Threading in Java

Overview

- Introduction
- Thread Lifecycle
- Data sharing
- Locking
- Deadlocks
- Sophisticated synchronization

Class Design

- Server architecture
- Other languages

Motivation

- Multithreaded environment is common in many situations
  - Web servers, app servers, etc
  - Operating systems kernels
  - GUIs
- It’s easy to make mistakes with code in a multithreaded environment
  - Code that is correct in single-threaded environment may behave wrongly in a multi-threaded one
  - You may not even realise that your code will be used with multithreading!

Difficulties I

- Multiple activities interleave in different ways every time the system runs
  - outcomes are not repeatable
    - “Non-determinism”
    - how do you know if a test passed or failed?
      - you can’t simply compare actual output to correct output!
  - debugging is hard
    - correctness means that every execution is correct
    - testing only checks some executions
    - how do you know whether a bug-fix worked?
Difficulties II

- One activity can cause unexpected changes in the data used in another activity
  - “Data corruption”
  - Errors may not be seen in every execution
    - Perhaps only under high load

Difficulties III

- System may keep running but nothing happens
  - “Deadlock”
  - Performance is unexpectedly poor

Assumed knowledge

- Object-oriented programming in Java
  - Objects, classes, fields, methods, constructors, control flow
  - Inheritance, polymorphism, interfaces, overloading
  - Collections, Exceptions
- See B. Eckel, “Thinking in Java (3rd ed)”

Further reading references

- Sun tutorial on Java: threads topic
- A. Holub “Taming Java Threads”
  - www.holub.com has presentation etc
- D. Lea “Concurrent Programming in Java (2nd ed)”
- New library classes in Java 1.5
  - (java.util.concurrent, etc)
  - Lead specifier is Doug Lea!
Acknowledgements

Based on materials developed with funding from “Building the Web Workforce” Science Lectureship, from Australian Government

- Part authored by Joe Thurbon and Masahiro Takatsuka

Overview

- Introduction
- Thread Lifecycle
- Data sharing
- Locking
- Deadlocks
- Sophisticated synchronization

Class Design

- Server architecture
- Other languages

Single Thread Execution

- Review from previous experience

Object model

- Each object has instance variables described by the class
  - They persist from object construction until garbage-collection
- Each method call has local variables described by the code
  - They live only for the duration of the call
- Each variable has a type (declaration)
  - Type can be primitive (boolean, int, …) or reference to a Class
  - At any time each variable has a value
  - Inter-object references form data structures (sharing is important)

Single thread (cntd)

- Execution starts at “main()”
  - Creates objects and calls methods on them
  - Control flow governed by syntax of code
    - Control can move to other classes on method call, then returns when method finishes
- A method call
  - can be treated as if atomic (all the code from call to return can be considered a step)
  - gets its own bit of memory (local variables on the stack)
    - they live till the method call returns
  - can access the global memory (via instance variables, which are on the heap)
Threading in Java

Single thread execution

Code
```java
public static void main() {
    Foo myFoo = new Foo();
    myFoo.bar(5);
}
```

Class Foo {
    private int myVar = 0;
    public void bar(int x) {
        int y = x+1;
        myVar = myVar – y;
        y = 3;
    }
}

Log (with state shown)
- Start of main()
- Construct a Foo
- Set myFoo to that instance
- Call bar(5) on myFoo
- Read x (value 5)
- Set y to be 6
- Read myVar (value 0)
- Read y (value 6)
- Set myVar to be -6
- Set y to be 3
- Return from bar(5)
- End of main()

In that example...
- The instance variable ‘myVar’ is stored on the heap (as part of the instance object refered to by myFoo)
  - It is accessible to all methods of the instance myFoo
  - It persists after the call bar(5) terminates
  - It would keep its value if another call were made
- The local variables ‘x’ and ‘y’ are local to the call bar(5) (stored on the stack)
  - They are not accessible in other methods
  - If x, y declared in other methods, those are different variables!

Multiple Threads Interleaving

A thread executes some steps
- Then the JVM swaps to another thread
- Which executes several steps
- Then swap to another thread again
  - No predictable order for threads to get turns
  - No predictable amount of time before the next swap
- Each time a thread gets to execute again, control continues from where that thread was, the previous time it was executing

Multiple threads executing

A method call in one thread
- Its steps can be interleaved with another thread’s activity (in fact, it’s worse than one thinks – stages within a single Java statement can be interleaved)
  - i.e. method calls are not atomic
  - still gets its own piece of memory on stack
  - for its local variables
  - shares the heap with other threads

When the thread starts running
- It executes method run()
  - This may call other methods, perhaps of other objects
Multithreaded execution example
(different objects, different code)

Code (class of thread 1)
public void run() {
    obj1.freckle(5);
}

In class of obj1
private int aVar;
public void freckle(int x) {
    boolean y = (x < 6);
    if (y)
        aVar = aVar - 3;
    else {
        aVar = aVar * x;
    }
}

Code (thread 2)
public void run() {
    obj2.gnarl();
}

In class of obj2
private int myVar;
public void gnarl() {
    int x = myVar+1;
    int y = x * 2;
    myVar = myVar - y;
}

Execution A

T1: call obj1.freckle(5)
T2: call obj2.gnarl()
T1: read x (value 5)
T1: read myVar (value: 3)
T2: set x to be 4
T1: is x < 6 (result: true)
T2: read x (value: 4)
T2: set y to be 8
T1: set y to be true
T1: read y, take "then" branch
T1: read aVar (value: 16)
T2: set y to be 8
T2: read myVar (value: 3)
T2: set myVar to be -5
T1: return from obj1.freckle()
T2: return from obj2.gnarl()

Stack
Of T1

Stack
Of T2

obj1.aVar
obj2.myVar

x y
16 3

T1: x y
5 ?
T2: x y
4 ?

T1: set y to be true
T1: read y, take "then" branch
T1: read aVar (value: 16)
T2: set y to be 8
T2: read myVar (value: 3)
T1: return from obj1.freckle()
T2: return from obj2.gnarl()

Interference

In this example, each thread dealt with separate data
- Outcome was the same for different interleavings
- In general, there can be shared data (on the heap, used in several threads)
  - When one thread’s activities interleave with another thread, shared data can be affected
  - Controlling this is a main focus of these lectures
- Stay tuned!
Threading in Java

Thread Lifecycle

- In the examples, the threads were both always runnable: the scheduler could make one active at anytime
- In general, there are four states for a thread
  - created
  - runnable
  - non-runnable
  - terminated

Created

- A thread is in this state between the time that it is constructed (a thread is just an object) and the time that start is called
- A thread in this state is basically useless
  - But you need to be able to create a thread and start it later, so you need this state

Runnable

- Once start is called, the thread is runnable
  - It begins to execute run()
- At a given time, it might not actually be running, though
  - There could be other threads
  - JVM scheduler gives each turns at running
- This is the most ‘normal’ state for a thread

Non-runnable

- The thread is not able to be run at the moment (i.e. the scheduler cannot pick it)
  - Thread executes Thread.sleep(..) – tell the JVM to stop this thread being runnable for a fixed period of time
  - pause – tell the thread to stop being runnable until someone calls resume
    - Don’t use pause or resume! They are deprecated.
  - Other mechanisms involving blocking on a lock
    - We’ll study these later
Terminated

- The thread has stopped because *run* finished
  - or because someone called *stop* on it
  - *stop()* is deprecated, so never call *stop*!

