We Are Not Getting Any Younger: A New Approach to Time and Timers

John Stultz, Nishanth Aravamudan, Darren Hart
IBM Linux Technology Center
{johnstul, dvhltc, nacc}@us.ibm.com

Abstract

The Linux® time subsystem, which once provided only tick granularity via a simple periodic addition to xtime, now must provide nanosecond resolution. As more and more unique timekeeping hardware becomes available, and as virtualization and low-latency demands grow, the complexity of maintenance and bug resolution increases.

We are proposing a significant re-work of the time keeping subsystem of the kernel. Additionally, we demonstrate some possible enhancements this re-work facilitates: a more flexible soft-timer subsystem and dynamic interrupt source management.

1 The New `timeofday` Subsystem

The functionality required of the `timeofday` subsystem is deceptively simple. We must provide a monotonically increasing system clock, a fast and accurate method for generating the time of day, and a method for making small adjustments to compensate for clock drift. In the existing code, these are provided by the `wall_to_monotonic` offset to `do_gettimeofday()`, the `do_gettimeofday()` function and the Network Time Protocal(NTP) `adjtimex()` interface. This basic functionality, however, is required to meet increasingly stringent demands. Performance must improve and time resolution must increase, while keeping correctness. Meeting these demands with the current code has become difficult, thus new methods for time keeping must be considered.

1.1 Terminology

Before we get into the paper, let us just cover some quick terminology used, so there is no confusion.

System Time A monotonically increasing value that represents the amount of time the system has been running.

Wall Time (Time of Day) A value representing the the human time of day, as seen on a wrist-watch.

Timesource A representation of a free running counter running at a known frequency, usually in hardware.

Hardware-Timer (Interrupt Source) A bit of hardware that can be programmed to generate an interrupt at a specific point in time. Frequently the hardware-timer can be used as a timesource as well.

Soft-Timer A software kernel construct that runs a callback at a specified time.

Tick A periodic interrupt generated by a hardware-timer, typically with a fixed interval defined by `HZ`. Normally used for timekeeping and soft-timer management.

1.2 Reasons for Change

1.2.1 Correctness

So, why are we proposing these changes? There are many reasons, but the most important one is correctness. It is critical that time flow smoothly, accurately and does not go backwards at any point. The existing interpolation-based timekeeping used by most arches[1] is error prone.

Quickly, let us review how the existing timekeeping code works. The code is tick-based, so the `timeofday` is incremented a constant amount (plus a minor adjustment for NTP) every tick. A simplified pseudo-code example of this tick-based scheme would look like Figure 1.

Since, in this code, `abs(ntp_adjustment)` is always smaller then `NSECS_PER_TICK`, time cannot go backwards. Thus, the only issue with this example is that the resolution of time is not very granular, e.g. only 1 millisecond when $\texttt{HZ} = 1000$. To solve this, the existing `timeofday` code uses a high-resolution timesource to interpolate with the tick-based time keeping. Again, using a simplified pseudo-code example we get something like Figure 2.

The idea is that the interpolation function (`cycles2ns(now - hi_res_base)`) will smoothly give the inter-tick position in time. Now, a sharp eye will notice that this has some potential problems. If the interpolation function does not cover the entire tick interval (`NSECS_PER_TICK + ntp_adjustment`), time will jump forward. More worrisome though, if at any time the value of the interpolation function grows to be larger than the tick interval, there is the potential for time to go backwards. These two conditions can be caused by a number of factors: calibration error, changes to `ntp_adjustment`'s value, interrupt handler delay, timesource frequency changes, and lost ticks.

The first three cases typically lead to a small single-digit microsecond error, and thus, a small window for inconsistency. As processor speeds increase, `gettimeofday` takes less time and small interpolation errors become more apparent.

Timesource frequency changes are more difficult to deal with. While they are less common, some laptops do not notify the `cpufreq` subsystem when they change processor frequency. Should the user boot off of battery power, and the system calibrate the timesource at the slow speed then later plug into the wall, the TSC frequency can triple causing much more obvious (two tick length) errors.

