第二章

并发进程

主 要 内 容

2.1 进程基本概念

2.2 处理机调度与死锁

2.3 UNIX的进程

2.4 中断的基本概念及UNIX中断处理

2.5 进程通信

中断的基本概念

CPU不可能时刻查询外设的工作状态。。。

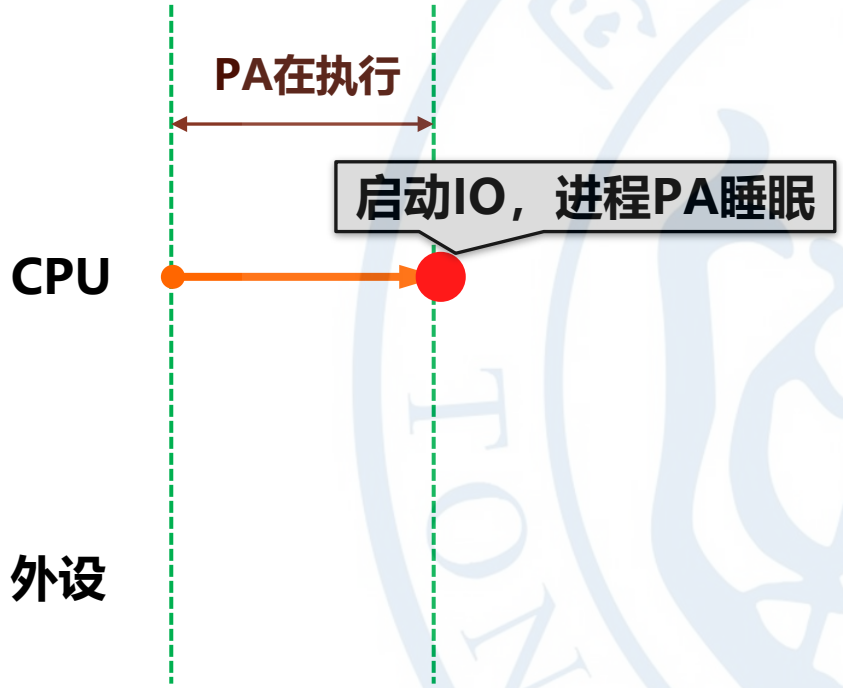
一种设备控制方式

一种外设的数据传输方式

中断的基本概念

CPU不可能时刻查询外设的工作状态。。。

一种设备控制方式　　一种外设的数据传输方式

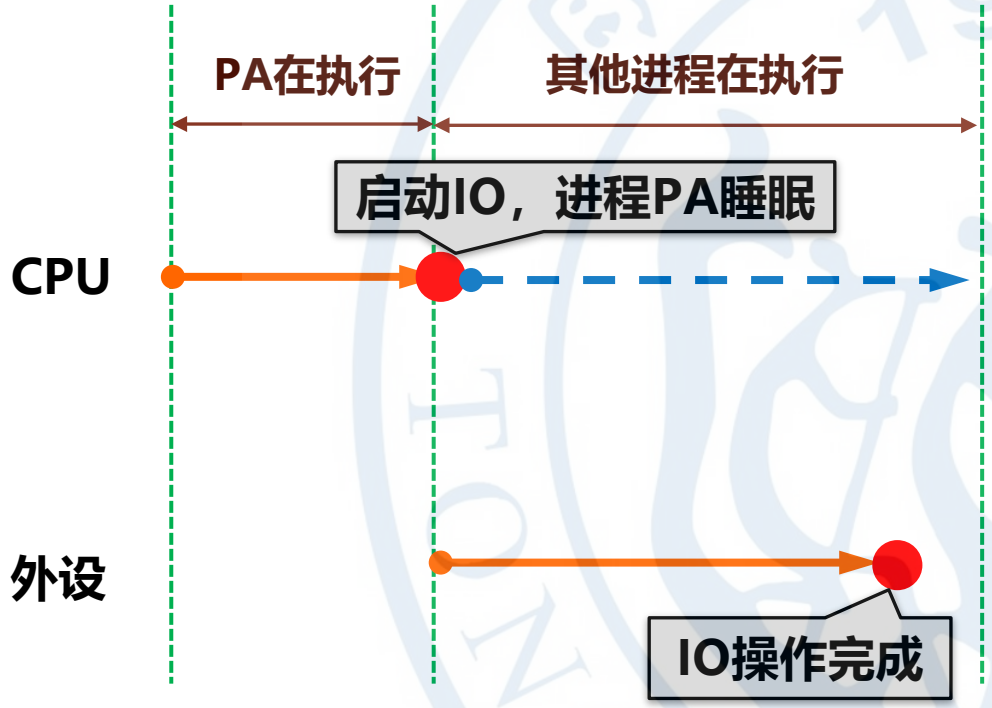


中断的基本概念

CPU不可能时刻查询外设的工作状态。。。

一种设备控制方式

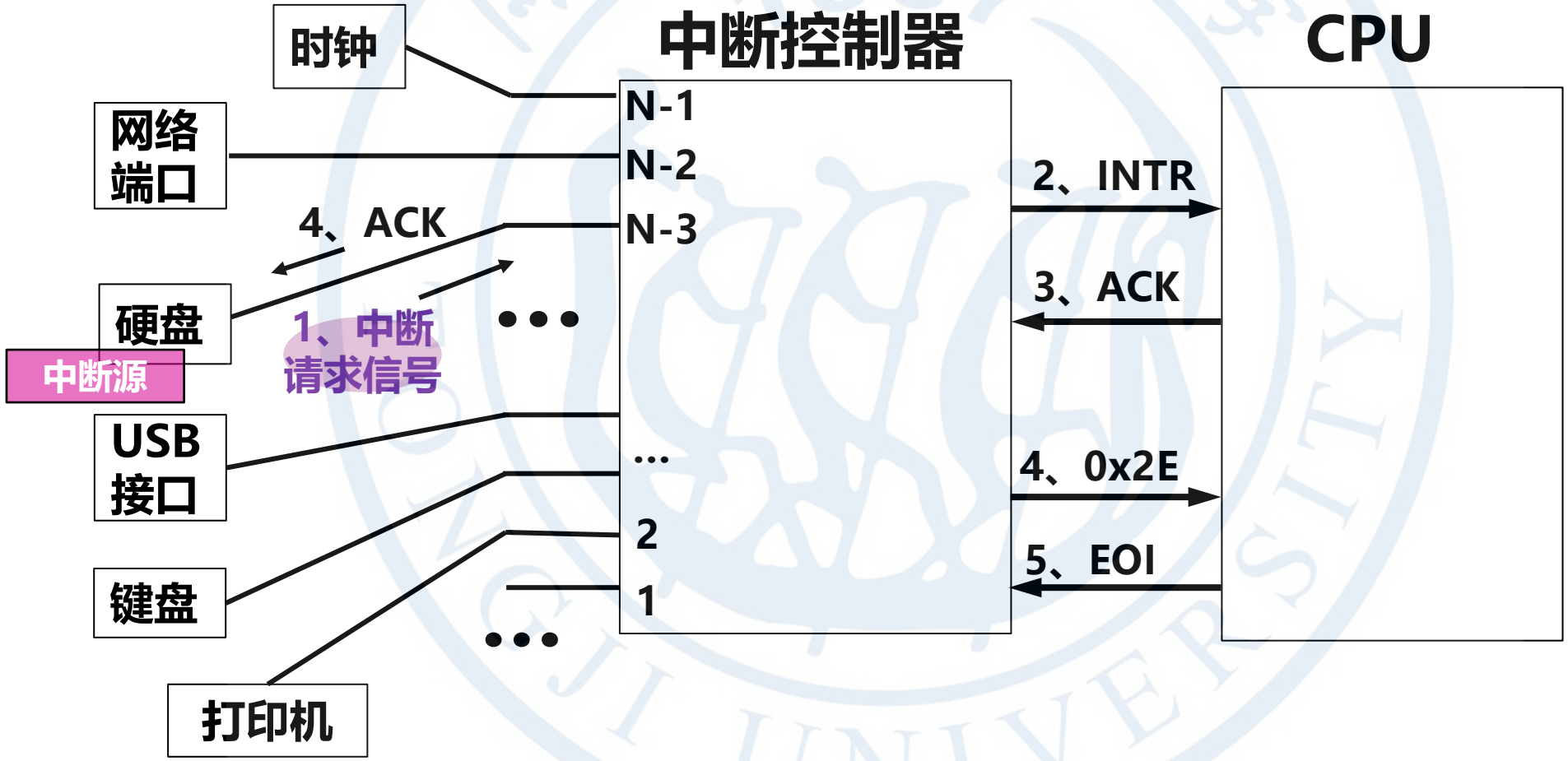
一种外设的数据传输方式



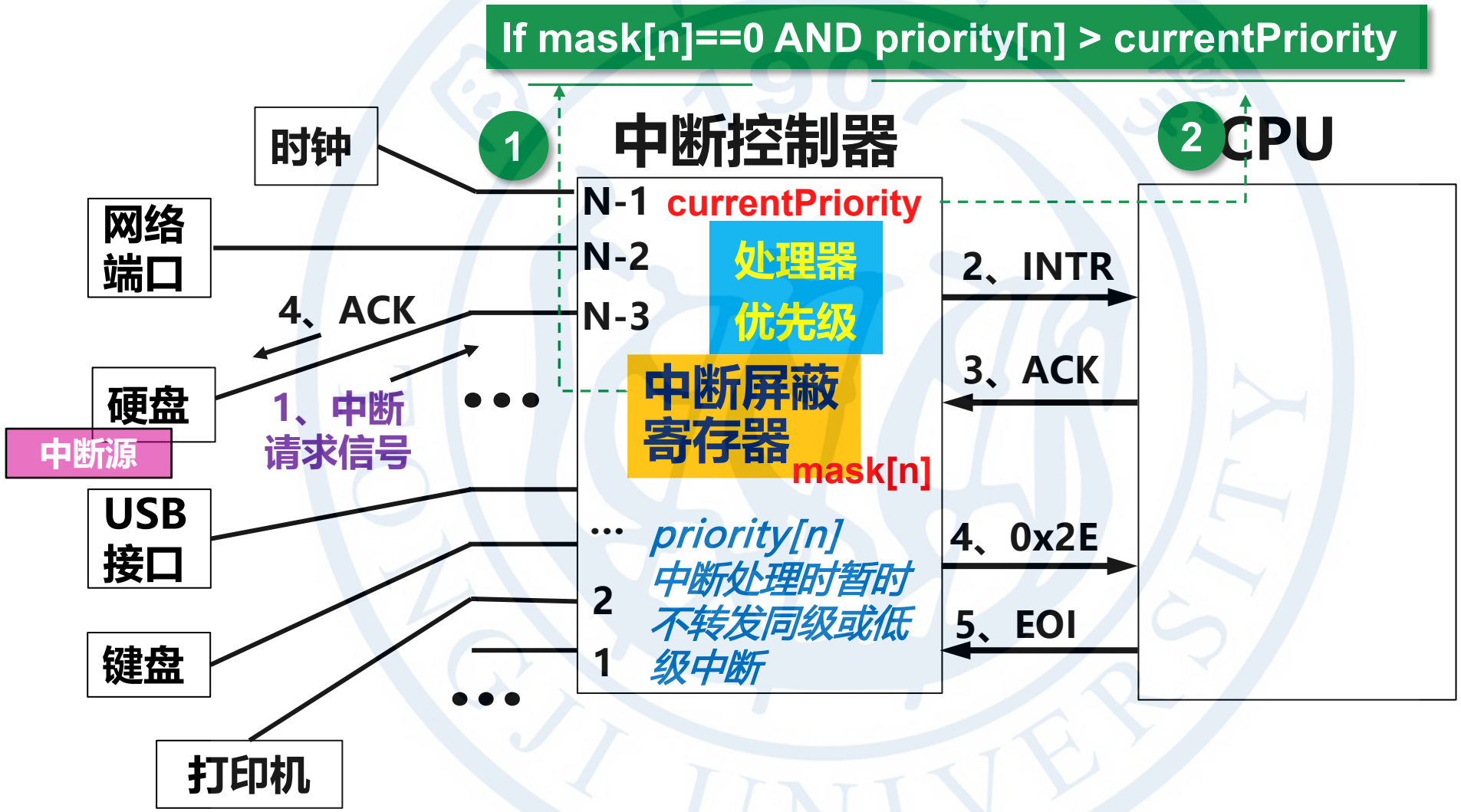
Q1>. CPU怎么知道外设的操作结束了?

中断的基本概念

1. I/O结束后，设备向中断控制器发出中断请求。中断控制器决定是否向CPU转发该请求。



中断的基本概念


If mask[n]==0 AND priority[n] > currentPriority
中断仲裁
时钟
网络端口
硬盘
中断源
USB接口
键盘
打印机
1
中断控制器
2
CPU
N-1
currentPriority
N-2
N-3
处理器优先级
中断屏蔽寄存器
mask[n]
2
1
priority[n]
中断处理时暂时不转发同级或低级中断
1、中断请求信号
2、INTR
3、ACK
4、ACK
4、0x2E
5、EOI

