8-CPU同步

搜狐焦点计算机基础学习系列课程

李少鹏 2019-04

目录结构

I/O

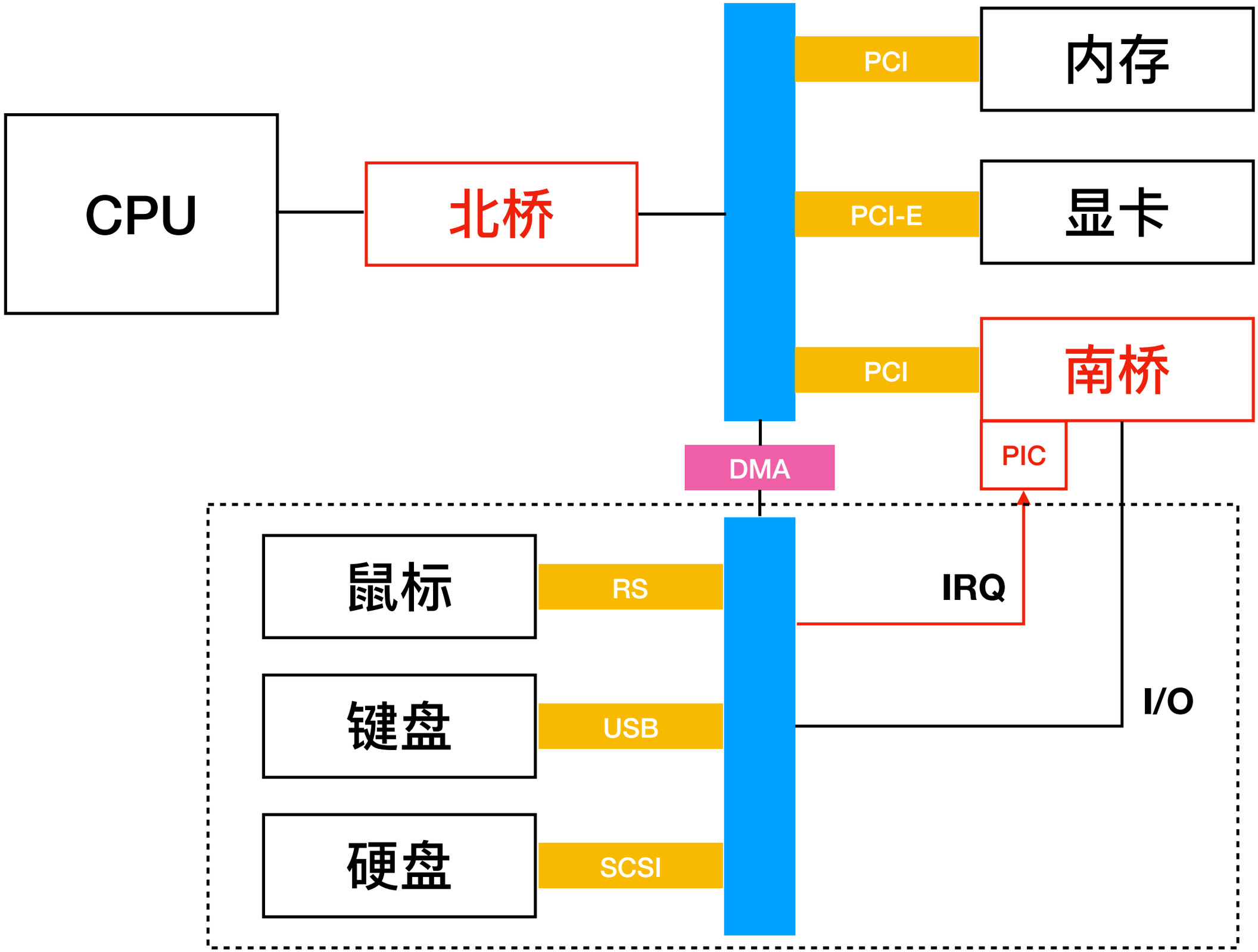
PIC/APIC

时钟

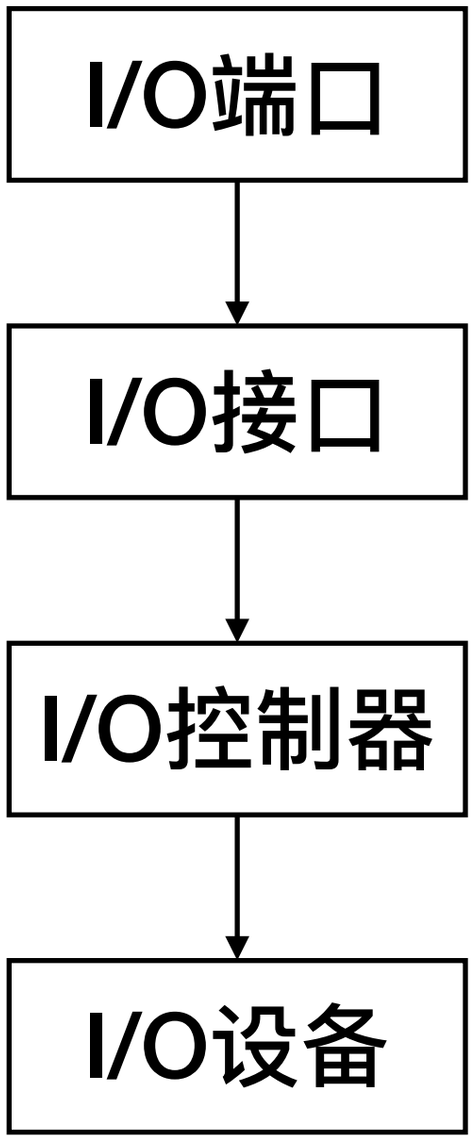
多核架构

同步

CPU-I/O：I/O的位置



CPU-I/O：I/O的四层抽象



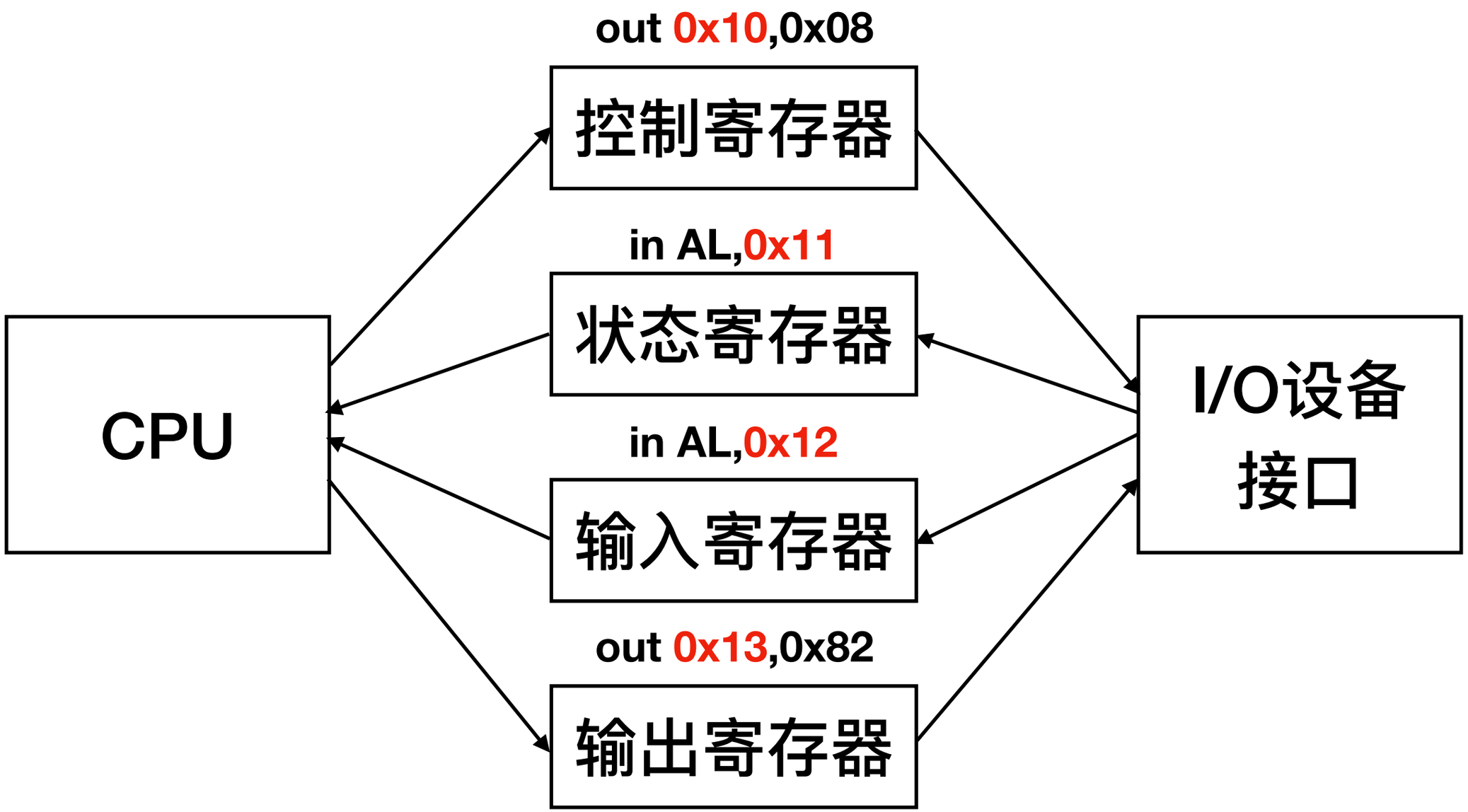
汇编语言中，in和out命令对应的设备编号，一个设备可以对应多个编号，每个编号对应一个1byte的数据地址。

