Concurrent Programming with Java

Douglas C. Schmidt

schmidt@cs.wustl.edu

Washington University, St. Louis

- Portions of this material are based on Doug Lea's book "Concurrent Programming in Java"
  - http://www.awl.com/cp/lea.html

Concurrent Processing in Java

- Java concurrency model
  - Combination of *Active* and *Passive Objects* based on *Threads* and *Monitors*
- Think of each Thread as having its own "logical processor"
  - *i.e.*, an "active object"
- On uni-processors Threads may share the CPU and have their execution interleaved
- On multi-processors Threads may be scheduled and run in parallel
  - Java specificiation doesn't mandate parallelism

Challenges with Java Threading

- Hard part is that threads are not independent
- You must provide for
  - *Synchronization*
    - ▷ How a thread knows whether or not another thread has completed a particular portion of its execution
  - *Shared resources*
    - ▷ Mutual access and mutual exclusion
  - *Communication*
    - ▷ Often done by a combination of synchronization and shared resources
    - ▷ *e.g.*, message passing and shared memory

Java Threading Problems

- *Avoiding Deadlock*
  - No work being done because every thread is waiting for something that another thread has
  - Particularly problematic in Java due to "nested monitor problem"
- *Avoiding Livelock* (Lockout)
  - A thread is indefinitely delayed waiting for resources being used elsewhere
- *Maintaining Liveness*
  - Nothing that is supposed to happen will be delayed indefinitely

Threading Problems (cont'd)

- *Scheduling*
  - Allocating shared resources "fairly"
    - ▷ Must adjust for the fact that some threads are more urgent ("higher priority") than others
- *Non-determinism*
  - The order in which events happen is not, in general, fully specified or predicatible
- *Performance*
  - Context switch, synchronization, and data movement can be bottlenecks

Goals of Java Concurrency Control

- Resource accessed by one thread at a time
- Each resource request satisfied in finite time
- Abnormal termination of thread does not directly harm other threads that do not call it
- Waiting for a resource should not consume processing time (*i.e.*, no "busy waiting")

Concurrency Control Techniques

- *Critical Regions* (*e.g.*, using mutexes and semaphores)
  - Define a critical region of code that accesses the shared resources and which can be executed by only one thread at a time
- *Tasking and task rendezvous* (*e.g.*, Ada)
  - Combines synchronization and communication
- *Monitors* (*e.g.*, Java)
  - A monitor is a collection of data and procedures where the only way to access the data is via the procedures and only one of the procedures in the monitor may be executing at one time and once a procedure starts to execute
  - All calls to other procedures in the monitor are blocked until the procedure completes execution

Threads in Java

- Java implements concurrency via Threads
  - Threads are a built-in language feature
  - The Java Virtual Machine allows an application to have multiple threads of executing running concurrently
- The Java `Thread` class extends `Object` and implements `Runnable`

```
public class java.lang.Thread
    extends java.lang.Object
    implements java.lang.Runnable
{
// ...
}

public interface Runnable {
  public void run();
}
```

Java Threading Example

- A Thread-safe Stack

```
import java.lang.*;

interface My_Stack
{
  public void push (Object item);
  public Object pop ();
  public Object top ();
}

class MT_Bounded_Stack implements My_Stack
{
  public MT_Bounded_Stack () { this (50); }

  public MT_Bounded_Stack (int max_size) {
      this.top_ = 0;
      this.max_ = (max_size);
      this.stack_ = new Object[this.max_];
  }

  private Object[] stack_;
  private int top_;
  private int max_;
```



```
  public synchronized void push (Object item) {
    while (this.is_full ())
      try { wait(); } catch (InterruptedException ex) {};
    this.stack_[top_] = item;
    this.top_++;
    notifyAll();
  }

  public synchronized Object pop () {
    while (this.is_empty ())
      try { wait(); } catch (InterruptedException ex) {};
    this.top_--;
    Object return_object = this.stack_[this.top_];
    notifyAll();
    return return_object;
  }

  public synchronized Object top () {
    while (this.is_empty ())
      try { wait(); } catch (InterruptedException ex) {};
    return this.stack_[this.top_ - 1];
  }

  protected boolean is_empty () { return this.top_ == 0; }
  protected boolean is_full () {
    return this.top_ == this.max_;
  }
}
```