中断的基本概念

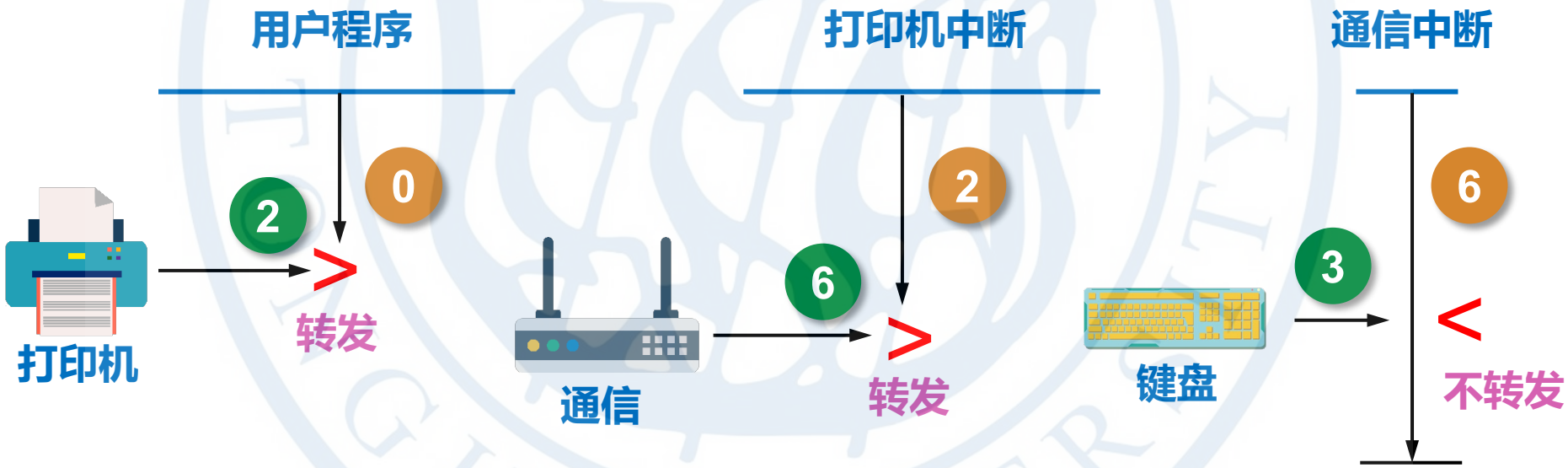
中断优先级

对不同的中断源，按重要性、紧迫程度分不同等级，并用一个正整数表示

处理机优先级

反映CPU正在执行的中断处理的优先级

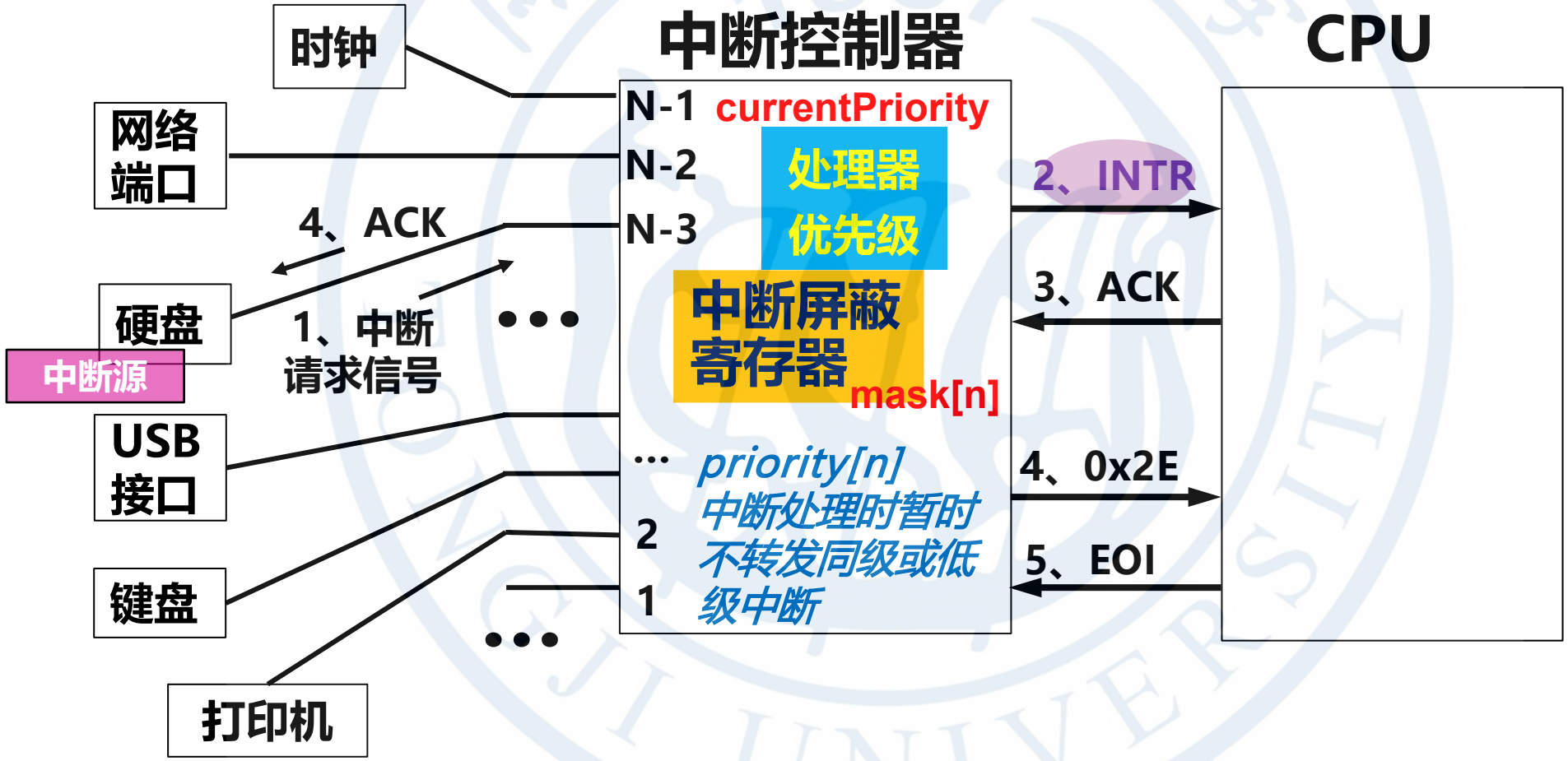
未执行中断服务子程序时：0



中断控制器只向CPU转发更高级的中断请求

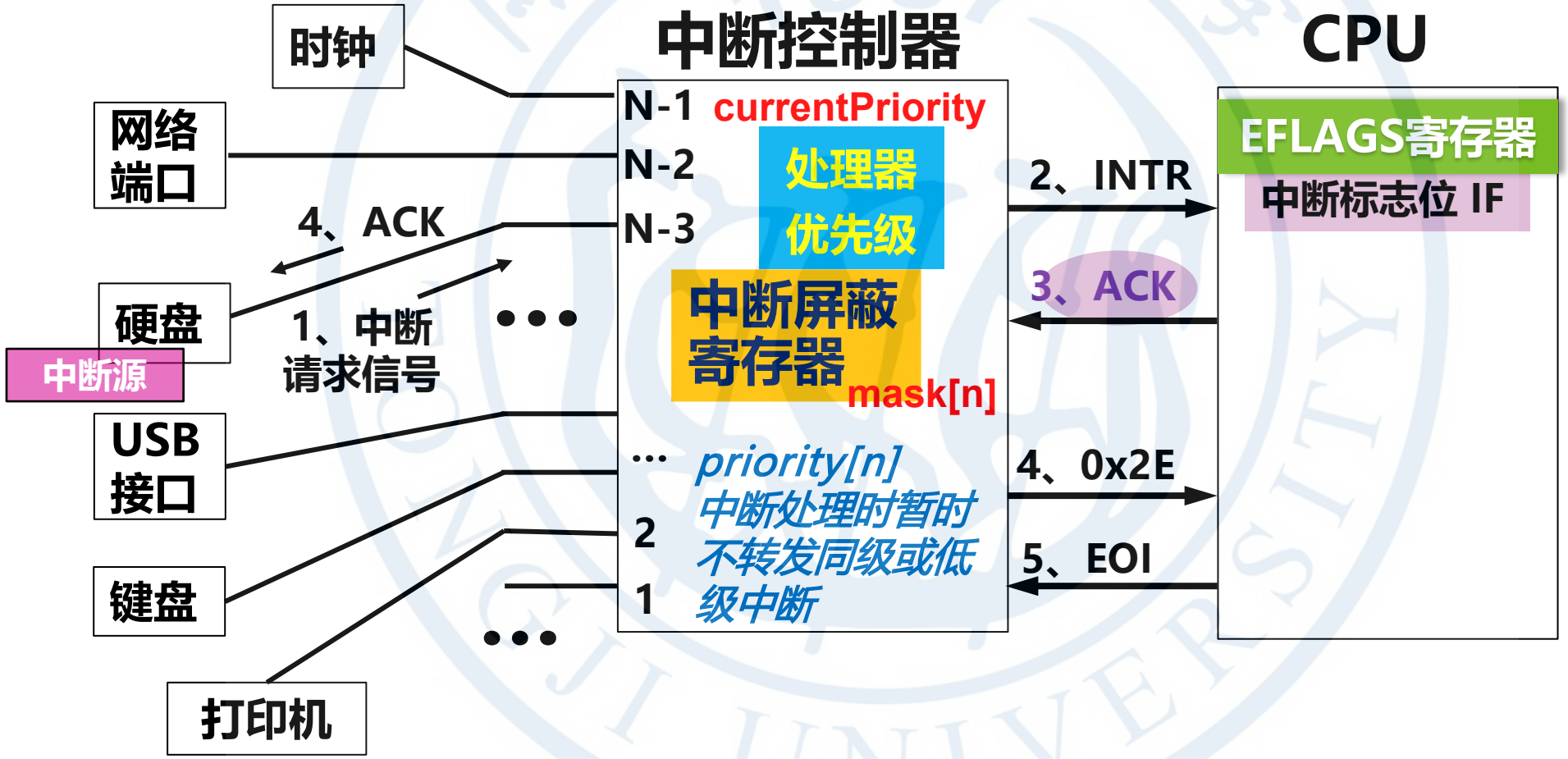
中断的基本概念

2. 仲裁通过，中断控制器置INTR连线为高电平通知CPU系统发生了中断。



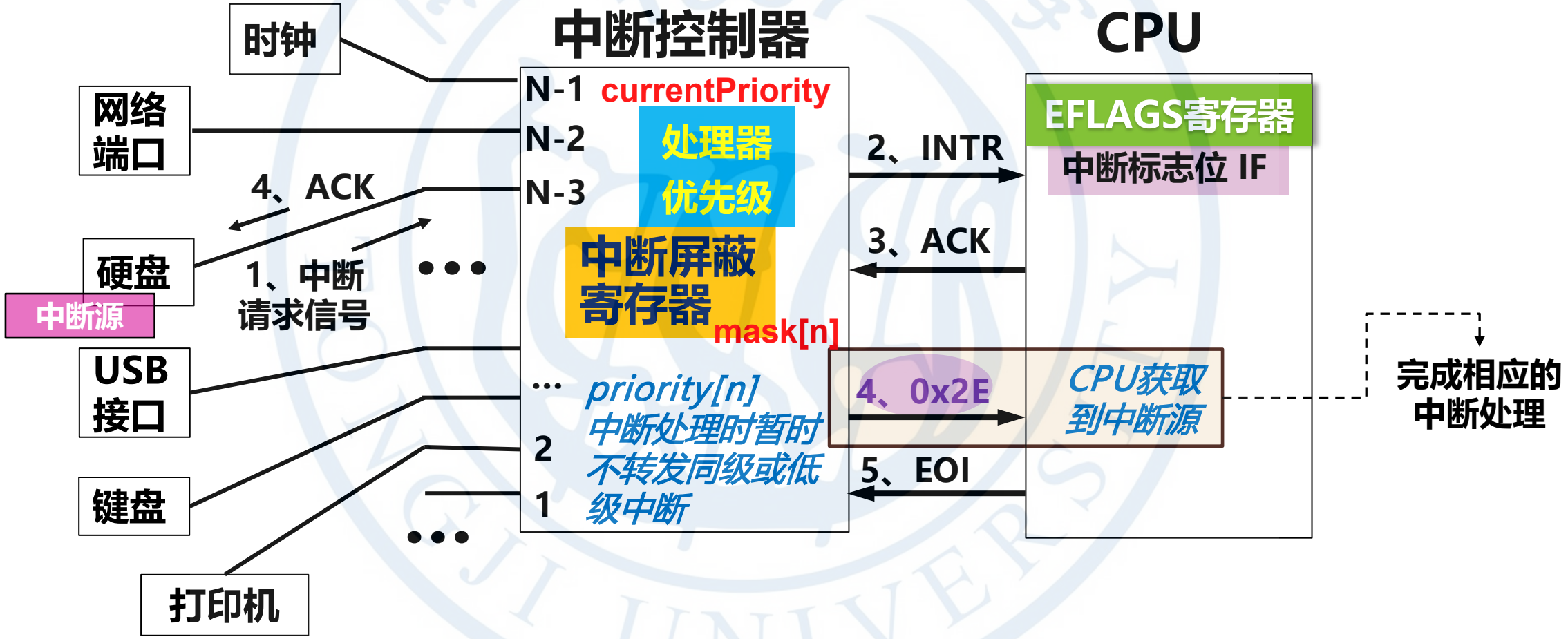
中断的基本概念

if IF==1 → 3. 处理器在每条指令执行完毕后，立即检查INTR管脚。如果有中断请求，则立即回送ACK信号。

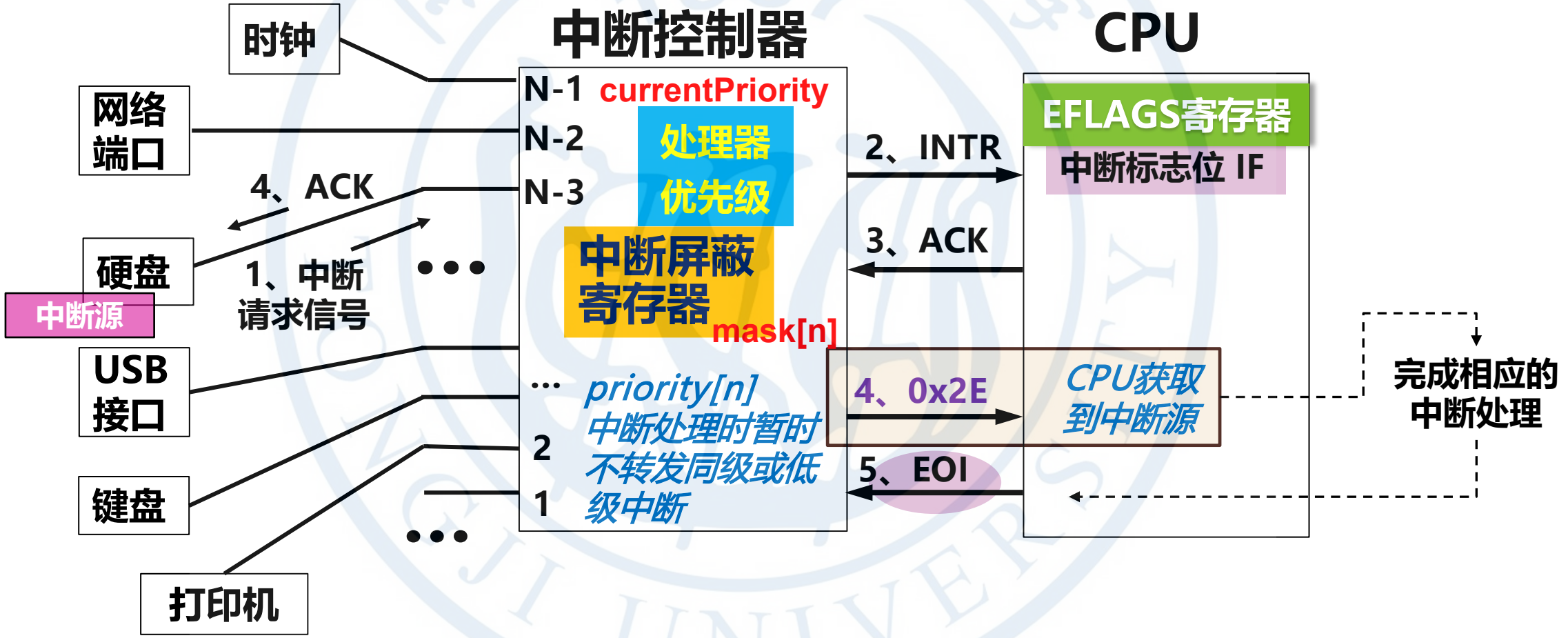


中断的基本概念

4. 得到ACK信号的中断控制器送中断号给CPU。清除INTR请求信号，向外设发送ACK信号表示中断请求已被处理。



5. CPU完成中断处理后，向中断控制器发送EOI。



中断的基本概念

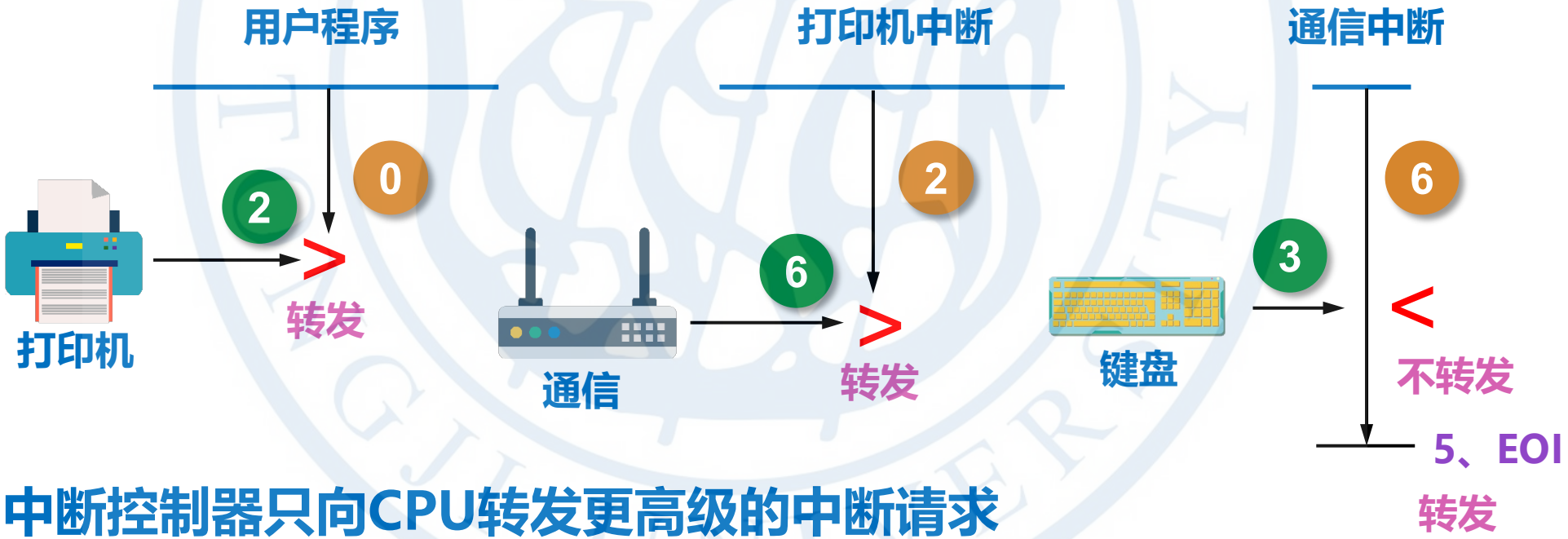
中断优先级

对不同的中断源，按重要性、紧迫程度分不同等级，并用一个正整数表示

处理机优先级

反映CPU正在执行的中断处理的优先级

未执行中断服务子程序时：0

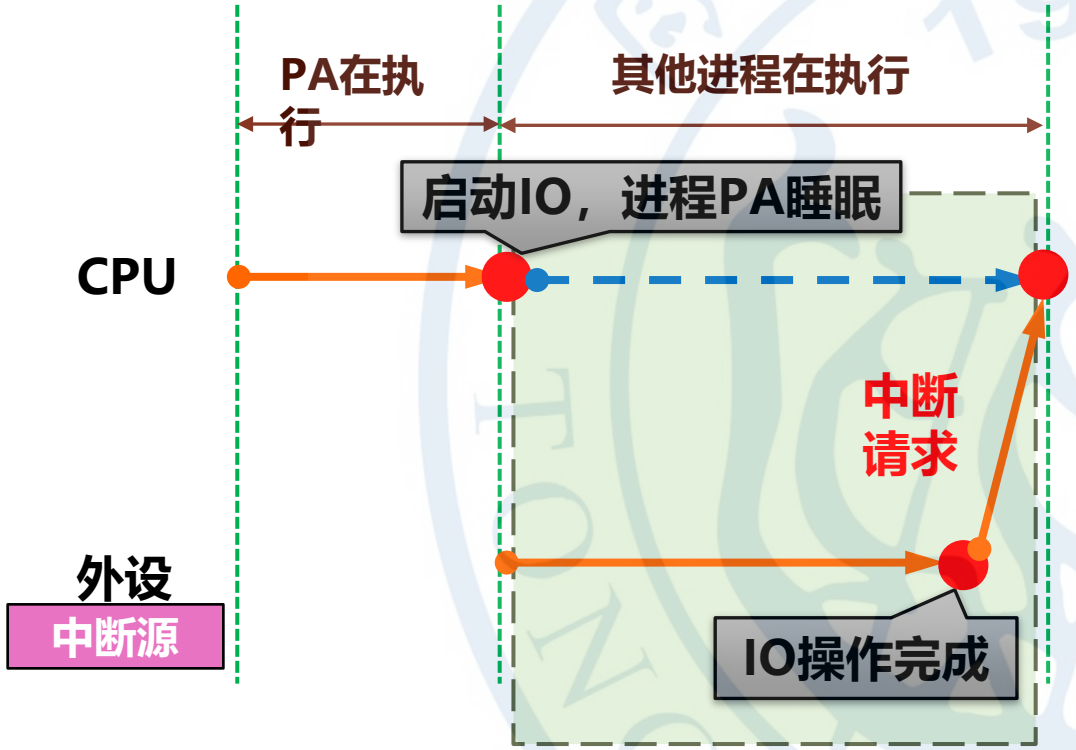


中断控制器只向CPU转发更高级的中断请求

中断的基本概念

CPU不可能时刻查询外设的工作状态。。。

一种设备控制方式　　一种外设的数据传输方式

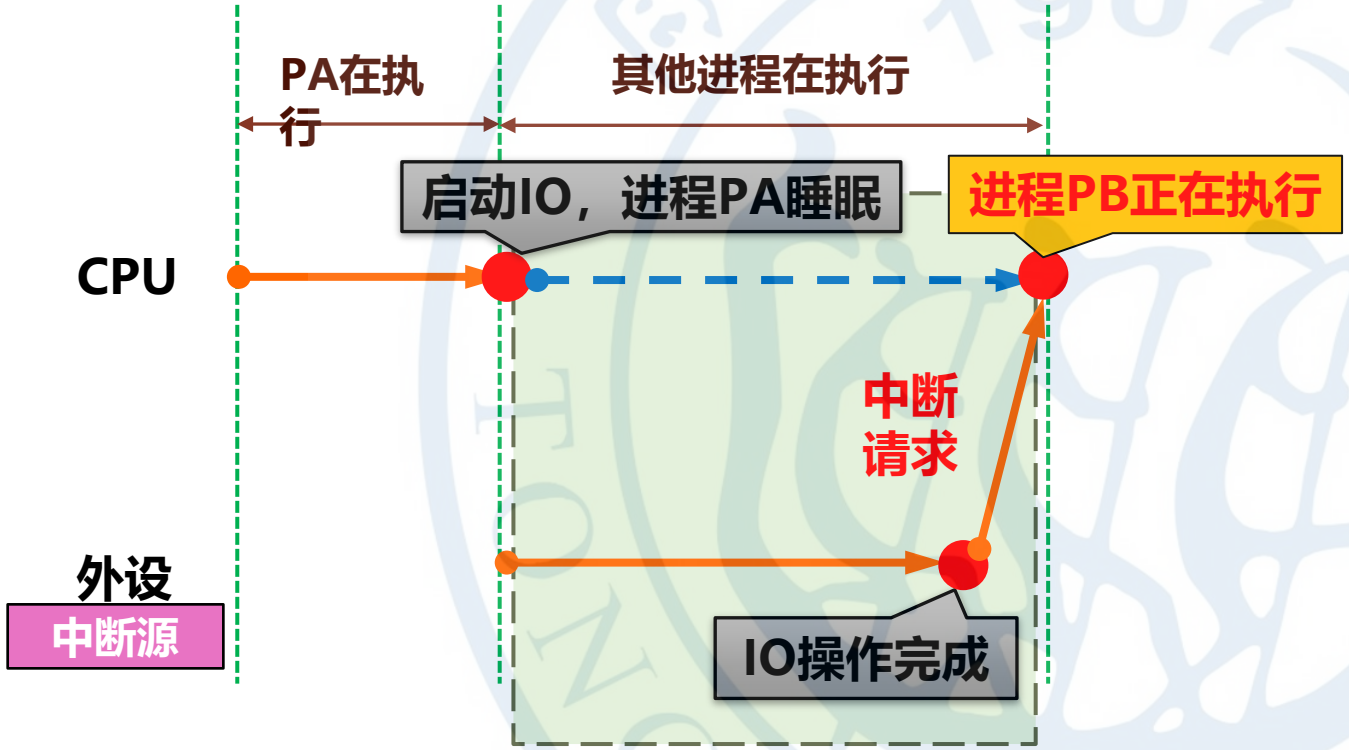


1. 保证了CPU和设备之间的并行操作

中断的基本概念

CPU不可能时刻查询外设的工作状态。。。

一种设备控制方式　　一种外设的数据传输方式



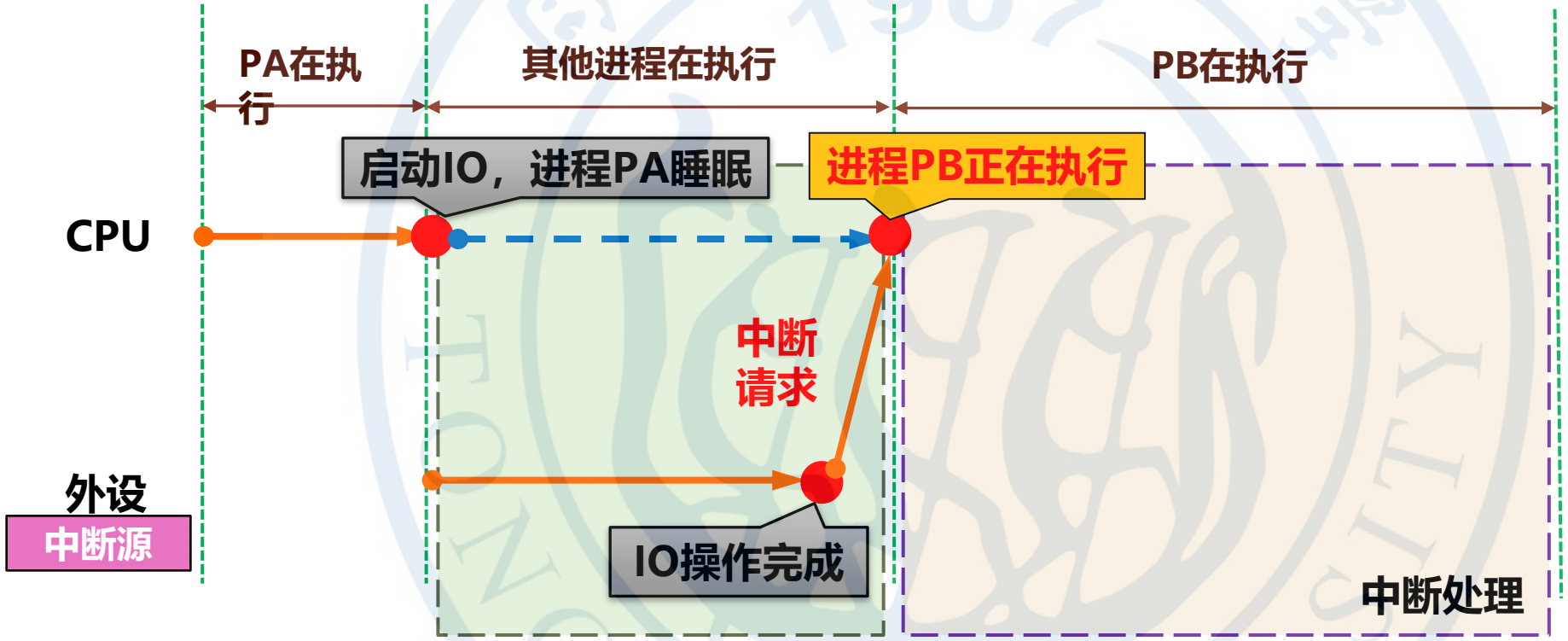
1. 保证了CPU和设备之间的并行操作

Q2>.CPU接收到中断请求后会发生什么?