连接I/O端口与I/O控制器的硬件电路，起翻译作用；连接到PIC上，代表设备发出IRQ中断请求；USB、SCSI都是I/O接口。

一般位于I/O设备中，连接I/O设备与I/O接口；例如磁盘控制器。

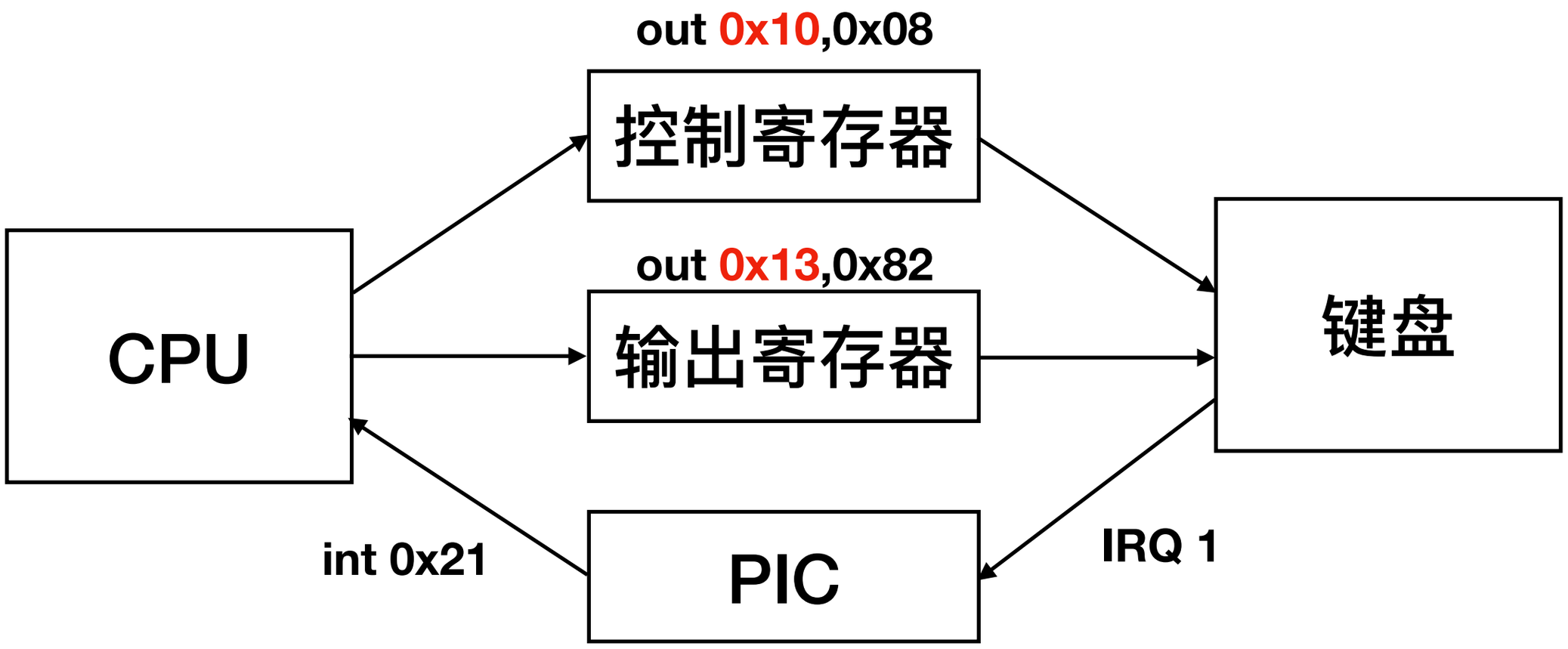
鼠标、键盘、磁盘、打印机、网卡等

CPU-I/O：CPU与I/O设备的四种交互



CPU的I/O端口空间为0x0-0xFFFF，一共65536个

CPU-I/O：CPU与I/O设备之间的唤醒



CPU的I/O端口空间为0x0-0xFFFF，一共65536个

IRQ：Interrupt Request

CPU-I/O：PMIO与MMIO

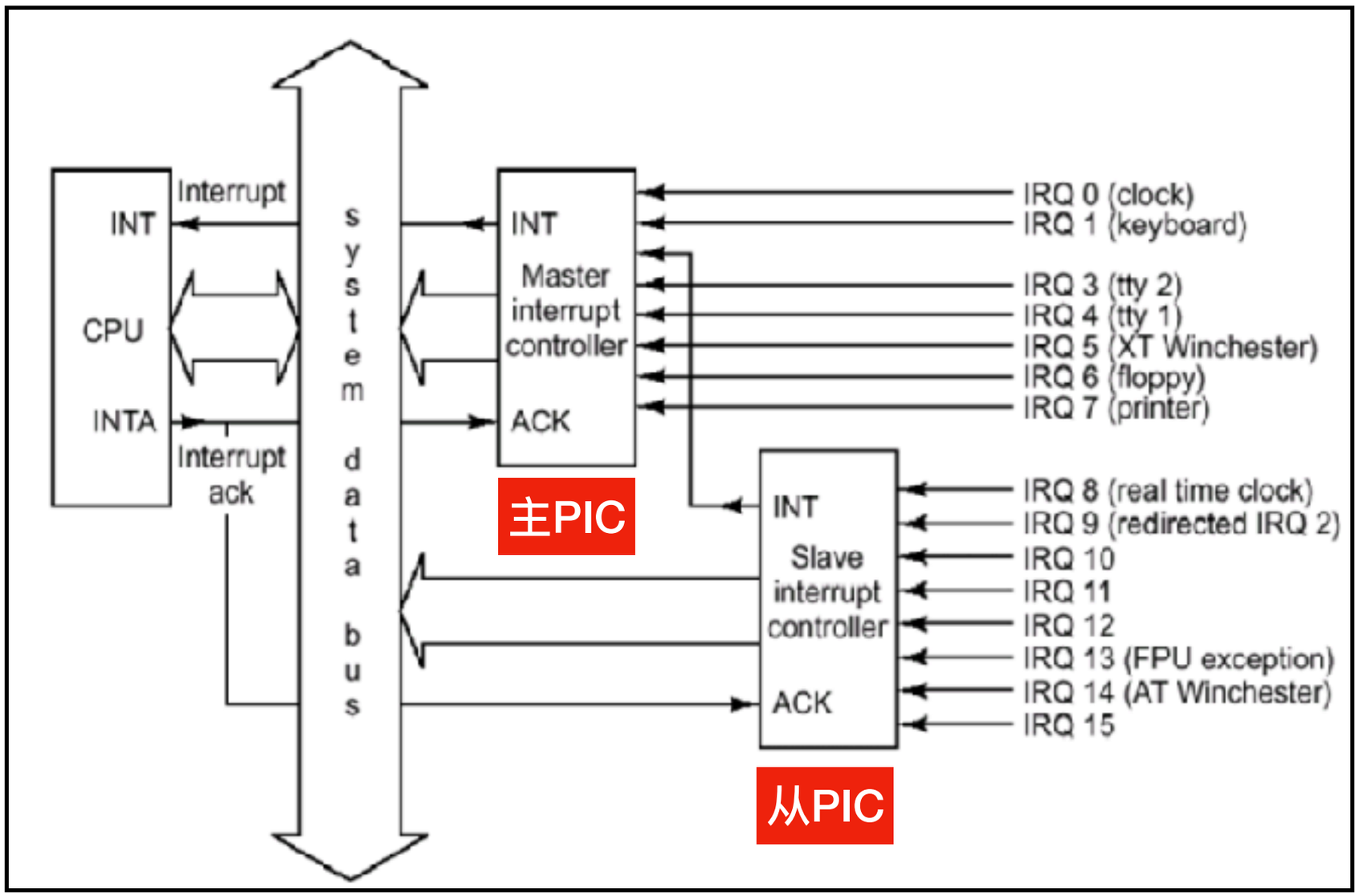
PMIO：Port-mapped I/O，I/O设备映射到端口空间，用in，out访问
MMIO：Memory-mapped I/O，I/O设备映射到内存空间，用mov访问

```
[root@localhost proc]# cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-0060 : keyboard
0064-0064 : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
02f8-02ff : serial
03c0-03df : vga+
03f8-03ff : serial
0800-0803 : ACPI PM1a_EVT_BLK
0804-0805 : ACPI PM1a_CNT_BLK
0808-080b : ACPI PM_TMR
0810-0815 : ACPI CPU throttle
0820-082f : ACPI GPE0_BLK
0850-0850 : ACPI PM2_CNT_BLK
0880-08ff : pnp 00:07
0900-091f : pnp 00:07
0920-0923 : pnp 00:07
```

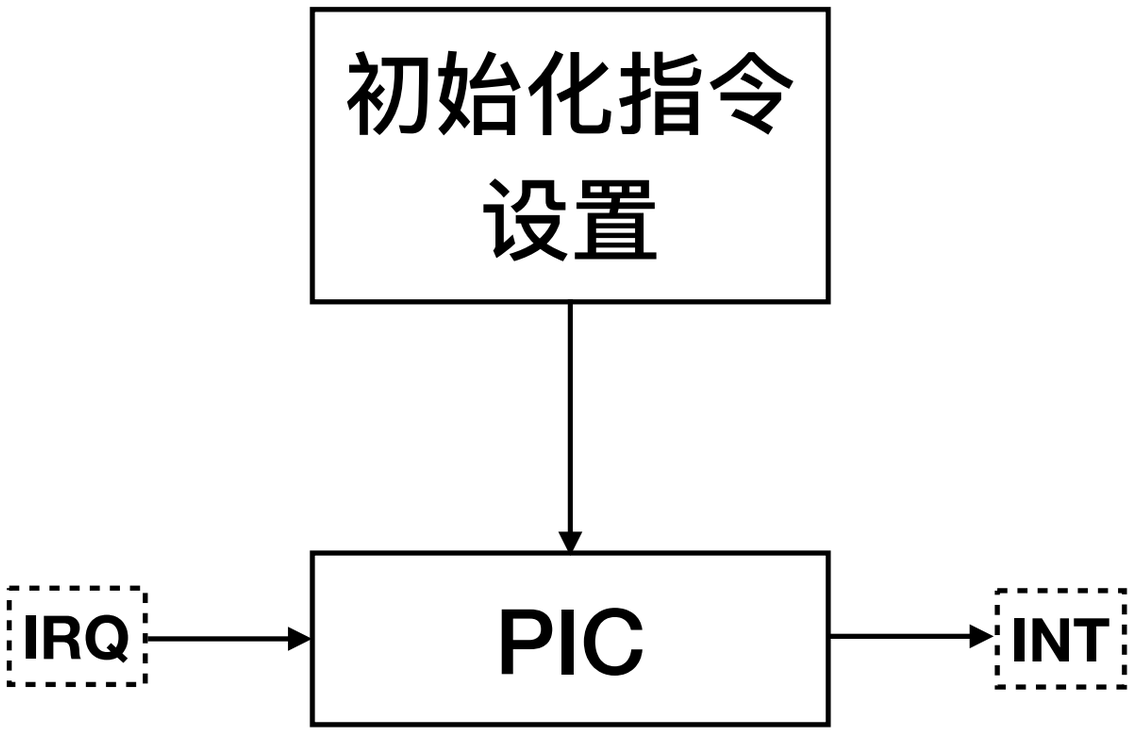
```
[root@localhost proc]# cat /proc/iomem
00010000-0009ffff : System RAM
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000c8000-000c8fff : Adapter ROM
000c9000-000cbdff : Adapter ROM
000cc000-000d21ff : Adapter ROM
000f0000-000fffff : System ROM
00100000-cf378fff : System RAM
  00400000-00619f35 : Kernel code
  00619f36-006f79db : Kernel data
cf379000-cf38efff : reserved
cf38f000-cf3cdfff : ACPI Tables
cf3ce000-cfffffff : reserved
d5800000-d5ffffff : PCI Bus #06
  d5800000-d5ffffff : 0000:06:03.0
d6000000-d9ffffff : PCI Bus #01
  d6000000-d7ffffff : 0000:01:00.0
    d6000000-d7ffffff : bnx2
  d8000000-d9ffffff : 0000:01:00.1
    d8000000-d9ffffff : bnx2
da000000-ddffffff : PCI Bus #02
  da000000-dbffffff : 0000:02:00.0
```

PIC：可编程中断控制器

PIC：Programmable Interrupt Controller
由intel的8259A芯片实现，实现把其他I/O的中断请求发送到CPU
8259A也是一种I/O设备，有自己的I/O端口号（0x20,0x04;0xA1,0x02）
两个8259A芯片串联，实现支持15路IRQ的硬件线路



