High resolution timers and dynamic ticks design notes

Further information can be found in the paper of the OLS 2006 talk "hrtimers and beyond". The paper is part of the OLS 2006 Proceedings Volume 1, which can be found on the OLS website: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf

The slides to this talk are available from: http://www.cs.columbia.edu/~nahum/w6998/papers/ols2006-hrtimers-slides.pdf

The slides contain five figures (pages 2, 15, 18, 20, 22), which illustrate the changes in the time(r) related Linux subsystems. Figure #1 (p. 2) shows the design of the Linux time(r) system before hrtimers and other building blocks got merged into mainline.

Note: the paper and the slides are talking about "clock event source", while we switched to the name "clock event devices" in meantime.

The design contains the following basic building blocks:

- hrtimer base infrastructure
- timeofday and clock source management
- clock event management
- high resolution timer functionality
- dynamic ticks

hrtimer base infrastructure

The hrtimer base infrastructure was merged into the 2.6.16 kernel. Details of the base implementation are covered in Documentation/timers/hrtimers.rst. See also figure #2 (OLS slides p. 15)

The main differences to the timer wheel, which holds the armed timer_list type timers are:

- time ordered enqueueing into a rb-tree
- independent of ticks (the processing is based on nanoseconds)

timeofday and clock source management

John Stultz's Generic Time Of Day (GTOD) framework moves a large portion of code out of the architecture-specific areas into a generic management framework, as illustrated in figure #3 (OLS slides p. 18). The architecture specific portion is reduced to the low level hardware details of the clock sources, which are registered in the framework and selected on a quality based decision. The low level code provides hardware setup and readout routines and initializes data structures, which are used by the generic time keeping code to convert the clock ticks to nanosecond based time values. All other time keeping related functionality is moved into the generic code. The GTOD base patch got merged into the 2.6.18 kernel.

Further information about the Generic Time Of Day framework is available in the OLS 2005 Proceedings Volume 1:

> http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf

The paper "We Are Not Getting Any Younger: A New Approach to Time and Timers" was written by J. Stultz, D.V. Hart, & N. Aravamudan.

Figure #3 (OLS slides p.18) illustrates the transformation.

clock event management

While clock sources provide read access to the monotonically increasing time value, clock event devices are used to schedule the next event interrupt(s). The next event is currently defined to be periodic, with its period defined at compile time. The setup and selection of the event device for various event driven functionalities is hardwired into the architecture dependent code. This results in duplicated code across all architectures and makes it extremely difficult to change the configuration of the system to use event interrupt devices other than those already built into the architecture. Another implication of the current design is that it is necessary to touch all the architecture-specific implementations in order to provide new functionality like high resolution timers or dynamic ticks.

The clock events subsystem tries to address this problem by providing a generic solution to manage clock event devices and their usage for the various clock event driven kernel functionalities. The goal of the clock event subsystem is to minimize the clock event related architecture dependent code to the pure hardware related handling and to allow easy addition and utilization of new clock event devices. It also minimizes the duplicated code across the architectures as it provides generic functionality down to the interrupt service handler, which is almost inherently hardware dependent.

Clock event devices are registered either by the architecture dependent boot code or at module insertion time. Each clock event device fills a data structure with clock-specific property parameters and callback functions. The clock event management decides, by using the specified property parameters, the set of system functions a clock event device will be used to support. This includes the distinction of per-CPU and per-system global event devices.

System-level global event devices are used for the Linux periodic tick. Per-CPU event devices are used to provide local CPU functionality such as process accounting, profiling, and high resolution timers.

The management layer assigns one or more of the following functions to a clock event device:

- system global periodic tick (jiffies update)
- cpu local update_process_times
- cpu local profiling
- cpu local next event interrupt (non periodic mode)

The clock event device delegates the selection of those timer interrupt related functions completely to the management layer. The clock management layer stores a function pointer in the device description structure, which has to be called from the hardware level handler. This removes a lot of duplicated code from the architecture specific timer interrupt handlers and hands the control over the clock event devices and the assignment of timer interrupt related functionality to the core code.

The clock event layer API is rather small. Aside from the clock event device registration interface it provides functions to schedule the next event interrupt, clock event device notification service and support for suspend and resume.

The framework adds about 700 lines of code which results in a 2KB increase of the kernel binary size. The conversion of i386 removes about 100 lines of code. The binary size decrease is in the range of 400 byte. We believe that the increase of flexibility and the avoidance of duplicated code across architectures justifies the slight increase of the binary size.