中断的基本概念

CPU不可能时刻查询外设的工作状态。。。

一种设备控制方式　　一种外设的数据传输方式



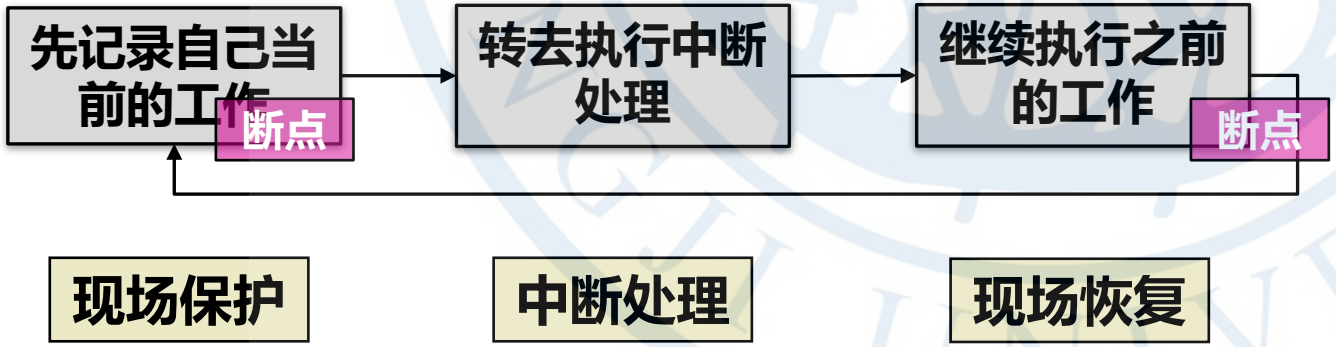
1. 保证了CPU和设备之间的并行操作

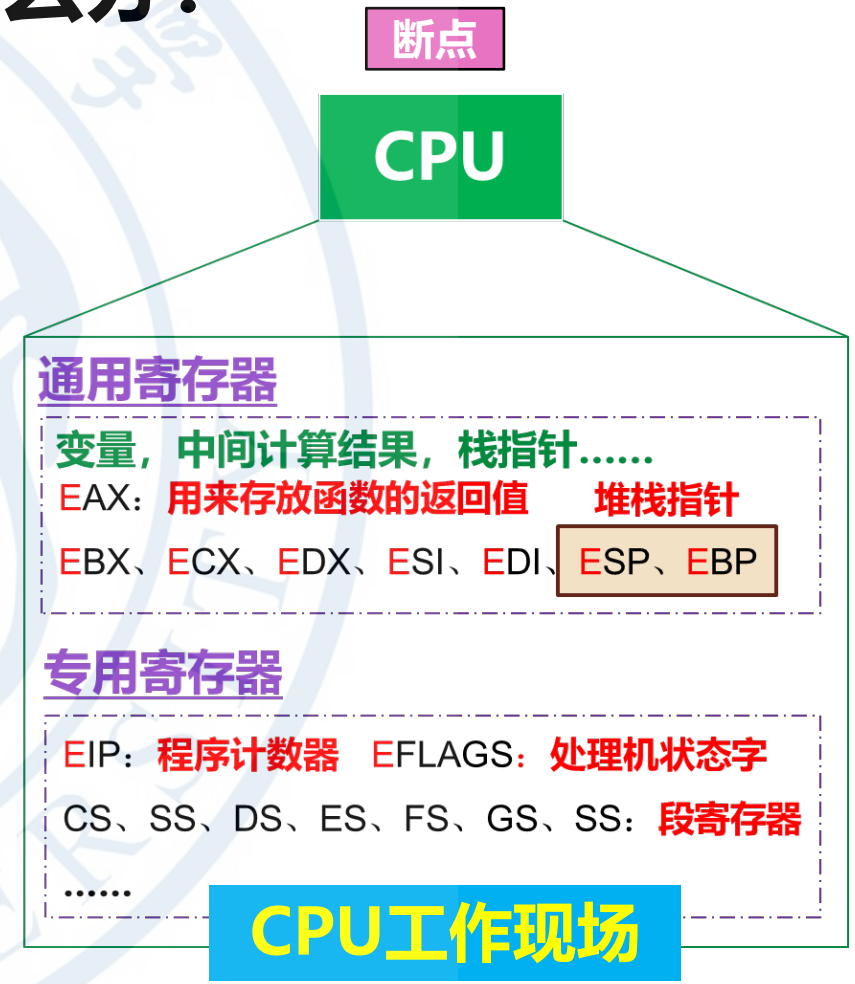
中断的基本概念

设想一下如果你的工作被打断，你怎么办？

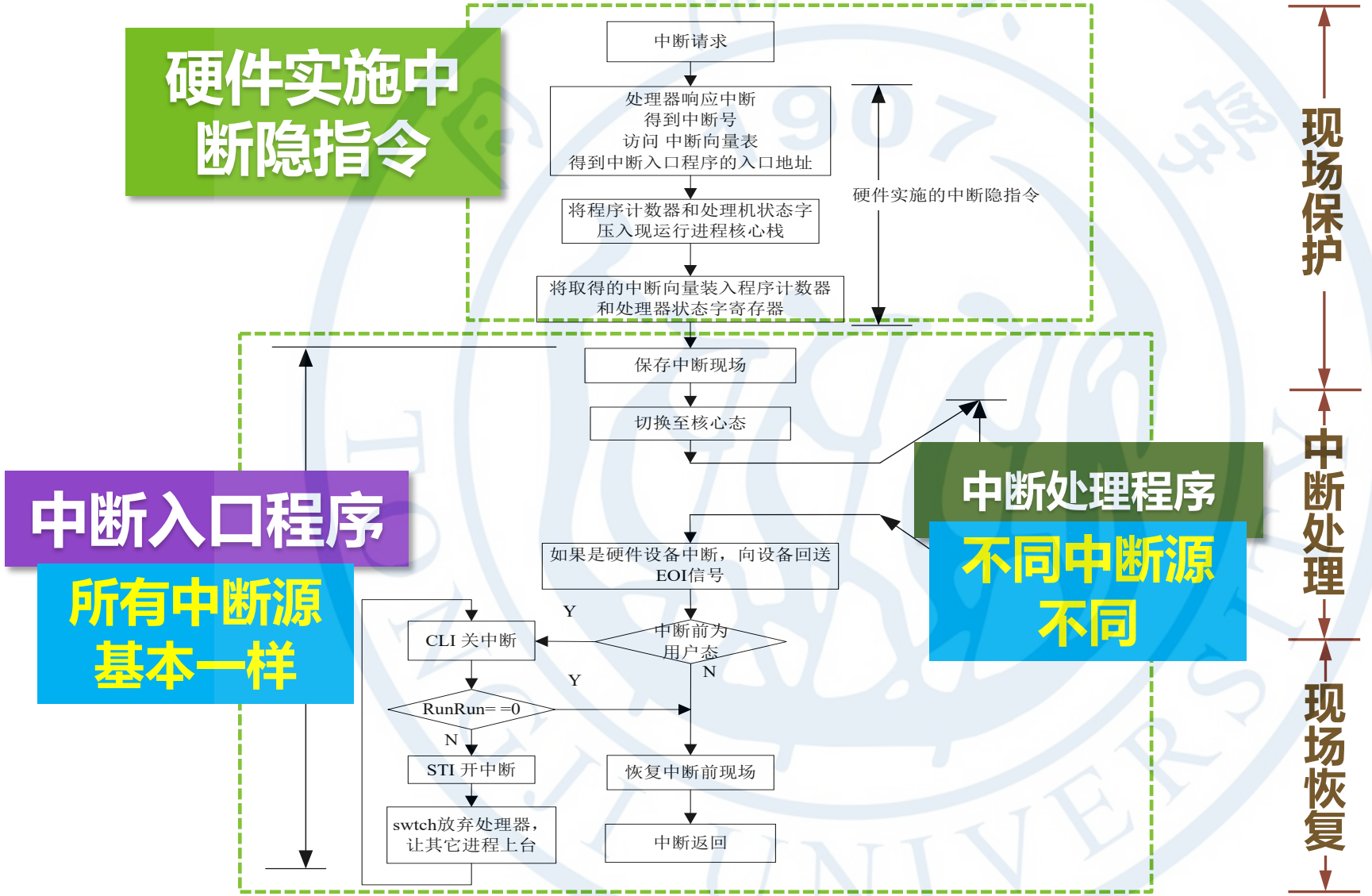
先  再 

进程PB现在该怎么办？





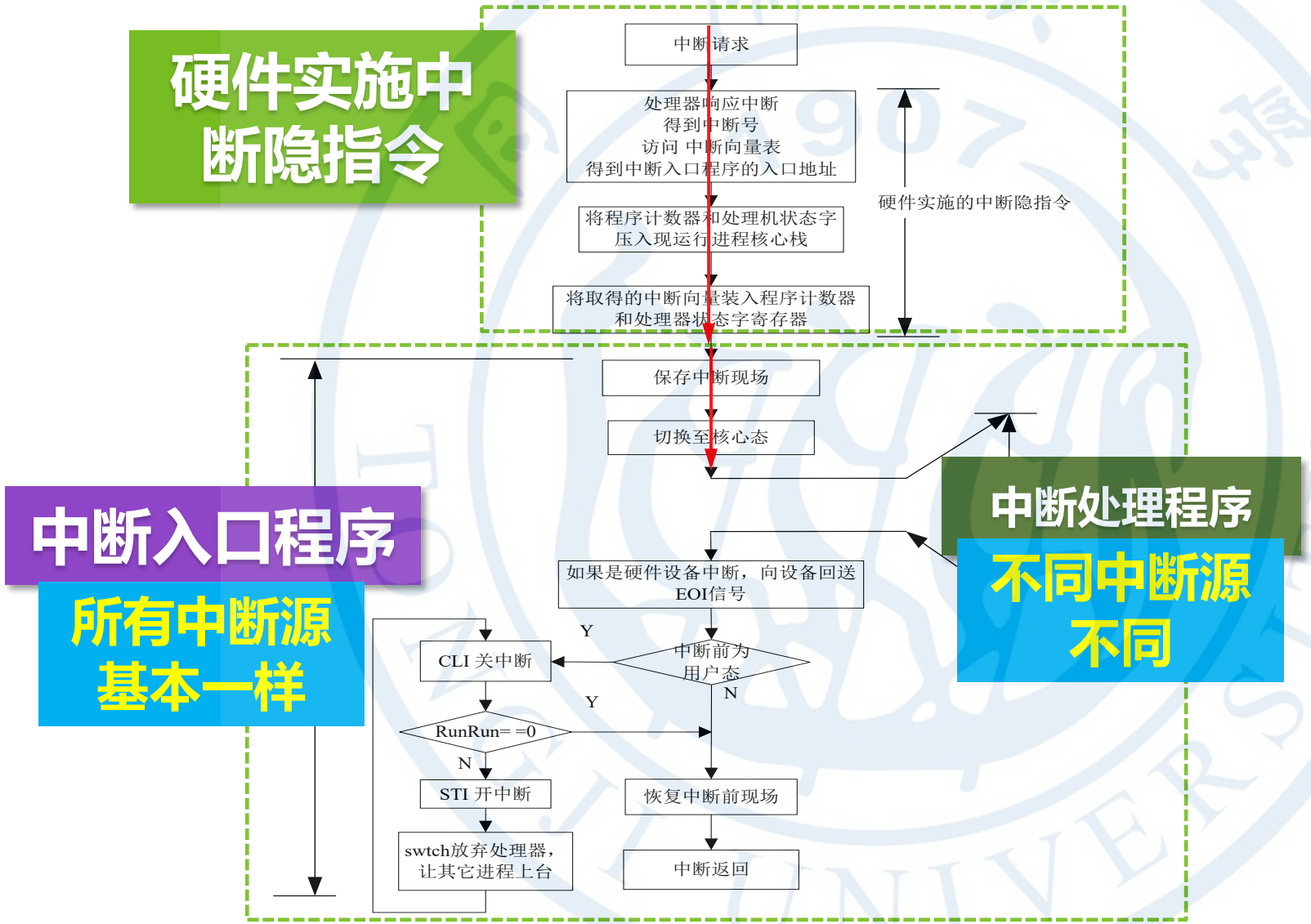
UNIX中断处理流程


硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
中断入口程序
所有中断源基本一样
保存中断现场
切换至核心态
中断处理程序
不同中断源不同
如果是硬件设备中断，向设备回送EOI信号
中断前为用户态
Y
N
CLI 关中断
Y
RunRun= =0
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回
现场保护
中断处理
现场恢复

UNIX中断处理流程


硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
中断入口程序
所有中断源基本一样
保存中断现场
切换至核心态
中断处理程序
不同中断源不同
如果是硬件设备中断，向设备回送EOI信号
中断前为用户态
Y
N
CLI 关中断
Y
RunRun= =0
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回
执行中断隐指令
跳转到中断入口程序
现场保护
中断处理
现场恢复

UNIX中断处理流程

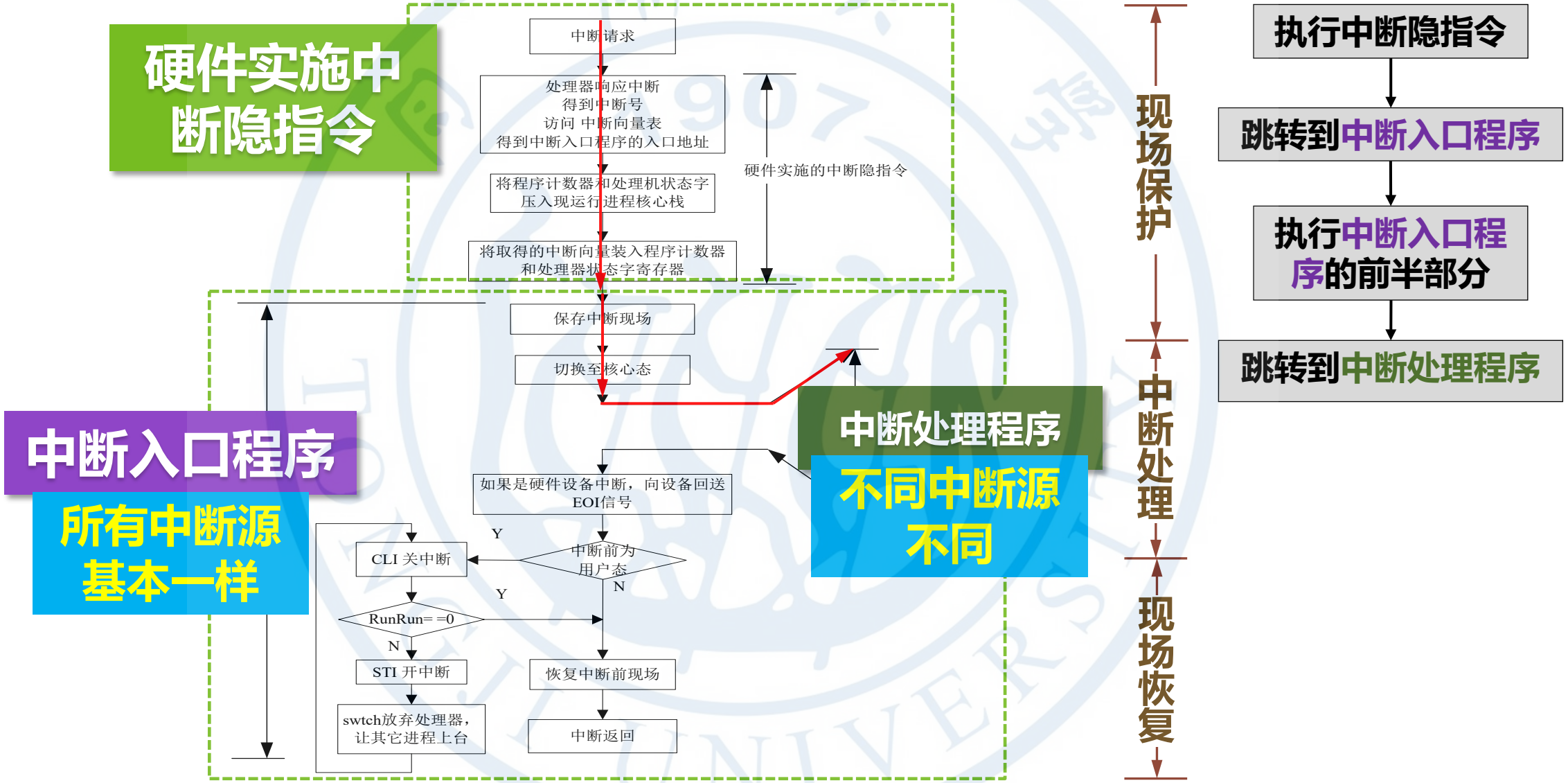


执行中断隐指令

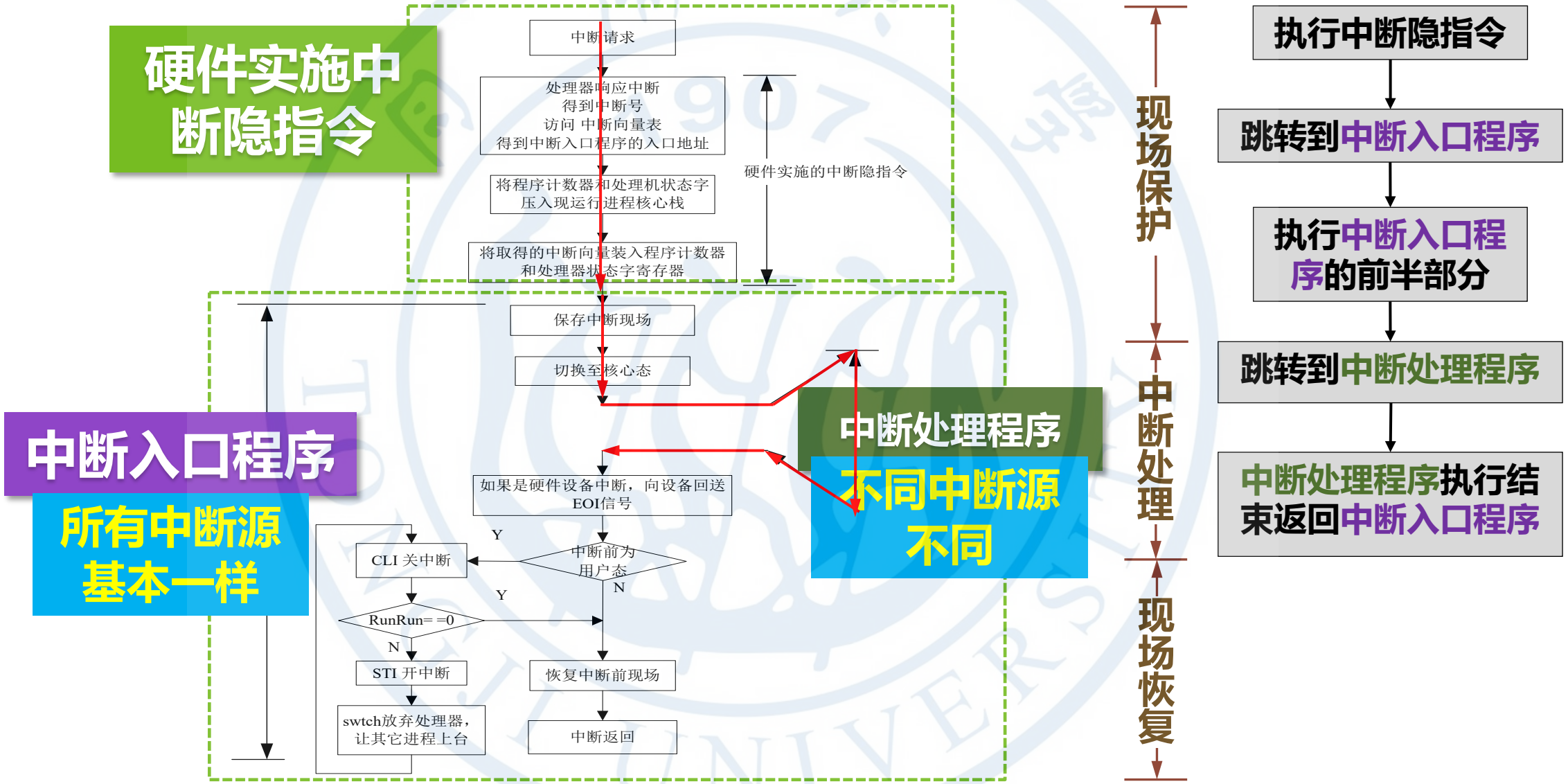
跳转到中断入口程序

执行中断入口程序的前半部分

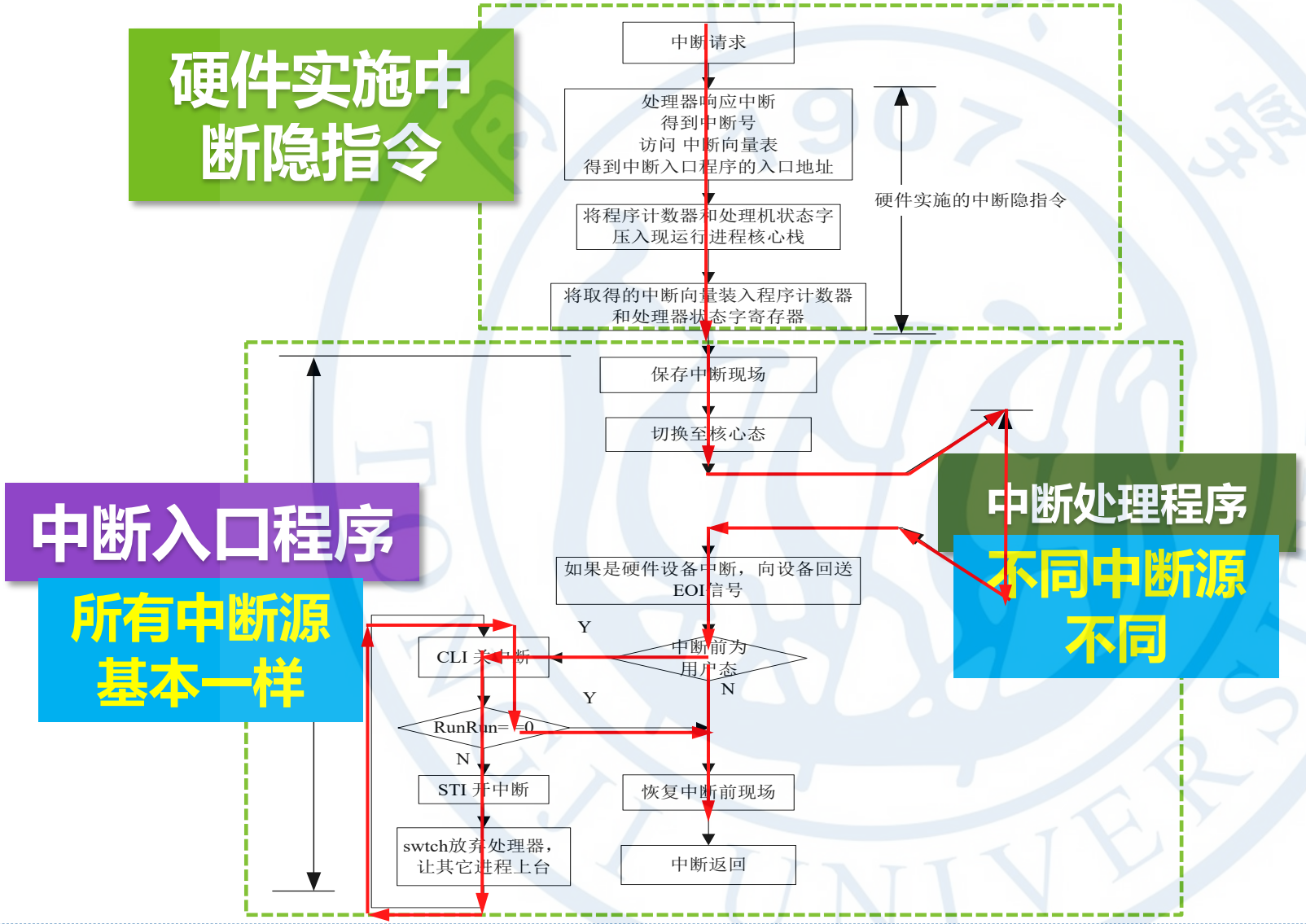
UNIX中断处理流程

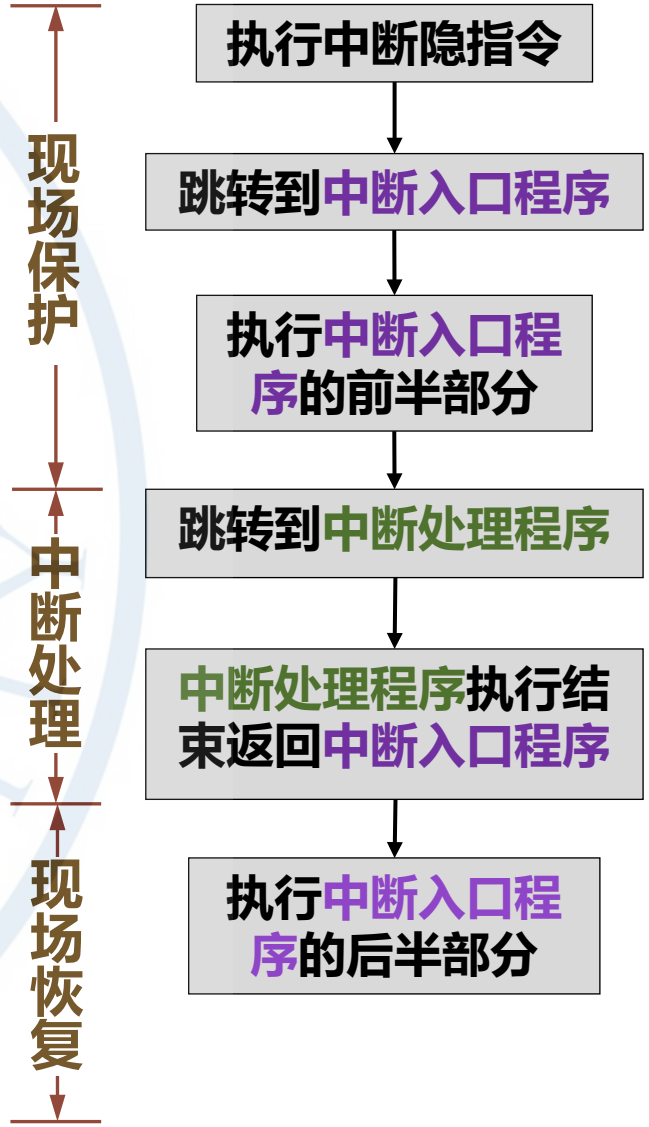

硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
保存中断现场
切换至核心态
中断入口程序
所有中断源基本一样
中断处理程序
不同中断源不同
如果是硬件设备中断，向设备回送EOI信号
中断前为用户态
Y
N
CLI 关中断
Y
RunRun= =0
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回
现场保护
中断处理
现场恢复
执行中断隐指令
跳转到中断入口程序
执行中断入口程序的前半部分
跳转到中断处理程序

UNIX中断处理流程

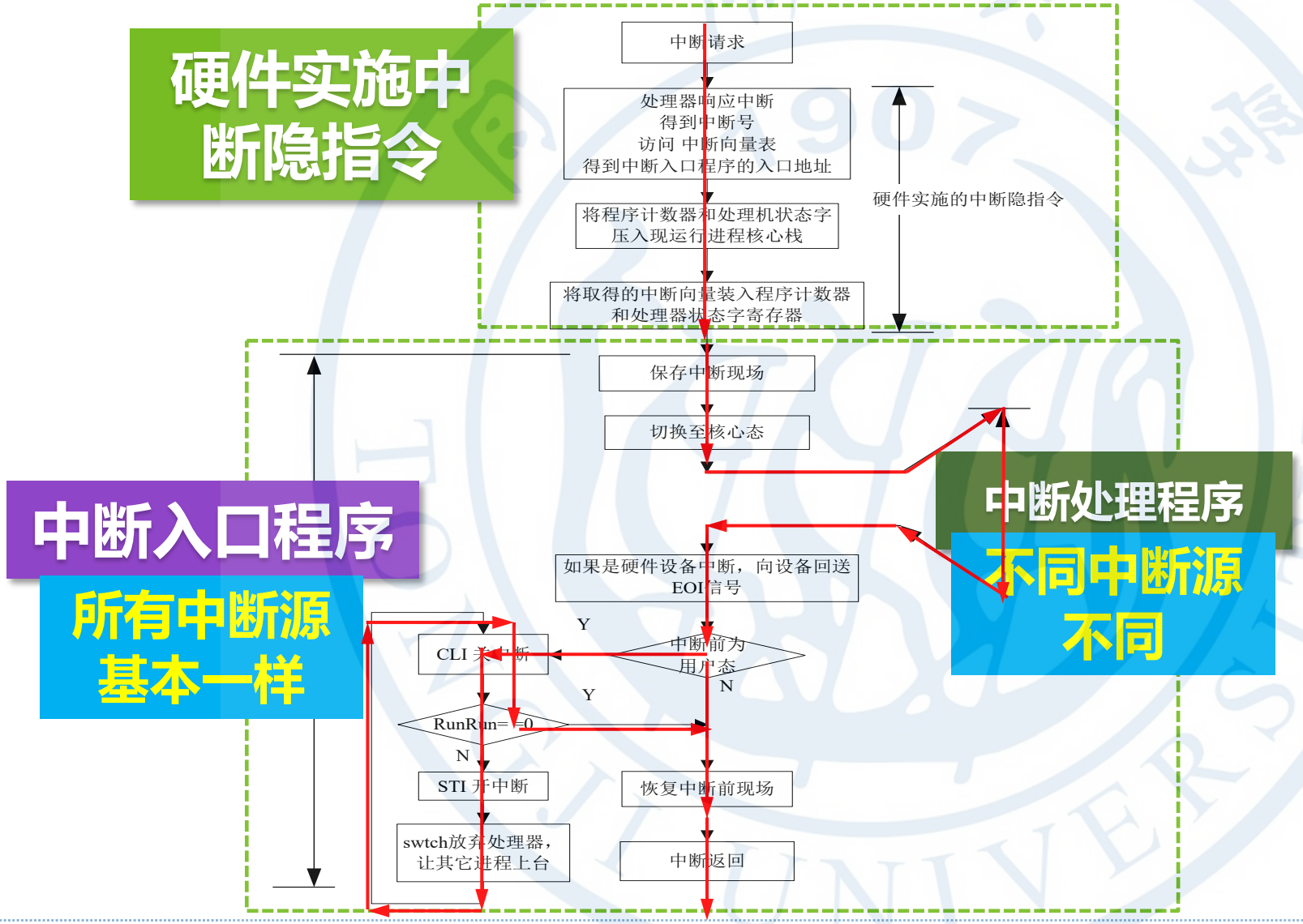

硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
保存中断现场
切换至核心态
中断入口程序
所有中断源基本一样
中断处理程序
不同中断源不同
如果是硬件设备中断，向设备回送EOI信号
中断前为用户态
Y
N
CLI 关中断
RunRun= =0
Y
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回
现场保护
中断处理
现场恢复
执行中断隐指令
跳转到中断入口程序
执行中断入口程序的前半部分
跳转到中断处理程序
中断处理程序执行结束返回中断入口程序

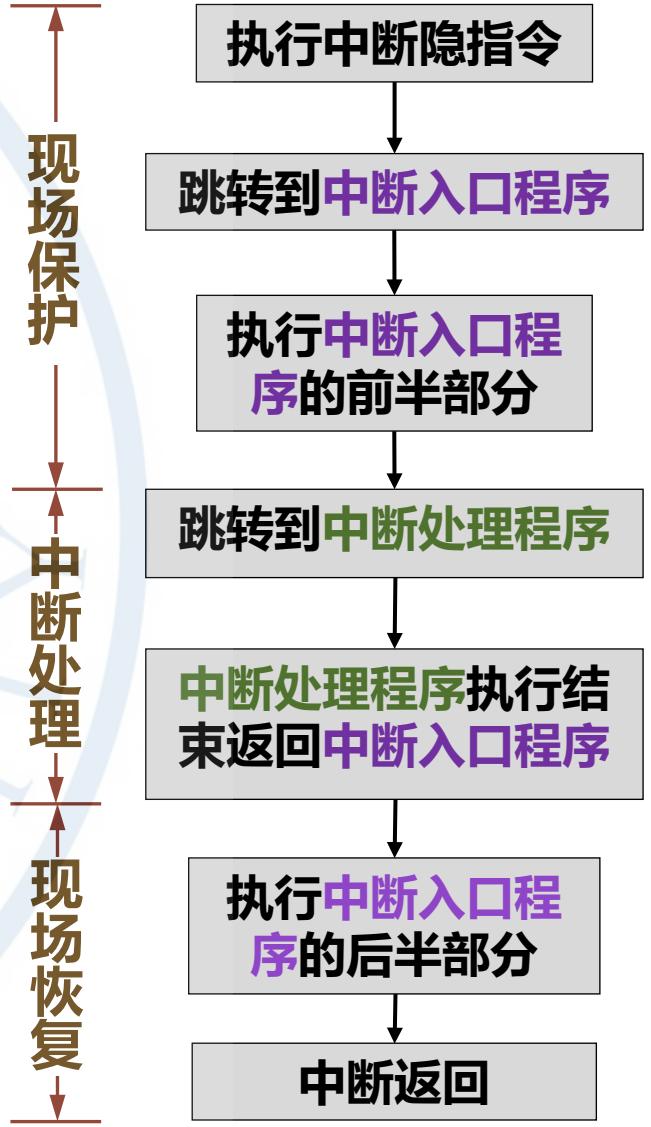
UNIX中断处理流程


硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
中断入口程序
所有中断源基本一样
保存中断现场
切换至核心态
中断处理程序
不同中断源不同
如果是硬件设备中断，向设备回送EOI信号
中断前为用户态
Y
N
CLI 关中断
Y
RunRun!=0
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回

