Let Sleeping CPUs Lie — S0ix

BY AYMERIC WIBO

Modern laptops promise a kind of magic. Shut the lid or press the sleep button, toss it in a backpack, and hours, days, or weeks later, it should wake up as if nothing happened with little to no battery drain. This sounds like a fairly trivial operation — y'know, you're literally just asking for the computer to do nothing — but in that quiet moment when the fans whir down, the screen turns dark, and your reflection stares back at you, your computer and all its little components are actually hard at work doing their bedtime routine.

History of Power Management, or: The Long Road to Doing Nothing Efficiently

Let's take a stroll through the history of power management on computers to better understand where we came from.

The first "mobile" computers were more like semi-portable machines that, if they had batteries at all, weren't really expected to stay unplugged from AC for very long. With the Osborne 1, while it did have an aftermarket battery expansion pack, it was expected that you'd be powering it up and down as you ran between outlets.

The first true mobile computer, as we know it, was probably the Dulmont Magnum, released in 1983. It introduced "suspend-to-RAM" functionality for the first time, which in its case meant that components could be switched off while memory was retained in battery-backed CMOS RAM. A few other DOS machines shipped proprietary suspend solutions, but for the most part, they were fairly ad hoc.

Zzzzz

A few other DOS machines shipped proprietary suspend solutions, but for the most part, they were fairly ad hoc.

In 1985, Intel released the i386 chips, the SL variant of which (itself released in 1990) targeted power-efficient machines such as laptops. This chip introduced SMM (System Management Mode), which allowed CPU execution to switch from the operating system to a region of memory written to and locked by the BIOS (the SMI handler, starting at SMBASE), which would run with higher privileges than ring 0 to run power management code.

This switch-over in execution was initiated by an SMI (System Management Interrupt), which could be asserted either by hardware (e.g., a power button press) or by the operating system via I/O access to a specific port. This method of power management had major flaws, not least that the operating system was a passive actor in this arrangement and completely at the mercy of the firmware. Indeed, if the SMI were asserted by hardware, the OS

would be entirely sidestepped, and execution would be essentially flash-frozen, and it would thus be fully unable to prepare for suspend entry.

SMI handlers were also a very fruitful source of exploits — notably many of the "implants" from the NSA's infamous ANT catalogue — which remotely flashed BIOSes, allowing a malicious payload to be loaded at SMBASE, which would then run periodically during the system's operation.

Introduced in 1992, APM (Advanced Power Management) finally provided a standardized power management interface for x86 platforms. Instead of the OS being interrupted at the hardware's whim, when an event that would've previously triggered an SMI occurs, the firmware simply notifies the OS via a power management event. The OS then decides what to do in response, if anything, and can take all the time it needs to prepare before actually requesting a power transition from the firmware via an APM function (usually via BIOS interrupt 0x15).

This was an absolutely massive improvement over SMM for power management. One issue still persisted, though: APM still assumed BIOS vendors would write software not riddled with bugs, and these APM functions were asking a lot of these BIOSes.

APM was eventually superseded by ACPI (Advanced Configuration and Power Interface) in 1996, which was a much more comprehensive standard and interface for hardware and firmware to communicate with the operating system and provide their functionality. It is still used on modern machines today.

ACPI aimed to lug even more power management responsibilities from the firmware to the OS side of things, which it named "OSPM" (OS-directed Power Management). It defines multiple "S"-states, which are high-level power states the system could be in.

Most notable are S0 (the computer is fully awake and running), S3 (the computer is suspended-to-RAM), S4 (the computer is suspended-to-disk, or hibernated), and S5 (the computer is shut down). Fundamentally, however, suspending through S3 follows a similar approach to APM, since the "OS is informed of a power event, then deciding and preparing for suspend, and finally asking firmware to actually suspend" flow is maintained.

But as our hardware progresses and we come to expect more from it as the spoiled consumers we are, this flow, and actually the whole "the system is either fully on or suspended" paradigm, is starting to show its limitations.

With the advent of smartphones and similar devices, we have come to expect our machines to be always connected, even when the lid is closed. We want to receive notifications about new uploads from our favourite AI slop content creators, emails from that 500th newsletter we're subscribed to, and to wake the machine to notify us of calendar events.