Lost timer interrupts also cause large and easily visible time inconsistencies. If a bad driver blocks interrupts for too long, or a BIOS SMI interrupt blocks the OS from running, it is possible for the value of the interpolation function to grow to multiple tick intervals in length. When a timer interrupt is finally taken though, only one interval is accumulated. Since changing `HZ` to 1000 on most systems, missed timer interrupts have become more common, causing large time inconsistencies and clock drift.

Attempts have been made (in many cases by one of the authors of this paper) to reduce the

[1] Throughout this paper, we will refer to the software architecture(s), i.e. those found in `linux-source/arch`, as arch(es) and to hardware architecture(s) as architecture(s).

```
timer_interrupt():
    xtime += NSECS_PER_TICK + ntp_adjustment

gettimeofday():
    return xtime
```

Figure 1: A tick-based `timeofday` implementation

```
timer_interrupt():
    hi_res_base = read_timesource()
    xtime += NSECS_PER_TICK + ntp_adjustment

gettimeofday():
    now = read_timesource()
    return xtime + cycles2ns(now - hi_res_base)
```

Figure 2: An interpolated `gettimeofday()`

impact of lost ticks by trying to detect and compensate for them. At interrupt time, the interpolation function is used to see how much time has passed, and any lost ticks are then emulated. One problem with this method of detecting lost ticks is that it results in false positives when timesource frequency changes occur. Thus, instead of time inconsistencies, time races three times faster on those systems, necessitating additional heuristics.

In summary, the kernel uses a buggy method for calculating time using both ticks and timesources, neither of which can be trusted in all situations. This is not good.

1.2.2 A Much Needed Cleanup

Another problem with the current time keeping code is how fractured the code base has become. Every arch calculates time in basically the same manner, however, each one has its own implementation with minor differences. In many cases bugs have been fixed in one or two arches but not in the rest. The arch independent code is fragmented over a number of files (time.c, timer.c, time.h, timer.h, times.h, timex.h), where the divisions have lost any meaning or reason. Without clear and explicit purpose to these files, new code has been added to these files haphazardly, making it even more difficult to make sense of what is changing.

The lack of opacity in the current timekeeping code is an issue as well. Since almost all of the timekeeping variables are global and have no clear interface, they are accessed in a number of different ways in a number of different places in the code. This has caused much confusion in the way kernel code accesses time. For example, consider the many possible ways to calculate uptime shown in Figure 3.

Clearly some of these examples are more correct than others, but there is not one clear way of doing it. Since different ways to calculate time are used in the kernel, bugs caused by mixing methods are common.

1.2.3 Needed Flexibility

The current system also lacks flexibility. Current workarounds for things like lost ticks cause

```
uptime = jiffies * HZ
uptime = (jiffies - INITIAL_JIFFIES) * HZ
uptime = ((jiffies - INITIAL_JIFFIES) * ACTHZ) >> 8
uptime = xtime.tv_sec + wall_to_monotonic.tv_sec
uptime = monotonic_clock() / NSEC_PER_SEC;
```

Figure 3: Possible ways to calculate uptime

bugs elsewhere. One such place is virtualization, where the guest OS may be halted for many ticks while another OS runs. This requires the hypervisor to emulate a number of successive ticks when the guest OS runs. The lost tick compensation code notes the hang and tries to compensate on the first tick, but then the next ticks arrive causing time to run too fast. Needless to say, this dependency on ticks increases the difficulty of correctly implementing deeper kernel changes like those proposed in other patches such as: Dynamic Ticks, `NO_IDLE_HZ`, Variable System Tick, and High Res Timers.

1.3 Related Work

1.3.1 `timer_opts`

Now that we are done complaining about all the problems of the current timekeeping subsystem, one of the authors should stand up and claim his portion of the responsibility. John Stultz has been working in this area off and on for the last three years. His largest set of changes was the i386-specific `timer_opts` reorganization. The main purpose of that code was to modularize the high-resolution interpolator to allow for easy addition of alternative timesources. At that time the i386 arch supported the PIT and the TSC, and it was necessary to add support for a third timesource, the Cyclone counter. Thanks, in part, to to the `timer_opts` changes the kernel now supports the ACPI PM and HPET timesources, so in a sense the code did what was needed, but it is not without its problems.

