KUtrace Overview

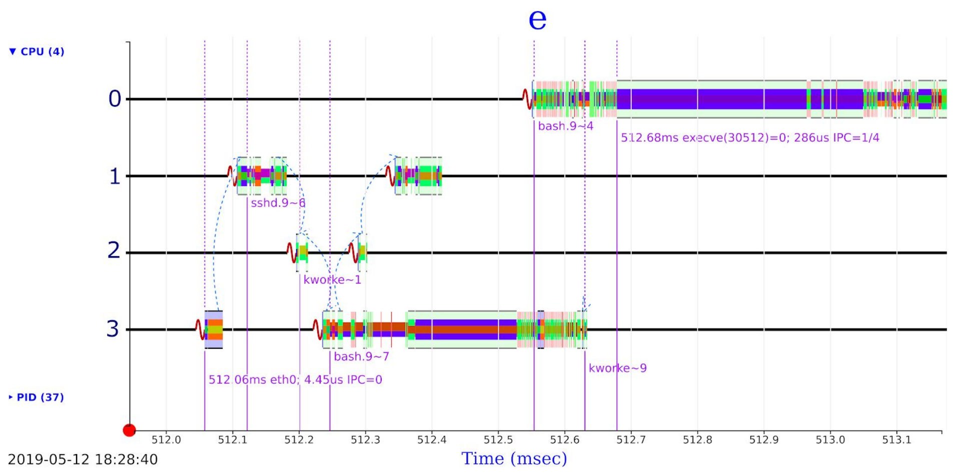
Richard L. Sites

2019.07.21

Summary

KUtrace is an extremely low-overhead Linux tracing facility for observing *all* the execution time on all cores of a multi-core processor, nothing missing, while running completely unmodified user programs written in any computer language. It has been used in live datacenters (x86 processors) and in real-time autonomous driving (ARM processors) to understand long-standing performance mysteries. The design goal of KUtrace is to reveal the root cause(s) of unexpected delayed responses in real-time transactions or database processing while having such low overhead that it does not distort the system under test. Its purpose is to understand the dynamics of large complex datacenter software *in situ* for live loads during the busiest hour of the day. But it can also be useful in more modest environments from desktops to cell phones or specialized real-time controllers. KUtrace is also useful in courses on operating system design or performance analysis, enabling students to observe complete real-world software dynamics instead of just simplified and idealized design concepts. See for example the ACM article on the complete execution and real dynamics of the simplest C program, Hello World [sites2018].

Also see the included sample output HTML file in kutrace/postprocess/HTML_output/e.html of this git download. It shows a program named mystery3nw_opt running on CPU 0, one of four CPUs. In the zoomed portion below, execution on four CPUs is shown. Horizontal black lines are the idle loop, half-height multicolored lines are user code execution, and full-height multicolored lines are kernel code execution. Starting with an Ethernet interrupt on CPU3 at the lower right, several make-runnable and interprocessor reschedule interrupts (arcs) run processes sshd, kworker1, bash7, bash4, and kworker9 to start up an executable in the 286 usec execve()syscall on CPU 0. The companion User Guide explains this display in more detail.



KUtrace is a one-trick pony. It only shows what every CPU core is doing every nanosecond in terms of user process ID and name, kernel code by syscall, interrupt, or fault number and name, or the idle loop. It does not identify individual procedures within user or kernel code. It is not useful for user-code debugging nor for kernel debugging -- it is just to observe where all the time went in complex real-time transactions and therefore to directly observe *why* a particular transaction was slow.

How it works

KUtrace works via small Linux kernel patches at the handful of points that implement transitions between kernel- and user-mode execution. Each patch takes about 40 CPU cycles. This git submission covers just x86 processors, tested on both AMD and Intel chips.

KUtrace timestamps and records every context switch and every *transition* between user-mode and kernel-mode execution -- system call, interrupt, fault -- into a kernel RAM buffer sized for 30-120 seconds of trace with optional wraparound (flight-recorder mode). Tracing 200,000 transitions per second per CPU core -- a rate observed during the busiest hour of live datacenter processing and twice the rate observed in autonomous driving -- is done with less than 1% overhead in CPU time and in RAM space. To achieve such low overhead, each simple trace entry is just four bytes and takes about 12.5 nsec to create. (In reality entries are eight bytes containing mostly-paired events, improving performance and information content over single four-byte entries.)

