Measuring time for the piece of code you might be interested in tends to be quintessential but could become quite tricky. As if the multiple functions weren't sufficient already, we have to deal with time in virtual machines as well. So here are some facts/experiences from figuring out what functions lead to what kind of timing output collected together as quick reference (in Linux/Xen land).

FAQs from a blog entry:

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

> 1. Is clock_gettime(CLOCK_REALTIME) consistent across all processors/cores?
> (Does arch matter? e.g. ppc, arm, x86, amd64, sparc).

It *should* or it's considered buggy.

However, on x86/x86_64, it is possible to see unsynced or variable freq TSCs cause time inconsistencies. 2.4 kernels really had no protection against this, and early 2.6 kernels didn't do too well here either. As of 2.6.18 and up the logic for detecting this is better and we'll usually fall back to a safe clocksource.

ppc always has a synced timebase, so that shouldn't be an issue.

arm, i'm not so familiar with, but i assume they do the right thing (i've not seen many arm bugs on this issue).

> 2. Does clock_gettime(CLOCK_REALTIME) pick up the new time when a user
> (e.g. root) changes the current time on their computer?

Yes. CLOCK_REALTIME is affected by settime()/settimeofday() calls and can also be frequency corrected by NTP via adjtimex(). CLOCK_MONOTONIC is not affected by settime()/settimeofday(), but is frequency adjusted by NTP via adjtimex().

In the future (I'm still trying to get the patch in) there will be a CLOCK_MONOTONIC_RAW that will not be modified at all, and will have a linear correlation with the hardware counters.

> 3. Does clock_gettime(CLOCK_REALTIME) pick up NTP changes?
> 4. Does clock_gettime(CLOCK_REALTIME) pick up NTP clock slew?

With Linux, NTP normally uses settimeofday() for large corrections (over half a second). The adjtimex() inteface allows for small clock frequency changes (slewing). This can be done in a few different ways, see the man page for adjtimex.

With pre-2.6.18 kernels, the slewing was done by adding a little or taking some time away each tick. This could cause small time inconsistencies at the tick boundary. With post 2.6.18 kernels, the frequency adjustment is made directly against the hardware frequency.

> 5. gettimeofday() returns usec. Aside from that, what are the differences between
>   gettimeofday(), sys/time.h's nanotime(), and clock_gettime(CLOCK_REALTIME)?
>   Which would you recommend programmers use (for linux in particular, but
>   also when trying to write portable code)?

gettimeofday and clock_gettime(CLOCK_REALTIME,..) use the same back end code in the kernel.

In fact, prior to 2.6.18 on many architectures, clock_gettime(CLOCK_REALTIME,) just returned basically gettimeofday()'s value multiplied up to nanoseconds.

More recently its been turned around, and the kernel keeps finer precision, so gettimeofday() uses the clock_gettime(CLOCK_REALTIME) back end.

The only benefits to gettimeofday() is that on powerpc, ia64 and x86_64, it is implemented with a userspace-only vsyscall/vdso, which avoids the syscall overhead. However, recent x86_64 kernels have added support for vsyscall clock_gettime() as well.

>   Is the 2038/2106 problem a possibility for any of these?

The timeval/timespec time_t second value maps to a long, so for 32 bit architectures, yes, the 2038 issue is present. However on 64bit architectures it is not.

> 6. How about setting the current time: what are the differences between
>   settimeofday() and clock_settime(CLOCK_REALTIME)?

Nope, only that clock_settime allows for nanosecond resolution in setting the time. They are just different interfaces for the same back end code.

> 7. How does clock_gettime(CLOCK_REALTIME) work? Let's assume linux 2.6.26 on
>   the following systems:

This one will be a little long winded. The code is always the best documentation here (as its the most current). I'll try to give a brief rundown, but if folks are really interested, they should grab the kernel source and dig in.

>   Dual Xeon 3.06ghz (two single-core 32bit cpus).

