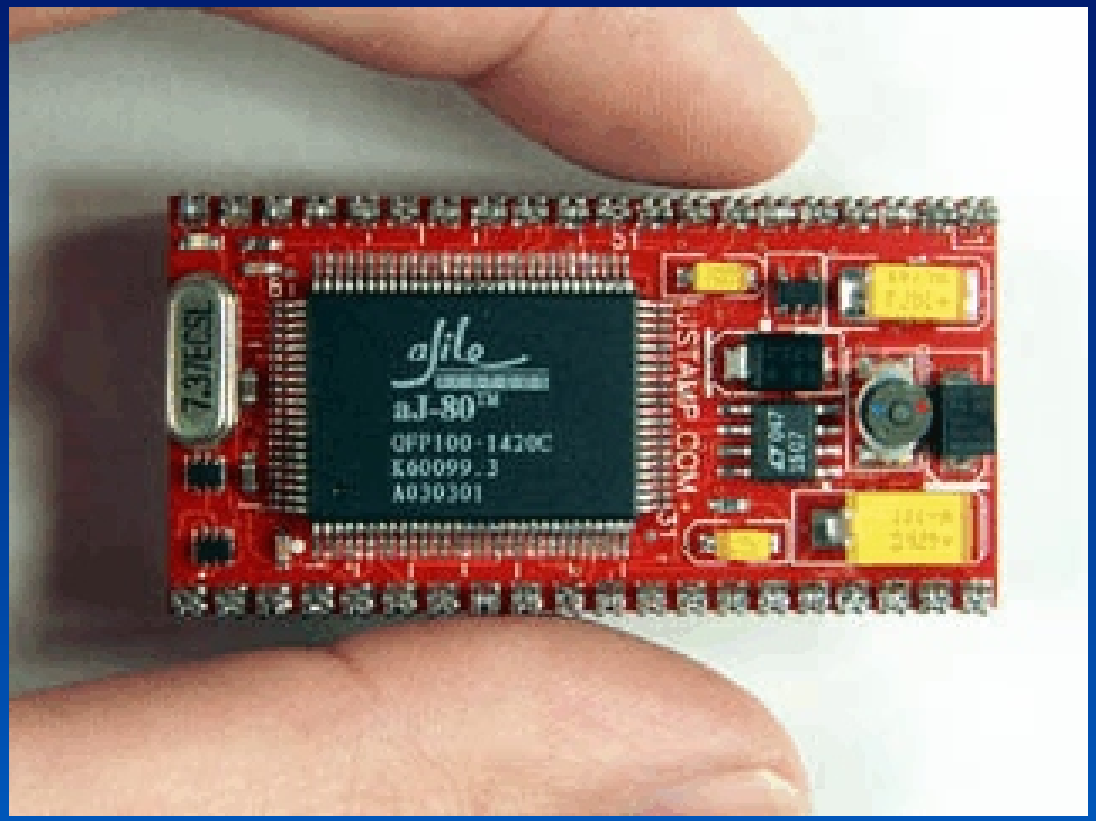
JStamp Threads

Using generic Java threads on JStamp/JStik/SaJe



Threads

- A thread is a path of program execution.
- Multi-threaded programming is one of the most powerful features of Java. Applied inappropriately, it can lead to an infinite number of difficult to diagnose bugs, and unpredictable program behavior.
- On a single-processor system, threads must always share that single processor, as well as all other system resources such as I/O pins, UARTs, etc. This is where problems can easily creep in.
- Threads also can “share” Java classes, easily leading to unexpected behavior.

Threads require native support

Threads cannot be tightly specified in any portable language

- Threads are one of the more vague aspects of Java. This is probably inevitable due to the wide variation of thread support in the underlying operating system and hardware on which a given JVM must execute.
- Java supports ten thread priorities (though the underlying OS may support more or less), RTSJ requires at least 32.
- For example, Solaris has 65535 thread priorities while Windows NT has seven. Win2000 and XP have 32. XP with INtime has 255. JStamp has 32.
- Thread support can use a mix of hard- and soft-ware.

