Threads and Synchronization

Mark Allen Weiss

1

Outline of Topics

- **What threads are**
- **The `Thread` class and starting some threads**
- **Synchronization: keeping threads from clobbering each other**
- **Deadlock avoidance: keeping threads from stalling over each other**

2

Multitasking

- *Multitasking* means that you can have several processes running at same time, even if only one processor.
- Can run a browser, VM, powerpoint, print job, etc.
- All modern operating systems support multitasking
- On a single processor system, multitasking is an illusion projected by operating system

3

Threads

- Inside each process can have several threads
- Each thread represents its own flow of logic
  - gets separate runtime stack
- Modern operating systems support threading too; more efficient than separate processes
- Example of threading in a browser:
  - separate thread downloads each image on a page (could be one thread per image)
  - separate thread displays HTML
  - separate thread allows typing or pressing of stop button
  - makes browser look more responsive

4

Threads in C/C++

- Threads are not part of C or C++
- Have to write different code for each operating systems
- Difficult to port

5

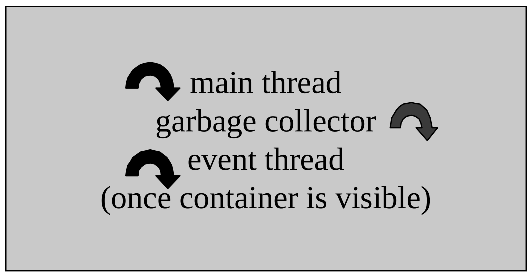
Threads in Java

- Part of language
- Same code for every Java VM
- Simpler than in most other languages
- Still very difficult:
  - When running multiple threads, there is nondeterminism, even on same machine
  - Often hard to see that your code has bugs
  - Requires lots of experience to do good designs

6

Threads in the Virtual Machine

- **VM has threads in background**
- **VM alive as long as a "legitimate thread" still around (illegitimate threads are "daemons")**
- **GUI programs will start separate thread to handle events once frame is visible**



7

Thread Class

- **Use `Thread` class in `java.lang`**
- **Two most important instance methods:**
  - **`start`: Creates a new thread of execution in the VM; then, invokes `run` in that thread of execution; current thread also continues running**
  - **`run`: explains what the thread should do**
- **`Thread` is not abstract, so there are default implementations**
  - **`start` does what is described above; should be final method (but isn't)**
  - **`run` returns immediately**

8

Creating A Do Nothing Thread

- **The following code creates a `Thread` object, then starts a second thread.**

```
public static void main( String[] args ) {
  Thread t = new Thread( );
  t.start( ); // now two threads, both running
  System.out.println( “main continues” );
}
```

- **In code above:**
  - **First line creates a `Thread` object, but `main` is the only running thread**
  - **Second line spawns a new VM thread. Two threads are now active.**
  - **main thread continues at same time as new thread calls its `run` method (which does nothing)**

Getting Thread to Do Something

- **Option #1: extend `Thread` class, override `run` method**

```
class ThreadExtends extends Thread {
  public void run( ) {
    for( int i = 0; i < 1000; i++ )
      System.out.println( "ThreadExtends " + i );
  }
}
class ThreadDemo {
  public static void main( String[] args ) {
    Thread t1 = new ThreadExtends( );
    t1.start( );
    for( int i = 0; i < 1000; i++ )
      System.out.println( "main  " + i );
   }
}
```

Alternative to Extending `Thread`

- **No multiple inheritance; might not have an extends clause available**
- **Might not model an IS-A relationship**
- **Really just need to explain to `Thread` what `run` method to use**
  - **Obvious function object pattern**
  - **run is encapsulated in standard Runnable interface**
  - **implement Runnable; send an instance to Thread constructor**
  - **preferred solution**

11

Alternative #2: Using `Runnable`

```
class ThreadsRunMethod implements Runnable {
  public void run( ) {
    for( int i = 0; i < 1000; i++ )
      System.out.println( "ThreadsRunMethod " + i );
  }
}

class ThreadDemo {
   public static void main( String[] args ) {
     Thread t2 = new Thread ( new ThreadsRunMethod( ) );
     t2.start( );
     for( int i = 0; i < 1000; i++ )
       System.out.println( "main  " + i );
   }
}
```

