

P56

Lecture four : KUtrace

Friday 15:00-17:00 SE02

Andrew.Moore@cl.cam.ac.uk

Rethinking the second report

Seeking to incorporate some performance effort driven by your own examples....

Report2 will include the later labs (in the same/similar style as report1)

And.....

a write up of your experiences, I propose we also do....

Presentations

(For presentation-shy; we could do presentations as a fixed 'credit' value)

Objective: see what each of you discovered – share and enjoy.

Your presentations (still ideas)

Pick a program whose performance you care about, details to follow. A program that you know well enough to **make a sketch** of where you think it spends all the time.

Run it with KUtrace to see where the time is really going

Talk about what you discovered for 10-12 minutes, with pictures

Picking a suitable program

One you wrote, or one whose source you have

Able to run it on our I51-piXXX machines

Can be configured to run for 10-20 seconds (or longer but you can usefully trace some 10-20 second subset)

Is not 100% CPU-bound nor 100% I/O-bound; multiple threads and network I/O are good

Allows you to add a few tracing lines to the source to delimit major sections or show some progress counter

Ideally: Something whose performance you care about and can sketch

Program proposals (still ideas)

If your proposal is a Python program, does your sketch of where the time goes involve the JIT compiler to bytecode and then the bytecode interpreter?

If your proposal needs extra software installed, do a trial run within this week and let me know what to install.

Presentations (idea TBC)

Pick a program whose performance you care about

Run it with KUtrace to see where the time is really going

Talk about what you discovered for 10 minutes, with pictures

Ten slides maximum and Ten minutes maximum.

Send me Google Slides link or PowerPoint the evening before

I will make one large slide deck for all the presentations in a day

No fly-ins or other fancy effects

Kernel-user tracing

What is KUtrace?

A complete trace of all *transitions* between kernel mode and user mode on all CPU cores of one server machine.

- Minimal events to assign 100% of the CPU time for all cores
- Not so many events that tracing quickly is impractical
- Can be postprocessed into complete timelines

Tracing is the *only* tool that can capture unexpected interactions or interference between programs

Motivation:
Observability wish lists
and KUtrace

The Four Fundamental Resources, plus one

CPU

Memory

Disk/SSD

Network

Software locks

Tracing all kernel-user transitions plus a bit more covers much of our wish lists

CPU

- + Trace different processes vs. time
- + Interrupts
 - Spinlocks instr/second [\[see locks below\]](#)
- + CPU clock speed [trace mwait in idle loop]
 - Hyperthread interference (functional units) **[no good view here]**
 - Hyperthread interference (memory) [\[see memory below\]](#)
- + Request arrival, response send [\[insert req/resp RPCid trace points in kernel\]](#)

Tracing all kernel-user transitions plus a bit more covers our wish lists

Memory **[no good views here]**

- Physical memory bandwidth

- TLB

- Access patterns

- + Cache misses caused by others [\[worse instr/cycle, IPC\]](#)

Disk -- we see all sysread(), syswrite, completion interrupts

- + Disk queue depth

- + Disk accesses

- + Disk access time/latency [blktrace can show this]

Tracing all kernel-user transitions plus a bit more covers our wish lists

Network -- we see all sendmsg(), recvmsg(), completion interrupts

- + Network load at small time scale, Tx Rx bytes/sec [tcpdump can show this]

- + Network queue depth

 - Network switch load, queues [not directly visible, but can see send/receive]

Locks [\[insert acquire/release trace points in lock library\]](#)

- + Lock access pattern

- + Lock hold time

- + Lock acquire time

Kernel-User implementation

Goals drive the design, CPU overhead

Less than 1% CPU overhead, about 200,000 transitions per CPU core per second

=> Record to RAM; nothing else is fast enough

=> Each CPU core must write to its own buffer block to avoid cache thrashing

200,000 transitions = one every 5 usec. 1% overhead = **50 nsec budget**

50 nsec \sim one cache miss to RAM, so

=> Size of each trace entry must be much less than a cache line; **4 8 and 16 bytes** are the only realistic choices.

Goals drive the design, memory overhead

Less than 1% memory overhead, for 30-120 seconds

For a server with 256 GB of main memory, 1% is ~2.5 GB

For N bytes per trace entry and 24 CPU cores and 200K transitions per second, 60 seconds needs

$$\begin{aligned} & 24 * 200K * 60 * N \text{ bytes} \\ & = 288M * N \text{ bytes} \end{aligned}$$

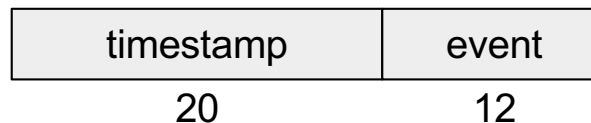
=> This implies that $N \leq 9$. [KUtrace](#) choses four bytes.

Goals drive the design, memory overhead

At four bytes each, a trace entry has room for

20-bit timestamp

12-bit event number



timestamp: cycle counter shifted 6 bits = ~20ns resolution, 20ms wrap
must guarantee at least one event per 20ms to reconstruct full time -- timer IRQ

event: ~300 system calls, 300 returns, 300 32-bit system calls, 300 returns,
12 interrupts, 12 faults, a few others

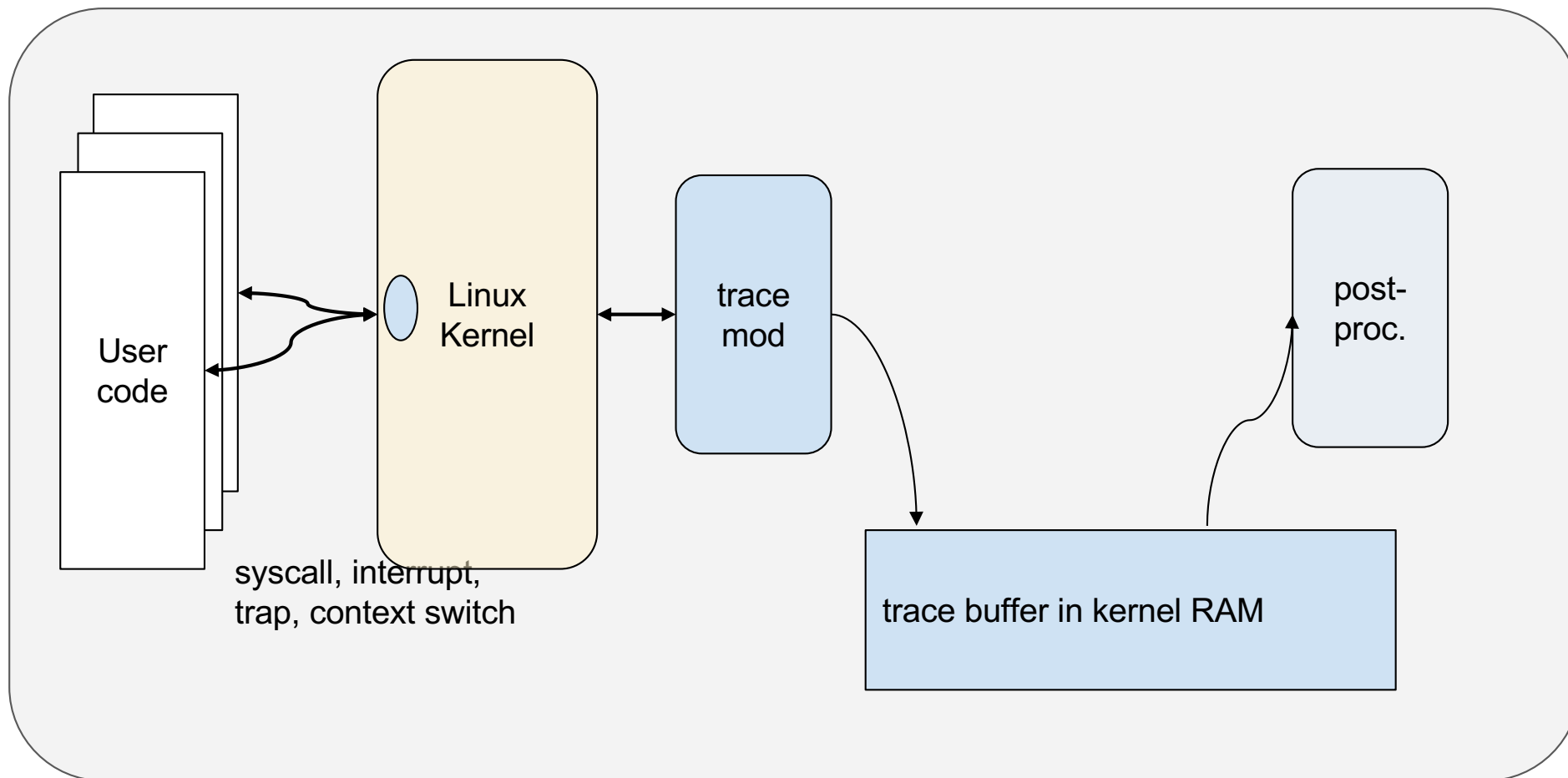
Goals drive the design -- important embellishment

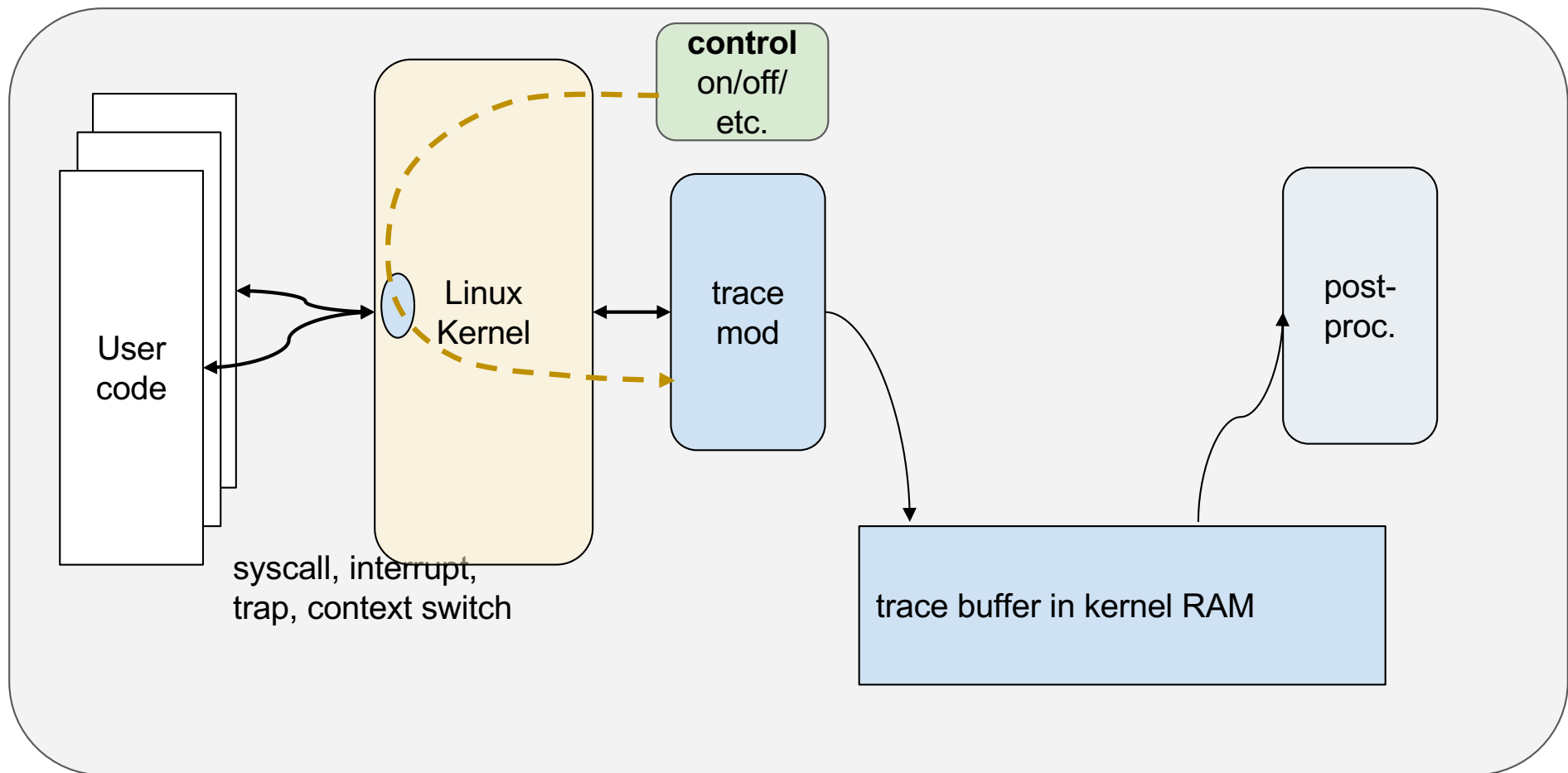
Ross Biro observed in 2006 that it is particularly useful to have some bits of the **first parameter** of any syscall, and to have the matching **return value**. To do so, we put call + return into a single 8-byte entry

