Recent Posts

Lots o' static

We Don't Need No Stinkin' Databases

Even Mo' Static

Custom-Tailored Configuration

Mo' Static

About...

Measuring Latency in Linux

For measuring latency in modern systems, we need to be able to measure intervals in microseconds at least, and preferably in nanoseconds or better. The good news is that with relatively modern hardware and software, it is possible to accurately measure time intervals as small as (some smallish number of) nanoseconds.

But, it's important to understand what you're measuring and what the different edge cases might be to ensure that your results are accurate.

TL;DR

The short version is that for best results you should be using:

- Linux kernel 2.6.18 or above – this is the first version that includes the hrtimers package. Even better is 2.6.32 or above, since this

includes support for most of the different clock sources.

- A CPU with a constant, invariant TSC (time-stamp counter). This means that the TSC runs at a constant rate across all sockets/cores, regardless of frequency changes made to the CPU by power management code. If the CPU supports the RDTSCP instruction, so much the better.

- The TSC should be configured as the clock source for the Linux kernel at boot time.

- You should be measuring the interval between two events that happen on the same machine (intra-machine timing).

- For intra-machine timing, your best bet is generally going to be to read the TSC directly using assembler. On my test machine it takes about 100ns to read the TSC from software, so that is the limit of this method's accuracy. YMMV, of course, which is why I've included source code that you can use to do your own measurements.

    - Note that the 100ns mentioned above is largely due to the fact that my Linux box doesn't support the RDTSCP instruction, so to get reasonably accurate timings it's also necessary to issue a CPUID instruction prior to RDTSC to serialize its execution. On another machine that supports the RDTSCP instruction (a recent MacBook Air), overhead is down around 14ns. More on that later...

The following sections will talk about how clocks work on Linux, how to access the various clocks from software, and how to measure the overhead of accessing them.

Intra-machine vs. Inter-machine Timings

However, before jumping into the details of the above recommendations, I want to talk a little about the different problems in intra-machine vs. inter-machine time measurements. Intra-machine timing is the simplest scenario, since it is generally pretty easy to ensure that you use the same clock source for all your

timing measurements.

The problem with inter-machine timing is that, by definition, you're dealing with (at least) two different clock sources. (Unless of course you are timing round-trip intervals – if that's the case, you're lucky). And the problem with having two clock sources is described somewhat amusingly by this old chestnut:

> *A man with a watch knows what time it is. A man with two watches is never sure.*
> — *Segal's Law*

For inter-machine timings, you're pretty much stuck with the CLOCK_REALTIME clock source (the source for gettimeofday), since you presumably need a clock that is synchronized across the two (or more) machines you are testing. In this case, the accuracy of your timing measurements will obviously depend on how well the clock synchronization works, and in all but the best cases you'll be lucky to get accuracy better than some small number of microseconds.[1]

We're not going to talk much more about inter-machine timing in this article, but may get into it another time.

How Linux Keeps Time

With that out of the way, let's take a look at how Linux keeps time. It starts when the system boots up, when Linux gets the current time from the RTC (Real Time Clock). This is a hardware clock that is powered by a battery so it continues to run even when the machine is powered off. In most cases it is not particularly accurate, since it is driven from a cheap crystal oscillator whose frequency can vary depending on temperature and other factors.[2] The boot time retrieved from the RTC is stored in memory in the kernel, and is used as an offset later by code that derives wall-clock time from the combination of boot time and the tick count kept by the TSC.

The other thing that happens when the system boots is that the TSC (Time Stamp Counter) starts running. The TSC is a register counter

that is also driven from a crystal oscillator – the same oscillator that is used to generate the clock pulses that drive the CPU(s). As such it runs at the frequency of the CPU, so for instance a 2GHz clock will tick twice per nanosecond.

There are a number of other clock sources which we'll discuss later, but in most cases the TSC is the preferred clock source for two reasons: it is very accurate, and it is very cheap to query its value (since it is simply a register). But, there are a number of caveats to keep in mind when using the TSC as a timing source.

- In older CPU's, each core had its own TSC, so in order to be sure that two measurements were accurate relative to each other, it was necessary to pin the measuring code to a single core.

- Also in older CPU's, the TSC would run at the frequency of the CPU itself, and if that changed (for instance, if the frequency was dynamically reduced, or the CPU stopped completely for power management), the TSC on that CPU would also slow down or stop. (It is sometimes possible to work around this problem by disabling power management in the BIOS, so all CPU's always run at 100% no more, no less).