The first and most irritating issue is the name. While it is called `timer_opts`, nothing in the structure actually uses any of the underlying hardware as a hardware-timer. Part of this confusion comes from the fact that hardware-timers can frequently be used as counters or timesources, but not the other way around, as timesources do not necessarilly generate interrupts. The lack of precision in naming and speaking about the code has caused continuing difficulty in discussing the issues surrounding it.

The other problem with the `timer_opts` structure is that its interface is too flexible and openly defined. It is unclear for those writing new `timer_opt` modules, how to use the interface as intended. Additionally, fixing a bug or adding a feature requires implementing the same code across each `timer_opt` module, leading to excessive code duplication. That along with additional interfaces being added (each needing their own implementation) has caused the `timer_opts` modules to become ugly and confusing. A cleanup is in order.

Finally, while the `timer_opts` structure is somewhat modular, the list of available timesources becomes fixed at build-time. Having the “clock=” boot-time parameter is useful, allowing users to override the default timesource; however, more flexibility in providing new timesources at run-time would be helpful.

1.3.2 `time_interpolator`

Right about the time the `timer_opts` structure got into the kernel, a similar bit of code called "time interpolation hooks" showed up implementing a somewhat similar interface. An additional benefit of this code was that it was arch independent, promising the ability to share interpolator "drivers" between different arches that had the same hardware. John followed a bit of its discussion and intended to move the i386 code over to it, but was distracted by other work requirements. That and the never-ending 2.6 freeze kept him from actually attempting the change.

John finally got a chance to really look at the code when he implemented the Cyclone interpolator driver. The code was nice and more modular than the `timer_opts` interface, but still had some of the same faults: it left too much up to the driver to implement and the `getoffset()`, `update()`, and `reset()` interfaces were not intuitive. Impressively, much of the time-interpolator code has been recently re-written, resolving many of issues and influencing this proposal. However, the time-interpolator design is still less than ideal. NTP adjustments are done by intentionally undershooting in converting from cycles to nanoseconds, causing time to run just a touch slow, and thus, forcing NTP to only make adjustments forward in time. This trick avoids time inconsistencies from NTP adjustments, but causes time drift on systems that do not run NTP. Additionally, while interpolation errors are no longer an issue, the code is still tick based, which makes it more difficult to understand and extend.

1.4 Our Proposal

Before we get into the implementation details of our proposal, let us review the goals:

1. Clean up and simplify time related code.
2. Provide clean and clear `timeofday` interfaces.
3. Use nanoseconds as the fundamental time unit.
4. Stop using tick-based time and avoid interpolation.
5. Make much of the implementation arch independent.
6. Use a modular design, allowing timesource drivers to be installed at runtime.

The core implementation has three main components: the `timeofday` code, timesource management, and the NTP state machine.

1.4.1 `timeofday` Core

The core of the timekeeping code provides methods for getting a monotonically increasing system time and the wall time. To avoid unnecessary complication, we layer these two values in a simple way. The monotonically increasing system time, accessed via `monotonic_clock()` is the base layer. On top of that we add a constant offset `wall_time_offset` to calculate the wall time value returned by `do_gettimeofday()`. The code looks like Figure 4.

The first thing to note in Figure 4, is that the `timeofday` code is not using interpolation. The amount of time accumulated in the `periodic_hook()` function is the exact same as would be calculated in `monotonic_clock()`. This means the `timeofday` code is no longer dependent on timer interrupts being received at a regular interval. If a tick arrives late, that is okay, we will just accumulate the actual amount of time that has past and reset the `offset_base`. In fact, `periodic_hook()` does not need to be called

```
nsec_t system_time
nsec_t wall_time_offset
cycle_t offset_base
int ntp_adj
struct timesource_t ts

monotonic_clock():
    now = read_timesource(ts)
    return system_time + cycles2ns(ts, now - offset_base, ntp_adj)

gettimeofday():
    return monotonic_clock() + wall_time_offset

periodic_hook():
    now = read_timesource(ts)
    interval = cycles2ns(ts, now - offset_base, ntp_adj)
    system_time += interval
    offset_base = now
    ntp_adj = ntp_advance(interval)
```

Figure 4: The new timekeeping psuedo-code

from `timer_interrupt()`, instead it can be called from a soft-timer scheduled to run every number of ticks. Additionally, notice that NTP adjustments are done smoothly and consistenetly throughout the time interval between `periodic_hook()` calls. This avoids the interpolation error that occurs with the current code when the NTP adjustment is only applied at tick time. Another benefit is that the core algorithm is shared between all arches. This consolidates a large amount of redundant arch specific code, which simplifies maintenance and reduces the number of arch specific time bugs.