20-bit timestamp	timestamp	event	delta	retval	arg0
12-bit event number	20	12	8	8	16
8-bit delta-time (call to return)					
8-bit retval					
16-bit arg0					

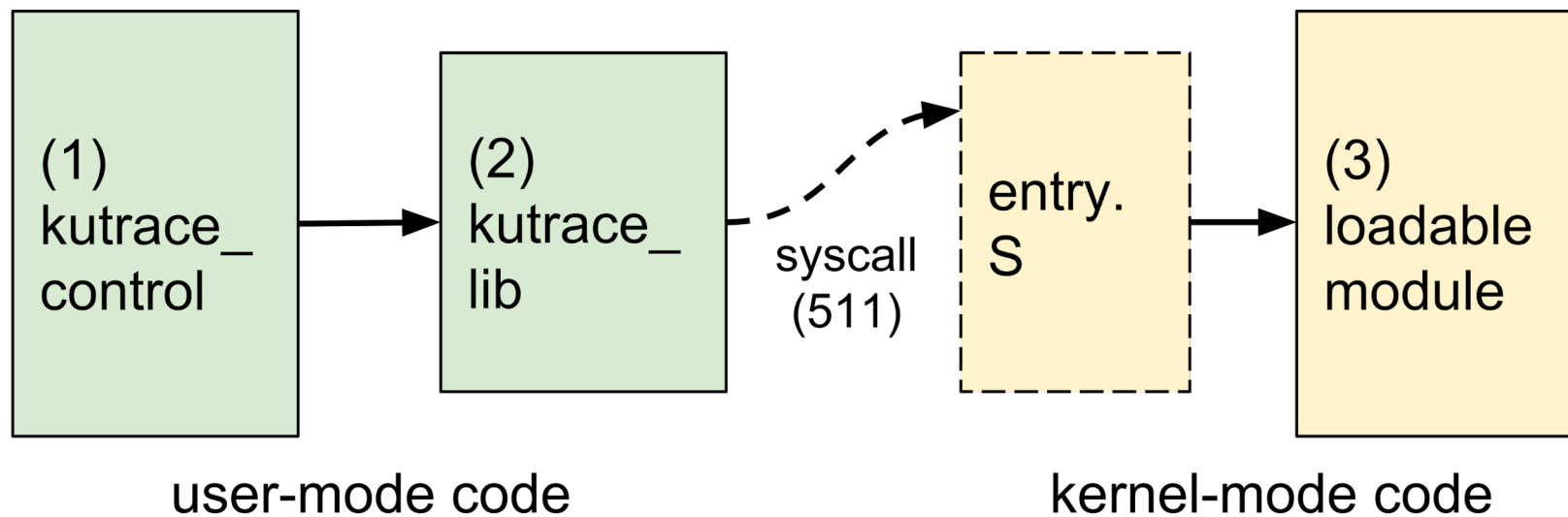
As a useful side-effect, this makes trace-entry recording substantially faster.

(Occasionally, if delta or retval does not fit in 8 bits, or if some event is inbetween, two entries are used.)





Controlling KUtrace



kutrace_control program

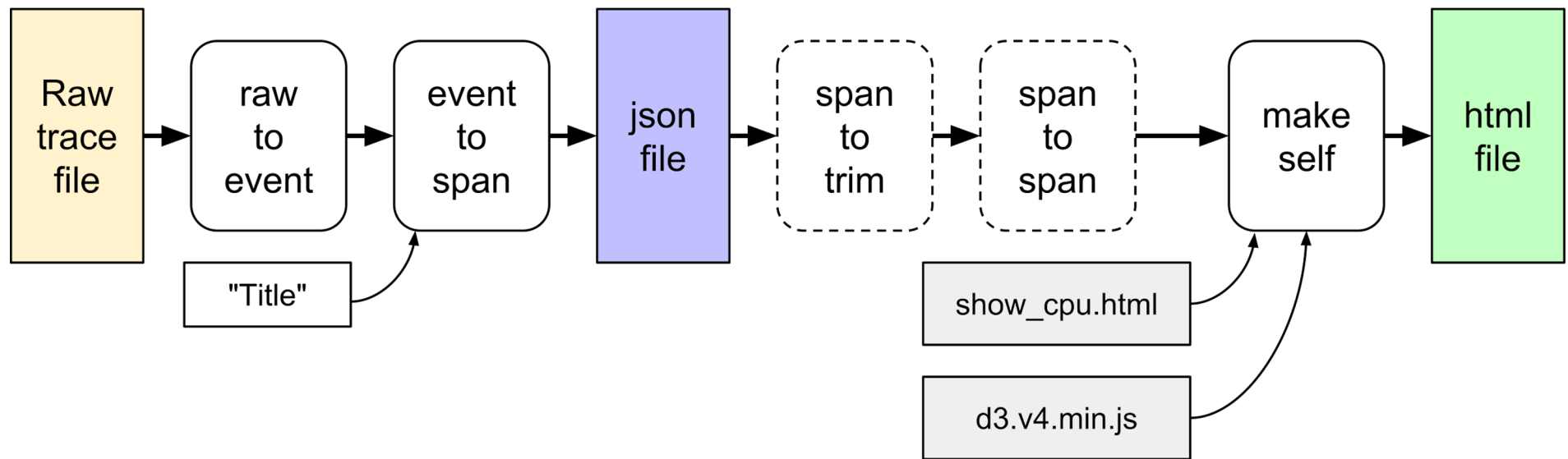
```
./kutrace_control
```

```
> go
```

```
> stop
```

```
ku_xxxx.trace written
```

Postprocessing



Postprocessing quick summary

```
export LC_ALL=C
```

```
cat ku_xxxx.trace | ./rawtoevent | sort -n | ./eventtospanspan "title" | sort > ku_xxxx.json
```

```
cat ku_xxxx.json | ./makeself show_cpu >ku_xxxx.html
```

```
google-chrome ku_xxxx.html
```


rawtoevent output sample (text)

[1] 2018-06-24_12:48:58.335301

T DUR EVENT CPU PID RPC ARG0 RETVAL IPC NAME (10ns time)

5940928129 0 1038 3 10870 0 0 0 13 page_fault (40e)

5940928563 0 1550 3 10870 0 0 0 10 /page_fault (60e)

5940928949 0 2050 3 10870 0 7520 0 13 open (802)

5940929482 0 2562 3 10870 0 0 3 13 /open (a02)

5940929517 0 2048 3 10870 0 3 0 0 read (800)

5940929761 0 2560 3 10870 0 0 832 2 /read (a00)

5940929967 59 2053 3 10870 0 3 0 3 fstat (805)

5940930094 0 2057 3 10870 0 0 0 0 mmap (809)

5940930394 0 2569 3 10870 0 0 12288 9 /mmap (a09)

eventspan output sample (JSON)

```
[ 59.40928129, 0.00000434, 3, 10870, 0, 1038, 0, 0, 10, "page_fault"],  
[ 59.40928563, 0.00000386, 3, 10870, 0, 76406, 0, 0, 13, "bash.10870"],  
[ 59.40928949, 0.00000533, 3, 10870, 0, 2050, 7520, 3, 13, "open"],  
[ 59.40929482, 0.00000035, 3, 10870, 0, 76406, 0, 3, 0, "bash.10870"],  
[ 59.40929517, 0.00000244, 3, 10870, 0, 2048, 3, 832, 2, "read"],  
[ 59.40929761, 0.00000206, 3, 10870, 0, 76406, 0, 832, 3, "bash.10870"],  
[ 59.40929967, 0.00000059, 3, 10870, 0, 2053, 3, 0, 0, "fstat"],  
[ 59.40930026, 0.00000068, 3, 10870, 0, 76406, 0, 832, 0, "bash.10870"],  
[ 59.40930094, 0.00000300, 3, 10870, 0, 2057, 0, 12288, 9, "mmap"],
```