  Both of these problems are solved in more recent CPUs: a *constant* TSC keeps all TSC's synchronized across all cores in a system, and an *invariant* (or *nonstop*) TSC keeps the TSC running at a fixed rate regardless of changes in CPU frequency. To check whether your CPU supports one or both, execute the following and examine the values output in flags:

```
$ cat /proc/cpuinfo | grep -i tsc
flags : ... tsc  rdtscp constant_tsc nonstop_tsc ...
```

The flags have the following meanings:

Flag	Meaning
tsc	The system has a TSC clock.
rdtscp	The RDTSCP instruction is available.

constant_tsc	The TSC is synchronized across all sockets/cores.
nonstop_tsc	The TSC is not affected by power management code.

Other Clock Sources

While the TSC is generally the preferred clock source, given its accuracy and relatively low overhead, there are other clock sources that can be used:

- The HPET (High Precision Event Timer) was introduced by Microsoft and Intel around 2005. Its precision is approximately 100 ns, so it is less accurate than the TSC, which can provide sub-nanosecond accuracy. It is also much more expensive to query the HPET than the TSC.
- The acpi_pm clock source has the advantage that its frequency doesn't change based on power-management code, but since it runs at 3.58MHz (one tick every 279 ns), it is not nearly as accurate as the preceding timers.
- jiffies signifies that the clock source is actually the same timer used for scheduling, and as such its resolution is typically quite poor. (The default scheduling interval in most Linux variants is either 1 ms or 10 ms).

To see the clock sources that are available on the system:

```
$ cat
/sys/devices/system/clocksource/clocksource0/available_cloc

tsc hpet acpi_pm
```

And to see which one is being used:

```
$ cat
/sys/devices/system/clocksource/clocksource0/current_clocks

tsc
```

Typically the clock source is set by the kernel automatically at boot time, but you can force a particular clock source by including the appropriate parameter(s) on the command line that boots Linux (e.g., in /boot/grub/grub.conf):

```
ro root=/dev/... clocksource=tsc
```

You can also change the clock source while the system is running e.g., to force use of HPET:

```
$ echo hpet >
/sys/devices/system/clocksource/clocksource0/current_clocks
```

The above discusssion refers to what I will call hardware clocks, although strictly speaking these clocks are a mixture of hardware and software. At the bottom of it all there's some kind of hardware device that generates periodic timing pulses, which are then counted to create the clock. In some cases (e.g., the TSC) the counting is done in hardware, while in others (e.g., jiffies) the counting is done in software.

Wall-Clock Time

The hardware (or hardware/software hybrid) clocks just discussed all have one thing in common: they are simply counters, and as such have no direct relationship to what most of us think of as time, commonly referred to as wall-clock time.

To derive wall-clock time from these counters requires some fairly intricate software, at least if the wall-clock time is to be reasonably accurate. What reasonably accurate means of course depends on how important it is (i.e., how much money is available) to make sure that wall-clock time is accurate.

The whole process of synchronizing multiple distributed clocks is hellishly complicated, and we're not going to go into it here. There

are many different mechanisms for synchronizing distributed clocks, from the relatively simple (e.g., NTP[3]) to the not-quite-so-simple (e.g., PTP[4]), up to specialized proprietary solutions[5].

The main point is that synchronizing a system's wall-clock time with other systems requires a way to adjust the clock to keep it in sync with its peers. There are two ways this can be done:

- Stepping is the process of making (one or more) discontinuous changes to the wall-clock component of the system time. This can cause big jumps in the wall-clock time, including backwards jumps, although the time adjustment software can often be configured to limit the size of a single change. A common example is a system that is configured to initialize its clock at boot time from an NTP server.

- Slewing (sometimes called disciplining) involves actually changing the frequency (or frequency multiplier) of the oscillator used to drive a hardware counter like the TSC. This can cause the clock to run relatively faster or slower, but it cannot jump, and so cannot go backwards.

Available Clock Sources

The most common way to get time information in Linux is by calling the gettimeofday() system call, which returns the current wall-clock time with microsecond precision (although not necessarily microsecond accuracy). Since gettimeofday() calls clock_gettime(CLOCK_REALTIME,), the following discussion applies to it as well.

Linux also implements the POSIX clock_gettime() family of functions, which let you query different clock sources, including:

CLOCK_REALTIME	Represents wall-clock time. Can be both stepped and slewed by time adjustment code (e.g., NTP, PTP).
	A lower-resolution version of

CLOCK_REALTIME_COARSE	CLOCK_REALTIME.
CLOCK_REALTIME_HR	A higher-resolution version of CLOCK_REALTIME. Only available with the real-time kernel.
CLOCK_MONOTONIC	Represents the interval from an abitrary time. Can be slewed but not stepped by time adjustment code. As such, it can only move forward, not backward.
CLOCK_MONOTONIC_COARSE	A lower-resolution version of CLOCK_MONOTONIC.
CLOCK_MONOTONIC_RAW	A version of CLOCK_MONOTONIC that can neither be slewed nor stepped by time adjustment code.
CLOCK_BOOTTIME	A version of CLOCK_MONOTONIC that additionally reflects time spent in suspend mode. Only available in newer (2.6.39+) kernels.