1.4.2 Timesource Management

The timesource management code defines a timesource, and provides accessor functions for reading and converting timesource cycle values to nanoseconds. Additionally, it provides the interface for timesources to be registered, and then selected by the kernel. The timesoure structure is defined in Figure 5.

In this structure, the `priority` field allows for the best available timesource to be chosen. The `type` defines if the timesource can be directly accessed from the on-CPU cycle counter, via MMIO, or via a function call, which are defined by the `read_fnct` and `mmio_ptr` pointers respectively. The `mask` value ensures that subtraction between counter values from counters that are less then 64 bits do not need special overflow logic. The `mult` and `shift` approximate the frequency value of cycles over nanoseconds, where $\texttt{frequency} \approx \frac{\texttt{mult}}{2^{\texttt{shift}}}$ Finally, the `update_callback()` is used as a notifier for a safe point where the timesource can change its `mult` or `shift` values if needed, e.g. in the case of `cpufreq` scaling.

A simple example `timesource` structure can be seen in Figure 6. This small HPET driver can even be shared between i386, x86-64 and ia64 arches (or any arch that supports HPET). All that is necessary is an initialization function that sets the `mmio_ptr` and `mult` then calls `register_timesource()`. This can be done

```
struct timesource_t {
    char* name;
    int priority;
    enum {
        TIMESOURCE_FUNCTION,
        TIMESOURCE_CYCLES,
        TIMESOURCE_MMIO_32,
        TIMESOURCE_MMIO_64
    } type;
    cycle_t (*read_fnct)(void);
    void __iomem *mmio_ptr;
    cycle_t mask;
    u32 mult;
    u32 shift;
    void (*update_callback)(void);
};
```

Figure 5: The `timesource` structure

```
struct timesource_t timesource_hpet = {
    .name = ''hpet'',
    .priority = 300,
    .type = TIMESOURCE_MMIO_32,
    .mmio_ptr = NULL,
    .mask = (cycle_t)HPET_MASK,
    .mult = 0,
    .shift = HPET_SHIFT,
};
```

Figure 6: The HPET `timesource` structure

at any time while the system is running, even from a module. At that point, the timesource management code will choose the best available timesource using the priority field. Alternatively, a `sysfs` interface allows users to override the priority list and, while the system is running, manually select a timesource to use.

1.4.3 NTP

The NTP subsystem provides a way for the kernel to keep track of clock drift and calculate how to adjust for it. Briefly, the core interface from the timekeeping perspective is `ntp_advance()`, which takes a time interval and increments the NTP state machine by that amount, and then returns the signed parts per million adjustment value to be used to adjust time consistently over the next interval.

1.4.4 Related Changes

Other areas affected by this proposal are VDSO or vsyscall `gettimeofday()` implementations. These are chunks of kernel code mapped into user-space that implement `gettimeofday()`. These implementations are somewhat hackish, as they require heavy linker magic to map kernel variables into the address space twice. In the current code, this

dual mapping further entangles the global variables used for timekeeping. Luckily, our proposed changes can adapt to handle these cases.

The `timesource` structure has been designed to be a fairly translucent interface. Thus, any `timesource` of type `MMIO` or `CYCLES` can be easily used in a VDSO. By making a call to an arch specific function whenever the base time values change, the arch independent and specific code are able to be cleanly split. This avoids tangling the `timeofday` code with ugly linker magic, while still letting these significant optimizations occur in whatever method is appropriate for each arch.

1.5 What Have We Gained?

With these new changes, we have simplified the way time keeping is done in the kernel while providing a clear interface to the required functionality. We have provided higher resolution, nanoseconds based, time keeping. We have streamlined the code and allowed for methods which further increase gettimeofday performance. And finally, we have organized and cleaned up the code to fix a number of problematic bugs in the current implementation.

With a foundation of clean code and clear interfaces has been laid, we can look for deeper cleanups. A clear target for improvement is the soft-timer subsystem. The `timeofday` rework clearly redefines the split between system time and `jiffies`. Changing the soft-timer subsystem to human-time frees the kernel from inaccurate, tick-based time (see §3.2).

