# Kernel trace subsystem and tracepoints

August, 2017, Beijing

**Joey Lee**
SUSE Labs Taipei

# Agenda

- Kernel history
- Tracing technologies
  - Kprobe
  - Function tracer/Dynamic ftrace
  - TracePoints
  - Perf event
  - Kprobe event
- Conceptual models
- Tracepoints
- Q&A

# Kernel history

# Kernel history (2005)

- v2.6.12-rc2
  - Kprobes
    - 1da177e4c Linux-2.6.12-rc2 Apr 16, 2005
- V2.6.13
  - Kprobes/IA64
    - fd7b231ff Kprobes/IA64: arch specific handling Jun 23, 2005

# Kernel history (2008)

- v2.6.27-rc1
  - Function tracer
    - 1b29b0188 ftrace: function tracer May 12, 2008
  - Dynamic ftrace
    - 3d0833953 ftrace: dynamic enabling/disabling of function calls May 12, 2008

# Kernel history (2008)

- v2.6.28-rc1
  - Tracepoints
    - 97e1c18e tracing: Kernel Tracepoints Jul 18, 2008
- v2.6.28-rc3
  - __mcount_loc section
    - 8da3821b ftrace: create __mcount_loc section Aug 14, 2008
  - Rename FTRACE to FUNCTION_TRACER
    - 606576ce8 ftrace: rename FTRACE to FUNCTION_TRACER Oct 6, 2008
- v2.6.31-rc1
  - Performance Counters
    - 0793a61d4 performance counters: core code Dec 4, 2008

# Kernel history (2009)

- v2.6.30-rc1
  - Event trace infrastructure
    - b77e38aa2 tracing: add event trace infrastructure Feb 24, 2009
  - Trace event
    - da4d03020 tracing: new format for specialized trace points Mar 9, 2009
- v2.6.32-rc1
  - Performance Counters -> Performance Events
    - Cdd6c482c perf: Do the big rename: Performance Counters -> Performance Events Sep 21, 2009
- v2.6.33
  - Kprobe events
    - 413d37d1e tracing: Add kprobe-based event tracer Aug 13, 2009

# Kernel history (2012)

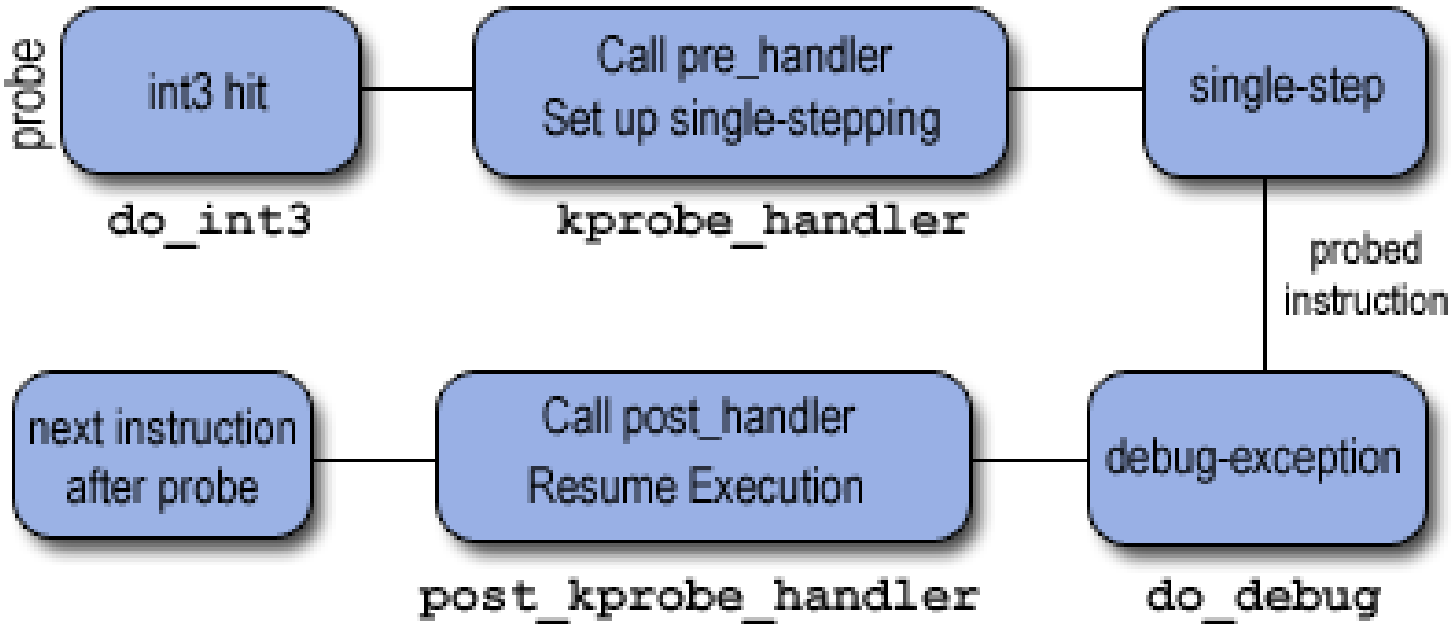
- v3.7-rc1
  - Kprobe on ftrace
    - ae6aa16fd kprobes: introduce ftrace based optimization
      Jun 5 2012

