Network

Hardware Threads

Scheduling

Resource Management

JVM Threads

User Threads

Stacks

Main
The Big Picture

Java Level

OS Level

Machine Level
The Big Picture

Java Level

OS Level

Machine Level

Memory Hierarchy

Network

Hardware Threads

Scheduling

Resource Management

Process

Process

Process

Software Threads

Software Threads

Software Threads

JVM Threads

User Threads

Stacks

Heap

Main
Making Threads

- There are two ways to create a Thread in Java
  - Sub-classing `java.lang.Thread`
  - Implementing Runnable
- Implement the `run` method
Running Threads

- Call start
- When you call start the JVM creates a new OS thread with the run method and then returns
- run is just a method, calling it will run in the current thread
- Creating OS threads is expensive
class MyThread extends Thread {
    int i;

    public void run() {
        i = 0;
        while (i <= 5) {
            System.out.println("" + i);
            i++;
        }
    }
}

... MyThread mt = new MyThread();
mt.start();
...
public interface Runnable{
    public abstract void run();
}

class MyRun implements Runnable {
    int i;

    public void run() {
        i = 0;
        while (i <= 5) {
            System.out.println("" + i);
            i++;
        }
    }
}

MyRun mr = new MyRun();
Thread mt = new Thread(mr);
mt.start();
...
The Dirty Way...

... using an anonymous inner class

new Thread(){
    public void run(){
        int i = 0;
        while (i <= 5) {
            System.out.println("" + i);
            i++;
        }
    }
}.start();
We’re Done

- We have concurrency - two bits of code running at the same time
We’re Done

• We have concurrency - two bits of code running at the same time
• So we can go home...
We’re Done

- We have concurrency - two bits of code running at the same time
- So we can go home...
- Not quite!
We’re Done

- We have concurrency - two bits of code running at the same time
- So we can go home...
- Not quite!
- The reasons we want concurrency might be
  - Neat organisation of code
  - Responsiveness
  - Efficiency
- But at the end of the day it needs to work
We’re Done

- We have concurrency - two bits of code running at the same time
- So we can go home...
- Not quite!
- The reasons we want concurrency might be
  - Neat organisation of code
  - Responsiveness
  - Efficiency
- But at the end of the day it needs to work
- We have shared variables being accessed by different threads
- Potential problems - data races, deadlock, livelock, fairness...
public class GoingToTheRaces {
    public static void main(String[] args) throws Exception {
        Go[] gos = new Go[10];
        for(int i=0;i<10;i++) gos[i] = new Go();
        for(int i=0;i<10;i++) gos[i].start();
        for(int i=0;i<10;i++) gos[i].join();
        System.out.println(counter);
    }
    private static int counter = 0;
    private static class Go extends Thread {
        public void run()
        {
            for(int i=0;i<10000;i++)
                counter+=1;
        }
    }
}
public class GoingToTheRaces {
    public static void main(String[] args) throws Exception {
        Go[] gos = new Go[10];
        for(int i=0;i<10;i++) gos[i] = new Go();
        for(int i=0;i<10;i++) gos[i].start();
        for(int i=0;i<10;i++) gos[i].join();
        System.out.println(counter);
    }

    private static int counter = 0;
    private static class Go extends Thread {
        public void run(){
            for(int i=0;i<10000;i++){
                int c = counter;
                counter = c+1;
            }
        }
    }
}
public class GoingToTheRaces {
    public static void main(String[] args) throws Exception {
        Go[] gos = new Go[10];
        for(int i=0;i<10;i++) gos[i] = new Go();
        for(int i=0;i<10;i++) gos[i].start();
        for(int i=0;i<10;i++) gos[i].join();
        System.out.println(counter);
    }
    private static int counter = 0;
    private static class Go extends Thread {
        public void run(){
            for(int i=0;i<10000;i++){
                int c = counter;  \ Interrupt here!!
                counter = c+1;
            }
        }
    }
}
Where it goes wrong

Compiled from "GoingToTheRaces.java"

```java
class GoingToTheRaces$Go extends java.lang.Thread{
    public void run();
    
    Code:
    0:  iconst_0
    1:  istore_1
    2:  iload_1
    3:  sipush 10000
    6:  if_icmpge 20
    9:  iconst_1
    10: invokestatic #3; //Method GoingToTheRaces.access$112:(I)I
    13:  pop
    14:  iinc 1, 1
    17:  goto 2
    20:  return

    GoingToTheRaces$Go(GoingToTheRaces$1); 
    
    Code:
    0:  aload_0
    1:  invokespecial #1; //Method "<init>"::<>
    4:  return
}

public class GoingToTheRaces extends java.lang.Object{

            .................

    static int access$112(int);
    
    Code:
    0:  getstatic #1; //Field counter:I
    3:  iload_0
    4:  iadd
    5:  dup
    6:  putstatic #1; //Field counter:I
    9:  ireturn
```
Where it goes wrong
Mutual Exclusion in Java