2 Human-Time

2.1 Why Human-Time Units?

Throughout the kernel, time is expressed in units of `jiffies`, which is only a timer interrupt counter. Since the interrupt interval differs between archs, the amount of time one jiffy represents is not absolute. In contrast, the amount of time one nanosecond represents is an independent concept.

When discussing human-time units, e.g. seconds, nanoseconds, etc., and the kernel, there are two main questions to be asked: "Why should the kernel interfaces change to use human-time units?" and "Why should the internal structures and algorithms change to use human-time units?" If a good answer to the latter can be found, then the former simply follows from it; a good answer can and will be provided, but we feel there are several reasons to make the interface change regardless of any changes to the infrastructure.

2.1.1 Interfaces in Human-Time

First of all, human-time units are the units of our thought; simultaneously, the units of computer design are in human-time (or their inverse for frequency measurements). The relation between human-time units and jiffies is vague, while it is clear how one human-time unit relates to another. Additionally, human-time units are effectively unbounded in terms of expressivity. That is to say, as systems achieve higher and higher granularity—currently expressed by moving to higher values of `HZ`—we simply multiply all of the existing constants by an appropriate power of 10 and change the internal resolution.

2.1.2 Infrastructure in Human-Time

The most straight-forward argument in favor of human-time interfaces stems from our proposed changes to the soft-timer subsystem. If the underlying algorithm adds and expires soft-timers in human-time units, then it follows that the interfaces to the subsystem should use the same units. But why change the infrastructure in the first place? All of the arguments mentioned in §2.1.1 apply equally well here. But our fundamental position—as alluded to in §1.2—is that the tick-based `jiffies` value is a poor representation of time.

The current soft-timer subsystem relies on the periodic timer tick, and its resolution is linked at compile time to the timer interrupt frequency value `HZ`. This approach to timer management works well for timers with expiration values at least an order of magnitude longer than that period. Higher resolution timers present several problems for a tick-based soft-timer system. The most obvious problem is that a timer set for a period shorter than a single tick cannot be handled efficiently. Even calls to `nanosleep()` with delays equal to the period of `HZ` will often experience latencies of three ticks!

On i386, for example, `HZ` is 1000, which indicates a timer interrupt should occur every millisecond. Because of limitations in hardware, the closest we can come to that is about 999,876 nanoseconds, just a little too fast. This actual frequency is represented by the `ACTHZ` constant. Since `jiffies` is kept in `HZ` units instead of `ACTHZ` units, when requests are made for one millisecond, two ticks are required to ensure one full millisecond elapses (instead of 999,876 nanoseconds). Then, since we do not know where in the current tick we are, an extra jiffy must be added. So a one millisecond `nanosleep()` turns into three jiffies.

2.1.3 A Concrete Example

To clarify our arguments, consider the example code in Figure 7.

It is clear that the old code is attempting to sleep for the shortest time possible (a single jiffy). Internally, a soft-timer will be added with an `expires` value of `jiffies` + 1. How long does this `expires` value actually represent? It is hard to say, as it depends on the value of `HZ`, which changed from 2.4 to 2.6. Perhaps it is 10 milliseconds (2.4), or perhaps it is one millisecond (2.6). What about the new kernel hacker who copies the code and uses it themselves—what assumption will they make regarding the timeout indicated? What happens when `HZ` has a dynamic value? Clearly, problems abound with this small chunk of code.

Consider, in contrast, the code after an update by the Kernel-Janitors[2] project. Now, it is unclear how the soft-timer subsystem will translate the milliseconds parameter to `msleep()` into an internal `expires` value, but in all honesty, that does not matter to the author. It is clear, however, that the author intends for the task to sleep for at least 10 milliseconds. `HZ` can change to any value it likes and the request is the same. In this case, it is up to the soft-timer subsystem to handle converting from human-time to `jiffies` and not other kernel developers (rejoice!).

These changes are in the best interest of the kernel; they will help with the long-term maintainability of much code, particularly in drivers.