Using Threads

Two approaches

- Extend Thread
- Implement Runnable

Class Thread

- Public class Thread
  - extends Object
  - implements Runnable
- Fields for priorities
- Constructors with and without Runnable object
- Methods - many
- Control methods
  - Thread.join, Thread.yield and Thread.sleep
  - but it's Object.wait and Object.notify

Extending Thread

more detailed example presented in class

```
public class GenericThread extends Thread {

    // you must override Thread.run() if you extend Thread
    public void run() {
        System.out.println ("Starting run method for " + aBlink.toString()
        + " at priority " + this.getPriority());
        for (;;) {
            aBlink.blinkPin();
        }
    }


}
```

Thread methods (1/3)

- void join() - wait for this thread to die
  - Throws InterruptedException - if another thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.
- void start() - begin execution.
  - JVM calls this thread's run() method
- void run() - sublcasses of Thread must override this.
  - If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns. There are few times when this is useful and even fewer when it's necessary or best.
  - Don't call this to start the thread - use start() instead.

Thread Methods (2/3)

- boolean isAlive()
  - true if thread has started and is not yet dead
- void setPriority(int)
  - Throws IllegalArgumentException - If the priority is not in the range MIN_PRIORITY to MAX_PRIORITY.
  - more on thread priorities in another slide
- int getPriority()
- void yield()
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Thread Methods (3/3)

- void sleep (long milliseconds)
  - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The thread does not lose ownership of any monitors.
  - Throws InterruptedException - if another thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.
  - See Object.notify()
  - Write (or obtain) a simple utility class to make this easy, for example com.systronix.util.Util.waitWhile(long howLong)

Priorities

- JStamp has 32 thread priorities
  - So does the RealTime Java spec...
  - 0 is low, 31 is high
  - generic (not RTJ) Java has 10 priorities
    - The default Java level of 5 equates to the JStamp level of 10
    - The 10 Java priorities are mapped to JStamp even priorities from 2 to 20
  - Some threads setup by the JStamp runtime system are [UC]:
    - Executive thread (priority 31) - Interrupts always disabled.
    - Serial thread for unloading internal UART FIFOs (priority 30).
      - Only present if serial I/O classes are used.
    - TCP/IP support threads (priority 25)
      - Only present if networking classes are used (so never on JStamp)
    - GC thread (priority 24)
    - Heap monitor threads (priority 12)
    - Idle thread (priority -1) - This priority is special and not available to users.

Threads vs Interrupts

Thread and interrupt priorities are completely independent

- There's no relationship between Thread and Interrupt priorities
- Any interrupt can interrupt any thread -- an arbitrarily high thread priority does not block interrupts.
  - It's easy to get this confused and think that a thread of priority 'n' will block an interrupt of lower priority < 'n'. It can't.

Thread execution

- Generic Java threads on JStamp are ‘cooperative’
  - They are not truly time-sliced (as on some systems such as TINI), where each thread is guaranteed a certain amount of time.
  - Thread execution can change at the “JSI Interval”.
    - The JSI interval (the default is 1000 usec) is set on the JemBuilder JVM0->aJ100 interrupts page.
    - Don’t make this arbitrarily short, it takes 1 usec on JStik (longer on Jstamp) to switch thread contexts
  - One thread can hog the controller forever (if no higher priority threads become ready)
  - A higher-priority thread can pre-empt an active lower-priority thread, but only at the JSI interval.
- Realtime threads
  - Realtime threads of a higher priority can interrupt non- realtime threads as soon as that realtime thread is ready to run

Synchronization

thread synchronization is (still) a fundamental problem

- The problem: conflicting access by multiple threads
  - Two or more threads could access the same object. One can be changing that object while others access it. Or two could be changing the same object at the same time.
- Easy solution: synchronization
  - A thread must obtain a lock on an object before it can execute any of its synchronized methods. Therefore no other thread can execute those methods at the same time.
  - This works for both instance and class methods
- Easy secondary problem: deadlock
  - Two threads are each waiting for the other to release a lock they need
  - Result: each waits forever and execution halts forever
  - This can be a nasty bug to troubleshoot

