第四章

进程管理

主 要 内 容

4.1 UNIX时钟中断与异常

4.2 UNIX系统调用

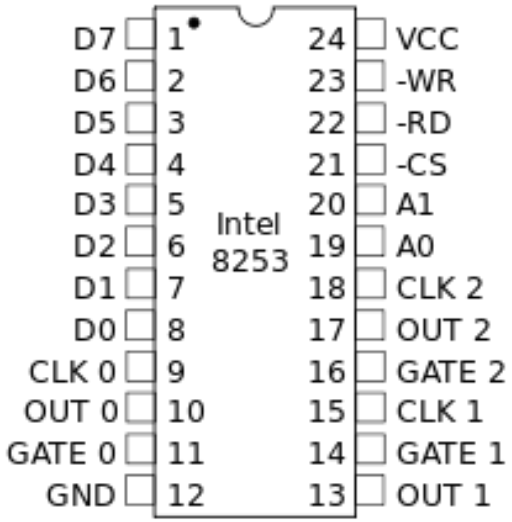
4.3 UNIX的进程调度状态

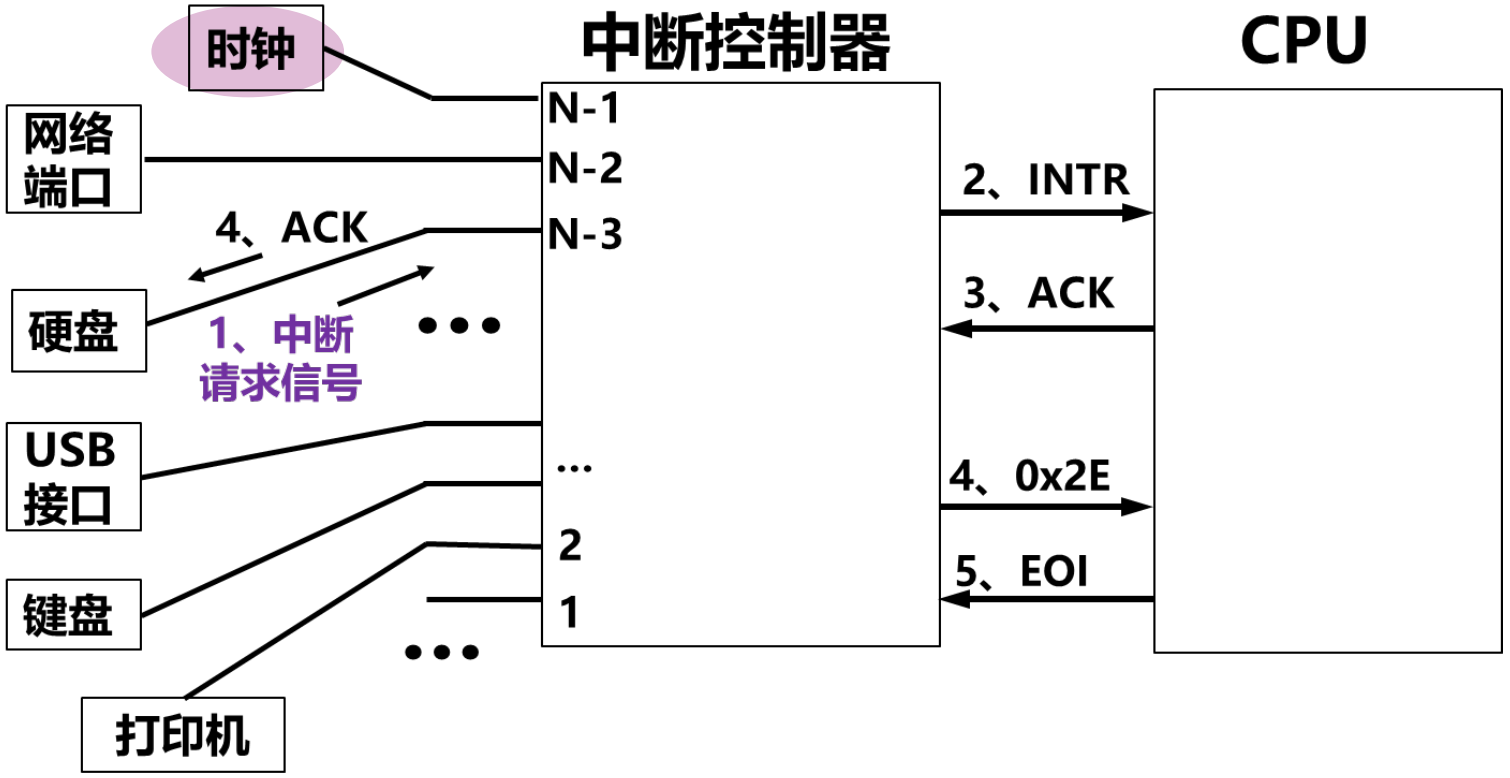
4.4 UNIX进程控制

UNIX时钟中断

PC中的可编程定时芯片：8253或8254
每秒60次定时中断：“心跳”