3 The New soft-timer Subsystem

We have already argued that a human-time soft-timer subsystem is in the best interest of the

[2] http://www.kerneljanitors.org/

Old:

```
set_current_state(TASK_UNINTERRUPTIBLE);
schedule_timeout(1);
```

New:

```
msleep(10);
```

Figure 7: Two possible ways to sleep

kernel. Is such a change feasible? More importantly, what are the potential performance impacts of such a change? How should the interfaces be modified to accomodate the new system?

3.1 The Status Quo in Soft-Timers

A full exposition of the current soft-timer subsystem is beyond the scope of this paper. Instead, we will give a rough overview of the important terms necessary to understand the changes we hope to make. Additionally, keeping our comments in mind while examining this complex code should make the details easier to see and understand. Like much of the kernel, the soft-timer subsystem is defined by its data structures.

3.1.1 Buckets and Bucket Entries

There are five "buckets" in the soft-timer subsystem. Bucket one is special and designated the "root bucket." Each bucket is actually an array of `struct timer_list`s. The root bucket contains 256 bucket entries, while the remaining four buckets each contain 64.[3] Each entry represents an interval of jiffies; all soft-timers in a given entry have `expires` values in that entry's interval. Thus, when a timer in a particular entry is expired, all timers in that entry are expired, i.e. sorting is not necessary. In bucket one, each entry represents one jiffy. In bucket two, each entry represents a range of 256 jiffies. In bucket three, each entry represents a range of $64 \times 256 = 16384$ jiffies. Buckets four and five's entries represent intervals of $64 \times 64 \times 256 = 1048576$ and $64 \times 64 \times 64 \times 256 = 6467108864$ jiffies, respectively.

Imagine that we fix the first entry in bucket one (which we will designate `tv1`[4]) to be the initial value of `jiffies` (which we can pretend is zero). Then, treating the buckets like the arrays they are, `tv1[7]` represents the timers set to expire seven jiffies from now. Similarly, `tv3[4]` represents the timers with `expires` values satisfying $82175 \leq$ `expires` < 98559. The perceptive reader may have noticed a very nice relationship between bucket `tv[n]` and `tv[n+1]`: a single entry of `tv[n+1]` represents a span of jiffies exactly equal to the total span of all of `tv[n]`'s entries. Thus, once `tv[n]` is empty, we can refill it by pulling, or "cascading," an appropriate entry from `tv[n+1]` down.[5]

[3] This was the only possibility before `CONFIG_SMALL_BASE` was introduced. If `CONFIG_SMALL_BASE`=y, then bucket one is 64 entries wide and the other four buckets are each 16. See `kernel/timer.c`.

[4] The name given to the buckets in the code is "time vector."

[5] See `kernel/timer.c::cascade()` if you have any doubts.

3.1.2 Adding Timers

Imagine we keep a global value, `timer_jiffies`, indicating the `jiffies` value the last time timers were expired. Then take the `expires` value stored in the `timer_list`, which is in absolute jiffy units, and subtract `timer_jiffies`, thus giving a relative jiffy value.[6] Then determine into which entry the timer should be added by simple comparisons.

Keep in mind that for each bucket, we know the exact value of the least significant *X* bits, i.e. for all entries in `tv2`, the bottom eight bits are zero. Therefore, we can throw away those bits when indexing the bucket. Similarly, we also know the maximum value of any timer's `expires` field in a given bucket. Thus, we can ignore the top 18 bits in `tv2`. We are now at a six-bit value, which exactly indexes our 64-entry wide bucket! Similar logic holds true for the remaining buckets. All the gory details are available in `kernel/timer.c::internal_add_timer()`.

3.1.3 Expiring Timers

Expiration follows the addition algorithm pretty closely. Compare `timer_jiffies` to `jiffies`: if `jiffies` is greater, then we know that time has elapsed since we last expired timers and there might be timers to expire. We then search through `tv1`, beginning at the index corresponding to the lower eight bits of `timer_jiffies`, which would be timers added immediately after the last time we added expired timers. We expire all those timers and then increment `timer_jiffies`. This process repeats until either `timer_jiffies` = `jiffies` or we have reached the end of `tv1`. In the former case, we are done expiring timers and we can exit the expiration routine. In the latter case, we need to cascade an appropriate entry from a higher bucket down into tv1.[7] The strategy is to figure out which interval we are currently in relative to our system-wide initial value and re-add the corresponding timers to the system. This forces those timers which should be expired now into `tv1`. Thus, we only ever need to consider `tv1` when expiring timers. Timer expiration is accomplished by invoking the timer's callback function and removing the timer from the bucket.

