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-determination"
    - 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 (4th ed)”

Acknowledgement

- Based on materials developed with funding from “Building the Web Workforce” Science Lectureship, from Australian Government
  - Authored by Alan Fekete, Joe Thurbon and Masahiro Takatsuka

Further reading references

- The best reference: B. Goetz "Java Concurrency in Practice"
- Brian Goetz articles
  - At http://www.briangoetz.com
- Lesson on threads in Sun Java tutorial
- Documentation on java.util.concurrent
- A. Holub "Taming Java Threads"
  - www.holub.com has presentation etc
- D. Lea "Concurrent Programming in Java (2nd ed)"

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 ("fields") 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)

- Variables!
  - If `x`, `y` declared in other methods, those are different
  - They are not accessible in other methods

- It would keep its value if another call were made
  - It persists after the call `bar(5)` terminates

- This may call other methods, perhaps of other objects
- It executes method `run()`
  - Its steps can be interleaved with another thread
  - It can be treated as if atomic (all the code from call to return can be considered a step)
  - It gets its own bit of memory (local variables on the stack)
    - They live till the method call returns
  - It shares the heap with other threads
    - It can access the global memory (via instance variables, which are on the heap)

In that example...

- The instance variable ‘myVar’ is stored on the heap (as part of the instance object referred 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 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
  - It gets its own piece of memory on stack
    - For its local variables
  - Shares the heap with other threads
    - All threads use the objects and their instance variables

- When the thread starts running
  - It executes method `run()`
  - This may call other methods, perhaps of other objects

Multithreaded execution example

- A thread executes some steps
  - Then the JVM swaps to another thread
    - Which executes several steps
  - Then swap to another thread
    - No predictable order for threads to get turn
    - 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

- 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
  - It gets its own piece of memory on stack
    - For its local variables
  - Shares the heap with other threads
    - All threads use the objects and their instance variables

- When the thread starts running
  - It executes method `run()`
  - This may call other methods, perhaps of other objects
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!

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 turn 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 `t.join()`
  - It’s 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");
    }
}
```

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

Extending Thread (cntd.)

- 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("In another thread");
        }
        public static void main(String [] args) {
            Thread t = new MyThread();
            t.run();
        }
    }

- 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.

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.

Treating methods as atomic

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

Class Design and Threads

- The code you write in a class can be executing in many threads at once.
  - Even if the class itself never creates threads, or does any thread-related things.
- Threading is a potential issue whenever a class is made available for other code to use.
  - Eg in a library
  - Eg in an application that is deployed in an app server.

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 (ie when threads run completely, one after another)
- Interference is the result of shared variables changing due to activity in another thread which sneaked 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 fields of objects (on the heap) are shared between the threads.
  - 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 ConnectServer {
  private int myConnectionCount;
  public void addConnection() {
    int currentCount = myConnectionCount;
    currentCount++;
    myConnectionCount = currentCount;
  }
  // other methods
}
```

Example I (continued)

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

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

No interleaving!
T1 then T2

Log of lost update

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

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: inconsistent read

class ConnectServerA {
    private int inwardConnectionCount;
    private int outwardConnectionCount;
    public void swapDirectionInToOut() {
        inwardConnectionCount--;
        outwardConnectionCount++;
    }
    public int totalConnections() {
        return (inwardConnectionCount
                + outwardConnectionCount);
    }
    // other methods
}

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()

Outline Log of interference

Initial state- inwardConnections: 30; outwardConnections: 6
- T1: call s.totalConnections()
- T1: read inwardConnections (value: 30)
- T2: call s.swapDirectionInToOut()    No interleaving: T1 then T2
- 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)

Example II (continued)
- 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

Common mistakes
- Thinking that interference only happens between threads that are executing the same code fragment
  - Example II shows interference between threads that are executing different methods
Common mistakes

Thinking that interference only happens between threads that are modifying the same variable
- Example II shows interference between threads where one modifies a variable, and the other reads the variable

