Hello World Walkthrough
Dick Sites 2022.08.14

This note is a quick walkthrough of the KUtrace output file `hello_world_demo.html` included in `https://github.com/dicksites/KUtrace`, highlighting the display features. KUtrace records all the transitions between kernel- and user-mode execution on all the CPUs of a computer and these are then postprocessed into a dynamic HTML file.

Open the HTML file and follow along. The opening view shows timelines for four CPU cores across 3.5 seconds, with the main program of `hello_world` marked on CPU 2. Most of the display shows the idle job as a black line intermixed with timer interrupts. Depending on your display width, most of the space will be empty (white) or a high density of timer interrupts (blue). We will use times typed into the search box at the center top to move to different portions of the display. You can at any time use the mouse to click-drag to move the time sideways and the mouse wheel to zoom in and out.

The traced machine's Linux version and build date are shown at the upper right, along with the CPU model name. The start date and time time for the trace itself are shown at the lower left. X-axis times are relative to this.

Type **`50.08-`** into the search box (if you copy-paste, remove any spaces). The display pans and zooms to show about 50 timer interrupts per CPU, some scattered detail, and red and yellow overlays indicating power-saving differences in CPU frequency: red for slow, yellow medium, and very light green for full speed. This overlay comes up first so you don't waste time tracking down slow software performance that is simply caused by power-saving slow CPU clocks. We don't need it here, so click twice on the **`Freq`** button at the top to get rid of the overlay.

Type **`50.084+`** into the search box and hit **<cr>** to remember this string. The display now shows four timer interrupts on the four CPUs at the left (and four more at the far right), multiple programs running on the CPUs, wakeup arcs between programs, and much more detail. Shift-click on the large purple/blue timespan on CPU 2 to label it (release shift first to keep the label visible):

```
        83.669ms execve(10960)=0; 435us ipc=1/4
```

`83.669ms`	start time
`execve(10960)=0`	system call with 10960 low 16 bits of 1st argument, and 0 return value
`435us`	duration of the timespan
`ipc=1/4`	instructions per cycle, IPC (one per four cycles)

Now shift-**right**-click on this timespan to display all the `evecve` spans on the screen. At the upper right you will see that 4 match, totaling 473.46us, the shortest 7.8us and the longest 435.1us. These four spans are actually a single system call that has three embedded page faults. Shift-left-click to go back to the normal display. System calls have a green outer edge with two arbitrary colors based on the call number in the middle. Faults have pink edges and interrupts have blue edges. This system call is transferring from the bash command interpreter to the `hello_world` program.

Click the **`User`** button at the top to label the first instance of each different user-mode program or thread. There are four onscreen: the ssh daemon `sshd.4101`, a kernel worker thread `kworker~4`, command interpreter `bash.4105`, and another `bash.8292`. Shift-click on the large red/green timespan on CPU 1 to see where bash clones the second copy.

Search for **`eth0`** to see Ethernet interrupt handling (blue edges). Shift-click on the leftmost one on CPU 3 and drag to near the red `hello` markers on CPU 2 to see that it takes about 2.15 milliseconds of execution across four programs to get to the main routine of hello_world. **If you zoom in and work your way through slowly, you can identify every nanosecond of this latency.**

Briefly, the execution flow is from an Ethernet interrupt on CPU 3 running sshd which wakes up kworker on CPU 2 which wakes up bash on CPU 1 (passing in the command line "./hello") which echoes the command via kworker and sshd and also clones itself. The clone does the execve to load hello_world, which runs on CPU 2 and then exits. Shift-click on the large green/red timespan at about time 085.0 to see it is an exit_group system call. Before it exits, hello_world writes 12 bytes "hello world<cr>", waking up kworker on CPU 3 and then sshd on CPU 0, and after it exits wakes up the first bash on CPU 1. At the far right, bash sends out the next command-line prompt via the kworker/sshd dance.

Search for **`rx|tx`** to see the network traffic (the search box is a regular expression). At the left there is an incoming (rx) packet shown as a small falling diagonal line above CPU 0 and the corresponding interrupt hard and soft (BH-bottom half) handling on CPU 3. The command-line echo next sends a packet (tx) near time 083.0 with a small rising diagonal line above CPU 0. At the end of hello_world, there is an outbound tx packet "hello world<cr>", its ACK (no diagonal line), and the outbound tx prompt packet.

Click the **`IPC`** button at the top to see the IPC values for all non-idle timespans. The triangles are little speedometers, with legend on the right. Search for **`50.0848+`** to zoom in on the main part of hello_world and hit <cr>. Then click the **`IPC`** button once more to display just kernel-mode IPC values. The pink-edge timespans are (minor) page faults. Notice how their IPC gets a little higher as their cache footprint improves.

Click on the large brown/blue timespan at the end of the main program to see that it is a write to file ID 1 (stdout) of 12 bytes.

Search for **`50.0850+`** or just click-drag the display to the left about 200 usec to get the sshd CPU 0 execution onscreen and then search for **`write`** to label the two write() system calls, one in hello and one in sshd. Shift-click on the sshd one to see that it it is a write to a network socket, file ID 3, of 52 bytes -- the original 12 bytes plus 20 bytes of IP header and 20 bytes of TCP header.

Finally, search for **`50.084862+`** to zoom way in on a couple of brk() system calls. The diagonal white lines at the transitions indicate the approximate overhead of KUtrace itself, so you can see if it is large enough to be distorting. The brk() on the left takes 460ns, including the approximate KUtrace overhead of 50ns.

Click the red dot where the axes cross to get back out to the original display and then click the back arrow at the far lower left to return exactly one level. If you reload this file later on the same computer and the back arrow is blue instead of gray, you can click it to return to where you left off.

Pan and zoom around to explore. For example, the Chrome browser runs briefly at time 50.613.

See `https://github.com/dicksites/KUtrace/docs/kutrace_user_guide.pdf` for more detail on the display options.