Main Application

```
class MT_Stack_App
{
  public static void main (String args[]) {
    if (args.length == 0)
      System.out.println (
        "usage: " + args[0] + " stacksize");
    else {
      Integer size = new Integer (args[0]);
      MT_Bounded_Stack stack =
        new MT_Bounded_Stack (size.intValue ());

      System.out.println (
        "starting up stack with size " + size);

      JoinableThreadGroup thread_group =
        new JoinableThreadGroup ("Producer/Consumer");

      new Thread (thread_group, new Producer (stack),
                  "Producer").start ();
      new Thread (thread_group, new Consumer (stack),
                  "Consumer").start ();
      try {
        thread_group.join ();
      } catch(InterruptedException ex) {
        System.out.println ("ThreadTest::main");
        System.out.println (ex);
      }
    }
  }
}
```


Producer and Consumer

```
class Producer implements Runnable {
  public Producer (MT_Bounded_Stack stack) {
    this.stack_ = stack;
  }
  public void run () {
    for (int count = 1; true; count++) {
      // Will block when stack is full.
      this.stack_.push (new Integer (count));
      System.out.println ("("
        + Thread.currentThread ().getName ()
        + ") pushed " + count);
    }
  }
  private MT_Bounded_Stack stack_;
}

class Consumer implements Runnable {
  public Consumer (MT_Bounded_Stack stack) {
    this.stack_ = stack;
  }
  public void run() {
    // Will block when stack is empty.
    for (;;) {
      System.out.println ("("
        + Thread.currentThread ().getName ()
        + ") popping " + this.stack_.pop ());
    }
  }
  private MT_Bounded_Stack stack_;
}
```

A Joinable ThreadGroup

```
class JoinableThreadGroup extends ThreadGroup
{
  public JoinableThreadGroup (String name)
  {
    super (name);
  }

  public JoinableThreadGroup (ThreadGroup parent,
                              String name)
  {
    super (parent, name);
  }

  // Wait for all the threads in the group to exit.
  public void join() throws InterruptedException
  {
    Thread list[] = new Thread[activeCount ()];

    enumerate (list, true);

    for (int i = 0; i < list.length; i++) {
      list[i].join();
    }
  }
}
```


Java Threading Model

- Unless you have better-than-average hardware, all the active threads in a Java application share the same CPU
  - This means that each runnable thread has to take turns executing for a while
  - A thread is *runnable* if it has been started but has not terminated, is not suspended, is not blocked waiting for a lock, and is not engaged in a `wait`
- When they are not running, runnable threads are held in priority-based scheduling queues managed by the Java run-time system

Java Threading Topics

- *Thread construction*
- *Thread execution*
- *Thread control*
- *Scheduling*
- *Priorities*
- *Miscellaneous*
- *Synchronization*
- *Waiting and Notification*

Thread Construction Methods

- `Thread` accept various arguments as constructors

```
    // Constructors
public Thread();
public Thread(Runnable target);
public Thread(Runnable target, String name);
public Thread(String name);
public Thread(ThreadGroup group, Runnable target);
public Thread(ThreadGroup group,
              Runnable target, String name);
public Thread(ThreadGroup group, String name);

    // Mark thread as a daemon
public final void setDaemon(boolean on);
```

Thread Construction Methods (cont'd)

- The `String` name serves as an identifier for the `Thread`
  - Useful for tracing debugging
- The `ThreadGroup` is where the new `Thread` is placed
  - Used to implement security
    - ▷ *e.g.*, prevent threads from being stopped arbitrarily
  - Defaults to same group as `Thread` issuing constructor
    - ▷ *e.g.*, will nest in a tree-like fashion
  - Can serve as target for group `stop`, `suspend`, and `resume`

Thread Construction Methods (cont'd)

- The `start` method activates the thread
  - *i.e.*, will call the `run` hook (defined by the user)
- The `setDaemon` method allows a thread to be terminated by the JVM when all other non-daemon threads have exited
  - Used for "background jobs"
  - Must be called before thread is started

Thread Control Methods

- Java defines methods for controlling threads:

```
public static Thread currentThread();
public void destroy();

public isAlive();
public void run();
public void start();

public final void stop();
public final void stop(Throwable obj);
```


Thread Control Methods (cont'd)

- The `start` method causes a thread to call its `run` hook
  - No synchronization locks held by the caller thread are automatically retained
- The `run` hook should be defined by a user to perform the desired task
  - The default behavior of `run` is to invoke the `run` method of the `Thread`'s runnable target (if it's not **null**)
  - A thread terminates when the `run` method returns
    - ▷ Unless it is stoped, an unhandled exception is thrown, or if `System.exit` is called
- The `isAlive` method returns `true` if a thread has started but not terminated
  - It will also return `true` if the thread is suspended

Thread Control Methods (cont'd)