Thinking that interference has happened whenever an interleaved execution gives different state or return value than what is given by a particular non-interleaved execution
- Some cases there are several possible non-interleaved executions, with different outcomes
- Interference is when an interleaved execution gives a result that couldn't happen in any non-interleaved execution

Identifying Interference

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
- Unless appropriate synchronization is done (see next lecture)
- A key skill is spotting this in code you write or review

Class Design and Interference

Encapsulate all access to instance variables
- Make the instance variables private
- Then any thread that is dealing with an instance variable must be executing a method of this class
- So we can consider all pairs of methods for interference potential

Class Design and Interference

Do not allow a reference to this to escape from a constructor
- This mistake often happens via the constructor starting a new thread, which shares the reference to this
- Then we don't have to consider interference between the constructor and other methods (or other threads running the constructor itself)

No risk of interference I

We don't have to worry about interference on data that is confined to one thread
- Eg in an object which is constructed within a thread and never passed to any other thread
No risk of interference II

- We don’t have to worry about interference on unchanging data
  - Eg an instance variable whose value is never modified after being initialized
  - Java keyword `final`

Subtleties

- The JVM on multiprocessor machines can give behaviour that can’t be understood through the interleaving model with shared heap
  - Due to caching of memory locations
- The rule to follow: you should do something to prevent interference whenever there are code fragments that deal with shared data, and at least one fragment modifies the data

Overview

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

Need for atomicity

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

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 (eg int)
What does it do?

- Before a thread can enter into a synchronized block, it must acquire the "intrinsic 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

```java
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
}
```

t1: call `X.foo(myA, myB)`
t1: assign y to 17
t1: get lock on `myA`
t1: read y (value 17)
t1: assign to y the value 18
```
t1: call `myB.bar()`
// steps in `bar()`
```
t1: return from `bar`
t1: release lock on `myA`
t1: read y (value 18)
t1: calculate 17+1
t1: return from `foo` (return value: 18)
t1 holds the lock on `myA` throughout the period marked
```

Blocking on a lock

- When a thread tries to acquire the lock but cannot 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 runs
  - 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

In class ConnectServerB

```java
public synchronized void addConnection() {
    int currentCount = myConnectionCount;
    currentCount += 1;
    myConnectionCount = currentCount;
}

public synchronized void removeConnection() {
    myConnectionCount--;    

```  

Log showing no lost update

Suppose T1 and T2 both have ConnectServerB instance cs as value of variable s. Here is an example with heap: cs = new ConnectServerB()

```
T1: call addConnection()  // body
T2: call addConnection()  // body
T1: read myConnectionCount (value: 5)
T2: read myConnectionCount (value: 5)
T1: set currentCount to 5
T2: set currentCount to 5
T1: call removeConnection()  // body
T2: call removeConnection()  // body
T1: set currentCount to 6
T2: set currentCount to 6
T1: call s.addConnection()  (value: 6)
T2: call s.addConnection()  (value: 6)
T1: try to get lock on cs (blocked)
T2: try to get lock on cs (blocked)
T1: remove lock on cs
T2: remove lock on cs
T1: release lock on cs
T2: release lock on cs
T1: get lock on cs, enter synchronized block
T2: get lock on cs, enter synchronized block
T1: return from s.addConnection()  
T2: return from s.addConnection()
```

```
T1: myConnectionCount=5
T2: myConnectionCount=5
T1: currentCount=5
T2: currentCount=5
T1: stack of T1: 5
T2: stack of T2: 5
```

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 mutable field is inside synchronized blocks involving the same lock.
- These combine to show: you never have concurrently two threads which deal with the shared field.
  - That is, no interference is possible on this field.

Warning

- On multiprocessor machines, the JVM can produce effects that can't be explained by any interleaving (due to caching of variables).
- The language ensures, though, that if all threads that share some mutable data use it only inside synchronized blocks (with the same instance locked) then everything works properly.

Dangerous talk!

- It is common to say "the access to the field 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 field 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() {
    // this is not synchronized
    inwardConnectionCount--;
    outwardConnectionCount++;
}
public synchronized int totalConnections() {
    return (inwardConnectionCount + outwardConnectionCount);
}
```