# Tracing technologies

# Kprobe

- v2.6.11.7 (2005)
- KProbes was developed by IBM as an underlying mechanism for another higher level tracing tool called Dprobes.
- KProbes is available on the following architectures however: ppc64, x86_64, sparc64 and i386.
- After the probes are registered, the addresses at which they are active contain the breakpoint instruction (int3 on x86).
- do_int3() is called through an interrupt gate therefore interrupts are disabled when control reaches there. [4]

# Kprobe (cont.)




**Execution of a KProbe**

Sudhanshu Goswami[4]

# Function tracer

- v2.6.27-rc1 (2008)
- Kernel Function Tracer (The old name is ftrace)
- CONFIG_FUNCTION_TRACER
  - Enable the kernel to trace every kernel function. This is done by using a compiler feature to insert a small, 5-byte No-Operation instruction to the beginning of every kernel function, which NOP sequence is then dynamically patched into a tracer call when tracing is enabled by the administrator. If it's runtime disabled (the bootup default), then the overhead of the instructions is very small and not measurable even in micro-benchmarks

# Ftrace and function tracer

- Ftrace is an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel.
- Ftrace is typically considered the function tracer, it is really a frame work of several assorted tracing utilities. [5]
  - latency tracing: interrupts disabled and enabled, preemption, from a time a task is woken to the task is actually scheduled in.
  - event tracing.

# Dynamic ftrace

- v2.6.27-rc1 (2008)
- If CONFIG_DYNAMIC_FTRACE is set, the system will run with virtually no overhead when function tracing is disabled.
- The way this works is the mcount function call (placed at the start of every kernel function, produced by the -pg switch in gcc), starts of pointing to a simple return. (Enabling FTRACE will include the -pg switch in the compiling of the kernel.) [5]

# __mcount_loc section

- v2.6.28-rc3 (2008)
- At compile time every C file object is run through the recordmcount program (located in the scripts directory).
- This program will parse the ELF headers in the C object to find all the locations in the .text section that call mcount.
- A new section called "__mcount_loc" is created that holds references to all the mcount call sites in the .text section.
- The recordmcount program re-links this section back into the original object. The final linking stage of the kernel will add all these references into a single table. [5]

# Dynamic ftrace (cont.)

- On boot up, before SMP is initialized, the dynamic ftrace code scans this table and updates all the locations into nops.
- It also records the locations, which are added to the available_filter_functions list. Modules are processed as they are loaded and before they are executed.
- When tracing is enabled, the process of modifying the function tracepoints is dependent on architecture. [5]
  - Old: kstop_machine
  - New: breakpoint

# Tracepoints

- v2.6.28-rc1 (2008)
- A tracepoint placed in code provides a hook to call a function (probe) that you can provide at runtime.
- Will explain later

# Performance counters to Perf event

- Performance counters
  - V2.6.31-rc1 (2008)
- Perference event
  - V2.6.32-rc1 (2009)
- The Linux Performance Counter subsystem provides an abstraction of performance counter hardware capabilities. It provides per task and per CPU counters, and it provides event capabilities on top of those.
- Performance Monitoring Unit (PMU)

# Kprobe event

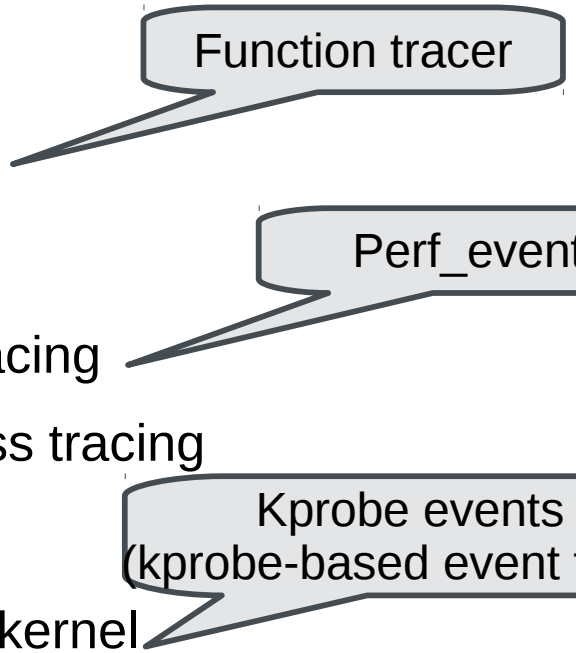
- v2.6.33 (2009)
- CONFIG_KPROBE_TRACER
  - 413d37d1eb
  - This tracer is similar to the events tracer which is based on Tracepoin infrastructure. Instead of Tracepoint, this tracer is based on kprobes (kprobe and kretprobe). It probes anywhere where kprobes can probe(this means, all functions body except for __kprobes functions).