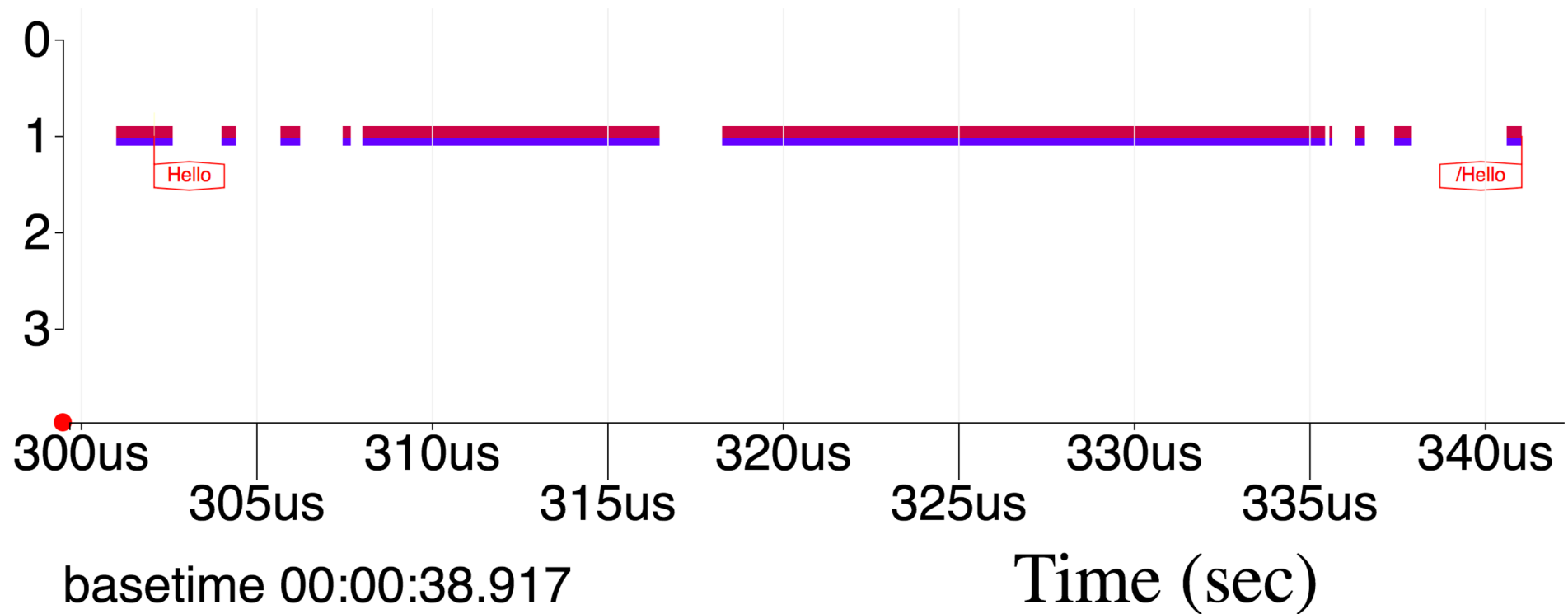
Hello World, briefly

Hello world

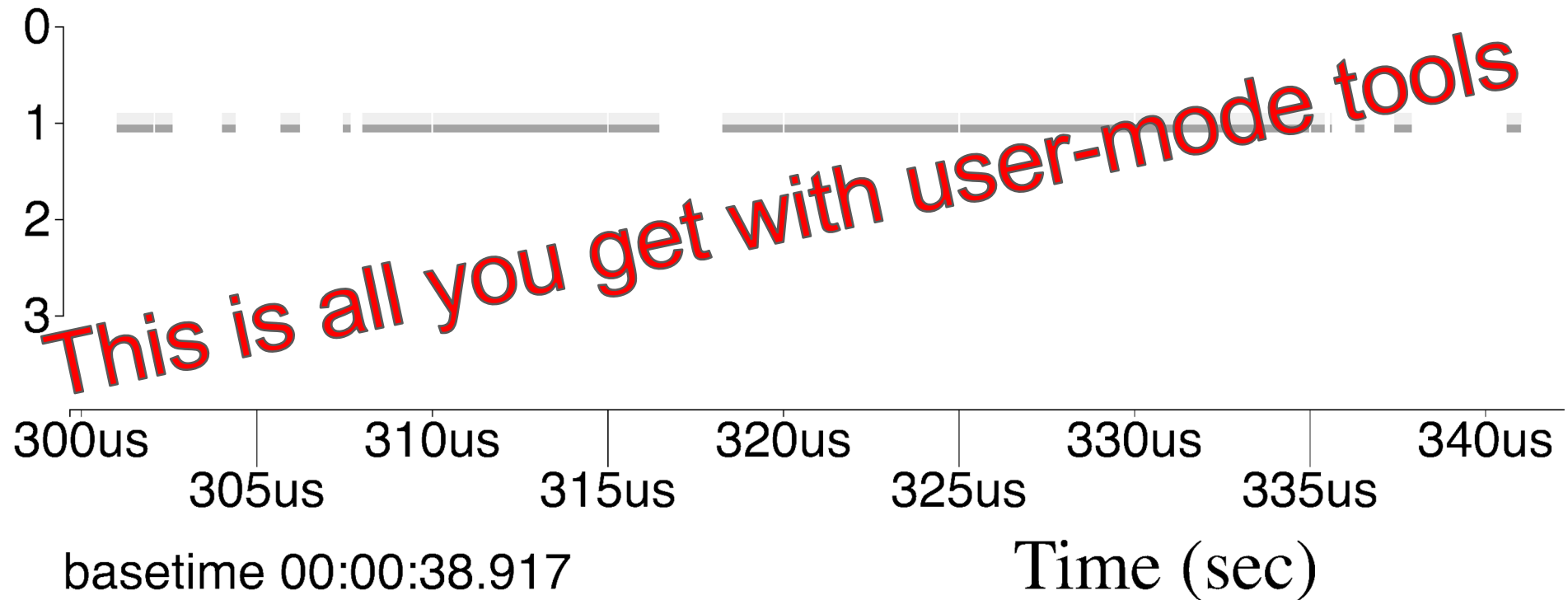
```
int main(int argc, const char** argv) {  
    kutrace::mark_a("Hello");  
    printf("hello world\n");  
    kutrace::mark_a("/Hello");  
    return 0;  
}
```



