KUtrace Installation Guide

Richard L. Sites
2019.07.21

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

Disclaimer: I am not a kernel expert, so this guide includes just enough for someone who is familiar with or willing to undertake building kernels.

There are four pieces of code in the KUtrace installation, to be built in order. Start with the kernel itself and the KUtrace patches, then move on to the KUtrace loadable kernel module. Follow with the user-mode control program and then the user-mode postprocessing programs.

Building a patched kernel

==========

The current KUtrace patches are for the Linux 4.19 longterm maintenance release, specifically the 4.19.19 sources. The Linux kernel is updated constantly, with the current version being 5.2 at this writing and the most recent 4.19 version is 4.19.57. This makes kernel patching a moving target.

Because the absolute line numbers to be patched will vary from kernel release to release, the 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. Application to 4.19.x should be relatively straightforward, while application to 5.2 and later kernel version may involve some code rearrangement. It took me about a couple of hours to update the patches from 4.4 to 4.19, which involved substantial changes in the source for system calls (to combat spectre malware).

There are 19 pairs of foo.original and foo.patched files in kutrace/patches subdirectories that match the Linux kernel organization:

```
14 *.c
 1 *.h
 2 Make*
 2 *config*
```

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 other patches trace system call/return, interrupt/return, fault/return, scheduler execution, and context switches. Tracing scheduler execution catches a number of non-standard ways to exit kernel

code back to user code. It also allows tracing normal make-runnable events, so you can observe the delay between a process becoming runnable and it being allocated a CPU and starting execution. The idle-loop tracing includes events CPU power-saving changes (and hence execution slowdown). The interrupt tracing includes sending and receiving interprocessor interrupts, such as scheduler should-taps or TLB shootdowns.

Some second-order events are not currently traced. The postprocessing structure is there for patching backwards-compatibility 32-bit system calls, but they are not currently traced. Some rare interrupts such as NMI that deal with hardware debugging are not traced, nor are faults that terminate process execution (leaving just page fault traced at the moment). The goal is understanding the complex performance of stable running datacenter software, not the performance of software that crashes. Other tools are better for kernel- and user-mode debugging.

The directions below are for exactly the 4.19.19 x86-64 kernel, but then later cover what you need to do for other kernel versions. Follow along with whatever version you have access to. These notes assume that the KUtrace github files are already all in ~/kutrace/docs, ~/kutrace/patches, etc.

Directions for a brand-new 4.19.19 build, installing patches:
0) Download kernel 4.19.19 sources from https://www.kernel.org/ into a subdirectory of yours with a similar name. Mine is ~/linux-4.19.19 in these notes.
    If that version is not available, use 4.19.xx for the most recent xx
    If that version is not available, use 5.xx or whatever else is available
1) Build vanilla 4.19.19 from the original downloaded sources unchanged and reboot (details below). Use $ uname -a to confirm the version and build information. This is just a sanity check.
2) Copy the 19 KUtrace github downloaded *.original and *.patched files to their respective locations in the downloaded kernel source directory. This will not overwrite any downloaded files.
3) For all 19, diff foo and foo.original to cross-check that they are still identical. If not, see the following discussion
4) Copy foo.patched to foo for all 19 files, giving you three versions of files: foo, foo.original, and foo.patched.
5) Use **`make menuconfig`** to update the .config file to add:
  CONFIG_KUTRACE=y
 and to be sure that there are periodic timer interrupts to every CPU core:
  CONFIG_HZ_PERIODIC=y
  CONFIG_HZ_250=y
I used 250 timer interrupts per second, i.e. one every 4 msec. You could use anything from CONFIG_HZ_100 on up.
6) Build the now-patched 4.19.19 from the source directory and reboot. You have a patched kernel running. Use $ uname -a to confirm the version and build information. Go on to the control and module parts in kutrace/control.

To back out:
- Copy foo.original to foo for all 19 files.

- Build the now-un-patched 4.19.19 from the original downloaded source directory and reboot. You have the original kernel running again.

Build details
Kernel builds change over time, so you have to be a bit resilient as you do this. For starters, you need a compatible version of gcc or clang, which surely has already come with your Linux distribution. I am using Ubuntu 18.04 with gcc. Next, you need several build tools:

```
$ sudo apt-get install build-essential libncurses-dev bison flex libssl-dev libelf-dev
```

This is a good time to check what version of the kernel is currently running on your build machine:

```
$ uname -a
Linux dclab11 4.4.0-142-generic #168-Ubuntu SMP Wed Jan 16 21:00:45 UTC 2019 x86_64 x86_64
x86_64 GNU/Linux
```

I suggest starting a text file, say notes.txt, and pasting into it the commands you use and their output, so you have a record of what you did (and how you got off track if that happens). Put in the uname command and its output from above.