- Once it’s terminated, it cannot ever come back to life
  - No-one can call *start()* again on it

The JVM scheduler

- Once another thread has been started, the original and the new one will take turns to run
  - The JVM scheduler will interleave them
  - If more than two threads are runnable, the JVM will interleave all of them
  - JVM decides when to swap between threads
  -Coder doesn’t know when one thread will have another interleave
    - It could be between any statements
    - It could even be between stages in a single statement

Influencing the scheduler

- The coder can’t control the scheduler
  - Code ought to be correct no matter how the scheduler arranges its interleaving!
- Code can give *hints* to the scheduler
- Thread calls *Thread.yield()*
  - Hint to encourage another thread to get a go at running
- Adjust thread priority
  - Hint to give more running time to higher priority threads

Inter-thread communication

- Each thread runs with its own stack
  - But they share access to the heap
- If one thread needs to tell another thread what is happening, a common way is to write the information in some instance variable that the other thread can read
- Also, use *wait/notify* (stay tuned!)
Join

One thread can block till another has terminated, by calling `join()`.

- It is an idiom for main to start several threads which interleave, then main blocks till all children have finished.
- `join()` can take a timeout as argument, so it won't wait forever if the child gets into an infinite loop etc.

```java
for (int i = 0; i < num; i++) {
    child[i] = new ChildThread();
    child[i].start();
}
try {
    for (int i = 0; i < num; i++) {
        child[i].join();
    }
} catch (InterruptedException e) {
    // try to clean up the mess somehow
}
```

Defining your own Threads

- Two mechanisms – implements and extends
- We discuss both
- In general, `implements` is more flexible
- In general, `extends` is easier

Extending Thread

```java
public class MyThread extends Thread {
    public void run() {
        System.out.println("I am in another thread");
    }
}
```

Extending Thread (cntd.)

```java
public class Main {
    public static void main(String[] args) {
        Thread t = new MyThread();
        t.start();
    }
}
```

MyThread actually is-a thread, and can thus be used exactly as a thread.

But remember that you can only extend one other class, so it's not always possible.

`run` effectively tells the thread 'what to do'.
Implementing Runnable

Rather than extending thread, you can explicitly tell it what to do by giving it a Runnable

```java
public class MyThingToDo implements Runnable {
    public void run() {
        System.out.println("I am in another thread");
    }
}

public class Main {
    public static void main(String[] args) {
        Runnable r = new MyThingToDo();
        Thread t = new Thread(r);
        t.start();
    }
}
```

Common Mistakes

Building a thread, but never starting it
- Just creating a thread doesn't make it runnable, you need to start it first.

- Very common beginner's mistake
Common Mistakes

public class MyThread extends Thread {
    public void run() {
        System.out.println("I am in another thread");
    }
    public static void main(String []args) {
        Thread t = new MyThread();
        t.run();
    }
}

Common Mistakes

- Calling `run` instead of `start`
  - You can call `run` (it's just a method), but that will execute the method `run` in the same thread where the call happened
  - Calling code doesn't proceed until call to run returns.
  - You need to call `start` to make the JVM build a separate thread on which to execute `run`
  - Caller keeps on going in its original thread

Common Mistakes

- Thinking only code in Thread is executed in multithreaded situation
  - But, all the method calls from within `run` are also executing on the thread
  - therefore, they execute in a multithreaded environment

Common Mistakes

- Treating methods as atomic
  - You saw how threads can cut in and out whenever you like
  - In fact, even statements are not atomic
  - `i = i + 1` is broken into a series of instructions in the JVM, and the thread can pause after any of those instructions
Overview

Introduction
Thread Lifecycle
Data sharing
Locking
Deadlocks
Sophisticated synchronization

Class Design
Server architecture
Other languages

Interference

Interference is whenever the behaviour with interleaving is different from what one would expect
- Expected: what happens when there is no interleaving (i.e., when threads run completely, one after another)
- Interference is the result of shared variables changing due to activity in another thread which snuck in while one thread was going about its business
  - or said another way: interference results from an action that was assumed to be atomic not actually being atomic
  - Remember: the instance variables of objects (on the heap) are shared between the threads
  - so their values can be changed by another thread between uses by one thread
  - But each thread has its own copy for local variables of methods

Example I: Lost Update

Consider a method used to count the number of clients connected to a server

```java
class ConnectServer1 {
    private int myConnectionCount;
    public void addConnection() {
        int currentCount = myConnectionCount;
        currentCount++;
        myConnectionCount = currentCount;
    }
    // other methods
}
```

Example I (cntd)

Now two connections are added concurrently, each thread calling addConnection

*In the worst case, the two threads:*
- both read the same myConnectionCount
- both add one to the same value
- both store the same value
- myConnectionCount is only incremented by one!
Log of expected behaviour

heap: sc.myConnectionCount
stack of T1 stack of T2
T1: call s.addConnection() 5 ?
T1: read myConnectionCount (value: 5) 5 5
T1: set currentCount to 5 6
T1: set currentCount to 6 6
T1: read currentCount (value: 6) 6
T1: set myConnectionCount to 6 6
T1: return from s.addConnection() ?
T2: call s.addConnection() 7
T2: read myConnectionCount (value: 6) 6
T2: set currentCount to 6 7
T2: set currentCount to 7 7
T2: set myConnectionCount to 7 7
T2: return from s.addConnection() 7

No interleaving!

T1 then T2

Log of lost update

heap: sc.myConnectionCount
stack of T1 stack of T2
T1: call s.addConnection() 5 5
T1: read myConnectionCount (value: 5) 5 5
T1: set currentCount to 5 6
T1: set currentCount to 6 6
T1: read currentCount (value: 6) 6
T1: set myConnectionCount to 6 6
T2: call s.addConnection() 6 6
T2: read myConnectionCount (value: 5) 5 5
T2: set currentCount to 5 6
T2: set currentCount to 6 6
T2: read currentCount (value: 6) 6 6
T2: set myConnectionCount to 6 6
T2: return from s.addConnection() 6
T1: set myConnectionCount to 6 6
T1: return from s.addConnection() 6

T2 interleaves during execution of T1

What went wrong?

It takes time to execute all of addConnection
The step (set myConnectionCount to a value) makes sense only if the value in myConnectionCount hasn’t changed since the earlier step where the thread read its value
That is, for the addConnection algorithm to be correct it must prevent changes to the shared state from concurrent threads

So, couldn’t we just…

Change the body of the method to

myConnectionCount++;

That’s a single statement, which looks pretty atomic!
- But it’s not (JVM actually does: load to register, increment register, store register – the same three steps that we already had)
Example II – Lost Update with call on another object

class ConnectServer2 {
    private Account owner;
    public void payFee(int amt) {
        int i = owner.getBal();
        owner.setBal(i + amt);
    }
    // other methods
}

Example II (cntd)

Two threads, each executing payFee, can interfere
- Final state could be different from what would happen without interleaving
- The argument is the same as for the previous example, but the integer being corrupted is not in the class with those methods
  - It’s the balance instance variable which is inside owner

But can’t we just…

- rewrite payFee to be a one-liner?
  public void payFee(int amt) {
      owner.setBal(owner.getBal()+amt);
  }
  - No, another thread can interleave between any JVM instructions
  - This Java statement is done as many instructions one after the other
    - Call getBal()
    - // steps in getBal
    - Return from getBal with result in a register
    - Load amt
    - Add
    - Call setBal with sum as arg
    - // steps in setBal
    - Return from setBal

Example III

class ConnectServer3 {
    private Account owner;
    public void payFee(int amt) {
        int i = owner.getBal();
        owner.setBal(i + amt);
    }
    public void changeOwner(Account a) {
        owner = a;
    }
}
Example III (cntd)

What happens if changeOwner is interleaved half way through payFee?

Note: interference can be between concurrent calls of different methods, as well as between concurrent calls of a single method.

Example III (cntd)

Note that in this case there are different outcomes, either would be expected:
- the state after non-interleaved execution of payFee then changeOwner
- the state after non-interleaved execution of changeOwner then payFee

Interference is when the actual outcome is not either of these!