Likely has a sync'ed TSC, so that would probably be the hardware clocksource used.

- App call's glibc's clock_gettime().
- glibc would make the syscall into the kernel
- The kernel would call sys_clock_gettime()
- sys_clock_gettime() calls getnstimeofday()
- getnstimeofday() would return the added combination of xtime and __get_nsec_offset()

- __get_nsec_offset() reads the clocksource hardware (in this case the TSC) and returns the time since xtime was last updated (its a little more complicated then this, but fundamentally that's what goes on).

> Dual Intel Xeon X5460 3.0ghz (two quad-core 64bit cpus).

Same as the above. The timekeeping core is generic between architectures.

The only possible difference is if the system was utilizing power saving and the TSC halted in the ACPI C3 state. Then the system would detect this and fall back to a slower but more reliable clocksource. Likely the HPET on a modern system like this.

However, if gettimeofday() was called, the same algorithm would be done completely in userspace using the vsyscall method (kernel maps a page of code and a page of data out as user-readable, and userland executes it directly, avoiding the syscall overhead).

> Single AMD Turion 1.9ghz (one dual-core 64bit cpu) on a laptop.

Since power saving is important here, its likely the TSC would either change frequency or halt in C3 idle, so it would not be usable. We'd detect this on boot up and switch to another available clocksource. On this system, I'm guessing the ACPI PM counter would be used. Then its exactly as in System A, but instead of reading the TSC, we read the ACPI PM.

Also, while the box is 64 bits, since the ACPI PM uses port-io to access it, it cannot be used in a vsyscall, so the vsyscall gettimeofday would be disabled.

> 8. Are there any interesting points / implementation notes worth knowing about
> clock_gettime(CLOCK_REALTIME)? Any notes regarding virtual machines?

Uh. There's potentially lots here. clock_gettime(CLOCK_REALTIME isn't super interesting on its own, but timekeeping in general is). Timekeeping with virtual machines is an interesting problem. Also the userland-only vdso/vsyscall methods for gettimeofday() are interesting.

> 9. Closing Comments.

Linux 2.6 is all I can really speak to off the top of my head. I'm not even really sure when clock_gettime() support was merged (I'm not really sure if it was around in early 2.4). Back in the 2.4 days, the hardware was simpler, so there was less to deal with and things basically just worked.

Summary of most common timing structures:

(http://www.delorie.com/gnu/docs/glibc/libc_428.html)

One way to represent an elapsed time is with a simple arithmetic data type, as with the following function to compute the elapsed time between two calendar times. This function is declared in `time.h'.

Function: double difftime (time_t time1, time_t time0)
The difftime function returns the number of seconds of elapsed time between calendar time time1 and calendar time time0, as a value of type double. The difference ignores leap seconds unless leap second support is enabled.

In the GNU system, you can simply subtract time_t values. But on other systems, the time_t data type might use some other encoding where subtraction doesn't work directly.

The GNU C library provides two data types specifically for representing an elapsed time. They are used by various GNU C library functions, and you can use them for your own purposes too. They're exactly the same except that one has a resolution in microseconds, and the other, newer one, is in nanoseconds.

Data Type: struct timeval
The struct timeval structure represents an elapsed time. It is declared in `sys/time.h' and has the following members:

long int tv_sec
This represents the number of whole seconds of elapsed time.

long int tv_usec
This is the rest of the elapsed time (a fraction of a second), represented as the number of microseconds. It is always less than one million.

Data Type: struct timespec
The struct timespec structure represents an elapsed time. It is declared in `time.h' and has the following members:

long int tv_sec
This represents the number of whole seconds of elapsed time.

long int tv_nsec
This is the rest of the elapsed time (a fraction of a second), represented as the number of nanoseconds. It is always less than one billion.

It is often necessary to subtract two values of type struct timeval or struct timespec. Here is the best way to do this. It works even on some peculiar operating systems where the tv_sec member has an unsigned type.

