Linux Timers Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

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

1.1 hrtimer base infrastructure

The hrtimer base infrastructure was merged into the 2.6.16 kernel. Details of the base implementation are covered in *hrtimers - subsystem for high-resolution kernel timers*. 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)

1.2 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.

1.3 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.

1.4 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.

1.5 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

CHAPTER

TWO

HIGH PRECISION EVENT TIMER DRIVER FOR LINUX

The High Precision Event Timer (HPET) hardware follows a specification by Intel and Microsoft, revision 1.

Each HPET has one fixed-rate counter (at 10+ MHz, hence “High Precision”) and up to 32 comparators. Normally three or more comparators are provided, each of which can generate oneshot interrupts and at least one of which has additional hardware to support periodic interrupts. The comparators are also called “timers”, which can be misleading since usually timers are independent of each other ... these share a counter, complicating resets.

HPET devices can support two interrupt routing modes. In one mode, the comparators are additional interrupt sources with no particular system role. Many x86 BIOS writers don’t route HPET interrupts at all, which prevents use of that mode. They support the other “legacy replacement” mode where the first two comparators block interrupts from 8254 timers and from the RTC.

The driver supports detection of HPET driver allocation and initialization of the HPET before the driver module_init routine is called. This enables platform code which uses timer 0 or 1 as the main timer to intercept HPET initialization. An example of this initialization can be found in arch/x86/kernel/hpet.c.

The driver provides a userspace API which resembles the API found in the RTC driver framework. An example user space program is provided in file:samples/timers/hpet_example.c

CHAPTER

THREE

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).

3.1 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 timer_delete() and timer_delete_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 64bit type: ktime_t. (Originally, the kernel-internal representation of ktime_t values and operations was implemented via macros and inline functions, and could be switched between a “hybrid union” type and a plain “scalar” 64bit nanoseconds representation (at compile time). This was abandoned in the context of the Y2038 work.)

3.2 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.

3.3 hrtimers - testing and verification

We used the high-resolution clock subsystem on top 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

CHAPTER

FOUR

NO_HZ: REDUCING SCHEDULING-CLOCK TICKS

This document describes Kconfig options and boot parameters that can reduce the number of scheduling-clock interrupts, thereby improving energy efficiency and reducing OS jitter. Reducing OS jitter is important for some types of computationally intensive high-performance computing (HPC) applications and for real-time applications.

There are three main ways of managing scheduling-clock interrupts (also known as "scheduling-clock ticks" or simply "ticks"):

1. Never omit scheduling-clock ticks (CONFIG_HZ_PERIODIC=y or CONFIG_NO_HZ=n for older kernels). You normally will -not- want to choose this option.
2. Omit scheduling-clock ticks on idle CPUs (CONFIG_NO_HZ_IDLE=y or CONFIG_NO_HZ=y for older kernels). This is the most common approach, and should be the default.
3. Omit scheduling-clock ticks on CPUs that are either idle or that have only one runnable task (CONFIG_NO_HZ_FULL=y). Unless you are running realtime applications or certain types of HPC workloads, you will normally -not- want this option.

These three cases are described in the following three sections, followed by a third section on RCU-specific considerations, a fourth section discussing testing, and a fifth and final section listing known issues.

4.1 Never Omit Scheduling-Clock Ticks

Very old versions of Linux from the 1990s and the very early 2000s are incapable of omitting scheduling-clock ticks. It turns out that there are some situations where this old-school approach is still the right approach, for example, in heavy workloads with lots of tasks that use short bursts of CPU, where there are very frequent idle periods, but where these idle periods are also quite short (tens or hundreds of microseconds). For these types of workloads, scheduling clock interrupts will normally be delivered any way because there will frequently be multiple runnable tasks per CPU. In these cases, attempting to turn off the scheduling clock interrupt will have no effect other than increasing the overhead of switching to and from idle and transitioning between user and kernel execution.

This mode of operation can be selected using CONFIG_HZ_PERIODIC=y (or CONFIG_NO_HZ=n for older kernels).

However, if you are instead running a light workload with long idle periods, failing to omit scheduling-clock interrupts will result in excessive power consumption. This is especially bad

on battery-powered devices, where it results in extremely short battery lifetimes. If you are running light workloads, you should therefore read the following section.

In addition, if you are running either a real-time workload or an HPC workload with short iterations, the scheduling-clock interrupts can degrade your applications performance. If this describes your workload, you should read the following two sections.

4.2 Omit Scheduling-Clock Ticks For Idle CPUs

If a CPU is idle, there is little point in sending it a scheduling-clock interrupt. After all, the primary purpose of a scheduling-clock interrupt is to force a busy CPU to shift its attention among multiple duties, and an idle CPU has no duties to shift its attention among.

