# KUtrace User Guide

Richard L. Sites
2019.07.21

This **User Guide** documents the kutrace_control program, the kutrace_lib library you can optionally use to add markers or other items to your own program, and the five postprocessing programs rawtoevent, eventtospan, spantospan, spantotrim, and makeself. It also documents the intermediate json files and the user interface in the final HTML files.

I assume you have already used the Installation Guide to install KUtrace and you have already looked a little at the sample trace output. Figure 1 shows the pieces of KUtrace and how they interact. We will discuss the control and postprocessing pieces and use of the resulting HTML display.

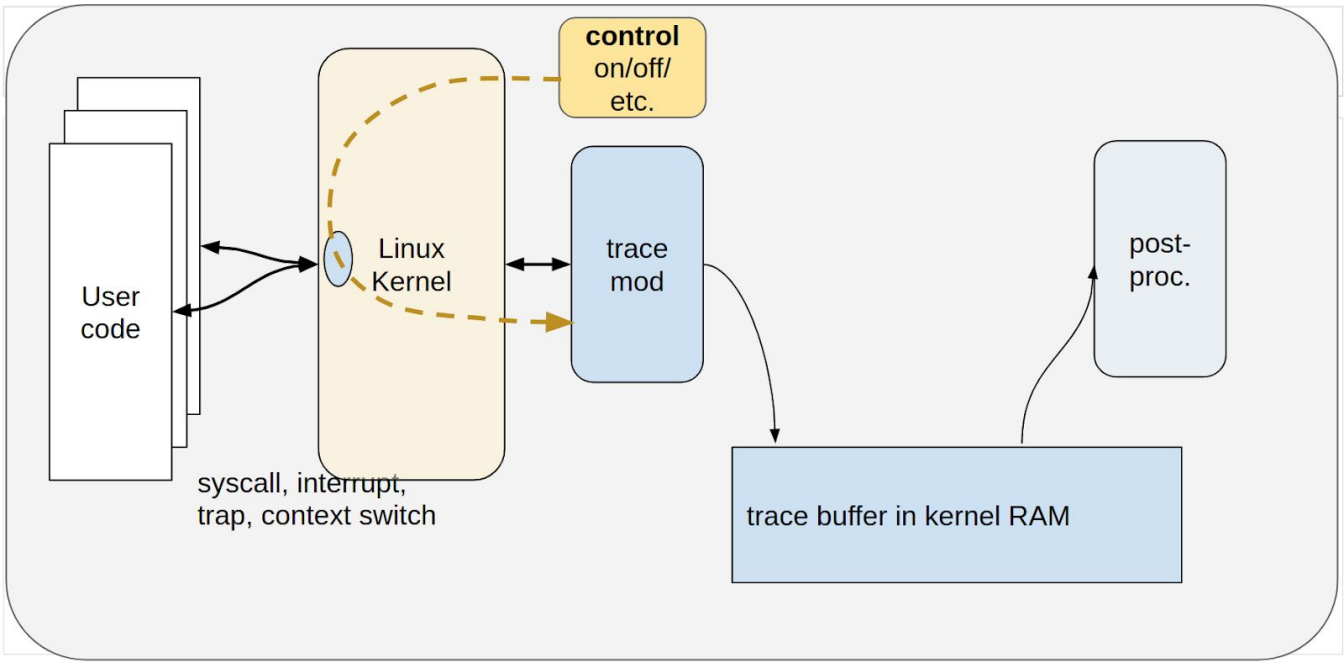


Figure 1. KUtrace overall flow.

## kutrace_control

The user-mode kutrace_control command-line program has a simple terminal interface that types a prompt and reads one-line commands until the user enters the quit command. It implements several commands, but the simplest are

**> go**
**> stop**

The go command resets the tracing buffer to empty and starts tracing. Variants **gowrap**, **goipc**, and **goipcwrap** set a couple of flag bits to enable flight-recorder wraparound tracing and/or tracking instructions per cycle (IPC) over each traced interval.

The stop command stops tracing (if it hasn't already been stopped) and writes the resulting raw trace to a file named with the current date and time and the process ID of the kutrace_control program, something like
  kutrace_20180606_121314_dclab-1_3456.trace

This raw trace file is self-contained, with human-readable names as embedded trace entries. Later sections describe the postprocessing of raw trace files into JSON and HTML files.

Some secondary commands are available to specify finer-grained control:
**> dump**
writes a raw trace file to disk/ssd. Must be preceded by off and then flush.

**> flush**
writes NOP trace entries to fill up each CPU's current partially-filled 64KB trace block. Must be preceded by off.

**> init**
inserts into the trace the names of all system calls, interrupts, and faults. It also inserts the current time and current process ID and name.  It should be called once at the beginning of a trace, just after reset.

**> off**
turns tracing off.

**> on**
turns tracing on. The first time, it must be preceded by at least reset.

**> quit**
turns off tracing and exits the calling program without dumping the trace buffer.

**> reset**
initializes the trace buffer to empty.

**> stat** or just <CR>
returns the current in-use size of the trace buffer. It may be called repeatedly during active tracing to observe progress in filling up the trace buffer.

**> test**
tests whether the kernel implements KUtrace at all.

The go command actually does the sequence
  reset, init, on
and the stop command does the sequence
  off, flush, dump.

In addition to the prompt/command interface, running kutrace_control with a single command-line argument of 0 or 1 turns tracing off or on respectively and exits immediately. This is possibly useful in scripts, once the initial tracing setup is done via the prompt/command interface.

## kutrace_lib

The user-mode kutrace_lib library has routines named DoDump ... DoTest that directly implement the above commands, plus several more low-level routines. All of these are declared inside the kutrace:: namespace.

