Entry/exit handling for exceptions, interrupts, syscalls and KVM

All transitions between execution domains require state updates which are subject to strict ordering constraints. State updates are required for the following:

- Lockdep
- RCU / Context tracking
- Preemption counter
- Tracing
- Time accounting

The update order depends on the transition type and is explained below in the transition type sections: Syscalls, KVM, Interrupts and regular exceptions, NMI and NMI-like exceptions.

Non-instrumentable code - noinstr

Most instrumentation facilities depend on RCU, so intrumentation is prohibited for entry code before RCU starts watching and exit code after RCU stops watching. In addition, many architectures must save and restore register state, which means that (for example) a breakpoint in the breakpoint entry code would overwrite the debug registers of the initial breakpoint.

Such code must be marked with the 'noinstr' attribute, placing that code into a special section inaccessible to instrumentation and debug facilities. Some functions are partially instrumentable, which is handled by marking them noinstr and using instrumentation_begin() and instrumentation_end() to flag the instrumentable ranges of code:

```
noinstr void entry(void)
{
      handle_entry();     // <-- must be 'noinstr' or '__always_inline'
      ...

      instrumentation_begin();
      handle_context();   // <-- instrumentable code
      instrumentation_end();

      ...
      handle_exit();      // <-- must be 'noinstr' or '__always_inline'
}
```

This allows verification of the 'noinstr' restrictions via objtool on supported architectures.

Invoking non-instrumentable functions from instrumentable context has no restrictions and is useful to protect e.g. state switching which would cause malfunction if instrumented.

All non-instrumentable entry/exit code sections before and after the RCU state transitions must run with interrupts disabled.

Syscalls

Syscall-entry code starts in assembly code and calls out into low-level C code after establishing low-level architecture-specific state and stack frames. This low-level C code must not be instrumented. A typical syscall handling function invoked from low-level assembly code looks like this:

```
noinstr void syscall(struct pt_regs *regs, int nr)
{
      arch_syscall_enter(regs);
      nr = syscall_enter_from_user_mode(regs, nr);

      instrumentation_begin();
      if (!invoke_syscall(regs, nr) && nr != -1)
              result_reg(regs) = __sys_ni_syscall(regs);
      instrumentation_end();

      syscall_exit_to_user_mode(regs);
}
```

syscall_enter_from_user_mode() first invokes enter_from_user_mode() which establishes state in the following order:

- Lockdep
- RCU / Context tracking
- Tracing

and then invokes the various entry work functions like ptrace, seccomp, audit, syscall tracing, etc. After all that is done, the instrumentable invoke_syscall function can be invoked. The instrumentable code section then ends, after which syscall_exit_to_user_mode() is invoked.

syscall_exit_to_user_mode() handles all work which needs to be done before returning to user space like tracing, audit, signals, task work etc. After that it invokes exit_to_user_mode() which again handles the state transition in the reverse order:

- Tracing
- RCU / Context tracking
- Lockdep

syscall_enter_from_user_mode() and syscall_exit_to_user_mode() are also available as fine grained subfunctions in cases where the architecture code has to do extra work between the various steps. In such cases it has to ensure that enter_from_user_mode() is called first on entry and exit_to_user_mode() is called last on exit.

Do not nest syscalls. Nested systcalls will cause RCU and/or context tracking to print a warning.

KVM

Entering or exiting guest mode is very similar to syscalls. From the host kernel point of view the CPU goes off into user space when entering the guest and returns to the kernel on exit.

kvm_guest_enter_irqoff() is a KVM-specific variant of exit_to_user_mode() and kvm_guest_exit_irqoff() is the KVM variant of enter_from_user_mode(). The state operations have the same ordering.

Task work handling is done separately for guest at the boundary of the vcpu_run() loop via xfer_to_guest_mode_handle_work() which is a subset of the work handled on return to user space.

Do not nest KVM entry/exit transitions because doing so is nonsensical.

Interrupts and regular exceptions

Interrupts entry and exit handling is slightly more complex than syscalls and KVM transitions.

If an interrupt is raised while the CPU executes in user space, the entry and exit handling is exactly the same as for syscalls.

If the interrupt is raised while the CPU executes in kernel space the entry and exit handling is slightly different. RCU state is only updated when the interrupt is raised in the context of the CPU's idle task. Otherwise, RCU will already be watching. Lockdep and tracing have to be updated unconditionally.

irqentry_enter() and irqentry_exit() provide the implementation for this.