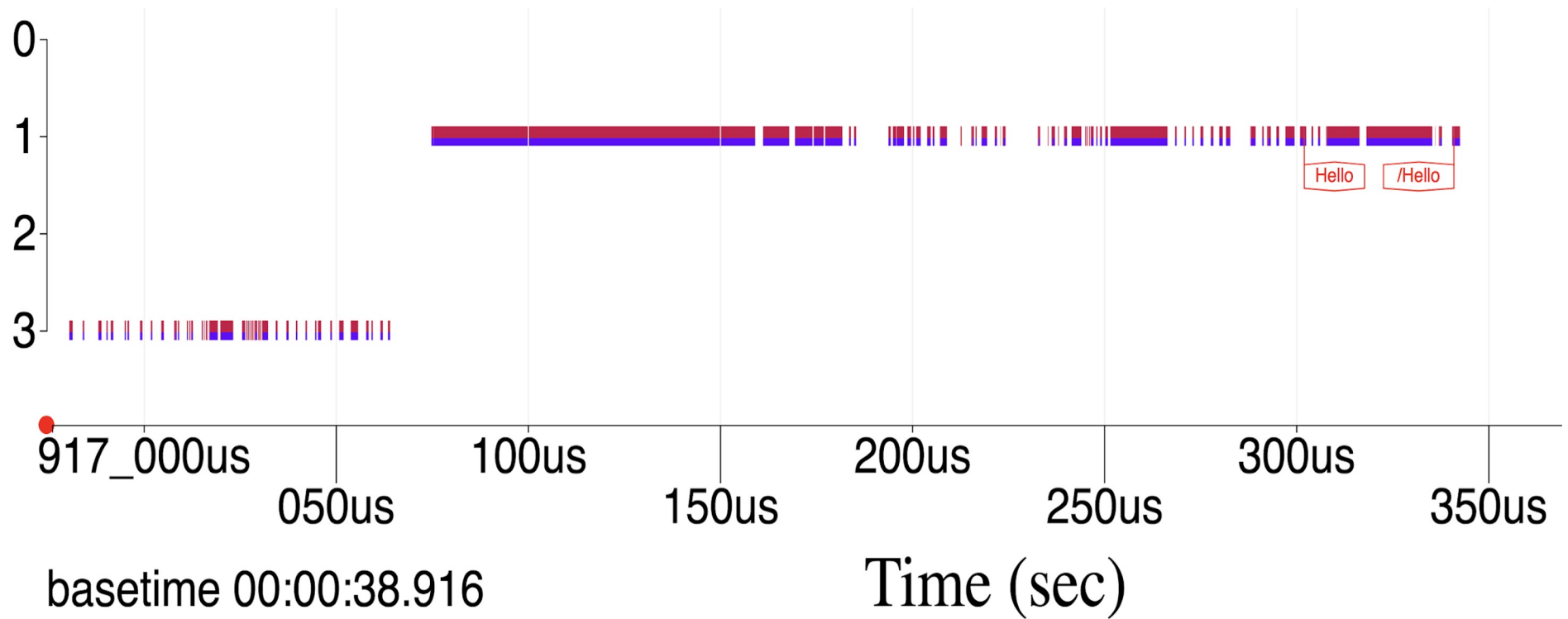
Hello world user-mode main program



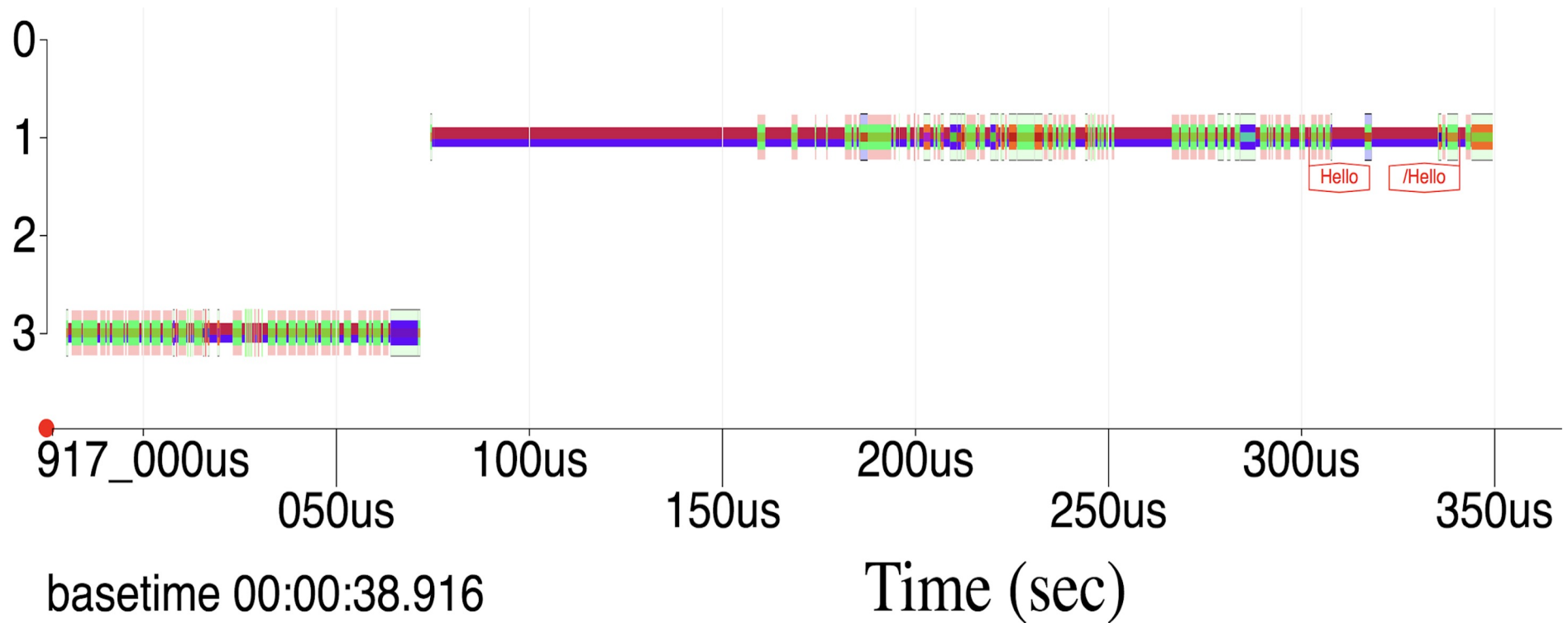
Hello world user-mode main program



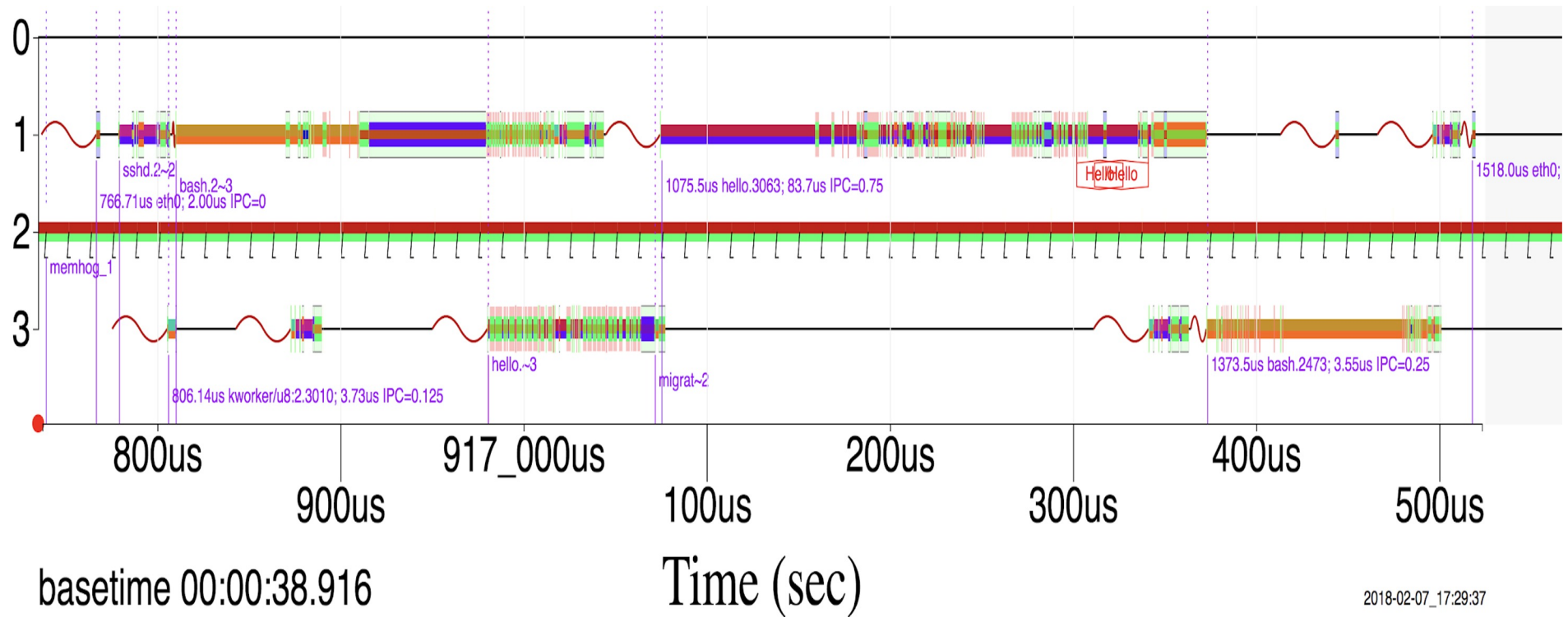
Hello world user-mode **all**



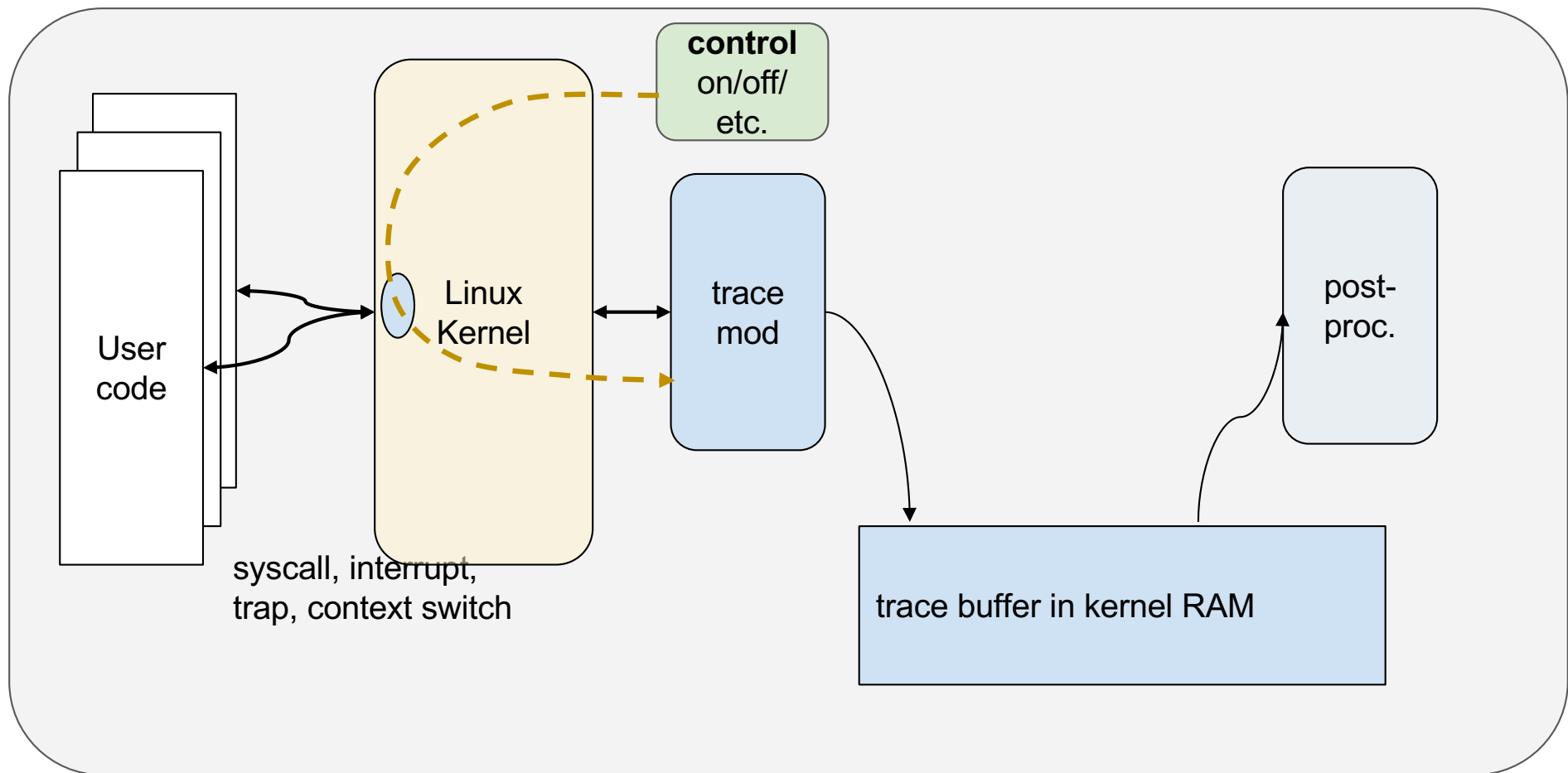
Hello world user-mode **plus kernel-mode**



Hello world **plus other programs**



Kernel-User implementation



kutrace_control program

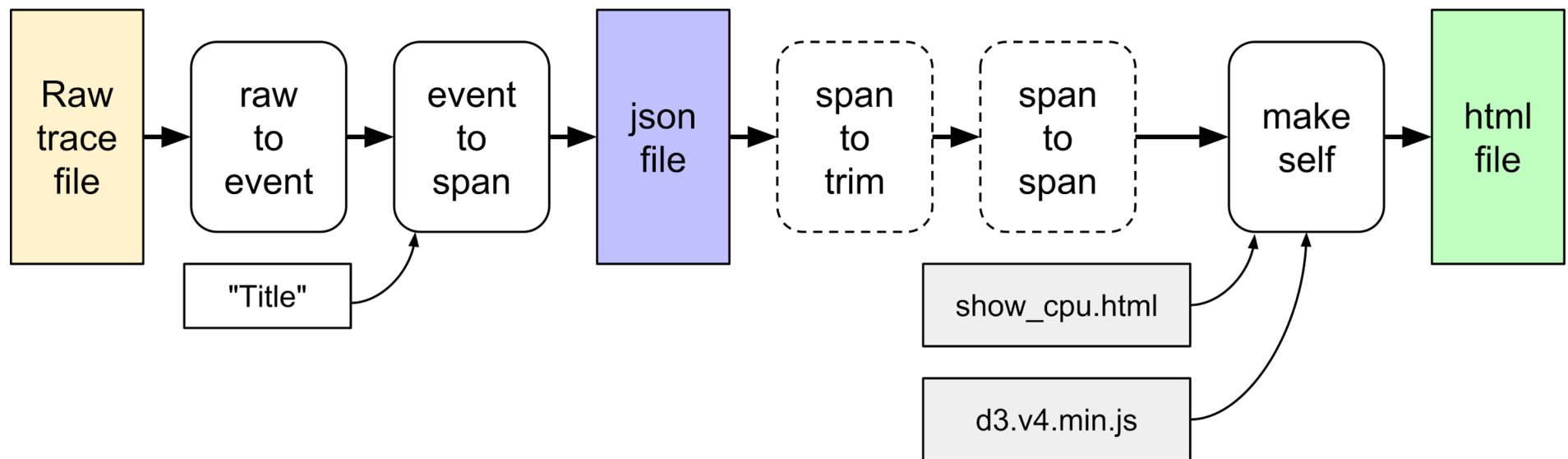
```
$ ./kutrace_control
```

```
> go
```

```
> stop
```

```
ku_xxxx.trace written
```

Postprocessing



Postprocessing quick summary

```
$ export LC_ALL=C
```

```
$ cat ku_xxxx.trace | ./rawtoevent | sort -n | ./eventtospan "title" | sort > ku_xxxx.json
```

```
$ cat ku_xxxx.json | ./makeself show_cpu_2019 >ku_xxxx.html
```

```
$ google-chrome ku_xxxx.html
```

Kernel-User Linux patches

Where we want patches

syscall / return
interrupt / return
fault / return
context switch
other
(missing patches)

All 12 patched code files

linux-4.19.19/drivers/idle/intel_idle.c.patched	mwait idle
linux-4.19.19/drivers/acpi/acpi_pad.c.patched	mwait power
linux-4.19.19/drivers/acpi/processor_idle.c.patched	mwait idle
linux-4.19.19/arch/x86/kernel/apic/apic.c.patched	irq timer
linux-4.19.19/arch/x86/kernel/irq_work.c.patched	ipi irq work vec
linux-4.19.19/arch/x86/kernel/smp.c.patched	ipi rfesched, fnc
linux-4.19.19/arch/x86/kernel/irq.c.patched	irq
linux-4.19.19/arch/x86/kernel/acpi/cstate.c.patched	mwait power
linux-4.19.19/arch/x86/mm/fault.c.patched	pagefault
linux-4.19.19/arch/x86/entry/common.c.patched	syscall, trace_control
linux-4.19.19/kernel/sched/core.c.patched	sched, runnable,
pidname	
linux-4.19.19/kernel/softirq.c.patched	bottom half

syscall / return patches

linux-4.19.19/arch/x86/entry/common.c

all syscalls, most returns [others exit
through the scheduler]

kutrace_control hook

interrupt / return patches

linux-4.19.19/arch/x86/kernel/irq.c

most interrupts, top half

linux-4.19.19/kernel/softirq.c

bottom half

linux-4.19.19/arch/x86/kernel/apic/apic.c

timer interrupt

linux-4.19.19/arch/x86/kernel/smp.c

send and receive interprocessor

interrupt, IPI

linux-4.19.19/arch/x86/kernel/irq_work.c

more IPI

fault / return patches

linux-4.19.19/arch/x86/mm/fault.c

page fault

context switch patches

linux-4.19.19/kernel/sched/core.c

context switch, scheduler itself, new
process names, make-runnable

other patches

linux-4.19.19/drivers/idle/intel_idle.c

linux-4.19.19/drivers/acpi/acpi_pad.c

linux-4.19.19/drivers/acpi/processor_idle.c

linux-4.19.19/arch/x86/kernel/acpi/cstate.c

power-saving in idle loop and elsewhere

User-code patches

dclab_rpc.c

RPC send/receive

spinlock.cc

Contended lock acquire/release

Syscall patch

Syscall Linux code, original

```
_visible void do_syscall_64(unsigned long nr, struct pt_regs *regs)
{
    enter_from_user_mode();
    local_irq_enable();
    ...
    nr &= __SYSCALL_MASK;
    if (likely(nr < NR_syscalls)) {
        nr = array_index_nospec(nr, NR_syscalls);
        regs->ax = sys_call_table[nr](regs);
    }
    syscall_return_slowpath(regs);
}
```

