mikro - Multiprocessor Init in Kernel

Julien Freche

julien.freche@lse.epita.fr
http://lse.epita.fr/

Outline I

Introduction

Too many questions

I will talk about x86 system with multiple processors.

- How to handle interruptions ?
- How to boot all CPUs ?
- What is the state of the system at boot ?
- How to detect the number of CPU(s) ?

Dealing with multiple processors requires to deal with interruptions. Let's see why.

Interruptions

I interrupt you

Interruption

Signal sent to a processor to report an event that requires immediate attention.

Interrupts can be caused by:

- Hardware: event caused by some device
- Software: system call from userland, debugging purposes, ..

Old PIC

PIC

It controls the CPU's interrupt mechanism, by accepting several interrupt requests and feeding them to the processor in order.

Limitations:

- Only 8 pin per PIC (16 IRQ on x86 with two PICs).
- No SMP support, can only send interrupts to one CPU.
- Programmed with IO ports

The PIC is no replaced by the IOAPIC on modern systems.

IOAPIC

IOAPIC

It collects interrupts from all devices and provide a way to send it to a CPU or a group of CPU.

Features:

- 24 pin per IOAPIC
- Memory mapped device
- You can have multiple IOAPICs
- Special bus to send interruptions: ICC bus
- Handle global interruptions (not specific to a CPU)

LAPIC

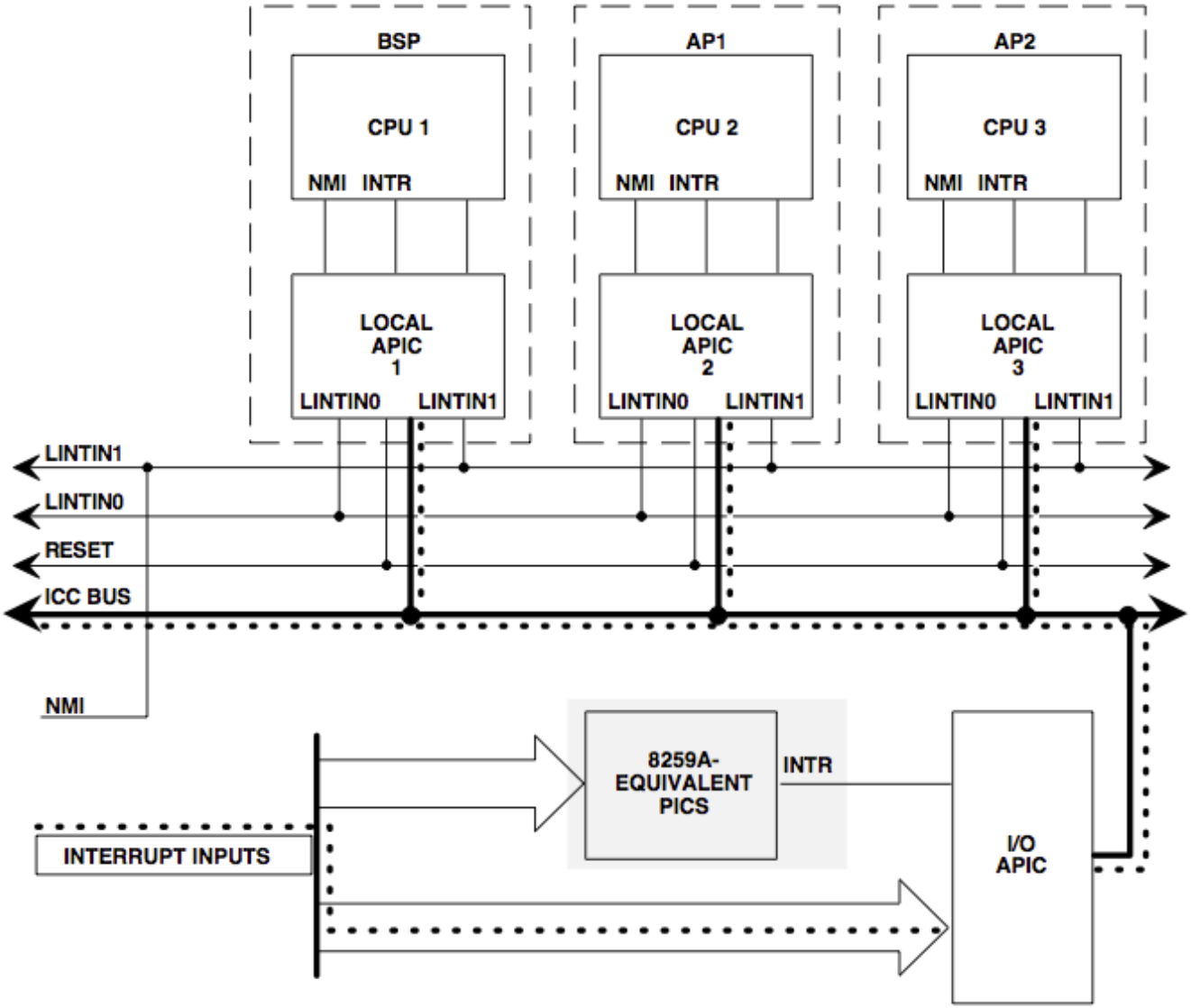
Local APIC

It collects local interrupts and provide a way to a CPU to accept interrupts.