Example IV – inconsistent read

class ConnectServer4 {
  private int inwardConnectionCount;
  private int outwardConnectionCount;
  public void swapDirectionInToOut() {
    inwardConnectionCount--;
    outwardConnectionCount++;
  }
  public int totalConnections() {
    return (inwardConnectionCount + outwardConnectionCount);
  }
}

Outline Log of an expected behaviour

Initial state- inwardConnections: 30; outwardConnections: 6
- T1: call s.totalConnections()
- T1: read inwardConnections (value: 30)
- T1: read outwardConnections (value: 6)
- T1: return from s.totalConnections (return value=36)
- T2: call s.swapDirectionInToOut()
- T2: read inwardConnections (value: 30)
- T2: set inwardConnections to 29
- T2: read outwardConnections (value: 6)
- T2: set outwardConnections to 7
- T2: return from s.swapDirectionInToOut()

No interleaving: T1 then T2

Final state- inwardConnections: 29; outwardConnections: 7
another expected behaviour

Initial state- inwardConnections: 30; outwardConnections: 6

T2: call s.swapDirectionInToOut()
T2: read inwardConnections (value: 30)
T2: set inwardConnections to 29
T2: read outwardConnections (value: 6)
T2: set outwardConnections to 7
T2: return from s.swapDirectionInToOut()

No interleaving: T2 then T1
T1: call s.totalConnections()
T1: read inwardConnections (value: 30)
T1: read outwardConnections (value: 7)
T1: return from s.totalConnections (return value=36)

Final state- inwardConnections: 29; outwardConnections: 7

Outline Log of interference

Initial state- inwardConnections: 30; outwardConnections: 6

T1: call s.totalConnections()
T1: read inwardConnections (value: 30)
T2: call s.swapDirectionInToOut()
T2: read inwardConnections (value: 30)
T2: set inwardConnections to 29
T2: read outwardConnections (value: 6)
T2: set outwardConnections to 7
T2: return from s.swapDirectionInToOut()
T1: read outwardConnections (value: 7)
T1: return from s.totalConnections (return value=37)

Final state- inwardConnections: 29; outwardConnections: 7

Example IV (cntd)

In an interleaved execution, the return value in one call might be different from what is expected

- Here, return value of s.totalConnections() in interleaved log is different from its value in either of the logs without interleaving
- Thus, interference has happened

Identifying Potential Conflict

If a resource (primitive type, reference, reference type) is

- shared between two threads which both use the resource,
- and at least one of those threads alters the resource

then there may be the potential for interference

A key skill is spotting this in code you write or review.
Exercise

- Write out detailed logs for example III
  - Showing expected behaviour with payFee then changeOwner
  - Showing expected behaviour with changeOwner then payFee
  - Showing interference between these methods

Need for atomicity

- Recall that data interference was caused by interleaving of another thread within code that was expected to be atomic
- We want to prevent interference
  - So we should prevent interleaving!
- Java provides synchronized block

Overview

- Introduction
- Thread Lifecycle
- Data sharing
- Locking
- Deadlocks
- Sophisticated synchronization

What is a synchronized block?

Here `var` is an instance variable of type `MyClass`

```java
public int foo(String arg) {
    if (arg != null) {
        synchronized(var) {
            int i = var.getX();
            var.setX(i + arg.length());
        }
        System.out.println("done");
    }
}
```
Syntax

- A synchronized block must be a block
  - Properly nested within other blocks, such as loop body or method body
- The expression on which to synchronize must evaluate to an object
  - Can't synchronize on a primitive (e.g., int)

What does it do?

- Before a thread can enter into a synchronized block, it must acquire the lock (or monitor) of the object that governs the block
  - In our example, the object which is the current value of the variable var
- The thread keeps the lock until it leaves the end of the synchronized block
  - Then it frees (or releases) the lock
- At any time, a given lock can be held by only one thread

Warning

- The lock is held by a thread, on the object which is the value of the expression in the synchronize statement
  - The lock is NOT held by a method, nor is it necessarily held on the object in which the thread is executing
- The lock is kept even during calls of other methods which happen inside the synchronized block
  - these may be methods of other objects!

Log

```
class X {
   int foo(A a, B b) {
      int y = 17;
      synchronized(a) {
         y++;
         b.bar();
      }
      return y+1;
   }
   // other methods
   // X's instance variables
}
```

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>call anX.foo(myA, myB)</td>
</tr>
<tr>
<td></td>
<td>t1: assign y to 17</td>
</tr>
<tr>
<td></td>
<td>t1: get lock on myA</td>
</tr>
<tr>
<td></td>
<td>t1: read y (value 17)</td>
</tr>
<tr>
<td></td>
<td>t1: assign to y the value 18</td>
</tr>
<tr>
<td></td>
<td>t1: call myB.bar()</td>
</tr>
<tr>
<td></td>
<td>// steps in bar()</td>
</tr>
<tr>
<td></td>
<td>t1: return from bar</td>
</tr>
<tr>
<td></td>
<td>t1: release lock on myA</td>
</tr>
<tr>
<td></td>
<td>t1: read y (value 18)</td>
</tr>
<tr>
<td></td>
<td>t1: calculate 17+1</td>
</tr>
<tr>
<td></td>
<td>t1: return from foo (return value: 18)</td>
</tr>
</tbody>
</table>

t1 holds the lock on myA throughout the period marked
**Blocking on a lock**

- When a thread tries to acquire the lock but can’t have the lock (because another thread holds it):
  - the thread that wants the lock is no longer allowed to run (we say that it is blocked)
  - until the holder gives up the lock, and the blocked thread can acquire it

- When the blocked thread eventually does acquire the lock and then run:
  - it continues execution inside the synchronized block
  - any other blocked ones remain blocked

**Warning**

- A common mistake is to think that non-interleaving is only between threads executing the same synchronized block

  - But in fact, interleaving is prevented between all threads which are executing blocks whose synchronization condition evaluates to a given object instance!
  - These blocks need not even be inside methods of the same class

**Lack of control**

- When a lock is released, there is no way for the programmer to control which among blocked threads will get the lock next
  - indeed, the lock might be obtained by a thread which had not been blocked earlier, but was now trying to enter a synchronized block for the first time

**Alternative syntax**

- “synchronized” in method header
  - Effect is the same as having the whole method body in a block synchronized on this (the object in which the method runs)
    - but often runs faster
  - Public synchronized int foo(String arg1) {
      // body
  }
  - Public int foo(String arg1) {
      synchronized(this) {
        // body
      }
  }
Examples of Locking

public synchronized void addConnection() {
    int currentCount = myConnectionCount;
    currentCount++;
    myConnectionCount = currentCount;
}

public synchronized void removeConnection() {
    myConnectionCount--;
}

Log showing no lost update

T1: call s.addConnection()
   T1: get lock on sc [value of s in T1]; enter synchronized block
   T1: read myConnectionCount (value: 5)
   T1: set currentCount to 5
   T1: set currentCount to 6
   T1: read currentCount (value: 6)
   T2: call s.addConnection()
   T2: try to get lock on sc; blocked!

