# THREADS AND CONCURRENCY

Lecture 20 – CS2110 – Fall 2009

## What is a Thread?

2

- A separate process that can perform a computational task independently and concurrently with other threads
  - Most programs have only one thread
  - GUIs have a separate thread, the *event dispatching thread*
  - A program can have many threads
  - You can create new threads in Java

## What is a Thread?

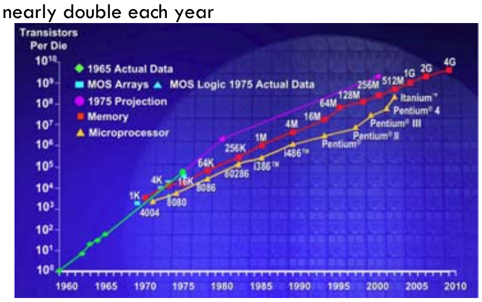
3

- On many machines, threads are an illusion
  - Not all machines have multiple processors
  - But a single processor can share its time among all the active threads
  - Implemented with support from underlying operating system or virtual machine
  - Gives the illusion of several threads running simultaneously
- But modern computers often have "multicore" architectures: multiple CPUs on one chip

## Why Multicore?

4

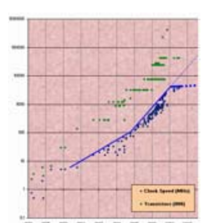
- Moore's Law: Computer speeds and memory densities nearly double each year




## But a fast computer runs hot

5

- Power dissipation rises as the square of the CPU clock rate
- Chips were heading towards melting down!
- Multicore: with four CPUs (cores) on one chip, even if we run each at half speed we get more overall performance!



## Concurrency (aka Multitasking)

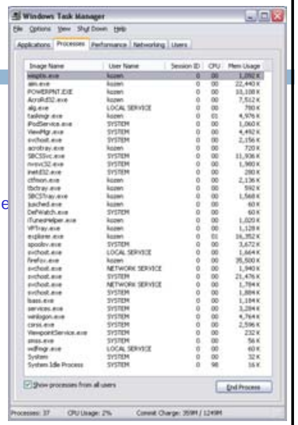
6

- Refers to situations in which several threads are running simultaneously
- Special problems arise
  - race conditions
  - deadlock

## Task Manager

7

- The operating system provides support for multiple "processes"
- In reality there there may be fewer processors than processes
- Processes are an illusion too – at the hardware level, lots of multitasking
  - memory subsystem
  - video controller
  - buses
  - instruction prefetching



## Threads in Java

8

- Threads are instances of the class **`Thread`**
  - can create as many as you like
- The Java Virtual Machine permits multiple concurrent threads
  - initially only one thread (executes **`main`**)
- Threads have a priority
  - higher priority threads are executed preferentially
  - a newly created **`Thread`** has initial priority equal to the thread that created it (but can change)

## Creating a new Thread (Method 1)

9

```
class PrimeThread extends Thread {
   long a, b;

   PrimeThread(long a, long b) {
      this.a = a; this.b = b;
   }

   public void run() {
      //compute primes between a and b
      ...
   }
}
```

overrides **`Thread.run()`**

can call **`run()`** directly – the calling thread will run it

```
PrimeThread p = new PrimeThread(143, 195);
p.start();
```

or, can call **`start()`** – will run **`run()`** in new thread

## Creating a new Thread (Method 2)

10

```
class PrimeRun implements Runnable {
   long a, b;

   PrimeRun(long a, long b) {
      this.a = a; this.b = b;
   }

   public void run() {
      //compute primes between a and b
      ...
   }
}
```

```
PrimeRun p = new PrimeRun(143, 195);
new Thread(p).start();
```

## Example

11

```
public class ThreadTest extends Thread {

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }

   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }
}
```

```
Thread[Thread-0,5,main] 0
Thread[main,5,main] 0
Thread[main,5,main] 1
Thread[main,5,main] 2
Thread[main,5,main] 3
Thread[main,5,main] 4
Thread[main,5,main] 5
Thread[main,5,main] 6
Thread[main,5,main] 7
Thread[main,5,main] 8
Thread[main,5,main] 9
Thread[Thread-0,5,main] 1
Thread[Thread-0,5,main] 2
Thread[Thread-0,5,main] 3
Thread[Thread-0,5,main] 4
Thread[Thread-0,5,main] 5
Thread[Thread-0,5,main] 6
Thread[Thread-0,5,main] 7
Thread[Thread-0,5,main] 8
Thread[Thread-0,5,main] 9
```

