Oracle Blog

David Holmes' Weblog

Technical Discussions

Main | Priorities, Scheduli... »

Inside the Hotspot VM: Clocks, Timers and Scheduling Events - Part I - Windows

By davidholmes on Oct 02, 2006

Clocks and Timers - General Overview

There are two kinds of time related devices in a system:

- a means to read a time value; and
- a means to schedule/trigger a time related event

The term "clock" is often used to refer to both of these because you read the time from a clock, and you can (often) set an alarm to have something happen when the clock reaches a certain time. Sometimes the term "passive clock" is used for a time source that can only be read, while "active clock" is used for one that can also trigger an event/interrupt/alarm. For the sake of discussion (and because they are different hardware devices) I will use the term "clock" to refer to the means by which a time value can be read, and the term "timer" for something that can trigger a time-related event.

To further complicate things both clocks and timers can be defined in absolute or relative terms. An absolute time relates to some external time

reference (such as UTC) while a relative time is just an elapsed interval. Systems typically have both an absolute clock (the time-of-day clock, with a low resolution) and a relative clock (some kind of "high-resolution" counter from which a "free-running" time can be calculated). Unfortunately, most systems do not have both absolute and relative timers. Instead they have to convert one form to the other. This is a problem because events related to absolute time need to be aware of changes in absolute time (eg the start/end of daylight savings time), while relative times should be immune to such changes. Invariably this means that there are problems (one way or the other) on most operating systems.

The resolution of clocks and timers in modern computers is typically very different. While time can be read at microsecond, or even better, resolution; timed events are triggered by operating system controlled interrupts that are normally much more coarse grained (typically 10ms). Depending on the platform the interrupt rate may, or may not, be configurable by application programs. Further, systems often have a quite different resolution for the time-of-day clock (often low resolution of 10ms or worse) and the free-running relative clock (microseconds or better), because they are actually derived from quite different pieces of hardware.

The different resolutions between clocks and timers makes things difficult when you want to try and measure the resolution of your timer using your clock (or indeed to measure any kind of execution time). Ideally you want the resolution of the clock to be a few orders of magnitude finer than that of the thing being measured. So to measure execution times in microseconds you need a clock with nanosecond resolution - so don't use a time-of-day clock to do this! But depending on your system you simply might not have a clock with the desired high resolution.

Java Programming API's for Clocks and Timers

The absolute "time-of-day" clock is represented by the `System.currentTimeMillis()` method, that returns a millisecond representation of wall-clock time in milliseconds since the epoch. As such it uses the operating system's "time of day" clock. The update resolution of this clock is often the same as the timer interrupt (eg. 10ms), but on some systems is fixed, independent of the interrupt rate.

The relative-time clock is represented by the `System.nanoTime()` method that returns a "free-running" time in nanoseconds. This time is useful only for comparison with other `nanoTime` values. The `nanoTime` method uses the highest resolution clock available on the platform, and while its return value is in nanoseconds, the update resolution is typically only microseconds. However, on some systems there is no choice but to use the same clock source as for `currentTimeMillis()` - fortunately this is rare and mostly affects old Linux systems, and Windows 98.

You should always try to use `nanoTime` to do timing measurement or calculation (and yes there are JDK API's that don't do this), in the hope that it will have a better resolution than `currentTimeMillis`.

In Java, time-triggered events are performed (at the lowest-level) through methods like `Object.wait(millis, nanos)`, `Thread.sleep(millis, nanos)` and the `java.util.concurrent.locks.LockSupport.park` methods. These latter methods underpin the concurrency utilities. Although there are issues with the legacy methods (`wait`, `sleep`) rounding (up or down) the millisecond value based on the nanosecond value, ultimately the times are passed through to the platform specific mechanism for doing a timed "wait" - and the same mechanism is typically used by all methods (primarily for consistent support of thread interruption). On Solaris and Linux these calls take time structures that allow microsecond, or nanosecond, values to be passed - but there is no guarantee of that resolution being achieved: In fact we know it won't be because of the typical timer interrupt rate of 10ms - so the operating system applies its own rounding algorithms internally to those calls. In contrast the Windows mechanism (`waitForSingleObject` and `waitForMultipleObjects`) only accepts millisecond timeout values, so there is no choice but to apply rounding at the Java or VM level.