[6] Relative to `timer_jiffies`, not `jiffies`.

3.2 What To Keep?

Our proposal is simple: keep the data structures and the algorithms for addition and expiration. Rather than fix the entry width to be one jiffy in `tv1`, we define a new unit: the `timerinterval`. This unit represents the best resolution of soft-timer addition and expiration. To convert from human-time units, we use a new `nsecs_to_timerintervals()` function. This allows us to preserve the algorithmic design of the soft-timer subsystem, which expects the timer's `expires` field to be an `unsigned long`. Correspondingly, we do not base our last expiration time (now stored in `last_timer_time` instead of `timer_jiffies`) and current expiration time on `jiffies`, but on the new `timeofday` subsystem's `do_monotonic_clock()`. Finally, we store the last expiration time in a more sensibly named variabled, `last_timer_time`, rather than `timer_jiffies`.

We actually require two conversion functions. On addition of soft-timers, we use `nsecs_to_timerintervals_ceiling()`, and on expiration of soft-timers, we use `nsecs_to_timerintervals_floor()`. This insures

[7] I hope all of you were highly suspicious of my claims and took a look at `kernel/timer.c::cascade()`.

that timers are not expired early. In the simplest case, where we wish to approximate the current millisecond granularity of `HZ` = 1000, the pseudocode shown in Figure 8 achieves the conversion.

In short, `timerintervals`, not `jiffies` are now the units of the soft-timer subsystem. The new system is extremely flexible. By changing the previous example's value of `TIMER_INTERVAL_BITS`, we are able to change the overall resolution of the soft-timer subsystem. We have made the soft-timer subsystem independent of the periodic timer tick.

3.3 New Interfaces

As was already mentioned, the new human-time infrastructure enables several new human-time interfaces. The reader should be aware that existing interfaces will continue to be supported, although they will be less precise as jiffies and human-time do not directly correspond to one another.

3.3.1 `add_timer`, `mod_timer`

After a careful review of the code, we believe the `add_timer()` interface should be deprecated. It duplicates the code in `mod_timer()`, using `timer->expires` as the `expires` value. Since we are moving away from the current use of `mod_timer()`, where the parameter is in jiffies, to a system using nanoseconds (a 64-bit value), we would like to avoid reworking the `timer_list` structure. `mod_timer()` is also deprecated with the new system, as we provide one clear interface to both add and modify timers, `set_timer_nsecs` (see §3.3.2).

3.3.2 `set_timer_nsecs`

This function accepts, as parameters, a `timer_list` to modify and an absolute number of nanoseconds, modifying the `timer_list`'s `expires` field accordingly. This is, in our new code, the preferred way to add and modify timers.

3.3.3 `schedule_timeout_nsecs`

`schedule_timeout_nsecs()` allows for relative timeouts, e.g. 10,000,000 nanoseconds (10 milliseconds) or 100 nanoseconds. The soft-timer subsystem will convert the relative human-time value to an appropriate absolute `timerinterval` value.

3.4 Future Direction and Enhancements

One area which has not received sufficient attention is the setting of timers using a relative `expires` parameter. That is, we should be able to specify `set_timer_rel_nsecs(timer, 10)` and `timer`'s `expires` value should be modified to 10 nanoseconds from now. Due to the higher precision of `do_monotonic_clock()` in contrast to `jiffies`, we must be careful to pick an appropriate and consistent function to determine when "now" is.

4 Dynamic Interrupt Source Management

4.1 Interrupt Source Management

With the changes to the soft-timer subsystem (see §3.2), we can address issues related to the

```
#define TIMER_INTERVAL_BITS 20
nsecs_to_timerintervals_ceiling(nsecs):
    return (((nsecs-1) >> TIMER_INTERVAL_BITS) & ULONG_MAX)+1

nsecs_to_timerintervals_floor(nsecs):
    return (nsecs >> TIMER_INTERVAL_BITS) & ULONG_MAX
```

Figure 8: Approximating HZ = 1000 with the new soft-timer subsystem

reliance on a periodic tick. With power constrained devices, we want to avoid unneccesary interrupts, keeping the processor in a low power mode as long as possible. In a virtual environent, it is useful to know how long between events a guest OS can be off the CPUs.