- The `stop` method irrevocably terminates a thread
  - It does not delete the **Thread** object, just stops the activity
    - ▷ Thus, you can call **start** again on the same **Thread** object
  - You can call **stop(Throwable)** to stop a thread by throwing the listed exception
  - When a thread is stopped, it releases all locks held by objects running in the thread
- The `destroy` method stops and kills a thread without giving it or the Java runtime system any chance to intervene
  - Not recommended for routine use

Scheduling Methods

- Java allows you to schedule threads explicitly

```
public final void join();
public final void join(long millis);
public final void join(long millis, int nanos);
public void interrupt();

public final void resume();
public final void suspend();

public static void sleep(long millis);
public static void sleep(long millis, int nanos);

public static void yield();
```


Scheduling Methods (cont'd)

- Threads can synchronize with the termination of other threads via `join`
- The `join` method suspends the caller until the target thread completes
  - *i.e.*, it returns when **isAlive** is **false**
  - The version with a (millisecond) time argument returns control even if the thread has not completed within the specified time limit

Scheduling Methods (cont'd)

- The `suspend` method temporarily halts a thread
  - Beware of using this -- it can be dangerous if the thread being suspended holds JVM resources...
- The `resume` method allows a suspended thread to continue normally
- The `sleep` method causes the thread to suspend for a given time (specified in milliseconds) and then automatically resume
  - The thread might not continue immediately after the given time if there are other active threads

Scheduling Methods (cont'd)

- The `interrupt` method causes a `sleep`, `wait`, or `join` to abort with an `InterruptedException`
  - This can be caught and dealt with in an application-specific way
  - The `interrupt` method itself is not fully implemented in Java 1.0.
- The `yield` method relinquishes control, which may enable one or more other threads of equal priority to be run

Priority Methods

- Every thread has a priority
  - Threads with higher priority are executed in preference to threads with lower priority
- A `Thread` inherits priorities from the `Thread` that created it
- Priorities can be changed by calling `setPriority` with an argument between `MIN_PRIORITY` and `MAX_PRIORITY`
  - The maximum thread priority can be limited by the `ThreadGroup` to which the thread belongs

```
    // Fields
public final static int MAX_PRIORITY;
public final static int MIN_PRIORITY;
public final static int NORM_PRIORITY;

    // Methods
public final void setPriority(int newPriority);
public final int getPriority();
```


Priority Methods (cont'd)

- If there are multiple runnable threads at any given time, the Java run-time system picks one with the highest priority to run
- If there are more than one thread with the highest priority, it picks any arbitrary one of them
  - *i.e.*, Java does not strictly require *fairness*
- A running lower-priority thread is *preempted* (artificially suspended) if a higher-priority thread needs to be run
  - This preemption need not occur immediately
  - Threads with equal priority are not necessarily preempted in favor of each other

Miscellaneous Methods

- There are also a number of miscellaneous `Thread` methods

```
{
      // Methods
   public static int activeCount();
   public void checkAccess();
   public int countStackFrames();

   public static void dumpStack();
   public static int enumerate(Thread tarray[]);
   public final String getName();
   public final ThreadGroup getThreadGroup();
   public static boolean interrupted();
   public final boolean isDaemon();
   public boolean isInterrupted();
   public final void setName(String name);
   public String toString();
}
```

Synchronization Methods

- Java guarantees that most access and assignment operations are *atomic* on primitive data (e.g., **char**, **short**, **int**)
  - *i.e.*, they will always work safely in multithreaded contexts without explicit synchronization
- Primitive operations include access and assignment to built-in scalar types *except* `long` and `double`
  - Without explicit synchronization, concurrent assignments to `long` and `double` variables are allowed to be interleaved

Synchronization Methods (cont'd)

- Synchronization is implemented by exclusively accessing the underlying and otherwise inaccessible internal mutex `lock` associated with each Java `Object`
  - This includes `Class` objects for `statics`
- Each lock acts as a counter
  - If the count value is not zero on entry to a synchronized method or block because another thread holds the lock, the current thread is delayed (*blocked*) until the count is zero
  - On entry, the count value is incremented
  - The count is decremented on exit from each `synchronized` method or block, even if it is terminated via an exception
    - ▹ But not if the thread is destroyed...

