Anomalies in Linux Processor Use

RICHARD L. SITES

Richard L. Sites is a semi-retired computer architect and software engineer. He received his PhD from Stanford University several decades ago. He was co-architect of the DEC Alpha computers and then worked on performance analysis of software at Adobe and Google. His main interest now is to build better tools for careful non-distorting observation of complex live real-time software, from datacenters to embedded processors in vehicles and elsewhere. dick.sites@gmail.com

Careful observation of Linux dynamic behavior reveals surprising anomalies in its schedulers, its use of modern chip power-saving states, and its memory allocation overhead. Such observation can lead to better understanding of how the actual behavior differs from the pictures in our heads. This understanding can in turn lead to better algorithms to control dynamic behavior.

We study here four such behaviors on x86-64 systems:

1. Scheduling dynamics across the Completely Fair Scheduler, the real-time FIFO scheduler, and the real-time Round-Robin scheduler
2. Dynamic use of `mwait`-sleep-wakeup to save power
3. Dynamic CPU clock frequency changes to save power
4. Invisible cost of heap allocation just *after* allocation

In each case, the interaction of Linux and the underlying hardware can be improved in simple ways.

The software observation tool is KUtrace [1–3], which timestamps and records every transition between kernel-mode and user-mode execution in a live computer system, using less than 1% CPU and memory overhead and thus observing with minimal distortion. Each transition is recorded in just four bytes in a kernel trace buffer—20 bits of timestamp and 12 bits of event number (syscall/sysreturn, interrupt/return, fault/return numbers plus context-switch new process ID, and a handful of other items). Everything else is done by postprocessing a raw binary trace. Depending on the processor, each trace entry takes an average of 12–20 nsec to record, about 30 times faster than `ftrace` [4]. The robustly achieved design point is to handle 200,000 transitions per second per CPU core with less than 1% overhead. I built the first such system at Google over a decade ago, and it and its offspring have been used in live production datacenters since.

Linux Schedulers: Not Completely Fair

The Linux CPU schedulers juggle program execution by assigning tasks to CPU cores at various times. The Completely Fair Scheduler (CFS) runs each task at equal speed, each getting CPUs/tasks speed over time. The FIFO real-time scheduler runs each task in FIFO order to completion or until it blocks. The Round-Robin real-time scheduler runs like FIFO but imposes a maximum time quantum, moving tasks to the end of a run queue in round-robin fashion at quantum boundaries.

On a four-core Intel i3-7100 processor (actually two physical cores hyperthreaded) running the Linux 4.19.19 LTS (long-term support) kernel version, I ran 1 to 12 identical CPU-bound threads and observed the true scheduling behavior [5]. Each thread repeatedly checksums a 240 KB array that fits into a per-core L2 cache. From the Linux documentation, I expected the resulting timelines for more than four tasks to show each task running periodically and all completing at nearly the same time. Not so.