There are higher-level API's that support frameworks for scheduling time-trigerred events. The `java.util.Timer` and `java.util.TimerTask` classes provide one such API - supporting scheduling of tasks at a fixed rate, or fixed delay, and with both absolute or relative start times. These classes combine the low-level timed wait routines with use of `currentTimeMillis()` to determine how long to wait. Consequently, they work best with tasks that have a delay/rate that is a number of multiples of 10ms (or 15ms depending on your platform).

Java 5.0 introduced the `java.util.concurrent` package and one of the concurrency utilities therein is the `ScheduledThreadPoolExecutor` (STPE) which is a thread pool for repeatedly executing tasks at a given rate or delay. It is effectively a more versatile replacement for the `java.util.Timer`/`TimerTask` combination, as it allows multiple service threads, accepts various time units, and doesn't require subclassing `TimerTask` (just implement `Runnable`). Configuring STPE with one thread makes it equivalent to the basic `java.util.Timer`. It is generally recommended to adopt STPE to replace uses of `Timer`/`TimerTask`. STPE (ultimately) makes use of the timed `park` methods and use of `nanoTime` and so is in a better position to support task rates/delays that are not 10ms/15ms multiples.

NOTE: Unless timed-waits habitually return early you should always expect a jitter in releases times on the order of one clock tick. In the original version of this blog I reported an apparent anomaly with STPE where it had bad release jitter. I had assumed the problem was with STPE because a simpler program using `nanoTime` and `Thread.sleep` showed almost zero jitter. Upon further investigation, experimenting on a completely different Windows platform (ES 2003 on dual-processor opteron vs. Windows 2000 on a PIII

laptop), I found they performed the same. It seems that the result on the laptop for the simpler program is the questionable one. This highlights the problems surrounding clocks and timers on Windows and shows that still further investigation is needed.

Clocks and Timers on Windows

Windows use of clocks and timers varies considerably from platform to platform and is plagued by problems - again this isn't necessarily Window's fault, just as it wasn't the VM's fault: the hardware support for clocks/timers is actually not very good - the references at the end lead you to more information on the timing hardware available. The following relates to the "NT" family (win 2k, XP, 2003) of Windows.

There are a number of different "clock" API's available in Windows. Those used by Hotspot are as follows:

- `System.currentTimeMillis()` is implemented using the `GetSystemTimeAsFileTime` method, which essentially just reads the low resolution time-of-day value that Windows maintains. Reading this global variable is naturally very quick - around 6 cycles according to reported information. This time-of-day value is updated at a constant rate regardless of how the timer interrupt has been programmed - depending on the platform this will either be 10ms or 15ms (this value seems tied to the default interrupt period).
- `System.nanoTime()` is implemented using the `QueryPerformanceCounter`/`QueryPerformanceFrequency` API (if available, else it returns `currentTimeMillis\*10\^6`). `QueryPerformanceCounter`(QPC) is implemented in different ways depending on the hardware it's running on. Typically it will use either the programmable-interval-timer (PIT), or the ACPI power management timer (PMT), or the CPU-level timestamp-counter (TSC). Accessing the PIT/PMT requires execution of slow I/O port instructions and as a result the execution time for QPC is in the order of microseconds. In contrast reading the TSC is on the order of 100 clock cycles (to read the TSC from the chip and convert it to a time value based on the operating frequency). You can tell if your system uses the ACPI PMT by checking if `QueryPerformanceFrequency` returns the signature value of 3,579,545 (ie 3.57MHz). If you see a value around 1.19Mhz then your system is using the old 8245 PIT chip. Otherwise you should see a value approximately that of your CPU frequency (modulo any speed throttling or power-management that might be in effect.)