时钟中断的中断号为：0x20
优先级最高



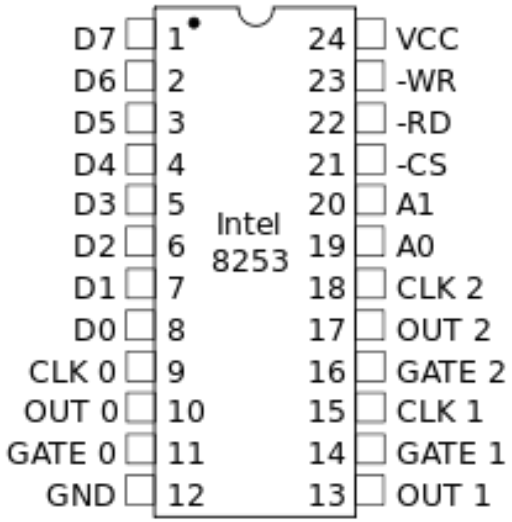


UNIX时钟中断

概况

PC中的可编程定时芯片：8253或8254
每秒60次定时中断：“心跳”

时钟中断的中断号为：0x20
优先级最高

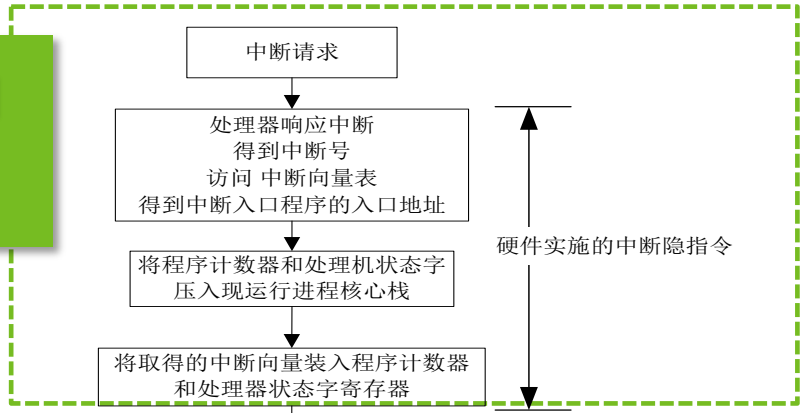



```
class Time
{
public:
    static const int SCHMAG = 10 * 2;          /* 每秒钟减少的p_cpu魔数 */
    static const int HZ = 60;                  /* 每秒钟时钟中断次数 */
    static int lbolt;                          /* 累计接收到的时钟中断次数 */
    static unsigned int time; /* 系统全局时间，自1970年1月1日至今的秒数 */
    static unsigned int tout; /* 各延时睡眠进程中应被唤醒的时刻中最小值 */

     /*时钟中断入口程序，其地址存放在IDT的时钟中断对应中断门中 */
    static void TimeInterruptEntrance();
    /* 时钟中断处理程序 */
    static void Clock(struct pt_regs* regs, struct pt_context* context);
};
```