# What’s dynamic?

- Dynamic probe function
  - Dynamic ftrace
- Dynamic trace event
  - TRACE_EVENT/Funcation Tracer vs. Kprobe

# Tracing Events

- Fixed Events
  - Tracepoints – Static event tracing
  - Mount – Function entry (exit) tracing
- Hardware Events
  - Performace counters – HW event tracing
  - HW Breakpoint – HW memory access tracing
- Dynamic Events
  - Kprobes – Dynamic event tracing in kernel
    - What's dynamic? - trace events in the function body
  - Uprobes – Dynamic event tracing in user space
    - Under development

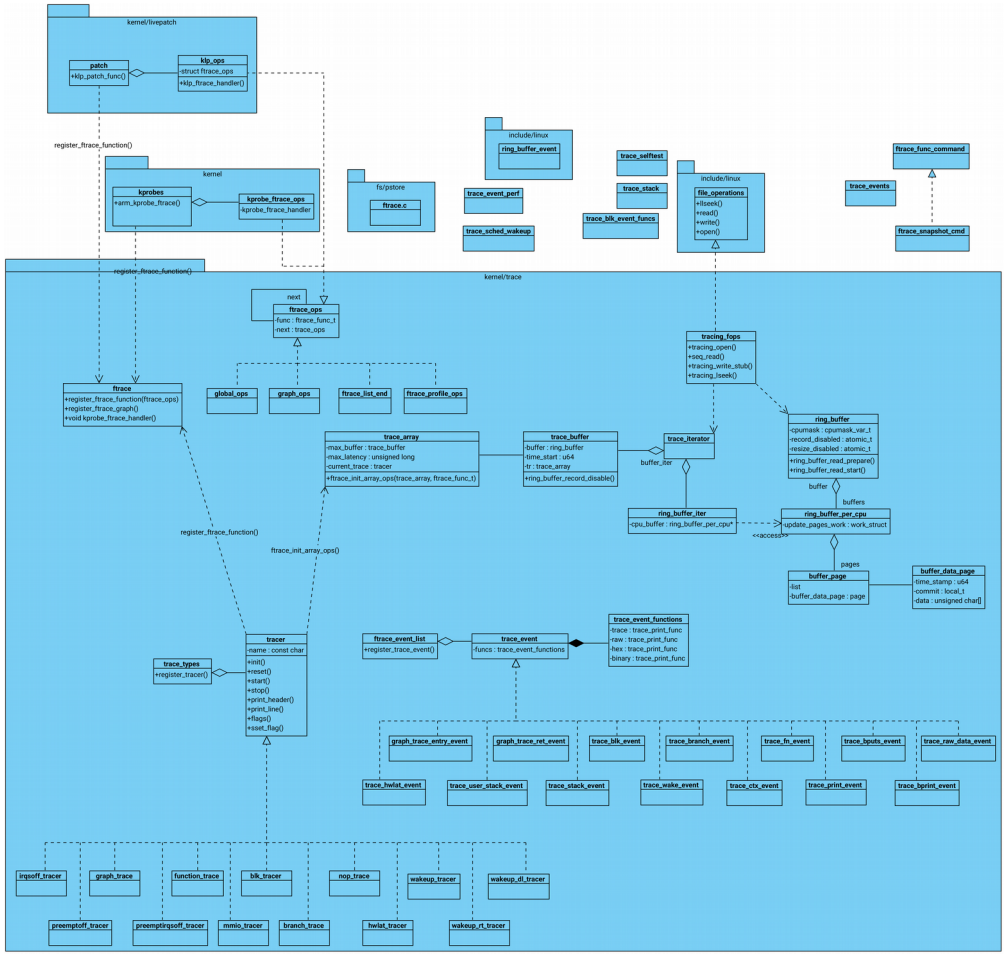


# Lockless Ring Buffer

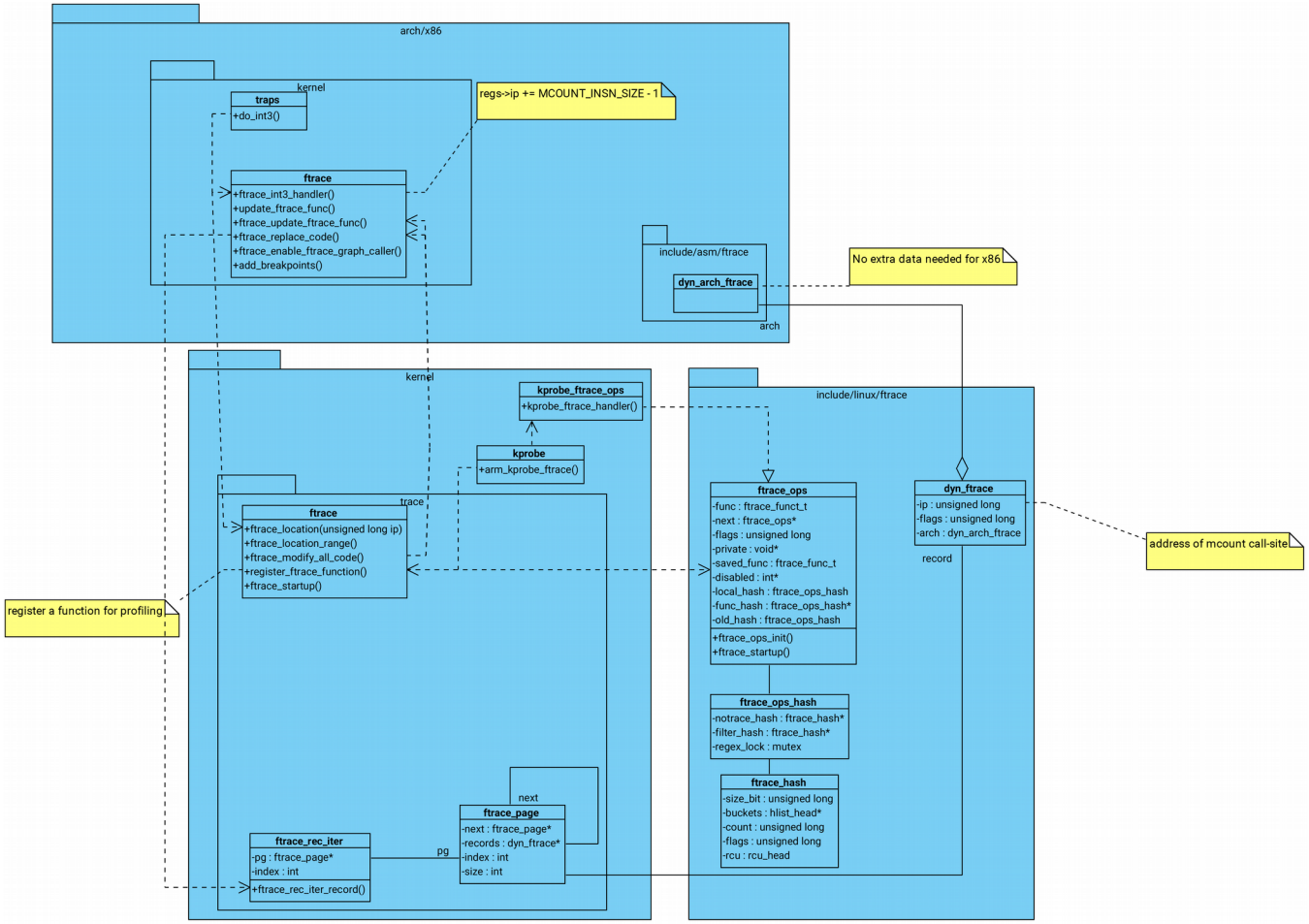
- Documentation/trace/ring-buffer-design.txt, Steven Rostedt, Linux Kernel