These are the routines that directly correspond to the prompt/commands above:
void DoDump(const char* fname);
void DoFlush();
void DoInit(const char* process_name);
bool DoOff(); Returns false if module is not loaded
bool DoOn(); Returns false if module is not loaded
void DoQuit();
void DoReset(u64 control_flags);
void DoStat(u64 control_flags);
bool DoTest(); Returns true if module is loaded and tracing is on, else false

The above routines use several low-level routines to do their work. All these are also exposed for direct calls from user-mode programs.

**`void addevent(u64 eventnum, u64 arg);`**
Inserts an arbitrary one-word trace entry. Redundant; Use DoEvent

**`const char* Base40ToChar(u64 base40, char* str);`**
Converts a uint32 value that contains six packed base-40 characters into a character string. The base-40 character set is a-z, 0-9, and -./ (minus, period, slash).

**`u64 CharToBase40(const char* str);`**
Converts a character string of up to six characters into a packed uint32 value. The mark_a/b/c routines use this. Letters A-Z are mapped to a-z.

**`u64 DoControl(u64 command, u64 arg);`**
Directly calls the loadable module implementation via our tracing syscall with a command and argument. All of the other routines use DoControl to call the loadable module. Returns ~0L, i.e. all-ones, if no module is loaded, else returns the command's return value.

**`void DoEvent(u64 eventnum, u64 arg);`**
Inserts one trace entry.

**`void DoMark(u64 n, u64 arg);`**
Redundant; Use **`mark_a/b/c/d`** below.

**`void EmitNames(const NumNamePair* ipair, u64 n);`**
Adds a pre-built list of names to the trace. DoReset uses it to add syscall, interrupt, and fault names to the trace. The default syscall names are from kutrace/control/kutrace_control_names.h, which is included in the library.

**`int64 GetUsec();`**
Returns the current gettimeofday() value of time since the January 1, 1970 Unix epoch, packed into a single uint64 microsecond value.

**`void go(const char* process_name);`**
**`void goipc(const char* process_name);`**
**`void goipcwrap(const char* process_name);`**
**`void gowrap(const char* process_name);`**
These all do reset/init/on, passing different option flags.

**`const char* MakeTraceFileName(const char* name, char* str);`**
Constructs the file name used by DoDump, with date, time, computer name, and PID.

**`void mark_a(const char* label); mark_b, and mark_c`**
Insert those respective mark entries into the trace. Each takes a character string label of up to six base-40 characters. They differ only in how the label is displayed.

**`void mark_d(u64 n);`**
Inserts a mark_d entry into the trace. It takes a uint32 numeric label for the entry.

**`void msleep(int msec);`**
Sleeps for the specified number of milliseconds. DoOff uses it to give other CPU cores time to quiesce any in-progress trace-entry creation.

**`int64 readtime();`**
Returns the current value of the running time counter.

**`void stop(const char* fname);`**
Does off/flush/dump.

**`bool test();`**
Returns true if tracing is on.

**`bool TestModule();`**
Returns false if the module is not loaded or is an old version.

**`u64 VersionModule();`**
Returns the module version number if loaded, else ~0L. The current version number is 3.

Any user code can link in the library and directly call these routines, so alternate control programs or self-tracing programs are easy to build.

## Postprocessing

Once a raw trace file is created, it can be postprocessed into an HTML file that can be displayed and dynamically panned and zoomed and annotated. There are five programs involved, two of which are optional, as shown in Figure 2.

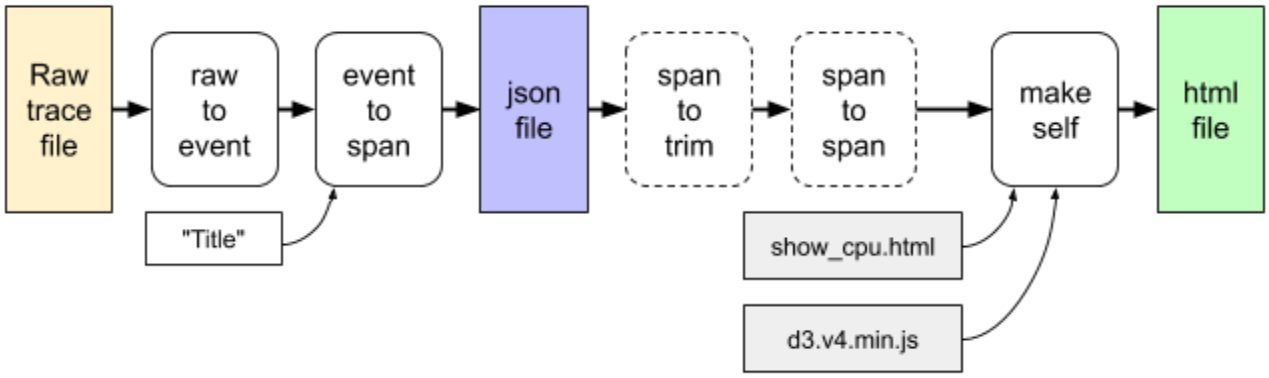


Figure 2. Overall postprocessing flow from raw binary trace to HTML/SVG display.

**`rawtoevent`** reads a raw binary trace and writes the contained events one per line as ASCII text. You can read this to see the individual events. Since the raw trace entries are recorded in per-CPU in blocks of 64KB, there will be 8K trace entries for CPU A, then 8K trace entries for CPU B, etc. A numeric system sort rearranges these into timestamp order with entries from different CPUs all interleaved.