   T1: set myConnectionCount to 6
   T1: release lock on sc
   T2: get lock on sc, enter synchronized block
   T1: return from s.addConnection()
   T2: read myConnectionCount (value: 6)
   T2: set currentCount to 6
   T2: set currentCount to 7
   T2: read currentCount (value: 7)
   T2: set myConnectionCount to 7
   T2: release lock on sc
   T2: return from s.addConnection()

Another example outline log

T1 runs addConnection(), and concurrently T2 runs removeConnection()
   T1: call addConnection()
   T2: call removeConnection() – obtain lock on ConnectServer object
   T2: read myConnectionCount (value: 5)
   T1: try to enter synchronized block – lock on ConnectServer is already held by T2, so T1 becomes non-runnable (blocked)
   T2: set myConnectionCount to 4
   T2: leave synchronized block – release lock on ConnectServer object, T1 now runnable
   T1: obtains lock on ConnectServer object, and enters synchronized block
   T1: read myConnectionCount (value: 4)
   T1: set myConnectionCount to 5
   T1: leave synchronized block and release lock
   T1: return from s.addConnection() – stack of T1
   T2: return from s.removeConnection() – stack of T2
   Final state is just as expected when T1 runs completely then T2 runs completely

Don’t jump to conclusions

Correctness of our code means: every log has same final state and return values as one of the expected (non-interleaved) logs

We can’t be sure that every interleaving behaves as expected just because we checked two particular logs

- In this case our code is in fact correct
- In general, we need some thorough way to check
  - This is best done with mathematically-based Formal Methods such as model checking tools
Thinking about correctness

- The JVM makes sure that you never have concurrently two threads inside synchronized blocks which involve the same lock.
- The programmers make sure that every piece of code that deals with some shared variable is inside synchronized blocks involving the same lock.
- These combine to show: you never have concurrently two threads which deal with the shared variable.
  - That is, no interference is possible on this variable.

Dangerous talk!

- It is common to say “the access to the variable is protected by the lock” (or “by the synchronized block”).
- But protection depends not just on this one lock/block, but on programmers to keep every possible activity on the shared variables inside synchronized blocks on the same lock.
  - The JVM does not prevent interleaving by other threads that access the data, but don’t try to get the same lock.

Example with inadequate locking

```java
public void swapDirectionInToOut() {
    inwardConnectionCount--; outwardConnectionCount++;
}

public int synchronized totalConnections() {
    return (inwardConnectionCount + outwardConnectionCount);
}
```
Outline Log of interference

Initial state- inwardConnections: 30; outwardConnections: 6

T1: call s.totalConnections()
T1 enter synchronized block – get lock on object
T1: read inwardConnections (value: 30)
T2: call s.swapDirectionInToOut()
T2: read inwardConnections (value: 30)
T2: set inwardConnections to 29
T2: read outwardConnections (value: 6)
T2: set outwardConnections to 7
T2: return from s.swapDirectionInToOut()
T1: read outwardConnections (value: 7)
T1: leave synchronized block – release lock on object
T1: return from s.totalConnections (return value=37)

Final state- inwardConnections: 29; outwardConnections: 7

Interleaving is not prevented, because T2 doesn’t try to get the lock: T1’s execution is not atomic, even though it does lock properly!

Locking convention

It’s essential for all programmers who write code that deals with a variable to agree on what object they will use for synchronized blocks that protect that variable.

The common rule is: synchronize on the object in which the variable is an instance variable.

Example - Locking

To protect against other threads dealing with the balance which is in the Account object referenced by owner

```java
public synchronized void payFee(int amt) {
    synchronized(owner) { // other methods
        int i = owner.getBal();
        owner.setBal(i + amt);
    }
}
```

Why do we also lock the ConnectServer object?

Static

The `static` keyword means that a member or method belongs to an entire class, not separately to a single instance.
Example

```java
public class Car {
    private int myID;
    private static int numberCreated = 0;
    public Car() {
        myID = Car.numberCreated;
        numberCreated++;
    }
    public static int getNumberCreated() {
        return numberCreated;
    }
}
```

Example Continued

- `getNumberCreated()` can be called either on an instance or on the class
  - The effect is identical
  - Static methods cannot mention instance variables (unless they have an instance to manipulate)

Locking on Static

- Since a static method is not associated with any particular instance, a synchronized static method locks on the class itself
- Note, this does not exclude other code that is locking in instances of that class
Syntactic Sugar

public static synchronized void getNumberCreated() {
    ...
}
Is shorthand for

public static void getNumberCreated() {
    synchronized(Car.class) {
        ...
    }
}

Common Traps

- Forgetting that synchronized instance methods lock on an instance but not the class
- So even synchronized instance methods can interfere with class (static) data
  - You should explicitly lock the class around any code that deals with class data that some other method might alter or use

Lock classes

- New library classes in java.util.concurrent.locks (introduced in Java 1.5)
- Programmer can use explicit lock/unlock operations on lock objects, instead of synchronized blocks
  - Lock l = new ReentrantLock();
  - // somehow get l to all other threads which might conflict with this code
  - l.lock();
  - try {
  -     // access the resource protected by this lock
  -     ...
  - } finally { l.unlock(); }

- More flexible
  - Lock and unlock need not be in same method body
  - Holding multiple locks need not be nested properly
- More dangerous
  - Must have convention so all conflicting threads use same lock object
  - Programmer can forget to release lock

Overview

- Introduction
- Thread Lifecycle
- Data sharing
- Locking
- Deadlocks
- Sophisticated synchronization
- Class Design
- Server architecture
- Other languages
Java Locks revisited

In Java, each object is associated with its lock
- only one thread can hold the lock at a time

Synchronised block is supported at the language level
- Acquire the lock on entry to the synchronized block
  - NB thread will block if the lock is already held by another thread when it tries to enter the synchronized block
- Release the lock on completing the synchronized block

Example – Nested Locking

```java
public synchronized void payFee(int amt) {
    synchronized(owner) {
        int i = owner.getBal();
        owner.setBal(i + amt);
    }
    // other methods
}
```
Log with nested locking

Initially, suppose there is ConnectionServer cs, whose owner variable refers to Account object acct1, whose bal variable has value 100

1. call cs.payFee(10)
2. obtain lock on cs for synchronized method
3. enter synchronized block – obtain lock on acct1 (the value of owner)
4. call acct1.getBal()
5. read acct1. bal (value is 100)
6. return from getBal (return value is 100)
7. set local variable i to 100
8. Determine value of owner (acct1)
9. call acct1.setBal(110)
10. assign acct1.bal to 110
11. return from setBal
12. release lock on acct1 on leaving synchronized block
13. release lock on cs on leaving synchronized method
14. return from payFee

Hidden nested locking

Call another method which has a synchronized block, from inside a synchronized block
- Even indirectly

```java
Thread executing foo gets a lock on the instance of Foo
Thread executing Foo.foo calls VarType.bar1 which calls VarType.bar2
Thread executing bar2 gets a lock on the instance of VarType
```

Nested locks on several objects of the same class

- When a method operates on several objects, it may need to lock them all to prevent interference
- Each synchronized block locks one object only
- Need to nest the synchronized blocks

```java
class ConnectServer5 {
    ConnectServ primary;
    ConnectServ backup;
    public void check() {
        synchronized(primary) {
            if (primary.failed()) {
                synchronized(backup) {
                    primary.unUse(this);
                    backup.use(this);
                }
            }
        }
    }
}
```

Blocking while holding a lock

- In general, code that holds a resource like a lock should be written to run quickly
- So other threads are not delayed long when they are waiting for the resource
- Eg avoid I/O inside synchronized block
- When a thread tries to acquire a nested lock, it may not succeed
- Thus, the thread could be blocked while it holds a lock gained in an outer synchronize block
- This blocked situation could last an unpredictable (perhaps long) time
- This can damage performance
- But even worse....
Deadlock

An extreme case of performance problems is when a thread t1 blocks, trying to get a lock which will never be released.

- Because the thread t2 holding that lock is itself blocked trying to get a lock held by t1!

Non-determinism

- A system is incorrect if even one execution exists with deadlock
  - Even though other executions are OK
- Deadlock, like data interference, may be an intermittent phenomenon
  - Only seen with particular interleaving pattern
  - Often, deadlock appears at high load
    - Because that is when many threads try to execute concurrently
Other deadlock situations

- T1: lock x
- T2: lock y
- T3: lock z
- T1: try to lock y, blocked
- T2: try to lock z, blocked
- T3: try to lock x, blocked

None of the threads can proceed; all are blocked.

