Absolute Power

By Arjan Van De Ven , Senior Staff Software Engineer at Intel.

Abstract:

Power consumption is a hot topic — from laptop, to datacenter. Recently, the Linux kernel has made huge steps forward in power conservation. This article focuses on the so-called "tickless idle" feature being integrated into the *2.6.21* and *2.6.23* kernel versions.

How to Save Power

Before explaining some of the improvements the kernel has undergone in the last few months, it's useful to list some general power-saving strategies and techniques:

*Turn the device off when not in use. Obviously, when a component isn't used, the operating system should turn it off as much as possible.

*Create long idle time between activities. Processors (but also disks and other components) have several degrees of power-saving when idle. If a component is idle for a long time, it can go into the deepest possible power-saving state; but if it's only idle for a short time, it can only go into a shallow power-saving state.

Each type of component defines what's meant by "long time." For processors, a "long time" is roughly twenty milliseconds. For disks, a "long time" is about two seconds. While the scale differs, the net effect is the same: it's better to do all the work in one step than to perform several smaller steps.

C-states

Several of the new kernel improvements center around processors, so it's appropriate to explain the so-called *C-states* present in contemporary (mobile) processors from the various vendors.

"C-states" is a term derived from the *ACPI* specification. The basic idea is that a processor, when it's not executing instructions, can save power in several ways, called "states". Each state has a different tradeoff, balancing power-saving versus latency and performance.

Table One is an example taken from the Intel *Core 2 Duo* processor datasheet. While these power consumption numbers denote the maximum possible (or "TDP") values and not actual values as you will see them on your laptop, the metrics give a reasonable indication of the differences in power usage in various C-states.

TABLE ONE: *The* C-states of the Intel *Core 2 Duo* processor

C-state	Max Power consumption
C0 (busy wait)	35 Watts (TDP)
C1	13.5 Watts
C2	12.9 Watts
C3	7.7 Watts
C4	1.2 Watts

While the power is reduced for higher number (deeper) C-states, this power reduction comes at a price. The deeper the C-state, the longer it takes to leave it, and the more energy this transition costs.

For example, if your computer is going in and out of idle at a high frequency (say every millisecond), and the kernel decides to go to the C3 or C4 state during the short idle periods, it could easily take the processor 185 microseconds to wake up. In addition, the wakeup would consume more energy than it conserved during its very brief resting period. However, had the kernel decided to use the C2 state, the processor would come out of its nap in maybe ten microseconds. While C2 in a steady state uses much more power than the C4 state, the energy costs to going into or out of C2 are much lower, so for this scenario, C2 would actually be more energy efficient.

Now, if your system is actually idle for longer periods of time, say 20 milliseconds, then C4 becomes the clear winner; the one time energy cost of the transition is dwarfed by the difference in energy savings, unless the exit latency cannot be tolerated. If you're doing high-precision audio recording, a 185 microseconds delay may well be more than you want to cope with, in which case the kernel should pick C2 again.

If you want to see what C-states your Linux laptop has and what the exit latencies for each state are, issue the following command:

```
$ cat /proc/acpi/processor/*/power
```

Kernel things that spoil this, and recent changes

Now that you've seen that to get good power savings from the processor, you want to be idle for at least 20 milliseconds, it's time for the kernel to ruin the party.

Until *2.6.21,* the Linux kernel programmed the *PIT* chip of the PC (or equivalent on other architectures) to generate interrupts at a regular interval of either 250 Hz (A four millisecond interval) or 1000 Hz (a one millisecond interval). This regular interrupt, often called the "timer tick," has several tasks:

- **Increment the "jiffies" variable.** This global variable represents the kernel's internal notion of time and is used in many places all across the kernel.

- **Process accounting.** For each timer tick, the kernel looks at which process is running (if any) and increments the CPU usage counter for the process. This information (including the idle information) is used by programs such as *top* to display which programs are consuming CPU time.]
- **Process scheduler time slicing.** The scheduler gives each process that is running a so-called "time slice," or an amount of time that it's allowed to run before other processes get their turn. The timer tick interrupts the process at the end of its time slice and the scheduler then gives the CPU to another process at the end of the timer tick processing.
- **Deferred events(timers).** Many things in a kernel are of a "do this after 50 milliseconds" or "if nothing else happens in three seconds, call this function to recover" nature. In Linux kernel speak, these are called "timers." With each timer tick, the kernel looks at the queue of outstanding future timer events to see if any have become due (this sounds expensive, but the timer list is sorted so it's a cheap operation), and will process the ones whose turn it is.

This timer tick approach has a certain elegance in its simplicity and has served Linux well since the early 1990s. However, in the light of the C-state hardware capabilities, a regular one millisecond or four millisecond interrupt has the effect of waking the CPU frequently from the deep sleep states (or even preventing the CPU from ever entering the deepest sleep states), which obviously makes the system consume more power than necessary.

Enter the tickless idle feature. In the last year or so, Thomas Gleixner and others have worked on a feature that's called "tickless idle" or just "tickless." It is the removal of the regular timer tick when the system is idle. This feature has become part of the 2.6.21 kernel for the *i386* architecture, and *2.6.23* will likely have the *x86_64* version as well.