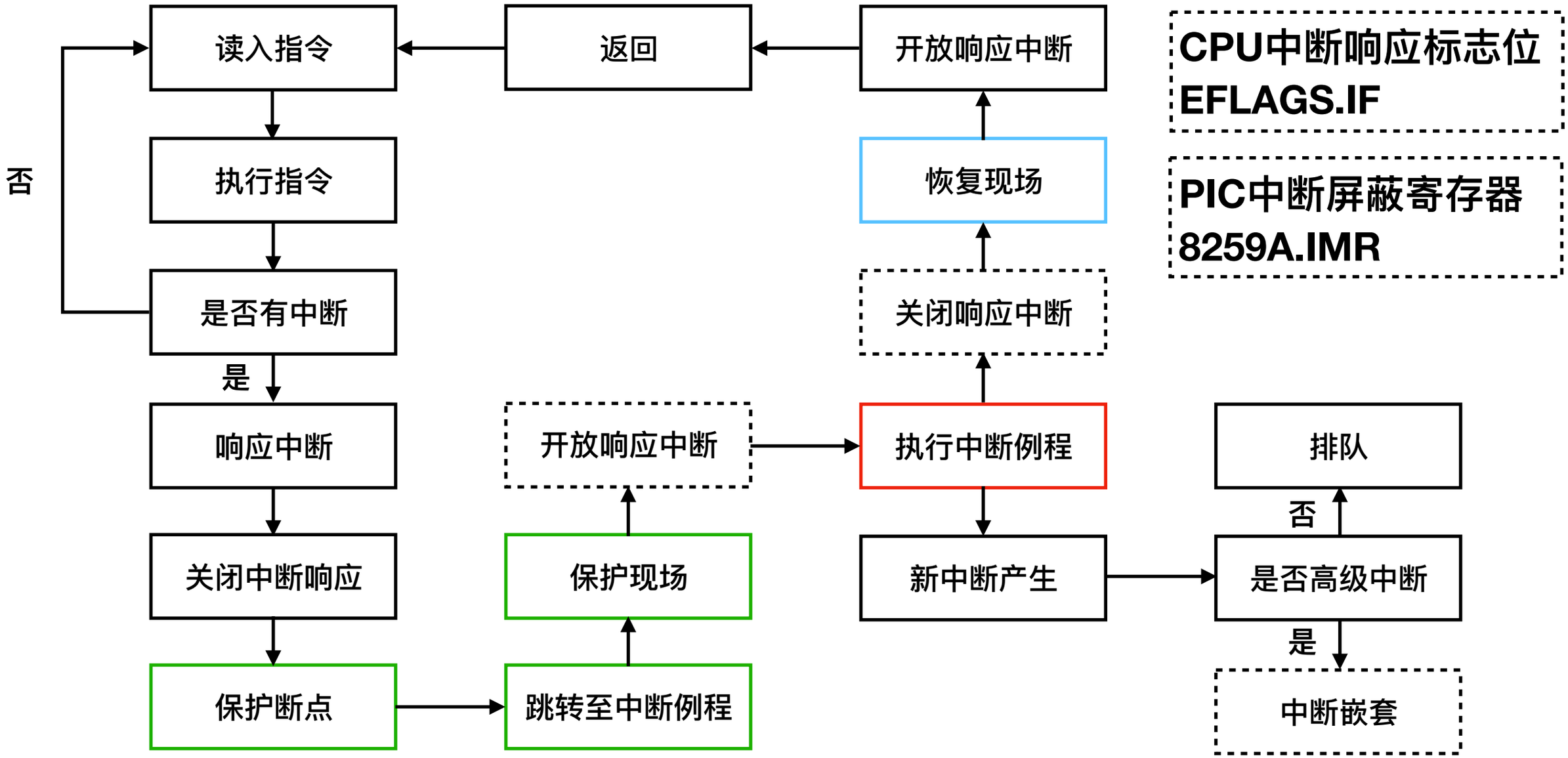
PIC：初始化




```
debian:~# cat /proc/interrupts
           CPU0
  0:    3555641          XT-PIC  timer
  1:        323          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  8:          3          XT-PIC  rtc
 13:          0          XT-PIC  fpu
 14:      73065          XT-PIC  ide0
NMI:          0
```

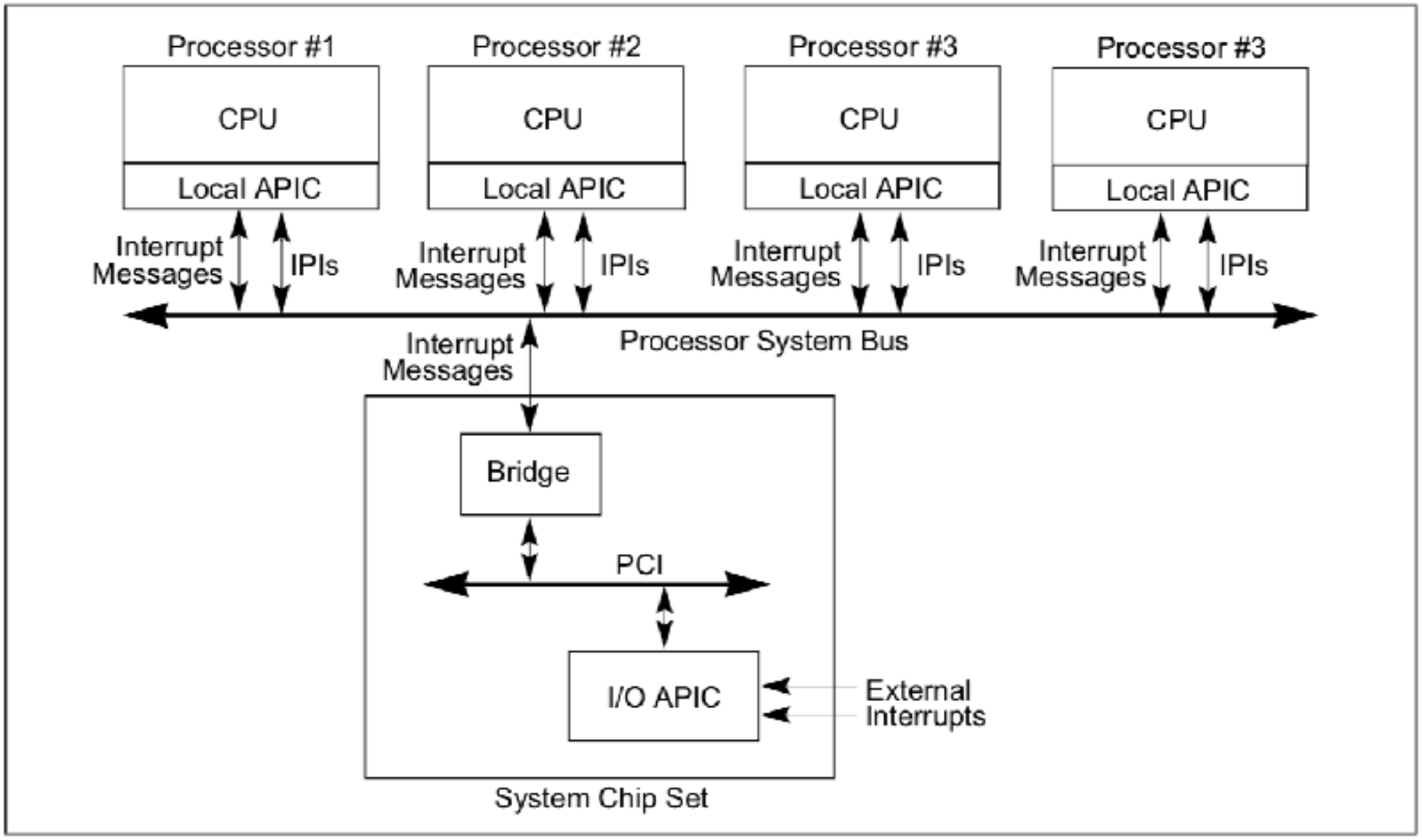
```
void init_pic(void)
{
    //禁止主从PIC所有中断
        io_out8(PIC0_IMR,  0xff  );
        io_out8(PIC1_IMR,  0xff  );

    //主PIC设置
        io_out8(PIC0_ICW1, 0x11  ); //边沿触发模式
        io_out8(PIC0_ICW2, 0x20  ); //IRQ0-7由INT 20-27接收
        io_out8(PIC0_ICW3, 1 << 2); //从PIC由主PIC IRQ2连接
        io_out8(PIC0_ICW4, 0x01  ); //无缓冲区模式

    //从PIC设置
        io_out8(PIC1_ICW1, 0x11  ); //边沿触发模式
        io_out8(PIC1_ICW2, 0x28  ); //IRQ8-15由INT 28-2f接收
        io_out8(PIC1_ICW3, 2     ); //从PIC由主PIC IRQ2连接
        io_out8(PIC1_ICW4, 0x01  ); //无缓冲区模式

        io_out8(PIC0_IMR,  0xfb  ); //11111011 主PIC允许IRQ2中断
        io_out8(PIC1_IMR,  0xff  ); // 11111111 从PIC禁止所有中断

        return;
}
```

PIC：硬中断响应流程


读入指令
返回
开放响应中断
CPU中断响应标志位
EFLAGS.IF
执行指令
恢复现场
PIC中断屏蔽寄存器
8259A.IMR
否
是否有中断
关闭响应中断
是
响应中断
开放响应中断
执行中断例程
排队
关闭中断响应
保护现场
新中断产生
否
是否高级中断
保护断点
跳转至中断例程
是
中断嵌套

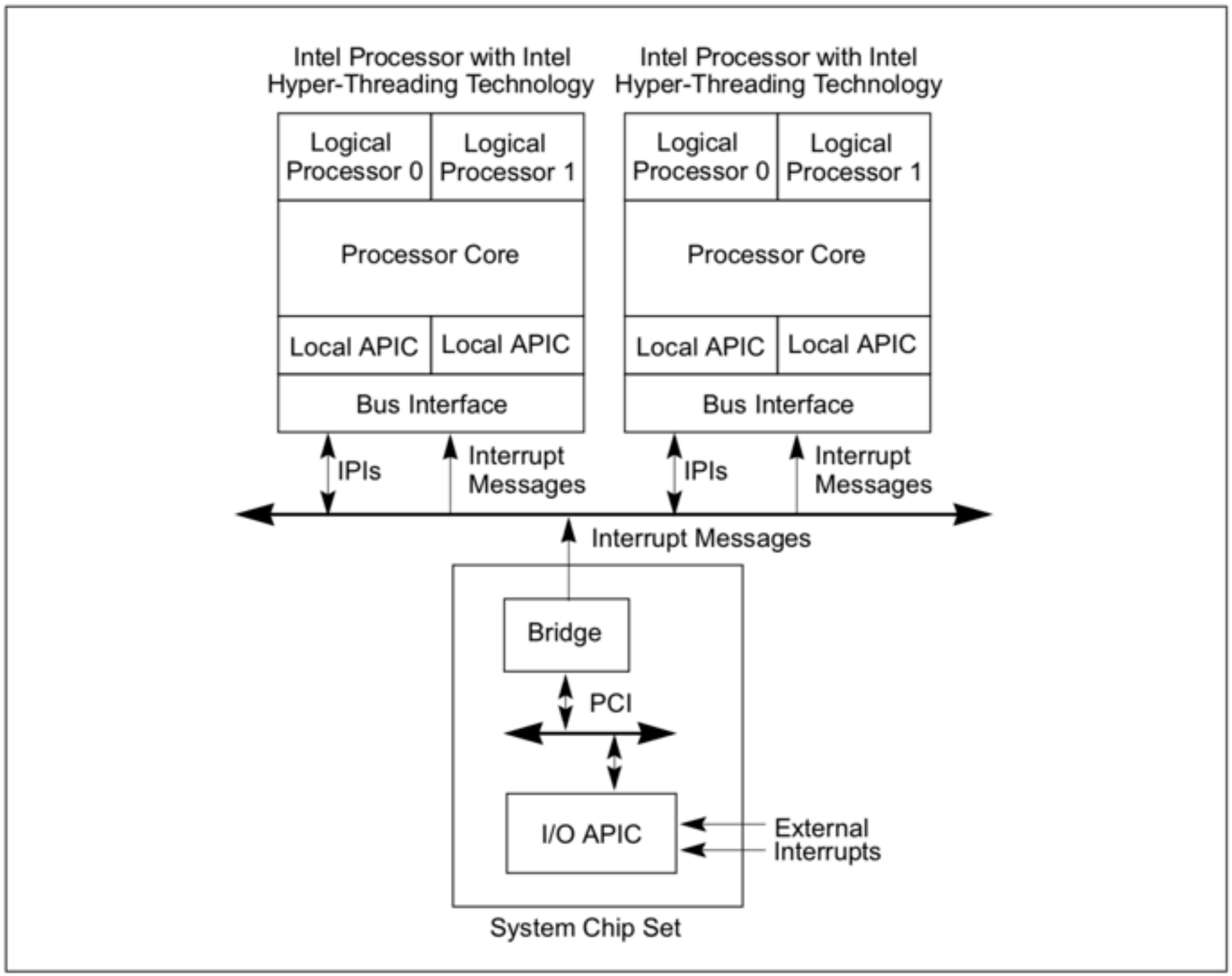
APIC：高级可编程中断控制器

APIC：Advanced Programmable Interrupt Controller，主要面向多核系统
APIC = Local-APIC ＋ I/O-APIC



