x2APIC Virtualization and Posted Interrupt Processing

Jack Tigar Humphries
jhumphri@cs.stanford.edu

I will discuss adding virtualization support for **interprocessor interrupts (IPIs)** using **x2APIC virtualization** and **posted interrupt processing** in a virtual machine monitor (VMM) based on Intel VT-x. I am extending Dune, an x86 Linux kernel module that uses VT-x to grant user-level applications safe access to privileged hardware resources, though this information will work with many different VMM designs. I will provide some background on VT-x, interrupt delivery, and x2APIC as necessary, though you will find it helpful to skim the information in chapters 6, 10, and 23 - 29 of the Intel manual.

1 Advanced Programmable Interrupt Controller

Interrupt delivery in a modern Intel system uses Intel's Advanced Programmable Interrupt Controller (APIC). Each processor core is equipped with a local APIC that sends and receives interrupts to and from other cores in the machine. There is a local APIC connected to each core and there is a global I/O APIC (connected to the system bus) that transfers requests between cores.

In this article, I will only concern myself with the local APIC. Some older machines are equipped with a local APIC known as an xAPIC. An xAPIC's registers are mapped to memory, so an operating system can interact with them through memory accesses. Newer machines are equipped with x2APICs, which are faster and can address more cores than xAPICs. An x2APIC has model-specific registers (MSRs), which an operating system can read from using the `rdmsr` instruction and write to using the `wrmsr` instruction.

2 Sending an IPI

I will explain how an operating system interacts with an x2APIC to send an IPI from one core to another.

An x2APIC has many different registers, though for sending an IPI without any advanced addressing techniques or tricks, only three are of concern: the local APIC ID register, the interrupt-command register (ICR), and the end-of-interrupt register (EOI).

Register	MSR Address
Local APIC ID	0x802
EOI	0x80B
ICR	0x830

The local APIC ID register contains the ID of the local APIC, which is unique for each core. This ID is used when addressing an IPI from one core to another. When a core wants to send an IPI, it writes the destination APIC ID, along with the interrupt vector and some other information, to the ICR. Once the write is complete, the IPI is sent.

On the receiving core, when an IPI is received, the interrupt vector number is used to select a handling function from the Interrupt Descriptor Table (IDT). Once the handling function has finished running, the receiving core writes a `0` to the EOI register, to indicate that it has finished processing the interrupt. Until `0` is written to the EOI register, the interrupt controller will not deliver additional interrupts to the operating system.

Below is some sample code for sending an IPI. Information about the specifics is included in the Intel manual.

```
void apic_send_ipi(uint8_t vector, uint32_t destination_apic_id) {
    uint64_t to_write = 0x0;

    //write the vector number to the least significant 8 bits
    to_write |= vector;
    //write the destination to the most significant 32 bits
    to_write |= ((uint64_t) destination_apic_id) << 32;

    //now write [to_write] to the ICR
    wrmsrl(0x830, to_write);
}
```

You can read a core's local APIC ID with the code below. The local APIC ID for an x2APIC is 32 bits.

```
uint32_t apic_id() {
    uint64_t apic_id;
    rdmsrl(0x802, apic_id);
    return (uint32_t)apic_id;
}
```

On the core that receives an interrupt, you should call the function below at the end of your handling function:

```
void apic_eoi() {
    //write 0x0 to the EOI register
    wrmsrl(0x80B, 0x0);
}
```

3 Virtualizing IPIs without Posted Interrupt Processing

Before Posted Interrupt Processing was added to Intel's processors, IPIs could still be virtualized and handled, but the delivery was slower. I will provide a simplified overview of the process.

The first step takes place in the guest OS running in VMX non-root mode. The guest first attempts to write to the ICR register of the local APIC, which consists of a performing an MSR write to address `0x830` if the guest assumes the local APIC is an x2APIC (or the guest OS will perform a memory write to the ICR register if it assumes the local APIC is an xAPIC).

Next, the VMM will intercept this MSR write. The VMM can intercept this write by setting the corresponding bit for address `0x802` in the MSR bitmap included in the VM control structure. Thus, once the guest OS writes to this MSR, a VM exit will occur and the VMM will regain control.

The VMM has discretion regarding how to handle this MSR write and deliver the interrupt to the destination core. One option is for the VMM (which, as a reminder, is running within the host OS kernel) to write information about the IPI to a shared piece of memory accessible to the destination core. For a simple IPI, the written information needs not include anything more than the vector of the IPI that the guest OS attempted to send. Once this information has been written, the VMM can send a real IPI to destination core in order to "ping" it so that it knows it needs to read the shared memory and pass the read IPI vector to the guest OS on that core.