The architecture-specific part looks similar to syscall handling:

```
noinstr void interrupt(struct pt_regs *regs, int nr)
{
	arch_interrupt_enter(regs);
	state = irqentry_enter(regs);

	instrumentation_begin();

	irq_enter_rcu();
	invoke_irq_handler(regs, nr);
	irq_exit_rcu();

	instrumentation_end();

	irqentry_exit(regs, state);
}
```

Note that the invocation of the actual interrupt handler is within a irq_enter_rcu() and irq_exit_rcu() pair.

irq_enter_rcu() updates the preemption count which makes in_hardirq() return true, handles NOHZ tick state and interrupt time accounting. This means that up to the point where irq_enter_rcu() is invoked in_hardirq() returns false.

irq_exit_rcu() handles interrupt time accounting, undoes the preemption count update and eventually handles soft interrupts and NOHZ tick state.

In theory, the preemption count could be updated in irqentry_enter(). In practice, deferring this update to irq_enter_rcu() allows the preemption-count code to be traced, while also maintaining symmetry with irq_exit_rcu() and irqentry_exit(), which are described in the next paragraph. The only downside is that the early entry code up to irq_enter_rcu() must be aware that the preemption count has not yet been updated with the HARDIRQ_OFFSET state.

Note that irq_exit_rcu() must remove HARDIRQ_OFFSET from the preemption count before it handles soft interrupts, whose handlers must run in BH context rather than irq-disabled context. In addition, irqentry_exit() might schedule, which also requires that HARDIRQ_OFFSET has been removed from the preemption count.

Even though interrupt handlers are expected to run with local interrupts disabled, interrupt nesting is common from an entry/exit perspective. For example, softirq handling happens within an irqentry_{enter,exit}() block with local interrupts enabled. Also, although uncommon, nothing prevents an interrupt handler from re-enabling interrupts.

Interrupt entry/exit code doesn't strictly need to handle reentrancy, since it runs with local interrupts disabled. But NMIs can happen anytime, and a lot of the entry code is shared between the two.

NMI and NMI-like exceptions

NMIs and NMI-like exceptions (machine checks, double faults, debug interrupts, etc.) can hit any context and must be extra careful with the state.

State changes for debug exceptions and machine-check exceptions depend on whether these exceptions happened in user-space (breakpoints or watchpoints) or in kernel mode (code patching). From user-space, they are treated like interrupts, while from kernel mode they are treated like NMIs.

NMIs and other NMI-like exceptions handle state transitions without distinguishing between user-mode and kernel-mode origin.

The state update on entry is handled in irqentry_nmi_enter() which updates state in the following order:

- Preemption counter
- Lockdep
- RCU / Context tracking
- Tracing

The exit counterpart irqentry_nmi_exit() does the reverse operation in the reverse order.

Note that the update of the preemption counter has to be the first operation on enter and the last operation on exit. The reason is that both lockdep and RCU rely on in_nmi() returning true in this case. The preemption count modification in the NMI entry/exit case must not be traced.

Architecture-specific code looks like this:

```
noinstr void nmi(struct pt_regs *regs)
{
      arch_nmi_enter(regs);
      state = irqentry_nmi_enter(regs);

      instrumentation_begin();
      nmi_handler(regs);
      instrumentation_end();

      irqentry_nmi_exit(regs);
}
```

and for e.g. a debug exception it can look like this:

```
noinstr void debug(struct pt_regs *regs)
{
      arch_nmi_enter(regs);

      debug_regs = save_debug_regs();

      if (user_mode(regs)) {
              state = irqentry_enter(regs);

              instrumentation_begin();
              user_mode_debug_handler(regs, debug_regs);
              instrumentation_end();

              irqentry_exit(regs, state);
      } else {
              state = irqentry_nmi_enter(regs);

              instrumentation_begin();
              kernel_mode_debug_handler(regs, debug_regs);
              instrumentation_end();

              irqentry_nmi_exit(regs, state);
      }
}
```

There is no combined irqentry_nmi_if_kernel() function available as the above cannot be handled in an exception-agnostic way.

NMIs can happen in any context. For example, an NMI-like exception triggered while handling an NMI. So NMI entry code has to be reentrant and state updates need to handle nesting.