For system calls, the low 16 bits of the first argument and the low 16 bits of the return value are also recorded. This is achieved by packing a syscall/sysreturn pair into an eight-byte entry containing the call time, call event number, argument, delta-time to the return, and return value. The first argument often contains useful information such as file ID or byte count, while the return value can contain error codes, byte counts, thread wakeup counts and other useful information. Over 90% of all events are paired this way, including interrupt/return and fault/return. In addition to the kernel-user transitions, there are events to record names for each event number and each newly-encountered process ID. A few trace entries show the wakeup of each blocked thread and hence show what it was waiting for. Other trace entries show changes of CPU sleep state and hence causes of delayed interrupt delivery coming out of deep sleep. User code can optionally add named entries to delimit sections of programs.

Traces are postprocessed to turn raw transitions into *timespans* showing what each CPU core was executing every nanosecond: user-mode specific program; kernel-mode specific system call, interrupt, or fault routine; or the idle loop. Further postprocessing turns these time spans into a dynamic HTML/SVG timeline display that can be panned and zoomed and searched. The resulting display allows a deep understanding of the complex interactions and interference among multiple programs and multiple threads within a single program, the dynamics of a dozen or more layers of complex software, and also the interactions and interference between user-mode code and kernel code.

The traces optionally include instructions-per-cycle, IPC, for every microsecond-scale timespan, which among other things allows direct observation of slowdowns due to cross-core cache interference. This level of detail has never been available before.

What is new here: *all* CPU time traced, nothing missing; low overhead (30 times lower than ftrace); reasons for each thread wakeup and hence what it was waiting for; no sampling or aggregating/profiling to obscure the exact behavior of every single transaction; IPC for tiny timespans; root causes for slow transactions.

The source code

The top-level github dicksites / KUtrace directory has several sub-directories:
- **docs**
- **patches**, for Linux kernel 4.19.19
- **module**, loadable kernel module with actual trace-creation code
- **control**, user-mode program to turn tracing on/off and dump a raw binary trace file
- **postprocess**, programs to turn raw traces into HTML/SVG per-CPU-core timelines that you can search and pan/zoom down to the nanosecond

Because the absolute line numbers to be patched will vary from release-to-release, the kernel patches are presented here as pairs of files: the original 4.19.19 source and the patched 4.19.19 source. This should make it possible for anyone skilled in the art to apply these patches to the corresponding places in other releases with perhaps an hour or two of analysis and editing.

There are 19 pairs of foo.original and foo.patched files in subdirectories that match the Linux kernel directory organization.

To minimize trace-buffer size, KUtrace records only the low 20 bits of timestamps and reconstructs the higher bits during postprocessing. The facility depends critically on periodic timer interrupts every 10 msec or better to every CPU core. The config file **must** therefore avoid building with NO_HZ.

The companion **Installation Guide** describes how to build and patch a kernel, how to build and insert the loadable module, how to build the control program, and how to build the postprocessing programs. There is a sample trace file and corresponding sample output in the postprocess directory.

The companion **User Guide** describes how to create meaningful traces, how to postprocess them, and how to display them.

Origin

A previous tracing design was done by Richard Sites and Ross Biro at Google in 2006. This tool is mentioned under the name Ktrace in [bligh2007], and a later Google design is mentioned in [sharp2010a] and [sharp2010b]. While the idea originated when I worked at Google, the code here for a public Linux kernel version, plus the associated control and postprocessing and display programs, is completely designed and written from scratch, after I retired from Google in 2016 with no access to their code. This KUtrace design was first presented in [sites2017].

References

[bligh2007] M Bligh, M Desnoyers, R Schultz , Linux Kernel Debugging on Google-sized clusters, Linux Symposium, 2007 https://www.kernel.org/doc/mirror/ols2007v1.pdf#page=29

[sharp2010a] Benchmarks of kernel tracing options (ftrace, ktrace, lttng and perf), David Sharp, 2010
https://lkml.org/lkml/2010/10/28/261

[sharp2010b] Reduce tracing payload size, David Sharp <dhsharp@google.com>, 2010
https://lwn.net/Articles/418709/

[sites2017] Richard Sites, *KUTrace: Where have all the nanoseconds gone?*, Tracing Summit 2017, Prague, October 2017
https://tracingsummit.org/wiki/TracingSummit2017 (11:00am talk slides, HTML and video)

[sites2018] Richard L. Sites, *Benchmarking "Hello, World!"*, ACM Queue Magazine, November 2018,
https://queue.acm.org/detail.cfm?id=3291278