**`eventtospan`** reads the sorted output of rawtoevent and turns call/return pairs into timespans. It tracks for each process ID and for each CPU the possible nesting of user/syscall/trap/interrupt routines (for preemptable kernels). Returns are paired up with stacked calls and popped to get back to the previous code that was running on a given CPU. Context switches save the old PID's stack and restore the new PID's stack. This allows a user-mode process on one CPU that is descheduled inside sys_futex to resume on a different CPU and almost immediately encounter a sysreturn from futex and still end up running the original user-mode process. This code also handles mismatched pairs, which always occur at the beginning of tracing and also occur due to various unusual kernel exit paths or patching oversights. Paired-up timespans are only emitted as the end of each span is encountered, so will be out of time sequence again. The output of eventtospan has some carefully-spaced JSON header lines added. One of those lines contains the supplied Title, which will be displayed at the top of the corresponding HTML. A second LC_ALL=C byte-wise system sort moves the JSON lines to the beginning and end, and puts the timespans into starting-timestamp order. Creating the final HTML/SVG depends on this order.

**`spantospan`** is one of two simple programs used to reduce the number of timespans. Displaying a million timespans is extremely slow inside a browser, and the screen resolution does not allow seeing many of them. spantospan reads a JSON file and takes a microsecond argument. It writes a JSON file with only timespans that are at least that many microseconds long. It accumulates missing time for

too-short spans and outputs it whenever the total exceeds the microsecond bound. The net effect is a coarse-grained time graph with fewer total timespans. Specifying 0 usec does no combining and keeps all the timespans.

**`spantotrim`** reads a JSON file and takes a pair of time arguments. It writes a JSON file with only timespans that overlap the time range [arg1..arg2]. As a special case, a single argument of 0 specifies to keep the entire range. Trimming is useful for producing a file with a modest number of timespans that cover a specific subset of the trace time.

Using spantospan with 10..100 usec followed by spantotrim 0 gives a quick overview picture of an entire trace. From there, it is usually easy to spot some "What is THAT?" sections. Reprocessing with spantospan 0 followed by spantotrim 30.344 30.346 picks out a specific full-resolution 2 msec window.

The last program, **`makeself,`** takes as input a JSON file and the filename of a template HTML file (`show_cpu_2019.html`). It produces a self-contained HTML file that references the external https://d3js.org/d3.v4.min.js Javascript library (so needs an internet connection to display properly).

## Postprocessing details (skip this on first reading)

A raw trace file consists of packed binary 8-byte trace entries. These entries largely record call-return pairs for system calls, interrupts, and faults, but they also include context switches, names of routines, and other markers. The `rawtoevent` program reads a trace file and produces a text list of the events. The `eventtospan` program reads this list and turns them into time spans, producing a json file. The optional `spantotrim` program reads a json file and produces a smaller one with just the spans in a specified start..stop time range. The optional `spantospan` program reads a json file and produces a smaller one with more granular time spans no shorter than a specified number of microseconds. The makeself program reads an html template, the d3 javascript library, and an input json file and then produces a self-contained html file with all the trace data and a dynamic UI. The output html file can reasonably support 100K to 1M time spans, but starts to run out of memory space in the browser around 2M time spans.

### The `rawtoevent` program

This program is a standard Unix filter, reading from stdin and writing to stdout. It reads each 64KB block in a trace file and if the file contains optional IPC (instructions per cycle) information it reads the additional 8KB of IPC per trace block.

```
Usage: rawtoevent [filename] [-v]
```

Normally `rawtoevent` reads from stdin and writes to stdout, but it optionally can take a filename parameter to read from. The -v parameter produces verbose output showing each 8-byte trace entry in hex, plus its timestamp, event number, call argument, delta time between call and return, and return value. This is just a debugging aid; with -v, the output is not suitable for reading by `eventtospan`.

The beginning of each trace file block has a full timestamp and corresponding gettimeofday() value in microseconds. The remaining trace file entries have 20-bit timestamps which wrap around every 15-30

milliseconds. It is the responsibility of `rawtoevent` to expand these short timestamps to full ones by prepending the high-order bits and incrementing this value at every wraparound. The very first trace file block also has two pairs of *<timestamp, gettimeofday>* values taken at the start of a trace and taken at the end. These are used by `rawtoevent` to map timestamp counts to seconds of wallclock time. These times are relative to the starting minute of the trace, so normally are in the range of 0.0 to about 150.0 seconds, depending on the trace size. The output values are given in integer increments of 10 nanoseconds, so there is an implied decimal point before the last eight digits of each timestamp.

Some traces continuously wrap around in the trace buffer for many minutes before some program or manual event stops the trace, which then contains the last few seconds before the stop. In this case, the start minute of the trace may be many minutes or a good fraction of an hour before the earliest retained trace entry. Since the earliest retained trace entry may not be known until the last block of the raw trace file is read, `rawtoevent` produces seconds of wallclock time that may be quite large. The next program, `eventtospan`, adjusts the start minute to make the earliest second of wallclock time be in the range [0.0 .. 60.0).

When tracing wraps around, the very first trace block is not overwritten; instead, the overwriting begins at the second trace block. This enables retention of the syscall/interrupt/fault names recorded in the first block and also the start/stop pairs of *<timestamp, gettimeofday>* values recorded there. The remaining entries in the very first trace block may describe events that are many minutes before the rest of the trace. To avoid a big gap in the final time spans, `rawtoevent` drops these orphan first-block trace entries.

The rawtoevent program is also responsible for extracting the names of every event and user-mode program, extracting the CPU number from the front of each trace block, extracting the current process ID at each instant, and extracting any identifier of the transaction (remote procedure call ID, rpcid) that each user-mode program is working on at each instant. It  records these in each output event. Some names appear physically in the raw trace file after their first mention in the trace file. This can occur when the first use and hence the name recording is on CPU A and a later use is on CPU B, but it turns out that B's traceblock was allocated earlier in the trace buffer than A's and hence is written to the trace file first. When this happens, some of the event output from `rawtoevent` will be missing names. But all the names are put into the event output in duplicate, at their original time and also at time -1. This lets them sort to the front and allows the next program, `eventtospan`, to add names to all entries.