Outline Log of interference

Initial state- inwardConnections: 30; outwardConnections: 6
- In both threads the value of s is the ConnectServerC instance cs
- T1: call s.totalConnections()
- T1: enter synchronized block – get lock on cs
- 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 cs
- T1: return from s.totalConnections (return value=37)

Final state- inwardConnections: 29; outwardConnections: 7

Locking convention

- It's essential for all programmers who write code that deals with a shared field to agree on what object they will use for the synchronized blocks that guard that field.
- The common rule is: synchronize on the object in which the field is an instance variable.
Locking is needed

- Any place where a shared mutable field is used
- The field might be mentioned in an expression, placed as an argument in a call, assigned to, or the target of a call
- Make sure the use is inside a synchronized block which obtains the appropriate lock
  - As given by the convention for the particular field

Multiple Use

- If a shared field is used several times in some code,
  - And the later use depends on nothing changing since the earlier use
- Then make sure the synchronized block surrounds all the uses of this field
  - As given by the convention for the particular field

Multiple Use

- A common pattern of multiple use is to get the value from some field, and then set that field based on what was got
- Code that does this should protect against another thread changing the field in between the time the getter executes, and the time the setter executes

Static fields and methods

- The static keyword means that a member or method belongs to an entire class, not separately to a single instance
- There is an instance of myID for each instance of Car
- The numberCreated field is shared between all instances of the Car class
  - It can even be accessed without any Car instances being created
  - Car.numberCreated = 7;
- 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 for static fields, methods

- 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
- Common mistake: forget to lock the class in non-static methods which access a static instance variable
  - Never changed after construction, doesn't need protection

When locking isn't needed

- Data that is confined to one thread
  - Fields in an object that can only be reached from local variables
  - Take care not to allow a reference to escape by being passed or returned
- Data that is immutable
  - All fields are final
  - No modification after construction
  - And reference doesn't escape during construction

But err on the side of caution: it is far more common, and far more dangerous, to leave out synchronization that is needed, than to put in unnecessary extra locking!
Overview

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

Log with nested locking

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

1. Call payFee in cs, lock on cs
2. Call getBal in acct1, lock on acct1
3. Set bal to bal + amount
4. Set lock on acct1
5. Return from getBal

Hidden nested locking

1. Call another method which has a synchronized block, from inside a synchronized block
2. Call getBal in VarType var, lock on VarType
3. Set var.call to 10
4. Set lock on VarType
5. Return from call

Blocking while holding a lock

- In general, code that holds a resource like a lock should be written to run quickly
- Thus, avoid I/O inside synchronized block

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!
class EgB { 
  private int thing;
  public synchronized int thing() { 
    return thing;
  } 
  public synchronized String getThing() { 
    return thing;
  } 
  public synchronized void setThing(int thing) {
    this.thing = thing;
  } 
} 

// JVM allocates T1 to run
T1 enters synchronized block, gets lock on y
T1 assigns "ABC" to val
T1 calls y.meth2(), returns
T1 leaves synchronized block, releases lock on y; T2 can now run
T2 calls x.meth1(), tries to get intrinsic lock on x, but lock on x is already held by T1; T2 is blocked
T2 increments thing, sets it to 4
T2 leaves synchronized block, asks for lock on x, but lock on x is already held by T2; T2 is blocked
T2 increments thing, sets it to 5
T2 gets lock on x and then carries on executing meth2() 

The lock is obtained on the value of the expression that governs the synchronized block
The expression can be a variable
In different threads, the same variable may refer to different objects
In one or more threads, different variables may refer to the same object (sharing)
So draw the object diagram, to work out which instances are actually being locked!