Synchronization Methods (cont'd)

- Any method or code block marked as `synchronized` is executed in its entirety (unless explicitly suspended via `wait`) before the object is allowed to perform any other `synchronized` method called from any other thread
- Code in one `synchronized` method may make a self-call to another method in the same object without blocking
  - Similarly for calls on other objects for which the current thread has obtained and not yet released a lock
  - Only those calls stemming from other threads are blocked
- Synchronization is retained when calling an unsynchronized method from a synchronized one

Synchronization Methods (cont'd)

- If a method is *not* marked as `synchronized` then it may execute immediately whenever invoked
  - *i.e.*, even while another synchronized method is executing
- Thus, declaring a method as *synchronized* is `not` sufficient to ensure exclusive access
  - *i.e.*, any other unsynchronized methods may run concurrently with it
- The `synchronized` qualifier for methods can be overridden in subclasses
  - A subclass overriding a superclass method must explicitly declare it as `synchronized`
- Methods declared in Java `interfaces` cannot be qualified as synchronized

Synchronization Methods (cont'd)

- Individual code blocks within any Java method can be synchronized as follows

```
synchronized(anyObject)
{
  anyCode();
}
```

- In Java, *block synchronization* is considered to be a more basic construct than *method synchronization*
  - A `synchronized` method is equivalent to one that is not marked as synchronized but has all of its code contained within a `synchronized(this)` block

- Class-level `static` methods and blocks within `static` methods may be declared as `synchronized`
  - A non-`static` method can also lock `static` data via a code block enclosed by `synchronized(getClass())`

Waiting and Notification

- Java implements Monitors for all `Objects`

- The methods `wait`, `notify`, and `notifyAll` may be invoked only when the synchronization lock is held on their targets
  - This is normally ensured by using them only within methods or code blocks synchronized on their targets

- Compliance cannot usually be verified at compile time
  - A `IllegalMonitorStateException` occurs at run-time if you fail to comply

Waiting and Notification (cont'd)

- A `wait` invocation results in the following actions:

1. The current thread is suspended
2. The Java run-time system places the thread in an internal and otherwise inaccessible `wait` set associated with the target object
3. The synchronization lock for the target object is released (n times if it was acquired n times), but all other locks held by the thread are retained
   - In contrast, `suspended` threads retain `all` their locks

Waiting and Notification (cont'd)

- A `notify` invocation results in the following actions:

1. If one exists, an arbitrarily chosen thread, say T, is removed by the Java run-time system from the internal wait set associated with the target object
2. T must re-obtain the synchronization lock for the target object
   - This will *always* cause it to block at least until the thread calling `notify` releases the lock
   - It will continue to block if some other thread obtains the lock first
   - Once T acquires the lock the lock count is restored to the value n when T had locked the object originally
3. T is then resumed at the point of its wait

Waiting and Notification (cont'd)

- A `notifyAll` invocation works in the same way as `notify`
  - Except that the steps occur for *all* threads waiting in the wait set for the target object

- Two alternative versions of the `wait` method take arguments specifying the maximum time to wait in the wait set
  - If a timed wait has not resumed before its time bound, the thread behaves as if a **notify** had selected it from the set of waiting threads

- If an `interrupt` occurs during a `wait` the same `notify` mechanics apply
  - Except that control returns to the **catch** clause associated with the **wait** invocation.

Synchronization Examples

- Bounded counter interface

```
public interface BoundedCounter {
  // minimum allowed value
  public static final long MIN = 0;
  // maximum allowed value
  public static final long MAX = 5;

  // invariant: MIN <= value() <= MAX
  // initial condition: value() == MIN
  public long value();

  // increment only when value() < MAX
  public void inc();
  // decrement only when value() > MIN
  public void dec();
}
```


Synchronization Examples (cont'd)

- Synchronized bounded counter

```
public class BoundedCounterVI
  implements BoundedCounter
{
  protected long count_ = MIN;

  public synchronized long value() { return count_; }

  public synchronized void inc()  {
    while (count_ == MAX)
      try { wait(); } catch(InterruptedException ex) {};
    if (count_++ == MIN)
      notifyAll(); // signal if was bottom
  }

  public synchronized void dec()  {
    while (count_ == MIN)
      try { wait(); } catch(InterruptedException ex) {};
    if (count_-- == MAX)
      notifyAll(); // signal if was top
  }
}
```


Synchronization Examples (cont'd)

- Synchronized bounded counter

```
public class BoundedCounterV1
  implements BoundedCounter
{
  protected long count_ = MIN;

  // Note that long values require
  // synchronization.
  public synchronized long value() {
    return count_;
  }

  public synchronized void inc()  {
    awaitIncrementable();
    setCount(count_ + 1);
  }

  public synchronized void dec()  {
    awaitDecrementable();
    setCount(count_ - 1);
  }
```

```
  protected synchronized void setCount(long newValue)
  {
    count_ = newValue;
    // wake up any thread depending on new value
    notifyAll();
  }
  protected synchronized void awaitIncrementable() {
    while (count_ >= MAX)
      try { wait(); } catch(InterruptedException ex) {};
  }

  protected synchronized void awaitDecrementable() {
    while (count_ <= MIN)
      try { wait(); } catch(InterruptedException ex) {};
  }
}
```