- Each object has a single lock associated with it
  - called a Monitor in the JLS
  - it’s some bits in the object header (mark word)

- Any synchronized code applied to an object must first obtain the lock for the object

- If the lock is already taken, then thread will block until lock is available

- Non-synchronized code still has free access

- Exceptions release locks

- Locks are re-entrant
The synchronized Keyword

The synchronized keyword can be used in two ways

- As a method modifier
  
  ```java
  synchronized type name(arguments) { statements }
  ```

- On an object directly
  
  ```java
  synchronized (object) { statements }
  ```

This provides *mutually exclusive* access to objects...
The synchronized Keyword

The synchronized keyword can be used in two ways

- As a method modifier
  
  \[
  \text{synchronized type name(arguments) \{} \text{ statements } \}\]

- On an object directly
  
  \[
  \text{synchronized (object) \{} \text{ statements} \}
  \]

This provides *mutually exclusive* access to objects... if all accesses to that object are guarded in this way.
More Mutual Exclusion

- A synchronized modifier on a method for Class C is the same as replacing c.method(...) with synchronized(c){
  c.method(...) }
for all instances c of class C
- A synchronized modifier on a static method takes the monitor of the Class object for that class
- ‘good’ object-orientated design is to only use synchronized at the method level
- Code in synchronized blocks occurs atomically and is its effects are immediately visible to all other threads
- Compiles to bytecode instructions monitorenter and monitorexit (javap - c)
public class GoingToTheRacesSafely {
    public static void main(String[] args) throws Exception {
        Go[] gos = new Go[10];
        for(int i=0;i<10;i++) gos[i] = new Go();
        for(int i=0;i<10;i++) gos[i].start();
        for(int i=0;i<10;i++) gos[i].join();
        System.out.println(counter);
    }
    private static int counter = 0;
    private static class Go extends Thread {
        public void run(){
            for(int i=0;i<10000;i++) {
                synchronized(GoingToTheRacesSafely.class){
                    int c = counter;
                    counter = c+1;
                }
            }
        }
    }
}
Where it goes right

Compiled from "GoingToTheRacesSafely.java"
class GoingToTheRacesSafely$Go extends java.lang.Thread{
    public void run(){
        Code:
        0:   iconst_0
        1:   istore_1
        2:   iload_1
        3:   sipush 10000
        6:   if_icmpge 36
        9:   ldc_w  #3; //class GoingToTheRacesSafely
       12:   dup
       13:   astore_2
       14:   monitorenter
       15:   iconst_1
       16:   invokestatic  #4; //Method GoingToTheRacesSafely.access$112:(I)I
       19:   pop
       20:   aload_2
       21:   monitorexit
       22:   goto   30
       25:   astore_3
       26:   aload_2
       27:   monitorexit
       28:   aload_3
       29:   athrow
       30:   iinc   1, 1
       33:   goto   2
       36:   return

        Exception table:
        from   to    target  type
          15    22    25     any
          25    28    25     any
Compiled from "GoingToTheRacesSafely.java"

```java
class GoingToTheRacesSafely$Go extends java.lang.Thread{
    public void run()
    {
        Code:
        0:  iconst_0
        1:  istore_1
        2:  iload_1
        3:  sipush 10000
        6:  if_icmpge  36
        9:  ldc_w    #3; //class GoingToTheRacesSafely
       12:  dup
       13:  astore_2
       14:  monitorenter
       15:  iconst_1
       16:  invokestatic  #4; //Method GoingToTheRacesSafely.access$112:(I)I
       19:  pop
       20:  aload_2
       21:  monitorexit
       22:  goto    30
       25:  astore_3
       26:  aload_2
       27:  monitorexit
       28:  aload_3
       29:  athrow
       30:  iinc    1, 1
       33:  goto    2
       36:  return

    Exception table:
    from   to    target  type
    15     22   25       any
    25     28   25       any
    }
```
Where it goes right
Unsynchronized code

A common problem: People forget that synchronized code is only synchronized with other synchronized code.

```java
public class Account{
    private int balance=0;
    public int get() return balance;
    public void set(int x){ balance=x; }
}

Account acc = new Account();

synchronized(acc){
    int value = acc.get();
    acc.set(value+10);
}

acc.set(0);
```
Unsynchronized code

A common problem: People forget that synchronized code is only synchronized with other synchronized code.

```java
public class Account{
    private int balance=0;
    public int get() return balance;
    public void set(int x){ balance=x; }
}

Account acc = new Account();
...  
synchronized(acc){
    int value = acc.get();
    acc.set(value+10);
    acc.set(value+10);
}
...  
acc.set(0);  
```
Unsynchronized code