The output of rawtoevent must be sorted to put everything into time order. This is normally done by piping the output through the system sort, specifying a numeric sort:

```
cat foo.trace |./rawtoevent |sort -n |...
```

This sorted output is then piped into the next program, `eventospan`.

The `rawtoevent` program writes a few summary lines to stderr as it finishes. These give the trace date and time, number of CPUs, number of events, a short breakdown by event type, and the time span covered from the base minute. This last information can be used to guide spantotrim below.

### The `eventtospan` program

This program is also a standard Unix filter, reading from stdin and writing to stdout. It reads the sorted events from `rawtoevent` and matches these kernel-user transitions into time spans. It is also responsible for producing the final start minute and matching seconds, as mentioned above.

```
Usage: eventtospan ["Title"] [-v]
```

The first parameter is a string used as the title in the resulting HTML diagram. The -v parameter produces verbose output showing each line of event input. This is just a debugging aid; with -v, the JSON output is not suitable for reading by later programs.

There are a few trace parameters passed along from `rawtoevent` in event file comments (beginning with #), in a rather ad hoc way. These include the original trace-start date and time, trace version number and flags, and wraparound-trace actual minimum/maximum times. Other than that, the input consists of name definitions and transition events.

Name definitions have the same timestamp, duration, and event fields as transition events, but then then have just an argument number and a name. The timestamp is unused except to position the names in the input file. The duration is completely unused. The event number specifies what kind of item is being named -- source file, PID, RPC method, trap, interrupt, lock, syscall, syscall32 -- and also the size of the name definition entry in 8-byte words, 1..8. The size includes the first word that contains the timestamp, duration etc., leaving room for up to 56 bytes of name in the remaining 0..7 words. The last word is zero-filled as needed. The `rawtoevent` program has already filled in all the names except some missing PID names, so `eventtospan` just remembers and fills in these.

In long traces, it is possible that the process PID numbers are reused or redefined via execv or suchlike. In that case, rewriting the names based on time will have the effect of tracking those changes.

The main work of turning kernel-user transitions into time spans is done in `ProcessEvent`. For each CPU, there is an existing time span that is being constructed, arbitrarily initialized to the idle process before the start of the trace. Call events stop the existing time span and start a time span for the subject of the call -- a specific system call, or interrupt, or fault. But return events need to stop the existing called time span and return to -- what? To reconstruct the what, `ProcessEvent` maintains a small stack of pending spans for each CPU. Each call pushes this stack and each return pops it. So if user-mode process 1234 is running and does a call to syswrite(), the matching return will start a new span of process 1234. Syscalls, faults, and interrupts can nest. In particular, interrupts can be delivered while in a syscall or in a fault handler. Including the user-mode program, the maximum stack depth can be four but no more.

With multiple CPUs, each one has its own reconstruction stack. In addition, processes can migrate from one CPU to another. So at each context switch event, the stack for the old PID is saved under that PID number and the stack for the new PID is restored. If a PID has no saved stack, a new one is created on the fly.

Events can be mismatched in various ways. At the beginning of a trace, there may be a return with no matching call. At other times, there may be a system call with no matching return because it exited in a non-standard way through the scheduler (which is treated as a dummy syscall routine). An interrupt routine that causes a context switch can exit directly to user mode instead of returning to an in-progress system call or fault handler, to be resumed at some later time. So at each push and pop of the reconstruction stack, there are little routines called that create dummy pushes and pops to keep things matched.

The `eventospan` program has a few hacks that are either unneeded or could be done more cleanly now. These include FixupEvent, Mwait/C-exit synthetic sine waves to show x86 time exiting from power-saving, and a fixup to insert dummy IPC values when none are supplied by an old version of `rawtoevent`.

The final output from eventospan is a json file that includes some header lines, some trailer lines, and a large number of time span lines. Since the time spans are created when the ending transition is encountered, but the eventual display depends on them being in order by start time, the output from `eventtospan` is sent again through the system sort routine. In order to make sure the json header and trailer lines appear in the proper places, the lines have carefully inserted initial spaces and punctuation to make them sort to the front or back respectively. Unfortunately, the default sort collation sequence might be to sort alphabetically ignoring spaces and punctuation. When this happens, the sorted json is ill-formed and nothing displays properly. So it is important to set the collation sequence to use bare byte values via

```
export LC_ALL=C
... | eventtospan "Title" | sort | ...
```

before using `eventtospan`. (It might be possible to use `LC_COLLATE=C`, but this fails if LC_ALL is set to something else.)

The sorted output of `eventospan` can be fed directly into `makeself`, but for large traces it can be useful to use the optional `spanto...` filters, described next.

The `eventtospan` program writes a few summary lines to stderr as it finishes. These give the trace date and time, number of spans, and a short breakdown by span type.

### The `spantotrim` program

This program is a standard Unix filter, reading from stdin and writing to stdout. It reads a json file and writes a smaller one that only contains a subset of the full time span of the incoming trace.

```
Usage: spantotrim start_sec [stop_sec]
Usage: spantotrim "string"
```

The start_sec parameter specifies a start time in seconds and fraction from the start-minute of the trace. If the stop_sec time is also given, `spantotrim` keeps just timespans that overlap the [start .. stop] interval.