When the destination core receives this IPI, a VM exit will occur and the VMM will regain control on that core (to ensure that a VM exit occurs when an external interrupt is received, set bit 0 in the Pin-Based VM-Execution Controls; see chapter 24 of the Intel manual). When the VMM reads the exit reason from the VM control structure, it will see that a VM exit occurred due to receipt of an external interrupt and it will read the interrupt vector from the interruption-information field (this requires setting bit 15 of the VM-Exit Control Field).

If the vector read from the interruption-information field matches the real IPI that the other core sent (the VMM can simply choose a vector value at startup to use in this type of case only, so checking for a match consists of just comparing the vector read from the interruption-information field to the pre-chosen value; KVM uses `0xf2` for example), then the VMM knows that it should read the virtual IPI information (namely, the virtual vector) from the shared memory. Otherwise, the VMM should simply call the appropriate handling function in the host OS, as the interrupt must have originated from elsewhere and has nothing to do with the guest OS.

Now that the destination core has the vector number of the virtual IPI, it can "inject" the interrupt into the guest OS. This is as simple as setting some bits in the VM control structure (including the virtual vector number) and performing a VM entry. The guest OS will immediately handle the virtual IPI.

This is a simplified explanation of virtualizing IPIs. There are other concerns that need to be addressed in a real system. For example, it is possible that the destination core is not in VMX non-root mode when the virtual IPI is sent. In this case, the interrupt will need to be queued and then injected into the destination core's non-root mode when it is about to reenter non-root mode. Furthermore, even if the destination core is running in VMX non-root mode when it receives the real IPI, it is possible that the guest OS on that core may not be able to accept interrupts at the moment and the interrupt will need to be queued, such as if the Interrupt Flag is set to 0 in the guest OS.

This method of virtual IPI delivery works well, but it is slower than virtual IPI delivery with Posted Interrupt Processing. Most notably two VM exits are required: one on the sending core to virtualize the MSR write and send a real IPI, and one on the destination core to handle the real IPI and inject the interrupt into the guest OS. As we will see with Posted Interrupt Processing, we can virtualize IPIs with just one VM exit on the sending core and no VM exits on the receiving core.

4 What is Posted Interrupt Processing?

Posted Interrupt Processing allows a VMM to virtualize IPIs in a similar manner as described above, but with lower overhead than with interrupt injection.

The process of the guest OS writing to the ICR register, which triggers a VM exit, is the same.

However, the way that the VMM sends the IPI to the destination core is slightly modified. To start, when Posted Interrupt Processing is enabled, two new fields must be filled in within the VM control structure on each core before the guest OS starts: the posted-interrupt notification vector and the posted-interrupt descriptor.

The posted-interrupt descriptor is a hardware-defined data structure that serves the same purpose as the "shared memory" I described above. The sending core simply writes the virtual IPI vector to the guest core's descriptor before sending the "real IPI." The posted-interrupt notification vector serves the same purpose as the "real IPI" vector I described above. It is a value that is chosen when the VMM starts; when the sending core wants to "ping" the destination core to let it know that a virtual IPI needs to be delivered, it simply sends a real IPI with the posted interrupt notification vector.

When the real IPI is received by the destination core, a VM exit does not automatically occur, since Posted Interrupt Processing is enabled. Instead, the receiving core (while still in non-root mode) compares the vector of the real IPI to the posted-interrupt notification vector set in the VM control structure. If the vectors match, then it acknowledges receipt of the interrupt to the local

APIC, reads the virtual vector from the posted-interrupt descriptor, clears the posted-interrupt descriptor, and delivers the virtual interrupt to the guest OS **without a VM exit**. If the vectors do not match, a VM exit occurs and the VMM must then ensure that the correct handling function is called within the host kernel, as the IPI was sent from elsewhere and is not meant for the guest OS.

Thus, a VM exit must still occur on the sending core but a VM exit must no longer occur on the receiving core. It is possible for the guest OS to send virtual IPIs without a VM exit on the sending core, such as by giving the guest OS direct access to the x2APIC's registers and access to the posted-interrupt descriptor (by modifying the EPT). Even though there are clear performance advantages to doing this, this is unsafe as it would enable the guest OS to send arbitrary interrupt vectors to any core in the system, which will be handled by the host OS if the vector does not match the posted-interrupt notification vector or if the destination core is not in VMX non-root mode. The guest OS could perhaps DOS the machine or somehow trigger a host kernel panic. Furthermore, the guest OS would need to include code for posted IPIs in its own kernel (e.g. the proper bits must be set in the posted-interrupt descriptor, the correct vector must be used, etc.) and the guest OS would need to be aware that it is being virtualized, which may be undesirable.

5 Supporting Posted Interrupt Processing

Support for Posted Interrupt Processing was added to the Dune project, which is hosted on GitHub. The code to support this feature is in commit 301c8614e82933abde8fd8167ff4b1a5b6341fd8.