A common problem: People forget that synchronized code is only synchronized with other synchronized code.

```java
public class Account{
    private int balance=0;
    public int get() return balance;
    public synchronized void set(int x){ balance=x; }
    public synchronized void add(int x) { balance += x; }
}

Account acc = new Account();
... acc.add(10);
... acc.set(0);
```
public class BatchBanking{

    private static class Transfer{
        public final Account from;
        public final Account to;
        public final int amount;
        public Transfer(Account f, Account t, int a)
        {
            from=f; to=t; amount=a;
        }
    }

    private static class Account{
        private int balance=0;
        public Account(int b){ balance=b;}
        public int get(){ return balance;}
        public void set(int x){ balance=x;}
        public void add(int x){ balance+=x;}
        public void subtract(int x){ balance-=x;}
    }

    ...
private static Account[] accounts = new Account[10];
static{
    for(int i=0;i<10;i++)
        accounts[i] = new Account(1000);
}

private static List<Transfer> generateTransfers(){
    List<Transfer> transfers = new ArrayList<Transfer>();
    for(int i=0;i<1000;i++){
        int amount = ((int)Math.random()+1)*10;
        Account from = accounts[((int)Math.random())*10];
        Account to = accounts[((int)Math.random())*10];
        transfers.add(new Transfer(from,to,amount));
    }
    return transfers;
}
Spot the problem

... public static void main(String[] args){
    (new Banker(generateTransfers())).start();
    (new Banker(generateTransfers())).start();
}
private static class Banker extends Thread{
    private final List<Transfer> transfers;
    public Banker(List<Transfer> t){transfers=t;}
    public void run() {
        for(Transfer transfer : transfers)
            synchronized(transfer.from){
                synchronized(transfer.to){
                    transfer.from.subtract(transfer.amount);
                    transfer.to.add(transfer.amount);
                    try sleep(100); catch(InterruptedException e){};
                }
            }
    }
}
Spot the problem

...
Deadlock

ThreadOne

synchronized(Account1)
  has lock for Account1
synchronized(Account2)
  waiting for lock for Account2

ThreadTwo

synchronized(Account2)
  has lock for Account2
synchronized(Account1)
  waiting for lock for Account1
Solutions?

• Need to break the stalemate
• Most common solution: Ordered Access
  - Give an ordering to threads - let the bigger thread win
  - Give an ordering to objects - take the biggest object first
  - Second is easier
    - Often an easy way to order objects i.e. hashCode
Solutions?

- Need to break the stalemate
Solutions?

• Need to break the stalemate
• Most common solution: Ordered Access
  • Give an ordering to threads - let the bigger thread win
  • Give an ordering to objects - take the biggest object first
  • Second is easier
  • Often an easy way to order objects i.e. hashCode
Coordinating Threads

Consider a system where we can have many Readers accessing a file but only a single Writer. What are our requirements?

- When a writer wants to enter it needs to *wait* for all readers to exit
- When a reader wants to enter it needs to *wait* for the writer to exit
- What if a reader wants to enter when a writer is waiting?
  1. Make the reader wait
  2. Let the reader in
- What can go wrong in the second case?
Condition Synchronization

Using wait and notify

- `wait()`: suspend and block thread; release lock
- `notify()`: unblock a suspended thread
- `notifyAll()`: unblock all suspended threads
- each lock has a *wait set* of waiting threads
synchronized type1 Method1 (...) throws InterruptedException {
    while (!condition){ wait(); }
    ...
}

synchronized type2 Method2 (...) {
    ...
    condition = true;
    notify(); // or notifyAll()
    ...
}

Note: signal-and-continue semantics for notify()
wait() and notify()

- Use `notifyAll` if any of the waiting threads are guarded by a condition to avoid ‘missed wake-up’
- Use `while(!condition) wait()` if it is only safe to proceed when condition is true
- When might `notify` by itself be okay?
Java Monitors: An Overview

- In HotSpot the Entry Set is a List
- In HotSpot `notify` adds to the EntryList (since 1.6)
Java Monitor vs. Monitor

- Java monitors are not the monitors discussed in this course
- But we can simulate these in Java
  - all class variables private
  - all non private methods synchronized
  - condition synchronization using \texttt{wait} and \texttt{notify}
  - see \texttt{java.util.concurrent.locks}
Thread Lifecycle

- **Created**
  - `new Thread()`
  - `start()`
  - `stop()`

- **Running**
  - `yield()`
  - `sleep()`
  - `suspend()`

- **Runnable**
  - `suspend()`
  - `resume()`
  - `stop()`
  - `or run() returns`

