What Does an Idle CPU Do

Gustavo Duarte, Oct 29th, 2014

In the last post I said the fundamental axiom of OS behavior is that *at any given time*, exactly **one and only one task is active** on a CPU. But if there's absolutely nothing to do, then what?

It turns out that this situation is extremely common, and for most personal computers it's actually the norm: an ocean of sleeping processes, all waiting on some condition to wake up, while nearly 100% of CPU time is going into the mythical "idle task." In fact, if the CPU is consistently busy for a normal user, it's often a misconfiguration, bug, or malware.

Since we can't violate our axiom, *some task needs to be active* on a CPU. First because it's good design: it would be unwise to spread special cases all over the kernel checking whether there *is* in fact an active task. A design is far better when there are *no exceptions*. Whenever you write an `if` statement, Nyan Cat cries. And second, we need to do *something* with all those idle CPUs, lest they get spunky and, you know, create Skynet.

So to keep design consistency and be one step ahead of the devil, OS developers create an **idle task** that gets scheduled to run when there's no other work. We have seen in the Linux boot process that the idle task is process 0, a direct descendent of the very first instruction that runs when a computer is first turned on. It is initialized in rest_init, where init_idle_bootup_taskinitializes the idle **scheduling class**.

Briefly, Linux supports different scheduling classes for things like real-time processes, regular user processes, and so on. When it's time to choose a process to become the active task, these classes are queried in order of priority. That way, the nuclear reactor control code always gets to run before the web browser. Often, though, these classes return `NULL`, meaning they don't have a suitable process to run – they're all sleeping. But the idle scheduling class, which runs last, never fails: it always returns the idle task.

That's all good, but let's get down to just *what exactly* this idle task is doing. So here iscpu_idle_loop, courtesy of open source:

cpu_idle_loop

```
while (1) {
    while(!need_resched()) {
        cpuidle_idle_call();
    }

    /*
      [Note: Switch to a different task. We will return to this loop when the
      idle task is again selected to run.]
    */
    schedule_preempt_disabled();
}
```

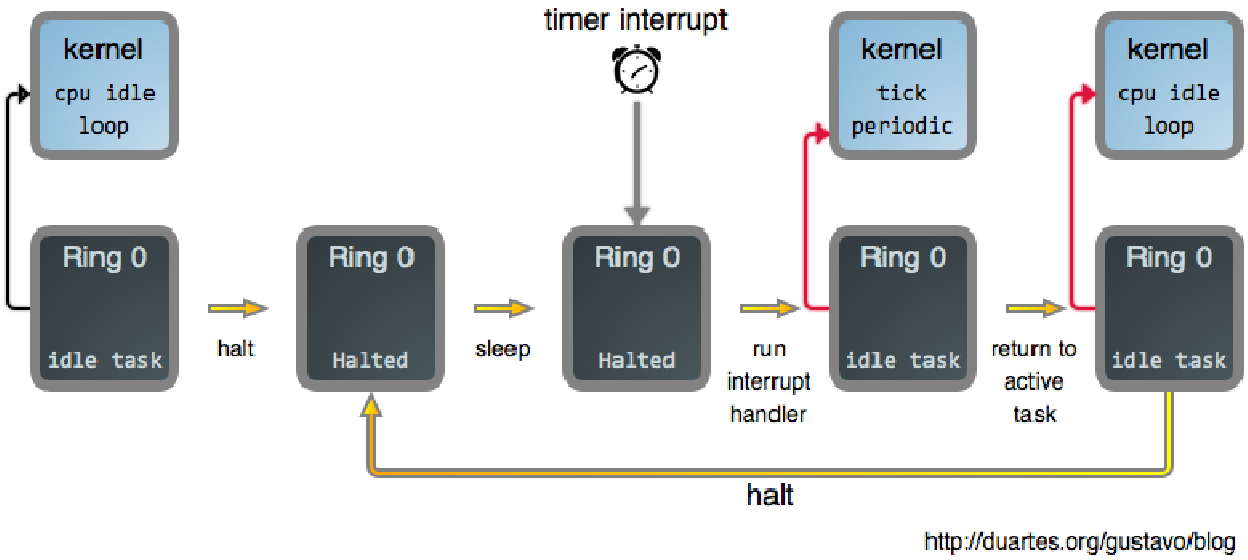
I've omitted many details, and we'll look at task switching closely later on, but if you read the code you'll get the gist of it: as long as there's no need to reschedule, meaning change the active task, stay idle. Measured in elapsed time, this loop and its cousins in other OSes are probably the most executed pieces of code in computing history. For Intel processors, staying idle traditionally meant running the halt instruction:

native_halt

```
1static inline void native_halt(void)
2{
3  asm volatile("hlt": : :"memory");
4}
```

