Tracepoints

Alice Ryhl

What are tracepoints?

```
TRACE_EVENT(binder_ioctl,
    TP_PROTO(unsigned int cmd, unsigned long arg),
    TP_ARGS(cmd, arg),

    TP_STRUCT__entry(
        __field(unsigned int, cmd)
        __field(unsigned long, arg)
    ),
    TP_fast_assign(
        __entry->cmd = cmd;
        __entry->arg = arg;
    ),
    TP_printk("cmd=0x%x arg=0x%lx", __entry->cmd, __entry->arg)
);
```

What are tracepoints?

```
# echo 1 > /sys/kernel/tracing/events/binder/binder_ioctl/enable
# echo 1 > /sys/kernel/tracing/tracing_on
# cat /sys/kernel/tracing/trace
   binder:2165_5-2602    [001] .....   330.711899: binder_ioctl: cmd=0xc0306201 arg=0x7500399538
     lowpool[14]-5276    [001] .....   330.711965: binder_ioctl: cmd=0xc0306201 arg=0x759d699ac8
     lowpool[14]-5276    [001] .....   330.713158: binder_ioctl: cmd=0xc0306201 arg=0x759d6998d0
  binder:1240_19-3743    [001] .....   330.713441: binder_ioctl: cmd=0xc0306201 arg=0x748c029290
  binder:1240_19-3743    [001] .....   330.713548: binder_ioctl: cmd=0xc0306201 arg=0x748c029538
     lowpool[14]-5276    [001] .....   330.713631: binder_ioctl: cmd=0xc0306201 arg=0x759d6999d8
     lowpool[14]-5276    [001] .....   330.737330: binder_ioctl: cmd=0xc0306201 arg=0x759d699af0
   binder:2844_4-4437    [000] .....   330.737614: binder_ioctl: cmd=0xc0306201 arg=0x749b6a2538
 hbox:interactor-2844    [002] .....   331.708449: binder_ioctl: cmd=0xc0306201 arg=0x7fd5d952c0
  binder:1240_19-3743    [001] .....   331.710181: binder_ioctl: cmd=0xc0306201 arg=0x748c029538
```

What are tracepoints?

```
static long
binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int ret;
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread;
    void __user *ubuf = (void __user *)arg;

    /*pr_info("binder_ioctl: %d:%d %x %lx\n",
            proc->pid, current->pid, cmd, arg);*/

    binder_selftest_alloc(&proc->alloc);

    trace_binder_ioctl(cmd, arg);
```

What are tracepoints?

- Tracepoints are usually turned off.
- Tracepoints rely on the *static key* optimization.
- Machine code is modified at runtime to switch between `nop` and `jmp` instruction.
- Static key implemented with inline asm.

Tracepoints in Rust

An important part of a production-ready Linux kernel driver is tracepoints.

To write production-ready Linux kernel drivers in Rust, we must be able to call tracepoints from Rust code.

include/trace/events/rust_sample.h

```
/* SPDX-License-Identifier: GPL-2.0-only */

#undef TRACE_SYSTEM
#define TRACE_SYSTEM rust_sample

#if !defined(_RUST_SAMPLE_TRACE_H) || defined(TRACE_HEADER_MULTI_READ)
#define _RUST_SAMPLE_TRACE_H

#include <linux/tracepoint.h>

TRACE_EVENT(rust_sample_loaded,
        TP_PROTO(int magic_number),
        TP_ARGS(magic_number),
        TP_STRUCT__entry(
                __field(int, magic_number)
        ),
        TP_fast_assign(
                __entry->magic_number = magic_number;
        ),
        TP_printk("magic=%d", __entry->magic_number)
);

#endif /* _RUST_SAMPLE_TRACE_H */

/* This part must be outside protection */
#include <trace/define_trace.h>
```

samples/rust/rust_print.rs

```
kernel::declare_trace! {
    unsafe fn rust_sample_loaded(magic: c_int);
}
```

samples/rust/rust_print.rs

```
kernel::declare_trace! {
    unsafe fn rust_sample_loaded(magic: c_int);
}

#[inline]
pub(crate) fn trace_rust_sample_loaded(magic: i32) {
    // SAFETY: Always safe to call.
    unsafe { rust_sample_loaded(magic as c_int) }
}
```

samples/rust/rust_print.rs

```
kernel::declare_trace! {
    unsafe fn rust_sample_loaded(magic: c_int);
}

#[inline]
pub(crate) fn trace_rust_sample_loaded(magic: i32) {
    // SAFETY: Always safe to call.
    unsafe { rust_sample_loaded(magic as c_int) }
}

trace_rust_sample_loaded(42);
```

samples/rust/rust_print_events.c

```
#define CREATE_TRACE_POINTS
#define CREATE_RUST_TRACE_POINTS
#include <trace/events/rust_sample.h>
```

samples/rust/rust_print_events.c

```
#define CREATE_TRACE_POINTS
#define CREATE_RUST_TRACE_POINTS
#include <trace/events/rust_sample.h>
```

samples/rust/Makefile

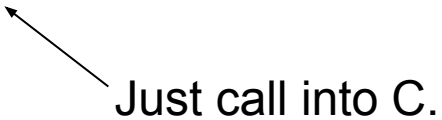
```
obj-$(CONFIG_SAMPLE_RUST_PRINT)  += rust_print.o \
                                    rust_print_events.o
```

Implementation

- Consists of two parts:
  - Check whether the tracepoint is enabled.
  - If it is, actually call the tracepoint.

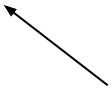
Implementation

- Consists of two parts:
  - Check whether the tracepoint is enabled.
  - If it is, actually call the tracepoint.

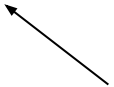


Implementation

- Consists of two parts:
  - Check whether the tracepoint is enabled.
  - If it is, actually call the tracepoint.