The conversion of an architecture has no functional impact, but allows to utilize the high resolution and dynamic tick functionalities without any change to the clock event device and timer interrupt code. After the conversion the enabling of high resolution timers and dynamic ticks is simply provided by adding the kernel/time/Kconfig file to the architecture specific Kconfig and adding the dynamic tick specific calls to the idle routine (a total of 3 lines added to the idle function and the Kconfig file)

Figure #4 (OLS slides p.20) illustrates the transformation.

high resolution timer functionality

During system boot it is not possible to use the high resolution timer functionality, while making it possible would be difficult and would serve no useful function. The initialization of the clock event device framework, the clock source framework (GTOD) and hrtimers itself has to be done and appropriate clock sources and clock event devices have to be registered before the high resolution functionality can work. Up to the point where hrtimers are initialized, the system works in the usual low resolution periodic mode. The clock source and the clock event device layers provide notification functions which inform hrtimers about availability of new hardware. hrtimers validates the usability of the registered clock sources and clock event devices before switching to high resolution mode. This ensures also that a kernel which is configured for high resolution timers can run on a system which lacks the necessary hardware support.

The high resolution timer code does not support SMP machines which have only global clock event devices. The support of such hardware would involve IPI calls when an interrupt happens. The overhead would be much larger than the benefit. This is the reason why we currently disable high resolution and dynamic ticks on i386 SMP systems which stop the local APIC in C3 power state. A workaround is available as an idea, but the problem has not been tackled yet.

The time ordered insertion of timers provides all the infrastructure to decide whether the event device has to be reprogrammed when a timer is added. The decision is made per timer base and synchronized across per-cpu timer bases in a support function. The design allows the system to utilize separate per-CPU clock event devices for the per-CPU timer bases, but currently only one reprogrammable clock event device per-CPU is utilized.

When the timer interrupt happens, the next event interrupt handler is called from the clock event distribution code and moves expired timers from the red-black tree to a separate double linked list and invokes the softirq handler. An additional mode field in the hrtimer structure allows the system to execute callback functions directly from the next event interrupt handler. This is restricted to code which can safely be executed in the hard interrupt context. This applies, for example, to the common case of a wakeup function as used by nanosleep. The advantage of executing the handler in the interrupt context is the avoidance of up to two context switches - from the interrupted context to the softirq and to the task which is woken up by the expired timer.

Once a system has switched to high resolution mode, the periodic tick is switched off. This disables the per system global periodic clock event device - e.g. the PIT on i386 SMP systems.

The periodic tick functionality is provided by an per-cpu hrtimer. The callback function is executed in the next event interrupt context and updates jiffies and calls update_process_times and profiling. The implementation of the hrtimer based periodic tick is designed to be extended with dynamic tick functionality. This allows to use a single clock event device to schedule high resolution timer and periodic events (jiffies tick, profiling, process accounting) on UP systems. This has been proved to work with the PIT on i386 and the Incrementer on PPC.

The softirq for running the hrtimer queues and executing the callbacks has been separated from the tick bound timer softirq to allow accurate delivery of high resolution timer signals which are used by itimer and POSIX interval timers. The execution of this softirq can still be delayed by other softirqs, but the overall latencies have been significantly improved by this separation.

Figure #5 (OLS slides p.22) illustrates the transformation.

dynamic ticks

Dynamic ticks are the logical consequence of the hrtimer based periodic tick replacement (sched_tick). The functionality of the sched_tick hrtimer is extended by three functions:

- hrtimer_stop_sched_tick
- hrtimer_restart_sched_tick
- hrtimer_update_jiffies

hrtimer_stop_sched_tick() is called when a CPU goes into idle state. The code evaluates the next scheduled timer event (from both hrtimers and the timer wheel) and in case that the next event is further away than the next tick it reprograms the sched_tick to this future event, to allow longer idle sleeps without worthless interruption by the periodic tick. The function is also called when an interrupt happens during the idle period, which does not cause a reschedule. The call is necessary as the interrupt handler might have armed a new timer whose expiry time is before the time which was identified as the nearest event in the previous call to hrtimer_stop_sched_tick.

hrtimer_restart_sched_tick() is called when the CPU leaves the idle state before it calls schedule(). hrtimer_restart_sched_tick() resumes the periodic tick, which is kept active until the next call to hrtimer_stop_sched_tick().

hrtimer_update_jiffies() is called from irq_enter() when an interrupt happens in the idle period to make sure that jiffies are up to date and the interrupt handler has not to deal with an eventually stale jiffy value.

The dynamic tick feature provides statistical values which are exported to userspace via /proc/stat and can be made available for enhanced power management control.

The implementation leaves room for further development like full tickless systems, where the time slice is controlled by the scheduler, variable frequency profiling, and a complete removal of jiffies in the future.

Aside the current initial submission of i386 support, the patchset has been extended to x86_64 and ARM already. Initial (work in progress) support is also available for MIPS and PowerPC.

Thomas, Ingo