- **Non-Runnable**

- **Terminated**
Thread Lifecycle

- Created
  - start()
  - stop()
- Running
  - yield()
  - suspend()
  - sleep()
- Runnable
  - resume()
  - suspend()
- Non-Runnable
  - resume()
  - suspend()
- Terminated
  - stop() or run() returns

Alive

isAlive()
Thread Lifecycle

- **Created**
  - new Thread()
  - start()
  - stop()

- **Running**
  - yield()
  - sleep()
  - suspend()
  - resume()

- **Runnable**
  - suspend()

- **Non-Runnable**
  - resume()

- **Terminated**
  - stop()
  - or run() returns
LTS for Thread Lifecycle

START

STOP

{end, stop}

RUN

DISPATCH

YIELD

SUSPEND

RESUME

STOP

STOP

STOP
Actual States

- The previous slides don’t exactly reflect that states returned by `getState`
- The actual states for a thread are given in the enumeration `java.lang.Thread.state`
  - NEW
  - Runnable
  - BLOCKED
  - WAITING
  - TIMED_WAITING
  - TERMINATED
- Can be used to check if a thread is blocked or waiting
- Can use `holdsLock(obj)` to see if a thread is holding the monitor for an object
Why are stop, suspend and resume deprecated?

- stop lets go of all locks

- suspend keeps hold of locks

- Should use a flag to stop a thread
  
  ```java
  private volatile boolean still_running = true;
  public void stop(){ still_running = false;}
  public void run(){
    while(still_running){ ... }
  }
  ```

- Use wait to suspend self, use flags to suspend other threads

http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html
Why are stop, suspend and resume deprecated?

- stop lets go of all locks
  - If the state protected by those locks is in an inconsistent state, this inconsistent state will be visible to other threads
- suspend keeps hold of locks

Should use a flag to stop a thread

```java
private volatile boolean still_running = true;
public void stop(){ still_running = false;}
public void run(){
    while(still_running){ ... }
}
```

- Use wait to suspend self, use flags to suspend other threads

http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html
Why are stop, suspend and resume deprecated?

- stop lets go of all locks
  - If the state protected by those locks is in an inconsistent state, this inconsistent state will be visible to other threads

- suspend keeps hold of locks
  - If a thread is holding locks and not making progress this is never good - can lead to deadlock and lack of progress in other threads

- Should use a flag to stop a thread
  ```java
  private volatile boolean still_running = true;
  public void stop(){ still_running = false; }
  public void run(){
      while(still_running){ ... } }
  }
  ```

- Use wait to suspend self, use flags to suspend other threads

http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html
IllegalMonitorStateException

- Thrown if you try and call a method on a thread when it’s not in a state where you can call that method
- Most commonly encountered when calling `wait` - you can only call `wait` on an object (including `this`) if you hold the lock for that object
The Thread class

- The Thread class provides other functionality
- We’re now going to look at some of them
- Note that most Thread methods are native
Interrupting

- interrupt - sets the interrupted status, and depending on the state (waiting or sleeping) of the target thread this may cause an InterruptedException to be thrown
- interrupted - checks and clears the interrupted status
- isInterrupted - checks the interrupted status
- Note that if you call interrupt on a thread that’s running, then that thread must poll its own interrupted status to find out
Joining in

- `join` - waits for another thread to finish executing before continuing
- If the other thread terminates due to an exception or if the `join` times out (if a timeout is given) then this is treated in the same way as if the other thread terminates normally
- `join` is interruptible
Having a Rest

- **sleep** - suspends the thread for a given length of time, is interruptible
- **yield** - indicates that the scheduler can preempt this thread
  - Is system dependent as uses system calls
  - Windows - `SwitchToThread()`
  - Linux - `sched_yield()`
  - These have different behaviors
  - Often `sleep` or `join` are better
Don’t hold me back

- Daemon threads do not prevent the JVM from finishing - it will carry on running whilst there is a non-daemon thread still running
- setDaemon - must be called *before* the thread is started
- isDaemon - returns true iff it’s a daemon
Who’s Boss

- You can give a thread a *priority*
- `setPriority`, `getPriority`
- In HotSpot this maps down to a priority in the OS and effects OS scheduling
- I’ve never had the need to use this
There’s no place like home

- Every thread belongs to a *Thread Group*
- Can be useful for organising groups of threads
- Defaults to the Thread Group of the thread that created it
- Is set to null when the thread dies - another way of checking if a thread is alive
Next Time

We’ll look at

- `java.util.concurrent`
- Focussing on concurrent access to Data Structures
- Using an example that makes use of `map reduce` and `work stealing`
Suggested Reading

Some places where you can read about Java concurrency:

- Java Language Specification : Chapter 17 -
  http://java.sun.com/docs/books/jls/
- Java Tutorials - http://docs.oracle.com/javase/tutorial/essential/concurrency/
- HotSpot internals -
- Java monitors -
- Lots of blogs and forums... google it!