Ordering resources

- One way to write code that doesn’t deadlock is
  - Place some fixed agreed order on the resources
  - Make sure a thread never asks for a resource if it already holds a later-ordered resource
- Thus, always obtain nested locks in the same order
  - Lock bank accounts which need to be simultaneously locked, in order of acctId
  - Lock pages in a cache in order from position 1 to position N
  - Lock objects down a tree
    - lock root, then lock a child of root, then lock a child of child of root, etc.

  The "order" has to be intrinsic and unvarying (eg can’t use tree structure if that changes, can’t use value in any mutable field)

The wait-for graph

- Draw a graph whose nodes are threads
- Edge from tA to tB if tA is blocked because it tries for a lock held by tB
  - Meaning: tA is blocked till tB releases the lock
- Deadlock = a cycle in the graph

T1 waits for lock held by t2

Warning

- The rule does not say either of:
  - make sure that a thread never asks for a resource if it has previously held a later-ordered resource
  - make sure a thread holds all items earlier than x before it requests resource x
Recursive locking

What happens if a thread t1 tries to obtain a lock it already holds?

```java
synchronized(x) {
    synchronized(x) {
        // …
    }
}
```

Answer: t1 is allowed to proceed
- System counts "how often" t1 holds the lock
- So lock is actually available for another thread after t1 has left all synchronized blocks on this object

Importance of recursive locking

- If t1 blocked on a lock it held itself, you could get deadlock with only one thread running!
- If particular, it would not be safe to have synchronized methods call other synchronized methods
  - This would force nasty design choices: which methods to synchronize
- Recursive locking reduces probability of deadlock

Recursive locking -cntd

- Rarely see explicit code which nests synchronized blocks with same object!
- However, you can have nested synchronized blocks on different variables, whose value happens to be the same object
- Or, one synchronized method can call another synchronized method!

Overview

- Introduction
- Thread Lifecycle
- Data sharing
- Locking
- Deadlocks
- Sophisticated synchronization
- Class Design
- Server architecture
- Other languages
Locking review

- Locking can be used to avoid data interference
- A lock can’t be held by two threads at once
  - So code inside a synchronized block is not interleaved with other code inside a synchronized block locking on the same object
- The effect is: there is no interaction between the threads
  - Outcome as if they didn’t run concurrently

Closer interaction

- Sometimes interaction between the threads is desirable (in a controlled way)
- Eg we want t1 to do X and then t2 to do Y and then t1 to do Z
  - Information flows between the threads
  - Execution should not be as if one ran completely then the other ran!
- Java provides wait(), notify() and notifyAll() mechanisms

The main idea

- A thread performs wait(), and then it won’t take more steps until another thread calls notify() or notifyAll()
- There are many complex details

The system state

- JVM keeps, for each object
  - Lock-Holder: Zero or one threads that hold the lock on that object
  - Lock-seeking queue: Zero or more threads that are blocked because they want to acquire the lock, but are prevented (due to another holder)
  - Wait-queue: Zero or more threads that are blocked because they called wait and no-one has since called notify() or notifyAll()
wait()

wait() is a method of every object
- Inherited from Object class
You must not perform obj.wait() except when the current thread holds the lock on obj
- Otherwise get runtime exception
- Usually, have obj.wait() directly inside synchronized(obj) block
- Syntactic sugar: have wait() inside synchronized method [using this as obj]

Effect of wait()

When t1 performs obj.wait()
- t1 releases the lock on obj
  - This may cause another thread (which was trying to acquire this lock) to be able to run
- t1 becomes blocked in wait-queue of obj

notifyAll()

notifyAll() is a method of every object
- Inherited from Object class
You must not perform obj.notifyAll() except when the current thread holds the lock on obj
- Otherwise get runtime exception
- Usually, have obj.notifyAll() directly inside synchronized(obj) block
- Syntactic sugar: have notifyAll() inside synchronized method [using this as obj]

Effect of notifyAll()

When t1 performs obj.notifyAll()
- All threads in the wait-queue of obj are moved to the lock-seeking queue
- Usually, soon after, t1 will release the lock on obj
  - Then, one lock-seeking thread will get lock and run
  - The first thread that gets the lock might not be one that had been waiting
**notify()**

- **notify()** is just like **notifyAll()** except that only one waiting thread is moved to lock-seeking queue
  - But no-one can control which waiting thread is affected
- **notify** is rarely useful unless you can be certain that only one thread is waiting

**Timeout**

- There is an overloaded method **wait(int millis)**
  - This moves the caller thread to the wait-queue and relinquishes the lock, but, if the thread has not been woken after millis time has passed, the system moves the thread automatically to the lock-seeking set

**Warnings**

- **wait(), notifyAll(), notify()** are methods of the object which is locked
  - They are not methods of the thread
- In designing code, always consider the possibility that **t1** does **notifyAll()** before **t2** does **wait()**!
  - The “lost signal” problem
**Threading in Java**

**Warnings**
- Other threads can run between `notifyAll()` and the woken-up thread that had been waiting
  - They may alter whatever condition the notifier had set up
  - Waiting thread should always wait inside a loop that is checking the condition

**The Idiom**
- To wait till a condition is true
  - In a synchronized method or block
    ```java
    while (!condition) {
      try {
        wait();
      } catch (InterruptedException e) {} 
    }
    ```
  - In every thread that could make condition become true
    ```java
    // make condition true
    notifyAll();
    ```

**Deadlock with wait()**
- It is much harder to analyse deadlock possibilities involving `wait()` and `notifyAll()` than just with synchronized blocks
  - A thread which is waiting may resume due to any one of a number of other threads
  - After resuming, a thread may wait again
    - This can lead to lockout rather than deadlock
  - Resource ordering does not protect you!

**An extended example**
- Message-box coordination between producer and consumer
Message-box coordination

- A classic example where mutual exclusion is not enough
- A source has a sequence of messages
  - It puts one message in a shared place
  - Then the next message, and then the next...
- A destination obtains the messages from the shared place
  - One after another
- Catch: make sure they stay in synch
  - Source never overwrites message that destination hasn’t grabbed yet
  - Destination never takes same message twice
    - We deal with this problem first!

Initial Attempt

class MessageBox {
    public boolean available;
    public String message;
}

A source can leave a message by doing:

myMessageBox.message = “Hello!”;
myMessageBox.available = true;

Checking a message box

The destination thread loops forever, checking the availability of the message every 50 msec.

There might be data interference at the block:

```
if (myBox.available) {
    System.out.println(myBox.message);
    myBox.available = false;
}try {
    sleep(50);
} catch (InterruptedException e) {}}
```
Preventing data interference

- By using a synchronized method or block, the data interference can be prevented.
- In order to make `MessageBox` class thread safe
  - make instance variable `private`
  - make each access be inside a synchronized method or block

```java
class MessageBox {
    private boolean available;
    private String message;

    public synchronized void setMessage(String message)
    {
        this.message = message;
        this.available = true;
    }

    public synchronized String getMessage() {
        if (!this.available)
            return null;
        this.available = false;
        return this.message;
    }
}
```

Modified code of destination

```java
public void run() {
    while (true) {
        String m = myBox.getMessage();
        if (m != null) {
            System.out.println(m);
        } else {
            try {
                sleep(50);
            } catch (InterruptedException e) {}
        }
    }
}
```

Drawback of this design

- This is thread-safe
- No interference
- Destination never prints same message twice
- But destination thread must “poll” or check regularly to see if there is a message
- We don’t know whether the 50 msec time delay used is appropriate or not.
- It might be too long; slow response to the client
- It might be too frequent; wasting CPU time.
Who is in charge?