Fun stuff happens here.



Just call into C.

Static key in C

```
asm goto(
    "1: nop                                     \n\t"
    ".pushsection    __jump_table, \"aw\"       \n\t"
    ".align    3                                \n\t"
    ".long     1b - ., %l[l_yes] - .            \n\t"
    ".quad     %c0 - .                          \n\t"
    ".popsection                                \n\t"
    :  :  "i"(k) :  : l_yes
);

return false;

l_yes:
return true;
```

Static key in C

```
asm goto(
    "1: jmp l_yes                                   \n\t"
    ".pushsection    __jump_table, \"aw\"           \n\t"
    ".align    3                                    \n\t"
    ".long     1b - ., %l[l_yes] - .                \n\t"
    ".quad     %c0 - .                              \n\t"
    ".popsection                                    \n\t"
    :  :  "i"(k) :  : l_yes
);

return false;

l_yes:
return true;
```

Static key in Rust (first try)

```
#[cfg(target_arch = "aarch64")]
macro_rules! _static_key_false {
    ($key:path, $keytyp:ty, $field:ident) => {'my_label: {
        core::arch::asm!(
            r#"
            1: nop

            .pushsection __jump_table,  "aw"
            .align 3
            .long 1b - ., {0} - .
            .quad {1} + {2} - .
            .popsection
            "#,
            label {
                break 'my_label true;
            },
            sym $key,
            const offset_of!($keytyp, $field),
        );

        break 'my_label false;
    }};
}
```

Do exactly the same as C. The inline asm is copied from the C implementation.

Adjust all of the jump_label.h headers

```
--- a/arch/arm64/include/asm/jump_label.h
+++ b/arch/arm64/include/asm/jump_label.h
@@ -19,10 +19,14 @@
 #define JUMP_TABLE_ENTRY(key, label)                    \
        ".pushsection   __jump_table, \"aw\"\n\t"        \
        ".align         3\n\t"                           \
-       ".long          1b - ., %l["#label"] - .\n\t"    \
-       ".quad          %c0 - .\n\t"                     \
-       ".popsection\n\t"                                \
-       :  :  "i"(key) :  : label
+       ".long          1b - ., " label " - .\n\t"       \
+       ".quad          " key " - .\n\t"         \
+       ".popsection\n\t"
+
+/* This macro is also expanded on the Rust side. */
+#define ARCH_STATIC_BRANCH_ASM(key, label)              \
+       "1:     nop\n\t"                         \
+       JUMP_TABLE_ENTRY(key, label)

 static __always_inline bool arch_static_branch(struct static_key * const key,
                                                const bool branch)
@@ -30,8 +34,8 @@ static __always_inline bool arch_static_branch(struct static_key * const key,
        char *k = &((char *)key)[branch];

        asm goto(
-               "1:     nop                                     \n\t"
-               JUMP_TABLE_ENTRY(k, l_yes)
+               ARCH_STATIC_BRANCH_ASM("%c0", "%l[l_yes]")
+               :  :  "i"(k) :  : l_yes
                );
```

Rust asm invocation

```
macro_rules! arch_static_branch {
    ($key:path, $keytyp:ty, $field:ident, $branch:expr) => {'my_label: {
        $crate::asm!(
            include!(concat!(env!("OBJTREE"), /rust/kernel/arch_static_branch_asm.rs"));
            l_yes = label {
                break 'my_label true;
            },
            symb = sym $key,
            off = const offset_of!($keytyp, $field),
            branch = const $branch,
        );

        break 'my_label false;
    }};
}
```

Generated file. (Run C preprocessor on Rust code.)

rust/kernel/arch_static_branch_asm.rs.S

```
// SPDX-License-Identifier: GPL-2.0

#include <linux/jump_label.h>

// Cut here.

::kernel::concat_literals!(ARCH_STATIC_BRANCH_ASM("{symb} + {off} + {branch}", "{l_yes}"))
```

rust/kernel/arch_static_branch_asm.rs.S

```
// SPDX-License-Identifier: GPL-2.0

#include <linux/jump_label.h>

// Cut here.

::kernel::concat_literals!(ARCH_STATIC_BRANCH_ASM("{symb} + {off} + {branch}", "{l_yes}"))
```

Expands to a bunch of C code, but defines the macro we need.

Expands to the inline asm we want.

scripts/Makefile.build

```
quiet_cmd_rustc_rs_rs_S = RSCPP $(quiet_modtag) $@
      cmd_rustc_rs_rs_S = $(CPP) $(c_flags) -xc -C -P $< | sed '1,/^\/\/ Cut here.$$/d' >$@

$(obj)/%.rs: $(obj)/%.rs.S FORCE
     +$(call if_changed_dep,rustc_rs_rs_S)
```


Delete everything before
// Cut here.

Summary

- Deduplicate inline asm between C and Rust.
- Update C headers to define macros.
- Run C preprocessor on Rust code to import asm from C.

Future possibilities

- Avoid repeating the tracepoint signature in both C and Rust.
- Use the same strategy for atomics.
- Other potential ways to deduplicate:
  - Use a more complex script to directly extract asm from C function.
  - Generate both C and Rust from shared format.
  - Teach rustc to reader .h files.

Alternate approach?

Look for these comments after running CPP.

```
// arch_static_branch asm begin
asm goto(
     "1: nop                                    \n\t"
     ".pushsection __jump_table, \"aw\"         \n\t"
     ".align    3                               \n\t"
     ".long     1b - ., %l[l_yes] - .           \n\t"
     ".quad     %c0 - .                         \n\t"
     ".popsection                               \n\t"
     :  :  "i"(k) :  : l_yes
);
// arch_static_branch asm end

return false;

l_yes:
return true;
```