## Example

12

```
public class ThreadTest extends Thread {

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }

   public void run() {
      currentThread().setPriority(4);
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }
}
```

```
Thread[main,5,main] 0
Thread[main,5,main] 1
Thread[main,5,main] 2
Thread[main,5,main] 3
Thread[main,5,main] 4
Thread[main,5,main] 5
Thread[main,5,main] 6
Thread[main,5,main] 7
Thread[main,5,main] 8
Thread[main,5,main] 9
Thread[Thread-0,4,main] 0
Thread[Thread-0,4,main] 1
Thread[Thread-0,4,main] 2
Thread[Thread-0,4,main] 3
Thread[Thread-0,4,main] 4
Thread[Thread-0,4,main] 5
Thread[Thread-0,4,main] 6
Thread[Thread-0,4,main] 7
Thread[Thread-0,4,main] 8
Thread[Thread-0,4,main] 9
```

## Example

13

```
public class ThreadTest extends Thread {

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }

   public void run() {
      currentThread().setPriority(6);
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }
}
```

```
Thread[main,5,main] 0
Thread[main,5,main] 1
Thread[main,5,main] 2
Thread[main,5,main] 3
Thread[main,5,main] 4
Thread[main,5,main] 5
Thread[Thread-0,6,main] 0
Thread[Thread-0,6,main] 1
Thread[Thread-0,6,main] 2
Thread[Thread-0,6,main] 3
Thread[Thread-0,6,main] 4
Thread[Thread-0,6,main] 5
Thread[Thread-0,6,main] 6
Thread[Thread-0,6,main] 7
Thread[Thread-0,6,main] 8
Thread[Thread-0,6,main] 9
Thread[main,5,main] 6
Thread[main,5,main] 7
Thread[main,5,main] 8
Thread[main,5,main] 9
```

## Example

14

```
public class ThreadTest extends Thread {
   static boolean ok = true;

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.println("waiting...");
         yield();
      }
      ok = false;
   }

   public void run() {
      while (ok) {
         System.out.println("running...");
         yield();
      }
      System.out.println("done");
   }
}
```

allows other waiting threads to run

```
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
done
```

## Stopping Threads

15

- Threads normally terminate by returning from their run method
- **`stop()`**, **`interrupt()`**, **`suspend()`**, **`destroy()`**, etc. are all deprecated
  - can leave application in an inconsistent state
  - inherently unsafe
  - don't use them
  - instead, set a variable telling the thread to stop itself

## Daemon and Normal Threads

16

- A thread can be *daemon* or *normal*
  - the initial thread (the one that runs **`main`**) is normal
- Daemon threads are used for minor or ephemeral tasks (e.g. timers, sounds)
- A thread is initially a daemon iff its creating thread is
  - but this can be changed
- The application halts when either
  - **`System.exit(int)`** is called, or
  - all normal (non-daemon) threads have terminated

## Race Conditions

17

- A *race condition* can arise when two or more threads try to access data simultaneously
- Thread B may try to read some data while thread A is updating it
  - updating may not be an atomic operation
  - thread B may sneak in at the wrong time and read the data in an inconsistent state
- Results can be unpredictable!

## Example – A Lucky Scenario

18

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   if (stack.isEmpty()) return;
   String s = stack.pop();
   //do something with s...
}
```

Suppose threads A and B want to call **`doSomething()`**, and there is one element on the stack

1. thread A tests **`stack.isEmpty()`** false
2. thread A pops    stack is now empty
3. thread B tests **`stack.isEmpty()`**   true
4. thread B just returns – nothing to do

## Example – An Unlucky Scenario

19

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   if (stack.isEmpty()) return;
   String s = stack.pop();
   //do something with s...
}
```

Suppose threads A and B want to call **doSomething()**, and there is one element on the stack

1. thread A tests **stack.isEmpty()** false
2. thread B tests **stack.isEmpty()** false
3. thread A pops stack is now empty
4. thread B pops Exception!