Syscall Linux code, patched

```
__visible void do_syscall_64(unsigned long nr, struct pt_regs *regs)
    ...
    if (likely(nr < NR_syscalls)) {
        nr = array_index_nospec(nr, NR_syscalls);
        kutrace1(KUTRACE_SYSCALL64 | kutrace_map_nr(nr), regs->di);

                                                // arg0

        regs->ax = sys_call_table[nr](regs);
        kutrace1(KUTRACE_SYSRET64 | kutrace_map_nr(nr), regs->ax);

                                                //ret
    }
    ...
```

Syscall Linux code, additional patch for control

```
if (likely(nr < NR_syscalls)) {  
    ...  
    regs->ax = sys_call_table[nr](regs);  
    kutrace1(KUTRACE_SYSRET64 | kutrace_map_nr(nr), regs->ax);  
} else if ((nr == __NR_kutrace_control) &&  
    (kutrace_global_ops.kutrace_trace_control != NULL)) {  
    /* Calling kutrace_control(u64 command, u64 arg) */  
    regs->ax = (*kutrace_global_ops.kutrace_trace_control)(  
        regs->di, regs->si);  
}
```

Syscall Linux code, the macro expansion

```
#define kutrace1(event, arg) \  
    if (kutrace_tracing) { \  
        (*kutrace_global_ops.kutrace_trace_1)(event, arg); \  
    }
```

kutrace_tracing is an added global kernel Boolean: tracing on/off.

The entire overhead of the patches when NOT tracing is this test:
load, branch.

The event and arg construction are only done if tracing is on.

Hard IRQ Linux code, most interrupts -- top half

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    unsigned vector = ~regs->orig_ax;
    entering_irq();
    kutrace1(KUTRACE_IRQ + (vector & 0xFF), 0);
    desc = __this_cpu_read(vector_irq[vector]);
    if (!handle_irq(desc, regs)) {
        ...
    }
    kutrace1(KUTRACE_IRQRET + (vector & 0xFF), 0);
    exiting_irq();
    set_irq_regs(old_regs);
    return 1;
}
```

Soft IRQ Linux code -- bottom half

```
asmlinkage __visible void __softirq_entry __do_softirq(void)
{
    in_hardirq = lockdep_softirq_start();
    while ((softirq_bit = ffs(pending))) {
        kstat_incr_softirqs_this_cpu(vec_nr);
        kutrace1(KUTRACE_IRQ + KUTRACE_BOTTOM_HALF, vec_nr);
        trace_softirq_entry(vec_nr);
        h->action(h);
        trace_softirq_exit(vec_nr);
        kutrace1(KUTRACE_IRQRET + KUTRACE_BOTTOM_HALF, 0);
    }
    lockdep_softirq_end(in_hardirq);
}
```

IRQ Linux code, send interprocessor interrupts

```
static void native_smp_send_reschedule(int cpu)
{
    kutrace1(KUTRACE_IPI, cpu);
    apic->send_IPI(cpu, RESCHEDULE_VECTOR);
}

void native_send_call_func_single_ipi(int cpu)
{
    kutrace1(KUTRACE_IPI, cpu);
    apic->send_IPI(cpu, CALL_FUNCTION_SINGLE_VECTOR);
}

void native_send_call_func_ipi(const struct cpumask *mask)
{
    kutrace1(KUTRACE_IPI, 0);
    ...
}
```

IRQ Linux code, receive interprocessor interrupts

```
__visible void __irq_entry smp_reschedule_interrupt(struct pt_regs *regs)
{
    ack_APIC_irq();
    kutrace1(KUTRACE_IRQ + RESCHEDULE_VECTOR, 0);
}

__visible void __irq_entry smp_call_function_interrupt(struct pt_regs *regs)
{
    ipi_entering_ack_irq();
    kutrace1(KUTRACE_IRQ + CALL_FUNCTION_VECTOR, 0);
    inc_irq_stat(irq_call_count);
    generic_smp_call_function_interrupt();
    kutrace1(KUTRACE_IRQRET + CALL_FUNCTION_VECTOR, 0);
    exiting_irq();
}

__visible void __irq_entry smp_call_function_single_interrupt(struct pt_regs *r)
```


Sched Linux code, receive interprocessor interrupts

```
void scheduler_ipi(void)
{
    irq_enter();
    /* dsites 2019.03.06 continued from smp.c */
    sched_ttwu_pending();
    kutrace1(KUTRACE_IRQRET + RESCHEDULE_VECTOR, 0);
    irq_exit();
}
```

Page Fault code

```
do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    unsigned long address = read_cr2(); /* Get the faulting address */
    ...
    if (trace_pagefault_enabled())
        trace_page_fault_entries(address, regs, error_code);
    kutrace1(KUTRACE_TRAP + KUTRACE_PAGEFAULT, 0);
    __do_page_fault(regs, error_code, address);
    kutrace1(KUTRACE_TRAPRET + KUTRACE_PAGEFAULT, 0);
    ...
}
```

Sched Linux code, make process runnable

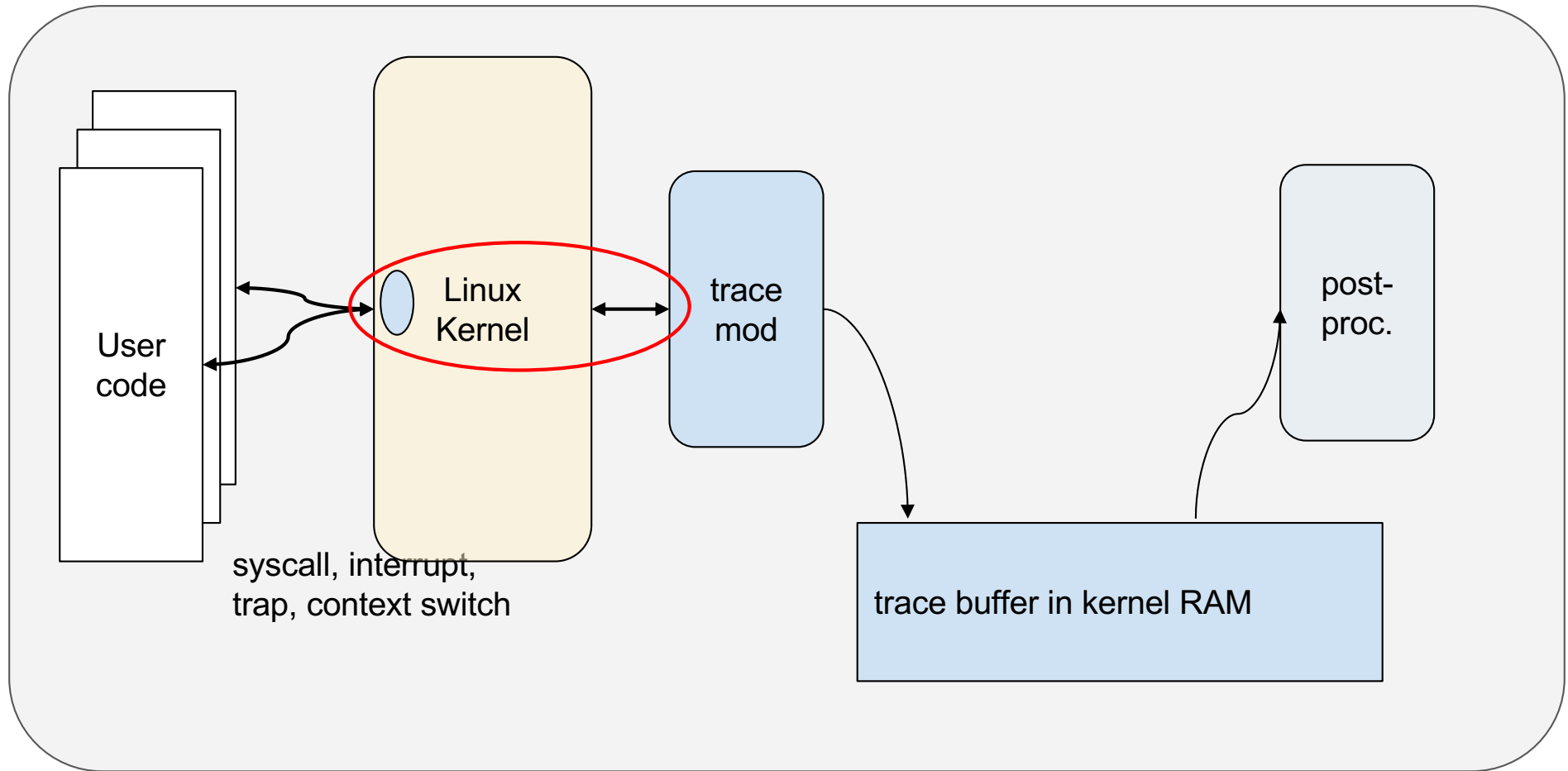
```
static int try_to_wake_up(struct task_struct *p, unsigned int state, int wake_flags)
{
    kutrace1(KUTRACE_RUNNABLE, p->pid);
}
```

```
static void try_to_wake_up_local(struct task_struct *p, struct rq_flags *rf)
{
    kutrace1(KUTRACE_RUNNABLE, p->pid);
}
```

Sched itself Linux code

```
static void __sched notrace __schedule(bool preempt)
{
    ktrace1(KUTRACE_SYSCALL64 + KUTRACE_SCHEDSYSCALL, 0);
    cpu = smp_processor_id();
    ...
    /* Put pid name into trace first time */
    ktrace_pidname(next);
    ktrace1(KUTRACE_USERPID, next->pid);
    /* Also unlocks the rq: */
    rq = context_switch(rq, prev, next, &rf);
    ...
    ktrace1(KUTRACE_SYSRET64 + KUTRACE_SCHEDSYSCALL, 0);
}
```

KUtrace kernel-module interface



Kernel-module interface

In addition to the patches to create trace entries, the kernel changes are just a few small data structure declarations.

The real code for creating and managing trace entries is in the loadable module, which runs in kernel mode.

The module can be changed, recompiled, and reloaded *without* changing the running kernel -- no reboot needed. Big win.

Kernel interface is tiny -- code is in loadable module

```
bool kutrace_tracing;
struct kutrace_ops {
    void (*kutrace_trace_1)(u64 num, u64 arg);
    void (*kutrace_trace_2)(u64 num, u64 arg1, u64 arg2);
    void (*kutrace_trace_many)(u64 num, u64 len, const char* arg);
    u64 (*kutrace_trace_control)(u64 command, u64 arg);
};

struct kutrace_traceblock { // Per-CPU struct (28 copies for 28 cores)
    atomic64_t next; // Holds a u64* that is atomically incremented
    u64* limit;
};
```


Kernel interface is tiny