# Conceptual models

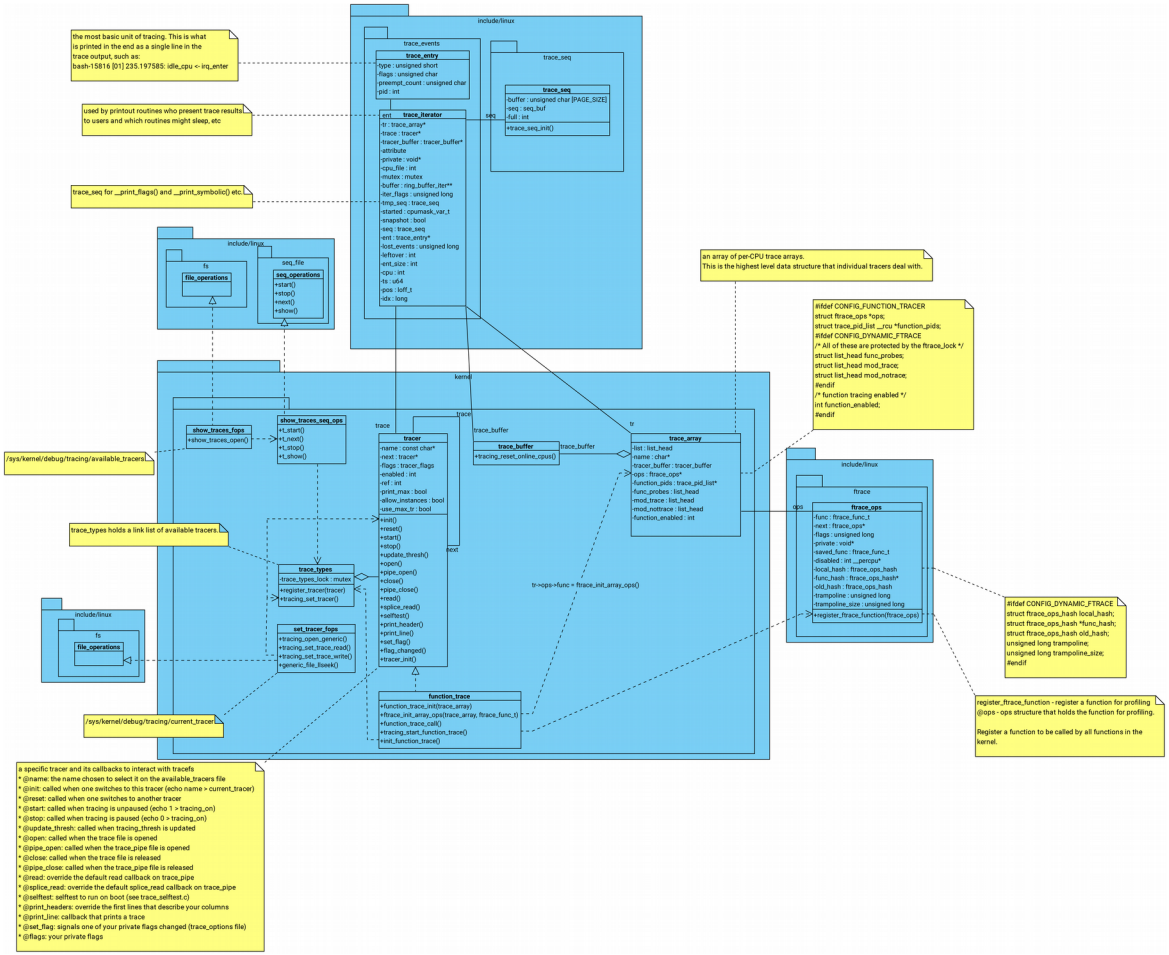
# Trace subsystem



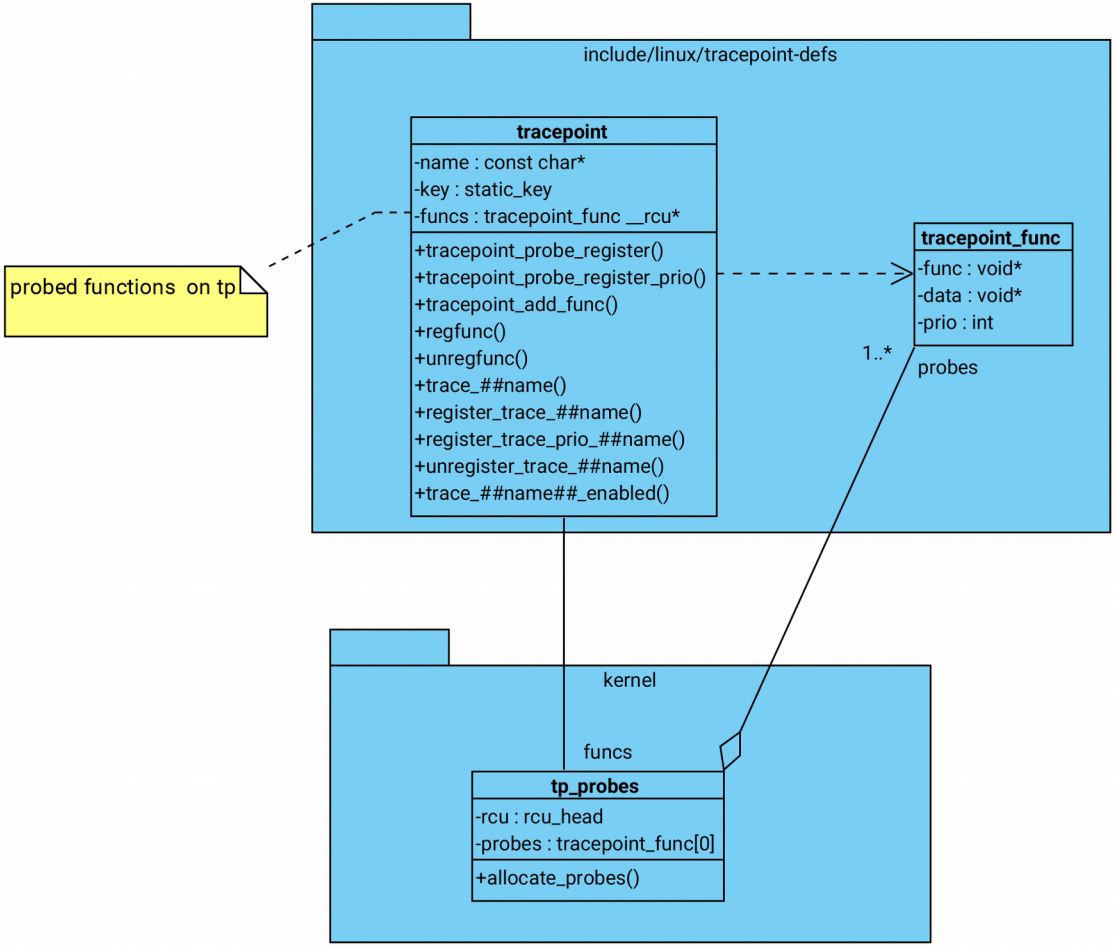
# Ftrace


arch/x86
kernel
traps
+do_int3()
regs->ip += MCOUNT_INSN_SIZE - 1
ftrace
+ftrace_int3_handler()
+update_ftrace_func()
+ftrace_update_ftrace_func()
+ftrace_replace_code()
+ftrace_enable_ftrace_graph_caller()
+add_breakpoints()
include/asm/ftrace
dyn_arch_ftrace
arch
No extra data needed for x86
kernel
kprobe_ftrace_ops
+kprobe_ftrace_handler()
kprobe
+arm_kprobe_ftrace()
trace
ftrace
+ftrace_location(unsigned long ip)
+ftrace_location_range()
+ftrace_modify_all_code()
+register_ftrace_function()
+ftrace_startup()
register a function for profiling