An idle CPU that is not receiving scheduling-clock interrupts is said to be "dyntick-idle", "in dyntick-idle mode", "in nohz mode", or "running tickless". The remainder of this document will use "dyntick-idle mode".

The CONFIG_NO_HZ_IDLE=y Kconfig option causes the kernel to avoid sending scheduling-clock interrupts to idle CPUs, which is critically important both to battery-powered devices and to highly virtualized mainframes. A battery-powered device running a CONFIG_HZ_PERIODIC=y kernel would drain its battery very quickly, easily 2-3 times as fast as would the same device running a CONFIG_NO_HZ_IDLE=y kernel. A mainframe running 1,500 OS instances might find that half of its CPU time was consumed by unnecessary scheduling-clock interrupts. In these situations, there is strong motivation to avoid sending scheduling-clock interrupts to idle CPUs. That said, dyntick-idle mode is not free:

1. It increases the number of instructions executed on the path to and from the idle loop.
2. On many architectures, dyntick-idle mode also increases the number of expensive clock-reprogramming operations.

Therefore, systems with aggressive real-time response constraints often run CONFIG_HZ_PERIODIC=y kernels (or CONFIG_NO_HZ=n for older kernels) in order to avoid degrading from-idle transition latencies.

There is also a boot parameter "nohz=" that can be used to disable dyntick-idle mode in CONFIG_NO_HZ_IDLE=y kernels by specifying "nohz=off". By default, CONFIG_NO_HZ_IDLE=y kernels boot with "nohz=on", enabling dyntick-idle mode.

4.3 Omit Scheduling-Clock Ticks For CPUs With Only One Runnable Task

If a CPU has only one runnable task, there is little point in sending it a scheduling-clock interrupt because there is no other task to switch to. Note that omitting scheduling-clock ticks for CPUs with only one runnable task implies also omitting them for idle CPUs.

The CONFIG_NO_HZ_FULL=y Kconfig option causes the kernel to avoid sending scheduling-clock interrupts to CPUs with a single runnable task, and such CPUs are said to be "adaptive-ticks CPUs". This is important for applications with aggressive real-time response constraints because it allows them to improve their worst-case response times by the maximum duration of a scheduling-clock interrupt. It is also important for computationally intensive short-iteration workloads: If any CPU is delayed during a given iteration, all the other CPUs will be forced

to wait idle while the delayed CPU finishes. Thus, the delay is multiplied by one less than the number of CPUs. In these situations, there is again strong motivation to avoid sending scheduling-clock interrupts.

By default, no CPU will be an adaptive-ticks CPU. The “nohz_full=” boot parameter specifies the adaptive-ticks CPUs. For example, “nohz_full=1,6-8” says that CPUs 1, 6, 7, and 8 are to be adaptive-ticks CPUs. Note that you are prohibited from marking all of the CPUs as adaptive-tick CPUs: At least one non-adaptive-tick CPU must remain online to handle timekeeping tasks in order to ensure that system calls like gettimeofday() returns accurate values on adaptive-tick CPUs. (This is not an issue for CONFIG_NO_HZ_IDLE=y because there are no running user processes to observe slight drifts in clock rate.) Therefore, the boot CPU is prohibited from entering adaptive-ticks mode. Specifying a “nohz_full=” mask that includes the boot CPU will result in a boot-time error message, and the boot CPU will be removed from the mask. Note that this means that your system must have at least two CPUs in order for CONFIG_NO_HZ_FULL=y to do anything for you.

Finally, adaptive-ticks CPUs must have their RCU callbacks offloaded. This is covered in the “RCU IMPLICATIONS” section below.

Normally, a CPU remains in adaptive-ticks mode as long as possible. In particular, transitioning to kernel mode does not automatically change the mode. Instead, the CPU will exit adaptive-ticks mode only if needed, for example, if that CPU enqueues an RCU callback.

Just as with dyntick-idle mode, the benefits of adaptive-tick mode do not come for free:

1. CONFIG_NO_HZ_FULL selects CONFIG_NO_HZ_COMMON, so you cannot run adaptive ticks without also running dyntick idle. This dependency extends down into the implementation, so that all of the costs of CONFIG_NO_HZ_IDLE are also incurred by CONFIG_NO_HZ_FULL.

2. The user/kernel transitions are slightly more expensive due to the need to inform kernel subsystems (such as RCU) about the change in mode.

3. POSIX CPU timers prevent CPUs from entering adaptive-tick mode. Real-time applications needing to take actions based on CPU time consumption need to use other means of doing so.

4. If there are more perf events pending than the hardware can accommodate, they are normally round-robined so as to collect all of them over time. Adaptive-tick mode may prevent this round-robining from happening. This will likely be fixed by preventing CPUs with large numbers of perf events pending from entering adaptive-tick mode.