```
bool ktrace_tracing = false;  
EXPORT_SYMBOL(ktrace_tracing);
```

```
struct ktrace_ops ktrace_global_ops = {NULL, NULL, NULL, NULL};  
EXPORT_SYMBOL(ktrace_global_ops);
```

```
u64* ktrace_pid_filter = NULL;  
EXPORT_SYMBOL(ktrace_pid_filter);
```

```
DEFINE_PER_CPU(struct ktrace_traceblock, ktrace_traceblock_per_cpu);  
EXPORT_PER_CPU_SYMBOL(ktrace_traceblock_per_cpu);
```

Module init code

```
static int __init kutrace_trace_mod_init(void)
{
    kutrace_pid_filter = (u64*)vmalloc(1024 * sizeof(u64));
    tracebase = vmalloc(tracemb << 20);

    /* Set up global tracing data state */
    DoReset();
    printk(KERN_INFO " kutrace_tracing = %d\n", kutrace_tracing);

    /* Finally, connect up the routines that can change the state */
    kutrace_global_ops.kutrace_trace_1 = &trace_1;
    kutrace_global_ops.kutrace_trace_2 = &trace_2;
    kutrace_global_ops.kutrace_trace_many = &trace_many;
    kutrace_global_ops.kutrace_trace_control = &trace_control;
    return 0;
}
```

pid_filter
tracebase

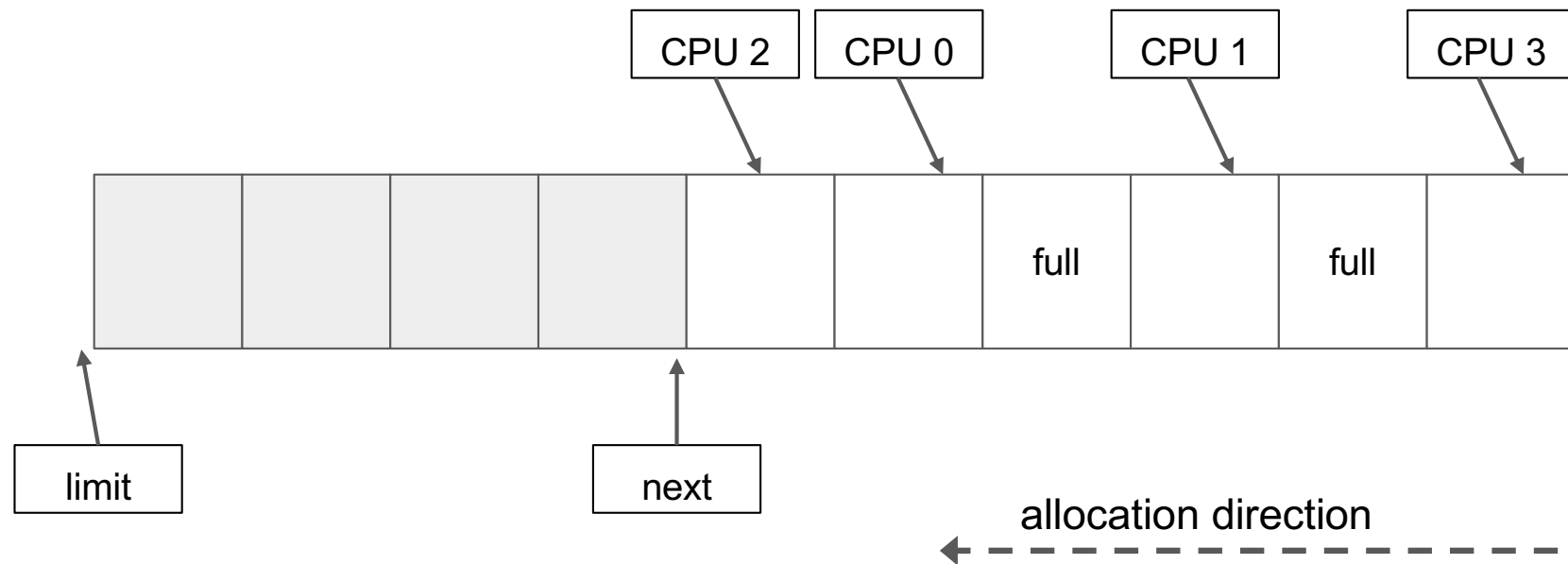
&trace_1
&trace_2
&trace_many
&trace_control

Module exit code

```
static void __exit kutrace_trace_mod_exit(void)
{
    kutrace_tracing = false;
    msleep(20);          /* wait 20 msec for any pending tracing to finish */
    /* Disconnect all the routines that can change state */
    kutrace_global_ops.kutrace_trace_1 = NULL;
    kutrace_global_ops.kutrace_trace_2 = NULL;
    kutrace_global_ops.kutrace_trace_many = NULL;
    kutrace_global_ops.kutrace_trace_control = NULL;
    /* Clear out all the pointers to trace data */
    for_each_online_cpu(cpu) {
        struct kutrace_traceblock* tb = &per_cpu(kutrace_traceblock, cpu);
        atomic64_set(&tb->next, (u64)NULL);
        tb->limit = NULL;
    } ...
}
```

pid_filter
tracebase
&trace_1
&trace_2
&trace_many
&trace_control

Per-CPU blocks of 64KB are allocated dynamically



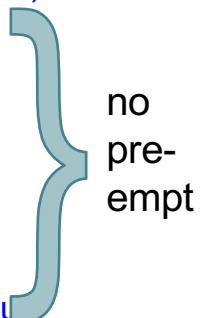
Goals drive the design -- speed even with preemptable kernel

Normal path for making one entry, ~40cy

```
void trace_1(u64 num, u64 arg) {
    if (!kutrace_tracing) {return;}
    Insert1((num << 32) | arg);
}

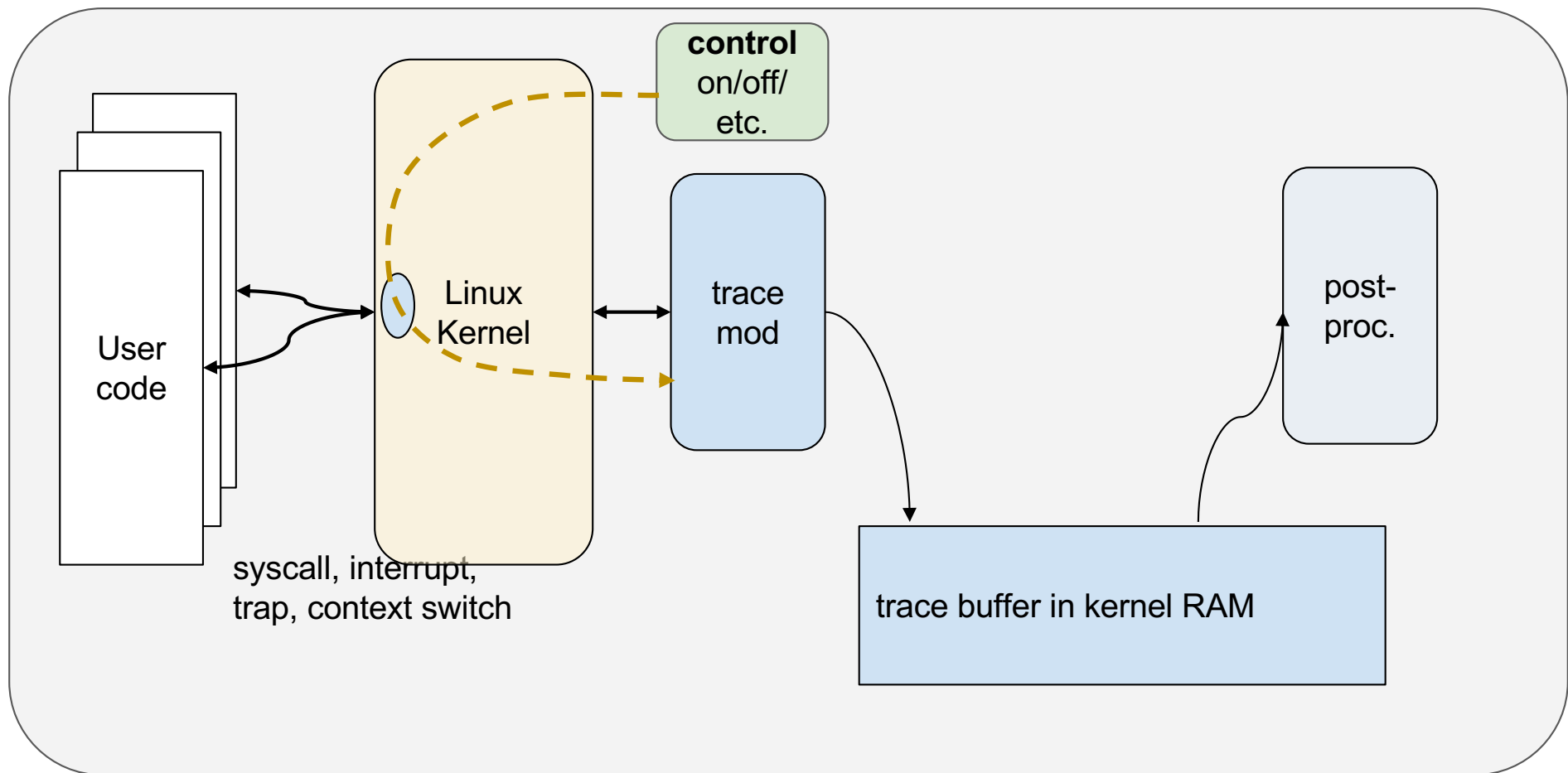
u64 Insert1(u64 arg1) {
    u64 now = get_cycles();
    u64* claim = GetClaim(1);
    if (claim != NULL) {
        claim[0] = arg1 | ((now >> RDTSC_SHIFT) << 44);
        return 1;
    }
    ...
}
```

```
u64* GetClaim(int len) {
    tb = &get_cpu_var(kutrace_traceblock_per_cpu);
    nexti = atomic64_read(&tb->next);
    limiti = tb->limit;
    if (nexti < limiti) {
        u64* myclaim = (atomic64_add_return(
            len * sizeof(u64), &tb->next)) - len;
        if (myclaim < limiti) {
            put_cpu_var(kutrace_traceblock_per_cpu);
            return myclaim;
        }
    }
    ... // Slower path to alloc new block
}
```



no pre-empt

KUtrace control



kutrace_control program

```
./kutrace_control
```

```
> go
```

```
> stop
```

```
kutrace_XXXX.trace written
```


kutrace_control.cc commands

```
./kutrace_control  
  > go  
  > stop  
ku_xxxx written
```

go: reset the tracing buffer to empty and start tracing. Variants **gowrap**, **goipc**, and **goipcwrap** enable flight-recorder wraparound tracing, tracking instructions per cycle (IPC)

stop: stop tracing, pad out any partial trace blocks with NOPs, and write the resulting raw binary trace file

ku_20180606_121314_3456.trace

This raw trace file is self-contained, with human-readable names as embedded trace entries.

kutrace_control.cc secondary commands

dump: write a raw trace file to disk/ssd. Must be preceded by off and then flush.

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

init: insert 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: turn tracing off.

on: turn tracing on. The first time, it must be preceded by at least reset.

kutrace_control.cc secondary commands

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.

kutrace_control.cc secondary commands

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.