If just start_sec is given and it is zero, `spantotrim` is a NOP, keeping everything. If it is a positive number of seconds and fraction, `spantotrim` keeps that many seconds from the beginning of the trace. This can be handy because it does not require copying over the start time from the `rawtoevent` summary. If start_sec is negative, it keeps that many seconds from the end of the trace.

An alternate form of spantotrim takes a non-numeric string as its only argument. In that case, it keeps only timespans that contain that string, which might be a PID number with punctuation or some kind of name. This form is of limited use; it differs from simple grep only in keeping the json header and trailer lines.

### The `spantospan` program

This program is a standard Unix filter, reading from stdin and writing to stdout. It reads a json file and writes a smaller one that contains more granular time resolution than the incoming trace.

```
Usage: spantospan resolution_usec
```

The resolution_usec parameter specifies the smallest timespan size in microseconds. If it is zero, `spantospan` is a NOP, keeping everything unchanged. Otherwise, for each CPU, spantospan accumulates smaller timespans while the total time deferred is less than resolution_usec. When that amount of time is reached or exceeded, the accumulated time is assigned to the most recent event and placed in the output file. This distorts the fine-grain timing of a trace, but places a tight bound on how many timespans are in the resulting output file. Such a file can be useful for getting a quick overview of the activity in a very large trace, followed by selective use of spantotrim to look at the more interesting portions.

Because `spantospan` produces output at the end not the beginning of each span, the resulting json must be sorted once again.

```
... | spantospan 10 | sort | ...
```

### The `makeself` program

This program is partly a Unix filter, reading from stdin and writing to stdout. It reads a json file and writes a self-contained html file that displays that json along with user interface controls for dynamically panning, zooming, and annotating the timespans. To do this, makeself reads a specified template HTML file.

```
Usage: ... | makeself template_filename
```

The template_filename parameter specifies an HTML file containing the user interface HTML and JavaScript. This file must have several stylized `selfcontained*` comments that delimit where to put the d3 library and where to put the json file from stdin. (With different templates, this program can be used to make self-contained disk or network traces.) There is nothing fancy here. Everything is read into large buffers, so an extremely large json file will fail. Use spantotrim or spantospan in that case.

Putting it all together, the usual flow is
**trace** ==> rawtoevent, sort -n, eventtospan, sort ==> **json** ==> makeself ==> **html**

This turns a raw trace into an HTML file that can be viewed, panned, zoomed, and searched in any browser.

## JSON output

The json output of eventtospan has some carefully-formatted initial and final metadata lines, designed to land in the proper place after sorting via LC_ALL=C. The rest of the file has one line per timespan, each an array of ten items:

```
  [ 35.50203293, 0.00000116, 3, 11652, 0, 77188, 0, 1, 0, "sshd.11652"],

35.50203293,     Timestamp in seconds
0.00000116,      Duration in seconds
3,               CPU number
11652,           Process ID, PID
0,               RPCID, if inserted by a user-mode RPC library
77188,           Event: 0-4095 raw events; 65536+ process ID number plus 64K
0,               Arg0 low 16 bits for system call events
1,               Return value low 16 bits for system call/return events
0,               IPC, quantized into 16 ranges, 0..15
"sshd.11652"     Name associated with this timespan: process, syscall, etc.
```

## HTML output

The HTML output of makeself is simply the json above turned into one very long string, plus an HTML and SVG wrapper that implements the user interface actions. The file refers to the external **`d3.v4.min.js`** library from the wonderful https://d3js.org/ website.

The self-contained HTML file comes up showing the entire trace range. You can always get back to this view by clicking the **red dot** at the lower left. Using a mouse with a **scroll wheel** zooms the time axis, centered on wherever the mouse cursor is. Mouse **click and drag** pans sideways. These give relatively fast way to pan and zoom around the trace diagram.

**Shift-click** on a timespan annotates it with the start time, name, and duration. Shift-click-unshift-unclick in that order leaves the annotation onscreen, so you can look at it for a while or shift-click other spans.

Unclicking first clears all the annotations. Process names have a period and the PID number appended. System calls have the argument low 16 bits and return value low 16 bits appended as sysfoo(xx)=yy.

**Shift-click-drag** annotates start/stop timespans and gives the total elapsed time at the bottom of the screen.

The shift-click actions snap to the front of the nearest in x-y timespan, so you can choose which starting and ending CPUs you care about and which particular start/end span.

There are several UI buttons and boxes at the top.

**File:** is unused in self-contained HTML files. It was originally for loading a JSON file from a server.

The **CB** button toggles colors for color-blind users. It simply rotates RGB values so they become GBR.

The **IPC** button toggles display of instructions-per-cycle information for each timespan, shown as speedometer-like arrows from 0 to 4 IPC, with the legend on the right. If there is no IPC information in the JSON, this button is grayed out.

The **Marks** button toggles the display of manually-inserted marker events. This is useful when the markers are so dense that they obscure the timespans. If there aren't any marker events in the trace, this button does nothing.

The **All** button adds short annotations for every timespan on the screen. It is most useful when you have zoomed in a lot. The short names are elided, indicated by "~", to 8 characters. Only the first instance of each user-mode process name is shown.

The **User** button adds short annotations for the first instance of every user-mode process name on screen. It is useful, for example, on the initial unzoomed display to see what processes exist.

The **Search:** box annotates any onscreen timespan whose name includes the case-sensitive search string. This is actually a Javascript regular expression match, so you can be a little fancy. Searches also show on the far right the number of matching timespans and the total time in those spans. The **[!]** button inverts the match, like grep -v.

The **usec:** boxes filter searches by minimum and maximum usec per matching timespan.

On the left, there are group names for the displayed timelines. By default, the CPU group is shown expanded and the PID group is collapsed. There may also be RPCID and Resource groups. These all display the same timespans, just sorted in different ways. You can use the mouse to pan and zoom vertically in the left area, and toggle the triangles on the group names to expand or contract each group.