5. Scheduler statistics for adaptive-tick CPUs may be computed slightly differently than those for non-adaptive-tick CPUs. This might in turn perturb load-balancing of real-time tasks.

Although improvements are expected over time, adaptive ticks is quite useful for many types of real-time and compute-intensive applications. However, the drawbacks listed above mean that adaptive ticks should not (yet) be enabled by default.

4.4 RCU Implications

There are situations in which idle CPUs cannot be permitted to enter either dyntick-idle mode or adaptive-tick mode, the most common being when that CPU has RCU callbacks pending.

Avoid this by offloading RCU callback processing to "rcuo" kthreads using the CONFIG_RCU_NOCB_CPU=y Kconfig option. The specific CPUs to offload may be selected using The "rcu_nocbs=" kernel boot parameter, which takes a comma-separated list of CPUs and CPU ranges, for example, "1,3-5" selects CPUs 1, 3, 4, and 5. Note that CPUs specified by the "nohz_full" kernel boot parameter are also offloaded.

The offloaded CPUs will never queue RCU callbacks, and therefore RCU never prevents offloaded CPUs from entering either dyntick-idle mode or adaptive-tick mode. That said, note that it is up to userspace to pin the "rcuo" kthreads to specific CPUs if desired. Otherwise, the scheduler will decide where to run them, which might or might not be where you want them to run.

4.5 Testing

So you enable all the OS-jitter features described in this document, but do not see any change in your workload's behavior. Is this because your workload isn't affected that much by OS jitter, or is it because something else is in the way? This section helps answer this question by providing a simple OS-jitter test suite, which is available on branch master of the following git archive:

git://git.kernel.org/pub/scm/linux/kernel/git/frederic/dynticks-testing.git

Clone this archive and follow the instructions in the README file. This test procedure will produce a trace that will allow you to evaluate whether or not you have succeeded in removing OS jitter from your system. If this trace shows that you have removed OS jitter as much as is possible, then you can conclude that your workload is not all that sensitive to OS jitter.

Note: this test requires that your system have at least two CPUs. We do not currently have a good way to remove OS jitter from single-CPU systems.

4.6 Known Issues

- Dyntick-idle slows transitions to and from idle slightly. In practice, this has not been a problem except for the most aggressive real-time workloads, which have the option of disabling dyntick-idle mode, an option that most of them take. However, some workloads will no doubt want to use adaptive ticks to eliminate scheduling-clock interrupt latencies. Here are some options for these workloads:

  a. Use PMQOS from userspace to inform the kernel of your latency requirements (preferred).

  b. On x86 systems, use the "idle=mwait" boot parameter.

  c. On x86 systems, use the "intel_idle.max_cstate=" to limit ` the maximum C-state depth.

  d. On x86 systems, use the "idle=poll" boot parameter. However, please note that use of this parameter can cause your CPU to overheat, which may cause thermal throttling to degrade your latencies -- and that this degradation can be even worse than that

of dyntick-idle. Furthermore, this parameter effectively disables Turbo Mode on Intel CPUs, which can significantly reduce maximum performance.

- Adaptive-ticks slows user/kernel transitions slightly. This is not expected to be a problem for computationally intensive workloads, which have few such transitions. Careful benchmarking will be required to determine whether or not other workloads are significantly affected by this effect.
- Adaptive-ticks does not do anything unless there is only one runnable task for a given CPU, even though there are a number of other situations where the scheduling-clock tick is not needed. To give but one example, consider a CPU that has one runnable high-priority SCHED_FIFO task and an arbitrary number of low-priority SCHED_OTHER tasks. In this case, the CPU is required to run the SCHED_FIFO task until it either blocks or some other higher-priority task awakens on (or is assigned to) this CPU, so there is no point in sending a scheduling-clock interrupt to this CPU. However, the current implementation nevertheless sends scheduling-clock interrupts to CPUs having a single runnable SCHED_FIFO task and multiple runnable SCHED_OTHER tasks, even though these interrupts are unnecessary.

  And even when there are multiple runnable tasks on a given CPU, there is little point in interrupting that CPU until the current running task's timeslice expires, which is almost always way longer than the time of the next scheduling-clock interrupt.