12

Anonymous Implementation

- **May see the `Runnable` implemented as an anonymous class in other people's code**

```
class ThreadDemo {
   public static void main( String[] args ) {
     Thread t3 = new Thread ( new Runnable( ) {
         public void run( ) {
         for( int i = 0; i < 1000; i++ )
           System.out.println( "ThreadAnonymous " + i );
         }
       }
     );
     t3.start( );
     for( int i = 0; i < 1000; i++ )
       System.out.println( "main  " + i );
   }
}
```

13

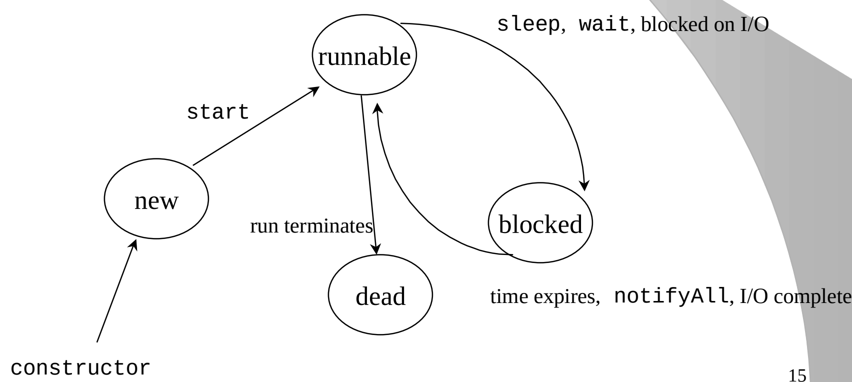
Common Mistake #1

- **You should NEVER call `run` yourself**
  - **will not create new VM thread**
  - **will not get separate stack space**
  - **will invoke `run` in the current thread**
- **`start` don't `run`**

14

Thread States

- **Thread is not runnable until start is called**
- **Thread can only unblock if cause of blocking is resolved**



15

Is The Thread Alive?

- **Cannot differentiate between being runnable and blocked.**
- **Thread that is runnable or blocked is alive**
- **Thread that has not started or is dead is not alive**
- **Can use `Thread` instance method `isAlive` to determine thread status**

16

Uncaught Exceptions

- **Uncaught exception terminates a thread's `run` method**
- **Does not terminate the VM unless there are only daemon threads left**
- **`run` cannot list any checked exceptions in its throws list (why not?)**

17

Thread Methods

- **instance methods**
  - **setDaemon**
  - **isDaemon**
  - **setPriority**
  - **getPriority**
  - **interrupt**
  - **join**
- **static methods**
  - **sleep**
  - **yield**

18

Current Thread

- **Before you can invoke any `Thread` instance method, you need a reference to the current thread**
  - **If you extend `Thread`, no problem. In your `run` method, `this` represents current `Thread` and can be omitted**
  - **If you use `Runnable`, in your `run` method `this` represents the `Runnable` object. Need to use static method `Thread.currentThread`**

```
Thread self = Thread.currentThread( );
```

19

Deamon Threads

- **By themselves do not keep a VM alive**
- **Can mark a thread as a daemon thread by calling `setDaemon(true)`**
- **Call must be before call to `start`; after call an exception is thrown**
- **Without call to `setDaemon` thread's daemon status is same as thread that spawned it**
- **Can call `isDaemon` to see if thread is a daemon**

20

Thread Priorities

- **Can *suggest* to VM that when there is contention for CPU, some threads should get preference over others.**
  - **Only considered when there's CPU contention; threads that are sleeping won't go any faster with higher priorities**
  - **If your program depends on priorities, you need to do more work; VM could ignore suggestions**
  - **Priority of thread is same as thread that created it**
  - **Only 10 priorities ranging from `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`, with `Thread.NORM_PRIORITY`**

21

Current Implementations

- **Windows 98/NT and Solaris Native Threads:**
  - **schedule highest priority thread**
  - **scheduling is fully preemptive: if a new highest priority thread becomes runnable, it gets scheduled**
  - **rule of thumb: at any given time, highest-priority thread is running. But this is not guaranteed by language spec.**
  - **Java platform does not time-slice, underlying thread platform does (Solaris Green Threads does not), so if several highest priority threads, system generally does simple, non-preemptive round-robin**