At the bottom left, there are five numbered history buttons. Shift-click one of these to save the current pan/zoom and other display parameters. Later click it to restore those. The double-arrow button toggles between the last two such displays.

At the very bottom, the [more] button brings up more detailed usage text just off the bottom of the screen. This briefly details several more shift-click options.

If you want to capture the current on-screen diagram, I highly recommend the svg Crowbar 2 plugin for Chrome. It creates one or more SVG files; you want to download the **`SVG#0:.svgtop`** one to save it to disk. You may need to trim the SVG bounding box using a tool such as Inkscape (select all the SVG elements, then Edit > Resize Page to Selection).

That's it. Have fun sleuthing!

## Adding tracepoints to user-mode code

By including the kutrace_lib.h and .cc files, you can add marker events, contended lock events, and other events to a trace. You can also build programs that turn tracing on and off over specific pieces of code. Here is hello_world with two markers added:

```
#include "kutrace_lib.h"
int main (int argc, const char** argv) {
  kutrace::mark_a("hello");
  fprintf(stdout, "hello world\n");
  kutrace::mark_a("/hello");
  return 0;
}
```

## Bugs, omissions, and future work

As it stands, this software is a proof of concept. It is suitable for classroom use and for some industrial use, but in its current form has a number of known shortcomings.

First and foremost, I am a poor coder. My strengths are exploring new concepts and implementing them efficiently. My weakness is hard-to-read code. Feel free to improve this, but please don't slow down the implementation much -- if it gets even 4x slower, the whole idea starts to become useless.

**Security.** KUtrace by design covers all execution on a computer. If there are 30 programs running, it traces all of them. This can be acceptable for the owner of that computer in understanding the performance of code created by that owner. But it might not be acceptable in a shared-hardware environment for one user to be able to trace code from other users. But there is no possible way to observe interference between programs if only one program is traced. There is thus a tension between security and observation. Once the KUtrace module has been loaded, any user program can create a trace and dump it to disk. In many environments, this would be a large security hole. In addition, the trace-buffer extraction call is subject to Spectre-like spoofing that could effectively read anywhere in kernel memory.

**Patches.** I wish there were a better way of communicating the patches. I am not signing up to move them upstream and maintain them across Linux releases, but someone else might. While I have done an ARM implementation, I do not currently have the debugging environment to update that and release it. Only a few files in linux-4.19.19/arch/arm need to be patched to mirror those in arch/x86. The IPC calculation does not have a very flexible way to convert an underlying non-x86 fast timebase into cycles.

**Module.** It is possible to compile the module directly into the kernel rather than having it loadable. In such an environment, the buffer size would be fixed, or some additional code would be needed to vary its size. The module loaded/not, module version, and tracing/not interface could be rationalized. The current extraction of the trace buffer to a file does millions of syscalls to extract 8 bytes at a time. This is not really a performance issue because it is user-mode code and disk-bound, but it would be somewhat more efficient to move 4KB blocks from kernel to user space. The capturing of newly-spawned process names also needs to be more robust. Currently clone, execve and suchlike do not put the new program's name into the trace but they should. The name of each program that is running at the beginning of a new 64KB trace block is captured, so some of the missing process names show up that way.

**kutrace_control.** The pre-built list of syscall names and fault names works well since these change only slowly over years. But the association between raw interrupt numbers and names is likely to vary from boot to boot. A more robust method of capturing these names would be useful. For example, the sample trace e.html has no name for interrupt number 526, which actually came from disk sda1 (we know it is from a disk because its bottom half processing is BH:block). There might be a startup bug if tracing is turned on while running a program that is doing frequent mark_x() calls on another CPU -- I have seen once the first trace block with mark_x entries instead of the expected startup list of syscall and other fixed names. There should be no entries added to the trace until after all the startup names are in place and then tracing is turned on. There may be a race condition or a missing MB or suchlike. The traces would be slightly more robust if they included initial entries for the CPU name, CPU type, and nominal CPU clock speed, with corresponding downstream support to display these. This is more relevant in an industrial setting with thousands of processors of several different types.

**eventtospan.** The handling of mismatched call/return pairs is still a little less robust than I would like. This mostly shows up as rare mistakes for the return point being in user instead of kernel code when exiting the scheduler. The insertion of the coming-out-of-deep-sleep sine waves is fairly hacky, currently using a hardwired table of likely exit times. But maybe the current stuff is OK just to highlight where possible delays are. A robust form of this would either export the corresponding Linux table values from the processor-specific idle loop code, or would export the processor type and nominal clock so that eventtospan could select from corresponding tables. A robust form would also track hypertheaded CPUs enough to mark all of a physical-CPU set as of logical CPUs not-in-deep-sleep if any in the set is executing full-bore.

**makeself.** The d3.min.js library was originally unconditionally inserted into the output HTML from a local copy, but instead now a reference to d3js.org is used. The resulting file thus needs an internet connection to display properly. A more robust makeself could take an option to either insert a local copy

to be completely self-contained, or use the current web reference. The json file-serving could also be connected up again.

**Future work.** In many remote-procedure call environments there is a fundamental ambiguity when a request or response message arrives late across the network -- was the sending machine slow to get it onto the wire/fiber, was the receiving machine slow to deliver it off the fiber/wire to user code, or was the network hardware itself slow (perhaps due to congestion or retransmits)? It would be helpful to have a patch that records incoming or both outgoing and incoming RPCID numbers and timestamp as close to the NIC hardware as possible. That is enough to identify where the delay occurred. KUtrace is most useful with RPC libraries that insert currently-being-processed RPCID numbers into the trace, and with locking libraries that insert failed-to-acquire, freed-contended, and now-acquired trace entries specifying which specific contended lock is involved (preferably by source file name and line number). I have done nothing here with virtual machine hypervisors. It would be useful to extend this work into hypervisors.