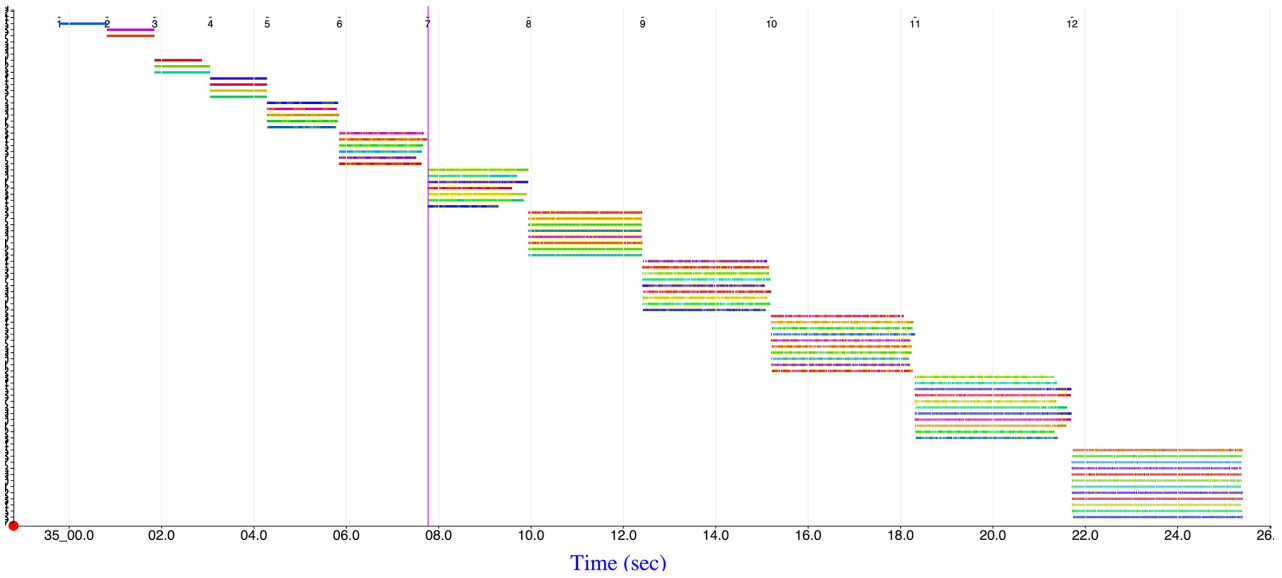

Figure 1: Running groups of 1 to 12 compute threads under CFS. The main program spawns one thread at the top left, and when that completes one second later it spawns two threads, then three, etc. With only four logical CPU cores, the scheduler starts its real work with five or more threads. The vertical line marks the group of seven that is expanded in Figure 2.

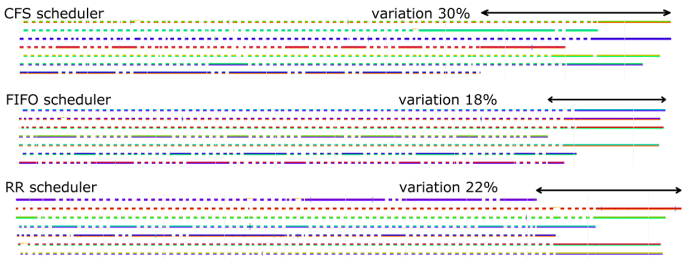

Figure 2: Running groups of seven compute-bound threads under the three Linux schedulers, shown over about two seconds total. In each case, the thread-preemption times vary substantially, and some threads complete unexpectedly much sooner than others—arrows depict the largest differences.

Figure 1 shows groups of 1 to 12 threads running under CFS. As the last thread of each group finishes, the next group starts, consuming about 26 seconds in all. The pictures for the other schedulers look similar at this scale. (Note that all the figures in this article appear in color in the online version.)

Looking at just the seven-thread group, Figure 2 shows it for each of the three schedulers. The smallest dashes are 12 ms execution periods (the quantum), chosen by the scheduler based on four cores and timer interrupts every 4 ms. This simple example does not stress the differences that the real-time schedulers would provide in a mixture of batch and real-time programs, but it does highlight their underlying dynamics.

The documentation for these schedulers did not lead me to expect that some tasks would run uninterrupted for a dozen quanta or more, nor did it lead me to expect a 20–30% variation in completion time between the earliest and latest ones. None of this approaches "completely fair." Observing these actual dynamics can lead to better algorithms.

Deep Sleep: Too Much Too Soon

Our second study concerns power-saving dynamics. Modern software passes hints to the chip hardware that nothing interesting will be executing for a while, so the hardware might well want to slow down or turn off a core to save (battery) power. For x86 processors, the Linux idle-process code issues `mwait` instructions to suggest sleep states to the hardware. Deep sleep states such as Intel C6 involve turning off a CPU core and its caches (first doing any necessary writebacks). When a subsequent interrupt arrives at that core, the hardware and microcode first crank up the CPU core's clock and voltage, write good parity/ECC bits in the cache(s), and eventually execute the first instruction of the interrupt handler. Coming out of C6 deep sleep in an Intel i3-7100 takes about 30 microseconds, delaying interrupt handling by that amount.