Synchronization Examples (cont'd)

- Synchronized bounded counter (using sub-classing)

```
// No synchronization.
public class GroundCounter
{
  GroundCounter (long value) {
    value_ = value;
  }

  // Methods are *not* synchronized.
  public long value_() { return value_; }

  public void inc_() {
    ++value_;
  }

  public void dec_() {
    --value_;
  }

  private long value_;
}
```



```
// Subclass adds synchronization.
public class BoundedCounterVSC
  extends GroundCounter
  implements BoundedCounter
{
  public BoundedCounterVSC() {
    super (MIN);
  }

  public synchronized long value() {
    return value_();
  }

  public synchronized void inc()   {
    while (value_() >= MAX)
      try { wait(); } catch(InterruptedException ex) {};
    inc_ ();
    notifyAll();
  }

  public synchronized void dec()   {
    while (value_() <= MIN)
      try { wait(); } catch(InterruptedException ex) {};
    dec_ ();
    notifyAll();
  }
}
```


Synchronization Examples (cont'd)

- Synchronized bounded counter (using delegation)

```
// No synchronization.
public class BareCounter
{
  BareCounter (long value) {
    if (value > BoundedCounter.MAX)
      value = BoundedCounter.MAX;
    else if (value < BoundedCounter.MIN)
      value = BoundedCounter.MIN;
    value_ = value;
  }

  // Methods are *not* synchronized.
  public long value() { return value_; }

  public void add(int value) { value_ += value; }
  public void sub(int value) { add (-value); }

  private long value_;
}
```

```
// Adapter adds synchronization.
public class BoundedCounterVD
  implements BoundedCounter
{
  // fixed, unique
  private BareCounter delegate_;

  public BoundedCounterVD() {
    delegate_ = new BareCounter(MIN);
  }

  public synchronized long value() {
    return delegate_.value();
  }

  public synchronized void inc()   {
    while (delegate_.value() >= MAX)
      try { wait(); } catch(InterruptedException ex) {};
    delegate_.add(1);
    notifyAll();
  }

  public synchronized void dec()   {
    while (delegate_.value() <= MIN)
      try { wait(); } catch(InterruptedException ex) {};
    delegate_.sub(1);
    notifyAll();
  }
}
```


Synchronization Examples (cont'd)

- Synchronized bounded counter (via "external notification")

```
public class BoundedCounterVNL
  implements BoundedCounter
{
  private NotifyingLong c_ = new NotifyingLong(this, MIN);

  public synchronized long value() {
    return c_.value();
  }

  public synchronized void inc()  {
    while (c_.value() >= MAX)
      try { wait(); } catch(InterruptedException ex) {};
    c_.setValue(c_.value()+1);
  }

  public synchronized void dec()  {
    while (c_.value() <= MIN)
      try { wait(); } catch(InterruptedException ex) {};
    c_.setValue(c_.value()-1);
  }
}
```



```
// Generic notification mechanism
public class NotifyingLong
{
  private long value_;
  private Object observer_;

  public NotifyingLong (Object o, long v) {
    observer_ = 0;
    value_ = v;
  }

  public synchronized long value () {
    return value_;
  }

  public void setValue (long v) {
    synchronized (this) {
      value_ = v;
    }
    synchronized (observer_) {
      observer_.notifyAll ();
    }
}
```


Synchronization Examples

- Synchronized bounded counter (uses a state machine)
  - Beware of interactions between state machines, inheritance, and synchronization...

```
public class BoundedCounterVSW
implements BoundedCounter
{
  static final int BOTTOM = 0;
  static final int MIDDLE = 1;
  static final int TOP    = 2;

  // the state variable
  protected int state_ = BOTTOM;
  protected long count_ = MIN;

  public synchronized long value() {
    return count_;
  }

  public synchronized void inc()  {
    while (state_ == TOP)
      try { wait(); } catch(InterruptedException ex) {};
    ++count_;
    checkState();
  }
```

```
  public synchronized void dec()  {
    while (state_ == BOTTOM)
      try { wait();} catch(InterruptedException ex) {};
    --count_;
    checkState();
  }

  protected synchronized void checkState() {
    int oldState = state_;
    if      (count_ == MIN) state_ = BOTTOM;
    else if (count_ == MAX) state_ = TOP;
    else                    state_ = MIDDLE;

    if (state_ != oldState &&
        (oldState == TOP || oldState == BOTTOM))
       notifyAll();
  }
}
```