  The default mechanism used by QPC is determined by the Hardware Abstraction layer(HAL), but some systems allow you to explicitly control it using options in `boot.ini`, such as `/usepmtimer` that explicitly requests use of the power management timer. This default changes not only across hardware but also across OS versions. For example Windows XP Service

> Pack 2 changed things to use the power management timer (PMTimer) rather than the processor timestamp-counter (TSC) due to problems with the TSC not being synchronized on different processors in SMP systems, and due the fact its frequency can vary (and hence its relationship to elapsed time) based on power-management settings. (The issues with the TSC, in particular for AMD systems, and how AMD aims to provide a stable TSC in future processors is discussed in Rich Brunner's article referenced below. You can also read how the Linux kernel folk have abandoned use of the TSC until a new stable version appears in CPUs.)

The timer related API's for doing timed-waits all use the `waitForMultipleObjects` API as previously mentioned. This API only accepts timeout values in milliseconds and its ability to recognize the passage of time is based on the timer interrupt programmed through the hardware.

Typically a Windows machine has a default 10ms timer interrupt period, but some systems have a 15ms period. This timer interrupt period may be modified by application programs using the `timeBeginPeriod`/`timeEndPeriod` API's. The period is still limited to milliseconds and there is no guarantee that a requested period will be supported. However, usually you can request a 1ms timer interrupt period (though its accuracy has been questioned in some reports). The hotspot VM in fact uses this 1ms period to allow for higher resolution `Thread.sleep` calls than would otherwise be possible. The sample `Sleeper.java` will cause this higher interrupt rate to be used, thus allowing experimentation with a 1ms versus 10ms period. It simply calls `Thread.sleep(Integer.MAX_VALUE)` which (because it is not a multiple of 10ms) causes the VM to switch to a 1ms period for the duration of the sleep - which in this case is "forever" and you'll have to ctrl-C the "java Sleeper" execution.

```
public class Sleeper {
  public static void main(String[] args) throws Throwable {
    Thread.sleep(Integer.MAX_VALUE);
  }
}
```

You can see what interrupt period is being used in Windows by running the `perfmon` tool. After you bring it up you'll need to add a new item to watch (click the `+` icon above the graph - even if it appears grayed/disabled). Select the interrupts/sec items and add it. Then right click on interrupts/sec under the graph and edit its properties. On the "data" tab, change the "scale" to 1 and on the graph tab, the vertical max to be 1000. Let the system settle for a few seconds and you should see the graph drawing a steady line. If you have a 10ms interrupt then it will be 100, for 1ms it will be 1000, for 15ms it will be 66.6, etc. Note: on a multiprocessor system show the interrupts/sec for each processor individually, not the total - one processor will be fielding the timer interrupts.

Note that any application can change the timer interrupt and that it affects the whole system. Windows only allows the period to be shortened, thus ensuring that the shortest requested period by all applications is the one that is used. If a process doesn't reset the period then Windows takes care of it when the process terminates. The reason why the VM doesn't just arbitrarily change the interrupt rate when it starts - it could do this - is that there is a potential performance impact to everything on the system due to the 10x increase in interrupts. However other applications do change it, typically multi-media viewers/players. Be aware that a browser running the JVM as a plug-in can also cause this change in interrupt rate if there is an applet running that uses the `Thread.sleep` method in a similar way to `Sleeper`.

Further note, that after Windows suspends or hibernates, the timer interrupt is restored to the default, even if an application using a higher interrupt rate was running at the time of suspension/hibernation.

Conclusions

If you are interested in measuring absolute time then always use `System.currentTimeMillis()`. Be aware that its resolution may be quite coarse (though this is rarely an issue for absolute times.)

If you are interested in measuring/calculating elapsed time, then always use `System.nanoTime()`. On most systems it will give a resolution on the order of microseconds. Be aware though, this call can also take microseconds to execute on some platforms.

If you are performing timed waits of any kind be aware that the only (near) portable resolution you should expect is 10ms. If you can limit yourself to Windows and are prepared to change the timer interrupt period then you can get a resolution of around 1ms out of the low-level wait routines. But be prepared for jitter on the order of a clock-tick. If you find that a framework like STPE doesn't seem "accurate" enough for your purposes, then you could try a simpler approach such as writing your own one-shot task repeater, something like:

```
public class TaskRepeater {  // code sketch only
    final long delay;  // delay between end of one execution and start of next
    final Runnable task;
    public TaskRepeater(Runnable r, long millisecondDelay) {
        task= r;
        delay = millisecondDelay;
    }

    volatile boolean terminated = false;
    final Thread worker = new Thread() {
        public void run() {
            while (!terminated) {
                try {
```

```
                    task.run();
                }
                catch(Throwable t) {
                    // log it or whatever, or let it escape and kill the worker
                }
                try {
                    Thread.sleep(delay);
                }
                catch(InterruptedException ex) {
                   // just re-loop and check condition
                }
            }
        }
    };
    public void start() {
        worker.start();  // will throw if already started
    }
    public void shutdown() {
        terminated = true;
        worker.interrupt();
    }
}
```

This might, on a particular machine perform better.