You might not think that this matters much until you observe the dynamics of multiple communicating threads sending interprocessor interrupts to each other just as the receiving core has gone to sleep, and when that one responds, the reply goes back to a core that in turn has just gone to sleep. Rinse and repeat.

Figure 3 shows just such a sequence, at the beginning of launching the group of seven threads in the program in the previous section. Note that Figures 1–6 show everything happening on every CPU core every nanosecond—all idle time and kernel and user execution time for all programs, with nothing missing. For

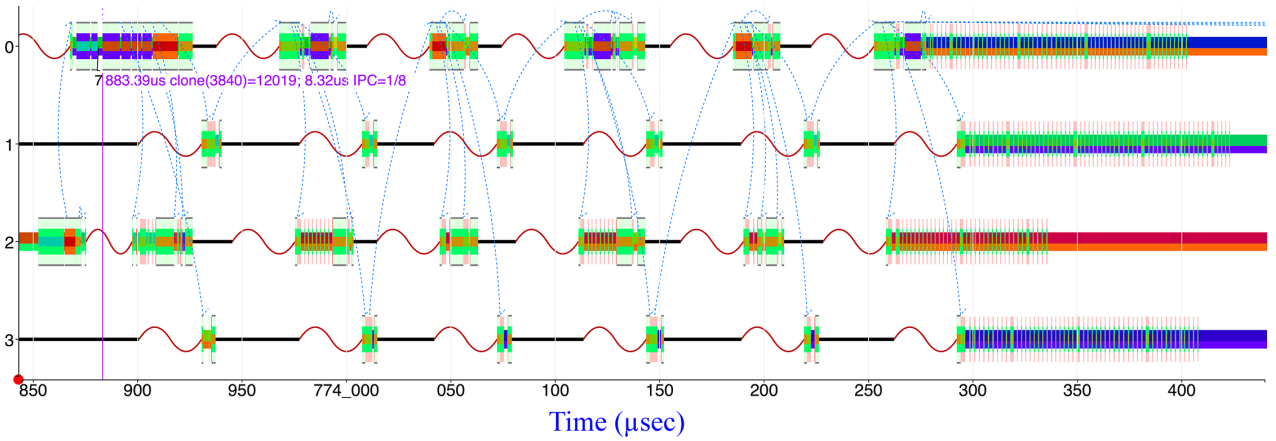

Figure 3: Premature sleep in the `intel_idle.c` Linux kernel code causes startup delays for seven spawned threads. Thin black lines are the idle process, and sine waves are the time it takes a chip core in deep sleep to wake up again. Heavier lines on the right are compute-bound execution of four of the seven threads on the four CPU cores.