APIC：高级可编程中断控制器

如果使用超线程技术，则每个逻辑内核一个Local-APIC



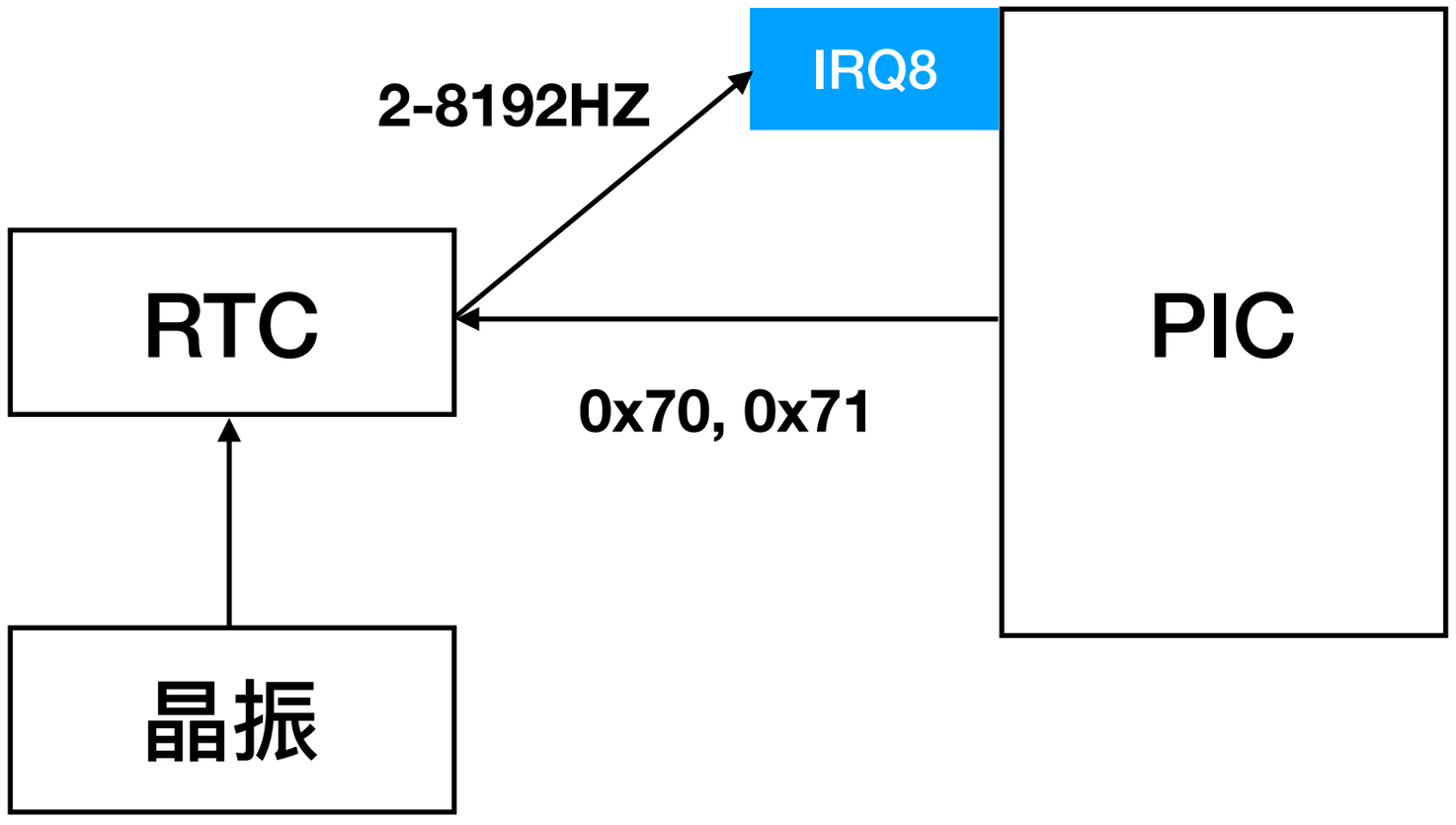
APIC：多核问题

APIC技术，可以让多个核心相互通信（IPI），可以支持多个核心共享一组I/O设备
这样在核心通信和IO设备共享层面，可以支持核心数量的任意扩展。

但是，由于多个核心共享北桥内存总线，同一时间内存总线只能给一个核心使用。
在核心数量扩张的时候，共享的内存总线就会成为系统瓶颈。

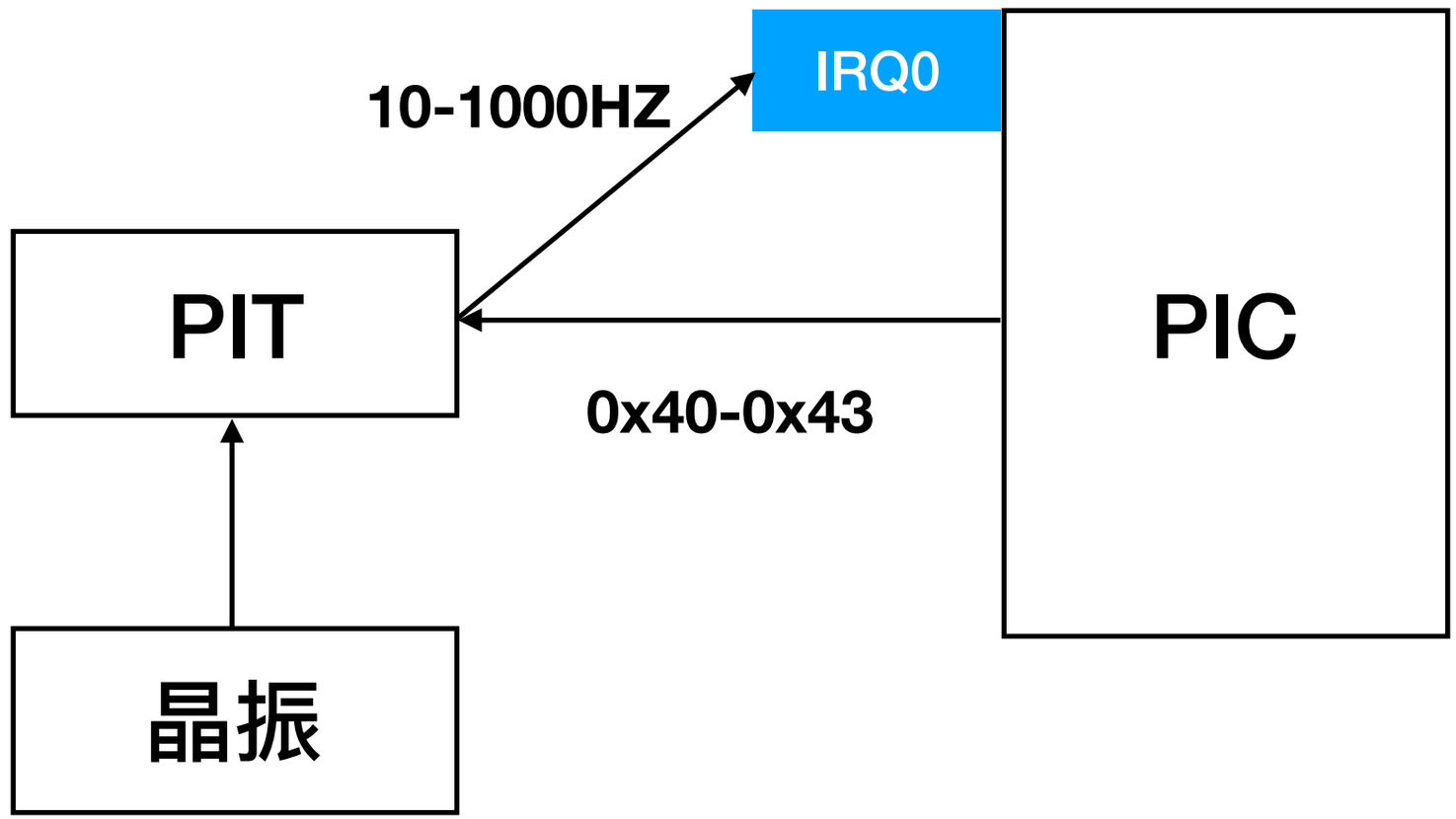
时钟：RTC

RTC：Real Time Clock，实时时钟，用于时间日期记录，有电池可断电
在确定的时间点发出中断（类比闹钟）



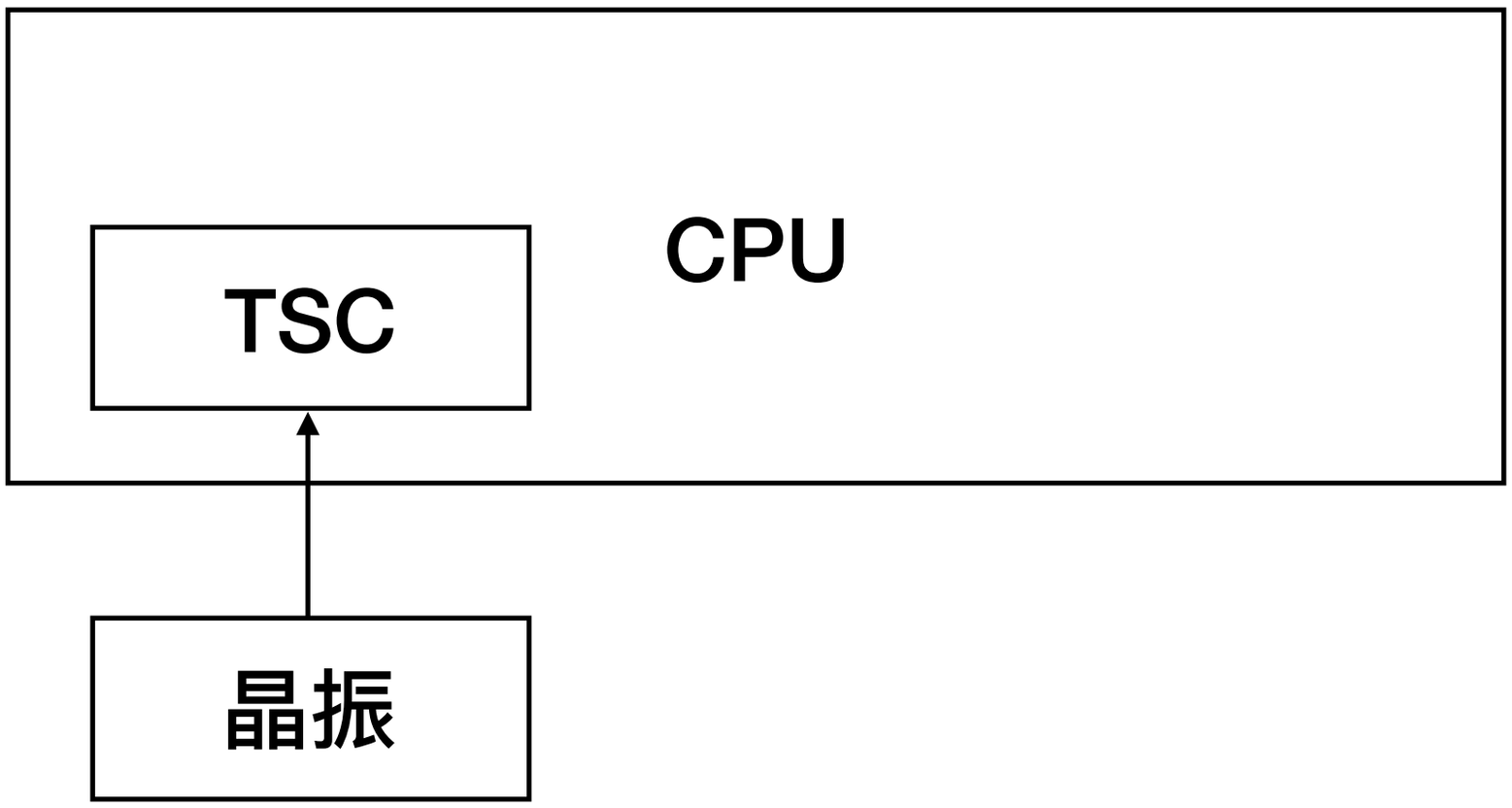
时钟： PIT

PIT：Programmable Interval Timer，可编程间隔定时器，用于产生一个固定时间间隔的中断信号（类比计时器）



时钟：TSC

TSC：Time Stamp Counter，时间戳计数器，用于记录自系统启动以来的CPU周期数（Cycle）。TSC中的时间间隔是CPU可感知的最小时间间隔。



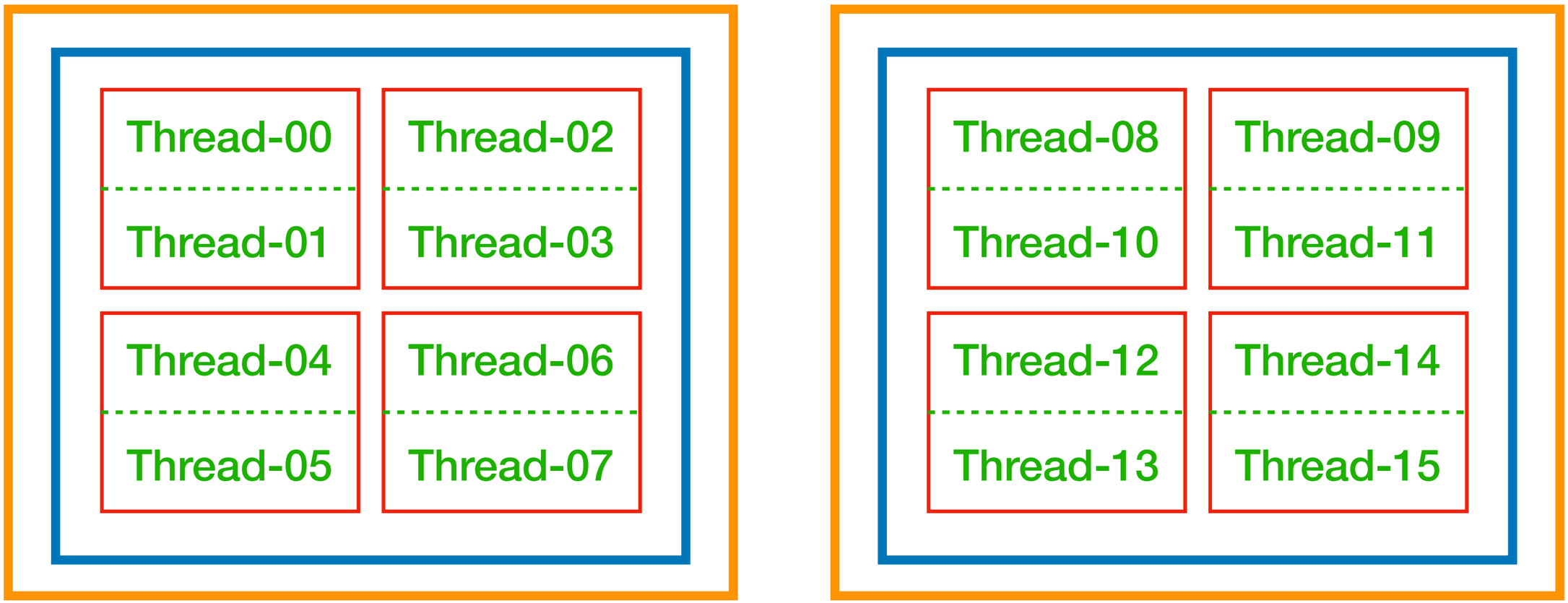
时钟：应用