To do this, we need to resume every so often or upon receiving a wake-on-LAN packet, just to, e.g., look at it a little more deeply and decide whether we actually want to wake the system for real or snooze and go back to sleep immediately. And this is a use case S3 is just too inflexible to properly accommodate — S3 is extremely costly in terms of energy consumed and time taken for firmware to start everything back up again.

S3 is all-or-nothing; if we want to wake from it, we pretty much have to act as if we were fully resuming each time.

If only there were a solution...

Enter... S0ix

Somewhere around 2018, in the everlasting quest to make computers do absolutely nothing as efficiently as possible and better and faster than ever before, Intel introduced S0ix. S0ix is actually a collection of states (S0i1, S0i2, S0i3, potentially &c), but which crucially all keep the system in the S0 — or awake — ACPI state. The bigger the "x", the deeper the sleep and the more the power saving, with the goal generally being S0i3. S0i0 is sometimes referred to as equivalent to S0, but I think that's somewhat confusing, so I won't use that term.

Contrary to S3 or APM in the past, the OS never actually explicitly asks the firmware to enter S0ix (erm, well, actually in practice it kind of does, but we'll get to that later). Instead, it just creates the conditions necessary for entry into an S0ix state and then lets the firmware decide when to transition the CPU to a given S0ix state on its own.

Those necessary conditions are:

- The devices (i.e., various components such as the GPU or USB controllers) on the machine are sufficiently powered down for a given S0ix state.
- The CPUs are all idling (that is, doing nothing) and have entered a low-power idle state.

Let's take a look at how we power down devices.

S0ix-enabled machines expose an ACPI device known as the SPMC (System Power Management Controller, sometimes also known as the PEP). Functions known as DSMs (Device-Specific Methods) in ACPI-land may be called on this SPMC device to provide hints to the platform about the OS's intention to suspend, but can also be used to get constraints about what the minimum power state a device needs to be in for firmware to want to enter an S0ix state.

Device power states are known as "D-states", and they range from D0 (fully on) to D3hot (off but powered) and D3cold (off but unpowered). We'll generally want to transition a device into D3cold anyway even if its D-state constraint doesn't require it to be if doing a full suspend, but we might want to keep a device in a more wakeful D-state to serve as a wake device — of course, if we power off a device meant to signal the system we would like to wake up, we would never be able to wake up.

As for idling CPUs, we do this by entering a special software state known as suspend-to-idle (aka s2idle).

So, to make things simple, these constraints from the SPMC just tell us whether keeping a device on during suspend would impede S0ix entry. As for idling CPUs, we do this by entering a special software state known as suspend-to-idle (aka s2idle), which pretty much takes the place S3 used to. Essentially, we just do this by stopping the scheduler clocks and pausing whatever the OS was executing to put all the CPUs in the system into a low-power idle state. These low-power idle states are known as C-states, and their entry methods, along with other information, are obtained either from the older `_CST` ACPI object or from its more modern replacement, the `_LPI` object.

Exiting the idle state is done by interrupting the CPU, so before idling, we disable all CPU interrupts except those we actually want to use to wake the CPU. These wake interrupts are usually just the SCI (ACPI System Control Interrupt), which let us know of GPEs (General Purpose Events), which could be something like pressing the power button or opening the lid (coarsely discriminated by the "GPE number").

SCIs can trigger for a variety of non-wake reasons, so we also configure ACPI as best we can to only trigger SCIs on actual wake GPEs, which is sometimes easier said than done. But we'll get to that later.

In theory, if both conditions are met, firmware can cut power to the CPU package and enter a S0ix state. In practice, though, things are a little more complicated and hardware-specific.

On AMD systems, for instance, the power management firmware (which they call PMFW) runs on a small on-die LatticeMico32 core known as the SMU (System Management Unit), which you'll also see more recently referred to as the "MP1" chip. This is ultimately responsible for clock and power gating the CPU package, and thus for transitioning the package in and out of S0ix states.