- Instead of the destination checking the MessageBox every 50 mil. sec.
  - the destination will block till the message is available
  - The MessageBox takes the responsibility to return the message only when it is available

```java
public synchronized String getMessage() {
    while (!this.available) {
        try {
            wait();
        } catch (InterruptedException e) {} 
    }
    this.available = false;
    return this.message;
}
```

Modification to `MessageBox`

```java
Modification to destination thread

```
```java
public void run() {
    while (true) {
        String m = myBox.getMessage();
        System.out.println(m);
    }
}
```

Discussion

- `MessageBox` class now manages its own instance variables
- It now checks the availability of the message.
- However, once destination thread goes into the `wait` state during call to `getMessage()`, it stays there
  - Some thread needs to wake it up.
Additional Modification to MessageBox

```java
public synchronized void setMessage(String message) {
    this.message = message;
    this.available = true;
    notifyAll();
}
```

Preventing over-run

- Code so far prevents destination reading when there is no new message
- Also, we should prevent source from overwriting a message that destination hasn’t seen yet
  - This is symmetrical problem

```
public synchronized void setMessage(String message) {
    while (this.available) {
        try {
            wait();
        } catch (InterruptedException e) {}  
    }
    this.message = message;
    this.available = true;
    notifyAll();
}
```
```
public synchronized String getMessage() {
    while (! this.available) {
        try {
            wait();
        } catch (InterruptedException e) {} 
    }
    this.available = false;
    notifyAll();
    return this.message;
}
```

Things to notice (1)
- We invoke `notifyAll()` rather than `notify()`.
  - `notify()` will move only one thread from `waiting` status to `seeking lock` state.
- With `notify()`, you would not know which one will be woken up: the selection of a thread is totally up to the implementation of a thread scheduler.
- Also, some interesting bugs can arise if we use `notify()` and there are several consumers and several producers.

Things to notice (2)
- We use `while` loop to check the change of a condition rather than `if` statement.
  - The thread re-checks the condition after re-obtaining the lock (being woken up).
  - We can be certain that once we leave the `while` loop, the condition is true.
  - Provided that all threads avoid data interference by proper use of synchronized block.

Overview
- Introduction
- Thread Lifecycle
- Data sharing
- Locking
- Deadlocks
- Sophisticated synchronization
- Class Design
- Server architecture
- Other languages
Designing classes

All OO Software design requires choices (not discussed here)
- What classes
- What methods
- What associations between classes

Our focus: Extra decisions in a multithreaded context
- What locking to do

Key questions

When to lock?
- find code which must be protected against interleaving
- maybe place block around even more code
  - Declare “synchronized” (ie block is whole method body)
  - Or even place synchronized block in a method of another class, which surrounds the method with the critical section

What to lock?
- decide on the object to use for synchronized block

Documentation

It’s important to describe the way multiple threads might behave, in the documentation for a class
- Put key facts in Javadoc comments
- Remember that “synchronized” keyword is not part of the API

Thread-Safety

A class is “thread-safe” if all methods contain sufficient synchronization that interference can’t happen even if multiple threads call an instance’s methods concurrently (with no synchronization outside the class)
Usual Approach to Thread-Safety

- Identify instance variables which could suffer interference
  - Make them all private
  - Thus any thread that accesses them, does so inside a method of the class
- Identify all methods that deal with these variables
  - Make them all synchronized

Identification of Critical Section

- You may choose to place a synchronized block only around the critical section (the region which needs to appear atomic)
  - Some activity in the method can be left outside the critical section, because it isn’t vulnerable to interference from interleaving
  - Sometimes re-arrange code to move an activity to the front or back of the method body to make this possible
    - Particularly if activity is slow (e.g., I/O)

Identifying the Critical Section

- Look for an instance variable which is accessed in several threads (and modified in one or more of them)
  - Eg variable is read, and another thread might write
  - Eg variable is written and another thread might read or write

Extent of the Critical Section

- Make sure the critical section includes any accesses that are related to each other, and should not have other accesses interleave between them
  - Look for an instance variable that is accessed several times in the thread
  - Look for several instance variables which might have some consistency requirement which is temporarily violated (between two modifications)
Warning

- Remember that a single statement may be done in several steps
  - Eg \( y = x \) is (read \( x \); write \( y \))
  - Eg \( x++ \) is (read \( x \); write increased value)
  - Eg \( \text{long } y = 0 \) is (write low word; write high word)
- This may need to be inside critical section to prevent interleaving

Immutable types

- Often one can redesign the underlying class so that the value of instance variables is never changed
  - We say that the class is immutable
  - The class then won’t need locking
- Methods don’t change the state, instead they construct and return new objects

Mutable vs Immutable

```java
StringBuffer sb;
StringBuffer sb1;
StringBuffer sb2;
// initialize sb to “foo”
sb1 = sb;
// sb and sb1 refer to
// the same instance
sb.append(“fred”);
// sb is now “foofred”
// sb1 is also “foofred”
append() modifies sb
```

```java
String s;
String s1;
String s2;
// initialize s to “foo”
s1 = s;
// s and s1 refer to
// the same instance
s2 = s.concat(“fred”);
// s is still “foo”
// s1 is still “foo”
// s2 is “foofred”
concat() does not modify s
```

Trade-offs

- Immutable class doesn’t need locking
- But, constructing new instances is often much more expensive than changing state
- Also, variables whose type is immutable are usually re-assigned often
- Leading to locking in the class where that variable is!
Adapters

How can you deal with a mutable class that is not thread-safe but which you can’t modify

- Eg many library classes (LinkedList, HashMap, JFrame, …)

Solution: Write a new class with the same interface, which encloses a reference to the non-thread-safe class, and delegates each operation to it from inside a synchronized block

Adapter Example

class SynchLL implements List {
  private List delegate;

  // other methods

  public void add(Object o) {
    synchronized(delegate) {
      delegate.add(o);
    }
  }
}

Warning: if any thread calls a method of delegate directly, rather than via SynchLL, then interference can happen

Trade-offs

Synchronization adds a lot to run-time

- Unsynchronized class is more efficient in single-threaded contexts

It’s easy to make a synchronized version from an unsynchronized one

- There’s no way to do the reverse!

Thus library often provides class that is unsynchronized, and let’s developers add synchronization when they need it

- Lesson learnt from SDK1.0 and 1.1

What to lock?

Recall that a synchronized block does not actually prevent interleaving

- Rather it prevents another thread entering a synchronized block on the same object

Thus thread-safety requires convention

- Programmers agree: what object they will synchronize on, around code that deals with a particular shared variable

- Usually: the object instance in which the variable is declared

- That is, all the instance variables in one instance are protected by locks on one object
Coarse-grained locking

- An alternative convention is to protect all accesses to any of the variables in several different objects by locks on a single object \( x \).
  - We say that the lock on \( x \) covers all the variables.
- Most often: all variables in any element of a data structure or collection, are protected by a lock on the head node or on the collection.
- Eg: a ConnectServer \( cs \) may have many connections, but we just lock \( cs \) before accessing any variable in any of its connections.

Trade-offs

- Interleaving will be prevented for methods that access different shared variables which are covered by the same lock.
  - Coarse-grained locking reduces possible concurrency.
    - This may reduce throughput.
- However, a thread might need to access many variables covered by the same lock.
  - Coarse-grained locking may reduce overheads.
    - This may increase throughput.
  - Coarse-grained locking may mean that each thread only needs one lock at a time.
    - This would eliminate any risk of deadlock.

Very-fine-grained locking

- An alternative convention assigns a different object to protect each shared variable.
  - Often construct an object specifically to be locked.
    - Eg provide balanceLock to protect balance, and addressLock to protect address.
  - This allows much concurrency, but may have high overheads.

Warning

- Whatever convention is chosen, it must be followed by all code that accesses the variable concerned.
  - A single method written with a different convention would risk data interference.
Threading in Java

**Threading and GUIs**
- The main AWT/Swing event-handling thread may start other background threads which perform lengthy processing.
- Several threads may concurrently access parts of the domain model.
  - Domain model needs to be thread-safe.
- Several threads may concurrently want to update the display.
  - AWT components are thread-safe.
  - Unfortunately most Swing ones are not! So other threads should use "invokeLater" to have GUI update done on main event-handling thread.