class EgA { 
  private int thing;
  public synchronized void meth1() { 
    thing += 1;
  } 
} 

public synchronized void
traverse(BankAccount target, int amount) 
{ 
  if (myBank != target.myBank) 
    throw new DifferentBankException(); 
  balance -= amount;
  synchronized(target) { 
    target.balance += amount;
  } 
} 

Note: private fields of another instance can be mentioned in code, but should be protected
Care with references

- Suppose in T1, myAccount is x and other is y.
- T1 executes myAccount.transfer(other, 10).

The effect is nested synchronized blocks:
- synchronized(x) {
- synchronized(y) {

Deadlock is possible!

Other deadlock situations

- Suppose in T2, myAccount is y and other is x.
- T2 executes myAccount.transfer(other, 20).

The effect is nested synchronized blocks:
- synchronized(y) {
- synchronized(x) {

None of the threads can proceed; all are blocked.

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.

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 account.
  - 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).

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.

Coarse-grained locking

- Usual convention: lock instance in which shared field occurs.
- An alternative locking convention is to protect all accesses to any of the fields in several different objects by locks on a single object x.
  - We say that the lock on x covers all the fields.
- Most often: all fields 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 field in any of its connections.
Trade-offs

- Interleaving will be prevented for methods that access different shared fields 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 fields 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 field
  - 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 field concerned
  - A single method written with a different convention would risk data interference
  - So document the convention where the field is declared

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

Recursive locking -cntd

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

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 likelihood of deadlock
Overview

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

Changes in Java SE 5

- A new library java.util.concurrent
  - Designed by Doug Lea and others
  - Many classes including
    - Concurrency-aware collections
    - Utilities for locking and other synchronization
  - Even when they provide the functionality already available, they have often been implemented in ways that work more efficiently
  - Especially on multiprocessor systems
- Clarifying the memory model
  - Helping programmers know what they can (and can't) rely on, in the way JVM works in multiprocessor systems

ConcurrentHashMap

- java.util.concurrent.ConcurrentHashMap<K,V>
- implemetns Map<K,V>
- Thread-safe (especially: state is never corrupted)
  - And highly concurrent
  - Several threads can concurrently look-up and others can insert
  - Note: java.util.HashMap<K,V> is not thread-safe
  - Retrievals or iterators may see partial impact from multi-step modifications done by other threads
    - Eq if T1 calls clear(), T2 may see some but not all pairs
    - Eq if T1 calls size() the value may be misled by concurrent changes

Additional methods

- Insert only if no value already known for this key
  - behaves like
    - if(!containsKey(key)){put(key,value);}
  - But without danger of interference between test and action
    - V putIfAbsent(K key, V value)
- Remove only if key currently has given value
  - boolean remove(K key, V value)
- Change value associated with given key, but only if currently mapped to oldValue
  - V replace(K key, V newValue, V oldValue)

CopyOnWriteArrayList

- java.util.concurrent.CopyOnWriteArrayList<E>
- implements List<E>
- Thread-safe
  - And highly concurrent
  - Note:LinkedList<E> or ArrayList<E> are not thread-safe
  - An iterator returns the elements in the collection when the iterator was constructed, no matter what later modifications are made in other threads
  - But Iterator doesn't support modification operations like add(), remove() or set()
- Performs well if there are many traversals but few modifications

BlockingQueue

- An interface with several implementations
  - LinkedBlockingQueue, PriorityBlockingQueue
- Optional capacity limit
- Remove (and return) head element
  - Block if queue is empty, until some other thread inserts something to remove
  - E take()
- Insert element at tail
  - Block if queue has reached capacity, until some other thread makes space by removing something
  - void put(E e)
Producer/Consumer

Many problems can use BlockingQueue to pass work items from one or more producer threads to one or more consumer threads.

Eg customers make requests which are handled by advisors.