22

Interrupting A Thread

- **Any thread can interrupt any other thread (if it has a reference to its `Thread` object) by invoking `interrupt` on that `Thread` object.**
  - **Used if target thread is deliberately blocked (sleeping, waiting, yielding or otherwise not interested in getting the processor right now, but not blocked on I/O)**
  - **If target thread is deliberately blocked, interrupt sends an `InterruptedException` to the thread, which wakes thread up**
  - **If target thread is no longer deliberately blocked, interrupt is ignored**

`Interrupted` Exception

- **`InterruptedException` is a checked exception; must be caught or propagated by host of `Thread` routines that cause thread to give up the processor**
  - **Really annoying**
  - **Probably should terminate thread**

join

- **The call `t1.join( )` causes the current thread to block until `t1` terminates**
- **Have to catch `InterruptedException`**
- **`main` can join on all threads it spawns to wait for them all to finish**

yield

- **Threads that are CPU intensive can hog all the cycles, especially if they are high priority**
- **Polite thread yields every now and then**
  - **not too often; could be spending too much time context switching**
  - **`yield` is a static method.**
- **Current thread**
  - **Gives up the processor if another thread of at least as high priority is waiting for the CPU**
  - **If no eligible thread, current thread retains processor**
- **Must catch `InterruptedException`**

`sleep`

- **Static method.**
- **Current thread**
  - **Gives up the processor for at least the time specified**
  - **Time is in milliseconds**
  - **No guarantee that you get processor back**
- **Must catch `InterruptedException`**

Timeouts

- **can invoke `wait` and `join` with a parameter that limits the amount of blocking (in milliseconds)**
  - **for `wait` not necessarily a great idea**
- **Example: thread needs to do I/O; what if nothing is typed?**
  - **Do I/O in a separate thread**
  - **main thread does a `join`, with timeout on the I/O thread**
  - **If no I/O, main thread will continue and can terminate itself and I/O thread if needed**

Shared Data

- **All threads share the VMs memory**
  - **useful if threads are going to do real work**
- **If two threads have references to the same object, they can potentially simultaneously invoke methods on the object**
  - **ok if both accessing**
  - **might be bad if one thread is mutating**
  - **could be a disaster if two threads are mutating**

Example

```
class TwoObjs {
  private int a = 15;
  private int b = 37;

  public int sum( )   { return a + b; } // should always be 52
  public void swap( ) { int tmp = a; a = b; b = tmp; }
}
```

- **Two threads share a reference to some `TwoObjs` object, and the following steps occur**
  - **Thread 1 invokes `swap`, and immediately after executing `a=b` is time-sliced out.**
  - **Thread 2 invokes `sum`, and returns 74.**
- **Despite private data, and object has been accessed while in an inconsistent state**

Two Mutators Do Serious Damage

- **Last example not so bad**
  - **We temporarily see object in a bad state**
  - **Thread 1 gets time-sliced in and object gets back in good state**
  - **Often we view objects in bad states, and we know that current information may be inaccurate, but will eventually be correct**
    - **bank accounts**
    - **frequent flyer accounts**
    - **credit card statements**
- **When two mutators interact, can irreversibly damage object state**

31

Two mutators

```
class TwoObjs {
  private int a = 15;
  private int b = 37;

  public int sum( )   { return a + b; }  // should always be 52
  public void swap( ) { int tmp = a; a = b; b = tmp; }
}
```

- **Starting from good state**
  - **Thread 1 invokes `swap`, and immediately after executing `tmp=a` is time-sliced out. In this thread `tmp=15`.**
  - **Thread 2 invokes `swap`, swapping `a` and `b`. `a` is now 37, `b` is now 15.**
  - **Thread 1 is time-sliced back in and continues: `a` is now 15, `b` is now `tmp`, so `b` is 15. OOPS!**

32

Can This Really Happen?

