Clock sources, Clock events, sched_clock() and delay timers

This document tries to briefly explain some basic kernel timekeeping abstractions. It partly pertains to the drivers usually found in drivers/clocksource in the kernel tree, but the code may be spread out across the kernel.

If you grep through the kernel source you will find a number of architecture- specific implementations of clock sources, clockevents and several likewise architecture-specific overrides of the sched_clock() function and some delay timers.

To provide timekeeping for your platform, the clock source provides the basic timeline, whereas clock events shoot interrupts on certain points on this timeline, providing facilities such as high-resolution timers. sched_clock() is used for scheduling and timestamping, and delay timers provide an accurate delay source using hardware counters.

Clock sources

The purpose of the clock source is to provide a timeline for the system that tells you where you are in time. For example issuing the command 'date' on a Linux system will eventually read the clock source to determine exactly what time it is.

Typically the clock source is a monotonic, atomic counter which will provide n bits which count from 0 to (2^n)-1 and then wraps around to 0 and start over. It will ideally NEVER stop ticking as long as the system is running. It may stop during system suspend.

The clock source shall have as high resolution as possible, and the frequency shall be as stable and correct as possible as compared to a real-world wall clock. It should not move unpredictably back and forth in time or miss a few cycles here and there.

It must be immune to the kind of effects that occur in hardware where e.g. the counter register is read in two phases on the bus lowest 16 bits first and the higher 16 bits in a second bus cycle with the counter bits potentially being updated in between leading to the risk of very strange values from the counter.

When the wall-clock accuracy of the clock source isn't satisfactory, there are various quirks and layers in the timekeeping code for e.g. synchronizing the user-visible time to RTC clocks in the system or against networked time servers using NTP, but all they do basically is update an offset against the clock source, which provides the fundamental timeline for the system. These measures does not affect the clock source per se, they only adapt the system to the shortcomings of it.

The clock source struct shall provide means to translate the provided counter into a nanosecond value as an unsigned long long (unsigned 64 bit) number. Since this operation may be invoked very often, doing this in a strict mathematical sense is not desirable: instead the number is taken as close as possible to a nanosecond value using only the arithmetic operations multiply and shift, so in clocksource_cyc2ns() you find:

    ns ~= (clocksource * mult) >> shift

You will find a number of helper functions in the clock source code intended to aid in providing these mult and shift values, such as clocksource_khz2mult(), clocksource_hz2mult() that help determine the mult factor from a fixed shift, and clocksource_register_hz() and clocksource_register_khz() which will help out assigning both shift and mult factors using the frequency of the clock source as the only input.

For real simple clock sources accessed from a single I/O memory location there is nowadays even clocksource_mmio_init() which will take a memory location, bit width, a parameter telling whether the counter in the register counts up or down, and the timer clock rate, and then conjure all necessary parameters.

Since a 32-bit counter at say 100 MHz will wrap around to zero after some 43 seconds, the code handling the clock source will have to compensate for this. That is the reason why the clock source struct also contains a 'mask' member telling how many bits of the source are valid. This way the timekeeping code knows when the counter will wrap around and can insert the necessary compensation code on both sides of the wrap point so that the system timeline remains monotonic.

Clock events

Clock events are the conceptual reverse of clock sources: they take a desired time specification value and calculate the values to poke into hardware timer registers.

Clock events are orthogonal to clock sources. The same hardware and register range may be used for the clock event, but it is essentially a different thing. The hardware driving clock events has to be able to fire interrupts, so as to trigger events on the system timeline. On an SMP system, it is ideal (and customary) to have one such event driving timer per CPU core, so that each core can trigger events independently of any other core.

You will notice that the clock event device code is based on the same basic idea about translating counters to nanoseconds using mult and shift arithmetic, and you find the same family of helper functions again for assigning these values. The clock event driver does not need a 'mask' attribute however: the system will not try to plan events beyond the time horizon of the clock event.

sched_clock()

In addition to the clock sources and clock events there is a special weak function in the kernel called sched_clock(). This function shall return the number of nanoseconds since the system was started. An architecture may or may not provide an implementation of sched_clock() on its own. If a local implementation is not provided, the system jiffy counter will be used as sched_clock().

As the name suggests, sched_clock() is used for scheduling the system, determining the absolute timeslice for a certain process in the CFS scheduler for example. It is also used for printk timestamps when you have selected to include time information in printk for things like bootcharts.

Compared to clock sources, sched_clock() has to be very fast: it is called much more often, especially by the scheduler. If you have to do trade-offs between accuracy compared to the clock source, you may sacrifice accuracy for speed in sched_clock(). It however requires some of the same basic characteristics as the clock source, i.e. it should be monotonic.

The sched_clock() function may wrap only on unsigned long long boundaries, i.e. after 64 bits. Since this is a nanosecond value this will mean it wraps after circa 585 years. (For most practical systems this means "never".)

If an architecture does not provide its own implementation of this function, it will fall back to using jiffies, making its maximum resolution 1/HZ of the jiffy frequency for the architecture. This will affect scheduling accuracy and will likely show up in system benchmarks.

The clock driving sched_clock() may stop or reset to zero during system suspend/sleep. This does not matter to the function it serves of scheduling events on the system. However it may result in interesting timestamps in printk().

The sched_clock() function should be callable in any context, IRQ- and NMI-safe and return a sane value in any context.

Some architectures may have a limited set of time sources and lack a nice counter to derive a 64-bit nanosecond value, so for example on the ARM architecture, special helper functions have been created to provide a sched_clock() nanosecond base from a 16- or 32-bit counter. Sometimes the same counter that is also used as clock source is used for this purpose.

On SMP systems, it is crucial for performance that sched_clock() can be called independently on each CPU without any synchronization performance hits. Some hardware (such as the x86 TSC) will cause the sched_clock() function to drift between the CPUs on the system. The kernel can work around this by enabling the CONFIG_HAVE_UNSTABLE_SCHED_CLOCK option. This is another aspect that makes sched_clock() different from the ordinary clock source.

Delay timers (some architectures only)

On systems with variable CPU frequency, the various kernel delay() functions will sometimes behave strangely. Basically these delays usually use a hard loop to delay a certain number of jiffy fractions using a "lpj" (loops per jiffy) value, calibrated on boot.

Let's hope that your system is running on maximum frequency when this value is calibrated: as an effect when the frequency is geared down to half the full frequency, any delay() will be twice as long. Usually this does not hurt, as you're commonly requesting that amount of delay *or more*. But basically the semantics are quite unpredictable on such systems.

Enter timer-based delays. Using these, a timer read may be used instead of a hard-coded loop for providing the desired delay.

This is done by declaring a struct delay_timer and assigning the appropriate function pointers and rate settings for this delay timer.

This is available on some architectures like OpenRISC or ARM.