Of the four functions of the timer tick, only two are really relevant when the CPU is idle: updating the kernel's notion of what time it is (jiffies) and processing deferred events (timers). When Linux was created, the PIT was more or less the only usable clock device on the PC; however, a modern PC has a wide range of clocks and clock-like devices. A fundamental step in the Linux kernel architecture in the last year was to split these devices into two categories: Clock sources and Event Sources.

Clock Sources are devices that can answer the "what time is it right now" question, and the clock source infrastructure abstracts away the differences between the various devices toward the rest of the kernel. The source code for the clock source infrastructure lives in *kernel/time/clocksource.c.*

Event Sources are those devices that can generate an interrupt after a software-specified amount of time. The event source layer abstracts these various devices for the rest of the kernel, and picks the best one for a task based on the various capabilities (in terms of precision, accuracy, and maximum duration) of each of the devices in the system. The source code for the event source infrastructure lives in *kernel/time/clockevents.c.*

This separation and the two hardware abstraction layers make the exercise of getting rid of the timer tick during idle a relatively simple task:

- Instead of updating jiffies by one every timer tick, jiffies gets updated to what the value should be based on the "what time is it" question to the clock source layer.
- Instead of looking every millisecond if any timer is due for processing, the kernel calculates when the first timer is due, and asks the event source layer to give a single interrupt at exactly the right time.

In order to fulfill the other two tasks of the timer tick (accounting and time-slicing), the timer tick is kept running when the CPU is not actually idle.

As with most simple things, the devil is in the details. While modern PCs have many clock-like components, the reliability of each of those varies wildly by manufacturer, technology generation, and even BIOS settings. This unreliability is the primary answer to the "why did it take so long" question. Thankfully, for most systems, the so-called HPET component of the chipset is the most promising and reliable event source.

The High-Precision Event Timer

The "High Precision Event Timer" (HPET) is often referred to as "Multimedia Timer." The actual HPET device is part of the chipset of modern PCs, and is capable of generating events and keeping track of time. An HPET has multiple "channels," and each channel has has its own timer that generates events independently.

The "HP" in HPET points out the biggest difference with the PIT timer: the HPET generates far more fine-grained events than the PIT does, and is thus much more useful for things like synchronizing audio with video during gaming or movie playback.

Unfortunately, many BIOSes disable the most reliable event source (HPET) by default for maximum compatibility with another popular operating system. On many systems, the HPET would be the only reliable event source, so the default BIOS setting is very unfortunate. The good news is that at least for Intel and NVidia chipsets, patches are being merged into the kernel that enable the HPET regardless of the BIOS setting.

In the Linux kernel, the driver for the HPET is in *drivers/char/hpet.c,* with an architecture specific portion in *arch/i397/kernel/hpet.c.*

By not having a regular timer tick when the processor is idle, it is theoretically possible to have really long periods of idle as long as there are no future timers planned. Unfortunately, in practice in a current Linux distribution, both the kernel and userspace applications set so many timers that it is not uncommon to have 500 or more of such events per second. The system can go to a one millisecond sleep time to an approximately two millisecond sleep time. This would obviously provide only minimal power gain over the kernel with the regular tick.

There are roughly two areas where this needs fixing: the kernel and the userspace side.

Fixing High-Frequency Events in the Kernel

On the kernel side, there are drivers and subsystems that have timers that are just "randomly short," and can be increased without any noticeable effects to the system or the user.

But doing that isn't quite enough. A second technique is to align as many timers as possible to happen at the same time. Many timers are of the "once every two seconds" type, but the user of the timer doesn't really care* exactly* when the timer happens. By making sure all timers of this class only fire at the start of a full second, the processor can (if no other timers exist) sleep the rest of the second without being interrupted.

The API that the kernel has for this is called `round_jiffies()`, and there are two primary functions that drivers and subsystems should use:

```
unsigned long round_jiffies(unsigned long time);
unsigned long round_jiffies_relative(unsigned long delta);
```

The `round_jiffies()` function, as the name suggests, rounds off its argument such that all callers of `round_jiffies()` that would be in the same second of time end up at exactly the same moment.

`round_jiffies_relative()` performs this exact same function, with the difference that `round_jiffies()` operates on absolute times (the "jiffies" variable), and `round_jiffies_relative()` operates on time deltas. This matches the diversity of kernel APIs for future work, some of which take absolute time, while others take time deltas.

Fixing High-Frequency Events

Unfortunately, the situation in user space is a much bigger problem. Many desktop programs behave very badly and have a large number of timers that aren't really needed. The most frequent scenario is that code polls for something, but could simply opt for an event if the programmer had bothered. In defense of the userspace programmers, until the kernel went tickless, this behavior not only was not affecting the system negatively, it wasn't even possible to see what was happening.

The environment has changed with the tickless kernel. Programs that poll frequently hurt power consumption. However, such errant code can also be identified readily with a new tool. The Linux *PowerTOP* program (http://www.linuxpowertop.org) is a tool that developers and users can use to see which software pieces are taking the processor out of its idle sleep state. In addition, PowerTOP shows how well the system is using the deeper C-states and provides tuning suggestions for the system.

Even when all the "needlessly polling" behavior gets fixed, user space programs may have housekeeping tasks that just have to happen once in a while. Similarly to the `round_jiffies()` infrastructure in the kernel, there is an API in recent versions of the *glib* library that provides rounding and grouping timers: `g_timeout_add_seconds()`.

For more information and to participate in power management technologies, visit http://www.lesswatts.org