## Solution – Locking

20

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   synchronized (stack) {
      if (stack.isEmpty()) return;
      String s = stack.pop();
   }
   //do something with s...
}
```

**synchronized** block

- Put critical operations in a **synchronized** block
- The **stack** object acts as a lock
- Only one thread can own the lock at a time

## Solution – Locking

21

- You can lock on any object, including **this**

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

is equivalent to

```
public void doSomething() {
   synchronized (this) {
      ...
   }
}
```

## File Locking

22

- In file systems, if two or more processes could access a file simultaneously, this could result in data corruption
- A process must *open* a file to use it – gives exclusive access until it is *closed*
- This is called *file locking* – enforced by the operating system
- Same concept as **synchronized(obj)** in Java

## Deadlock

23

- The downside of locking – *deadlock*
- A *deadlock* occurs when two or more competing threads are waiting for the other to relinquish a lock, so neither ever does
- Example:
  - thread A tries to open file X, then file Y
  - thread B tries to open file Y, then file X
  - A gets X, B gets Y
  - Each is waiting for the other forever

## wait/notify

24

- A mechanism for event-driven activation of threads
- Animation threads and the **GUI** event-dispatching thread in can interact via **wait/notify**

## wait/notify



animator:

```
boolean isRunning = true;

public synchronized void run() {
   while (true) {
      while (isRunning) {
         //do one step of simulation
      }
      try {
         wait();
      } catch (InterruptedException ie) {}
      isRunning = true;
   }
}
```

relinquishes lock on **animator** – awaits notification

```
public void stopAnimation() {
   animator.isRunning = false;
}

public void restartAnimation() {
   synchronized(animator) {
      animator.notify();
   }
}
```

notifies processes waiting for **animator** lock

## Summary



- Use of multiple processes and multiple threads within each process can exploit concurrency
  - Which may be real (multicore) or "virtual" (an illusion)
- But when using threads, beware!
  - Must lock (synchronize) any shared memory to avoid non-determinism and race conditions
  - Yet synchronization also creates risk of deadlocks
  - Even with proper locking concurrent programs can have other problems such as "livelock"
- Serious treatment of concurrency is a complex topic (covered in more detail in cs3410 and cs4410)

## Reminder



- A "race condition" arises if two threads try and share some data
- One updates it and the other reads it, or both update the data
- In such cases it is possible that we could see the data "in the middle" of being updated
  - A "race condition": correctness depends on the update racing to completion without the reader managing to glimpse the in-progress update
  - Synchronization (aka mutual exclusion) solves this

## Java Synchronization (Locking)



```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   synchronized (stack) {
      if (stack.isEmpty()) return;
      String s = stack.pop();
   }
   //do something with s...
}
```

**synchronized** block

- Put critical operations in a **synchronized** block
- The **stack** object acts as a lock
- Only one thread can own the lock at a time

## Java Synchronization (Locking)



- You can lock on any object, including **this**

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

is equivalent to