Crontab、date 通过RTC实现

sleep()、usleep()、nanosleep() 通过PIT+TSC的组合来实现，使用PIT时，进程挂起，使用TSC时不挂起。

Linux使用PIT调用int-0x80中断，陷入系统进程，进行调度处理

多核架构：基本概念



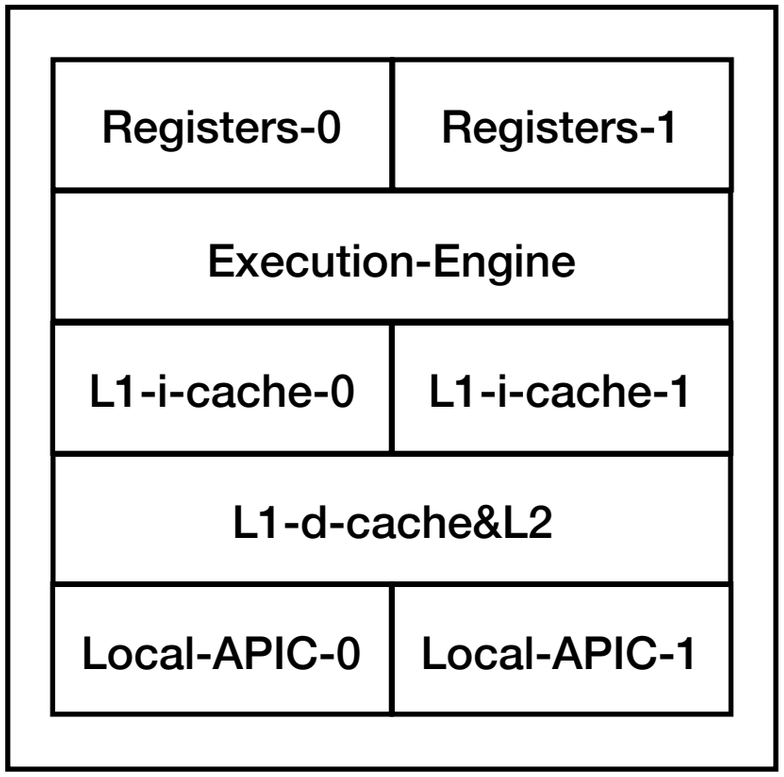
Thread：逻辑核心（CPU）

Numa Node：Numa节点

Core：处理器核心

Socket：插槽

多核架构：Thread-Core



CORE

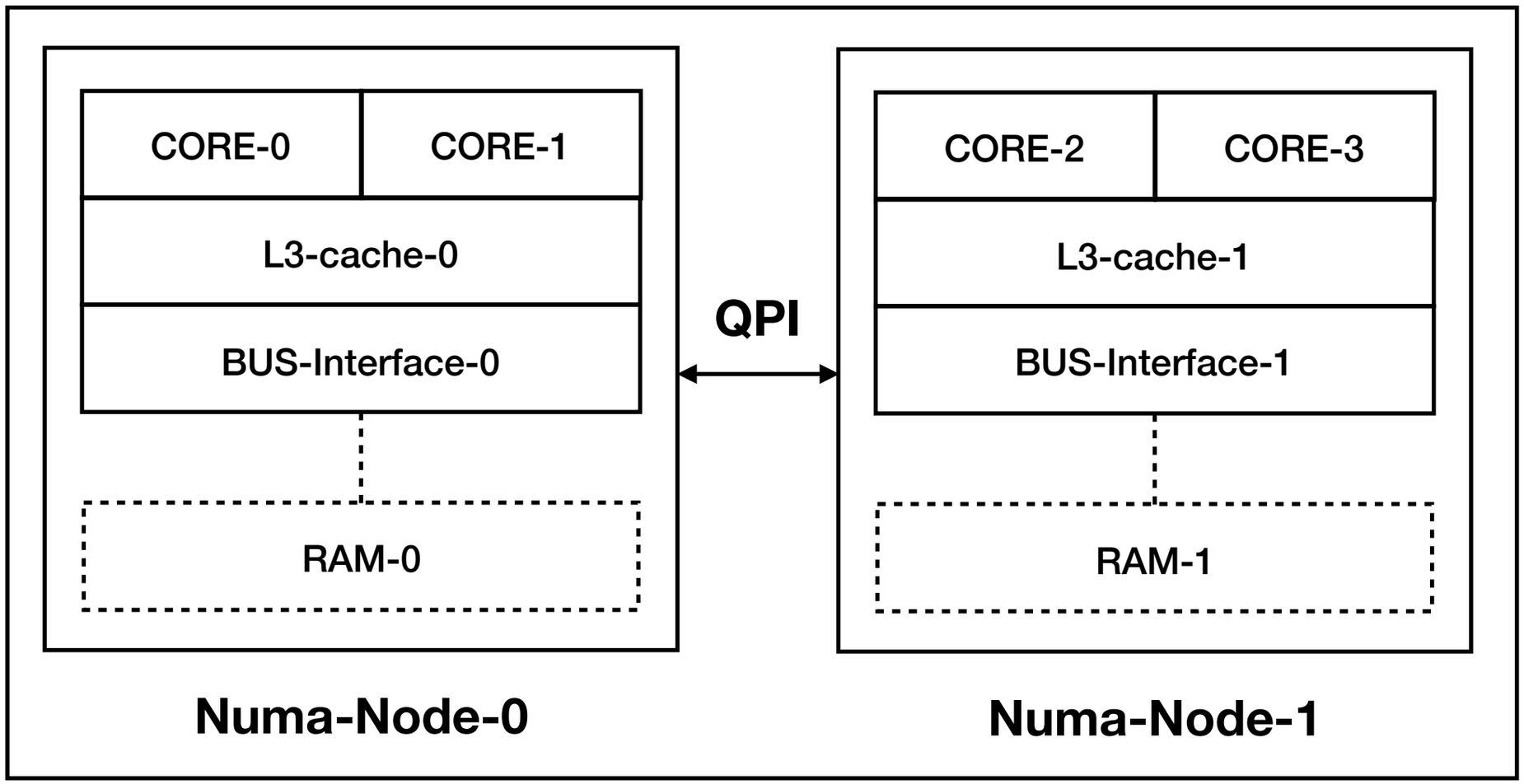
Intel的超线程技术（HT）
也叫同步多线程（SMT）技术

将一个物理内核（Core）
模拟成两个逻辑内核（Thread）

每个Thread有自己的：
寄存器组、Local-APIC和i-cache

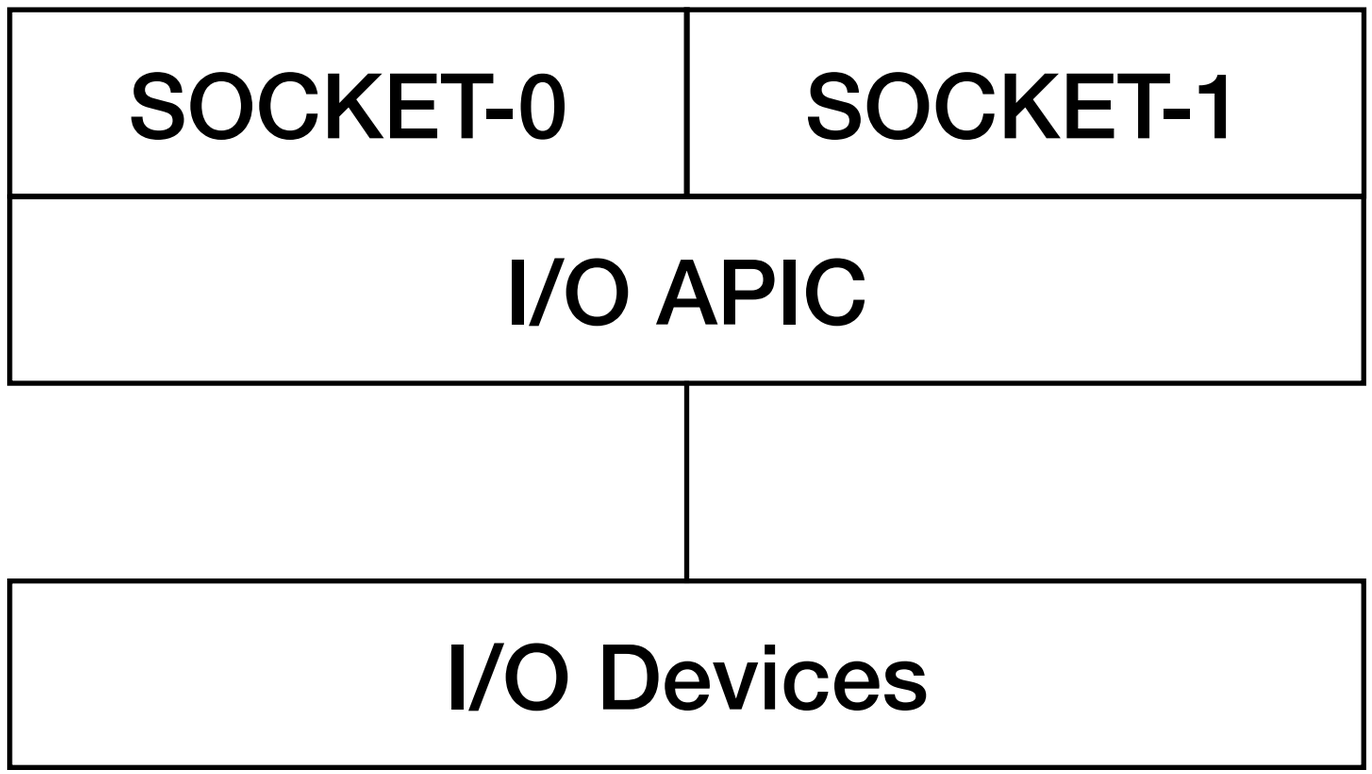
两个Thread共享：
L1-d-cache、L2-cache、CPU执行引擎

多核架构：Core-NumaNode-Socket



同一个Numa-Node上的Core共享：L3-cache、系统总线、内存
物理内存平均分给不同的Node
同一个Socket上的不同Numa-Node有各自的：L3-cache、系统总线、内存
