Kernel Interrupt: A Major Overhaul

- APIC Initialization

&

- Vector Allocation

Dou Liyang  douly.fnst@cn.fujitsu.com

June 20 2018
Outline

- Basics of an interrupt
- Overhaul of interrupt
  - APIC Initialization
  - Vector Allocation
- Future work

What's next?
What is An Interrupt?

- A hardware signal
- Emitted from a peripheral to a CPU
- Indicating that a device-specific condition has been satisfied

From Marc Zyngier <marc.zyngier@arm.com>
Multiplexing Interrupts

- Having a single interrupt for the CPU is usually not enough
- Most systems have tens, hundreds of them
- **An interrupt controller** allows them to be multiplexed
- Very often architecture or platform specific
- In old x86 machine, there was a PIC called 8259A
  - a chip responsible for sequentially processing multiple interrupt requests from multiple devices
  - Called **PIC Mode**
Multiplexing Interrupts in SMP System

- Only a CPU is usually not enough
- Most systems have tens, hundreds of CPUs
- An new interrupt controller should be used
- In x86 machine, there is an APIC
  - Local APIC is located on each CPU core, handles the CPU-specific interrupt configuration
  - I/O APIC distribute external interrupts from multiple devices to multiple CPU cores
  - Called Symmetric I/O Mode
More than wired interrupts: MSIs

- **Message Signaled Interrupts** are an alternative to line-based interrupts
  - Trigger an interrupt by writing a value to a particular memory
  - Allow the use of the same buses as the data
Handle an Interrupt

- Preempt current task ① ②
  - *Pause execution* of the current process.

- Execute interrupt handler ③ ～ ⑤
  - Search for *the handler of the interrupt* and transfer control

- Resume the task ⑥
  - *Return to execute* the current process;

<Diagram>

1. CPU
2. Current Task
3. Has an Interrupt
4. Interrupt Handler
5.⑤
6.⑥

Copyright 2018 FUJITSU LIMITED
How Does “Handle an Interrupt” Work?

**APIC** and **Vector** mechanism make it work

1. Delivery the IRQ through the **APIC**
2. CPU search the handler in IDT through the **vector**
3. Get the **irq_desc** structure through the **vector**.
4. Use the **irq_desc** to get what the interrupt needs
   - **device info**
   - **interrupt controller info**
   - **IRQ action list info**
5. Execute the interrupt service routine (ISR)
Why “APIC and Vector” Can Work?

Do many initialization and setup works when Linux boots up

- For the interrupt delivery
  - Initialize 8259A
  - Switch interrupt delivery mode
  - Initialize APIC
    - Local APIC setup
    - I/O APIC setup

- For IDT table,
  - Initialize the mapping of Vector and Handler

- For each Interrupt,
  - Allocate an IRQ
  - Allocate an irq_desc
  - Assign a vector

APIC Initialization

Device

CPU

Interrupt Context

Normal Context

Vector Allocation

Interrupt handler
Outline

Basics of an interrupt

Overhaul of interrupt

APIC Initialization
Vector Allocation

Future work

What's next?

Copyright 2018 FUJITSU LIMITED
Existing Problems

- Interrupt in x86 is a **conglomerate** of ancient bits and pieces
  - Subject to 'modernization' and **features** over the years
    - Kdump
    - CPU Hotplug/System hibernation
    - Multi-queue devices

- It looks like a penguin full of band-aids
  - Can work, but can’t see how it works easily.
Problems of APIC Initialization

- Horrible interrupt mode setup
  - Setup the mode at random places
  - Run the kernel with the potentially wrong mode

- Tangle the timer setup with interrupt initialization
Overhaul of APIC Initialization

1. Unify the APIC and interrupt mode setup
   - Construct a **selector** for the interrupt delivery mode

Kconfig

- CONFIG_X86_64
- CONFIG_X86_LOCAL_APIC
- CONFIG_x86_IO_APIC
- CONFIG_SMP

CPU Capability

- `boot_cpu_has(X86_FEATURE_APIC)`

MP table

- `smp_found_config`

ACPI table

- `acpi_lapic`
- `acpi_ioapic`
- `nr_ioapic`

Command line options

- `disable_apic`
- `skip_ioapic_setup`
- `nolapic/noapic/apic=`

See arch/x86/kernel/apic/apic.c apic_intr_mode_select()
Overhaul of APIC Initialization

