Monitors in Java

Model and Examples

Threads

A “thread”
  - represents an independent activity
    - animations (one thread is performing the animation while another thread is responding to the user)
    - servers (separate threads are created for each client)
  - is separately scheduled by the system
  - can manipulated by the program
  - coexists with other threads in the same address space
Motivations for Using Threads

- performance (on multiprocessor machines)
- simplicity (direct modeling of autonomous events)
- availability (unblocked threads can make progress)
- controllability (start/stop threads as needed)
- asynchrony (thread can block without halting program)
- required (by some Java services)

Classical Problems

Producer - Consumer

Producer → Buffer ← Consumer

Reader-Writer

Writer → Database ← Reader
Java as a Concurrency Programming Language

- Language features:
  - threads class and synchronization constructs
  - platform independent
- Libraries for basic network programming
  - sockets/URL
  - Remote Method Invocation (RMI)
- Used to implement distributed agent systems
  - Aglets
  - Voyager
  - Odyssey

Java Threads

Thread class:
- extends Object class
- implements Runnable interface

Attributes of a thread:
- target - in what object it will begin execution
- name - for identification by program
- group - of which it is a member
- priority - for scheduling
Thread Basics

Operations performed on a thread:
- start - begin execution
- stop - end execution (deprecated)
- suspend - await resumption (deprecated)
- resume - awake suspended thread (deprecated)
- interrupt - cause exception to be thrown

Operations performed by a thread:
- sleep - dormant for specified time period
- yield - allow scheduler to select among ready threads
- join - await completion of another thread
- wait - delay until notified
- notify/notifyAll - awaken one/all waiting threads

Application-Specific Threads

There are two ways of defining a thread to perform application specific activities by creating a class that:

- extends the Thread class using inheritance
- implements the Runnable interface

In each case, the application specific class defines a run() method where execution of the thread begins.
**Extending the Thread Class**

```java
public class Worker extends Thread {
    ...
    public void run() //defines where thread will begin
    {
        // code for worker
    }
}

public class Boss {
    private Worker Worker;
    ...
    public startWorker() {
        Worker = new Worker();
        Worker.start();
    }
}
```

**Implementing the Runnable Interface**

```java
public class Worker implements Runnable {
    ...
    public void run() //defines where thread will begin
    {
        // code for worker
    }
}

public class Boss {
    private thread workerThread;
    ...
    public startWorker() {
        Worker worker = new Worker();
        workerThread = new Thread(worker);
        workerThread.start();
    }
}
```
Forms of Synchronization

- mutual exclusion - preventing concurrent access to shared objects to preserve the consistency of the object

- condition synchronization - blocking attempted operations on a shared object until that object is in a state where the operation will preserve the consistency of the object (Monitor model)

Mutual Exclusion

The synchronized keyword can be used to:

- provide mutual exclusion among methods of the same object
- provide mutually exclusive ownership of an object

Mutual exclusion is needed to insure the consistency of the state of objects that can be accessed by multiple threads.
Synchronized Methods

```java
public class Value {
    private int current;

    synchronized public void increment() {
        current = current + 1;
    }

    synchronized public void decrement() {
        current = current - 1;
    }

    public int current() {
        return current;
    }
}
```

Mutually Exclusive Ownership

Employee worker; // shared object

```
Thread 1
  synchronized(worker) {
    // access and output
    // worker information
  }

Thread 2
  synchronized(worker) {
    // update worker
    // information
  }
```
Monitors in Java

- every object has a single lock and a waiting queue
- synchronized methods acquire the lock before executing
- the wait() operation suspends the executing thread on the object’s waiting queue and releases the object’s lock
- the notify() operation awakens exactly one thread suspended on the object’s waiting queue but does not release the lock
- the notifyAll() operation awakens all threads suspended on the object’s waiting queue but does not release the lock
- wait, notify, and notifyAll must be in synchronized methods
- awakened threads must reacquire the lock before continuing
wait() Operation

notify() Operation
**notifyAll() Operation**

- **acquire**: Obtains the lock.
- **lock**: Represents the object.
- **waiting queue**: Queue of waiting threads.
- **awaken**: wakes up waiting threads.
- **object**: The object method is called on.

---

**Generic Monitor Code**

```java
public class MonitorClass {
    ...
    // private data
    
    public synchronized void enter(...) {
        ...
        while (!condition) // test for desired condition
            wait(); // block execution
        ...
        // continue here when notified
    }

    public synchronized void change(...) {
        ...
        // change the object’s state
        notify(); // unblock any single waiter
        ...
        notifyAll(); // unblock all waiters
    }
    ...
}
```

---
Semaphore Example

// file Semaphore.java
// note: some details missing
public class Semaphore
{
    private int count;
    public Semaphore(int initial)
    {
        count = initial;
    }

    synchronized public void P()
    {
        count = count - 1;
        if(count < 0) wait();
    }

    synchronized public void V()
    {
        count = count + 1;
        if (count <= 0) notify();
    }
}

Readers-Writers Example

public class ReadersWriters //some details missing
//see Wait Exceptions
{
    private boolean writing;
    private int readers;
    public ReadersWriters()
    {
        writing = false;
        readers = 0;
    }

    public synchronized void startRead()
    {
        while (writing) wait();
        readers = readers + 1;
    }

    public synchronized void endRead()
    {
        readers = readers - 1;
        if (readers == 0) notify();
    }

    public synchronized void startWrite()
    {
        while (writing || readers > 0) wait();
        writing = true;
    }

    public synchronized void endWrite()
    {
        writing = false;
        notifyAll();
    }
}
Wait Exceptions

The wait operation is defined to return an exception if the wait is terminated abnormally. So the wait must be written as follows.

```java
public class MonitorClass {
    public synchronized void enter(...) {
        ...
        while (! condition) // test for desired condition
        { try { wait(); } // block execution
            catch(InterruptedException ie) {
            // handle wait exception
            ... // continue here when notified
            }
        }
    ...
}
```

The wait operation is defined to return an exception if the wait is terminated abnormally. So the wait must be written as follows.

Timed Waits

The time to wait for a synchronization condition can be bounded.

```java
public class Resource {
    private boolean available = true;
    ...

    public synchronized boolean timedAcquire(int maxTime) {
        if (! available)
            { try { wait(maxTime); } // block execution
                catch(InterruptedException ex) {
                return false;
            }
        available = false;
        return true;
    }
    ...
}
```