Many people have attacked these various problems individually and have been met with some success and some resistance. The new time system discussed in §1.5 enables the time system to do without the periodic tick and, therefore, frees the soft-timer system to follow suit. Currently, when the system timer interrupt is raised, its interrupt handler is run and all the expired timers are executed—which could be a lot, a few, or none at all. This polled approach to timer management is not very efficient and hinders improvements for virtualization and power constrained devices.

4.2 A New Approach

By changing the soft-timer implementation to schedule interrupts as needed, we can have a more efficient event based (rather than polled) soft-timer system. Our proposed changes leverage the existing NO_IDLE_HZ code to calculate when the next timer is due to expire and schedule an interrupt accordingly. This frees soft-timers from the periodic system tick and the associated overhead it imposes. Unfortunately, some decisions still have to be made as to how often we are willing to stop what we are doing and expire the timers. This period of time, the timerinterval, is configurable (see §3.2). The length of a timerinterval unit places the lower bound on the soft-timer resolution, while the hard-timer defines the upper bound of how long we can wait between expiring timers. The default of our proposed changes places the lower bound at about one millisecond, and the upper bound of the PIT, for example, would be 55 milliseconds. Hardware permitting, higher resolution timers are acheived by simply reducing the timerinterval unit length and we get the functionality of NO_IDLE_HZ for free!

4.3 Implementation

At the time of this writing, the implementation is undergoing development. This section outlines our proposed changes with some extra detail given to portions that are already implemented.

The new interrupt source management system consists of a slightly modified version of the NO_IDLE_HZ code, arch specific set_next_timer_interrupt() and a new next_timer_interrupt() routines, and calls to these routines in __run_timers() and set_timer_nsecs() (see §3.3.2).

The former next_timer_interrupt() routine has been renamed to next_timer_expires() to avoid confusion between the next timer's expires and when the the next interrupt is due to fire. The routine was updated to use the slightly modified soft-timer structures discussed in §3.2.

The arch-specific `next_timer_interrupt()` routine returns the time in absolute nanoseconds of when the next hard-timer interrupt will fire.

The arch-specific `set_next_timer_interrupt()` routine accepts an absolute nanosecond parameter specifying when the user would like the next interrupt to fire. Depending on the hard-timer being used, the routine calculates the optimal time to fire the next interrupt and returns that value to the caller. Because interrupt sources vary greatly in their implementation (counters vs. decrementers, memory mapped vs. port I/O vs. registers, etc.), each source must be treated individually. For example, older hardware that is dependant on the PIT as an interrupt source will not get higher resolution soft-timers or very long intervals between interrupts simply because the PIT is painfully slow to program (about 5.5 microseconds in our tests), and only 16 bits wide. At about 1.2 MHz the PIT's maximum delay is only 55 milliseconds. Fortunately, systems that must use the PIT can do so without incurring a penalty since the PIT interrupt scheduling function is free to reprogram the hardware only when it makes sense to do so. We have discussed the specifics of the PIT, but other interrupt sources such as local APICs, HPETs, decrementers, etc. provide more suitable interrupt sources. Since `set_next_timer_interrupt()` is arch specific, it can be `#defined` to do nothing for those archs that would prefer to rely on a periodic interrupt.

Projects such as Dynamic Ticks, Variable System Tick, High Res Timers, `NO_IDLE_HZ`, etc. attempt to solve the limitations of the current soft-timer system. They approach each problem individually by adding code on top of the existing tick based system. In contrast, by integrating dynamically scheduled interrupts with the new time and soft-timer systems discussed earlier, we create a clean, simple solution that avoids the overhead of periodic ticks and provides similar functionality.

Conclusion

We have reimplemented the `timeofday` subsystem to be independent of `jiffies`, thus resolving a number of outstanding bugs and limitations. We have also demonstrated how these changes facilitate cleanups and new features in the soft-timer subsystem. We have reoriented the time and timer subsystems to be human-time based, thus improving flexibility, readability, and maintainability.

Legal Statement

Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*