## Conclusion

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 is valuable for observing the true dynamics of complex datacenter and database software and for finding root causes of slow real-time transactions. It is the only tool in the industry to give a 100% complete picture of CPU time with low enough overhead to use in live datacenters during the busiest hour of the day.

In contrast to the simplified pictures in our heads, KUtrace allows one to observe *everything* being executed, the true interactions and interference between threads, between unrelated programs, and between kernel and user code.

## Appendix: Walkthrough of sample output

In the postprocess directory, there is a sample trace file and corresponding sample output:

**`kutrace/postprocess/HTML_output/e.html`**

We will walk through a little of that together. Open the file in your browser. The timeline goes left to right from about 39 seconds to 44 seconds. The four CPUs go top to bottom, 0..3. The group of 37 process IDs is collapsed. There is one major program running, almost entirely on CPU0, shown in blue. Only timespans that are one pixel wide or larger are shown, so there initially is not much detail visible.

The mystery3 program writes 40MB of random bytes to disk, zeros a 40MB buffer, and then asynchronously reads the random 40MB into the pre-zeroed buffer, scanning the buffer to record the microsecond time at which each 4KB block arrives from disk. The purpose of the program is to observe the fine-grained timing of a big disk or SSD transfer, not just the average transfer rate reported by other tools. The program is structured as:

```
{
  kutrace::mark_c("pseud");
    <fill a buffer with 40MB of random bytes>
  kutrace::mark_c("/pseud");

  kutrace::mark_a("initr");
    <write the 40MB to disk, initializing for the read below>
  kutrace::mark_a("/initr");

  kutrace::mark_c("zero");
    <zero the buffer>
  kutrace::mark_c("/zero");

  syncfs(fd);    // Force the write data to disk,
                 // not just sitting in the in-RAM file-system buffers.

  kutrace::mark_a("read");
    <asynchronous read of the 40MB random bytes into the pre-zeroed buffer>
    <scan, recording time of zero ==> non-zero change per 4KB block>
  kutrace::mark_a("/read");
}
```

In this initial display, the initr and pseud markers overlap at center-left, the zero markers are not visible, and the read markers are clearly distinct at center-right. We will zoom in on these pieces to see the substructure.

Just to get calibrated a little, click the User button near top center to see the 30 different programs that are running during the traced five seconds. In the search box, type **page** and hit <cr> to see that there are 10502 page fault timespans on screen, totalling 9.1641 msec. Now use the mouse wheel to zoom in on the section around 40.5 seconds, and pan a little to show the interval 40.500 to 40.600 on screen. The base time will change from 18:28:00 to 18:28:40, and the X-axis labels will change from seconds to milliseconds, so you want 500..600 msec on screen. You can start to see timer interrupts every 4 msec, and some activity on other CPUs. If you go back to the search box, still reading **page**, and hit <cr> again, the top right display will now say about 10400 page faults and 9.05 msec. So now you know that almost all the page faults are in this little portion of the trace and are not scattered uniformly across the entire five seconds. Simple **top**-like page-fault counts won't tell you where they are coming from and how they are clustered the way KUtrace does.

Backspace over **page** to clear the search box. You can now see the pseud and /pseud markers, the /initr marker, and the /zero marker. The init and zero markers are also there but are suppressed because that are at the same pixel as one of the others. Zoom in a lot near time 40.540900 so that only the 100 usec from 870 to 970 usec are on screen. Now you can see the /pseud marker and also the initr marker. Shift-click near one of those and drag to the other, then unshift-unclick in that order. You will see at the bottom an elapsed time of 36.9 usec or so between those two.

The pseud code has 10,240 of all the page faults. When the 40MB buffer is allocated, Linux just builds page table entries (PTEs) for 10,240 pages but points them all to a single kernel all-zero page and marks the pages read-only. As the pseud code writes to each new page it gets a page fault, which causes the kernel to allocate a real page in memory and do a 4KB copy-on-write of the zeros to that page, then return to retry the faulting instruction. There are about 27 msec between the pseud markers and during this time page-fault takes about 9 msec, or 1/3 of the total time. Standard performance tools don't show you any of this casue-and-effect detail.

Let's look at how this program got started. Pan and zoom around time 40.512 to put 512.0 to 513.0 msec on screen, then click the User button to see the five processes that run during this millisecond. You will notice one copy of bash on CPU 3 and a second copy on CPU 0. At the far left at time 40.51206 (just after the red sine wave on CPU 3), shift-click on the thin light-blue/dark-blue/green rectangle to see that it is an eth0 interrupt running for 4.45 usec. Shift-click on the adjacent light-blue/red/olive rectangle to see that it is BH:rx running for 22.3 usec. "BH" stands for "bottom half" -- the part of kernel interrupt processing that is interruptible by subsequent interrupts. This code is also known as soft IRQ handling, in contrast to the not-interruptable hard IRQ handling of the first 4.45 usec. No tools other than KUtrace show you this much detail with sufficiently low overhead to use on live real-time code.

The Ethernet interrupt brings in a remote shell (ssh) command-line packet, "./mystery3". The interrupt handler causes the ssh daemon process sshd to run on CPU 1, which runs kworke~1 on CPU 2 and then bash on CPU 3. Bash echoes the command line back through kworke~1 and sshd, then parses the command line and at time 512.38 does a long 152 usec clone() system call to make a copy of itself, which then runs on CPU 0. If you search for **`ipi`** (interprocessor interrupt) in the search box, you will see the sendipi action and subsequent reshedule_ipi interrupt that connect these together. Searching for **`ipi|runn`** will also reveal the points at which each blocked process is made runnable. Dotted blue arcs connect the make-runnable points to the first subsequent execution of the specified process.