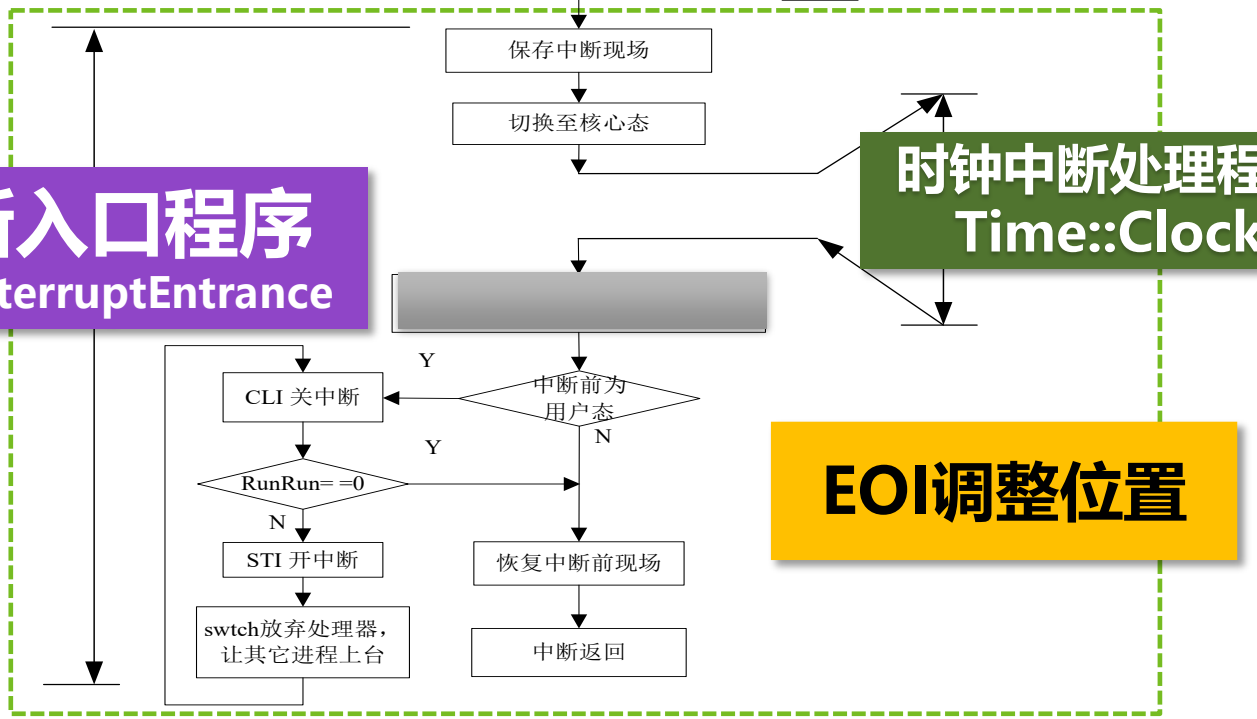
UNIX时钟中断

概况

硬件实施中断隐指令



时钟中断入口程序
Time::TimeInterruptEntrance



时钟中断处理程序
Time::Clock

EOI调整位置

时钟中断中处理所有和时间相关的操作

```
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();
}
```

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 Time::Clock(struct pt_regs* regs, struct pt_context* context)
0x21	键盘中断	KeyboardInterrupt::KeyboardI uptEntrance()	void Keyboard::KeyboardHandler(struct pt_regs* reg, struct pt_context ext)
…	…		
0x2E	硬盘中断	void DiskInterrupt::Disk	void ATADriver::ATAHandler(struct pt_regs *reg, struct pt_context
….	…	…	
0x80	系统调用	void SystemCall::SystemCallEntrance()	void SystemCall::Trap(struct pt_regs* regs, struct pt_context* context)
…	…	…	…

在中断描述符表的0x20号中断处登记地址

入口程序跳转至处理程序

UNIX时钟中断

概况

	名称	类型	含义
进程标识	p_uid	short	用户ID
	p_pid	int	进程标识数，进程编号
	p_ppid	int	父进程标识数
进程图象在内存中的位置信息	p_addr	unsigned long	ppda区在物理内存中的起始地址
	p_size	unsigned int	进程图象（除代码段以外部分）的长度，以字节单位
	p_textp	Text *	指向该进程所运行的代码段的描述符
进程调度相关信息	p_stat	ProcessState	进程当前的调度状态
	p_flag	int	进程标志位，可以将多个状态组合
	p_pri	int	进程优先数（值越大，优先级越小）
	p_cpu	int	cpu值，用于计算p_pri
	p_nice	int	进程优先数微调参数
	p_time	int	进程在盘交换区上（或内存内）的驻留时间
	p_wchan	unsigned long	进程睡眠原因
信号与控制台终端	p_sig	int	进程信号
	p_ttyp	TTy*	进程tty结构地址

UNIX时钟中断

概况

	名称	类型	含义
进程标识	p_uid	short	用户ID
	p_pid	int	进程标识数，进程编号
	p_ppid	int	父进程标识数
进程图象在内存中的位置信息	p_addr	unsigned long	ppda区在物理内存中的起始地址
	p_size	unsigned int	进程图象（除代码段以外部分）的长度，以字节单位
	p_textp	Text *	指向该进程所运行的代码段的描述符
进程调度相关信息	p_stat	ProcessState	进程当前的调度状态
	p_flag	int	进程标志位，可以将多个状态组合
	p_pri	int	进程优先数（值越大，优先级越小）
	p_cpu	int	
	p_nice	int	
	p_time	int	进程在盘交换区上（或内存内）的驻留时间
	p_wchan	unsigned long	进程睡眠原因
信号与控制台终端	p_sig	int	进程信号
	p_ttyp	TTy*	进程tty结构地址