Features:

- 2 pin per local APIC
- Memory mapped device
- Each CPU has a local APIC
- Timer, Performance monitoring, Thermal sensor
- Use ICC bus to speak with IOAPIC(s)
- Handle local interruptions
- Each Lapic has an unique ID

All together



IPI

InterProcessor Interrupts

IPIs are interrupts issued by one processor and sent to another.

- Issued by the Local Apic
- Destination
  - All including self
  - All excluding self
  - Self
  - One processor
- Delivery Mode
  - Low priority
  - NMI
  - INIT, STARTUP
  - ...

CPU init

(Gla)DOS: I am still alive

For backward compatibility all modern PCs starts in the following state:

- Only one CPU active
- Real mode (16 bits)
- PIC available to handle interruptions

BootStrap Processor

Bootstrap Processor

This processor is chosen by the BIOS to start executing the bootloader code. It is the first to execute the kernel code and must wake up the other processors if needed.

Duties on mikro:

- Create a GDT and an IDT
- Init the paging
- Move the kernel code to its final location
- Parse MP Tables and/or MADT table
- Init IOAPIC(s) and its own LAPIC
- Perform some per-cpu init
- Wake every processors
- Start scheduling tasks

MultiProcessor Table

Created by Intel in 1997. Provides MP related informations to the OS.

Informations:

- Processors list
- Buses list
- IOAPICs list
- Interrupts list

Limitations:

- qemu: list only processors, not core nor threads.
- On some real hardware: list only cores, not threads.
- On other hardwares: not present

MADT: Multiple APIC Description Table

Part of the ACPI spec. Provides informations about an SMP system.

Informations:

- Processors, cores, threads
- IOAPICs, x2APIC list
- Interrupt Source Override

Kind of the easy part of the ACPI (no AML).

Application Processor

Application Processor

Processor in the halt state, waiting for a special IPI to start executing code. This processor is in real mode.

Duties on mikro:

- Jump to protected mode
- Create its own GDT
- Load the existing IDT
- Perform some per-cpu init
- Init its LAPIC
- Start scheduling tasks

INIT-SIPI-SIPI

Wake up!

A processor will start executing code when receiving this sequence of IPIs.

- Copy some code in low memory
- Send an INIT IPI
- Send a StartUP IPI
- Send another StartUP IPI

Some magic

I found this code on the internet.

```
.init:
  ; set NMI handler
  mov dword [4*2], .boot
  xor  eax, eax
  mov  ebx, (11b shl 18) + (0 shl 15) + (1 shl 14)
                    + (0 shl 11) + (100b shl 8) + 2

  ;trigger interrupts in every processor
  mov [dword 0xFEE00300 + 16], eax
  mov [dword 0xFEE00300 + 00], ebx
  ret

align 16
.boot:
```

It works on qemu and some systems.

Let me explain it

Steps:

- Register boot function as handler into the IDT
- Set eax to zero
- Set IPI parameters into ebx:
  - NMI
  - Physical, Assert level, Edge trigger mode
  - All excluding self
- Send the IPI using the LAPIC

Kind of the quickest hack to wake every processors !

What is wrong?

Issues:

- Do not follow the spec..
- Assume that LAPIC is at 0xfee00000 (default addr but can be changed)
- Wake every processor (even processor disabled by the BIOS)

Kind of the quickest hack to wake every processors !

Percpu variables

Usage

My precious variable

Per-cpu variable

A cpu local variable. Each cpu can has a different value stored in this variable.

Considerations:

- Kind of TLS (Thread Local Storage) but for processors
- Accessed using macro on Linux
- Useful to change behavior of the code depending on the CPU

Implementation

How to implement it ?

- Create a special section in binary file for these variables
- Allocate needed space by every processor for these variables
- Set a special entry in every GDT (at the same offset for every processor) but with a different values.
- Set FS to this special entry in the GDT
- Access variables relatively to FS

Using clang

clang: 1, gcc: 0

With clang you can do the following:

```
# define FS_RELATIVE address_space(257)
# define PCPU_S section(".cpuvar")
# define __percpu __attribute__((FS_RELATIVE, PCPU_S))
int __percpu myvar;

void do_something()
{
  myvar = 42;
}
```

After per-cpu init, you can access your variable normally without macros. Pretty clean.

Conclusion

Almost done

To finish:

- To support SMP in kernel, think about it early
- Very interesting subject closely related to interruptions
- Think about a clever algorithm to dispatch interrupts
- All SMP systems has IOAPIC and LAPIC, try to play with it !

Contacts

Julien Freche

- julien.freche@lse.epita.fr
- @JulienFreche

Thank you for your attention