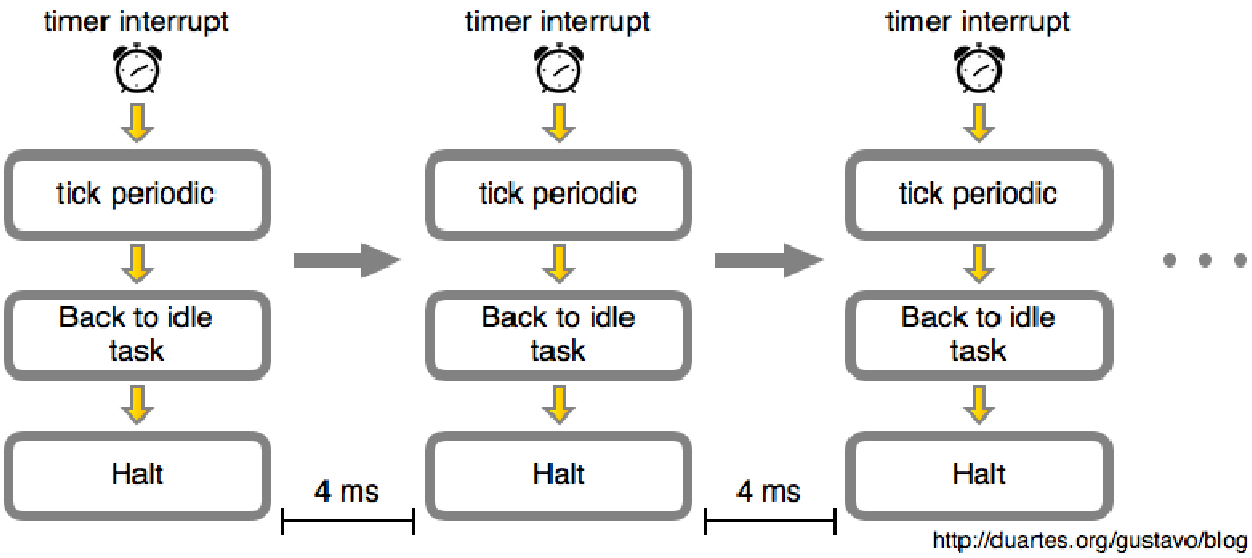
`hlt` stops code execution in the processor and puts it in a halted state. It's weird to think that across the world millions and millions of Intel-like CPUs are spending the majority of their time halted, even while they're powered up. It's also not terribly efficient, energy wise, which led chip makers to develop deeper sleep states for processors, which trade off less power consumption for longer wake-up latency. The kernel's cpuidle subsystem is responsible for taking advantage of these power-saving modes.

Now once we tell the CPU to halt, or sleep, we need to somehow bring it back to life. If you've read the last post, you might suspect *interrupts* are involved, and indeed they are. Interrupts spur the CPU out of its halted state and back into action. So putting this all together, here's what your system mostly does as you read a fully rendered web page:



Other interrupts besides the timer interrupt also get the processor moving again. That's what happens if you click on a web page, for example: your mouse issues an interrupt, its driver processes it, and suddenly a process is runnable because it has fresh input. At that point `need_resched()` returns true, and the idle task is booted out in favor of your browser.

But let's stick to idleness in this post. Here's the idle loop over time:



In this example the timer interrupt was programmed by the kernel to happen every 4 milliseconds (ms). This is the *tick period*. That means we get 250 ticks per second, so the *tick rate*or *tick frequency* is 250 Hz. That's a typical value for Linux running on Intel processors, with 100 Hz being another crowd favorite. This is defined in the `CONFIG_HZ` option when you build the kernel.

Now that looks like an awful lot of pointless work for an *idle CPU*, and it is. Without fresh input from the outside world, the CPU will remain stuck in this hellish nap getting woken up 250 times a second while your laptop battery is drained. If this is running in a virtual machine, we're burning both power and valuable cycles from the host CPU.

The solution here is to have a dynamic tick so that when the CPU is idle, the timer interrupt is either deactivated or reprogrammed to happen at a point where the kernel *knows* there will be work to do (for example, a process might have a timer expiring in 5 seconds, so we must not sleep past that). This is also called *tickless mode*.

Finally, suppose you have *one active process* in a system, for example a long-running CPU-intensive task. That's nearly identical to an idle system: these diagrams remain about the same, just substitute the one process for the idle task and the pictures are accurate. In that case it's still pointless to interrupt the task every 4 ms for no good reason: it's merely OS jitter slowing your work ever so slightly. Linux can also stop the fixed-rate tick in this one-process scenario, in what's called adaptive-tick mode. Eventually, a fixed-rate tick may be gone altogether.

That's enough idleness for one post. The kernel's idle behavior is an important part of the OS puzzle, and it's very similar to other situations we'll see, so this helps us build the picture of a running kernel. More next week, RSS and Twitter.