include/linux/ftrace
ftrace_ops
-func : ftrace_funct_t
-next : ftrace_ops*
-flags : unsigned long
-private : void*
-saved_func : ftrace_func_t
-disabled : int*
-local_hash : ftrace_ops_hash
-func_hash : ftrace_ops_hash*
-old_hash : ftrace_ops_hash
+ftrace_ops_init()
+ftrace_startup()
dyn_ftrace
-ip : unsigned long
-flags : unsigned long
-arch : dyn_arch_ftrace
record
address of mcount call-site
ftrace_ops_hash
-notrace_hash : ftrace_hash*
-filter_hash : ftrace_hash*
-regex_lock : mutex
ftrace_hash
-size_bit : unsigned long
-buckets : hlist_head*
-count : unsigned long
-flags : unsigned long
-rcu : rcu_head
next
ftrace_page
-next : ftrace_page*
-records : dyn_ftrace*
-index : int
-size : int
pg
ftrace_rec_iter
-pg : ftrace_page*
-index : int
+ftrace_rec_iter_record()

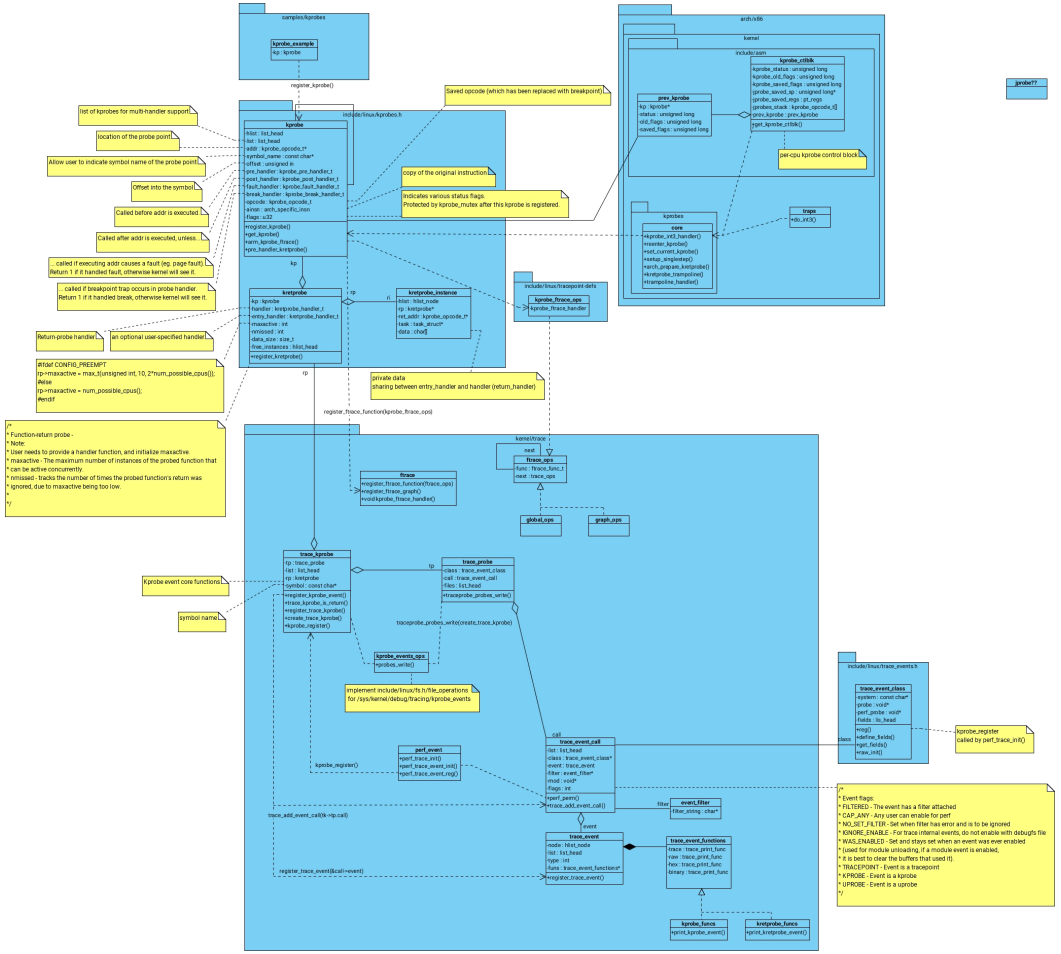
# Function tracer



# Tracepoint


include/linux/tracepoint-defs
tracepoint
-name : const char*
-key : static_key
-funcs : tracepoint_func __rcu*
+tracepoint_probe_register()
+tracepoint_probe_register_prio()
+tracepoint_add_func()
+regfunc()
+unregfunc()
+trace_##name()
+register_trace_##name()
+register_trace_prio_##name()
+unregister_trace_##name()
+trace_##name##_enabled()
tracepoint_func
-func : void*
-data : void*
-prio : int
probed functions on tp
1..*
probes
kernel
funcs
tp_probes
-rcu : rcu_head
-probes : tracepoint_func[0]
+allocate_probes()