(1) Build vanilla 4.19.19
Put yourself in your ~/linux-4.19.19 or whatever directory. This is important, since the build commands depend on being in that directory:

```
cd ~/linux-4.19.19
```

You need a kernel .config file to drive the upcoming make of the entire kernel. Note the leading dot in the filename, which means that normal ls will not show this file; use ls -a to see it. Look to see what you have (note: no dot in the grep operand):

```
~/linux-4.19.19$ ls -a |grep config
```

You should see at least the Kconfig file, but likely will not see a .config file. If not, you can depend on your first run of make to walk you through creating one, or you can copy over from the current one on your build machine. To copy, do something like this:

```
~/linux-4.19.19$ ls -la /boot/ |grep config
-rw-r--r--  1 root root   217278 Jun  4 13:33 config-4.15.0-52-generic
-rw-r--r--  1 root root   217278 Jun 24 02:39 config-4.15.0-54-generic
```

Pick the latest version to copy.

```
~/linux-4.19.19$  sudo cp /boot/config-4.15.0-54-generic .config
```

The next step is an initial make, which likely will ask several dozen config (NEW) questions. Answer all these with <cr> to get the simple default.

```
~/linux-4.19.19$ make -j 4
```

The "4" above assumes you have a four-CPU build machine and specifies to use four parallel threads in the build, for improved speed. If your have N CPUs, use N or N+1 instead of 4. The first make will take half an hour or more, depending on the speed of your build machine and disk/SSD. It will build hundreds of drivers that you don't need. Be patient; we will speed things up later.

After the make finishes, copy the kernel and loadable modules (mostly drivers) to the right places via:

```
~/linux-4.19.19$ sudo make modules_install
~/linux-4.19.19$ sudo make install
```

You now have a bootable kernel version built from the so-far-unmodified downloaded sources and grub has been updated to boot from it. Run it:

```
~/linux-4.19.19$ sudo reboot
```

After rebooting, double-check what you have:

```
~/linux-4.19.19$ uname -a
Linux dclab-1 4.19.19 #29 SMP Wed Mar 13 14:34:23 PDT 2019 x86_64 x86_64 x86_64 GNU/Linux
```

If the boot fails, grub should fallback to your original build machine's kernel. You will need to sort out what happened to make the initial unmodified build fail. The rest of these notes assume that you have achieved a clean initial build.

(2) Copy the 19 KUtrace github files into the corresponding kernel source directories

```
cp ~/kutrace/patches/linux-4.19.19/arch/x86/mm/fault.c.* ~/linux-4.19.19/arch/x86/mm/
```

etc.

This will leave you with 19 sets of foo, foo.original and foo.patched files in the kernel source directories. If you started with exactly the 4.19.19 downloaded sources, foo and foo.original will exactly match.

(3) diff foo vs. foo.original for all 19
For identical pairs, you are done (many of these files can be completely unchanged across versions). The brand-new foo.patched files have empty corresponding foo.original files so you can diff mechanically.

If the github foo.original and your vanilla kernel source foo files differ, the vanilla source files have changed since the KUtrace github preparation. You will need to re-insert the patch lines in the new source files, using the github *.original and *.patched files for guidance. Unless the source code has been modified extensively, this will be largely mechanical and should take under an hour of your time. Establish new foo.original and foo.patched files.

Once you have a stable set of patches for a newer version of Linux, be sure your name and date are in the sources and then consider submitting them back to the kutrace github.

The rest of this note assumes you have created updated foo, foo.original, and foo.patched triples as necessary.

(4) Copy foo.patched to foo for all 19 files

Within each triple, foo will be identical to foo.patched.

(5) Update .config to allow KUtrace and to have periodic timer interrupts
~/linux-4.19.19$ cp .config .config.original

~/linux-4.19.19$ make menuconfig
setting
CONFIG_KUTRACE=y
CONFIG_HZ_PERIODIC=y

~/linux-4.19.19$ cp .config .config.patched

NOTEs: KUtrace depends critically on periodic timer interrupts to every CPU core. The config file must therefore avoid building with CONFIG_NO_HZ=y. If you don't add CONFIG_KUTRACE=y, none of the KUtrace code will be compiled in. The github kutrace/patches/.config* files are probably not exactly what you need, so the steps above don't use them.

(6) Build the now-patched 4.19.19 from the source directory and reboot.
~/linux-4.19.19$ sudo make modules_install
~/linux-4.19.19$ sudo make install
~/linux-4.19.19$ sudo reboot
Use **uname -a** to confirm the patched build. Congratulations! Go reward yourself. Then come back and build the loadable module and control software.

Loadable module
==========
The kernel patches are minimal calls to the runtime code that actually creates trace entries. The runtime code itself is not in the kernel but in a loadable module. The loadable module is built separately from the kernel but uses symbol definitions from the current kernel, so that when loaded it can be dynamically linked against the kernel. After each kernel rebuild, the module is rebuilt to match.