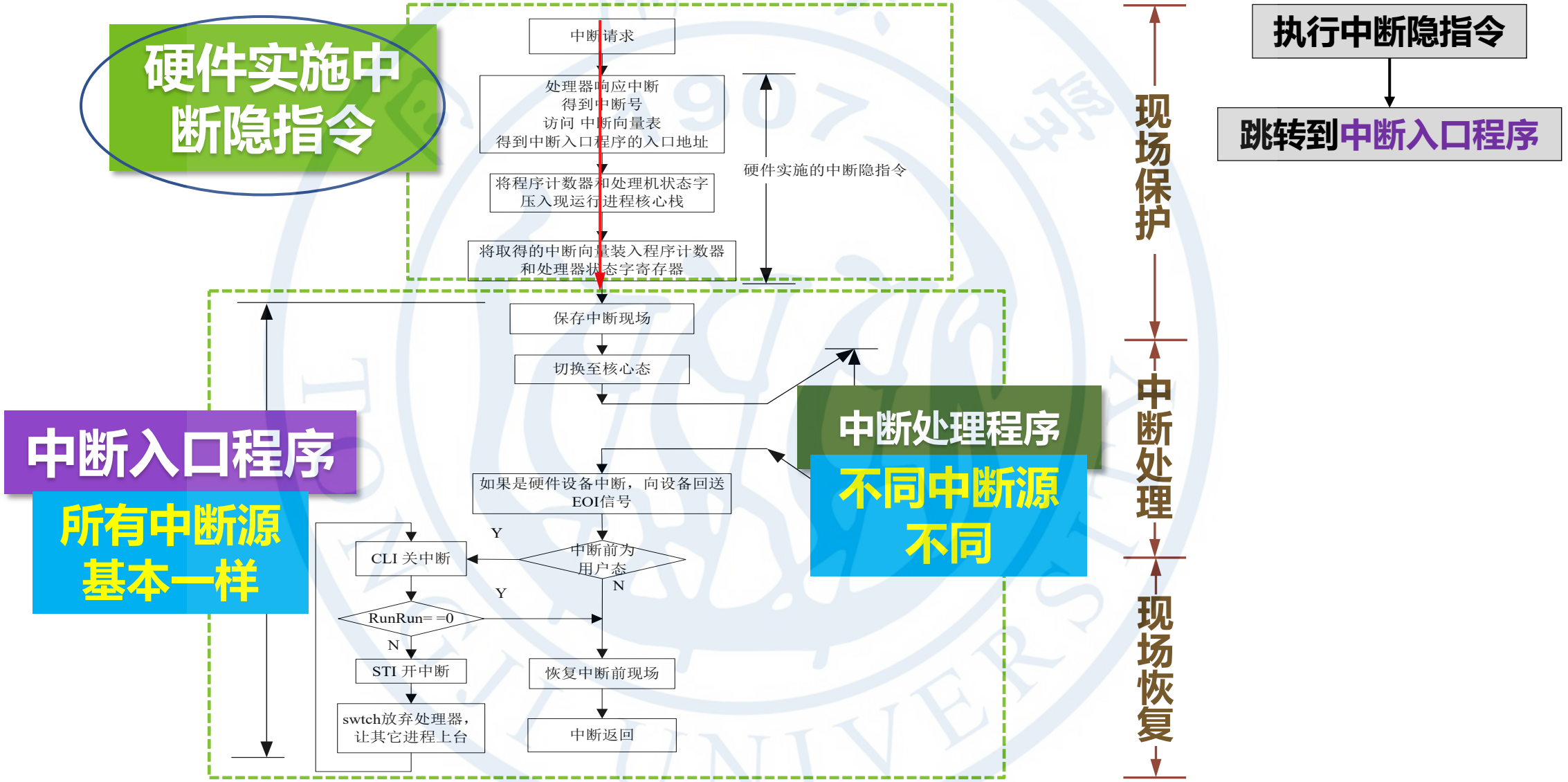

现场保护
中断处理
现场恢复
执行中断隐指令
跳转到中断入口程序
执行中断入口程序的前半部分
跳转到中断处理程序
中断处理程序执行结束返回中断入口程序
执行中断入口程序的后半部分

UNIX中断处理流程


硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
中断入口程序
所有中断源基本一样
保存中断现场
切换至核心态
中断处理程序
不同中断源不同
如果是硬件设备中断，向设备回送EOI信号
中断前为用户态
Y
N
CLI 关中断
Y
RunRun=?=0
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回


现场保护
中断处理
现场恢复
执行中断隐指令
跳转到中断入口程序
执行中断入口程序的前半部分
跳转到中断处理程序
中断处理程序执行结束返回中断入口程序
执行中断入口程序的后半部分
中断返回

UNIX中断处理流程


硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
保存中断现场
切换至核心态
中断入口程序
所有中断源基本一样
中断处理程序
不同中断源不同
如果是硬件设备中断，向设备回送EOI信号
中断前为用户态
Y
N
CLI 关中断
Y
RunRun= =0
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回
执行中断隐指令
跳转到中断入口程序
现场保护
中断处理
现场恢复

UNIX中断处理流程

1. 关中断

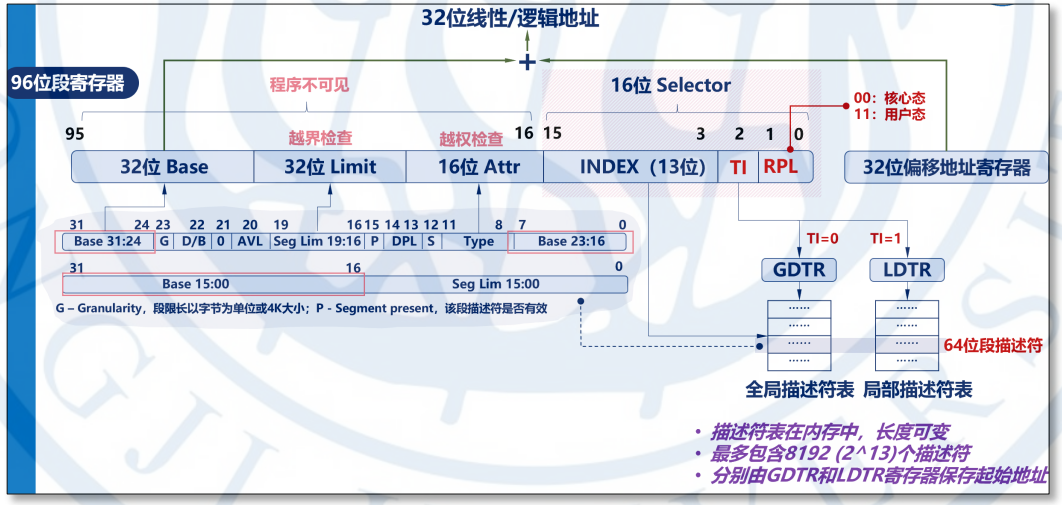
Q3>.这里为什么要关中断?

UNIX中断处理流程

2. 实施硬件现场保护（中断前核心寄存器的值）

ESP	SS	EFLAGS	EIP	CS
栈顶指针	堆栈段寄存器	CPU状态寄存器（含IF）	程序计数器	代码段寄存器

最重要的一组寄存器由硬件保护



UNIX中断处理流程

2. 实施硬件现场保护（中断前核心寄存器的值）

为什么要保存? CS + EIP：断点的位置

中断隐指令中，现场保存结束后，会装入中断入口程序所需的CS和EIP，以实现跳转到中断入口程序

中断向量 不同的中断源有不同的中断向量

UNIX中断处理流程

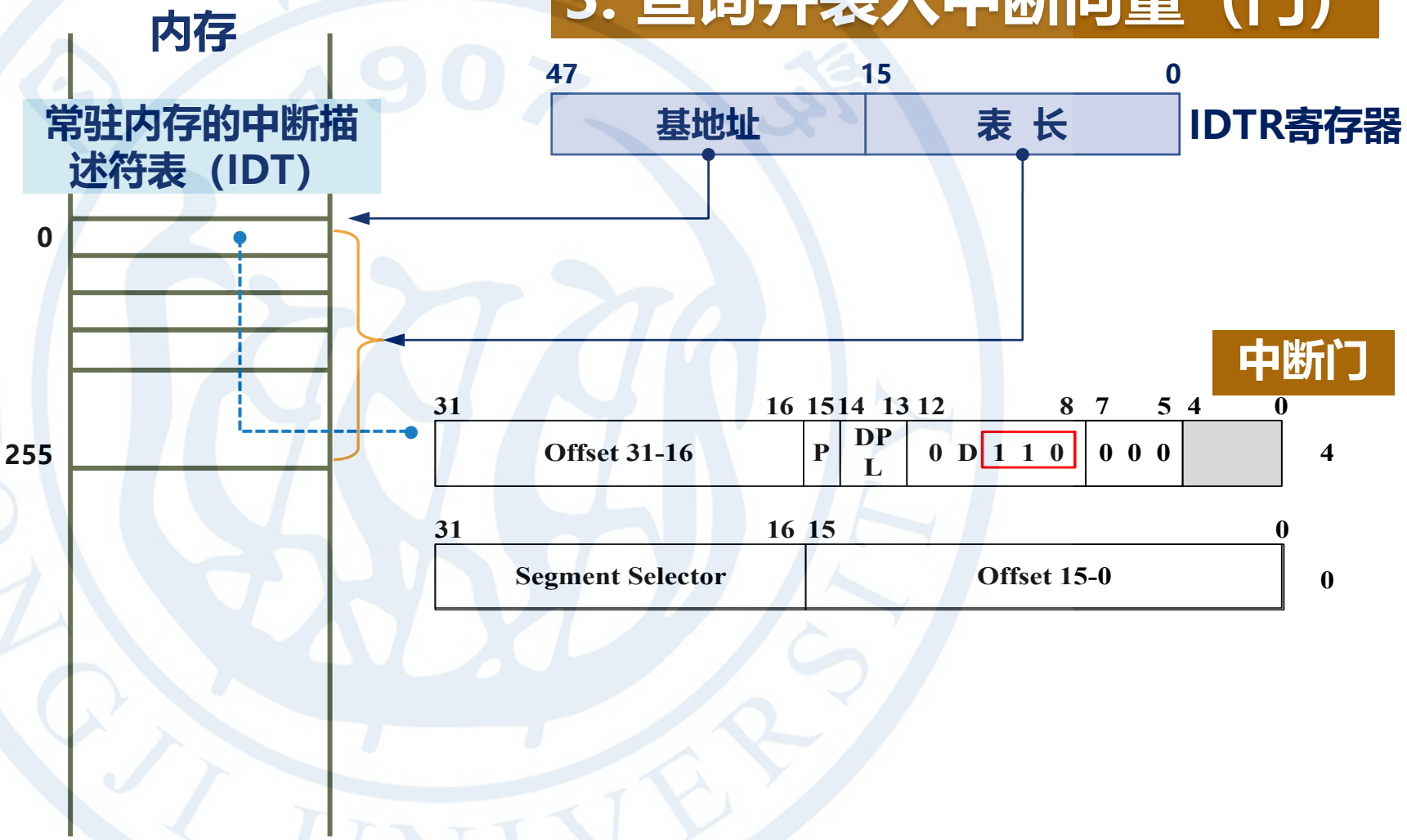
UNIX V6++定义的中断入口程序

中断号/中断源		中断入口程序	中断处理子程序
0x0	除0错	Exception::DivideErrorEntrance()	Exception::DivideError (struct pt_regs* regs, struct pt_context* context);
0x1	调试异常	Exception::DebugEntrance();	Exception::Debug (struct pt_regs* regs, struct pt_context* context);
0x2	NMI非屏蔽中断	Exception::NMIEntrance();	Exception:: NMI (struct pt_regs* regs, struct pt_context* context);
0x3	调试断点	Exception::BreakpointEntrance();	Exception:: Breakpoint (struct pt_regs* regs, struct pt_context* context);
……	……	……	……
0x1F	保留异常		
0x20	时钟中断	Time::TimeInterruptEntrance()	void Ti…t* context)
0x21	键盘中断	KeyboardInterrupt::KeyboardInterruptEntrance()	void K… struct pt_conte…
…	…		
0x2E	硬盘中断	void DiskInterrupt::DiskInterruptEntrance()	void ATADriver::ATAHandler(struct pt_regs *reg, struct pt_context *context)
….	…	…	…
0x80	系统调用	void SystemCall::SystemCallEntrance()	void SystemCall::Trap(struct pt_regs* regs, struct pt_context* context)
…	…	…	…

每一个中断入口程序有自己的中断向量，怎么装入CS和EIP呢?

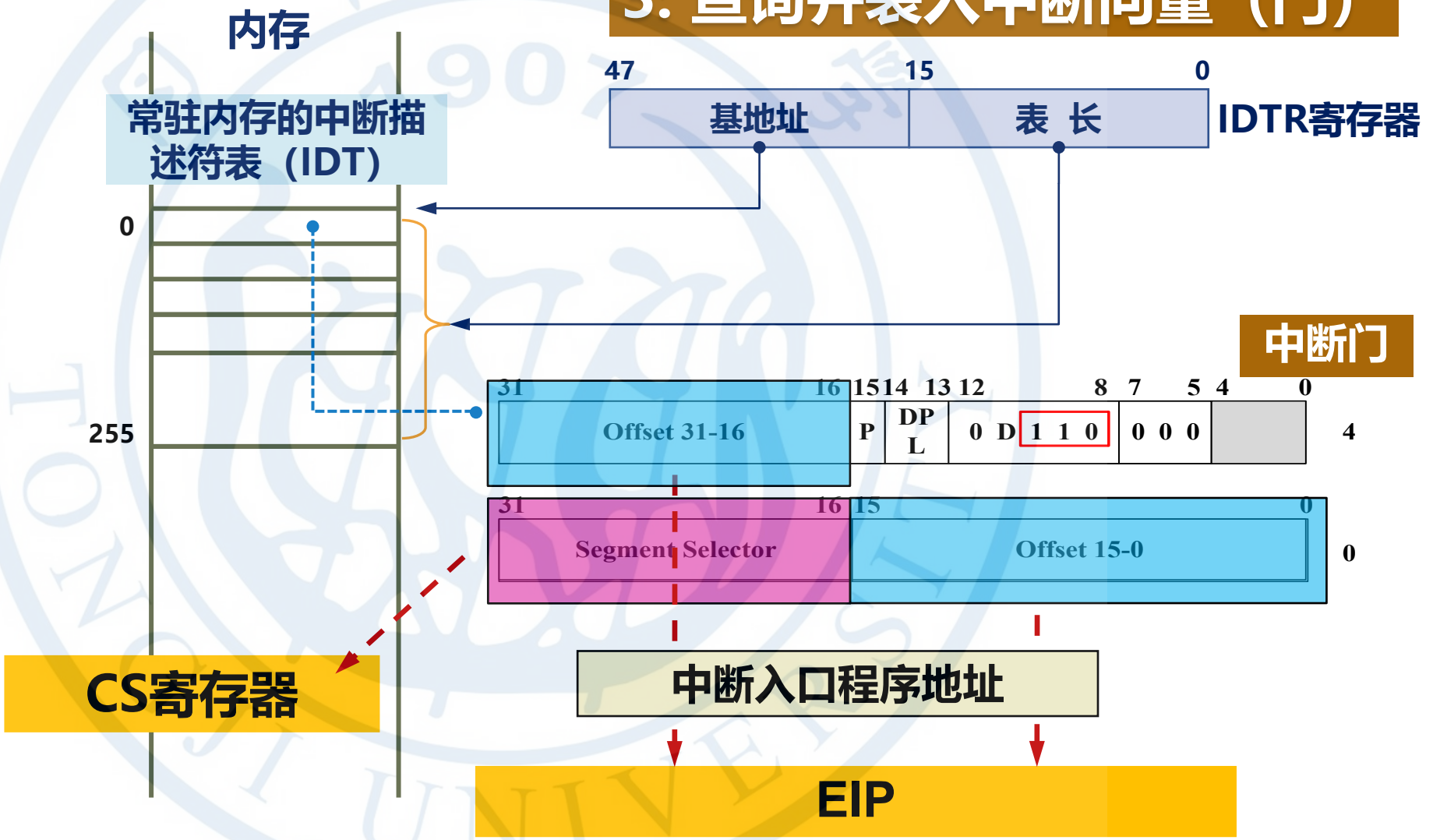
UNIX中断处理流程

3. 查询并装入中断向量（门）



UNIX中断处理流程

3. 查询并装入中断向量（门）



UNIX中断处理流程

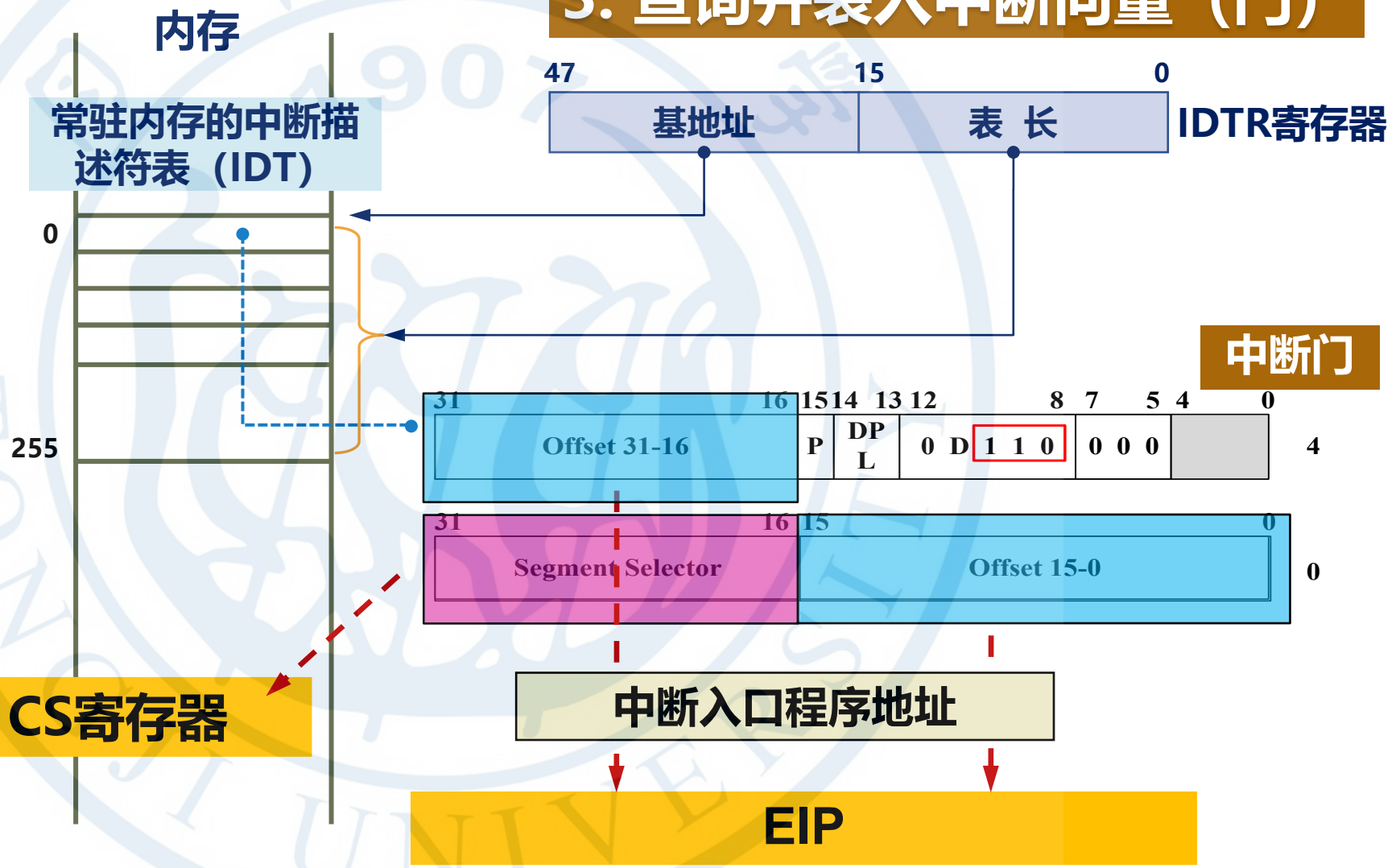
4、0x2E → CPU获取到中断源

如何由中断号找到对应的中断入口程序？？？

IDT基地址
+
中断号×(64/8)

获取对应中断源的中断向量

3. 查询并装入中断向量（门）



UNIX中断处理流程

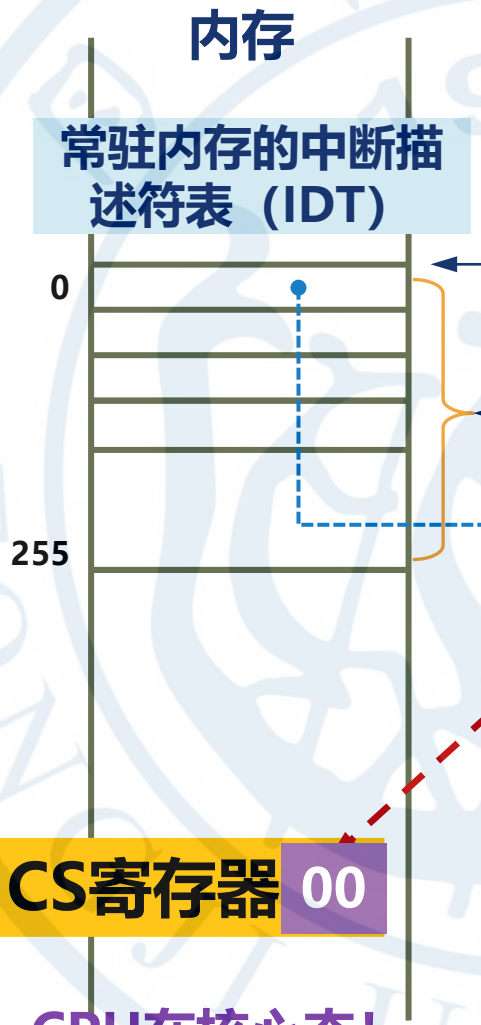
3. 查询并装入中断向量（门）

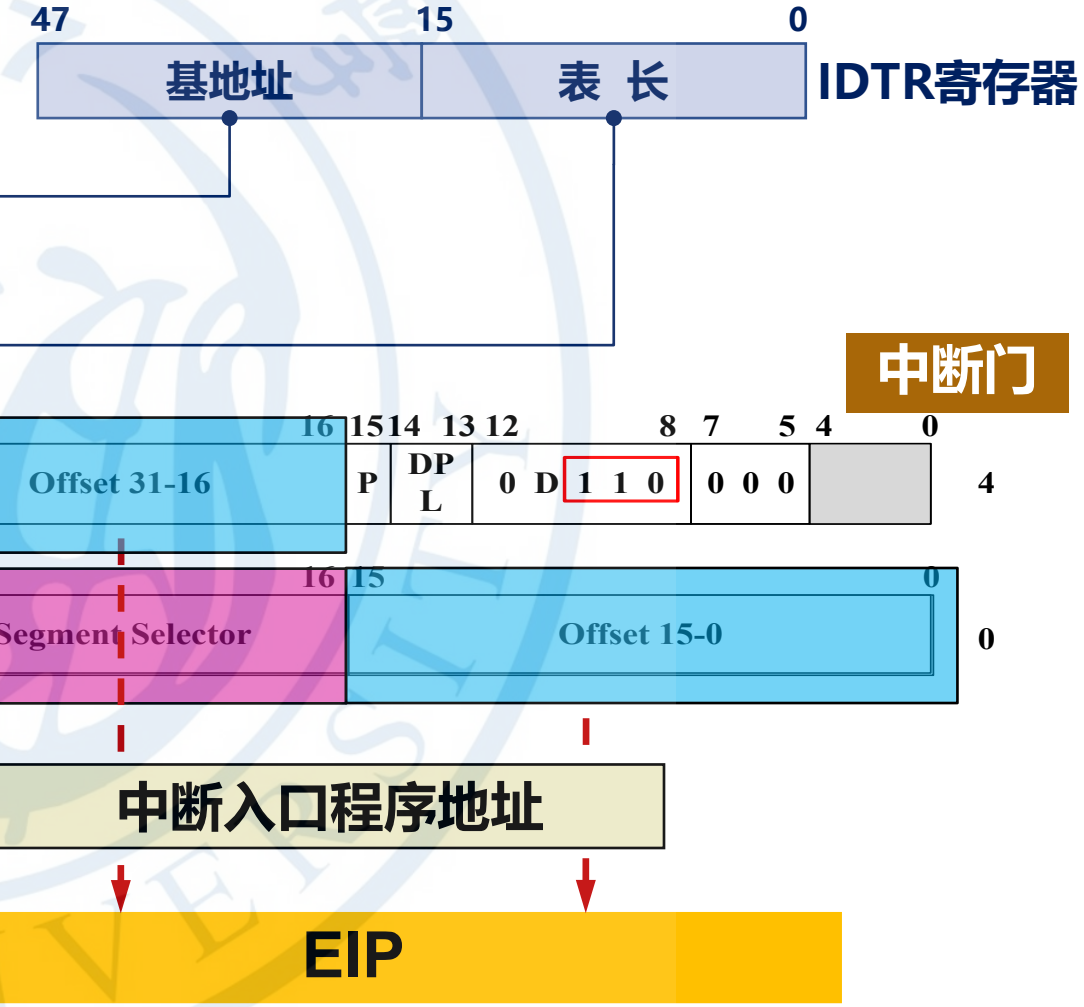
4、0x2E → CPU获取到中断源

如何由中断号找到对应的中断入口程序？？？

IDT基地址
+
中断号×(64/8)

获取对应中断源的中断向量





CS寄存器 00

执行中断处理程序时，CPU在核心态！

UNIX中断处理流程

用户态
User Mode

执行用户程序，
提供用户功能

核心态
Kernel Mode

执行内核程序，
提供系统功能

可以在一定时机相互转换

内核不是与用户进程平行运行的孤立的进程集合。

在核心态下执行内核代码的进程完成了内核功能！

UNIX中断处理流程

2. 实施硬件现场保护（中断前核心寄存器的值）

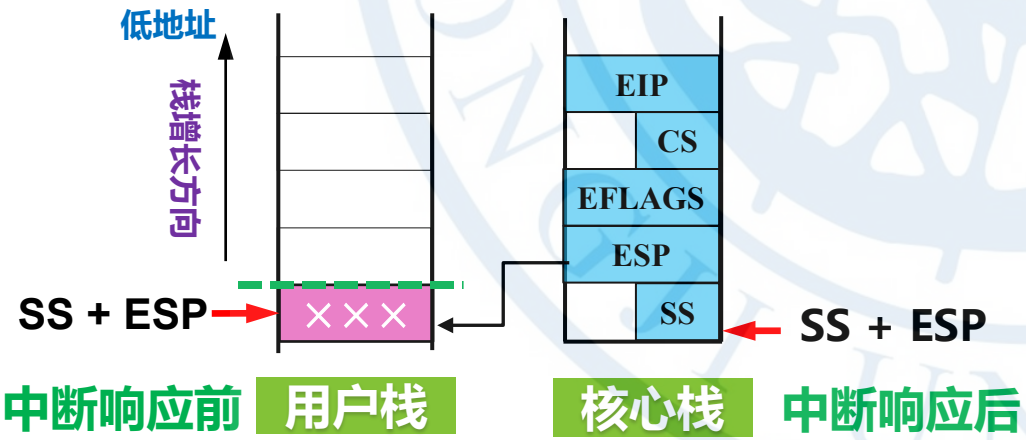
为什么要保存?