When the OS actually wants to suspend, it has to send the PMFW a little command to hint at it to enter S0i3. The SMU has its own requirements that must also be met, in addition to the regular S0ix requirements. The most notable of these requirements is for the GPU and USB4 controllers: recent versions of the amdgpu driver use different paths when entering S3 or S0ix, and USB4 controllers need to be explicitly powered off by an NHI driver. This is necessary even if the operating system doesn't support USB4, because the controllers are activated by the pre-OS connection manager (required for USB4 to work before the OS boots) and remain active by the time the OS boots.

In theory, if both conditions are met, the firmware can cut power to the CPU package and enter a S0ix state.

The SMU also communicates debugging information, letting the user know if the system entered S0i3 successfully and, if not, what prevented it from doing so. This information is exposed through the `dev.amdsmu.0.ip_blocks` and `dev.amdsmu.0.metrics` sysctl trees on FreeBSD.

Since the PMFW is running purely on-die, to let the wider platform know of our will to enter S0ix, we have to tell the SPMC we want to enter suspend through those "display off" and "entry" notification DSMs alluded to earlier. This usually prompts the platform to provide visual feedback to the user that the system is suspended, such as slowly fading the power button LED in and out on Framework laptops. It can also do some more subtle stuff, such as reducing the frequency of some GPEs that would otherwise constantly wake the CPU from idle and thus S0i3.

Indeed, on the Framework laptops, the EC (Embedded Controller) sends a GPE for the battery device about once per second in normal operation, and we cannot suppress this before entering suspend, as the battery GPEs share a number with the lid GPE. So if we wanted to disable those noisy battery GPEs, we'd have to sacrifice being able to wake the system with the lid, which is no good.

Hinting the SPMC of entry to suspend tells the EC to send a battery GPE only about once a minute, which is perfectly fine considering how quickly we can exit S0i3 and re-enter. And having our system wake every so often can actually be fairly convenient for tracking what the power draw and battery state look like over time while we're in suspend-to-idle (even though we could probably replicate this with the RTC alarm, but I digress).

Since the CPU might be woken from idle for a whole host of reasons that don't necessarily mean we want to wake the whole system, we put the CPU in an "s2idle loop" when it's idling. This allows us to actually look at the GPE we got from the system and decide whether it's worth waking for. This GPE could be as simple as noisy battery events or as complex as a wake-on-LAN event from the NIC, where, as mentioned earlier, we might want to examine a packet in a bit more detail before deciding whether to wake the system.

All in all, the full flow for entering suspend on S0ix enabled systems is the following:

- Suspend all userspace execution and mounted filesystems.
- Go through the device hierarchy and suspend all devices, usually by setting them to one of the D3 states and at least setting them to the minimum required D-state for S0ix as reported by the SPMC.
- Halt the scheduler clocks on all CPUs and disable all interrupts, except for SCIs.
- Start the "s2idle loop" and idle the CPUs.

Zzzzz

Why should I bother with this newfangled S0ix stuff when I can stick to S3?

Resuming is pretty much the first three steps, but in reverse.

Now, dear reader, at this point you might be thinking to yourself "well gee wizz, that's good and all but I don't need my laptop to respond to notifications while its sleeping. Why should I bother with this newfangled S0ix stuff when I can stick to S3?", and to that the unfortunate response is that most modern systems nowadays don't come with S3 support at all anymore, having completely replaced it with S0ix.

Supporting both is very complex for vendors, so most don't bother, especially considering most major operating systems support S0ix just fine now. Luckily, FreeBSD is well on its way to adding support for suspend-to-idle and S0ix.

You can see if your system supports suspend-to-idle by checking the **`kern.power.supported_stype`** sysctl, and you can configure ACPI to enter suspend-to-idle on certain events through the **`hw.acpi`** sysctl tree. Then, if your system supports it and FreeBSD supports it, an S0ix state should be entered automatically.

Currently, there is no unified way of checking this, but if you're on an AMD system with an SMU chip, you can use the aforementioned SMU sysctls.