The availability of the various clocks, as well as their resolution and accuracy, depends on the hardware as well as the specific Linux implementation. As part of the accompanying source code for this article I've included a small test program (clocks.c) that when compiled[6] and run will print the relevant information about the clocks on a system. On my test machine[7] it shows the following:

```
clocks.c
                    clock              res (ns)
secs              nsecs
             gettimeofday                 1,000
1,391,886,268       904,379,000
```

```
           CLOCK_REALTIME                     1
1,391,886,268       904,393,224
    CLOCK_REALTIME_COARSE               999,848
1,391,886,268       903,142,905
          CLOCK_MONOTONIC                     1
136,612     254,536,227
      CLOCK_MONOTONIC_RAW           870,001,632
136,612     381,306,122
   CLOCK_MONOTONIC_COARSE               999,848
136,612     253,271,977
```

Note that it's important to pay attention to what clock_getres() returns – a particular clock source can (and does, as can be seen above with the COARSE clocks) sometimes return what may look like higher-precision values, but any digits beyond its actual precision are likely to be garbage. (The exception is gettimeofday – since it returns a timeval, which is denominated in micros, the lower-order digits are all zeros).

Also, the value returned from clock_getres() for CLOCK_MONOTONIC_RAW is clearly garbage, although I've seen similar results on several machines.

Finally, note that the resolution listed for CLOCK_REALTIME is close to, but not quite, 1 million – this is an artifact of the fact that the oscillator cannot generate a frequency of exactly 1000 Hz – it's actually 1000.15 Hz.

Getting Clock Values in Software

Next up is a brief discussion of how to read these different clock values from software.

Assembler

In assembler language, the RDTSC instruction returns the value of the TSC directly in registers edx:eax. However, since modern CPU's support out-of-order execution, it has been common practice to insert a serializing instruction (such as CPUID) prior to the RDTSC instruction in order to ensure that the execution of RDTSC is not

reordered by the processor.

More recent CPU's include the RDTSCP instruction, which does any necessary serialization itself. This avoids the overhead of the CPUID instruction, which can be considerable (and variable). If your CPU supports RDTSCP, use that instead of the CPUID/RDTSC combination.

C/C++

Obviously, the RDTSC instruction can be called directly from C or C++, using whatever mechanism your compiler provides for accessing assembler language, or by calling an assembler stub that is linked with the C/C++ program. (An example can be found at Agner Fog's excellent website).

Calling gettimeofday() or clock_gettime() is pretty straightforward – see the accompanying clocks.c source file for examples.

Java

Java has only two methods that are relevant to this discussion:

- System.currentTimeMillis() returns the current wall-clock time as the number of milliseconds since the epoch. It calls gettimeofday(), which in turn calls clock_gettime(CLOCK_REALTIME, ...).
- System.nanoTime returns the number of nanoseconds since some unspecified starting point. Depending on the capabilities of the system, it either calls gettimeofday(), or clock_gettime(CLOCK_MONOTONIC,).

The bad news is that if you need clock values other than the above in Java, you're going to need to roll your own, e.g. by calling into C via JNI. The good news is that doing so is not much more expensive than calling nanoTime (at least in my tests).

Overhead of Clock Queries

The Heisenberg Uncertainty Principle says, in a nutshell, that the act of observing a phenomenom changes it. A similar issue exists

with getting timestamps for latency measurement, since it takes a finite (and sometimes variable) amount of time to read any clock source. In other words, just because the TSC on a 2GHz machine ticks twice per nanosecond doesn't mean we can measure intervals of a nanosecond – we also need to account for the time it takes to read the TSC from software.

So, how expensive is it to perform these different clock queries? Included is some sample code that you can use to measure the time it takes to query various clock sources, from both C++ and Java (using JNI to call C code).

Both the C++ and Java versions take the same approach: call the particular clock function in a tight loop, and store the result. We do this a large number of times, and hang on to the results from the final iteration. This has the effect of allowing Java to do any jitting it needs to, and for both the C++ and Java versions to help ensure that code and data is in the processor's cache memory.