中断隐指令中，现场保存结束后，会将SS设置为核心栈段选择子。

UNIX中断处理流程

2. 实施硬件现场保护（中断前核心寄存器的值）

中断发生时现运行进程在用户态运行



UNIX中断处理流程

2. 实施硬件现场保护（中断前核心寄存器的值）

堆栈段寻址

代码段寻址

ESP	SS	- -	EFLAGS	EIP	CS	- -
栈顶指针	堆栈段寄存器		CPU状态寄存器（含IF）	程序计数器	代码段寄存器	

00：核心态
11：用户态

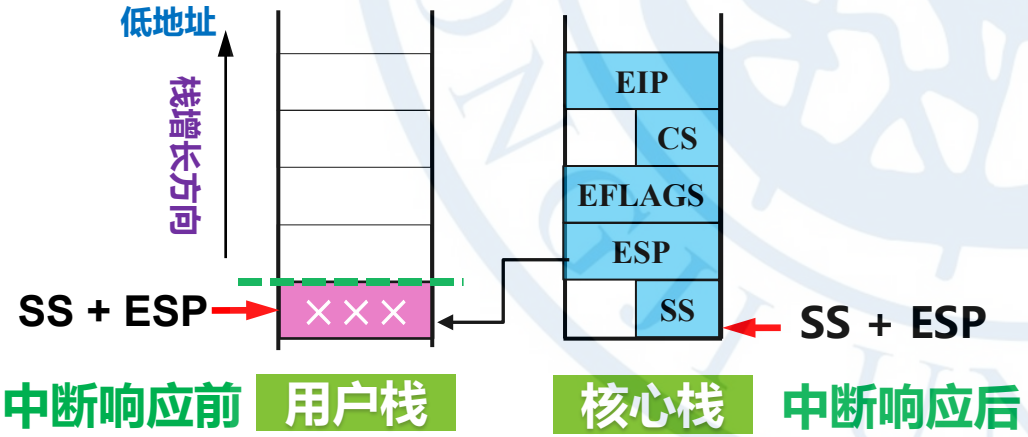
最重要的一组寄存器由硬件保护

栈顶指针

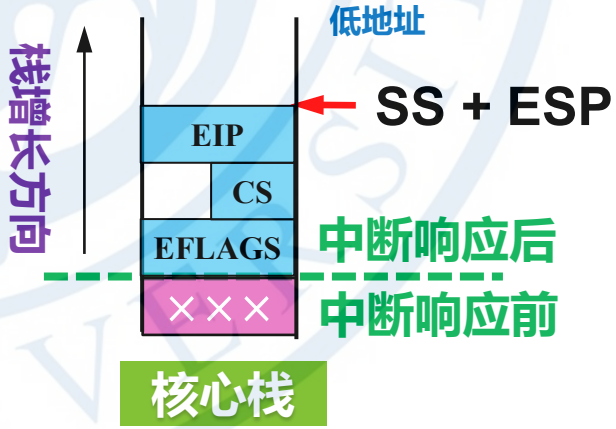
CPU的重要标志位

中断发生时，下一条指令位置和CPU等级

中断发生时现运行进程在用户态运行



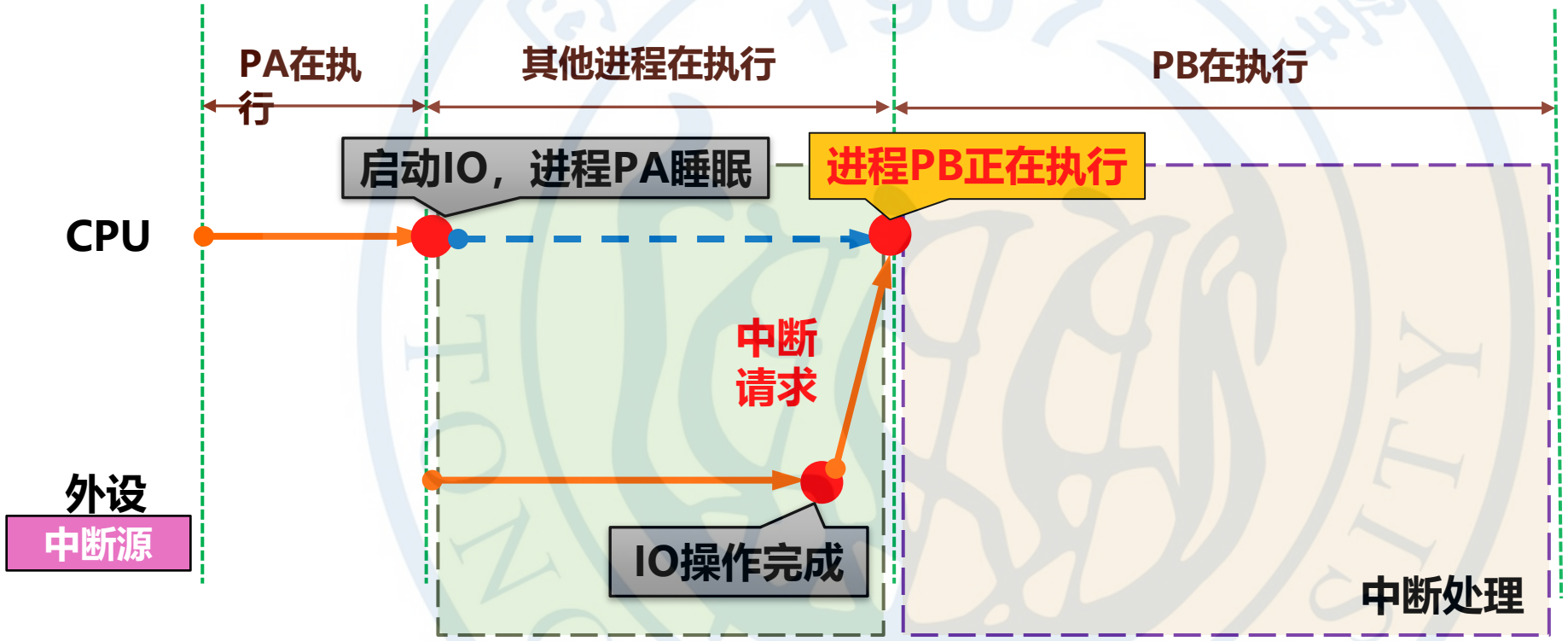
中断发生时现运行进程在核心态运行 ？ 中断嵌套



中断的基本概念

CPU不可能时刻查询外设的工作状态。。。

一种设备控制方式　　一种外设的数据传输方式



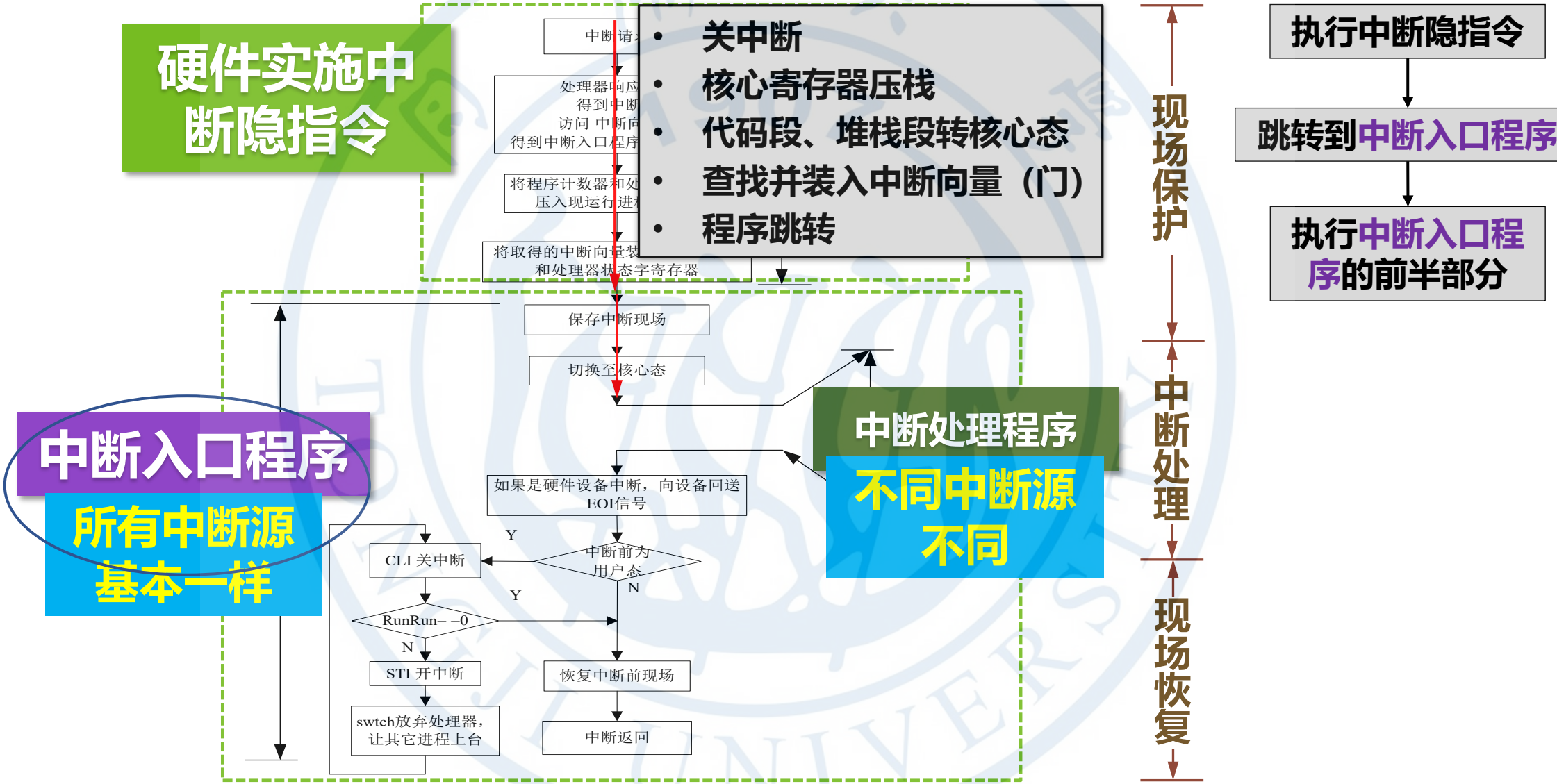
1. 保证了CPU和设备之间的并行操作

2. 提供了进程执行内核代码的机会

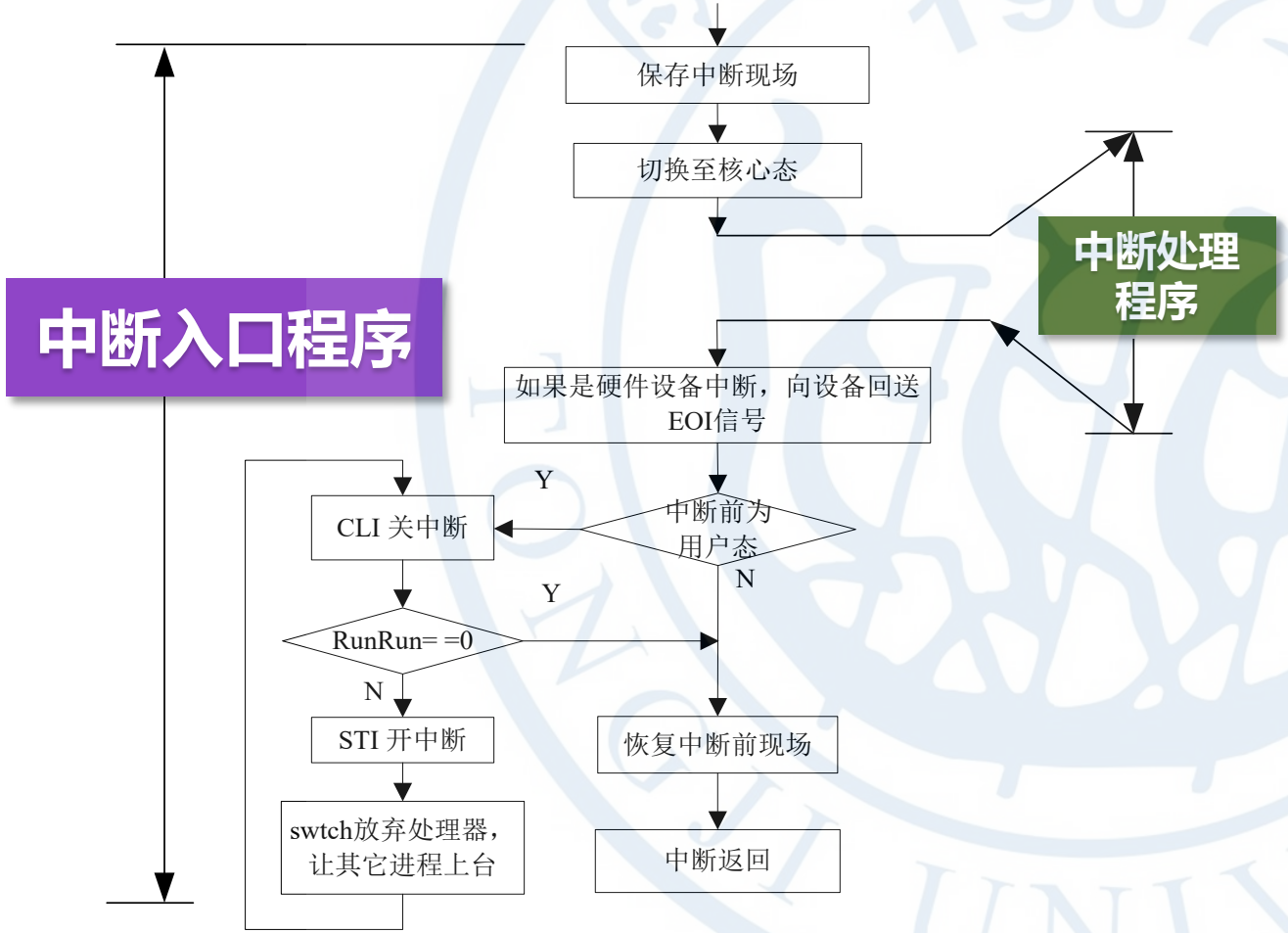
UNIX中断处理流程

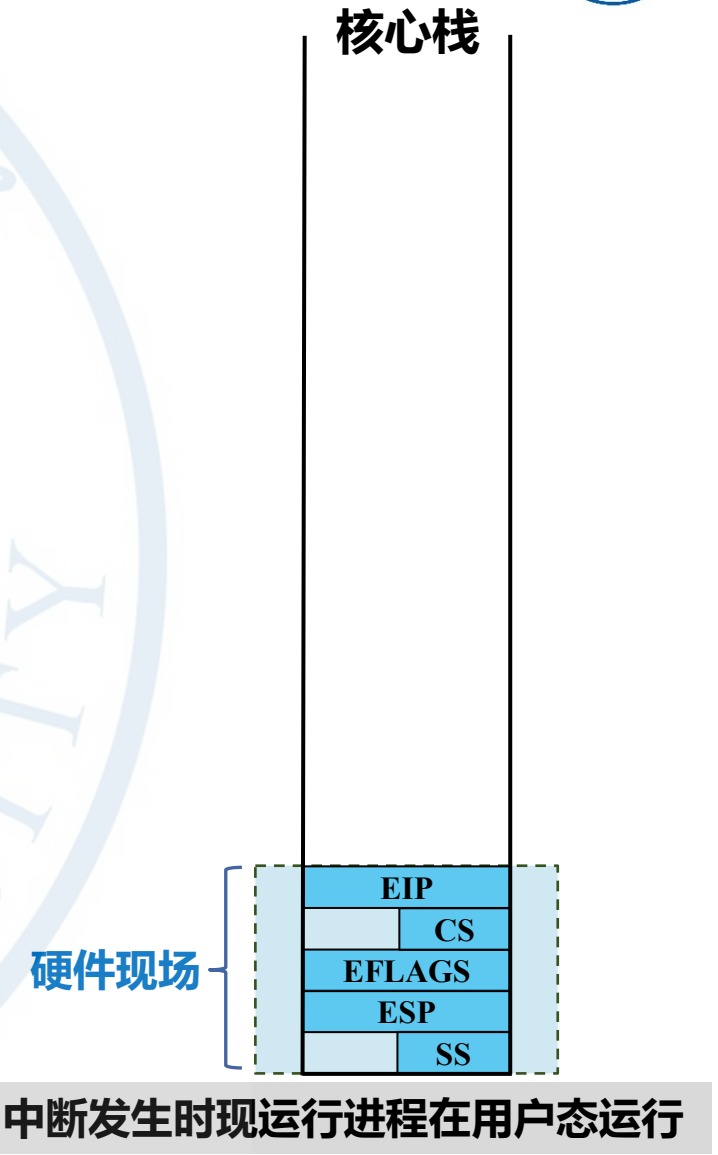

硬件实施中断隐指令
关中断
核心寄存器压栈
代码段、堆栈段转核心态
查找并装入中断向量（门）
程序跳转
执行中断隐指令
跳转到中断入口程序
现场保护
中断处理
现场恢复
中断入口程序
所有中断源基本一样
中断处理程序
不同中断源不同
保存中断现场
切换至核心态
如果是硬件设备中断，向设备回送EOI信号
中断前为用户态
Y
N
CLI 关中断
RunRun= =0
Y
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回

UNIX中断处理流程


硬件实施中断隐指令
关中断
核心寄存器压栈
代码段、堆栈段转核心态
查找并装入中断向量（门）
程序跳转
中断请求
处理器响应
得到中断
访问 中断向
得到中断入口程序
将程序计数器和处
压入现运行进程
将取得的中断向量装
和处理器状态字寄存器
保存中断现场
切换至核心态
中断入口程序
所有中断源
基本一样
中断处理程序
不同中断源
不同
如果是硬件设备中断，向设备回送
EOI信号
中断前为
用户态
Y
N
CLI 关中断
RunRun= =0
Y
N
STI 开中断
swtch放弃处理器，
让其它进程上台
恢复中断前现场
中断返回
现场保护
中断处理
现场恢复
执行中断隐指令
跳转到中断入口程序
执行中断入口程序的前半部分

UNIX中断处理流程

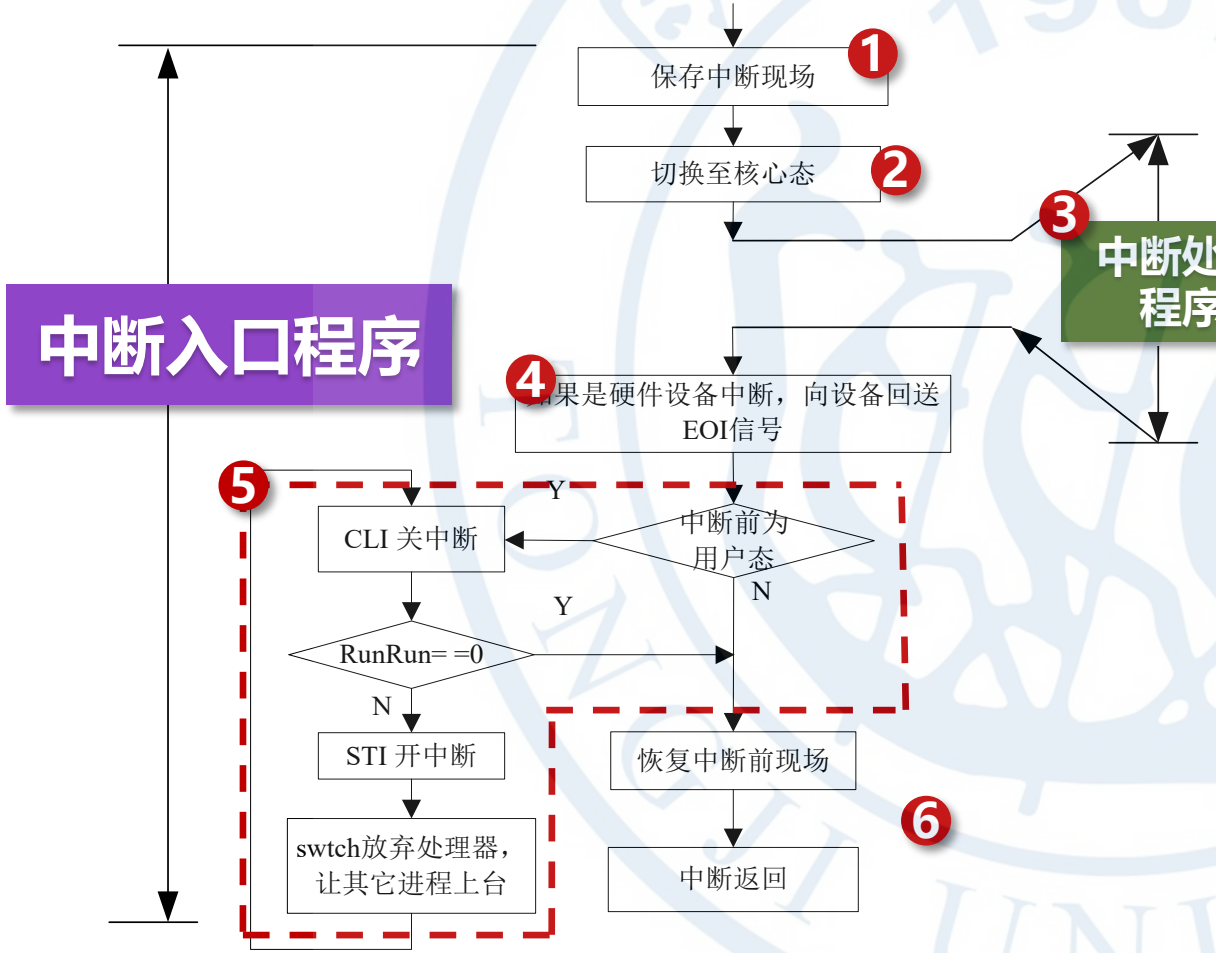




中断发生时现运行进程在用户态运行

UNIX中断处理流程

UNIX V6++中，所有的中断处理程序有相同的结构。以时钟中断为例：



内联的汇编指令

C语言

内联的汇编指令

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext(); 1

    /* 完全进入核心态 */
    SwitchToKernel(); 2

    /* 调用时钟中断处理程序 */
    CallHandler(Time, Clock); 3

    /* 例行调度 */ 5
    ......