If you click-drag from the original eth0 interrupt's make-runnable to the other end of the blue dashed arc, you will see that it took over 33 usec to get the sshd process running, part of that time spent in getting CPU 1 out of deep sleep in the idle loop (the red sine wave of 13.3 us), part in the resched interrupt 1.32us, part in the scheduler code 5.00us, and part in the tail end of the select() system call 9.23us that blocked long before this trace started. The user-mode sshd code starts at time 40.512122, which is over 77 usec from the delivery of the hardware Ethernet interrupt when we count the 13.3 usec CPU 3 also takes to get out of deep sleep. If you care about interrupt latency, KUtrace can explain the root causes of long ones.

Move now to the /pseud marker (time 40.540886) and zoom in to see the initr marker with about 500 usec across the screen. Shift-click on the light-green/blue/green rectangle at 40.540925 to see that it is an openat(65436) system call with return-value 3. This 3 is the file descriptor number for the 40MB write(3) that happens next at time 40.540980. Zoom out to see that this write goes on for almost 8.5 msec, but it is broken up, first by timer interrupt processing at 40.5414, and then by another one 4 msec later at 40.5454. At 40.5486 an unexpected (to me) thing happens -- the write makes the kswapd0 process runnable on CPU 1. The swapper then runs for almost 8 msec, at which point (40.557) the write resumes and finishes. This is followed by the close(3) call at 40.559186 and the /initr marker. All

this complexity, and the 40MB is not yet done. The write() is done, but the data is not yet on disk. We know this because the write start-to-finish took about 18 msec but our particular cheap slow disk only transfers about 50-60 MB/sec so 40 MB should take on the order of 700-800 msec, not 18.

Zoom out via the red dot and search for syncfs. You will see /zero marker and then the syncfs(3) system call at about 40.564, completing at 41.287, some 723 msec later. This is about the right amount of time for the full 40 MB transfer to disk. But there is some interesting sub-structure to this transfer.

Put the synchfs start and end on the screen but not much more, then search for **`block`** to find disk interrupts. BH:block is actually the bottom half of the unnamed #526 disk interrupt. These disk interrupts are all delivered on CPU 2, and there are about 57 of them for our 40MB transfer, an odd number indeed. The transfer is 10,240 4KB blocks, and we have 57 interrupts, so one every ~179 blocks. This is suspiciously close to the track size on our cheap disk.

If you zoom in to about 100 msec across the screen and somewhere in the middle of the syncfs and search for **`block`** again, you will notice that the block interrupts are not evenly spaced, but are instead alternately spaced closer together and farther apart. I don't know why. Zoom in on one that is after a smaller gap, such as 40.875. Here we find the bottom handler runs for a long time (253 usec) and during that time makes the jbd2/sda-8 process (file system journal) runnable 23 times and makes the mystery3 process runnable once. I don't know what is going on here, but if your program is spending a lot of time doing this, it might be time to look more carefully at the syncfs kernel implementation. Simpler tools such as **`iotop`** will tell you jbd2/sda-8 runs, but not when or why.

Finally, let's look at the asynchronous read. Go to the read marker and notice the clone() system call at 41.287370. This is process ID 9994 (mystery3) on CPU 0 starting a second thread 9996 on CPU 3. The original thread continues executing and the second thread immediately does a (synchronous) pread64 and then blocks for about 675 msec until the 40MB pread64 finishes. This is how asynchronous I/O is implemented -- synchronous I/O on a second thread. Meanwhile, the main thread on CPU 0 is scanning the pre-zeroed buffer looking for non-zero disk blocks to arrive. But this thread has an unusual behavior.

With the entire read../read marker pair time filling the screen, click the IPC button. At the beginning of the read, mystery3 on CPU0 runs at low instructions per cycle, about 3/8 -- 3 instructions every 8 cycles. But it gradually speeds up to about 2.5 instructions per cycle near the end of the read. Through the little IPC speedometer triangles, you are looking at a direct measurement of a performance optimization in mystery3. Scanning the 40MB buffer looking at the first word of each 4KB block to see if it is still zero uses just the first cache line of each of the 10,240 4KB blocks. But these lines all fall into the same set in the L1 set-associative cache since they are exactly 4KB apart, and also fall into the same set in the L2 cache. The set size of both is 8 lines on the processor we used, into which we are trying to cram 10,240 different cache lines. So we see nearly 100% cache misses out to L3 or main memory. It is the same 100% miss story for accessing these 10,240 pages in the translation lookaside buffer, TLB. This makes the scanning initially slow. (It would be a lot slower if there weren't a dozen or so reads outstanding at once in this sophisticated processor.)

The optimization is that mystery3 keeps a modest-size array of the 10K observed transition times, all initially zero. Only if the transition time for a given 4KB block is still zero does mystery3 look at the 40MB buffer to see if that block has just arrived. Near the end of the read, transitions for almost all the blocks have been seen and recorded as non-zero times, so only occasionally does the code look at the 40MB buffer itself. This makes the scanning loop get progressively faster as more and more blocks arrive in memory. The overall effect in the mystery3 program is that the block-arrival timestamps are more precise near the end of the read.

I hope that this little walkthrough has given you some appreciation for the power of KUtrace to show the microsecond-level dynamics of complex software. When it is used to trace real-time software -- in datacenters, in database processing, and even in embedded real-time control systems -- it can exactly show the root causes for all delays. Enjoy.