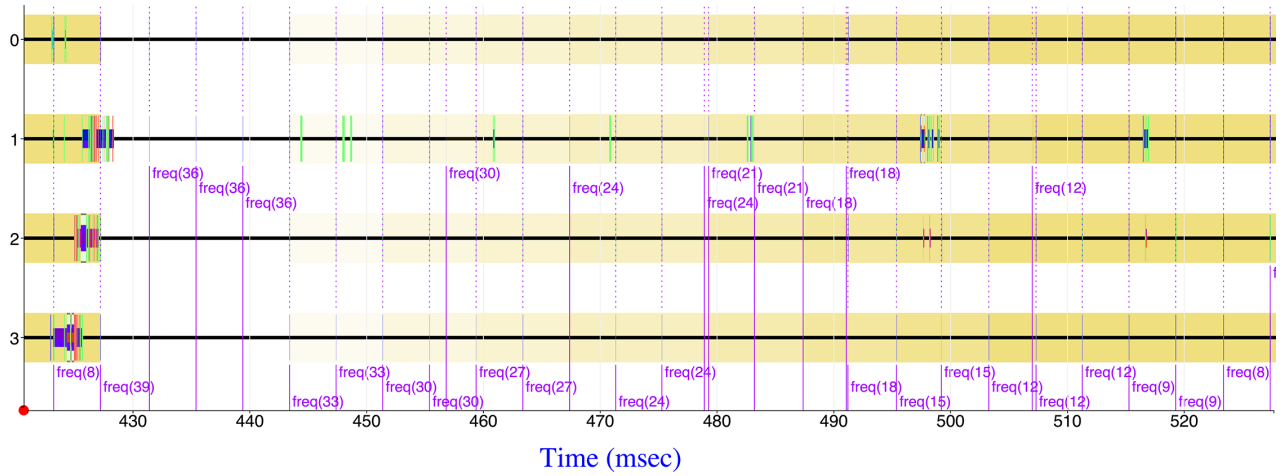

Figure 4: Non-idle execution on three CPUs at the left triggers a jump in all four CPU clock frequencies from slowest 800 MHz to fastest 3.9 GHz, which then step back to slow (frequency in units of 100 MHz indicated by the shading surrounding the lines for each CPU).

example, Figure 1 also has threads with names like `systemd-journal`, `cron`, `avahi-daemon`, `sshd`, and `DeadPoolForage`. None of these take any appreciable CPU time, so I cropped out most of them except the three cron jobs that run near time 1.8 seconds and take up a little vertical space between the group of two threads and the group of three threads in that figure.

The thin black lines in Figure 3 are the idle process executing, while the tall gray/colored lines are kernel-mode execution, and the half-height gray/colored lines are user-mode execution. The sine waves are the time coming out of C6 sleep (the time spent in deep sleep is short here, but is often several milliseconds). The dotted arcs show one process waking up another.

The idle threads do an `mwait` instruction to sleep after spinning for only 400–900 nsec, which is much too soon. In the diagram, the first four of seven `clone()`calls are on CPU 0 at the upper left, and the spawned threads start executing on CPUs 3, 2, 2, and 1, respectively, just after and just below. Each child thread blocks almost immediately inside `page_fault`, waiting for the parent to finish setting up shared pages. Full execution of four threads begins only on the right side of the diagram. The bouncing back and forth between parent and child keeps encountering ~50 µs delays because the CPU cores prematurely go into deep sleep.

There are two problems here: (1) 30 µs is a long time to be recovering from a siesta, ruining any expectations of fast interrupt response, for example, and (2) violation of the Half-Optimal Principle [6]:

> `If it takes time T to come out of a state waiting for some event at unknown time E in future, spin at least time T before going into that state. This is at least half-optimal in all cases, even though you don't know the future.`

In this case, the half-optimal strategy is to spin for 30 µs instead of 0.4–0.8 µs before dropping into a C6 sleep state that takes

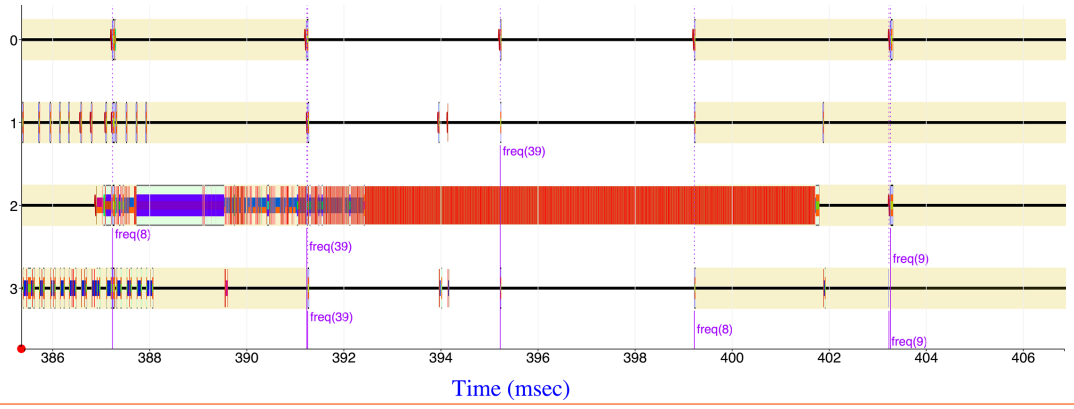

Figure 5: Non-idle execution at the left triggers a jump in CPU clock frequencies from slowest to fastest, which prematurely jump back to slow while CPU 2 is still 100% busy.