CPU访问自己Node的内存快，访问别的Node的内存慢（约2-3倍）

多核架构：Socket-I/O



所有socket共享I/O-APIC

多核架构：三层架构

SMP

Symmetrical Multi-Processing
对称多处理架构，多个CPU共享内存总线

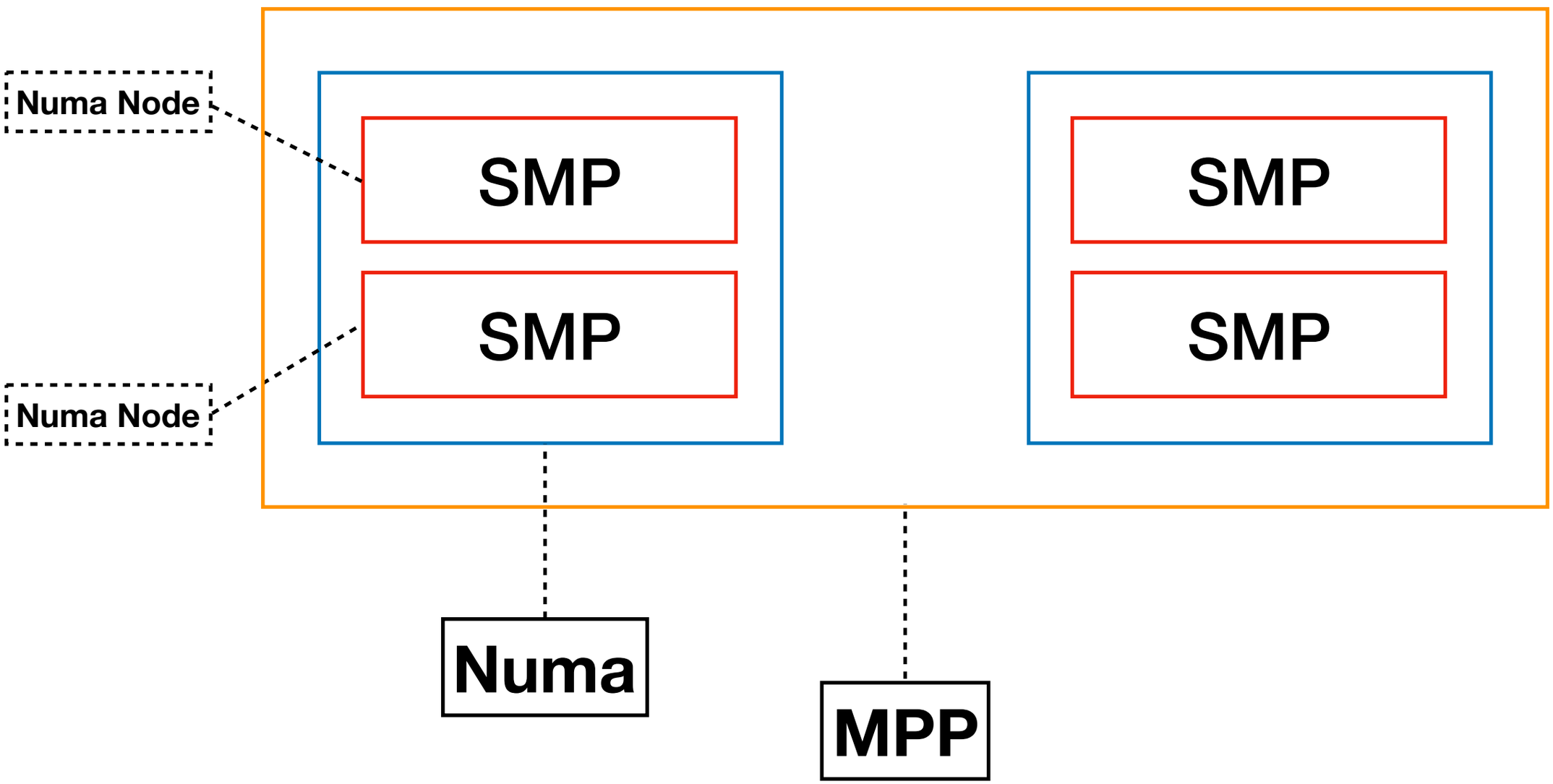
NUMA

Non-Uniform Memory Access
非一致内存访问架构
CPU分成多个Node组，每个Node有自己的内存总线

MPP

Massive Parallel Processing
大规模并行处理架构
多个服务器通过高速局域网连接在一起

多核架构：三种架构



多核架构：SMP-NUMA实例

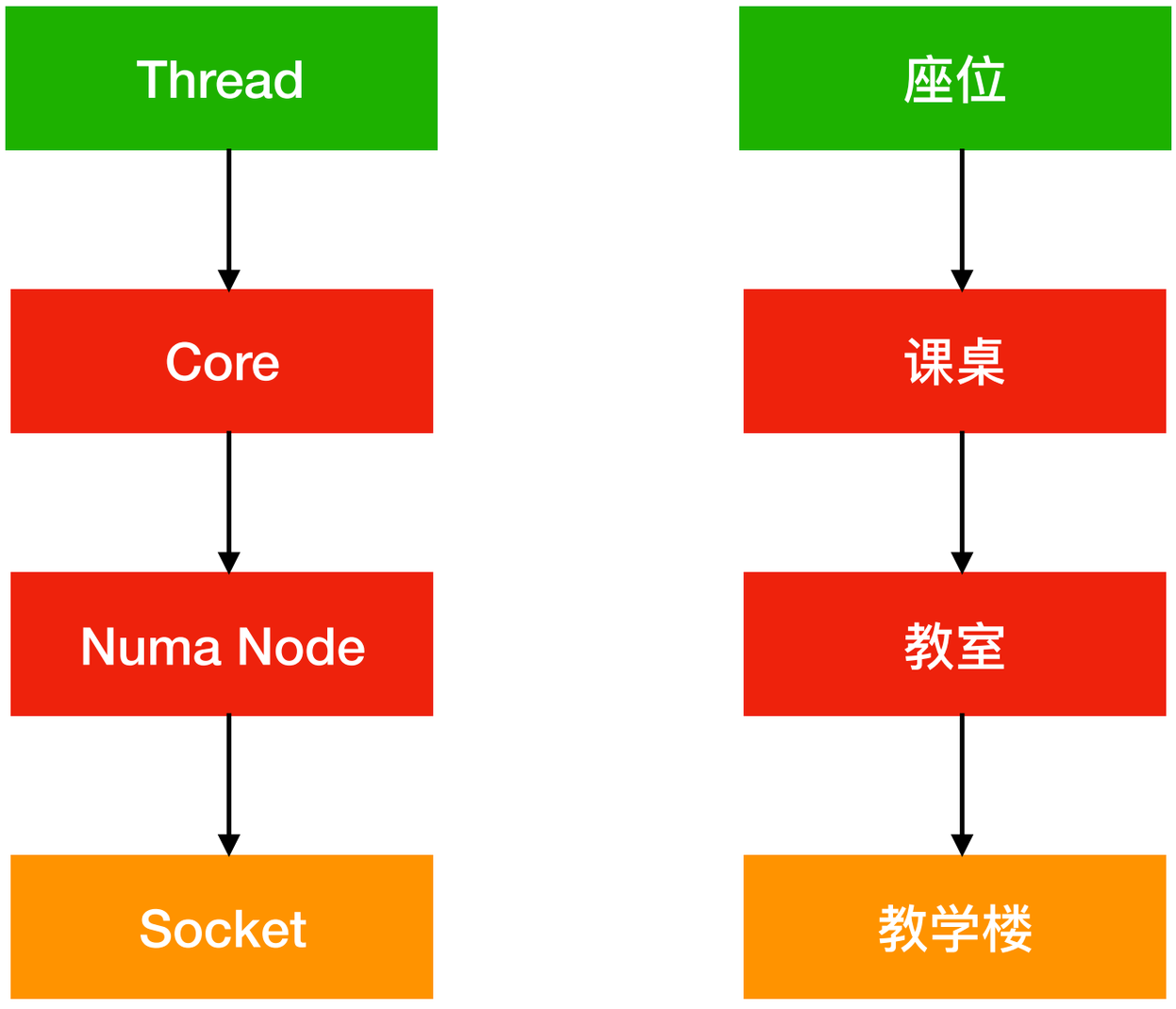
```
[@bx.49.182 /var/log]# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
Stepping:              4
CPU MHz:               1199.960
BogoMIPS:              5205.26
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
```