1. Unify the APIC and interrupt mode setup
   - Provide a single function

- `init_bsp_APIC()`
- `native_smp_prepare_cpus()`
- `smp_init()`
- `apic_intr_mode_init()`

See arch/x86/kernel/apic/apic.c apic_intr_mode_init()
Overhaul of APIC Initialization

2. Disentangle the timer setup from the APIC initialization

- Refactor the delay logic during APIC initialization process.
  - Either use TSC or a simple delay loop to make a rough delay estimate

Split local APIC timer setup from the APIC setup
Overhaul of APIC Initialization

3. **Reorganize** the interrupt initialization

- Set up the final interrupt delivery mode **as soon as possible**.

1) Set up the legacy timer (PIT/HPET)
   - `x86_init.timers.timer_init()`

2) Set up APIC/IOAPIC
   - `x86_init.irqs.intr_mode_init()`

3) TSC calibration
   - `tsc_init()`

4) Local APIC timer setup
   - `x86_init.timers.setup_percpu_clockev()`
Overhaul of APIC Initialization

4. Some others

- Refactor some common APIC function
- Compatible with ACPI initialization
- Bypass the hypervisor, Such as KVM and Xen

5. Can check which mode the interrupt is by ‘dmesg’:

```
0.000000] Kernel command line: BOOT_IMAGE=/vmlinuz-4.16.0 root=UUID=10f10326-c923-4098-86aa-3
0.000000] Memory: 1465920K/2096616K available (12300K kernel code, 2367K rdata, 3948K rodata
0.000000] SLUB: Hwalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
0.000000] ftrace: allocating 35971 entries in 141 pages
0.000000] Hierarchical RCU implementation.
0.000000] RCU restricting CPUs from NR_CPUS=8192 to nr_cpu_ids=4.
0.000000] Tasks RCU enabled.
0.000000] RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=4
0.000000] NR_IRQS: 524544, nr_irqs: 456, preallocated irqs: 16
0.000000] Offload RCU callbacks from CPUs: .
0.000000] Console: colour VGA+ 80x25
0.000000] console [tty0] enabled
0.000000] console [ttyS0] enabled
0.000000] ACPI: Core revision 20180105
0.000000] ACPI: 1 ACPI AML tables successfully acquired and loaded
0.000000] clocksource: hpet: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 191126044
0.000000] APIC: Switch to symmetric I/O mode setup
0.006000] ..TIMER: vector=0x30 apicid=0 pin=2 apic2=-1 pin2=-1
0.011000] tsc: Fast TSC calibration using PIT
0.012000] tsc: Detected 3292.164 MHz processor
0.013000] tsc: Marking TSC unstable due to TSCs unsynchronized
0.014334] Calibrating delay loop (skipped), value calculated using timer frequency.. 6584.32
```
Outline

What's next?

- Basics of an interrupt
- Overhaul of interrupt
  - APIC Initialization
  - Vector Allocation
- Future work
Problems of \textit{Vector Allocation}

- Horrible worst vector management mechanism
  - \textit{Abuse} the interrupt allocation for different type interrupts
  - Serve all different use cases \textit{in one go}
  - Based on \textit{nested loops} to search
  - Cause vector space \textit{exhaustion}
  - Allocate vectors at the \textit{wrong} time and on the \textit{wrong} place

- Some dubious properties, causes \textit{high complexity}
  - Multi CPU affinities for an IRQ
  - Priority level spreading

- Lack of instrumentation
  - All of this is a black box which allows no insight into the actual vector usage
Overhaul of *Vector Allocation*

- 1. Classify the *types* of vectors
- 2. Refactor the *vector allocation mechanism*
- 3. Switch to a *reservation scheme*
- 4. Some Others

### Diagram

1. Vector Classifier
   - Request IRQ
   - IRQ enabled
   - IRQ startup
   - Any functions which request an vector

2. Vector Allocator
   - An Vector ID

3. Reservation Scheme
1. Classify the types of vectors

Each CPU has 256 vectors, but some are fixed.