```
public void doSomething() {
   synchronized (this) {
      ...
   }
}
```

## How locking works



- Only one thread can "hold" a lock at a time
  - If several request the same lock, Java somehow decides which will get it
- The lock is released when the thread leaves the synchronization block
  - synchronized(someObject) { *protected code* }
  - The protected code has a *mutual exclusion* guarantee: At most one thread can be in it
- When released, some other thread can acquire the lock

## Locks are associated with objects

31

- Every Object has its own built-in lock
  - Just the same, some applications prefer to create special classes of objects to use just for locking
  - This is a stylistic decision and you should agree on it with your teammates or learn the company policy if you work at a company
- Code is "thread safe" if it can handle multiple threads using it… otherwise it is "unsafe"

## File Locking: Same idea

32

- In file systems, if two or more processes could modify a file simultaneously, this could result in data corruption
  - A process must *open* a file to modify it – gives exclusive access until it is *closed*
  - Multiple processes can open the same file to read it
- This *file locking* synchronization rule is enforced by the operating system

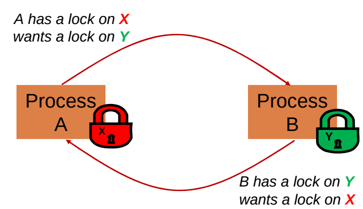
## Deadlock

33

- The downside of locking – *deadlock*
- A *deadlock* occurs when two or more competing threads each hold a lock, and each are waiting for the other to relinquish a lock, so neither ever does
- Example:
  - thread A tries to open file X, then file Y
  - thread B tries to open file Y, then file X
  - A gets X, B gets Y
  - Each is waiting for the other forever

## Visualizing deadlock

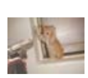
34




## Deadlocks always involve cycles

35

- They can include 2 or more threads or processes in a waiting cycle
- Other properties:
  - The locks need to be mutually exclusive (no sharing of the objects being locked)
  - The application won't give up and go away (no timer associated with the lock request)
  - There are no mechanisms for one thread to take locked resources away from another thread – no "preemption"



## `wait/notify`

36

- A mechanism for event-driven activation of threads
- Animation threads and the **`GUI`** event-dispatching thread in can interact via **`wait/notify`**

## wait/notify

37

animator:

```
boolean isRunning = true;

public synchronized void run() {
   while (true) {
      while (isRunning) {
         //do one step of simulation
      }
      try {
         wait();
      } catch (InterruptedException ie) {}
      isRunning = true;
   }
}
```

relinquishes lock on **animator** – awaits notification

```
public void stopAnimation() {
   animator.isRunning = false;
}

public void restartAnimation() {
   synchronized(animator) {
      animator.notify();
   }
}
```

notifies processes waiting for **animator** lock

## A producer/consumer example

38

- Thread A produces loaves of bread and puts them on a shelf with capacity K
  - For example, maybe K=10
- Thread B consumes the loaves by taking them off the shelf
  - Thread A doesn't want to overload the shelf
  - Thread B doesn't wait to leave with empty arms

*producer*   *shelves*  *consumer*

## Producer/Consumer example

39

```
class Bakery {
    int nLoaves = 0;    // Current number of waiting loaves
    final int K = 10;   // Shelf capacity

    public synchronized void produce() {
       while(nLoaves == K) this.wait();  // Wait until not full
       ++nLoaves;
       this.notifyall();                 // Signal: shelf not empty
    }

    public synchronized void consume() {
       while(nLoaves == 0) this.wait();  // Wait until not empty
       --nLoaves;
       this.notifyall();                 // Signal: shelf not full
    }
}
```

## Things to notice

40

- Wait needs to wait on the same Object that you used for synchronizing (in our example, "this", which is this instance of the Bakery)
- Notify wakes up just one waiting thread, notifyall wakes all of them up
- We used a while loop because we can't predict exactly which thread will wake up "next"

## Trickier example

41

- Suppose we want to use locking in a BST
  - Goal: allow multiple threads to search the tree
  - But don't want an insertion to cause a search thread to throw an exception

## Code we're given is unsafe

```
class BST {
    Object name;      // Name of this node
    Object value;     // Value of associated with that name
    BST left, right;  // Children of this node

    // Constructor
    public void BST(Object who, Object what) { name = who; value = what; }

    // Returns value if found, else null
    public Object get(Object goal) {
        if(name.equals(goal)) return value;
        if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
        return right==null? null: right.get(goal);
    }

    // Updates value if name is already in the tree, else adds new BST node
    public void put(Object goal, object value) {
        if(name.equals(goal)) { this.value = value; return; }
        if(name.compareTo(goal) < 0) {
            if(left == null) { left = new BST(goal, value); return; }
            left.put(goal, value);
        } else {
            if(right == null) { right = new BST(goal, value); return; }
            right.put(goal, value);
        }
    }
}
```

## Attempt #1

43

- Just make both put and get synchronized:
  - public synchronized Object get(…) { … }
  - public synchronized void put(…) { … }
- Let's have a look….

## Safe version: Attempt #1

```
class BST {
    Object name;      // Name of this node
    Object value;     // Value of associated with that name
    BST left, right;  // Children of this node