每个Core上两个Thread
每个NUMA-Node上8个core
每个Socket上一个NUMA-Node

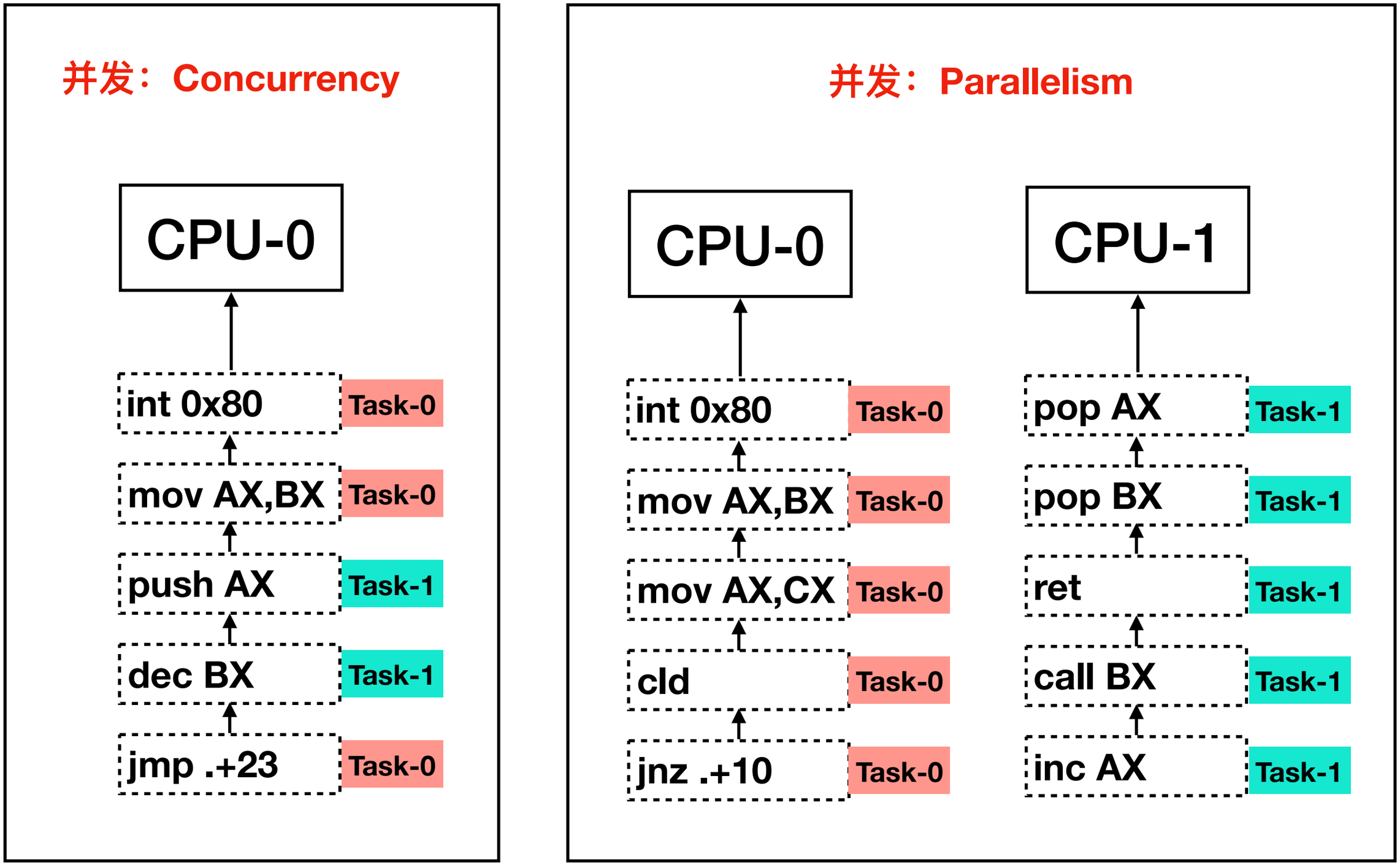
```
[@bx.49.182 /var/log]# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
node 0 size: 32722 MB
node 0 free: 10313 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
node 1 size: 32768 MB
node 1 free: 27415 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

不同NUMA-Node对应不同的物理内存区域，在Linux的实现中，CPU优先使用自己Node分配的物理内存，导致不同Node的物理内存使用分配不均匀

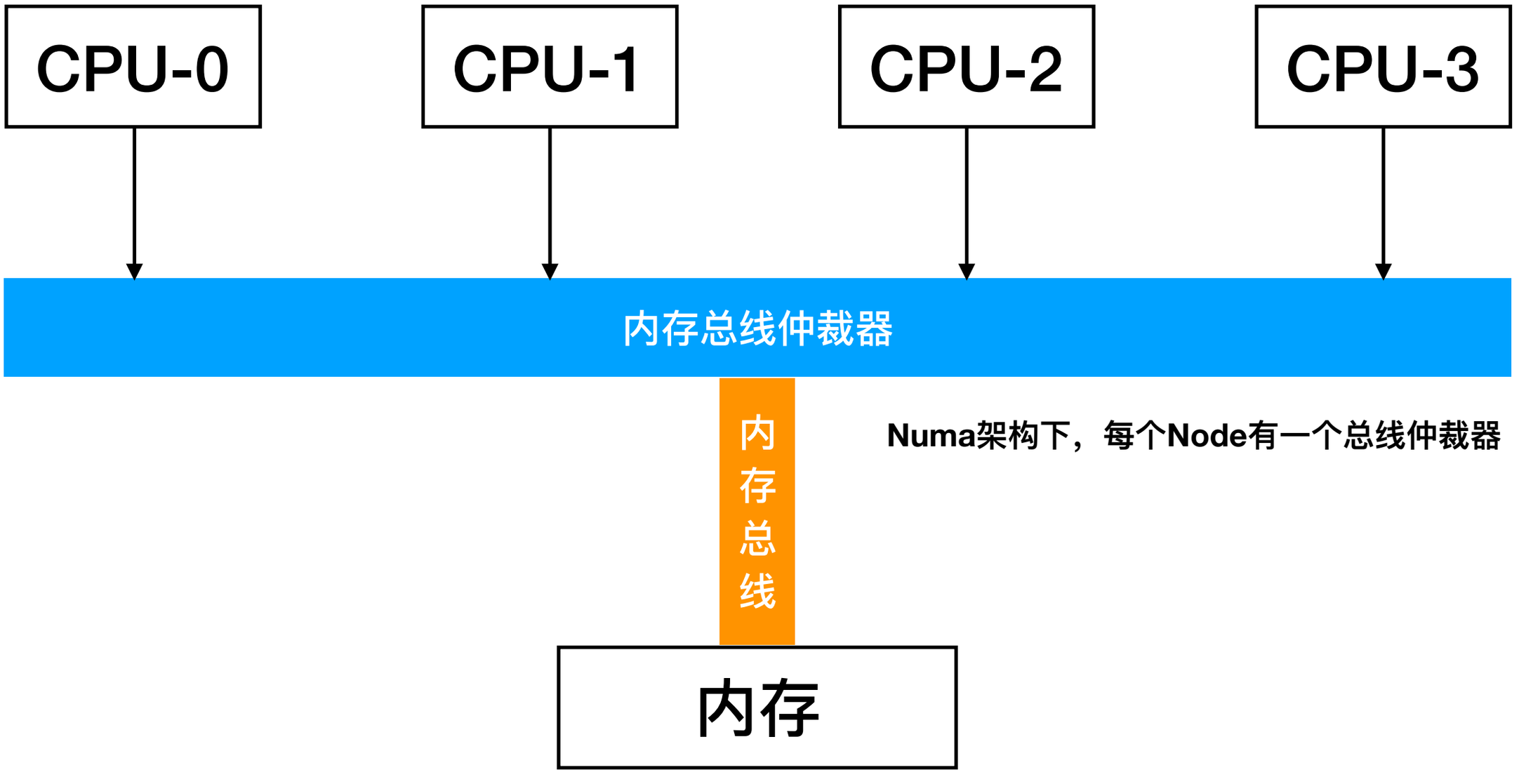
多核架构：类比



同步：并发与并行


并发：Concurrency
CPU-0
int 0x80
Task-0
mov AX,BX
Task-0
push AX
Task-1
dec BX
Task-1
jmp .+23
Task-0
并发：Parallelism
CPU-0
int 0x80
Task-0
mov AX,BX
Task-0
mov AX,CX
Task-0
cld
Task-0
jnz .+10
Task-0
CPU-1
pop AX
Task-1
pop BX
Task-1
ret
Task-1
call BX
Task-1
inc AX
Task-1

同步：基本问题



Numa架构下，每个Node有一个总线仲裁器

多个CPU在并行过程中按照规则和顺序访问内存，确保边际效应的影响与单CPU运行时一致

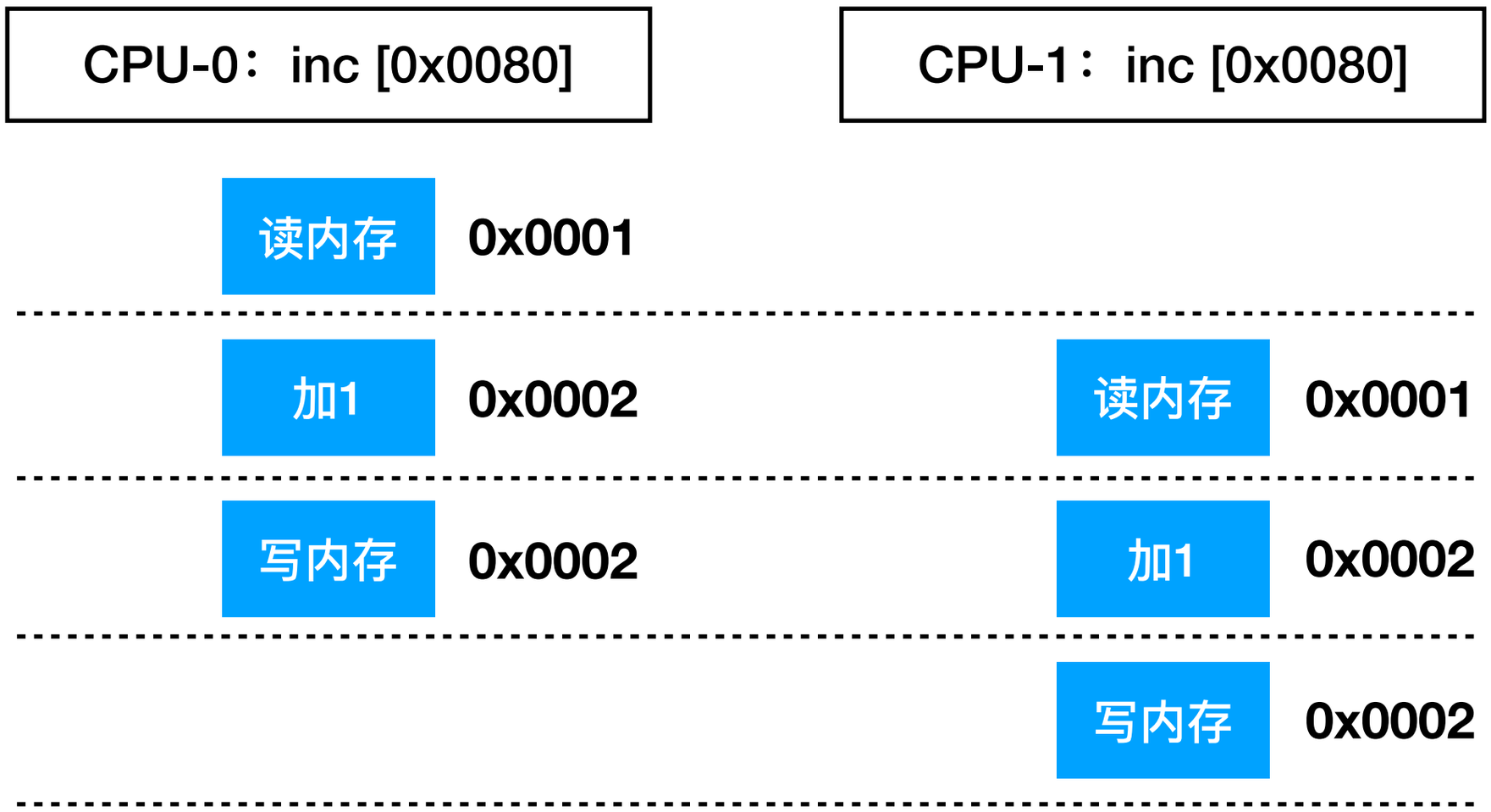
单核系统在CPU层面不存在同步问题

同步：三类多核同步问题及其解决方案

单指令多总线请求问题	原子操作
缓存可见性问题	MESI协议
执行期重排序问题	内存屏障

同步：单指令多总线请求问题

对于ADD、DEC、INC、SUB、NOT、OR等指令，当操作数为内存地址时，单条指令会访问总线两次，读内存—>修改—>写回内存，在多核情况下会出现数据不一致。



同步：原子操作

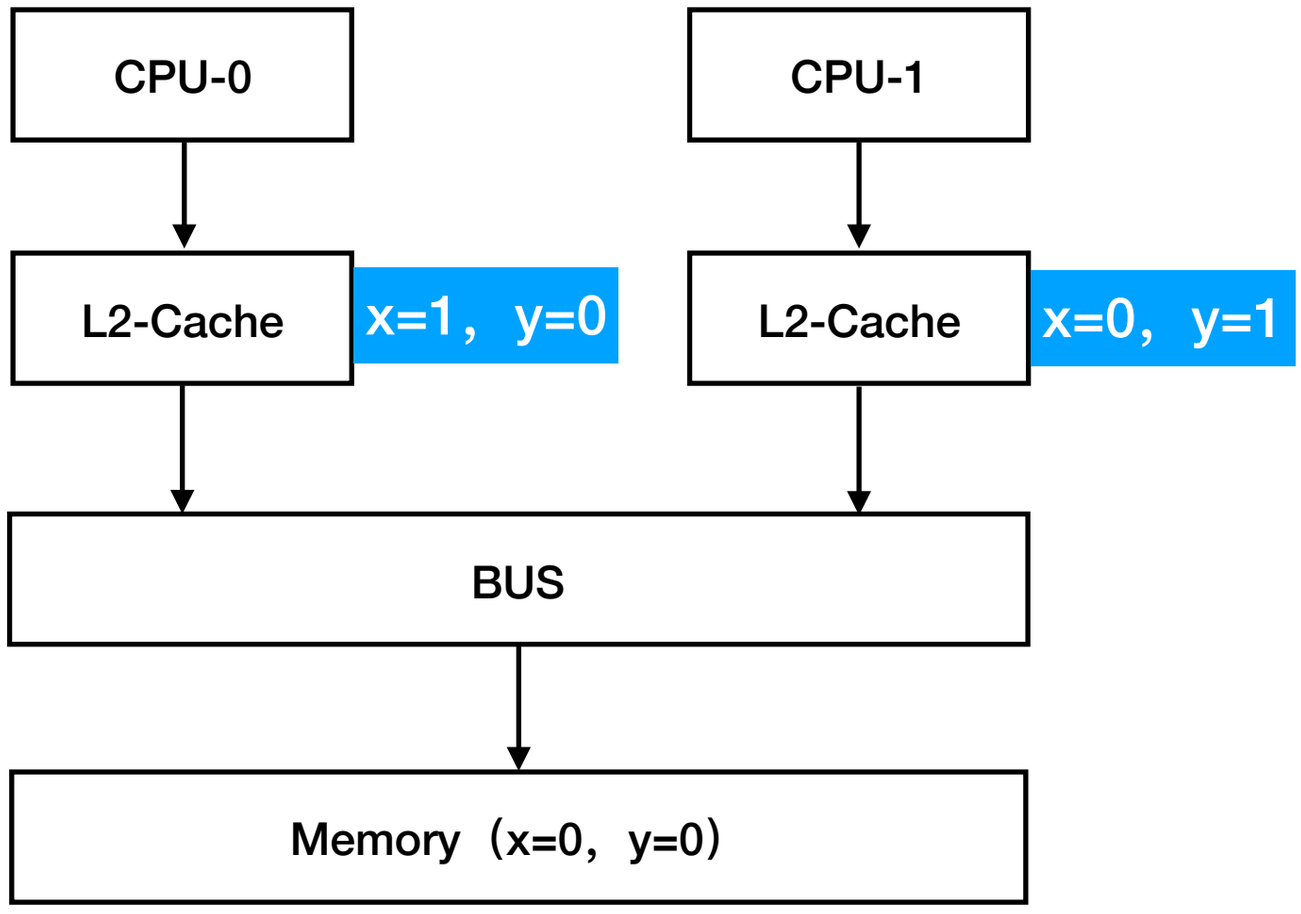
对于ADD、DEC、INC、SUB、NOT、OR等指令，当操作数为内存地址时，指令之前加lock前缀，可以确保指令在多核竞争总线的情形下的原子性。

lock inc [0x0080]，确保该inc指令在执行过程中独占总线，即inc指令从执行开始到执行结束的时间内，只有该inc指令影响内存。

在C语言中，不能保证a=a+1、a++这类写法在编译时自动增加lock前缀，在多核状态下执行时有可能出现问题。

C或者Java语言中，在变量之前加volatile修饰符，汇编层面本质就是对改变量的汇编操作语句之前加lock

同步：缓存可见性问题



多核环境下，由于不同cpu有不同的2级缓存，造成在同一时间周期内，