The results of running the test on my test machine are (all timings are in nanoseconds):

```
ClockBench.cpp
                  Method       samples     min     max
avg  median   stdev
           CLOCK_REALTIME       255       54.00   58.00
55.65   56.00    1.55
    CLOCK_REALTIME_COARSE       255        0.00    0.00
0.00    0.00    0.00
          CLOCK_MONOTONIC       255       54.00   58.00
56.20   56.00    1.46
      CLOCK_MONOTONIC_RAW       255      650.00 1029.00
690.35  839.50   47.34
   CLOCK_MONOTONIC_COARSE       255        0.00    0.00
0.00    0.00    0.00
              cpuid+rdtsc       255       93.00   94.00
93.23   93.50    0.42
                    rdtsc       255       24.00   28.00
25.19   26.00    1.50
Using CPU frequency = 2.660000

ClockBench.java
                  Method       samples     min     max
avg  median   stdev
          System.nanoTime       255       54.00   60.00
55.31   57.00    1.55
```

```
           CLOCK_REALTIME       255       60.00   84.00
62.50   72.00    1.92
              cpuid+rdtsc       255      108.00  112.00
109.03  110.00    1.39
                    rdtsc       255       39.00   43.00
39.85   41.00    1.37
Using CPU frequency = 2.660000
```

A few things to note about these results:

- Both of the COARSE clocks show a latency of zero for getting the clock value. This tells us that the time it takes to get the clock value is less than the resolution of the clock. (Our previous test showed a resolution of 1ms for the COARSE clocks).

- For some reason, the CLOCK_MONOTONIC_RAW clock is very expensive to query. I can't explain this – you would think that its lack of adjustment would make it faster, not slower. This is unfortunate, as otherwise it would be an excellent choice for intra-machine timing.

- As you might expect, the combination of CPUID and RDTSC is slower than RDTSCP, which is slower than RDTSC alone. In general, this would suggest that RDTSCP should be preferred if available, with a fallback to CPUID+RDTSC if not. (While RDTSC alone is the fastest, the fact that it can be inaccurate as a result of out-of-order execution means it is only useful for timing relatively long operations where that inaccuracy is not significant – but those are precisely the scenarios where its speed is less important).

- Also as expected, the Java versions are slightly slower than the C++ versions, presumably due to the overhead of going through JNI.

Update

Well, I admit that Java is not my strong suit, but I nevertheless understand the implications of "warm-up" and JIT-ing when benchmarking Java code. My understanding (and the docs seem to agree) is that Java methods get JIT-ed after approx. 10,000 invocations. I also thought (and still do) that 100 iterations of 200 invocations would be more than 10,000. Whatever – I adjusted the number of iterations for the Java benchmark, and that made a big

difference – especially in the timings for System.nanotime, which now agree much more closely with other published benchmarks, specifically the results published in "Nanotrusting the Nanotime". Thanks, Aleksey!

Update #2

Slyain Archenault pointed out that the omission of unistd.h in SysTime.c caused it to (silently) fall back to using gettimeofday. I've updated the code to include the proper header, and also to issue a warning if _POSIX_TIMERS remains undefined. Thanks, Sylvain!

Conclusion

I thought this would be a very brief and somewhat trivial research project. In fact, it turned out to be far more complicated (and less well-documented) than I expected. I guess I should have known: everything related to time and computers turns out to be a major pain in the neck!

Anyway, I hope this proves helpful. (I know I would have been very happy to have had this when I started looking into clock sources).

As always, please feel free to contact me directly with comments, suggestions, corrections, etc.

Additional Resources

Following are the main anchor points that I kept coming back to you as I researched this article.

http://elinux.org/Kernel_Timer_Systems

http://elinux.org/High_Resolution_Timers

http://juliusdavies.ca/posix_clocks/clock_realtime_linux_faq.html

http://en.wikipedia.org/wiki/Time_Stamp_Counter

http://stackoverflow.com/questions/10921210/cpu-tsc-fetch-operation-especially-in-multicore-multi-processor-environment

http://www.citihub.com/requesting-timestamp-in-applications/

http://www.intel.com/content/www/us/en/intelligent-systems/embedded-systems-training/ia-32-ia-64-benchmark-code-execution-paper.html

1. The best case being hardware on each machine with a CSAC (chip-scale atomic clock) or OCXO (oven-controlled crystal oscillator). These can be a bit pricey, however.↩

2. The accuracy of a typical RTC in a PC-type computer is rated at +/- 20ppm, so it can gain or lose 20 us each second. This turns out to be approximately one minute per month, which may be OK for a cheap digital watch, but for a computer is not too good. For more information, see http://www.maximintegrated.com/app-notes/index.mvp/id/58.↩

3. Network Time Protocol, RFC 1305 (https://tools.ietf.org/html/rfc1305)↩

4. Precision Time Protocol, IEEE 1588 (http://www.nist.gov/el/isd/ieee/ieee1588.cfm)↩

5. From companies like Symmetricon, Corvil, TS Associates and others.↩

6. Note that the program must be compiled, as well as run, on the target system – it uses the presence or absence of pre-processor symbols to determine whether a particular clock source is available.↩

7. CentOS 6.5 running on a Dell 490 with dual Xeon 5150's at 2.6 GHz.↩

Posted by Bill Torpey • • linux

« You may ask yourself - "How did I get here?"

Using clang's Address Sanitizer (without clang) »

Comments