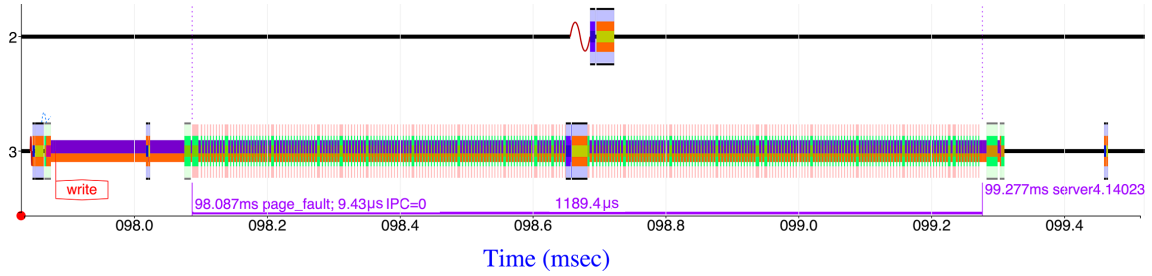

Figure 6: The page faults immediately after allocating memory take over 100x more time than the allocation itself.

30 µs to exit. Doing so would completely avoid sleeping in the trace above and would speed up the initial song-and-dance by over 3x. Observing these actual dynamics can lead to better algorithms.

Fluctuating Frequency: Mismatched to Goal

Our next study looks at another power-saving technique—varying the CPU clock frequency for each core. The goal is to use slow clocks when not much execution is happening and to use fast clocks when doing real computing. The measured Intel i3-7100 chip core clocks vary between 800 MHz and 3.9 GHz. For this processor, Linux allows the chip hardware to dynamically vary the clock frequency—"HWP enabled" in the Linux kernel Intel x86 jargon. Once enabled, no operating system code runs to vary the frequency or even to deliver an event when the frequency changes. However, a machine-specific register can be read to get some hint of the likely upcoming frequency. I added code to read that register at every timer interrupt and add it to the raw KUtrace.

For a computing load to observe, I ran a command to find some files and look for a regular expression in them:

```
find ~/linux-4.19.19/ -name "*.h" |xargs grep rdmsr
```

and then traced that for 20 seconds. This runs three programs, `find`, `xargs`, and `grep`. The first two mostly wait for disk while reading directories, and the last is mostly CPU-bound scanning a file. I picked this combination because I expected low CPU clock rates while waiting for disk and higher ones while scanning files.

Figure 4 shows an execution timeline on four CPU cores running mostly the `bash`, `find`, and `xargs` programs but with a little bit of other processes such as `jbd2`, `ssh`, and `chrome`. The gray overlay (yellow in the online version) shows CPU clock speeds: dark gray for slow clocks and lighter and lighter for faster clocks. The `freq` numbers are multiples of 100 MHz. Based on the non-idle program execution at the far left on CPUs 1, 2, and 3, the chip switches from 800 MHz to 3.9 GHz on all four CPU clocks, then slowly, over about 100 ms, drops the frequency back to 800 MHz. These are the true clock dynamics and match what one would expect from reading the (sparse) documentation. Note, however, that the execution bursts on CPU 1 in the right half of the diagram do not raise the clock frequency.

In contrast to the intended behavior, Figure 5 shows a region of the same trace eight seconds later. This time the clock frequency jumps up from 800 MHz to 3.9 GHz as expected, but 8 ms later it jumps back to 800 and then 900 MHz, even though CPU 2 is still quite busy running grep.