CountDownLatch

- Initialized with a count N
- Any thread which calls await() will block until after N calls have been made to countDown()
- Once N countDowns have been made, all blocked threads become runnable again, and subsequent calls to await() are not blocked.

Example

```java
class Driver {
    void main() throws InterruptedException {
        CountDownLatch startSignal = new CountDownLatch(1);
        CountDownLatch doneSignal = new CountDownLatch(N);
        for (int i = 0; i < N; ++i) {
            new Thread(new Worker(startSignal, doneSignal)).start();
        }
        doSomethingElse();
        startSignal.countDown();
        doSomethingElse();
        doneSignal.await();
    }
}

class Worker implements Runnable {
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
        this.startSignal = startSignal;
        this.doneSignal = doneSignal;
    }
    public void run() {
        try {
            startSignal.await();
            doWork();
            doneSignal.countDown();
        } catch (InterruptedException ex) {
        }
    }
    void doWork() { ... }
}
```

Taken from Sun documentation

ReentrantLock

- Construct an object of this class, and use it to provide synchronization
- Methods
  - void lock(); // acquire the lock
  - void unlock(); // release the lock
  - In Java SE6 this performs much better than synchronized blocks, but advantage disappears in Java SE6

Comparison with synchronized blocks

- Explicit lock/unlock() need not be nested in the way blocks always are.
  - Eg A.lock(); B.lock(); A.unlock(); B.unlock();
- The unlock() may be in a different method or even class from the lock()
- This allows flexibility, but it risks programmer errors.
- Eg forgetting to unlock()
- Also has methods
  - boolean tryLock() // doesn't block if lock unavailable
  - boolean tryLock(long timeout, TimeUnit unit)

ReentrantReadWriteLock

- Once created, you get two lock objects
  - Act in concert, like two aspects of a single synchronization management
  - Readers should acquire and release one, and writers acquire/release the other
  - The JVM ensures either
    - only one writer and no readers hold the locks
    - Or no writers and any number of readers
State-dependent classes

- Like BlockingQueue, sometimes one wants objects which only perform an operation when the state is suitable
  - Otherwise, the thread requesting the operation should block until some other thread fixes the state to be suitable
- Java (even before SE5) provides two methods which can build this behaviour in a general way

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
    - These threads are all non-runnable
  - 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

wait() and notify()/notifyAll()

- 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

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

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
  - In a synchronized method or block
    ```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!

Overview

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

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

Documenting

- When you write a class, you should explain how it behaves in multi-threaded situations
  - Put explanation in the Javadoc
    - Remember that "synchronized" keyword is not part of the API
  - Or use annotations (Java SE5)
    - net.jcip.annotations has useful ones such as "@ThreadSafe"
- Examples:
  - "This class is thread-safe"
  - "Every use of this class must be guarded by a lock on the root of the tree"
Thread-Safety

- A class is "thread-safe" if all methods contain sufficient synchronization that they will appear to execute without interference.
  - Return the appropriate values, and affect the state in the appropriate way, as if each method ran without interleaving.
  - This should happen even if multiple threads call an instance's methods concurrently (with no synchronization outside the class).

Usual Approach to Thread-Safety

- Identify fields 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 fields:
  - Make them all synchronized.

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 a field which is accessed in several threads (and modified in one or more of them):
  - Eg. field is read (or used as target, etc), and another thread might write.
  - Eg. field 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 a field that is accessed several times in the thread.
  - Look for several fields which might have some consistency requirement which is temporarily violated (between two modifications).

Reminder: even if a field is accessed only once, that one access must be in a critical section.