时间每过去1秒，p_time++;
图像每交换一次，p_time = 0

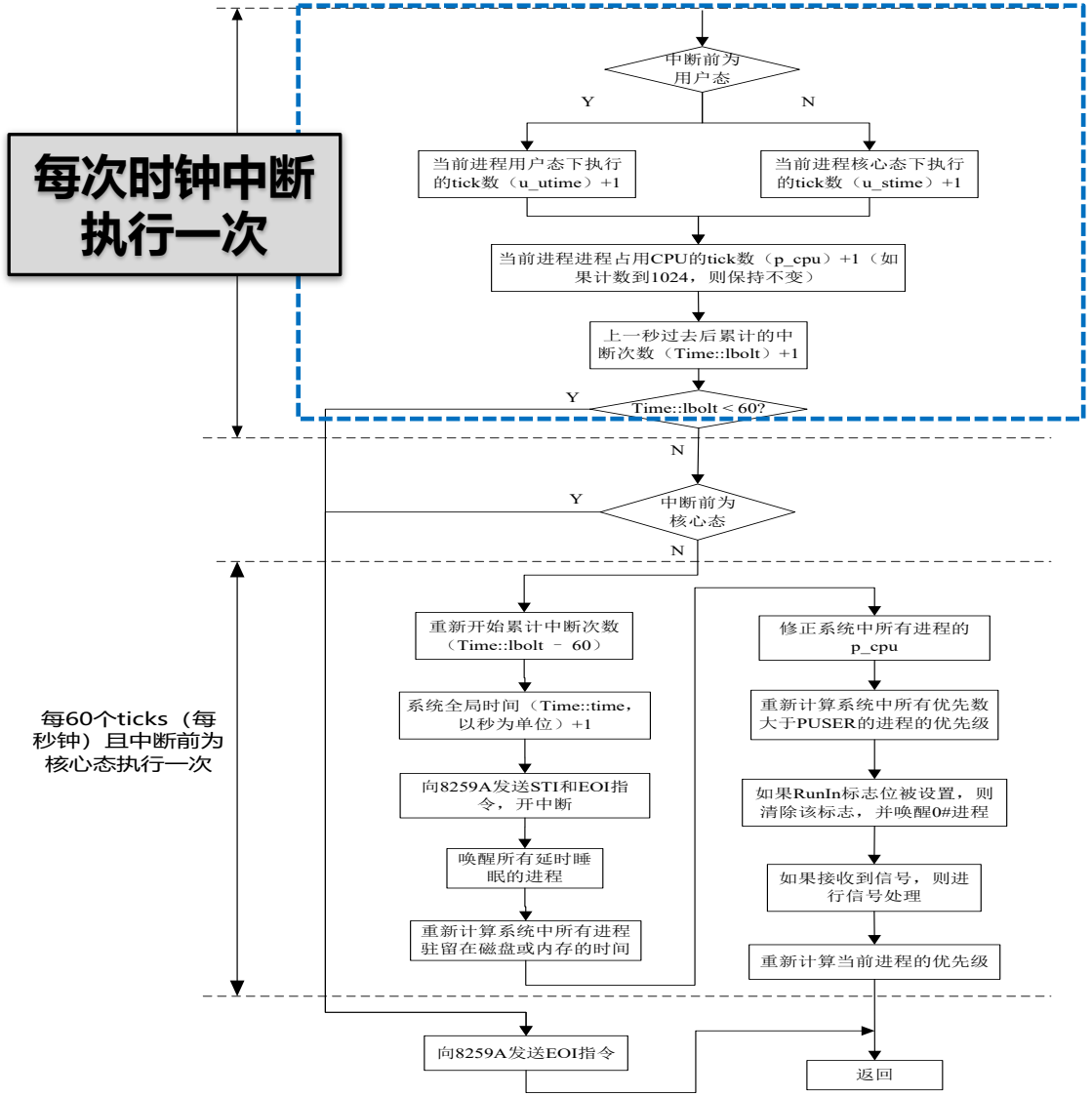
UNIX时钟中断

概况

	名称	类型	含义
进程的用户标识	u_uid	short	有效用户ID
	u_gid	short	有效组ID
	u_ruid	short	真实用户ID
	u_rgid	short	真实组ID
进程的时间相关	u_utime	int	进程用户态时间
	u_stime	int	进程核心态时间
	u_cutime	int	子进程用户态时间总和
	u_cstime	int	子进程核心态时间总和
现场保护相关	u_rsav[2]	unsigned long	用于保存esp与ebp指针
	u_ssav[2]	unsigned long	用于对esp和ebp指针的二次保护
内存管理相关	*u_procp	Process	指向该u结构对应的Process结构
	u_MemoryDescriptor	MemoryDescriptor	封装了进程的图象在内存中的位置、大小等信息
出错	u_error	ErrorCode	存放错误码，具体数值及其含义请查阅源代码