This dynamic is mismatched to the performance goal of the power-management design. Observing these actual dynamics can lead to better algorithms.

Cost of Malloc: Not There but Soon Thereafter

Our final study looks at memory allocation. In a client-server environment with the client sending 1,000,000-byte database write messages to the server, the server trace reveals a user-mode allocation of 1,000,000 bytes for receiving the message, followed by 245 page faults (`ceil` of 1,000,000/4096), the repeating blips on CPU 3 in Figure 6. You can see similar page-fault bursts in the completely different program at the far right of Figure 3. The big blips near time 98.7 ms are timer interrupts. You can directly see in the ~30 µs skew in delivering timer interrupts on sleeping CPU 2 and on busy CPU 3.

The allocation takes a few microseconds in the underlying system call just before the page faults, but the page faults themselves take over 1100 microseconds. The (good) Linux design for extending heap allocation simply creates 245 read-only page table entries pointing to the kernel's single all-zero page. As the user-mode program moves data into this buffer, each memory write to a new page takes a page fault, at which time the page-fault handler allocates a real memory page, does a copy-on-write (CoW) to zero it to prevent data security leaks between programs, sets the page table entry to read-write, and returns to redo the memory write. This goes on for 245 pages, taking much, much longer than the allocation time that is visible in many profiling tools. The dominant page-fault time is *invisible* to normal observation tools.

The copy-on-write path itself is inefficient in several ways. First, it could do groups of 4–16 pages at once, saving some page-fault entry/exit overhead without spending excess time in the fault routine and without allocating too many real pages that might never be used. Second, the kernel code does not special-case the Linux `ZERO_PAGE` as source to avoid reading it, by something like:

```
if (src == ZERO_PAGE)
        memset(dst, 0, 4096);
else
        memcpy(dst, src, 4096);
```

Doing so would avoid reading an extra 4 KB of zeros into the L1 cache each time and would avoid half of the memory accesses. It would also speed up the instructions per cycle (IPC) of the CoW inner loop.

A `malloc` call that reuses previously allocated heap space does not have the behavior seen here, but one that extends the heap does. Some programs aggressively extend and then shrink the heap repeatedly, wasting time not only in `malloc`/`free` but also in page faults. Allocating a buffer once and then explicitly reusing it in user code can be faster, for example. Observing these actual dynamics can lead to better algorithms.

Conclusion

Careful observation of Linux dynamic behavior reveals surprising anomalies in its schedulers, its use of modern chip power-saving states, and its memory allocation overhead. Such observation can lead to better understanding of how the actual behavior differs from the pictures in our heads. This understanding can in turn lead to better algorithms and better control of dynamic behavior.

As an industry, we have poor nondistorting tools for observing the true dynamic behavior of complex software, including the operating system itself. KUtrace is an example of a better tool. I encourage operating-system designers to provide such extremely-low-overhead, and hence nondistorting, tools in future releases.

References

[1] R. L. Sites, "Benchmarking 'Hello World'," *ACM Queue Magazine,* vol. 16, no. 5 (November 2018): https://queue.acm.org/detail.cfm?id=3291278.

[2] R. L. Sites, "KUTrace: Where Have All the Nanoseconds Gone?" Tracing Summit 2017 (11:00 a.m., slides and video): https://tracingsummit.org/wiki/TracingSummit2017.

[3] Open-source code for KUtrace: https://github.com/dicksites/KUtrace.

[4] Ftrace function tracer: https://www.kernel.org/doc/html/latest/trace/ftrace.html.

[5] Credit to Lars Nyland at Nvidia for first showing me this.

[6] I have used this principle for many years but only created the name while writing this article. A related Half-Useful Principle applies to disk transfers and other situations with startup delays: if you spend 10 ms doing a seek, then try to spend 10 ms transferring data (1 MB+ these days), so that at least half the time is useful work.