    /* 恢复现场 */
    RestoreContext();
    /* 手工销毁栈帧 */ 6
    Leave();
    /* 退出中断 */
    InterruptReturn();
}
```

UNIX中断处理流程

中断入口程序

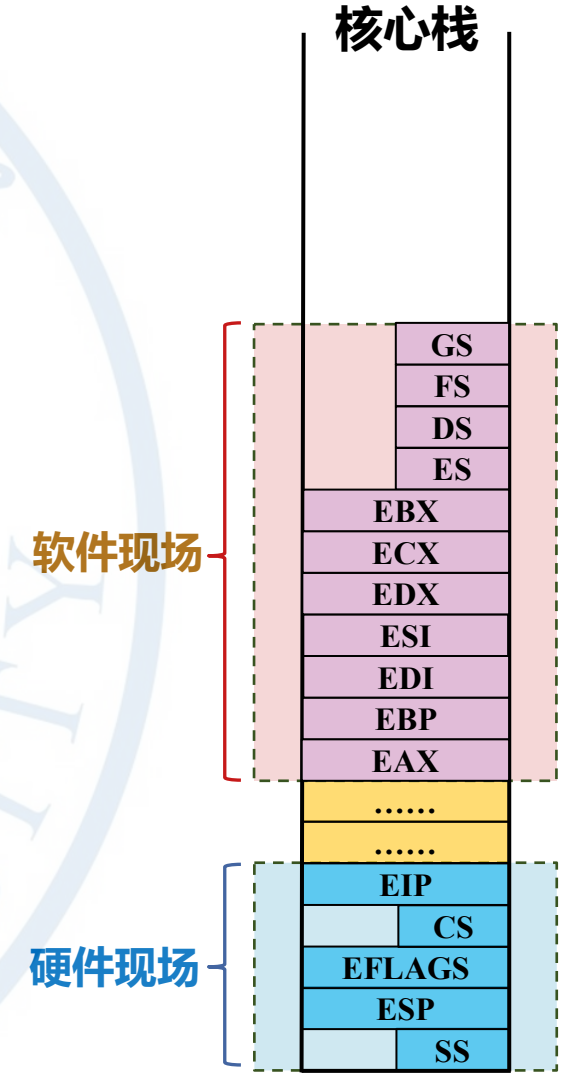
（1）：继续现场保护的工作

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();   ①

    /* 完全进入核心态 */
    SwitchToKernel();

    ……

    /* 恢复现场 */
    RestoreContext();
    /* 手工销毁栈帧 */
    Leave();
    /* 退出中断 */
    InterruptReturn();
}
```

通过一组汇编指令，将CPU中其他的寄存器值压栈，并压入两个分别指向软件现场和硬件现场的指针

SaveContext（内联汇编）

```
#define SaveContext() \
__asm__ __volatile__("cld; \
pushl %%eax; \
pushl %%ebp; \
pushl %%edi; \
pushl %%esi; \
pushl %%edx; \
pushl %%ecx; \
pushl %%ebx; \
pushl %%es;  \
pushl %%ds;  \
pushl %%fs;  \
pushl %%gs;  \
lea 0x4(%%ebp), %%edx; \
pushl %%edx; \
lea 0x4(%%esp), %%edx; \
pushl %%edx"::);
```



中断发生时现运行进程在用户态运行

UNIX中断处理流程

中断入口程序

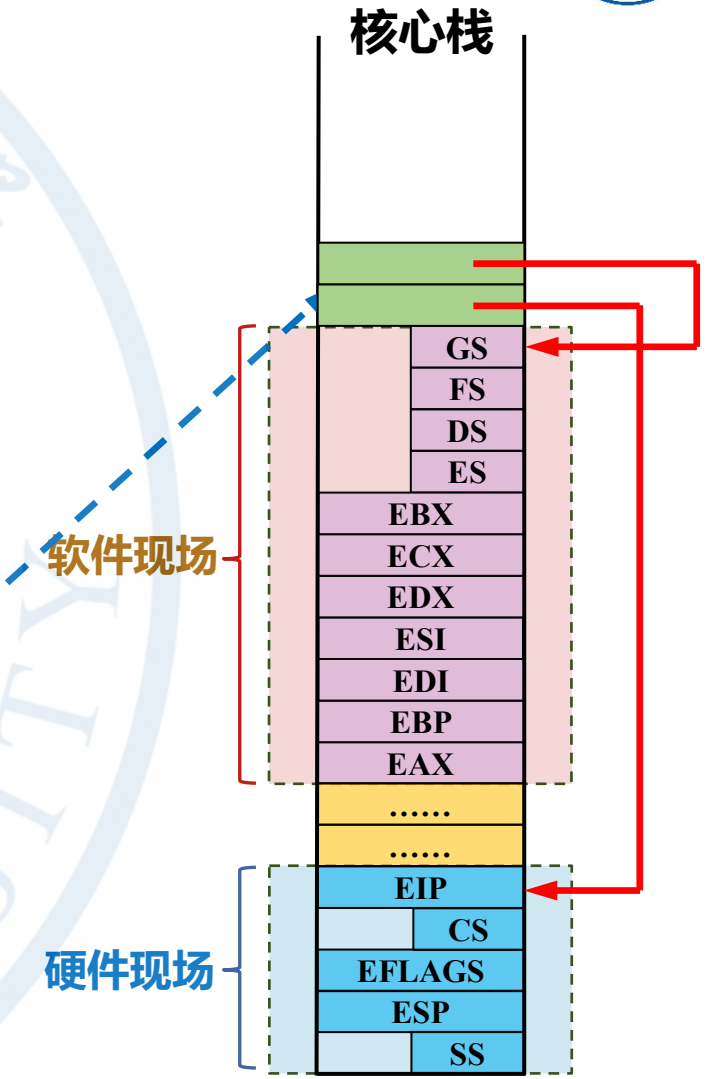
（1）：继续现场保护的工作

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();
    /* 完全进入核心态 */
    SwitchToKernel();
    ……
    /* 恢复现场 */
    RestoreContext();
    /* 手工销毁栈帧 */
    Leave();
    /* 退出中断 */
    InterruptReturn();
}
```

通过一组汇编指令，将CPU中其他的寄存器值压栈，并压入两个分别指向软件现场和硬件现场的指针

SaveContext（内联汇编）

```
#define SaveContext() \
__asm__ __volatile__("cld; \
pushl %%eax; \
pushl %%ebp; \
pushl %%edi; \
pushl %%esi; \
pushl %%edx; \
pushl %%ecx; \
pushl %%ebx; \
pushl %%es;  \
pushl %%ds;  \
pushl %%fs;  \
pushl %%gs;  \
lea 0x4(%%ebp), %%edx; \
pushl %%edx; \
lea 0x4(%%esp), %%edx; \
pushl %%edx"::);
```



中断发生时现运行进程在用户态运行

UNIX中断处理流程

（2）：内存管理转入核心态

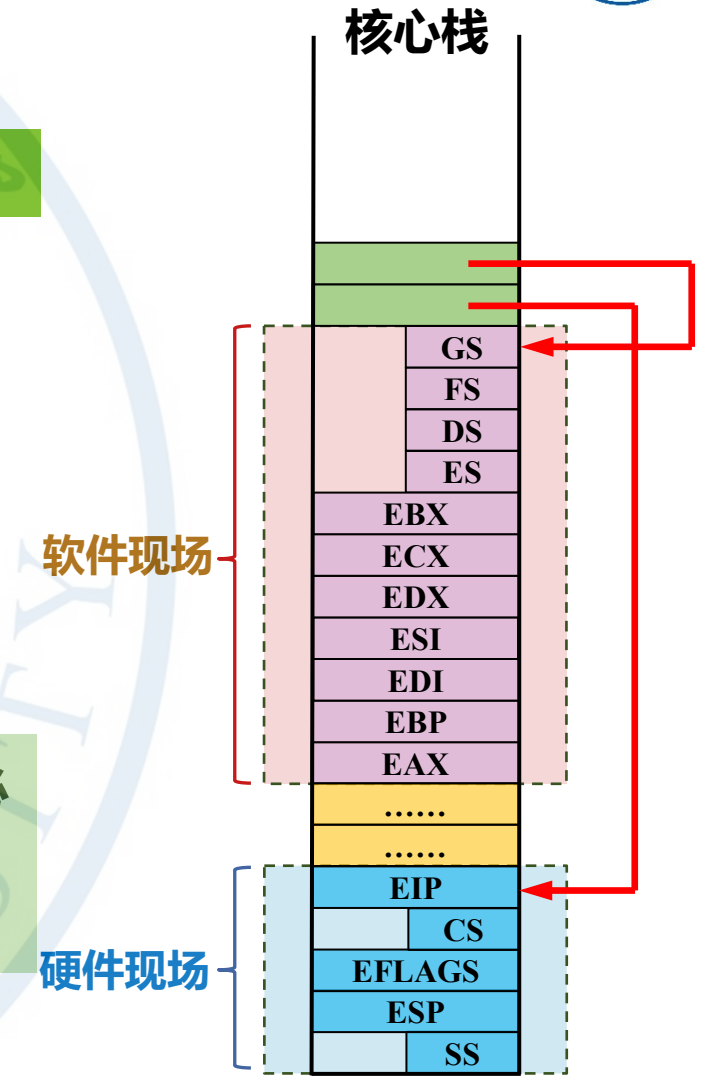
```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();                    ❶

    /* 完全进入核心态 */
    SwitchToKernel();                 ❷

    /* 调用时钟中断处理程序 */
    CallHandler(Time, Clock);

    Leave();
    /* 退出中断 */
    InterruptReturn();
}
```

通过一组汇编指令，将CPU中用于寻址的段寄存器全部改为核心态
（*中断隐指令仅对CS赋值，指向核心态代码段的描述符*）

SwitchToKernel （内联汇编）

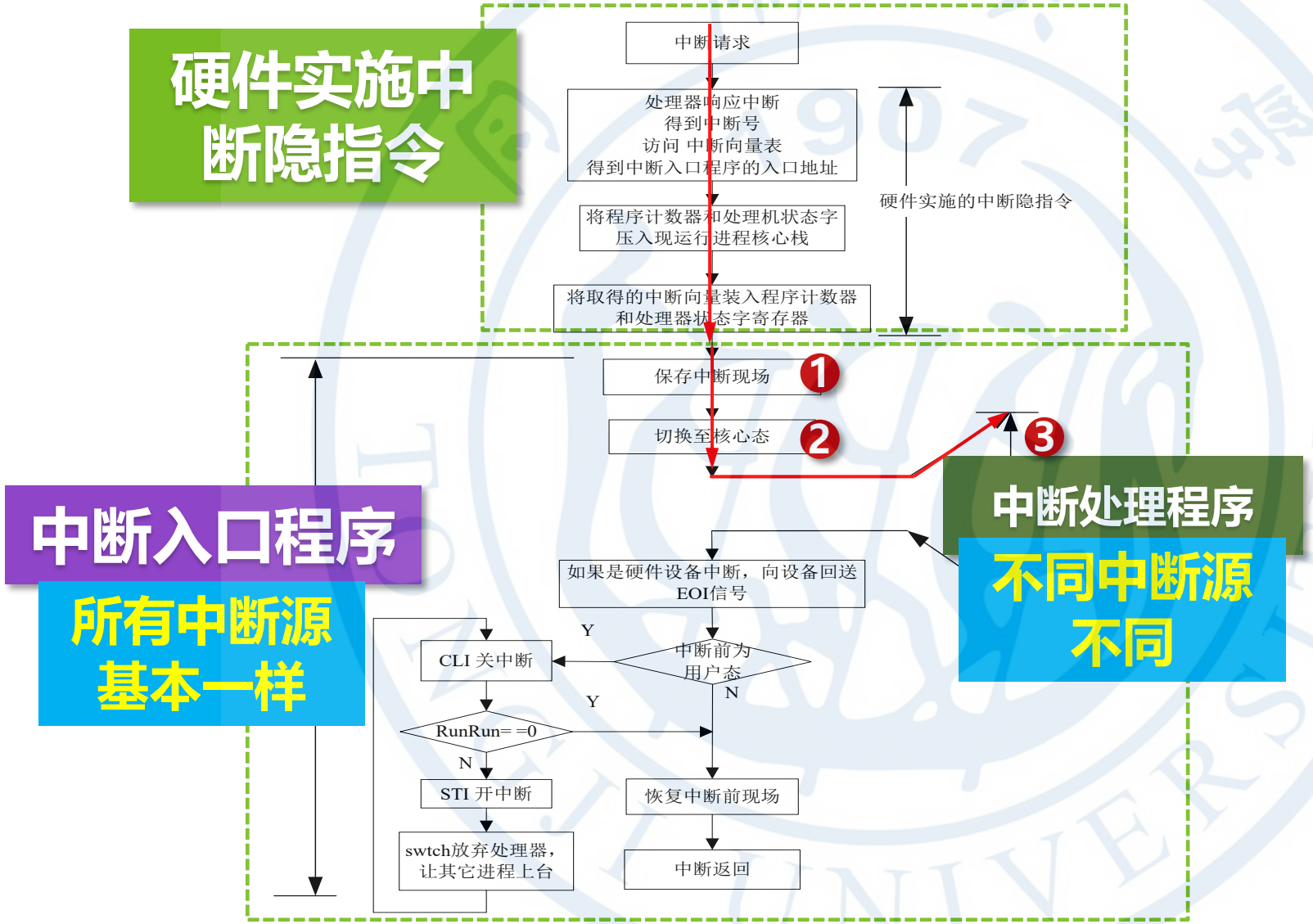
```
#define SwitchToKernel() \
__asm__ __volatile__("
        mov $0x10, %%dx;\
        mov %%dx, %%ds;\
        mov %%dx, %%ss"::);
```

在进入中断处理程序之前，必须用指令对DS和ES赋值，所有段描述符转向核心态。



中断发生时现运行进程在用户态运行

UNIX中断处理流程



执行中断隐指令

跳转到中断入口程序

执行中断入口程序的前半部分

跳转到中断处理程序

UNIX中断处理流程

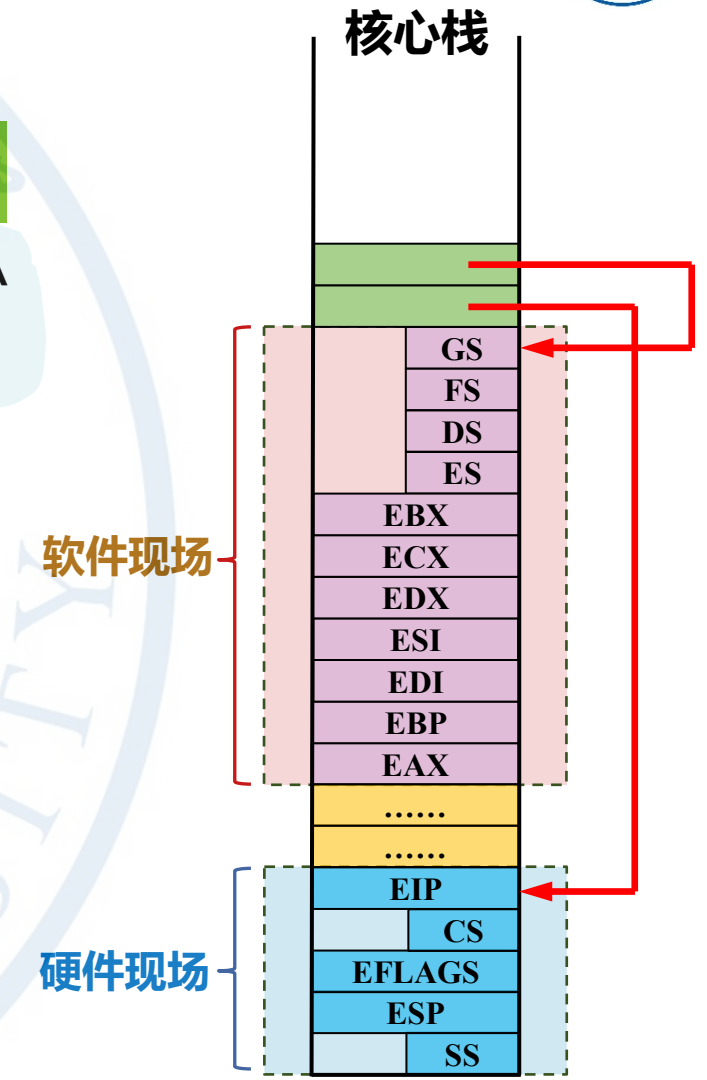
（3）：调用中断处理程序

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();                    ❶

    /* 完全进入核心态 */
    SwitchToKernel();                 ❷

    /* 调用时钟中断处理程序 */
    CallHandler(Time, Clock);         ❸

    /* 恢复现场 */
    RestoreContext();
    /* 手工销毁栈帧 */
    Leave();
    /* 退出中断 */
    InterruptReturn();
}
```

通过一组汇编指令，实现指令跳转，转向不同中断处理程序

CallHandler（内联汇编）

```
#define CallHandler(Class, Handler) \
__asm__ __volatile__("
call *%%%eax" :: "a" (Class::Handler) );
```



中断发生时现运行进程在用户态运行

UNIX中断处理流程

（3）：调用中断处理程序

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();                    ❶

    /* 完全进入核心态 */
    SwitchToKernel();                 ❷

    /* 调用时钟中断处理程序 */
    CallHandler(Time, Clock);         ❸

    RestoreContext();
    /* 手工销毁栈帧 */
    Leave();
    /* 退出中断 */
    InterruptReturn();
}
```

通过一组汇编指令，实现指令跳转，转向不同中断处理程序

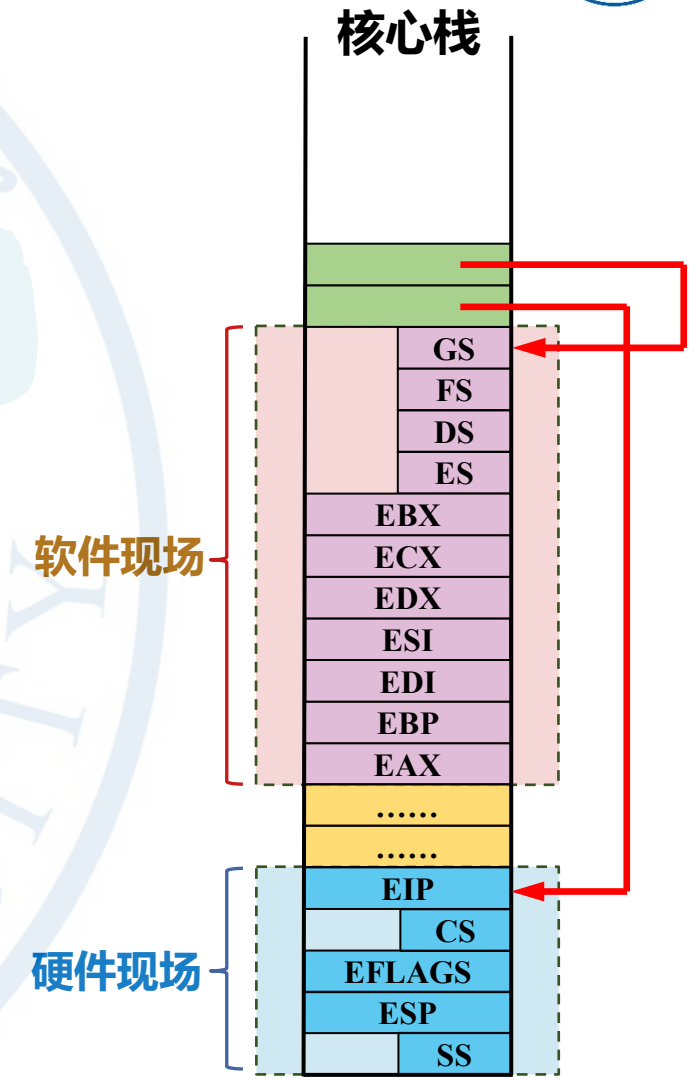
CallHandler（内联汇编）

```
#define CallHandler(Class, Handler) \
__asm__ __volatile__("
call *%%%eax" :: "a" (Class::Handler) );
```

以时钟中断为例：

宏CallHandler(Time, Clock)

```
mov   Time::Clock, eax
call  %eax
```



中断发生时现运行进程在用户态运行

UNIX中断处理流程

（3）：调用中断处理程序

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();                          ❶

    /* 完全进入核心态 */
    SwitchToKernel();                       ❷

    /* 调用时钟中断处理程序 */
    CallHandler(Time, Clock);               ❸

    /* 恢复现场 */
    RestoreContext();
    /* 手工销毁栈帧 */
```

通过一组汇编指令，实现指令跳转，转向不同中断处理程序

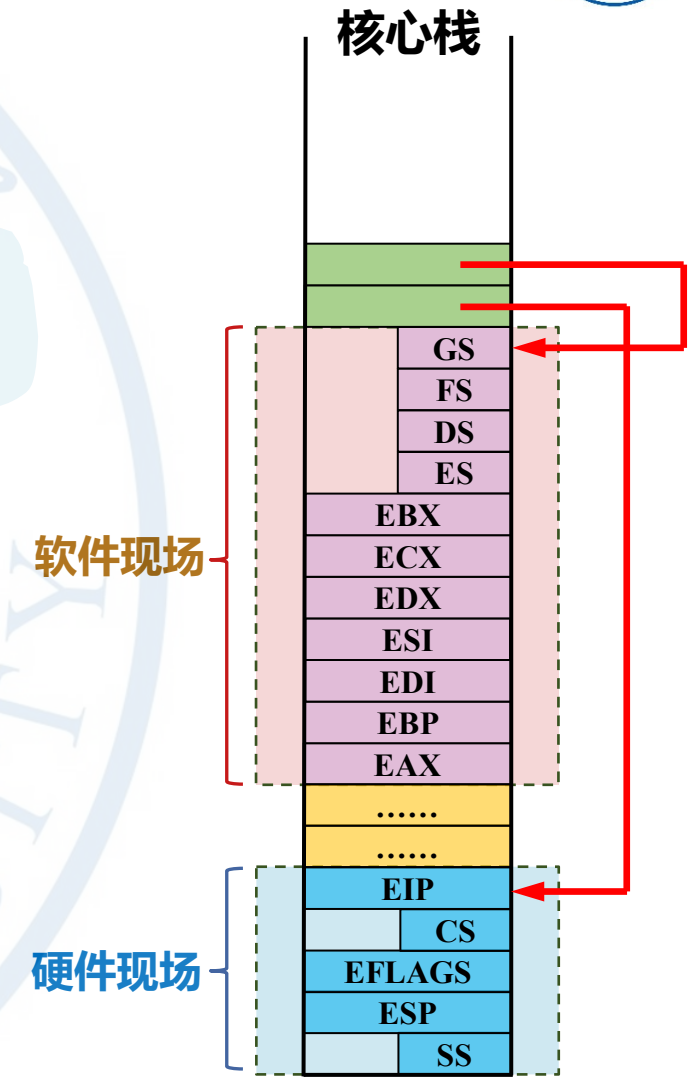
CallHandler（内联汇编）

```
#define CallHandler(Class, Handler) \
__asm__ __volatile__("
call *%%%eax" :: "a" (Class::Handler) );
```