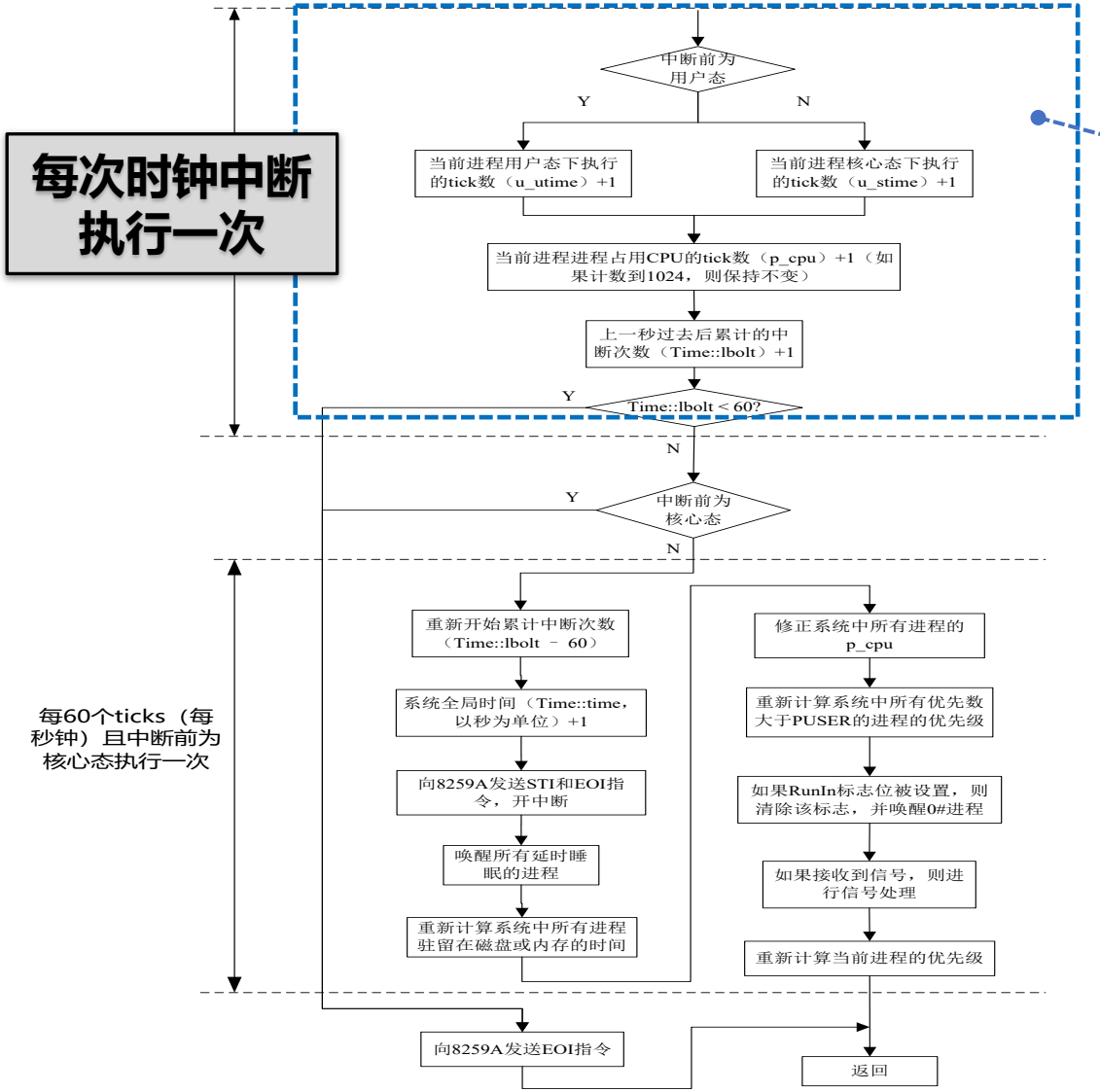
UNIX时钟中断

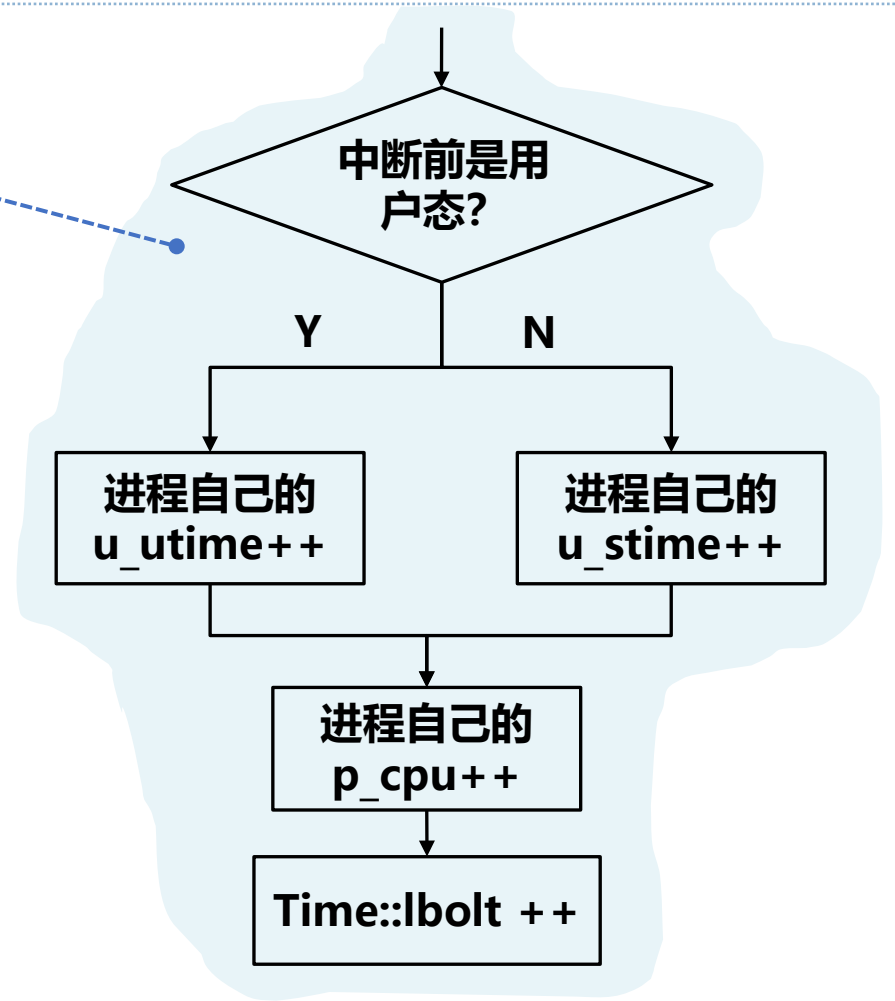