1. System Vector
   * Vectors 0 ... 31
   * Vector 128
   * Vectors INVALIDATE_TLB_VECTOR_START ... 255

2. Legacy Vector
   * Vectors 0x30 ... 0x3f

Others are allocated dynamically for normal and managed interrupts.
### Overhaul of *Vector Allocation*

1. Classify the **types** of vectors
   - For external interrupts
   - Depend on **Interrupt Affinity** (the set of CPUs that can handle this interrupt)

### 3. Normal Vector

<table>
<thead>
<tr>
<th>At setup time</th>
<th>Normal Interrupt</th>
<th>Managed Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Affinity may be NULL</td>
<td>Affinity must have been setup</td>
<td></td>
</tr>
<tr>
<td>A subset of the online CPUs</td>
<td>the possible CPUs may be included</td>
<td></td>
</tr>
</tbody>
</table>

### 4. Managed Vector

<table>
<thead>
<tr>
<th>User space</th>
<th>Normal Interrupt</th>
<th>Managed Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Affinity can be modified</td>
<td>Affinity is fixed</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>When migration</th>
<th>Normal Interrupt</th>
<th>Managed Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ can be moved to any online CPUs</td>
<td>IRQ can move only in the affinity.</td>
<td></td>
</tr>
<tr>
<td>Affinity can be even reset</td>
<td>But, can be shutdown and restarted.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Affinity can’t be reset</td>
<td></td>
</tr>
</tbody>
</table>
Overhaul of *Vector Allocation*

2. Refactor the *vector allocation mechanism*

- Create a new bitmap matrix allocator——*IRQ Matrix*
Overhaul of *Vector Allocation*

2. Refactor the vector allocation mechanism

- Use the matrix for *System* vector

![Diagram showing vector allocation mechanism with system bitmap, available, and allocated counters for global and percpu levels.](image-url)
Overhaul of Vector Allocation

2. Refactor the vector allocation mechanism

- Use the matrix for Legacy vector
Overhaul of *Vector Allocation*

2. Refactor the vector allocation mechanism

- Use the matrix for *Normal* vector

![Diagram showing vector allocation mechanism]

- **Global**
  - System bitmap
    - 1111111111000000000000000000000011111

- **Global Counters**
  - system
  - available
  - allocated

- **CPU 0 Percpu**
- **CPU 1 Percpu**
- **CPU n Percpu**

- **Percpu**
  - Allocated bitmap
    - 000000111111110000000000000000
  - Managed bitmap
    - 000000000000000000001110000000

**Step 1**

**Step 2**
Overhaul of *Vector Allocation*

2. Refactor the vector allocation mechanism

- Use the matrix for *Managed* vector
Overhaul of *Vector Allocation*

3. Switch to *reservation scheme*
   - Reserve a *new system vector*, just in case

3.1 When the interrupt is allocated and initialized:

Previously

Assign a *real vector* for each interrupts

wasteful

Now

1. Update the *reservation request counter*

2. Assign *the reserved vector* for each interrupts
Overhaul of *Vector Allocation*

3. Switch to *reservation scheme*
   - *Separate* activation and startup
   - Assign the *real* vector

3.2 When the interrupt is *requested*:

**Startup**
- **Activate**
- **Fail?**
  - **Continue…**

**Vector Space Saving**

**Activate**
- Assign a real vector for *normal* interrupts
- **Can fail**
  - **Startup**
    - Assign a real vector for *managed* interrupts
  - **Continue…**
Overhaul of **Vector Allocation**

- Some Others:
  - Change from Multi CPU targets to *single interrupt targets*.
  - Remove priority level spreading
  - Simplify hotplug vector accounting
  - Equip with trace points and detailed debugfs information

- Can see the Vector Allocation by:
  - `cat /sys/kernel/debug/irq/irqs/$N`
  - `cat /sys/kernel/debug/irq/domains/$N`
What's next?

- Basics of an interrupt
- Overhaul of interrupt
  - APIC Initialization
  - Vector Allocation

*Future work*
Future work

- Kernel's notion of possible CPU count should be realistic
  - Once the kernel initialized:
    Make the possible CPU count realistic

- The vector allocation is a *generic* mechanism
  - Can be used to other architectures
Thank you !