以时钟中断为例：

宏CallHandler(Time, Clock)

```
mov   Time::Clock, eax
call  %eax
```



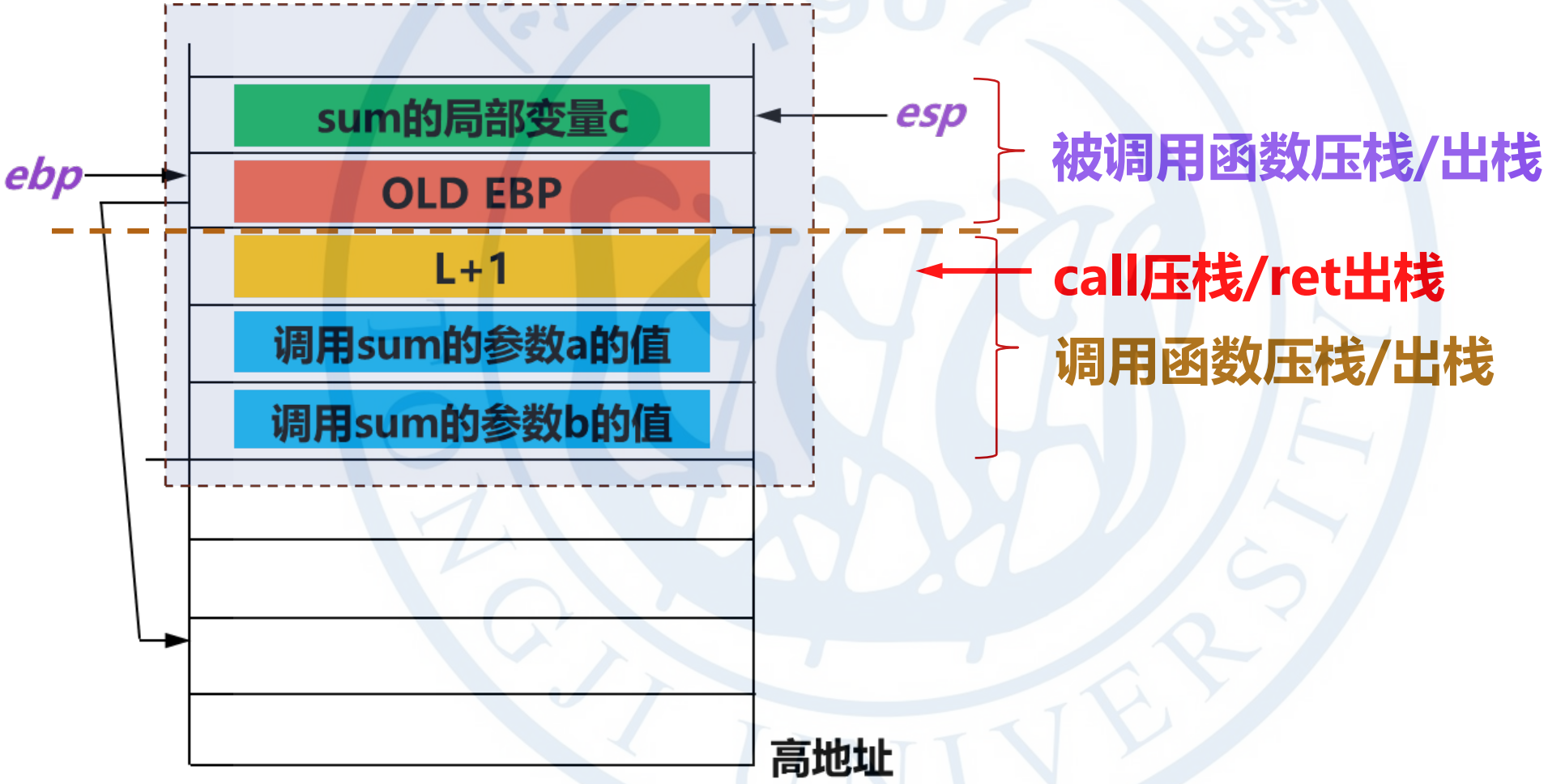
中断发生时现运行进程在用户态运行

中断入口程序执行到此，全是汇编指令。call之后，CPU执行中断处理程序（又称设备处理程序，由C++语言编写）。

所以，……（这里是汇编程序调用了C程序）

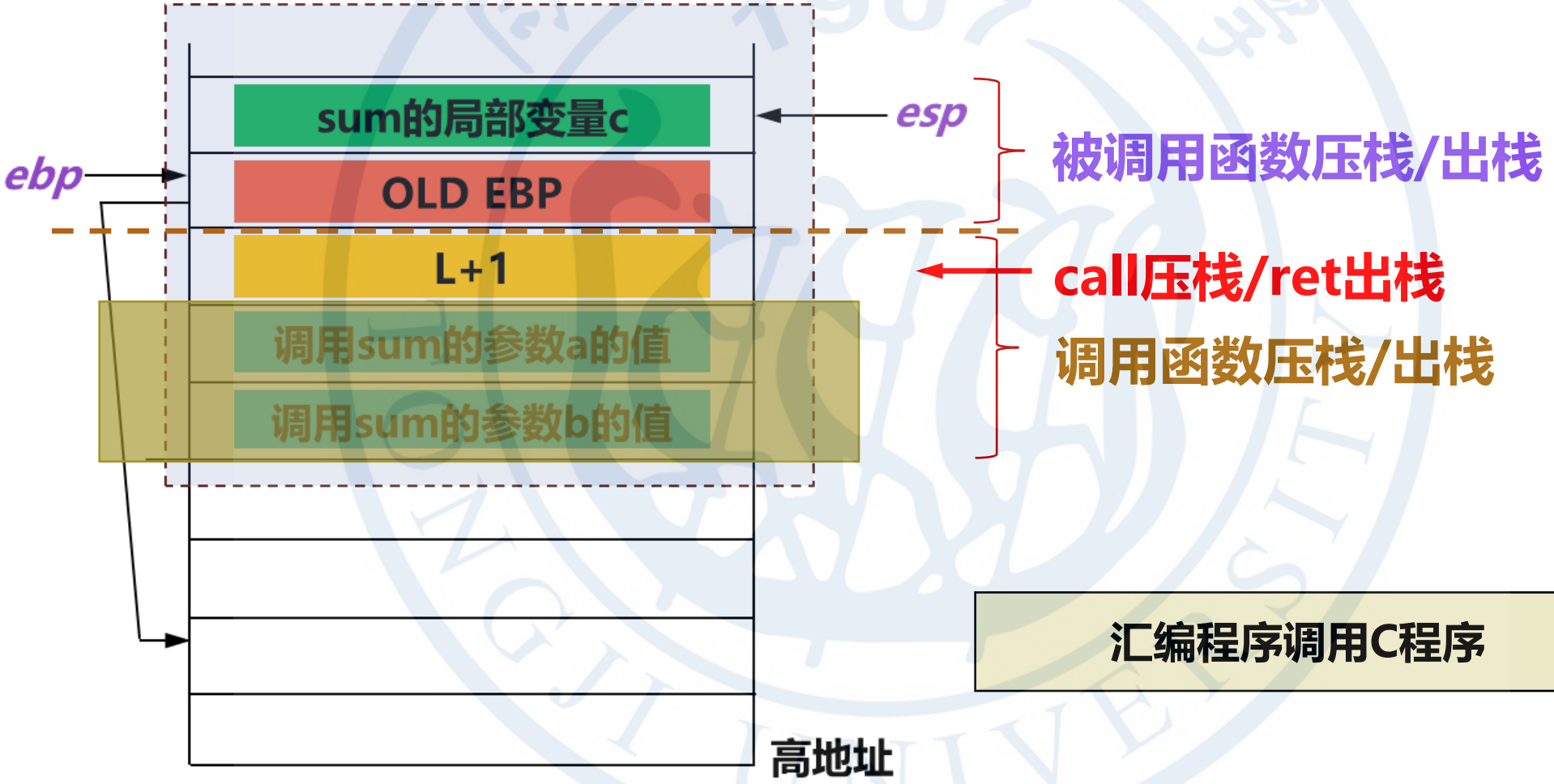
UNIX中断处理流程

先来复习一下C语言中对栈帧的处理



UNIX中断处理流程

先来复习一下C语言中对栈帧的处理



汇编程序调用C程序

UNIX中断处理流程

（3）：调用中断处理程序

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();                    ❶

    /* 完全进入核心态 */
    SwitchToKernel();                 ❷

    /* 调用时钟中断处理程序 */
    CallHandler(Time, Clock);         ❸

    /* 恢复现场 */
    RestoreContext();
    /* 手工销毁栈帧 */
```

通过一组汇编指令，实现指令跳转，转向不同中断处理程序

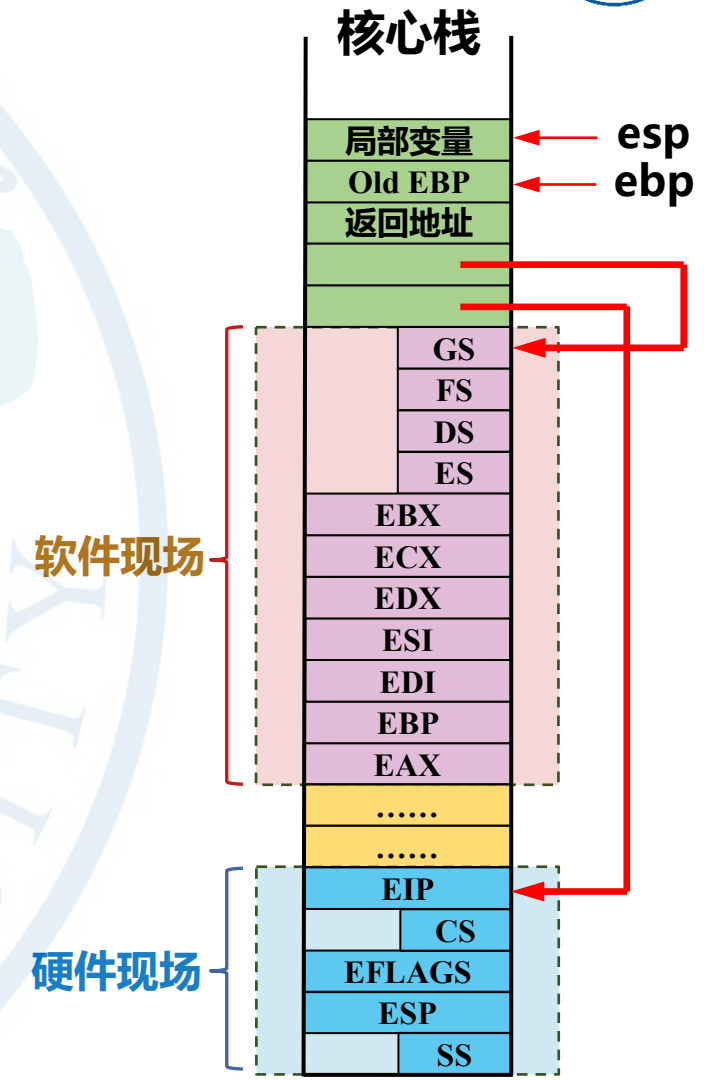
CallHandler（内联汇编）

```
#define CallHandler(Class, Handler) \
__asm__ __volatile__("
call *%%%eax" :: "a" (Class::Handler) );
```

以时钟中断为例：

宏CallHandler(Time, Clock)

```
mov   Time::Clock, eax
call  %eax
```



中断发生时现运行进程在用户态运行

中断入口程序执行到此，全是汇编指令。call之后，CPU执行中断处理程序（又称设备处理程序，由C++语言编写）。

所以，……（这里是汇编程序调用了C程序）

UNIX中断处理流程

中断入口程序

（3）：调用中断处理程序

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();                    ❶

    /* 完全进入核心态 */
    SwitchToKernel();                 ❷

    /* 调用时钟中断处理程序 */
    CallHandler(Time, Clock);         ❸

    RestoreContext();
}
```

通过一组汇编指令，实现指令跳转，转向不同中断处理程序

CallHandler（内联汇编）

```
#define CallHandler(Class, Handler) \
__asm__ __volatile__("
call *%%%eax" :: "a" (Class::Handler) );
```

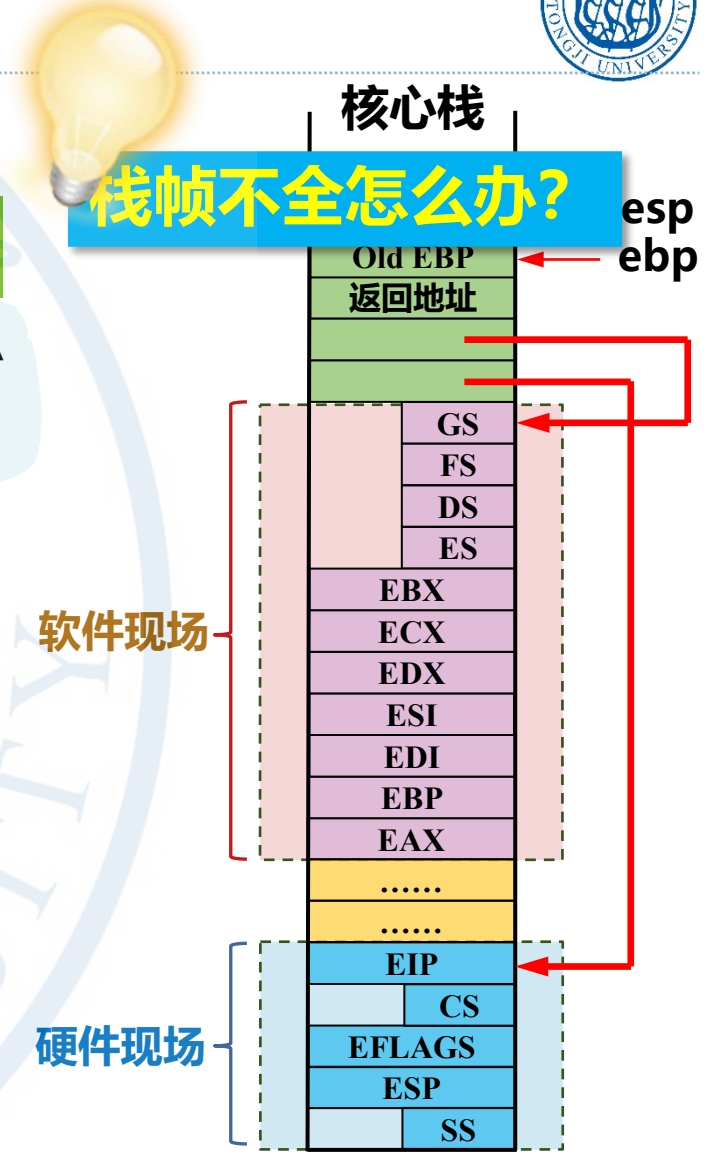
以时钟中断为例：

宏CallHandler(Time, Clock)

```
mov   Time::Clock, eax
call  %eax
```

中断入口程序执行到此，全是汇编指令。call之后，CPU执行中断处理程序（又称设备处理程序，由C++语言编写）。

所以，……（这里是汇编程序调用了C程序）



中断发生时现运行进程在用户态运行

UNIX中断处理流程

1.先定义两个结构体

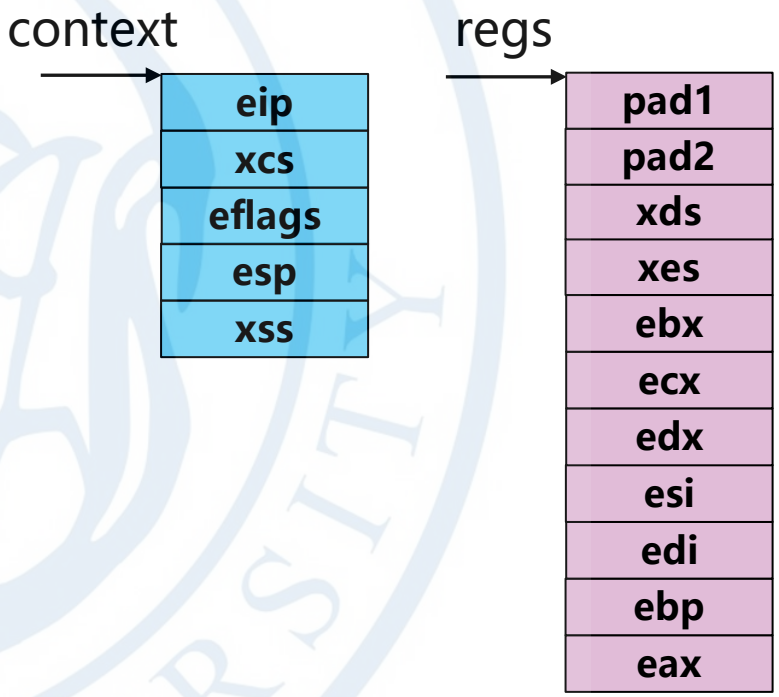
```
struct pt_context{
        unsigned int eip;
        unsigned int xcs;
        unsigned int eflags;
        unsigned int esp;
        unsigned int xss;
};
```

```
struct pt_regs{
        unsigned int pad1;
        unsigned int pad2;
        unsigned int xds;
        unsigned int xes;
        unsigned int ebx;
        unsigned int ecx;
        unsigned int edx;
        unsigned int esi;
        unsigned int edi;
        unsigned int ebp;
        unsigned int eax;
};
```

如果程序中声明：

struct pt_regs* regs;

struct pt_context* context;



程序中可使用： regs->eax regs->ebx ……

context->eip context->xcs ……

访问各个分量。

UNIX中断处理流程

1.先定义两个结构体

```
struct pt_context{
        unsigned int eip;
        unsigned int xcs;
        unsigned int eflags;
        unsigned int esp;
        unsigned int xss;
};
```

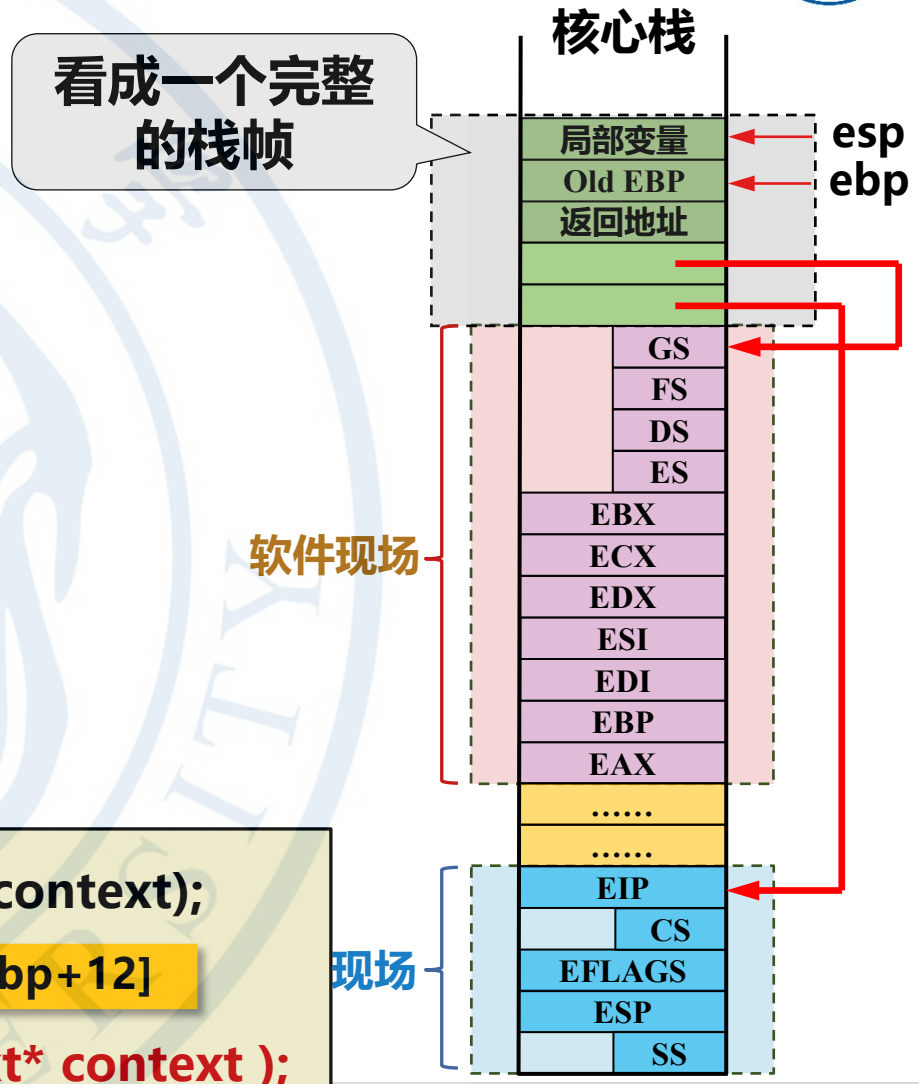
```
struct pt_regs{
        unsigned int pad1;
        unsigned int pad2;
        unsigned int xds;
        unsigned int xes;
        unsigned int ebx;
        unsigned int ecx;
        unsigned int edx;
        unsigned int esi;
        unsigned int edi;
        unsigned int ebp;
```

2. 再用这样统一的方式声明中断处理程序：

```
void Handler( struct pt_regs* regs,  struct pt_context* context);
```

以时钟中断为例：regs被汇编成[%ebp+8];context被汇编成[%ebp+12]

```
void Time::Clock( struct pt_regs* regs, struct pt_context* context );
```



UNIX中断处理流程

1.先定义两个结构体

```
struct pt_context{
        unsigned int eip;
        unsigned int xcs;
        unsigned int eflags;
        unsigned int esp;
        unsigned int xss;
};
```

```
struct pt_regs{
        unsigned int pad1;
        unsigned int pad2;
        unsigned int xds;
        unsigned int xes;
        unsigned int ebx;
        unsigned int ecx;
        unsigned int edx;
        unsigned int esi;
        unsigned int edi;
        unsigned int ebp;
```

2. 再用这样统一的方式声明中断处理程序：

```
void Handler( struct pt_regs* regs,  struct pt_context* context);
```

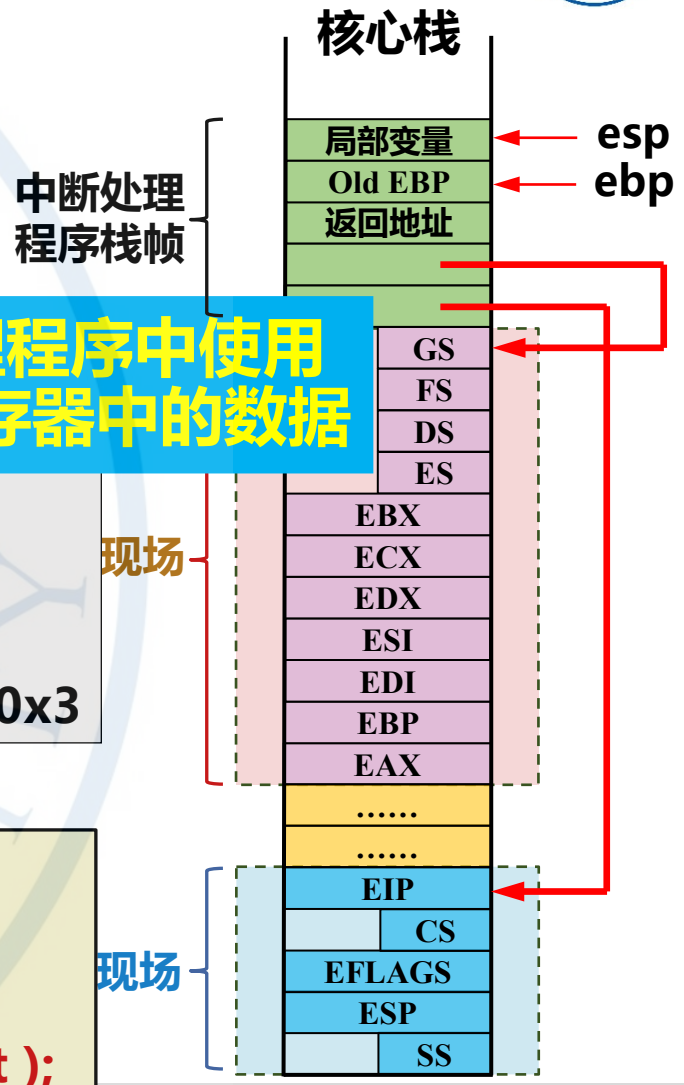
以时钟中断为例：regs被汇编成[%ebp+8];context被汇编成[%ebp+12]

```
void Time::Clock( struct pt_regs* regs, struct pt_context* context );
```

3. 在中断处理程序中使用现场保护的寄存器中的数据

例：

```
regs->eax
regs->ebx
context->xcs & 0x3
```



中断发生时现运行进程在用户态运行

UNIX中断处理流程

中断入口程序

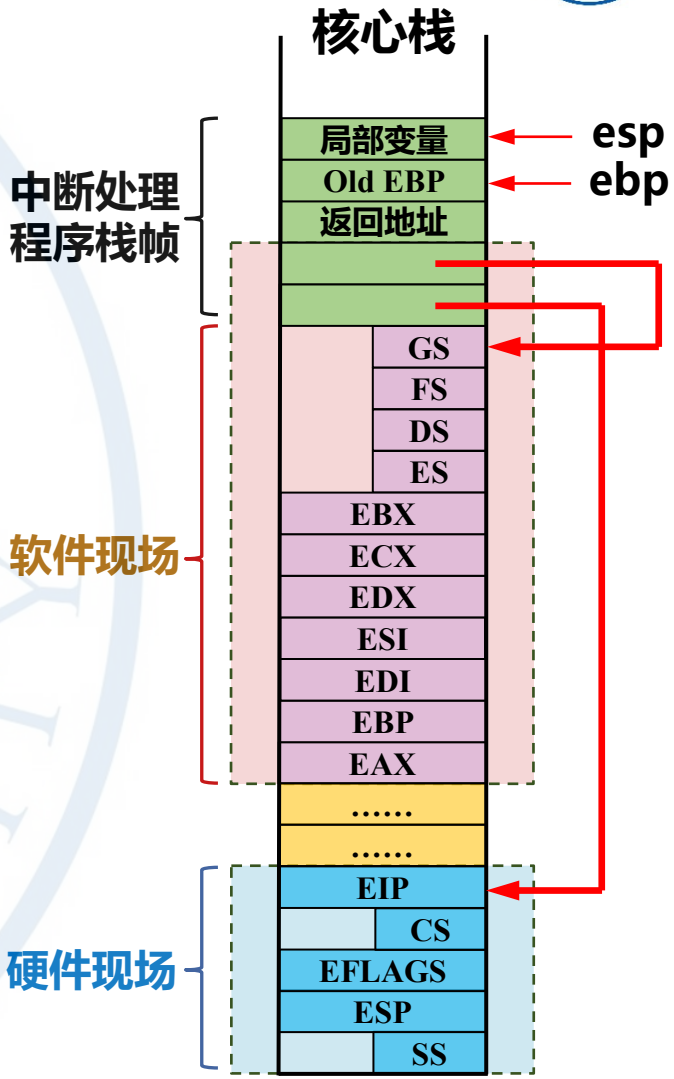
(3)：调用中断处理程序

不用两个指针，采用如下方式声明参数：

```
void Handler( int gs, int fs, int ds, ..., int eflags, int esp, int ss);
```