# Kprobe


samples/kprobes
kprobe_example
register_kprobe()
include/linux/kprobes.h
kprobe
kretprobe
kretprobe_instance
arch/x86
kernel
kprobes
core
include/linux/tracepoint-defs
kprobe_ftrace_ops
kernel/trace
ftrace
ftrace_ops
global_ops
graph_ops
trace_kprobe
trace_probe
kprobe_events_ops
include/linux/trace_events.h
trace_event_class
trace_event_call
trace_event
perf_event
event_filter
trace_event_functions
kprobe_func
kretprobe_func
list of kprobes for multi-handler support
location of the probe point
Allow user to indicate symbol name of the probe point
Offset into the symbol
Called before addr is executed.
Called after addr is executed, unless...
... called if executing addr causes a fault (eg. page fault).
Return 1 if it handled fault, otherwise kernel will see it.
... called if breakpoint trap occurs in probe handler.
Return 1 if it handled break, otherwise kernel will see it.
Return-probe handler
an optional user-specified handler
#ifdef CONFIG_PREEMPT
#else
#endif
saved opcode (which has been replaced with breakpoint)
copy of the original instruction
Indicates various status flags.
Protected by kprobe_mutex after this kprobe is registered.
per-cpu kprobe control block
traps
private data
sharing between entry_handler and handler (return_handler)
register_ftrace_function(kprobe_ftrace_ops)
Kprobe event core functions
symbol name
implement include/linux/fs.h/file_operations
for /sys/kernel/debug/tracing/kprobe_events
kprobe_register
called by perf_trace_init()
trace_add_event_call(&.->tp.call)
register_trace_event(&call->event)
traceprobe_probes_write(create_trace_kprobe)

# Tracepoints

# Purpose of tracepoints

- A tracepoint placed in code provides a hook to call a function (probe) that you can provide at runtime.
- A tracepoint can be "on" (a probe is connected to it) or "off" (no probe is attached).
- When a tracepoint is "on", the function you provide is called each time the tracepoint is executed, in the execution context of the caller. [1]

# Purpose of tracepoints (cont.)

- Tracepoints can be used without creating custom kernel modules to register probe functions using the event tracing infrastructure.
- Simplistically, tracepoints represent important events that can be taken in conjunction with other tracepoints to build a "Big Picture" of what is going on within the system. [2]

# Register a probe

- Connecting a function (probe) to a tracepoint is done by providing a probe (function to call) for the specific tracepoint through register_trace_subsys_eventname(). [1]

# include/linux/tracepoint.h

```
/*
 * Make sure the alignment of the structure in the __tracepoints section will
 * not add unwanted padding between the beginning of the section and the
 * structure. Force alignment to the same alignment as the section start.
 *
 * When lockdep is enabled, we make sure to always do the RCU portions of
 * the tracepoint code, regardless of whether tracing is on. However,
 * don't check if the condition is false, due to interaction with idle
 * instrumentation. This lets us find RCU issues triggered with tracepoints
 * even when this tracepoint is off. This code has no purpose other than
 * poking RCU a bit.
 */
#define __DECLARE_TRACE(name, proto, args, cond, data_proto, data_args) \
        extern struct tracepoint __tracepoint_##name;                   \
        static inline void trace_##name(proto)                          \
        {                                                               \
                if (static_key_false(&__tracepoint_##name.key))         \
                        __DO_TRACE(&__tracepoint_##name,                \
                                TP_PROTO(data_proto),                   \
                                TP_ARGS(data_args),                     \
                                TP_CONDITION(cond), 0);                 \
                if (IS_ENABLED(CONFIG_LOCKDEP) && (cond)) {             \
                        rcu_read_lock_sched_notrace();                  \
                        rcu_dereference_sched(__tracepoint_##name.funcs);\
                        rcu_read_unlock_sched_notrace();                \
                }                                                       \
        }                                                               \
        __DECLARE_TRACE_RCU(name, PARAMS(proto), PARAMS(args),          \
                PARAMS(cond), PARAMS(data_proto), PARAMS(data_args))    \
        static inline int                                               \
        register_trace_##name(void (*probe)(data_proto), void *data)    \
        {                                                               \
                return tracepoint_probe_register(&__tracepoint_##name,  \
                                                (void *)probe, data);   \
        }                                                               \
```