Choosing the API

- If there is a natural sequence of method calls, that clients will want to do indivisibly, it is good to offer the combination as a single synchronized method.
- Otherwise, even if individual methods are thread-safe, the client will still need to synchronize around a sequence.
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

- StringBuffer
  - StringBuffer sb;
  - StringBuffer sb1;
  - StringBuffer sb2;
  - // initialize ab to "foo"
  - sb1 = sb;
  - // sb and sb1 refer to the same instance
  - sb.append("fred");
  - // sb is now "foofred"
  - // sb1 is also "foofred"

- String
  - 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 "foofred"
  - // s2 is also "foofred"

append() modifies sb
concat() does not modify s
It makes a new String

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
Overview

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

Client-Server programming

- examples:
  - Web (HTTP)
  - Database
  - File (FTP/NSF)
  - mail (POP)

Client/Server basics

- Each client and each server runs as a separate process,
  - on a separate JVM, if written in Java
  - may be on separate computers, or on the same one.
- They communicate using the network to send and receive messages
- Usually, client sends a request, which leads server to perform some activities, and then server sends back a response, which the client receives.

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).

![ServerSocket](image)

### 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.
```
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();
        }
    }
}
```

### The problem 1

- Since `accept()` is invoked in the main thread, that's only task this SimpleServer can perform.

### 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.
The problem 2

- SimpleServer processes one request at a time.
- This would acceptable only if the interaction with each client is very brief.
  - 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.

  - Quite simple
    - Works well with a fairly small number of possibly 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) { 
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Answer to the problem 2 – cont’d

```java
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();
  }
}
```
Problems of thread-per-request (1)

- It creates a new thread each time.
  - Creating a new object is time-consuming.
  - The overhead is significant when the number of objects is large.
  - It could spend more time creating/destroying threads than the actual data processing.

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.

Other mechanisms

- Fixed size Thread pool: keep a fixed collection of threads available, and allocate each job to one available thread
  - If no thread is available, job waits till some thread finishes whatever it was doing.
- Caching Thread pool: reuse available thread if possible, but create new thread if needed
  - Like thread-per-request but with less overheads.
  - Jobs are never delayed.

Using Executor

- In Java SE5, there is an interface `Executor` with several standard implementations produced in static factory methods, which provide various thread management policies.
- Key idea: construct an appropriate `Executor`, and construct a Runnable to perform one job, and pass that Runnable to the `Executor`'s `execute` method.
  - The Runnable is often an anonymous inner class to allow access to data in the server.

```java
public class MyJob implements Runnable {
    Socket mySocket;
    public MyJob(Socket aSocket) {
        mySocket = aSocket;
    }
    public void run() {
        try {
            InputStream is = mySocket.getInputStream();
            OutputStream os = mySocket.getOutputStream();
            while (true) {
                int ch = is.read();
                if (ch == -1) {
                    break;
                }
                os.write(ch);
                os.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```
public class FlexibleServer {
    private static final int NTHREADS = 10;
    public static void main(String[] args) throws IOException {
        Executor exec = Executors.newFixedThreadPool(NTHREADS);
        try {
            ServerSocket ss = new ServerSocket(port);
            while (true) {
                try {
                    Socket s = ss.accept();
                    OneJob job = new OneJob(s);
                    exec.execute(job);
                } catch (IOException e) {} 
            } 
        } catch (IOException e) {} 
    }
}

Just replace this by newSingleThreadExecutor, Or newCachedThreadPool

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

Two examples
- Threading in C
  - Pthreads library
  - syntax and semantics
  - building higher-level constructs
- Threading in C#
  - syntax and semantics
  - comparison with Java

Further reading
- "Computer Systems: A Programmers Perspective" by Bryant and Halloran
- "Multithreading with C#" by R. Krikorian
  http://www.oreillynet.com/pub/a/dotnet/2001/08/06/csharp.html

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

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 notifyAll() and notify() except that thread need not hold mutex when this function is called

Semaphores

- 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
    - 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
- `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
- `AcquireReaderLock, AcquireWriterLock, UpgradeToWriterLock, ReleaseLock`
- A similar class is in Java 1.5 library:
  - `java.util.concurrent.locks.ReentrantReadWriteLock`