同样可以达到访问现场信息的目的，但是：

1. 参数表太长
2. 如果软件现场发生变化。。。。



中断发生时现运行进程在用户态运行

UNIX中断处理流程

(3)：调用中断处理程序

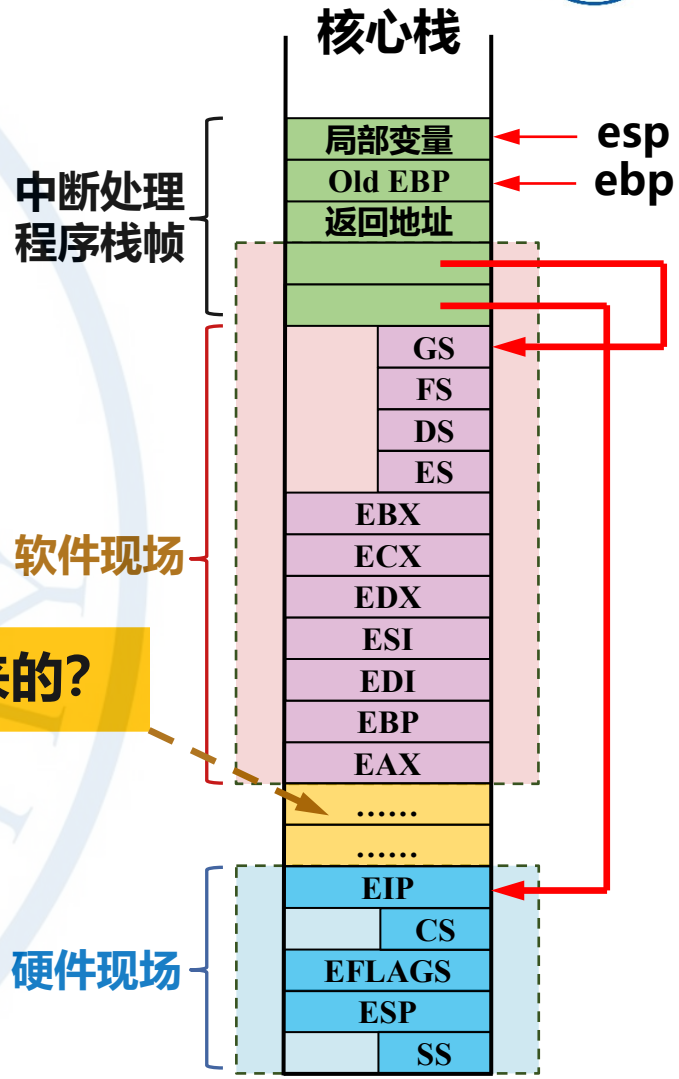
不用两个指针，采用如下方式声明参数：

```
void Handler( int gs, int fs, int ds, ..., int eflags, int esp, int ss);
```

同样可以达到访问现场信息的目的，但是：

1. 参数表太长
2. 如果软件现场发生变化。。。。
3. 黄色区域部分。。。

这是什么？从哪里来的？



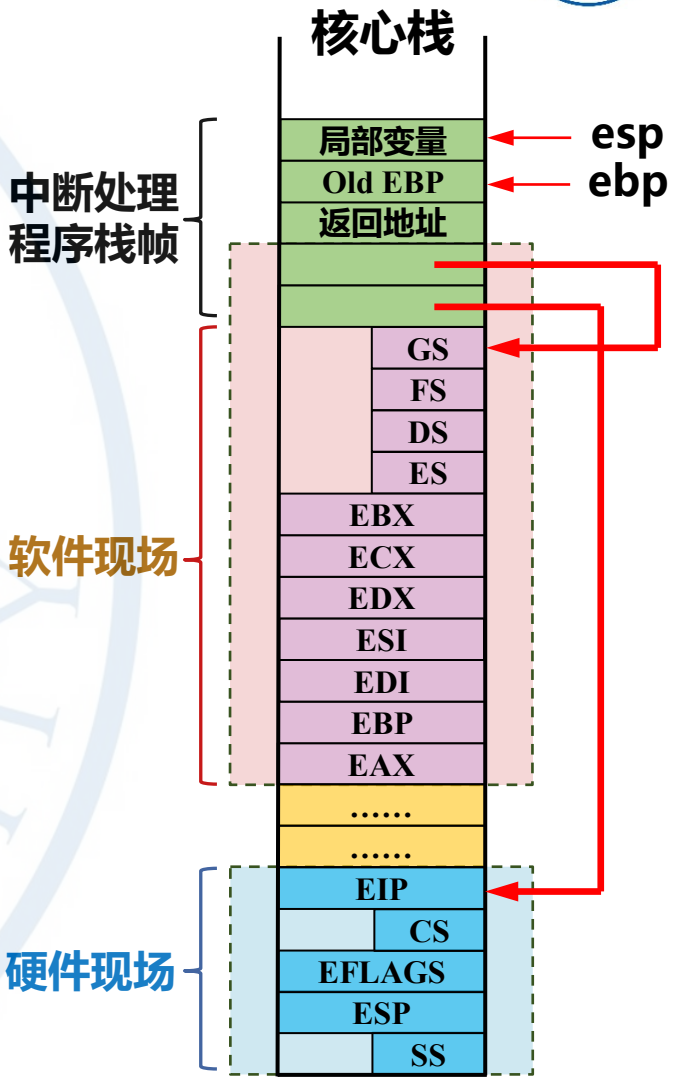
中断发生时现运行进程在用户态运行

UNIX中断处理流程

中断入口程序 ← **中断隐指令**

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();

    /* 完全进入核心态 */
    SwitchToKernel();

    /* 调用时钟中断处理程序 */
    CallHandler(Time, Clock);

    /* 例行调度 */
    ……

     /* 恢复现场 */
    RestoreContext();
     /* 手工销毁栈帧 */
    Leave();
    /* 退出中断 */
    InterruptReturn();
}
```

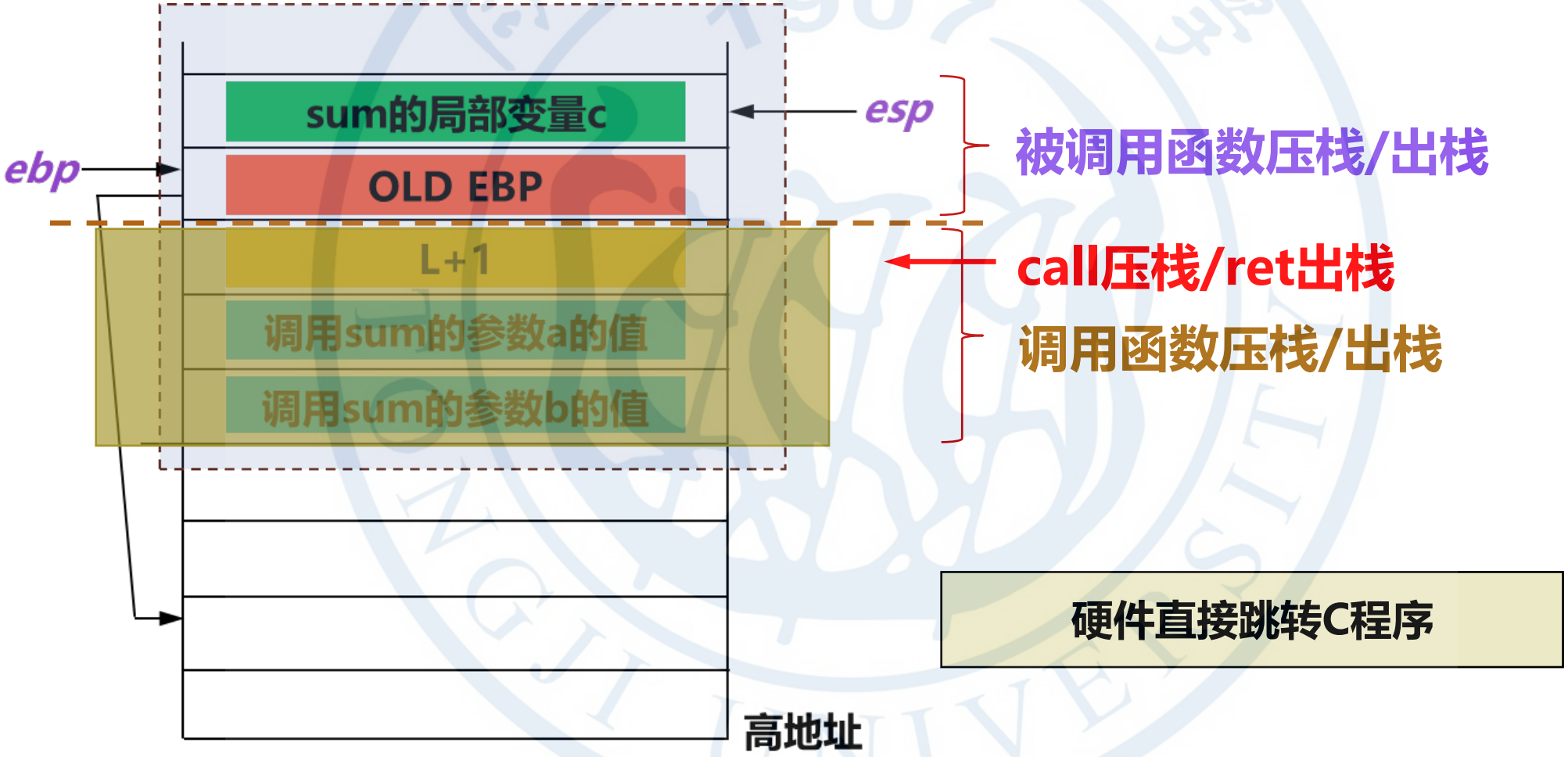
由中断隐指令进入C++语言编写的中断入口程序，所以，这里是机器指令跳转到C程序



中断发生时现运行进程在用户态运行

UNIX中断处理流程

先来复习一下C语言中对栈帧的处理



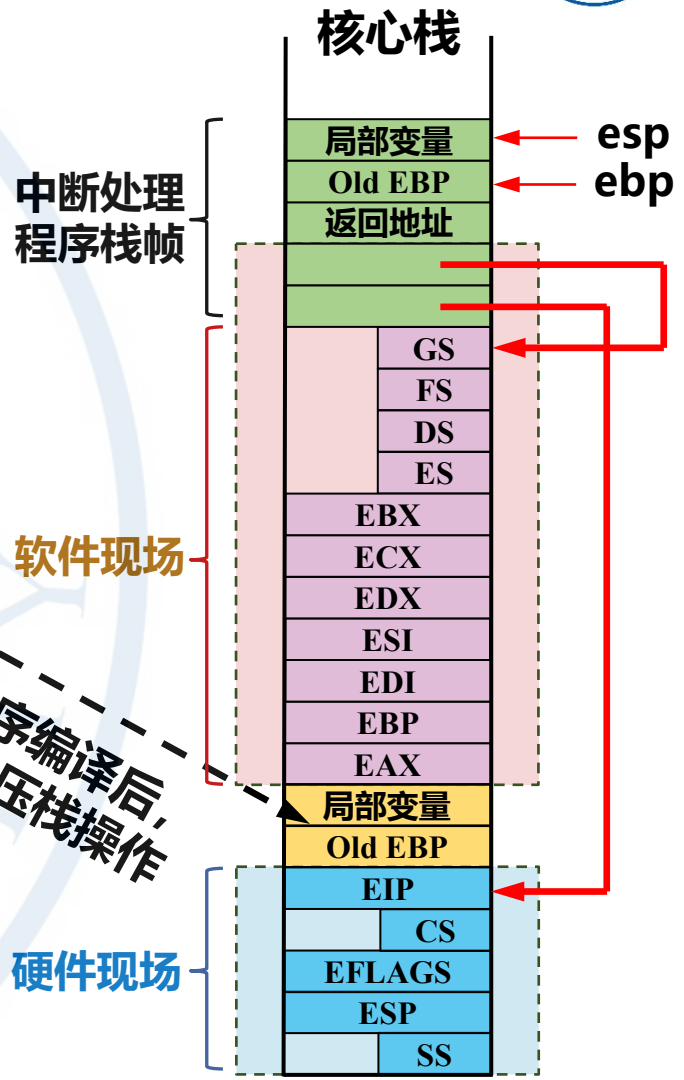
UNIX中断处理流程

中断入口程序 ← **中断隐指令**

```
void Time::TimeInterruptEntrance()
{
    /* 保存中断现场*/
    SaveContext();

    /* 完全进入核心态 */
    SwitchToKernel();

    /* 调用时钟中断处理程序 */
    CallHandler(Time, Clock);

    /* 例行调度 */
    ……

     /* 恢复现场 */
    RestoreContext();
     /* 手工销毁栈帧 */
    Leave();
    /* 退出中断 */
    InterruptReturn();
}
```

由中断隐指令进入C++语言编写的中断入口程序，所以，这里是机器指令跳转到C程序

C++编写的中断入口程序编译后，完成的局部变量和EBP的压栈操作



中断发生时现运行进程在用户态运行

UNIX中断处理流程

(3)：调用中断处理程序

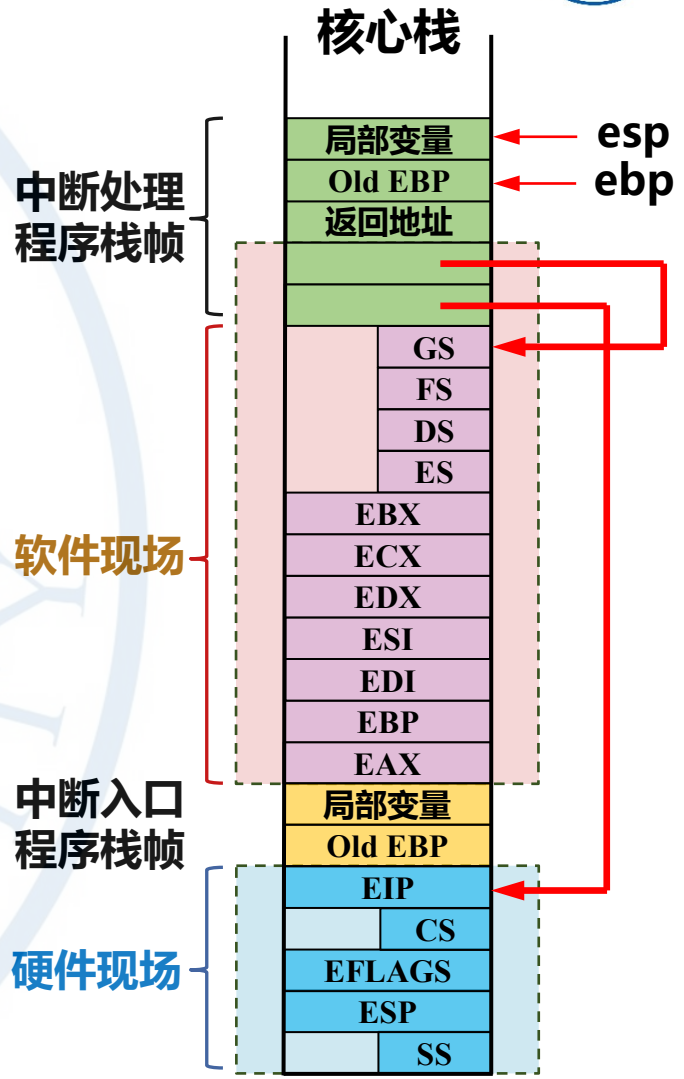
不用两个指针，采用如下方式声明参数：

```
void Handler( int gs, int fs, int ds, ..., int eflags, int esp, int ss);
```

同样可以达到访问现场信息的目的，但是：

1. 参数表太长
2. 如果软件现场发生变化。。。。
3. 黄色区域部分。。。

```
void Handler( struct pt_regs* regs,  struct pt_context* context);
```



中断发生时现运行进程在用户态运行

本节小结

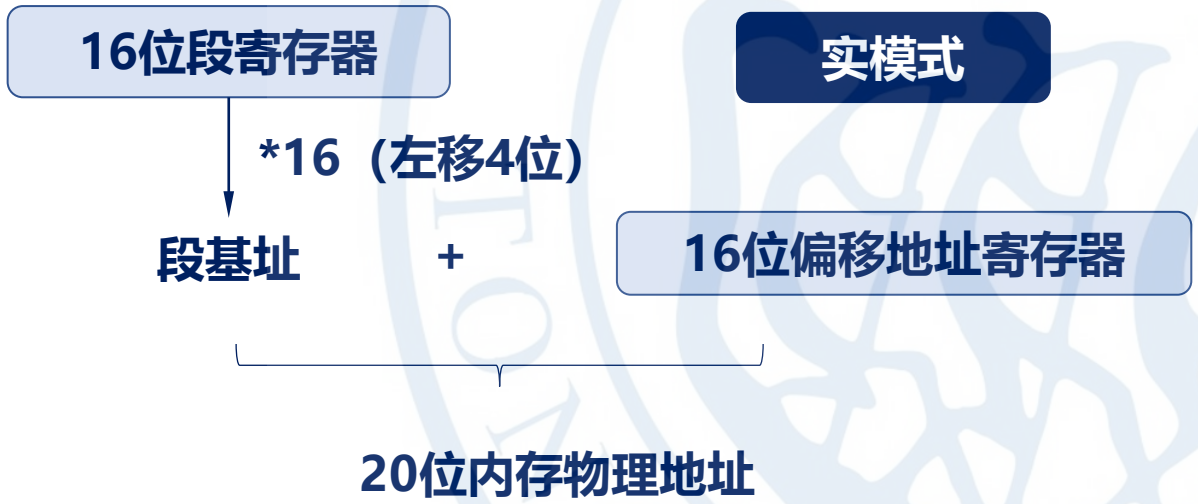
1　中断的基本概念

2　UNIX中断的处理过程

阅读讲义：84页 ~ 103页

实模式与保护模式

段寄存器诞生之初是因为8086的CPU内部寄存器为16位，而地址线为20位。利用段寄存器解决寻址大小不一致的问题。



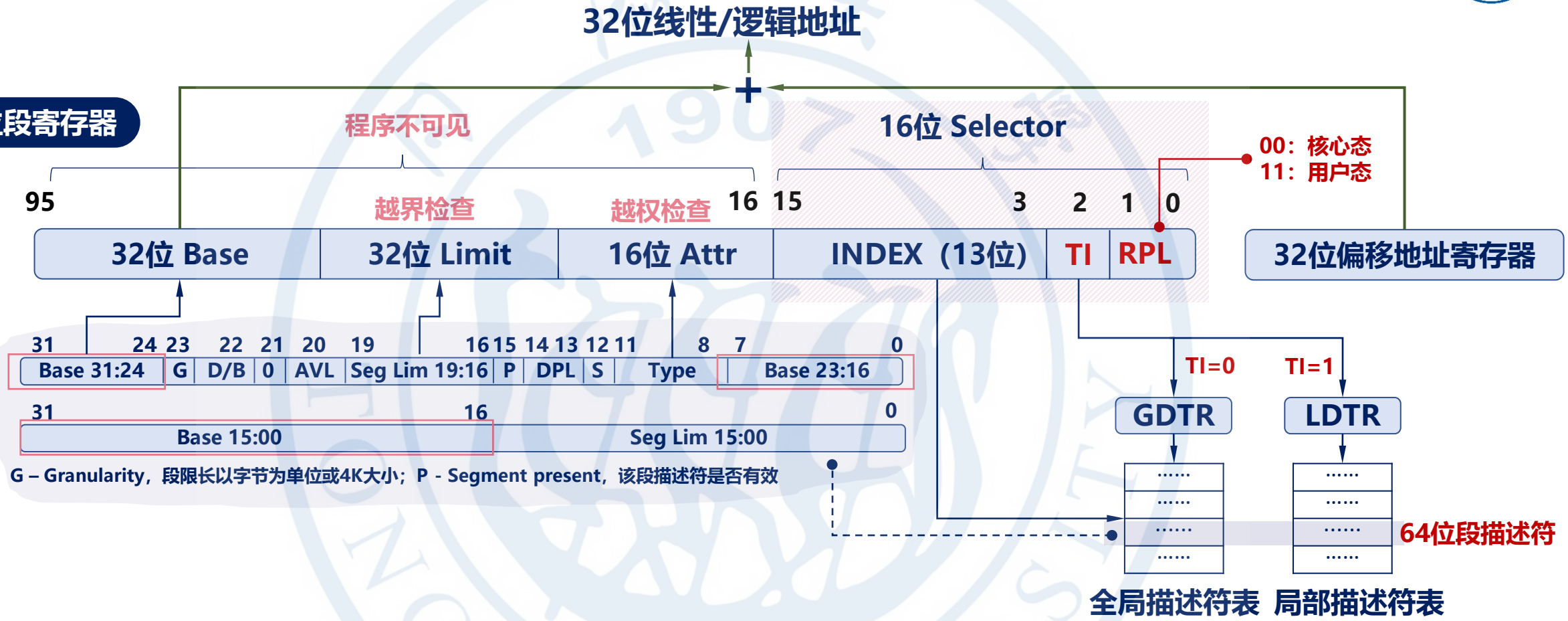
代码段寄存器：CS

数据段寄存器：DS

堆栈段寄存器：SS

实模式与保护模式

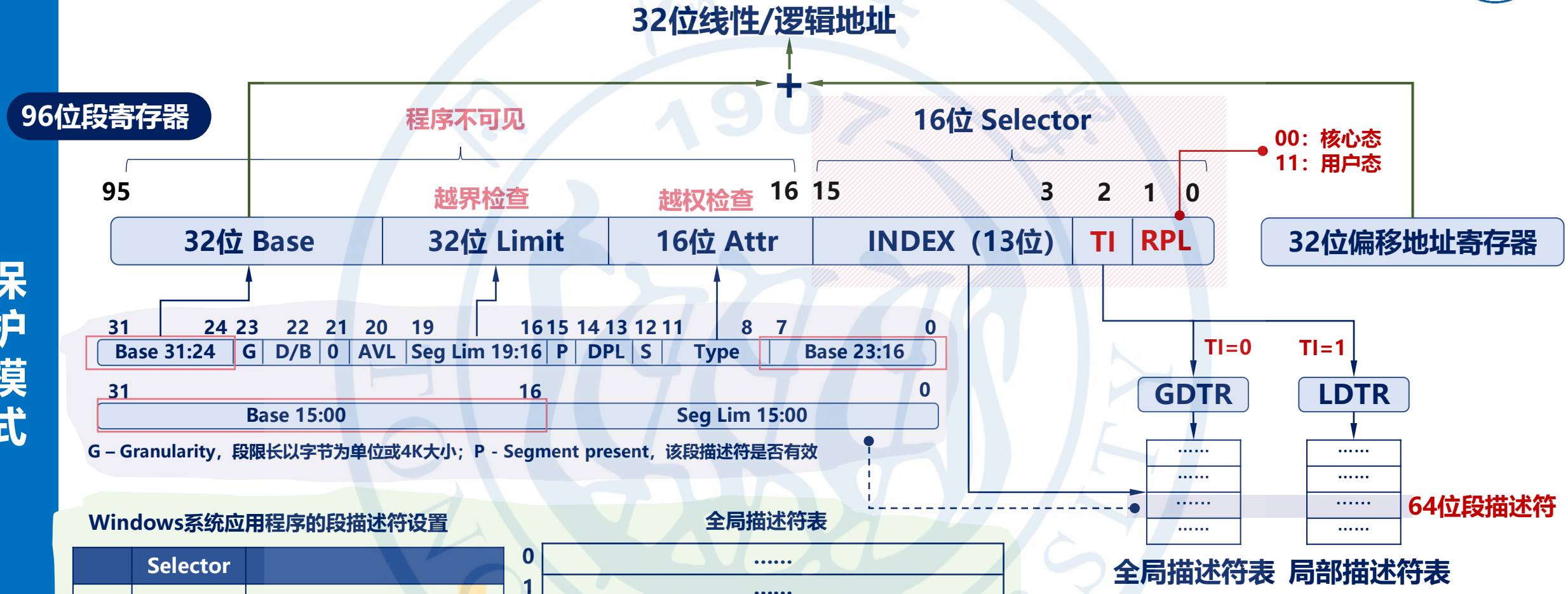
保护模式



- *描述符表在内存中，长度可变*
- *最多包含8192 (2^13)个描述符*
- *分别由GDTR和LDTR寄存器保存起始地址*

实模式与保护模式

保护模式



Windows系统应用程序的段描述符设置

	Selector	
CS	001B	0000 0000 0001 1011
ES	0023	0000 0000 0010 0011
SS	0023	0000 0000 0010 0011
DS	0023	0000 0000 0010 0011

全局描述符表

0	……
1	……
2	……
3	Base：0；Limit：0xFFFFFFFF；可读，可执行
4	Base：0；Limit：0xFFFFFFFF；可读，可写
5	……

- *描述符表在内存中，长度可变*
- *最多包含8192 (2^13)个描述符*
- *分别由GDTR和LDTR寄存器保存起始地址*