**Overview**
- Introduction
- Thread Lifecycle
- Data sharing
- Locking
- Deadlocks
- Sophisticated synchronization

**Client-Server programming**
- examples:
  - Web (HTTP)
  - Database
  - File (FTP/NSF)
  - mail (POP)

**Client-Server architecture**
- a client: initiates the connection
- a server: responds to the connection
- a socket: part of the code on client or server that represents the connection
  - It binds the client and server together connection.
**ServerSocket**
- part of the server code
  - it can exist without any particular client
- binds to a particular port on the server machine
  - listens for incoming connection requests
- when a request is accepted, it creates a “socket” which actually binds the server and client.

**ServerSocket – cont’d**
- Only one ServerSocket can listen to a particular port at one time.
- Many requests will queue up at the ServerSocket/port
  - usually this queue is created and managed by the operating system.
  - If the queue fills up, other requests are refused.

**java.net.ServerSocket**
- It is constructed on a particular port.
- It invokes `accept()` to listen connections.
- The `accept()` blocks until a connection is available (some requests are in the queue).

**Constructing java.net.ServerSocket**
- It is constructed on a particular port.
- try {
  ServerSocket ss = new ServerSocket(8080);
} catch (IOException ioe) {
  ioe.printStackTrace();
}
- No more than one process or thread (including non-Java ones) can listen to a particular port at one time.
- `java.net.BindException` (subclass of `IOException`) will be thrown, if there is already a server running on the particular port.
Finding a free port for dynamic port

If you specify "0" as a port number, Java will pick an available port. The picked port number can be retrieved by invoking the `getLocalPort()` method.

```java
try {
    ServerSocket ss = new ServerSocket(0);
    int dedicatedPort = ss.getLocalPort();
    // pass "dedicatedPort" to the client
    // for the dedicated communication.
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

This is useful when:
- a server and client need low-traffic (dedicated) channel,
- a server is inside a firewall.

Simple Example

```java
import java.io.*;
import java.net.*;
public class SimpleServer {
    public static void main(String[] args) {
        int port = 2048;
        try {
            ServerSocket ss = new ServerSocket(port);
            while (true) {
                try {
                    Socket s = ss.accept();
                    OutputStream os = s.getOutputStream();
                    InputStream is = s.getInputStream();
                    while (true) {
                        int ch = is.read();
                        if (ch == -1) {
                            break;
                        }
                        os.write(ch);
                        os.flush();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Single background thread

- `SimpleServer` can be executed from a separate thread running as a background process.
- Requested tasks will be stored in a task queue and the background thread will process each task at a time.
- This would not be a problem as long as scheduling is not important and each task is very short.
- Used in AWT and Swing's event thread.
Threading in Java

The problem 2

- SimpleServer processes one request at a time.
- This would acceptable only a **very brief** interaction with each client is required.
  - Otherwise many clients will face long delays while previous requests are processed.

The incoming connection queue

- When a ServerSocket is created, the underlying operating system creates a queue to store incoming connections (requests).
- The default size of the queue is 50. If your server is slow processing the requests, it can be expanded:

  ```java
  try {
    ServerSocket ss = new ServerSocket(8080, 60);
  } catch (IOException ioe) {
    ioe.printStackTrace();
  }
  ```

- The Operating System has a maximum queue length. If you ask for more than the OS can provide, the size of queue will be determined by the OS.

Server & Multithreading

- A server may need to communicate with many clients at once.
- The server needs multiple threads to concurrently handle requests.
  - You cannot do much by constructing ServerSocket with a large queue.

Thread-per-request

- A separate thread is created for each Socket object.
- The thread that has finished processing its Socket object dies.
  - Works well with a **small** number of **long-running** tasks.
Answer to the problem 2

```java
... public class SimpleServer extends Thread {
    Socket mySocket;
    public SimpleServer(Socket socket) {
        this.mySocket = socket;
    }
    public static void main(String[] args) {
        int port = 2048;
        try {
            ServerSocket ss = new ServerSocket(port);
            while (true) {
                try {
                    Socket s = ss.accept();
                    SimpleServer server = new SimpleServer(s);
                    server.start();
                } catch (IOException e) {}
            }
        } catch (IOException e) {)
    }
    public void run() {
        try {
            OutputStream os = mySocket.getOutputStream();
            InputStream is = mySocket.getInputStream();
            while (true) {
                int ch = is.read();
                if (ch == -1) {
                    break;
                }
                os.write(ch);
                os.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    }
Answer to the problem 2 – cont’d

```
Problems of thread-per-request

(2)

- Wasting system resources
  - Too many threads may result in JVM causing the system to run out of memory (java.lang.OutOfMemoryError).
  - Or it may cause too much disk-memory swapping, which leads to a significant performance drop.

A Thread Pool for SimpleServer

- Create a java.net.ServerSocket object
- Create a pool of threads.
- pass the ServerSocket object to each thread when it is constructed.
- Start each thread immediately after the construction.
- Each thread invokes accept() method of the ServerSocket to obtain a java.net.Socket object.
- Each thread processes the Socket object.

A Thread Pool for SimpleServer (cntd)

... ServerSocket socket; public SimpleServer(ServerSocket socket) { this.socket = socket; } public static void main(String[] args) { int port = 2048; try { ServerSocket ss = new ServerSocket(port); for (int i = 0; i < sNumOfThreads; i++) { SimpleServer server = new SimpleServer(ss); Server.start(); } } catch (IOException e) { e.printStackTrace(); } } ...

A Thread Pool for SimpleServer (cntd)

... public void run() { while (true) { try { Socket s = this.socket.accept(); OutputStream os = s.getOutputStream(); InputStream is = s.getInputStream(); while (true) { int ch = is.read(); if (ch == -1) break; os.write(ch); os.flush(); } } catch (IOException e) { e.printStackTrace(); } } catch (IOException e) { e.printStackTrace(); } }
**Threading in Java**

### A Thread Pool for SimpleServer (cntd)

[Diagram showing the interaction between servers, sockets, threads, and accept() method.]

### Thread Pool for non Client-Server applications

- Extend the client-server architecture to more generic multi-task processing architecture.
- Tasks can come from either outside or inside.
- Tasks may not be passed via Socket.

### Tracing Thread Pool Execution (1)

1. Create a pool of threads, each of which is started immediately after its creation.

   ```java
   public SimpleThreadPool(int numOfThreads) {
       this.threads = Collections.synchronizedList(new ArrayList());
       for (int i = 0; i < numOfThreads; i++) {
           Thread th = new ThreadWorker(this.threads); th.start();
       }
   }
   ```

### Tracing Thread Pool Execution (2)

- When a task (Runnable) arrives, pick (remove) an available (non-working) thread from the pool.

   ```java
   public void executeJob(Runnable r) {
       try {
           ThreadWorker th;
           synchronized (this.threads) {
               th = (ThreadWorker) this.threads.get(0);
               this.threads.remove(0);
           }
           th.execute(r);
       } catch (IndexOutOfBoundsException e) {
           e.printStackTrace();
       }
   }
   ```
Let the picked thread execute the task.

Put the thread back in the pool when it finishes its job.

Generic Thread Pool

A Pool of threads is not enough.
- What will happen if no threads are available?