  Better handling of these sorts of situations is future work.
- A reboot is required to reconfigure both adaptive idle and RCU callback offloading. Runtime reconfiguration could be provided if needed, however, due to the complexity of reconfiguring RCU at runtime, there would need to be an earthshakingly good reason. Especially given that you have the straightforward option of simply offloading RCU callbacks from all CPUs and pinning them where you want them whenever you want them pinned.
- Additional configuration is required to deal with other sources of OS jitter, including interrupts and system-utility tasks and processes. This configuration normally involves binding interrupts and tasks to particular CPUs.
- Some sources of OS jitter can currently be eliminated only by constraining the workload. For example, the only way to eliminate OS jitter due to global TLB shootdowns is to avoid the unmapping operations (such as kernel module unload operations) that result in these shootdowns. For another example, page faults and TLB misses can be reduced (and in some cases eliminated) by using huge pages and by constraining the amount of memory used by the application. Pre-faulting the working set can also be helpful, especially when combined with the mlock() and mlockall() system calls.
- Unless all CPUs are idle, at least one CPU must keep the scheduling-clock interrupt going in order to support accurate timekeeping.
- If there might potentially be some adaptive-ticks CPUs, there will be at least one CPU keeping the scheduling-clock interrupt going, even if all CPUs are otherwise idle.

  Better handling of this situation is ongoing work.
- Some process-handling operations still require the occasional scheduling-clock tick. These operations include calculating CPU load, maintaining sched average, computing CFS entity vruntime, computing avenrun, and carrying out load balancing. They are currently accommodated by scheduling-clock tick every second or so. On-going work will eliminate the need even for these infrequent scheduling-clock ticks.

CHAPTER

FIVE

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.

5.1 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.

5.2 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.

5.3 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.

5.4 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.

CHAPTER

SIX

DELAYS - INFORMATION ON THE VARIOUS KERNEL DELAY / SLEEP MECHANISMS

This document seeks to answer the common question: “What is the RightWay (TM) to insert a delay?”

This question is most often faced by driver writers who have to deal with hardware delays and who may not be the most intimately familiar with the inner workings of the Linux Kernel.

6.1 Inserting Delays

The first, and most important, question you need to ask is “Is my code in an atomic context?” This should be followed closely by “Does it really need to delay in atomic context?” If so...

ATOMIC CONTEXT:

You must use the **delay* family of functions. These functions use the jiffie estimation of clock speed and will busy wait for enough loop cycles to achieve the desired delay:

ndelay(unsigned long nsecs) udelay(unsigned long usecs) mdelay(unsigned long msecs)

udelay is the generally preferred API; ndelay-level precision may not actually exist on many non-PC devices.

mdelay is macro wrapper around udelay, to account for possible overflow when passing large arguments to udelay. In general, use of mdelay is discouraged and code should be refactored to allow for the use of msleep.

NON-ATOMIC CONTEXT:

You should use the **sleep[_range]* family of functions. There are a few more options here, while any of them may work correctly, using the “right” sleep function will help the scheduler, power management, and just make your driver better :)

-- Backed by busy-wait loop:

udelay(unsigned long usecs)

-- Backed by hrtimers:

usleep_range(unsigned long min, unsigned long max)

-- Backed by jiffies / legacy_timers

msleep(unsigned long msecs) msleep_interruptible(unsigned long msecs)

Unlike the **delay* family, the underlying mechanism driving each of these calls varies, thus there are quirks you should be aware of.

SLEEPING FOR "A FEW" USECS (< ~10us?):

- Use udelay
- **Why not usleep?**
  On slower systems, (embedded, OR perhaps a speed- stepped PC!) the overhead of setting up the hrtimers for usleep *may* not be worth it. Such an evaluation will obviously depend on your specific situation, but it is something to be aware of.

SLEEPING FOR ~USECS OR SMALL MSECS (10us - 20ms):

- Use usleep_range
- **Why not msleep for (1ms - 20ms)?**

  Explained originally here:
  https://lore.kernel.org/r/15327.1186166232@lwn.net

  msleep(1~20) may not do what the caller intends, and will often sleep longer (~20 ms actual sleep for any value given in the 1~20ms range). In many cases this is not the desired behavior.
- **Why is there no "usleep" / What is a good range?**
  Since usleep_range is built on top of hrtimers, the wakeup will be very precise (ish), thus a simple usleep function would likely introduce a large number of undesired interrupts.

  With the introduction of a range, the scheduler is free to coalesce your wakeup with any other wakeup that may have happened for other reasons, or at the worst case, fire an interrupt for your upper bound.

  The larger a range you supply, the greater a chance that you will not trigger an interrupt; this should be balanced with what is an acceptable upper bound on delay / performance for your specific code path. Exact tolerances here are very situation specific, thus it is left to the caller to determine a reasonable range.

SLEEPING FOR LARGER MSECS (10ms+)

- Use msleep or possibly msleep_interruptible
- **What's the difference?**
  msleep sets the current task to TASK_UNINTERRUPTIBLE whereas msleep_interruptible sets the current task to TASK_INTERRUPTIBLE before scheduling the sleep. In short, the difference is whether the sleep can be ended early by a signal. In general, just use msleep unless you know you have a need for the interruptible variant.

FLEXIBLE SLEEPING (any delay, uninterruptible)

- Use fsleep