```
./kutrace_control 1  
./kutrace_control 0
```

This is possibly useful in scripts, once the initial tracing setup is done via the prompt/command interface.

kutrace_lib.cc, for programs to control tracing

The user-mode kutrace_lib.cc library has routines that directly implement the above commands. These are declared inside the **kutrace::** namespace. Any user code can link in the library and directly call these routines, to build alternate control or self-tracing programs.

```
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
```

kutrace_lib.cc, for programs to control tracing

const char* **Base40ToChar**(u64 base40, char* str); Converts a uint32 value that contains six packed base-40 characters into a character string.

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. Base40: **a-z0-9./-**

u64 **DoControl**(u64 command, u64 arg); Directly calls the loadable module implementation via our 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 **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.

kutrace_lib.cc, for programs to control tracing

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 host name, and PID.

kutrace_lib.cc, for programs to control tracing

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, as further described next week.

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.

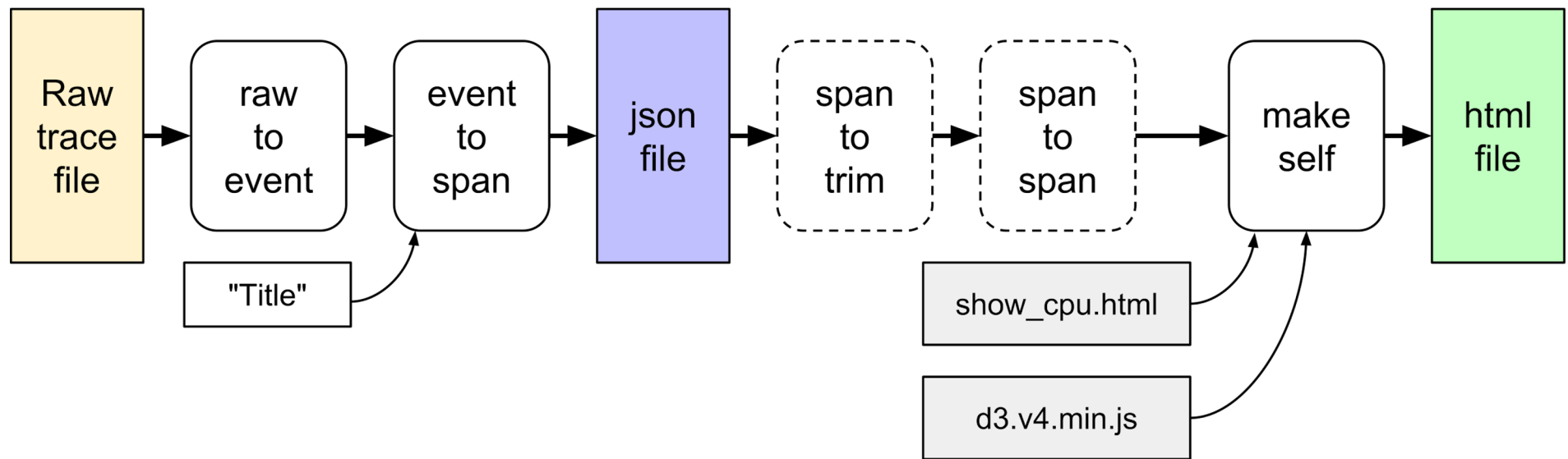
void **stop**(const char* fname); Does off/flush/dump.

kutrace_lib.cc, for programs to control tracing

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.

Postprocessing overview

Postprocessing



Postprocessing quick summary

```
cat ku_xxxx.trace | ./rawtoevent | LC_ALL=C sort -n | ./eventtospan "title" |  
LC_ALL=C sort > ku_xxxx.json
```

```
cat ku_xxxx.json | ./makeself show_cpu >ku_xxxx.html
```

(optional...)

```
google-chrome ku_xxxx.html
```

rawtoevent

rawtoevent

Convert raw binary trace file to readable text lines, one per event

- extend timestamps, convert to 10ns increments
linear interpolation between trace start/end time pairs:
<gettimeofday, time counter>
- propagate cpu, pid, rpcid
so every event has each value
- put in names
so every event has the syscall name, IRQ name, process name, etc.

rawtoevent output sample (text)

[1] 2018-06-24_12:48:58.335301

#	T	DUR	EVENT	CPU	PID	RPC	(10ns time)
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0
39	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0
43	0	0	0	0	0	0	0
44	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0
46	0	0	0	0	0	0	0
47	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0
49	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0
51	0	0	0	0	0	0	0
52	0	0	0	0	0	0	0
53	0	0	0	0	0	0	0
54	0	0	0	0	0	0	0
55	0	0	0	0	0	0	0
56	0	0	0	0	0	0	0
57	0	0	0	0	0	0	0
58	0	0	0	0	0	0	0
59	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0
61	0	0	0	0	0	0	0
62	0	0	0	0	0	0	0
63	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0
65	0	0	0	0	0	0	0
66	0	0	0	0	0	0	0
67	0	0	0	0	0	0	0
68	0	0	0	0	0	0	0
69	0	0	0	0	0	0	0
70	0	0	0	0	0	0	0
71	0	0	0	0	0	0	0
72	0	0	0	0	0	0	0
73	0	0	0	0	0	0	0
74	0	0	0	0	0	0	0
75	0	0	0	0	0		

						ARG0 RETVAL IPC NAME

5940928129 0 1038 3 10870 0 0 0 13 page_fault (40e)

5940928563 0 1550 3 10870 0 0 0 10 /page_fault (60e)

5940928949 0 2050 3 10870 0 7520 0 13 open (802)

```
5940929482 0 2562 3 10870 0 0 3 13 /open (a02)
```

5940929517 0 2048 3 10870 0 3 0 0 read (800)

```
5940929761 0 2560 3 10870 0 0 832 2 /read (a00)
```

5940929967 59 2053 3 10870 0 3 0 3 fstat (805)

