hrtimers - subsystem for high-resolution kernel timers

This patch introduces a new subsystem for high-resolution kernel timers.

One might ask the question: we already have a timer subsystem (kernel/timers.c), why do we need two timer subsystems? After a lot of back and forth trying to integrate high-resolution and high-precision features into the existing timer framework, and after testing various such high-resolution timer implementations in practice, we came to the conclusion that the timer wheel code is fundamentally not suitable for such an approach. We initially didn't believe this ('there must be a way to solve this'), and spent a considerable effort trying to integrate things into the timer wheel, but we failed. In hindsight, there are several reasons why such integration is hard/impossible:

- the forced handling of low-resolution and high-resolution timers in the same way leads to a lot of compromises, macro magic and #ifdef mess. The timers.c code is very "tightly coded" around jiffies and 32-bitness assumptions, and has been honed and micro-optimized for a relatively narrow use case (jiffies in a relatively narrow HZ range) for many years - and thus even small extensions to it easily break the wheel concept, leading to even worse compromises. The timer wheel code is very good and tight code, there's zero problems with it in its current usage - but it is simply not suitable to be extended for high-res timers.
- the unpredictable [O(N)] overhead of cascading leads to delays which necessitate a more complex handling of high resolution timers, which in turn decreases robustness. Such a design still leads to rather large timing inaccuracies. Cascading is a fundamental property of the timer wheel concept, it cannot be 'designed out' without inevitably degrading other portions of the timers.c code in an unacceptable way.
- the implementation of the current posix-timer subsystem on top of the timer wheel has already introduced a quite complex handling of the required readjusting of absolute CLOCK_REALTIME timers at settimeofday or NTP time - further underlying our experience by example: that the timer wheel data structure is too rigid for high-res timers.
- the timer wheel code is most optimal for use cases which can be identified as "timeouts". Such timeouts are usually set up to cover error conditions in various I/O paths, such as networking and block I/O. The vast majority of those timers never expire and are rarely recascaded because the expected correct event arrives in time so they can be removed from the timer wheel before any further processing of them becomes necessary. Thus the users of these timeouts can accept the granularity and precision tradeoffs of the timer wheel, and largely expect the timer subsystem to have near-zero overhead. Accurate timing for them is not a core purpose - in fact most of the timeout values used are ad-hoc. For them it is at most a necessary evil to guarantee the processing of actual timeout completions (because most of the timeouts are deleted before completion), which should thus be as cheap and unintrusive as possible.

The primary users of precision timers are user-space applications that utilize nanosleep, posix-timers and itimer interfaces. Also, in-kernel users like drivers and subsystems which require precise timed events (e.g. multimedia) can benefit from the availability of a separate high-resolution timer subsystem as well.

While this subsystem does not offer high-resolution clock sources just yet, the hrtimer subsystem can be easily extended with high-resolution clock capabilities, and patches for that exist and are maturing quickly. The increasing demand for realtime and multimedia applications along with other potential users for precise timers gives another reason to separate the "timeout" and "precise timer" subsystems.

Another potential benefit is that such a separation allows even more special-purpose optimization of the existing timer wheel for the low resolution and low precision use cases - once the precision-sensitive APIs are separated from the timer wheel and are migrated over to hrtimers. E.g. we could decrease the frequency of the timeout subsystem from 250 Hz to 100 HZ (or even smaller).

hrtimer subsystem implementation details

the basic design considerations were:

- simplicity
- data structure not bound to jiffies or any other granularity. All the kernel logic works at 64-bit nanoseconds resolution - no compromises.
- simplification of existing, timing related kernel code

another basic requirement was the immediate enqueueing and ordering of timers at activation time. After looking at several possible solutions such as radix trees and hashes, we chose the red black tree as the basic data structure. Rbtrees are available as a library in the kernel and are used in various performance-critical areas of e.g. memory management and file systems. The rbtree is solely used for time sorted ordering, while a separate list is used to give the expiry code fast access to the queued timers, without having to walk the rbtree.

(This separate list is also useful for later when we'll introduce high-resolution clocks, where we need separate pending and expired queues while keeping the time-order intact.)

Time-ordered enqueueing is not purely for the purposes of high-resolution clocks though, it also simplifies the handling of absolute timers based on a low-resolution CLOCK_REALTIME. The existing implementation needed to keep an extra list of all armed absolute CLOCK_REALTIME timers along with complex locking. In case of settimeofday and NTP, all the timers (!) had to be dequeued, the time-changing code had to fix them up one by one, and all of them had to be enqueued again. The time-ordered enqueueing and the storage of the expiry time in absolute time units removes all this complex and poorly scaling code from the posix-

timer implementation - the clock can simply be set without having to touch the rbtree. This also makes the handling of posix-timers simpler in general.

The locking and per-CPU behavior of hrtimers was mostly taken from the existing timer wheel code, as it is mature and well suited. Sharing code was not really a win, due to the different data structures. Also, the hrtimer functions now have clearer behavior and clearer names - such as hrtimer_try_to_cancel() and hrtimer_cancel() [which are roughly equivalent to del_timer() and del_timer_sync()] - so there's no direct 1:1 mapping between them on the algorithmic level, and thus no real potential for code sharing either.

Basic data types: every time value, absolute or relative, is in a special nanosecond-resolution type: ktime_t. The kernel-internal representation of ktime_t values and operations is implemented via macros and inline functions, and can be switched between a "hybrid union" type and a plain "scalar" 64bit nanoseconds representation (at compile time). The hybrid union type optimizes time conversions on 32bit CPUs. This build-time-selectable ktime_t storage format was implemented to avoid the performance impact of 64-bit multiplications and divisions on 32bit CPUs. Such operations are frequently necessary to convert between the storage formats provided by kernel and userspace interfaces and the internal time format. (See include/linux/ktime.h for further details.)

hrtimers - rounding of timer values

the hrtimer code will round timer events to lower-resolution clocks because it has to. Otherwise it will do no artificial rounding at all.

one question is, what resolution value should be returned to the user by the clock_getres() interface. This will return whatever real resolution a given clock has - be it low-res, high-res, or artificially-low-res.

hrtimers - testing and verification

We used the high-resolution clock subsystem ontop of hrtimers to verify the hrtimer implementation details in praxis, and we also ran the posix timer tests in order to ensure specification compliance. We also ran tests on low-resolution clocks.

The hrtimer patch converts the following kernel functionality to use hrtimers:

- nanosleep
- itimers
- posix-timers

The conversion of nanosleep and posix-timers enabled the unification of nanosleep and clock_nanosleep.

The code was successfully compiled for the following platforms:

i386, x86_64, ARM, PPC, PPC64, IA64

The code was run-tested on the following platforms:

i386(UP/SMP), x86_64(UP/SMP), ARM, PPC

hrtimers were also integrated into the -rt tree, along with a hrtimers-based high-resolution clock implementation, so the hrtimers code got a healthy amount of testing and use in practice.

Thomas Gleixner, Ingo Molnar