时钟中断处理程序的详细流程



UNIX时钟中断

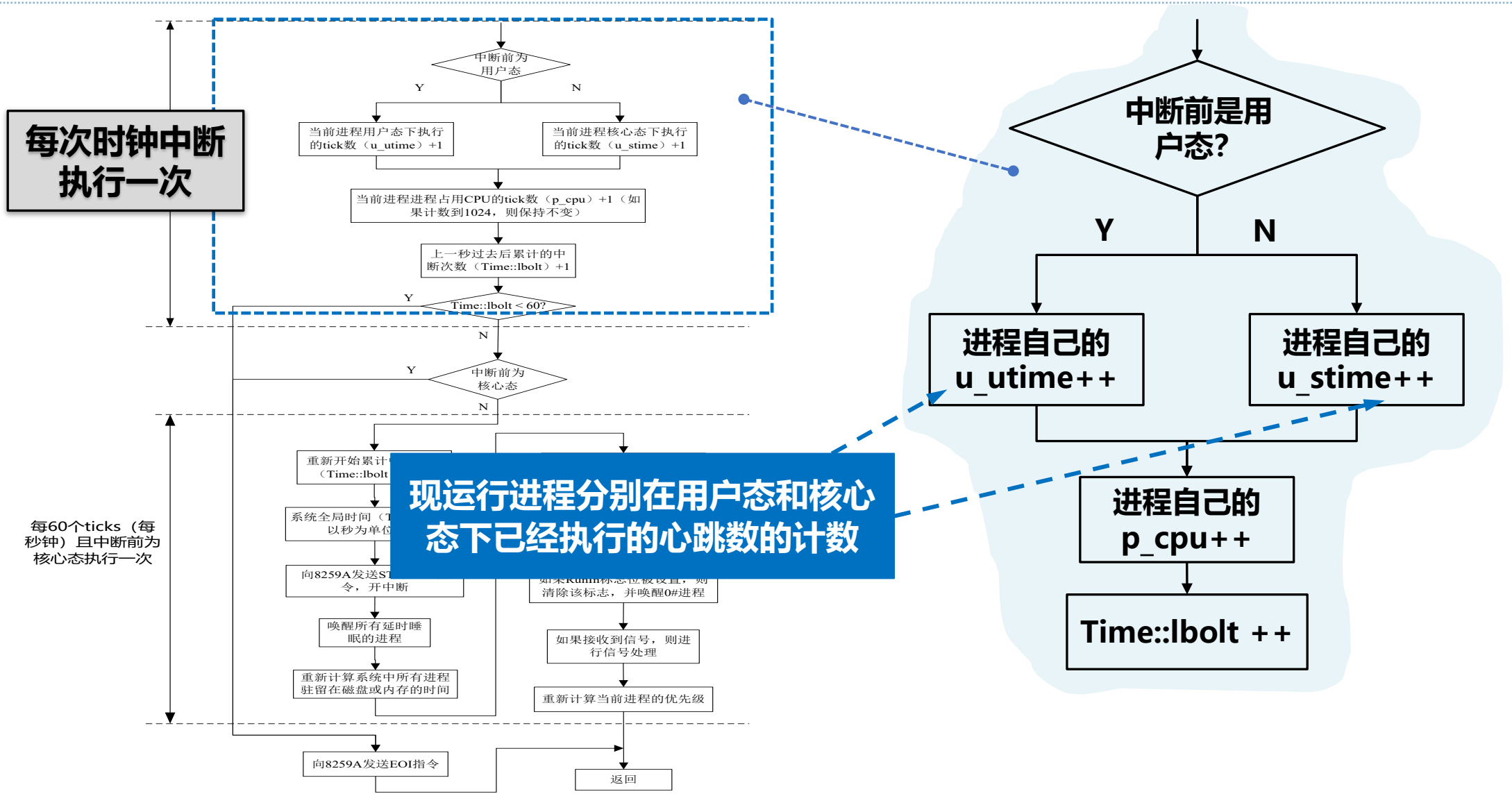