- **Yes but,**
  - **It can be fairly rare**
  - **Depends on speed of processors**
  - **Depends on number of processors**
  - **Depends on thread priorities**
  - **Depends on luck of the draw**
- **Worst kind of bug**
  - **`TwoObjs` class is not thread-safe**
  - **Could do millions of operations and never see a problem**
  - **Hard to know you've messed up**

How Java Solves The Problem

- **Use the `synchronized` keyword**
- **Marking an instance method as synchronized means that in order to invoke it the thread must gain possession of the "monitor" for the invoking object (i.e. the "monitor" for `this`).**
- **The *monitor* is an abstraction**
  - **every object has one and only one**
  - **no getMonitor method, however**

How It Works

- **To enter a synchronized method, thread must**
  - **either already own the monitor (perhaps this method is being called from another synchronized method)**
  - **get the monitor**
  - **once in, if you are timesliced out, you will keep the monitor, blocking other threads out**
- **If another thread already owns the monitor and has been timesliced out, you will be blocked from obtaining the monitor**
- **When thread leaves method from which it obtained monitor, monitor is released by VM**

35

Unsynchronized Methods

- **Only synchronized methods require the obtaining of a monitor**
- **Synchronization is very expensive**
- **Sun recommends:**
  - **synchronize everything**
- **Less drastic:**
  - **synchronize mutators**
  - **synchronize accessors depending on the tradeoff of occasional bad data versus performance**

36

Example #1

- **Assume both print and swap are synchronized**
  - **Thread #1 does `obj.swap( )`**
    - **can obtain `obj`'s monitor and enter**
  - **Thread #1 is timesliced out in the middle of swap**
    - **Thread #1 holds on to `obj`'s monitor**
  - **Thread #2 does `obj.print()`**
    - **Thread #2 needs `obj`'s monitor. Can't get it, so thread is blocked**
  - **Thread #1 is timesliced in; finishes swap**
    - **Thread #1 releases `obj`'s monitor**
  - **Thread #3 does `obj.print()`**
    - **Thread #3 gets the monitor and proceeds**

37

Example #2

- **Assume only swap is synchronized**
  - **Thread #1 does `obj.swap( )`**
    - **can obtain `obj`'s monitor and enter**
  - **Thread #1 is timesliced out in the middle of swap**
    - **Thread #1 holds on to `obj`'s monitor**
  - **Thread #2 does `obj.print()`**
    - **Thread #2 does not need `obj`'s monitor, so it proceeds**
  - **Thread #1 is timesliced in; finishes swap**
    - **Thread #1 releases `obj`'s monitor**

38

Example #3

- **Assume swap is synchronized, and `obj1` and `obj2` are different objects**
  - **Thread #1 does `obj1.swap( )`**
    - **can obtain `obj1`'s monitor and enter**
  - **Thread #1 is timesliced out in the middle of swap**
    - **Thread #1 holds on to `obj1`'s monitor**
  - **Thread #2 does `obj2.print()`**
    - **can obtain `obj2`'s monitor and enter, so it proceeds**
    - **when it finishes it releases `obj2`'s monitor**
  - **Thread #1 is timesliced in; finishes swap**
    - **Thread #1 releases `obj1`'s monitor**

39

Static Methods

- **Synchronized static methods require the obtaining of a monitor also**
  - **can't be the objects monitor because there is not**
  - **the monitor it needs to obtain the monitor for the `Class` object.**
- **May be important for fancy stuff**
- **Just remember that instance methods and static methods use different monitors**

40

Synchronized Block

- **Often don't need to synchronize entire method**
  - **just need to synchronize a "critical section"**
  - **few lines of code that should be viewed as an "atomic" single operation**
- **Use a synchronized block**

```
synchronized( anyobject )
{
   // must have possession of monitor for anyobject

   // will release if obtained (not just inherited)
}
```

41

These are Equivalent

```
public class Foo  // Version #1
{
  synchronized public void foo( ) { ... }
  synchronized static void bar( ) { ... }
}

public class Foo  // Version #2
{
  public void foo( )
  {
    synchronized( this ) { ... }
  }
  static void bar( )
  {
    synchronized( Foo.class ) { ... }
  }
}
```

42

Synchronized Is Not Inherited

- **As previous slide shows, synchronized in method header is just a convenience**