    // Constructor
    public void BST(Object who, Object what) { name = who; value = what; }

    // Returns value if found, else null
    public synchronized Object get(Object goal) {
        if(name.equals(goal)) return value;
        if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
        return right==null? null: right.get(goal);
    }

    // Updates value if name is already in the tree, else adds new BST node
    public synchronized void put(Object goal, object value) {
        if(name.equals(goal)) { this.value = value; return; }
        if(name.compareTo(goal) < 0) {
            if(left == null) { left = new BST(goal, value); return; }
            left.put(goal, value);
        } else {
            if(right == null) { right = new BST(goal, value); return; }
            right.put(goal, value);
        }
    }
}
```

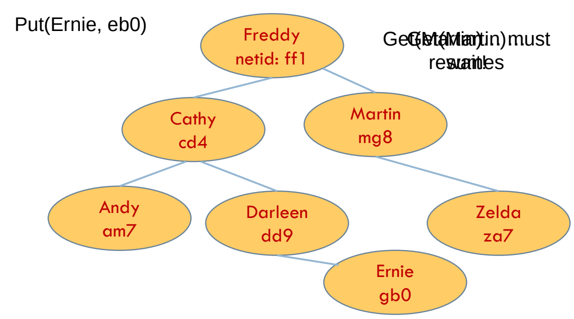
## Attempt #1

45

- Just make both put and get synchronized:
  - public synchronized Object get(…) { … }
  - public synchronized void put(…) { … }
- This works but it kills ALL concurrency
  - Only one thread can look at the tree at a time
  - Even if all the threads were doing "get"!

## Visualizing attempt #1

46




## Attempt #2

47

- put uses synchronized in method declaration
  - So it locks every node it visits
- get tries to be fancy:

```
    // Returns value if found, else null
    public Object get(Object goal) {
        synchronized(this) {
          if(name.equals(goal)) return value;
          if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
          return right==null? null: right.get(goal);
        }
    }
```

- Actually this is identical to attempt 1! It only looks different but in fact is doing exactly the same thing

## Attempt #3

48

```
    // Returns value if found, else null
    public Object get(Object goal) {
        boolean checkLeft = false, checkRight = false;
        synchronized(this) {
          if(name.equals(goal)) return value;
          if(name.compareTo(goal) < 0) {
                if (left==null) return null; else checkLeft = true;
          } else {
                if                                              true;
          }
        }
        if (checkLeft) return left.get(goal);
        if (checkRight) return right.get(goal);

        /* Never executed but keeps Java happy */ return null;
    }
```

relinquishes lock on **this** – next lines are "unprotected"

- Risk: "get" (read-only) threads sometimes look at nodes without locks, but "put" always updates those same nodes.
- According to JDK rules this is unsafe

## Attempt #3 illustrates risks

49

- The hardware itself actually needs us to use locking and attempt 3, although it looks right in Java, could actually malfunction in various ways
  - Issue: put updates several fields:
    - parent.left (or parent.right) for its parent node
    - this.left and this.right and this.name and this.value
  - When locking is used correctly, multicore hardware will correctly implement the updates
  - But if you look at values without locking, as we did in Attempt #3, hardware can malfunction!

## Why can hardware malfunction?

50

- Issue here is covered in cs3410 & cs4410
  - Problem is that the hardware was designed under the requirement that if threads contend to access shared memory, then readers and writers must use locks
  - Solutions #1 and #2 used locks and so they worked, but had no concurrency
  - Solution #3 violated the hardware rules and so you could see various kinds of garbage in the fields you access!
- In fact it is quite hard to design concurrent data structures that respect the hardware rules

## Summary

51

- Use of multiple processes and multiple threads within each process can exploit concurrency
  - Which may be real (multicore) or "virtual" (an illusion)
- But when using threads, beware!
  - Must lock (synchronize) any shared memory to avoid non-determinism and race conditions
  - Yet synchronization also creates risk of deadlocks
  - Even with proper locking concurrent programs can have other problems such as "livelock"
- Serious treatment of concurrency is a complex topic (covered in more detail in cs3410 and cs4410)
  - ECE/CS 3420, looks at why the hardware has this issue but not from the perspective of writing concurrent code