/* Subtract the `struct timeval' values X and Y,
storing the result in RESULT.
Return 1 if the difference is negative, otherwise 0. */

```
int
timeval_subtract (result, x, y)
struct timeval *result, *x, *y;
{
/* Perform the carry for the later subtraction by updating y. */
if (x->tv_usec < y->tv_usec) {
int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
y->tv_usec -= 1000000 * nsec;
y->tv_sec += nsec;
}
if (x->tv_usec - y->tv_usec > 1000000) {
int nsec = (x->tv_usec - y->tv_usec) / 1000000;
y->tv_usec += 1000000 * nsec;
y->tv_sec -= nsec;
}

/* Compute the time remaining to wait.
tv_usec is certainly positive. */
result->tv_sec = x->tv_sec - y->tv_sec;
result->tv_usec = x->tv_usec - y->tv_usec;

/* Return 1 if result is negative. */
return x->tv_sec < y->tv_sec;
}
```

Common functions that use struct timeval are gettimeofday and settimeofday.

There are no GNU C library functions specifically oriented toward dealing with elapsed times, but the calendar time, processor time, and alarm and sleeping functions have a lot to do with them.

A] Measuring user application time from command line:
/usr/bin/time –f "%e" <application>
gives total elapsed time for the application. It also has options to get user and system time. In my experience this command gives wall clock elapsed time even when run in a para-virtualized VM although I would appreciate a few more confirmations for this one. For virtual time spent by the application, the other options (for process time, system time) could possibly work.

From Mukil:
The GNU time command (the one with the formatting options) with the elapsed time option does in fact measure the wall clock time. The default bash time command also is measuring the wall clock time and so is uptime and most other forms of command line time measurement from domU.

It looks like the real trick is to measure the "virtual time" inside domU that more accurately tells you how long your application actually ran as opposed to wall clock time that also includes your domU's waiting time in the xen rq while it is de-scheduled.

I think this could be arranged by using the CLOCK_PROCESS* from clock_gettime() and the User and System times from the command line of time command although I have never tried it myself

B] Measuring user application time from within the application:
For measuring time within the user application, clock_gettime() is the glibc interface for using the nanosecond supporting getnstimeoday() function implemented in the Linux kernel (check the FAQs above). The C version of timer interface using clock_gettime() is posted here (http://www.cc.gatech.edu/~vishakha/useful/index.php). C++ version will be posted here (http://www.cc.gatech.edu/~romain/Computer/index.php). The only requirement for using this function is the presence of librt which has to be linked with your application. Hence it could pose portability issues although I have not faced a problem yet.

A lot of sample programs use gettimeofday() (http://www.opengroup.org/onlinepubs/000095399/functions/gettimeofday.html) which returns time with microseconds accuracy. This is a portable function and doesn't need any library support.

C] Measuring time in kernel:
Kernel implementations of timing calls are listed here (http://lxr.linux.no/#linux+v2.6.31/kernel/time.c#L318). The recommended call here is getnstimeofday() (http://www.gnugeneration.com/books/linux/2.6.20/kernel-api/re32.html) with <linux/time.h> included.

D] Getting down to Timestamp registers:
From Romain:

For greater accuracy and self-calculation, the last resort is reading the timebase registers supported by the corresponding instruction set. One has to be careful about measuring these timebase registers on different cores and comparing them because they may not be synchronized across cores. It is also important to put a check for overflow even though it may not occur during the course of your process.
On x86:
Using rdtsc (32/64 bit) or rdtsp (64bit)

On PowerPC:
http://www.ibm.com/developerworks/power/library/pa-timebase/index.html

E] Time measurement and signaling:
An extremely useful tutorial here –
http://www.helsinki.fi/atk/unix/dec_manuals/DOC_40D/APS33DTE/TOC.HTM
(Chapters 5 & 6 particularly) (Shared by Romain)