# Registering probe function

```
kernel/trace/trace_syscalls.c   static int reg_event_syscall_enter(struct trace_event_file *file,
                                    ret = register_trace_sys_enter(ftrace_syscall_enter, tr);
        include/linux/tracepoint.h      register_trace_##name(void (*probe)(data_proto), void *data)    \
                kernel/tracepoint.c     int tracepoint_probe_register(struct tracepoint *tp, void *probe, void *data)
                        kernel/tracepoint.c     int tracepoint_probe_register_prio(struct tracepoint *tp, void *probe,
                                kernel/tracepoint.c     static int tracepoint_add_func(struct tracepoint *tp,
                                                                struct tracepoint_func *func, int prio)
```

# kernel/tracepoint.c

```
/*
 * Add the probe function to a tracepoint.
 */
static int tracepoint_add_func(struct tracepoint *tp,
                               struct tracepoint_func *func, int prio)
[...snip]
        /*
         * rcu_assign_pointer has a smp_wmb() which makes sure that the new
         * probe callbacks array is consistent before setting a pointer to it.
         * This array is referenced by __DO_TRACE from
         * include/linux/tracepoints.h. A matching smp_read_barrier_depends()
         * is used.
         */
        rcu_assign_pointer(tp->funcs, tp_funcs);
        if (!static_key_enabled(&tp->key))
                static_key_slow_inc(&tp->key);
        release_probes(old);
        return 0;
}
```

# static_key

- The advantage of using the trace_<tracepoint>_enabled() is that it uses the static_key of the tracepoint to allow the if statement to be implemented with jump labels and avoid conditional branches. [1]

```
struct tracepoint {
        const char *name;                       /* Tracepoint name */
        struct static_key key;
        int (*regfunc)(void);
        void (*unregfunc)(void);
        struct tracepoint_func __rcu *funcs;
};
```

# include/linux/tracepoint.h

```
#define __DECLARE_TRACE(name, proto, args, cond, data_proto, data_args) \
        extern struct tracepoint __tracepoint_##name;                   \
        static inline void trace_##name(proto)                          \
        {                                                               \
                if (static_key_false(&__tracepoint_##name.key))         \
                        __DO_TRACE(&__tracepoint_##name,                \
                                TP_PROTO(data_proto),                   \
                                TP_ARGS(data_args),                     \
                                TP_CONDITION(cond), 0);                 \
```

# include/linux/jump_label.h

```
#ifdef HAVE_JUMP_LABEL
...
static __always_inline bool static_key_false(struct static_key *key)
{
        return arch_static_branch(key, false);
}
...
#else  /* !HAVE_JUMP_LABEL */

static __always_inline bool static_key_false(struct static_key *key)
{
        if (unlikely(static_key_count(key) > 0))
                return true;
        return false;
}
```

# Unregister a probe

- Removing a probe is done through unregister_trace_subsys_eventname(); it will remove the probe.
- tracepoint_synchronize_unregister() must be called before the end of the module exit function to make sure there is no caller left using the probe. This, and the fact that preemption is disabled around the probe call, make sure that probe removal and module unload are safe.
  - Didn’t find preempt_disable() around unregister code, but saw rcu code.

# __DO_TRACE()

```
#define __DO_TRACE(tp, proto, args, cond, rcucheck)                     \
        do {                                                            \
                struct tracepoint_func *it_func_ptr;                    \
                void *it_func;                                          \
                void *__data;                                           \
                                                                        \
                if (!(cond))                                            \
                        return;                                         \
                if (rcucheck) {                                         \
                        if (WARN_ON_ONCE(rcu_irq_enter_disabled()))     \
                                return;                                 \
                        rcu_irq_enter_irqson();                         \
                }                                                       \
                rcu_read_lock_sched_notrace();                          \
                it_func_ptr = rcu_dereference_sched((tp)->funcs);       \
                if (it_func_ptr) {                                      \
                        do {                                            \
                                it_func = (it_func_ptr)->func;          \
                                __data = (it_func_ptr)->data;           \
                                ((void(*)(proto))(it_func))(args);      \
                        } while ((++it_func_ptr)->func);                \
                }                                                       \
                rcu_read_unlock_sched_notrace();                        \
                if (rcucheck)                                           \
                        rcu_irq_exit_irqson();                          \
        } while (0)
```

# Q&A

# Reference


- [1] Documentation/trace/tracepoints.txt, Mathieu Desnoyers, Linux Kernel
- [2] Documentation/trace/tracepoint-analysis.txt, Mel Gorman, Linux Kernel
- [3] Dynamic Event Tracing in Linux Kernel, Masami Hiramatsu, 4th Linux Foundation Collaboration Summit
- [4] An introduction to Kprobes, Sudhanshu Goswami, LWN.net, April 18, 2005
- [5] Documentation/trace/ftrace.txt, Steven Rostedt, Linux Kernel
- [6] Documentation/trace/ring-buffer-design.txt, Steven Rostedt, Linux Kernel

Feedback to
jlee@suse.com

Thank you.


SUSE®
We adapt. You succeed.


**Corporate Headquarters**
Maxfeldstrasse 5
90409 Nuremberg
Germany

+49 911 740 53 0 (Worldwide)
www.suse.com

Join us on:
www.opensuse.org