```
5940930094 0 2057 3 10870 0 0 0 0 mmap (809) ...
```

not merged, dur > 255

not merged, ret > 127

merged call/ret

Ten event fields

timestamp Start time for a span in seconds and fractions. For easy correlation with logs and other files, the seconds part be offset from an exact multiple of one minute in the gettimeofday() time domain, rather than starting at zero.

duration Duration of a span in seconds and fractions.

cpu CPU number as a small integer.

pid Process ID (actually thread ID; the kernel calls this "pid"), low 16 bits.

rpcid RPC ID for datacenter work; may be used for anything

event Event number that starts the time span. Values 0-512 are names, markers, and other specials. 512-4096 kernel events system call, interrupt, fault. 64K+pid are user-mode.

Ten event fields

arg0 The low 16 bits of the first argument to a syscall, else 0.

retval For a call/return span, the low 16 bits of the return value. Byte counts, etc. can be considered unsigned. Return codes can be considered signed.

ipc Instructions per cycle, *quantized* via truncation into 4 bits.
Values 0-7 are multiples of 0.125 IPC, i.e. less than one instruction per cycle.
Values 8-11 are 1.0, 1.25, 1.5, and 1.75 IPC.
Values 12-15 are 2.0, 2.5, 3.0, and 3.5+ IPC.

name Name of the kernel routine or user PID (originally from the kernel 16-byte command field per pid). These names come from naming entries in the raw trace itself.

eventtospan

eventtospan

Convert event text file to timespans, formatted as proper JSON

- Start a timespan at each call transition, end it at each return transition
- Run a small stack of pending executions: Before call, push current execution name, after return pop back to previous. This is how we know what is running after each return.
- If a process migrates to a different CPU core, migrate its execution stack also.
- Add synthetic sine waves to indicate coming out of deep sleep
- Add some causality arcs from make-runnable to that process actually running

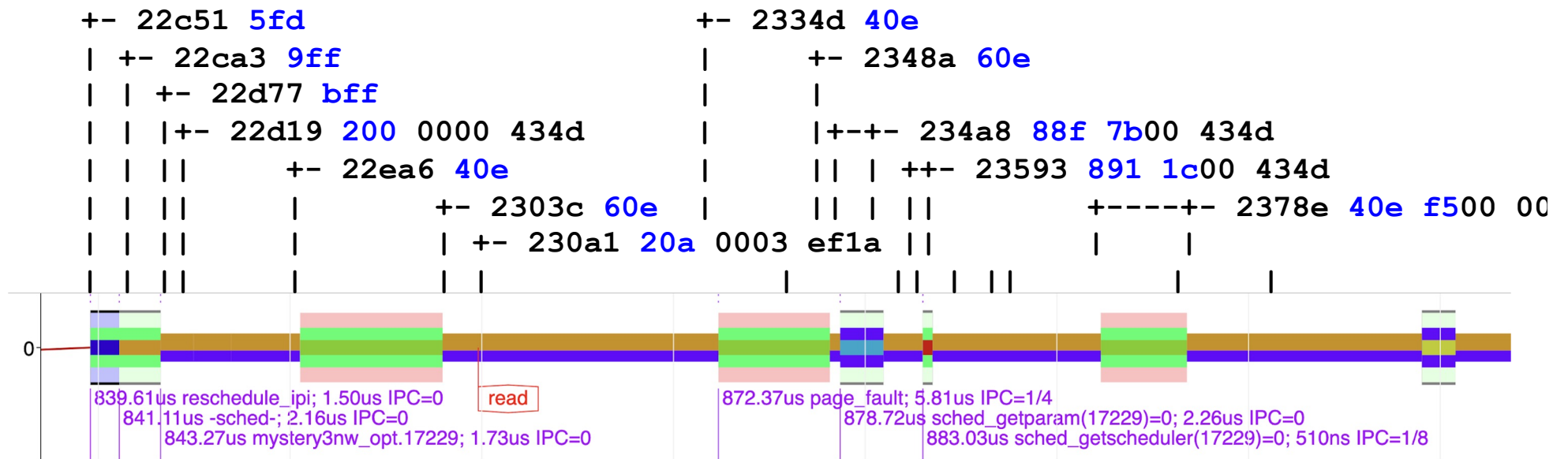
eventspan output sample (JSON)

#	T	DUR	CPU	PID	RPC	EVENT	ARG0	RET	IPC	NAME
[59.40928129,	0.00000434,	3,	10870,	0,	1038,	0,	0,	10,	"page_fault"],
[59.40928563,	0.00000386,	3,	10870,	0,	76406,	0,	0,	13,	"bash.10870"],
[59.40928949,	0.00000533,	3,	10870,	0,	2050,	7520,	3,	13,	"open"],
[59.40929482,	0.00000035,	3,	10870,	0,	76406,	0,	3,	0,	"bash.10870"],
[59.40929517,	0.00000244,	3,	10870,	0,	2048,	3,	832,	2,	"read"],
[59.40929761,	0.00000206,	3,	10870,	0,	76406,	0,	832,	3,	"bash.10870"],
[59.40929967,	0.00000059,	3,	10870,	0,	2053,	3,	0,	0,	"fstat"],
[59.40930026,	0.00000068,	3,	10870,	0,	76406,	0,	832,	0,	"bash.10870"],
[59.40930094,	0.00000300,	3,	10870,	0,	2057,	0,	12288,	9,	"mmap"],

Transitions vs. timespans: 12 raw trace entries

Time	Event	delta	ret	arg0
22c51	5fd	0000	0000	reschedule_ipi interrupt
22ca3	9ff	0000	0000	enter scheduler
22d19	200	0000	434d	context switch(17229)
22d77	bff	0000	0000	exit scheduler
22ea6	40e	0000	0000	fault page fault
2303c	60e	0000	0000	faultreturn page fault
230a1	20a	0003	ef1a	mark_a("read")
2334d	40e	0000	0000	fault page fault
2348a	60e	0000	0000	faultreturn page fault
234a8	88f	7b00	434d	syscall+ret sched_getparam
23593	891	1c00	434d	syscall+ret sched_getscheduler
2378e	40e	f500	0000	fault+ret page fault

Transitions vs. timespans: 12 raw trace entries



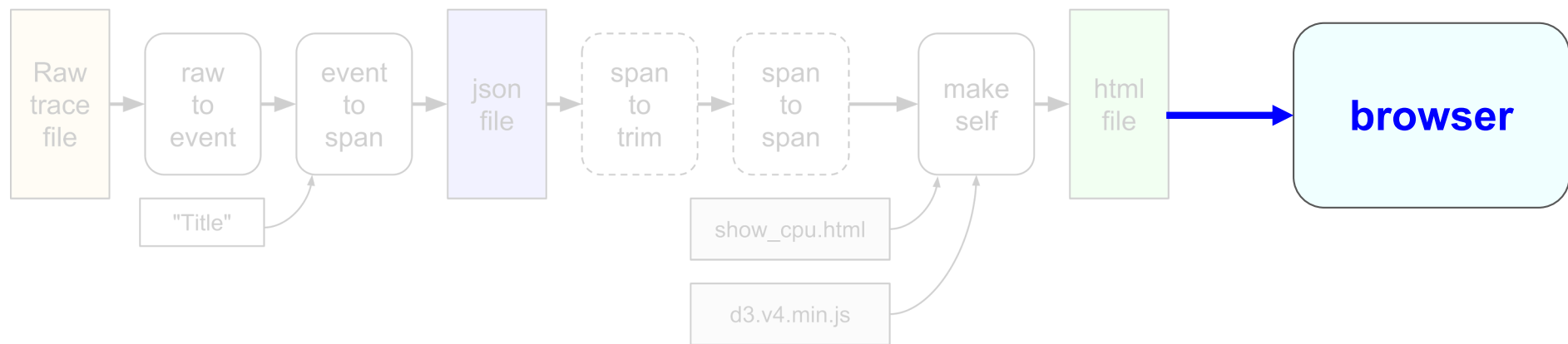
JSON format

JSON; note leading spaces to sort properly

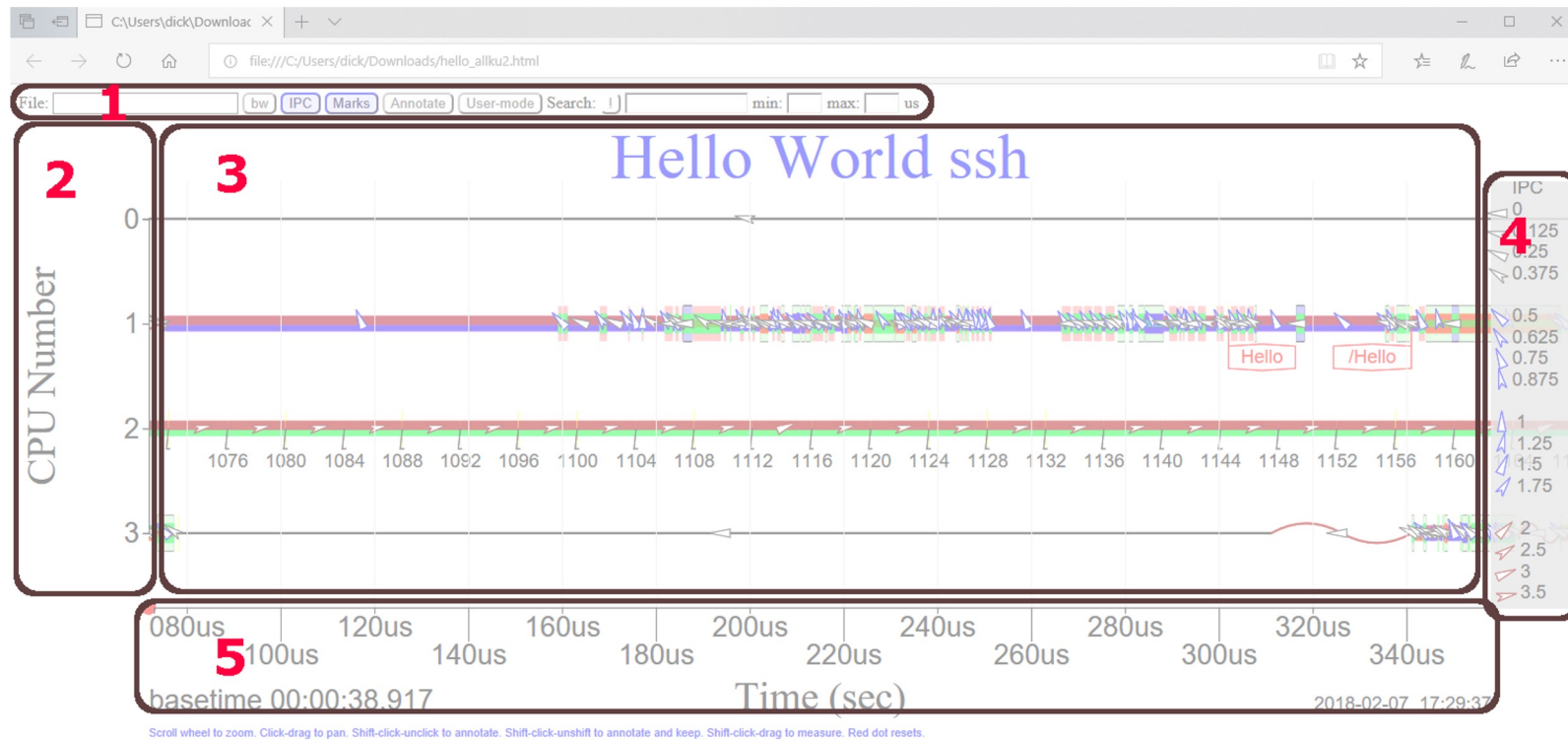
```
{
  "Comment" : "V2 with IPC field",
  "axisLabelX" : "Time (sec)",
  "axisLabelY" : "CPU Number",
  "flags" : 128,
  "shortMulX" : 1,
  "shortUnitsX" : "s",
  "thousandsX" : 1000,
  "title" : "Hello World",
  "tracebase" : "2018-01-18_22:54:21",
  "version" : 2,
  "events" : [
    [ 23.66655382, 0.00000218, 2, 1995, 0, 2055, 59216, 1, 0, "poll"],
    ...
    [999.0, 0.0, 0, 0, 0, 0, 0, 0, 0, ""]
  ]
}
```


Kernel-user tracing, HTML display

Postprocessing



KUtrace display of timelines



Display issues

Problem:

Possibly 1-10 million execution spans to display

Solutions:

- spantospan reduces granularity, so fewer spans
- spantotrim keeps only a subset time interval, so fewer spans
- HTML initially shows entire trace at modest resolution, allows user to pan and zoom to reveal detail at various different places.
- HTML *elides* spans that are narrower than 1 pixel, combining several until they total at least one pixel. The total is then displayed with a single-color line, not three or four different colors -- just idle or kernel or user.

Display issues

Problem:

Showing different processes or syscalls in different colors is too hard to distinguish -- only about 12-20 distinct colors on screen

Solutions:

- Use pairs of colors, PID mod 15 to pick one, PID mod 17 to pick the other, giving $15 \times 17 = 255$ combinations total. Display as two stacked vertical lines.
- For kernel-mode, add third lightened background color to distinguish syscall, interrupt, and fault code.

Display issues

Problem:

Display CPU timelines plus PID timelines plus RPC timelines, but now the vertical dimension is too crowded

Solutions:

- Group the spans: CPU, PID, RPC
- Make groups collapsible, and by default collapse PID and RPC.
- Allow Y-axis to pan and zoom
- Suppress lines whose non-idle content is off-screen
- Allow shift-click on Y-axis labels to highlight one or more only
- Allow shift-click on group labels to collapse un-highlighted rows
- Show vertically narrow spans as single-color lines, to speed up display

Display issues

Problem:

Color-blind users

Solutions:

- Try limited gray-scale combinations of three lines per span -- **fails**
- Try color-Brewer color-blind safe colors <http://colorbrewer2.org> -- **fails** with not enough distinct colors (about 7-9).
- Equal-luminance and thus low-contrast syscall/IRQ/fault background colors are particularly problematic. So add high-contrast one-pixel edges -- black, gray, white
- Add CB button that simply toggles RGB channels to GBR

Display issues

Problem:

Too many digits in X-axis time labels

Solutions:

- Give overall base (start) time in HH:MM:SS and then just seconds from there on the X-axis.
- When zoomed in, move millisecond digits to base time and just give milliseconds from there on the X-axis.
- When zoomed in further, move millisecond and microsecond digits to base time and just give microseconds from there on the X-axis.
- Change axis **units** to match

More display issues

easy to get lost -- how to get back? -- red button

hard to get overview -- user button to label all processes, just first instance, short names

hard to click on one span at a time to see detail -- all button to label all spans on screen

want to know details about one span -- long label has start time, full name, syscall(arg)=ret, process id, duration, IPC.

Labels overlap -- stack them vertically in ~3 tracks, use semi-opaque white background

labelling all takes forever to display when 1M spans -- at most one label start per pixel location

same with search results

More display issues

easy to lose place after pan/zoom -- retain nearest label and redisplay it after pan/zoom

want grid, but not intrusive -- almost-white vertical grid lines

straight-line wakeup arrows blend into span lines, not visible if overlapping -- curve them

want to search for unusually long or short spans -- usec min/max range

can't find a particular rarely-executed process -- use search by name or pid number

hard to distinguish IPC triangle angles -- color-code groups of four, notch alternate shapes

More display issues

want to see total time for particular spans -- search result gives count and total time

want labels to show which row they label -- put just underneath that row

want to line up labelled items across different rows -- extend solid line below label row, dotted line above

want to determine time between events -- click-drag snaps to event row and start time, giving elapsed time

More display issues

clutter from too many identical PID labels -- only label user-mode once at first time from left edge

clutter from too many marks -- suppress if too close together, button to suppress them all

clutter from IPC triangles -- suppress if too close together or X width is too small, button to suppress them all, off by default

too much clutter with wakeup arcs -- suppress short ones if X width is too small, button to suppress them all

More display issues

in a presentation, want to quickly show specific subsets of a trace -- 1..5 buttons to save/restore pan, zoom, and buttons

fonts too small for presentations -- click title to change font sizes

explain UI without taking much space -- single Usage() line at bottom, with [more} to expand to a screenful of detail

Videos!

- KUtrace author Dick Sites has a youtube channel.
<https://www.youtube.com/@dicksites>
- [What is KUtrace? https://www.youtube.com/watch?v=OiirqAtqXI8](https://www.youtube.com/watch?v=OiirqAtqXI8)
- Walk through of the hello world example
<https://youtu.be/qa3DtEzHvBs?si=f33cXG22SgzqFvAN>
- Linked from the materials tab on the department P56 page
<https://www.cl.cam.ac.uk/teaching/2324/P56/materials.html>