Note:. you should execute the above with the `-XX:+ForceTimeHighResolution` VM option, which due to a flaw in its implementation actually disables the internal attempts to use the high-resolution timer for sleeps. Otherwise, if you set an interrupt period other than 1ms, the internal sleep implementation will change it to 1ms if the requested delay is not a multiple of 10ms.

Finally, for Windows users, particularly on dual-core or multi-processor systems (and it seems most commonly on x64 AMD systems) if you see erratic timing behaviour either in Java, or other applications (games, multi-media presentations) on your system, then try adding the `/usepmtimer` switch in your `boot.ini` file.

References

A good synopsis of the state of Windows timers is given at: http://www.gamedev.net/reference/programming/features/timing/

There is a good discussion of hardware related timer issues at http://www.microsoft.com/whdc/system/CEC/mm-timer.mspx

An excellent discussion of the problems using the TSC as a timer, particularly on AMD systems, and how AMD will address this in future processors, is given in an article by Rich Brunner (AMD Fellow): http://lkml.org/lkml/2005/11/4/173

Also read how the Linux kernel folk have abandoned the TSC: "Counting on

the time stamp counter"

There are a number of related Hotspot bug reports for clock/timer issues on Windows:

- 6313903 Thread.sleep(3) might wake up immediately on windows
- 5005837 rework win32 timebeginperiod usage
- 6435126 ForceTimeHighResolution switch doesn't operate as intended

Category: Sun

Tags: none

Permanent link to this entry

Main | Priorities, Scheduli... »
Comments:

Since Thread.stop() is deprecated, the only other remaining (non-deprecated) way to *generically* tell a thread to quit is by interrupting it, triggering an InterruptedException. Right? If so, then you needn't have a terminated flag; the interrupt status is sufficient.

Posted by **David Smiley** on December 29, 2006 at 10:19 PM EST #

You need the terminated flag in case the task itself consumes the interrupt by resetting the interrupt bit, or else swallowing the InterruptedException.

Posted by **David Holmes** on January 30, 2007 at 12:41 AM EST #

Is there a Part II of this article?
Thanks,
Nanjunda

Posted by **Nanjunda Somayaji** on October 16, 2007 at 09:54 PM EST #

are there any guidelines on the lengths o f the intervals one can reliably measure via the likes of nanotime? being free running it will not be adjusted for running fast or slow, so presumably as time passes... as it were ... the absolute value of the error longer and longer measured intervals increases yes? whereas if a systems timeofday clock is synced via NTP to a good timesource, the absolute value of the error of longer and longer measured intervals should remain pretty much fixed. assuming of course I've not charged off into the weeds in this set of assumptions...

Posted by **rick jones** on October 31, 2007 at 03:37 AM EST #

Apologies for the delayed response as I didn't get notified of the new comments.

Nanjunda: No, sorry, there is no part II. There are others far more expert than I that can explain time-keeping in Solaris and Linux - both of which are very complicated; and in the Linux case has undergone recent major change with the "high resolution timer" support.

Rick: The underlying timer devices have specifications that define their own allowed error rates (eg. HPET specification), but beyond that I don't know how to characterise the errors involved in using the different time sources.

Posted by **David Holmes** on January 01, 2008 at 11:54 AM EST #

Typo: "hence it's relationship" --> "hence its relationship"

Posted by **Trevor** on April 14, 2009 at 07:29 PM EST #

The Microsoft link has gone bad.

On a related note, are there any other tutorials on timer issues? I'm still a little foggy on the exact process of the operating system updating its timers in response to an interrupt, and how that affects accuracy and resolution. I'd like to read something with a more generic perspective. (This one tends to focus specifically on Windows' quirks.)