Synchronization Rule #1

- **Can only synchronize methods and code**
- **Can never synchronize data, so**
- **RULE #1: ALL DATA MUST BE PRIVATE OR YOU LOSE**

Synchronization Rule #2

- **RULE #2: Any code/methods that makes changes to shared variables must use `synchronized` to ensure safe concurrent access.**
- **Accessors are often decided based on performance requirements.**

45

Synchronization Rule #3

- **RULE #3: Be careful about propagating exceptions through a critical section.**
  - **Can have a half-way done operation if you do this**
  - **This is why `stop` is deprecated**

46

Synchronization Rule #4

- **Rule #4: Never call `sleep` in a synchronized block.**
  - **If you call `sleep`, you give up the processor, but not the monitor.**
  - **Anybody else who needs the monitor will be blocked**
  - **Can cause deadlock**
  - **This is why `suspend` is deprecated**

47

How to Wait For Conditions

- **If you are in a synchronized block and need to stall for an external event**
  - **use `mon.wait()`, where `mon` is the monitor that you own.**
- **`wait`**
  - **gives up the processor**
  - **gives up the monitor**
  - **makes you ineligible to ever be rescheduled unless either a timeout expires, an interrupt occurs, or somebody else issues a `notifyAll`**

48

notify vs notifyAll

- **Once thread has done a wait, another thread the rectifies situation should issue a `mon.notifyAll()`.**
- **`mon.notifyAll` reinstates scheduling eligibility for all threads that issued a `mon.wait()`**
- **`mon.notify` reinstates scheduling eligibility for one thread (VM chooses, not you) that issued a `mon.wait()`**
  - **extremely dangerous to use `notify` unless you know there is only one thread waiting. This method should be deprecated**

49

`wait` and `notifyAll`

- **You must own the monitor when you execute either of these**
- **Runtime exception thrown if you don't own monitor**
- **Common mistake is to use `wait()` or `notifyAll()` without specifying monitor. Defaults to `this.wait()` and `this.notifyAll()`, which only works if the monitor is `this`.**
- **Typically, `wait` is in a very tight while loop, NOT an if statement**

50

Synchronization Rule #5

- **The `wait`/`notifyAll` pattern:**
  - **Place `wait` in a tight while loop that loops as long as a required condition is not yet met**
  - **Code that could fix the condition issues `notifyAll`**
  - **Never use `notify`**
  - **remember that these are instance methods for the monitor that you are willing to release**

51

Deadlock

- **Occurs when two threads are each waiting for monitors they can't both get.**
- **Example:**
  - **Thread #1 needs monitors A and B**
  - **Thread #2 needs monitors A and B**
  - **Thread #1 has A**
  - **Thread #2 has B**
  - **Deadlock**
- **Java does not detect deadlocks**
- **Avoiding deadlocks very difficult; requires lots of experience**

52

Sun's Deadlock Avoidance Trick

- **Use an internal private object to synchronize:**

```
class Account {
  public void deposit( int d ) {
    synchronized( CRITICAL_SECTION_1 ) {
      balance += d;
    }
  }
  public void withdraw( int d ) throws OverdraftException {
    synchronized( CRITICAL_SECTION_1 ) {
      if( balance >= d )
        balance -= d;
      else
        throw new OverdraftException( "" + d );
    }
  }
  private int balance = 0;
  private Object CRITICAL_SECTION_1 = new Object( );
}
```

53

Synchronization Rule #6

- **Rule #6: Always obtain monitors in the same order**
  - **Often involves finding an immutable totally-orderable property of the object's whose monitor you will need, and obtaining monitors using that order**
  - **Example: obtaining monitors for two bank accounts, use account #s, and obtain lower account #'s monitor first**

54

Summary

- **Threading is an essential part of Java and any real program. Easier in Java than elsewhere**
  - **tells you how hard it is elsewhere**
- **Follow the rules**
  - **start don't run**
  - **don't rely exclusively on priorities**
  - **no public data**
  - **synchronize mutators, maybe accessors**
  - **leave critical section only after object is restored**
  - **no sleeping in synchronized block**
  - **use `wait`/`notifyAll` pattern**
  - **obtain monitors in same order**