A queue of tasks would be very useful.
- in the case of Socket-based programming, the queue is provided

Thread Queue

Generic Thread Pool

import java.util.*;

public class ThreadPool {
    private WorkQueue runnableTasks = new WorkQueue();
    public ThreadPool(int threadCount) {
        for (int i = 0; i < threadCount; i++) {
            ThreadWorker thread = new ThreadWorker(this);
            thread.start();
        }
    }
    /**
     * Add a task to the queue. An available thread will execute it.
     */
    public void run(Runnable task) {
        this.runnableTasks.put(task);
    }
    public Runnable getNextRunnable() {
        return (Runnable) this.runnableTasks.get();
    }
    class WorkQueue {
        private List list = new ArrayList();
        /**
         * Add object to the queue and notify threads waiting in get() for available jobs.
         */
        synchronized void put(Object object) {
            this.list.add(object);
            notify();
        }
        /**
         * Suspend threads in a wait until there is a job available.
         */
        synchronized Object get() {
            while (this.list.isEmpty()) {
                try {
                    wait();
                } catch (InterruptedException ex) {
                }
            }
            return this.list.remove(0);
        }
    }
    class ThreadWorker extends Thread {
        private ThreadPool pool;
        ThreadWorker(ThreadPool pool) {
            this.pool = pool;
        }
        public void run() {
            while (true) {
                Runnable task = this.pool.getNextRunnable();
                try {
                    task.run();
                } catch (Exception ex) {
                }
            }
        }
    }
}
Problems of using thread pool

An application using a thread pool is still at the risk of:
- synchronization errors
- deadlock
- pool-related deadlock
- resource thrashing
- thread leakage.

Thread Pool related deadlock

Tasks in a queue interact with one another.

- Some tasks are already in the process of execution by threads in the pool.
- those executing threads are blocked waiting the results from some other task.
- The other task is still in the queue, but no thread is available in the pool.

Resource Thrashing

If the pool size is not properly tuned,
- waste of memory
  - memory for Thread object
  - execution call stacks
- waste of other system resources
  - Java threads are associated with the native threads, which consume the systems resources
  - context switching overhead becomes significant

Other concurrency errors

incorrect execution of wait()/notify() methods,
- loss of notification may lead to:
  - threads remain in a wait state.

request overload
- The number of requests is simply larger than the size of the queue or thread pool.
- The system design has to specify how to handle this case:
  - put a limit on maximum request.
  - Or, dynamically resizable queue or pool
Other concurrency errors

- thread leakage
  - occurs if a thread is not returned to the pool after finishing its job.
  - if a thread does not catch RuntimeException or Error, it exits
    - no longer available in the pool.
  - A thread that permanently stalls will not be available in the pool.

Overview

- Introduction
- Thread Lifecycle
- Data sharing
- Locking
- Deadlocks
- Sophisticated synchronization

POSIX

- POSIX defines a standard for the system-call interface to operating systems
  - covers syntax
    - ie names/arguments/return types for functions
  - and covers semantics
    - ie behaviour of functions
  - almost all varieties of UNIX comply
**Pthreads**

- Pthreads is the part of POSIX dealing with threads
  - available to C programmers in a library `pthread.h`
  - third-party libraries provide the same functionality on Windows

**Creating and running a thread**

- Type: `pthread_t`
- `pthread_create(&tid, &tattr, start_func, arg)`
  - create a new thread
    - `tid` is used by caller to refer to the new thread
    - new thread can find its own id by `pthread_self`
  - created thread immediately begins by executing `start_func` on argument `arg`
  - contrast with Java where `t = new Thread()` happens in separate statement from `t.start()`
    - also in Java `run()` has no argument

**Termination of threads**

- A thread t terminates when its `start_func` is finished
  - or when another thread calls `pthread_cancel(t)`
- Its memory and other resources are not necessarily freed
  - unless another thread calls `pthread_join(t)`
    - which blocks till t finishes then "reaps" its resources
  - or a thread calls `pthread_detach(t)`
    - which makes t "detached" so system will reclaim its resources

**Shared variables**

- Global variables can be accessed in any thread
- Local automatic variables are stored separately for each thread in its own stack
  - but these stacks are in virtual memory and so can be accessed by another thread if it has a pointer to that location
- Local static variables are shared between threads running that function
Concurrent execution

- Threads which are executing can interleave
  - each machine instruction is indivisible
  - a C statement may not be indivisible
  - just as in Java, x++ is really done as separate instructions
    - load x into register
    - increment register
    - store register into x
- This can lead to interference (lost update etc)

Coordination between threads

- Pthreads provides defined types for coordination
  - pthread_mutex_t (lock)
  - pthread_cond_t (condition variable)
  - sem_t (semaphore)
- These are very similar to the internal mechanisms used in Java
  - but they are available to programmers

Using a Mutex

- pthread_mutex_lock(&m)
- pthread_mutex_unlock(&m)
- These are called explicitly to acquire and release a lock
  - compare to Java where it happens implicitly at start and end of synchronized block
- You must initialize the mutex before use
  - pthread_mutex_init(&m, &mattr)

Preventing interference

- If you have code segments that might interfere with one another, and so should be prevented from interleaving
  - define a mutex (static or global) to use as protection for these segments
  - have each code segment begin by locking the mutex
  - each code segment unlocks the mutex at its end
  - danger: don’t forget to have exactly one unlock per lock
- Warning: its up to programmers to remember the association between the mutex and the code
  - compare to Java where you usually lock the object in which is located the variables at risk
**Condition variables**

- `pthread_cond_wait(&cond, &mutex)`
  - Wait on queue associated with `cond` and release mutex (which thread must hold when calling this function)
  - When thread resumes, it will re-acquire mutex
  - Like Java `wait()` except that the condition variable is not associated with the object
  - Warning: danger if you mix up the link between condition variable and the mutex that protects code using it

- `pthread_cond_broadcast(&cond); pthread_cond_signal(&cond)`
  - Like Java `notifAll()` and `notify()` except that thread need not hold mutex when this function is called

**Semaphore**

- Very widely used in pthreads code
- Can provide both non-interference and coordination
- Initialize with `sem_init(&s, 0, N)`
  - `N` is the initial value
- `sem_post(&s)`
  - Increase `s` by 1
- `sem_wait(&s)`
  - Wait till `s > 0`, then decrease `s` by 1

**Semaphore for mutual exclusion**

- Mutex is just like semaphore with initial value 1
  - Current value is “how many more threads can acquire the mutex”
- Lock is just `sem_wait`
- Unlock is just `sem_post`

**Threading in C#**

- C# is Microsoft’s clean OO language with C syntax
  - Very similar to Java in style
  - Fixes some errors of early versions of Java
  - Eg all types can be treated as Objects
  - Likely to become widespread in Windows environments

- Thread features from `System.Threading`
Creating and running a thread

- Explicitly name the method that is to be run in the thread
  - it must have no argument or result
- Thread t = new Thread (new ThreadStart(MethodName));
  t.Start();

Preventing interference

- lock block
- lock(obj) {
  code
}
- just like Java synchronized block

Explicit access to lock

- In C#, the programmer can also explicitly acquire and release the lock
- Monitor.Enter(obj)
- Monitor.Exit(obj)
- Monitor.TryEnter(obj)
  - if it can't get the lock, it returns "false" instead of blocking until the lock is available
- Danger if these are used incorrectly

Interlocked

- Simple operations on a variable can be protected against interleaving
- Interlocked.Increment( ref x )
  - does x = x+1 indivisibly
- similarly Interlocked.Decrement( ref x )
- Interlocked.Exchange( ref x, ref y )
  - does {tmp=x; x=y; y=tmp;} indivisibly
- On multiprocessor, supported by hardware
Condition variables

- `Monitor.Wait(obj)`
  - like Java `obj.wait()`
- `Monitor.PulseAll(obj)`
  - like Java `obj.notifyAll()`

Read/write locking

- `ReaderWriterLock`
  - allows several concurrent readers
  - or one writer and no readers
- methods `AcquireReaderLock`, `AcquireWriterLock`, `UpgradeToWriterLock`, `ReleaseLock`
- a similar class is in Java 1.5 library: `java.util.concurrent.locks.ReentrantReadWriteLock`

Further reading

- "Computer Systems: A Programmers Perspective" by Bryant and Halloran
- "Multithreading with C#" by R. Krikorian on www.oreillynet.com