Posted by **Trevor** on April 14, 2009 at 07:45 PM EST #

Thanks for the typo (fixed) and bad link info. Hopefully the link problem is temporary as that doc is still referenced from a number of MS sites/docs.

As for other tutorials ... I don't think you'll find anything "generic" because this is fairly OS specific stuff. For Solaris there is a chapter in "Inside the Solaris OS". For Linux I expect there is also some kernel document describing timers - google "linux timer management" yields some interesting hits.

Posted by **David Holmes** on April 15, 2009 at 07:09 AM EST #

I am using java.util.Timer to schedule a TimerTask that schedules another TimerTask and so on according to predefined schedule. The schedule is not regular and actually has only three times to start a task during a day. I noticed that tasks are invoked with a delay approximately proportional to the time passed since they are scheduled. On Linux for 5 hours error exceeds 2 sec, 10 hours gives 4 sec. On PC/Windows the error is smaller - 5 hours gave 15 msec - but it also accumulates. I would understand an error in some limits defined by granularity of time measurment but this accumulation is strange to me. Do you have any information about it?

Posted by **Boris Shukhat** on April 27, 2009 at 05:00 PM EST #

In addition to my previous post, those delays actually happen in

Object.wait(long) and easily reproducable

Posted by **Boris Shukhat** on April 27, 2009 at 07:09 PM EST #

I can't comment specifically on what you have observed as I'd need to see how the timers are being scheduled and check how Timer is implemented. You may be better off posting to a mailing-list (such as concurrency-interest@cs.oswego.edu) or a Java forum.

Posted by **David Holmes** on April 29, 2009 at 04:41 AM EST #

This is very much useful information about java.util.Timer.
Really its very fruitful.

I also liked the following links regarding java job scheduling

Java Job Scheduling with java.util.Timer
http://jksnu.blogspot.com/2011/02/java-job-scheduling.html

Quartz Scheduling with JSP-Servlet
http://jksnu.blogspot.com/2011/03/quartz-framework-implementation-with.html

Quartz Scheduling with Spring Framework
http://jksnu.blogspot.com/

Posted by **Jitendra Kumar Singh** on June 18, 2011 at 01:18 PM EST #

Thank you for this article.
One thing I am looking for is the elapsed time independent of the computer clock. If I am planning to do something every minute, and the user adjusts his clock back by a year then my event wontt happen for a year if I simply use the computer clock. GetTickCount() under Windows has this effect, that the measured elapsed time will work properly even if the user of the PC plays with the PC clock. Am I right in that System.nanoTime() would also work this way?

Posted by **Jozsef Bekes** on January 16, 2013 at 10:47 PM EST #

Post a Comment:

- Name: guest
- E-Mail:
- URL:
- Notify me by email of new comments

- Remember Information?
- Your Comment:
- HTML Syntax: NOT allowed
- Please answer this simple math question

  8 + 36 =
- Preview Post

About

The views expressed on this blog are my own and do not necessarily reflect the views of Oracle.

Search

Enter search term:

Search only this blog

Recent Posts

- Parallel Classloading Revisited: Fully Concurrent Loading
- To Blog or Not to Blog
- Minimize Garbage Generation: GC is your Friend, not your Servant
- Real-time Java at OOPSLA 2009
- Real-time Java at OOPSLA 2008
- Roll up! Roll up! It's JavaOne time again and Real-Time is hitting the Big Time
- Priorities, Scheduling and Real-time
- Inside the Hotspot VM: Clocks, Timers and Scheduling Events - Part I - Windows

Top Tags

- concurrency

- java
- javaone
- jrts
- priority
- real-time
- rts
- rtsj
- scheduling

Categories

- Personal
- Sun

Archives

« March 2016

Sun	Mon	Tue	Wed	Thu	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

Today

Bookmarks

- blogs.sun.com
- java.com
- java.net
- opensolaris.org

Menu

- Blogs Home
- Weblog
- Login

Feeds

RSS

- All
- /Personal
- /Sun

- Comments

Atom

- All
- /Personal
- /Sun
- Comments