不同cpu看到同一内存数据的不同副本

同步：MESI协议

MESI：Modified Exclusive Shared Or Invalid（多核CPU本地缓存中数据的四种状态）

Modified	被修改的 数据只在当前CPU本地缓存中，并且与主存不一致
Exclusive	独享的 数据只在当前CPU本地缓存中，并且与主存一致
Shared	共享的 数据在多个CPU的本地缓存中，并且与主存一致
Invalid	无效的 数据在多个CPU的本地缓存中，其他CPU修改了该数据 则当前CPU的本地缓存中的该数据处于无效状态

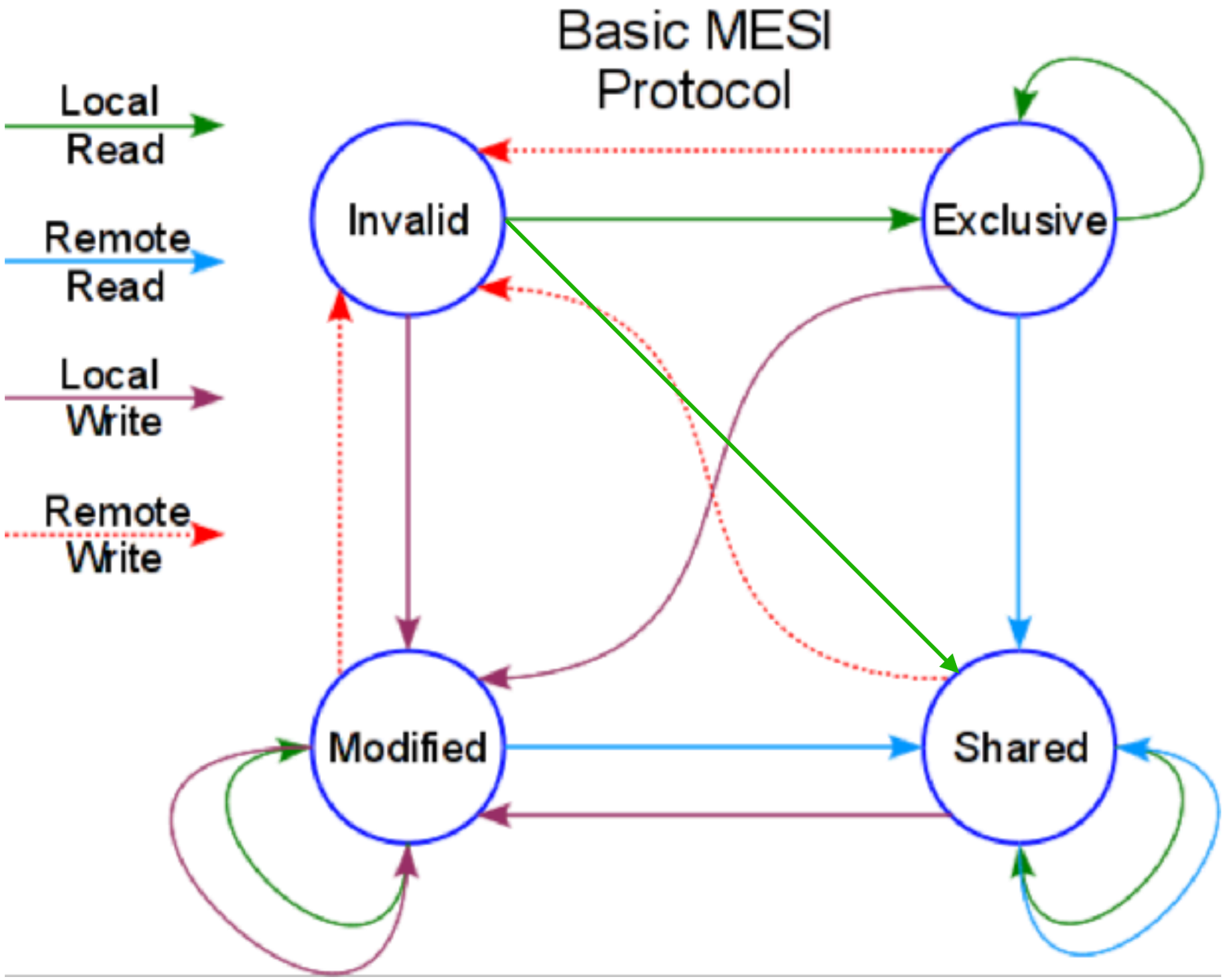
同步： MESI协议

当前CPU读

其他CPU读

当前CPU写

其他CPU写



同步：执行期重排序问题

程序定义

全局变量：a=0，b=0，x=0，y=0
线程A：a=1; x=b;
线程B：b=1; y=a;

单核环境下

如果x=b在y=a之前，则a=1在y=a之前
如果x=b在y=a之后，则b=1在x=b之前
则线程A和B执行完之后，x和y中至少有一个为1

多核环境下

线程A和B完全独立
总线上会出现以下执行序列：
线程A：x=b;
线程B：b=1;
线程B：y=a;
线程A：a=1;

在此执行序列下：x=0且y=0
脱离程序语义预期

重排序问题：多核环境下，由于cpu自身的指令乱序优化功能，造成不同cpu上访问同一内存数据的线程，在执行后脱离语义预期

同步：内存屏障

使用内存屏障（memory barrier）解决CPU运行期的重排序问题

LFENCE	LFENCE指令确保之前的读取内存的指令，在LFENCE指令执行之前都已经完成
SFENCE	SFENCE指令确保之前的写入内存的指令，在SFENCE指令执行之前都已经完成
MFENCE	MFENCE指令确保之前的读取和写入内存的指令，在MFENCE指令执行之前都已经完成

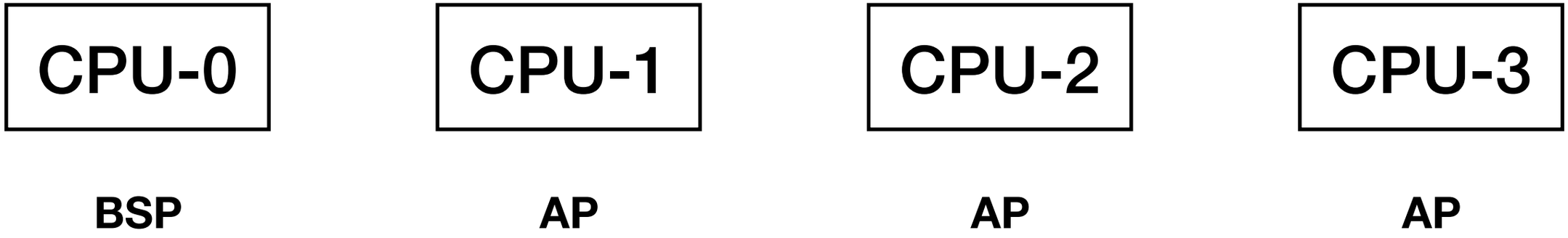
线程A改为：a=1; sfence; x=b;　　线程B改为：b=1; sfence; y=a;

读写I/O端口操作和lock前缀指令，也可以起到mfence指令一样的作用

多核下的CPU初始化：BSP与AP

BSP：the Bootstrap Processor

AP：the Application Processor



多核下的CPU初始化：BSP与AP

1、接电
2、每个CPU获得一个唯一的APIC-ID标志
3、每个CPU进行电路自检
4、从CPU选出BSP
5、BSP执行BIOS代码
6、BSP把TSC初始化为1
7、BSP向总线上所有AP广播
8、各个AP串行的依次执行BIOS代码
9、BSP开始执行操作系统代码

回顾

二进制

整数表示　浮点数表示

二级管　MOS管

时序电路

门电路　RS触发器

RS锁存器　D触发器

流水线作业

微架构　寄存器

IPC　多任务系统

多级存储

数据冒险　乱序执行

控制冒险　局部性原理

寻址

实模式　保护模式

分段　分页

中断

GDT　TSS

LDT　IDT

任务

CPL　段选择符

门描述符　段描述符

同步

IRQ　SMP

APIC　NUMA

内存屏障

答疑