时钟中断处理程序的详细流程





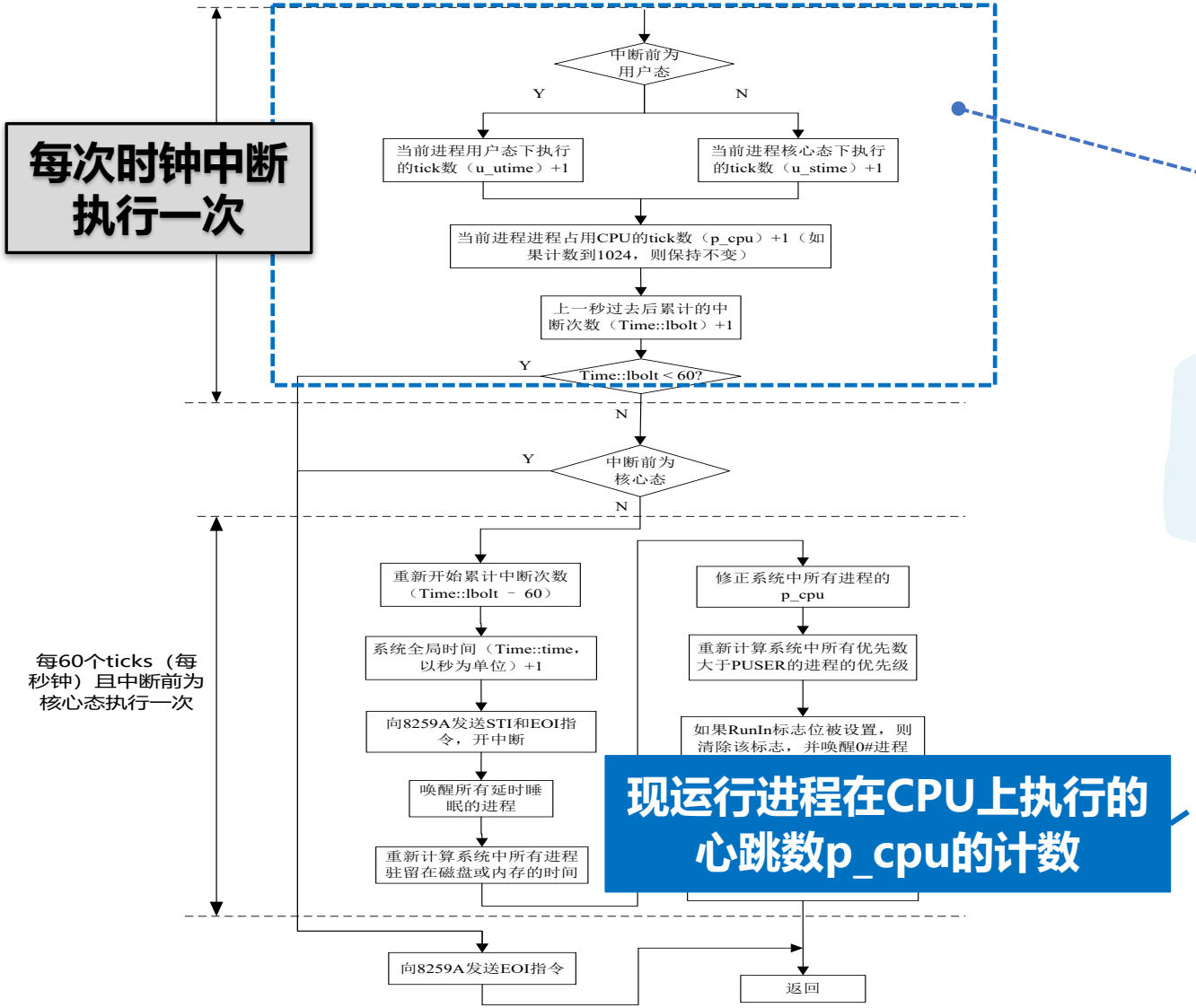
UNIX时钟中断