Wait and Notify

these are Object (not Thread) methods

- Objects (not threads) have locks
- Objects can contain a list of waiting threads
- CLDC Thread class has no methods interrupt() or interrupted()
  - Thread interruption is intended to wake up a blocking thread, not to suspend its execution as you might guess.
  - CLDC Thread objects can still throw an InterruptedException (how?)
- The contracts for wait() and notify() are rather complex
- wait(), wait (long timeout), wait(long timeout, int nanos)
  - timeout in msec and/or nsec
  - wait() is same as wait(0)

Object.wait()

public final void wait() throws InterruptedException

Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object. In other words this method behaves exactly as if it simply performs the call wait(0).

The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

This method should only be called by a thread that is the owner of this object's monitor. See the notify method for a description of the ways in which a thread can become the owner of a monitor.

Object.wait(long timeout)

public final void wait(long timeout) throws InterruptedException

Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. The current thread must own this object's monitor.

This method causes the current thread (call it T) to place itself in the wait set for this object and then to relinquish any and all synchronization claims on this object. Thread T becomes disabled for thread scheduling purposes and lies dormant until one of four things happens:

1) Some other thread invokes the notify method for this object and thread T happens to be arbitrarily chosen as the thread to be awakened.
2) Some other thread invokes the notifyAll method for this object.
3) The specified amount of real time has elapsed, more or less.
4) If timeout is zero, however, then real time is not taken into consideration and the thread simply waits until notified.

The thread T is then removed from the wait set for this object and re-enabled for thread scheduling. It then competes in the usual manner with other threads for the right to synchronize on the object; once it has gained control of the object, all its synchronization claims on the object are restored to the status quo ante - that is, to the situation as of the time that the wait method was invoked. Thread T then returns from the invocation of the wait method. Thus, on return from the wait method, the synchronization state of the object and of thread T is exactly as it was when the wait method was invoked.

The wait method, as it places the current thread into the wait set for this object, unlocks only this object; any other objects on which the current thread may be synchronized remain locked while the thread waits.

Object.notify()

public final void notify()

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.

The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:

1) By executing a synchronized instance method of that object.

2) By executing the body of a synchronized statement that synchronizes on the object.

3) For objects of type Class, by executing a synchronized static method of that class.

Only one thread at a time can own an object's monitor.

Object.notifyAll()

public final void notifyAll()

Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. See the notify method for a description of the ways in which a thread can become the owner of a monitor

Cautions

Don't rely on implementation specifics – and engineer your system!

- Much of threading is implementation dependent
  - It's easy to unintentionally code in assumptions about a given JVM implementation. Some effort and wrapper classes will help.
- There are no universal solutions
  - Threading by itself is complex. Add interrupts, realtime issues such as periodic threads, and other requirements of a given system (time granularity in a control loop for example) and it's easy to imagine how what's good for one system is not good for another.
  - Even fancy and costly tools won't think for you
- Be careful instrumenting threads
  - Most instrumentation adversely affects thread performance. Avoid System.out.print classes
  - Multiple threads can clobber or deadlock your instrumentation, too...

Some help with threads

Multi-threading is a fundamental problem so there is a lot of help available --or at least a lot of opinions about how to (or not to) do it right.

- JCSP (free)
  - http://www.cs.kent.ac.uk/projects/ofa/jcsp/
  - There's a mail list forum but it's not real active
  - Why isn't this a hotter topic?
- JCSP for networks (commercial)
  - http://www.quickstone.com/
- Taming Java Threads (book w/software)
  - http://www.holub.com/software/taming.java.threads.html
- Many other texts:
  - Concurrent Programming in Java(TM): Design Principles and Pattern (2nd Edition) by Doug Lea
  - Effective Java Programming Language Guide by Joshua Bloch

References

[AJTRM] *aJ-100TM Reference Manual Version 2.1* - Dec 06, 2001

[UC] *Unpublished correspondence* - Systronix and aJile 2002, 2003

[IEEE] *C# and the .NET Framework: Ready for Real Time?* - http://www.computer.org/software/homepage/2003/s1lap.htm

[CLDC] *CLDC Library API Specification 1.0* - Sun Microsystems 2000