Hibernate and ACPI S4

After all this work getting S0i3 to work, you might be a little disappointed to learn that in S0i3 (and S3, too, for that matter), you're not even getting very good power savings. There's a lot of variance between systems in this regard, but a system lasting only a few days in S0i3 is not out of the ordinary. That's where hibernate, or suspend-to-disk, comes in.

When suspending to memory, as in S3 or S0i3, your machine still has to constantly refresh DRAM, which isn't an inconsequential source of power drain, and this actually in-

creases as you add more memory to your system. With suspend-to-disk, you are copying memory to a hard drive and effectively completely turning off your computer — including DRAM — as if you were shutting it down (well, for all intents and purposes you *are* shutting it down). This means your machine is consuming almost no power at all. The only components actively consuming power are things like your EC if your machine has one, but that's just the same as if your machine were powered off anyway.

In ACPI S-state parlance, hibernate is S4.

At the very beginning, when dinosaurs still roamed the Earth, BIOS vendors implemented this in firmware via something called S4BIOS to ease adoption. In this arrangement, the BIOS was entirely responsible for copying OS memory to disk, much as suspend-to-RAM worked pre-APM.

Though FreeBSD supports S4BIOS, it hasn't been relevant for a long time, as most operating systems nowadays only support regular S4 hibernate. In S4, the operating system does most of the work of copying its memory to disk, while the firmware essentially just powers off the machine. In fact, you can totally implement hibernate with no firmware S4 support at all, just using your platform's poweroff functionality (which means S5 on ACPI platforms, or whatever else on non-ACPI platforms).

The main drawback of hibernate is the latency at entry and exit. You have to wait for the operating system to copy all pages to disk, then, on resume, go through much of the boot process and finally restore those pages from disk to memory. But we can actually combine the advantages of S0ix and hibernate into a hybrid approach!

With suspend-to-disk, you are copying memory to a hard drive and effectively completely turning off your computer

By setting an RTC alarm, we can let the system stay in S0i3 for, say, an hour, and then wake up again and enter hibernate if the user has not woken the system up in the meantime. This means that if we wake up less than an hour before suspending, we get the super-fast resume times from S0ix, but if we leave the system suspended overnight or for multiple days, we get the much *much* improved battery life of hibernate. That does mean it will take significantly longer to resume the next time we wake the machine if we didn't wake it during the hour, but that's a pretty good compromise, all things considered.

We can also do other cool stuff with this hybrid approach, like configuring the ACPI `_BLT` (Battery Level Threshold) control method to wake the machine once the battery has reached a certain threshold — say 5% — so that we can enter hibernate and prevent a hard poweroff if the battery runs out while we are in S0ix.

State of Things in FreeBSD

The FreeBSD Foundation, as part of its "Laptop Support and Usability Project" effort, has been spearheading sponsored work on S0ix and hibernate. Much of the suspend-to-idle and S0ix code has already landed, and hibernate has recently been kicked off, but entry into S0i3 and especially resumption from it are not entirely reliable yet, and the necessary parts of the USB4 driver to make the SMU happy have not yet landed.

Furthermore, much of the focus has been on AMD systems and not so much Intel (though work on an initial PMC driver — their SMU equivalent — has recently begun). Even among AMD platforms, the only testing has so far occurred on Phoenix, with the `amdsmu` driver theoretically only supporting Cezanne, Rembrandt, and Phoenix chips.

The only USB4 controller to have been tested with the work-in-progress USB4 driver (`thunderbolt`) is Pink Sardine (found on Phoenix systems), though any generic quirkless USB4 controllers should also be supported.

Much of the remaining work at the time of writing focuses on reliably entering the deepest C-state (C3 on Phoenix systems), as CPUs often wake unexpectedly, which in turn causes the system to exit S0i3. But fingers crossed, FreeBSD will soon have support for modern standby and hibernate, and be fully caught up with the likes of Linux and Windows!

For a more technical and focused look at S0ix and its implementation in FreeBSD, plus some additional references, you can take a look at my blog: https://obiw.ac/s0ix

---

AYMERIC WIBO is a software freelancer from Belgium specializing in kernel development and graphics programming. He has been involved with FreeBSD since high school and is currently working for the FreeBSD Foundation on laptop power management and at Klara Inc.