Go to the kutrace/module subdirectory and build the module
cd ~/kutrace/module
~/kutrace/module$ make

This just builds the module so takes only seconds. This build is successful only when done while running the patched kernel.

Install the module, allocating 20 MB of trace buffer
~/kutrace/module$ sudo insmod kutrace_mod.ko tracemb=20

If this complains, the kutrace_mod.ko file does not match the running kernel. Rebuild it to match.

If you want to remove the module, do:

```
~/kutrace/module$ sudo rmmod kutrace_mod.ko
```

You can run forever with the patched kernel and the module loaded but tracing off. The only overhead is the trace buffer space, plus unmeasurably-small CPU time for testing that the tracing Boolean is off. No other tracing code is executed.

User-mode control program

==========

Tracing is normally off, but can be started and stopped from a user-mode control program, with the resulting binary trace written out to a disk file. The supplied kutrace_control.cc program does this. It is built on a small user-mode library that can be accessed by other programs, making it easy to create alternate control structures.

Go to the kutrace/control subdirectory and build the control program.

```
cd ~/kutrace/control
~/kutrace/control$ make
```

This just builds the control program so takes only seconds.

Now the moment of truth. Run the control program, turn tracing on (go), and turn it off (stop). If the kernel doesn't crash, you have a successful KUtrace installation!

```
~/kutrace/control$ ./kutrace_control
Entering kutrace_control
TestModule DoControl = 0000000000000003
TestModule KUtrace module/code is version 3.
control> go
control> stop
DoOff DoControl = 0000000000000000
DoFlush
TestModule DoControl = 0000000000000003
TestModule KUtrace module/code is version 3.
DoFlush DoControl returned
wordcount = 32768
blockcount = 4
  ku_20190709_121346_dclab-1_22244.trace written (0.2MB)
~/kutrace/control$
```

The ku_*.trace file written is your very first trace. Keep it around as input to the postprocessing tools.

If it does crash, there are very limited things that can be wrong. Go back and copy all the foo.original to foo, rebuild the vanilla kernel and reboot, then run kutrace_control again to confirm that it says there is no module:

```
~/kutrace/control$ ./kutrace_control
Entering kutrace_control
TestModule DoControl = ffffffffffffffff
KUtrace module/code not loaded
```

```
~/kutrace/control$
```

Rebuild and run insmod again to confirm that you have no tracing kernel.

Now do binary search, incrementally adding KUtrace patches starting with the syscall patch, which implements the control call for starting and stopping tracing. Rebuild, reboot, rebuild the module, run kutrace_control and do go/stop. Repeat until you find what went off track.

The rest of this note assumes you have created a successful small trace.

Postprocessing

The postprocessing tools take a raw binary trace file and produce an HTML file showing timelines of all the CPUs. You can pan and zoom this file in your browser.

Go to the kutrace/postprocess subdirectory and build the programs

```
cd ~/kutrace/postprocess
~/kutrace/postprocess$ make
```

This builds the five postprocessing programs. Now run the postprocessing sequence against your first trace file above:

```
./postprocess ku_20190709_121346_dclab-1_22244.trace First_KUtrace_output JSON
```

You should now have
JSON_output/First_KUtrace_output.json
HTML_output/First_KUtrace_output.html

The json file has named and numbered timespans from the raw binary events and the html file has these packaged along with a browser-based user interface.

The postprocessing directory also has sample files:
test.trace is a small raw binary trace file,
JSON_output/e.json is the corresponding timespan file, and
HTML_output/e.html is the corresponding displayable form.
The HTML file shows three mostly-idle CPUs and one running a user program. A few named markers (pseud, initr, zero, read) were added by hand to this program, delimiting major portions.

HTML display

==========

Open **~/kutrace/postprocess/HTML_output/e.html** in your browser (Chrome preferred). It shows a program named mystery3nw_opt running on CPU 0, one of four CPUs. Click the top-center User button to see the names and first occurrences of all the 30 different processes in this particular trace. Horizontal black lines are the idle loop, half-height multicolored lines are user code execution,

and full-height multicolored lines are kernel code execution. Between time 40.6 and 41.3 there is a ~700 msec gap while the program waits for a disk syncfs system call. Use your mouse to pan via click-drag and zoom via the mouse wheel. As you zoom in, more detail will emerge, all the way down to 10 nsec resolution.

After zooming out, search in the top-middle box for something like "timer" of "fault" to find some interrupts or page faults. After zooming in somewhere, shift-click on any item to see what it is. Click on the red dot to reset the display. The companion User Guide explains the user interface controls in more detail.

Now open your own HTML_output/First_KUtrace_output.html file from above in a browser. You have a usable trace and its display. Enjoy!