时钟中断处理程序的详细流程

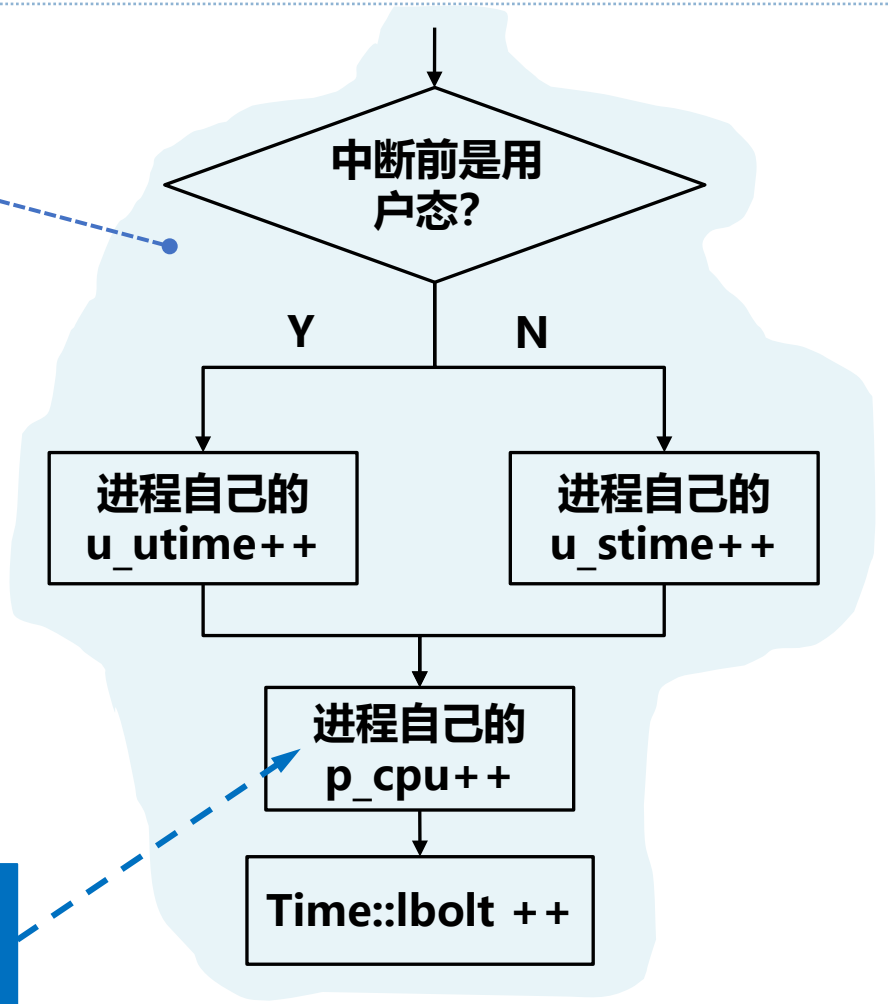


UNIX时钟中断

时钟中断处理程序的详细流程

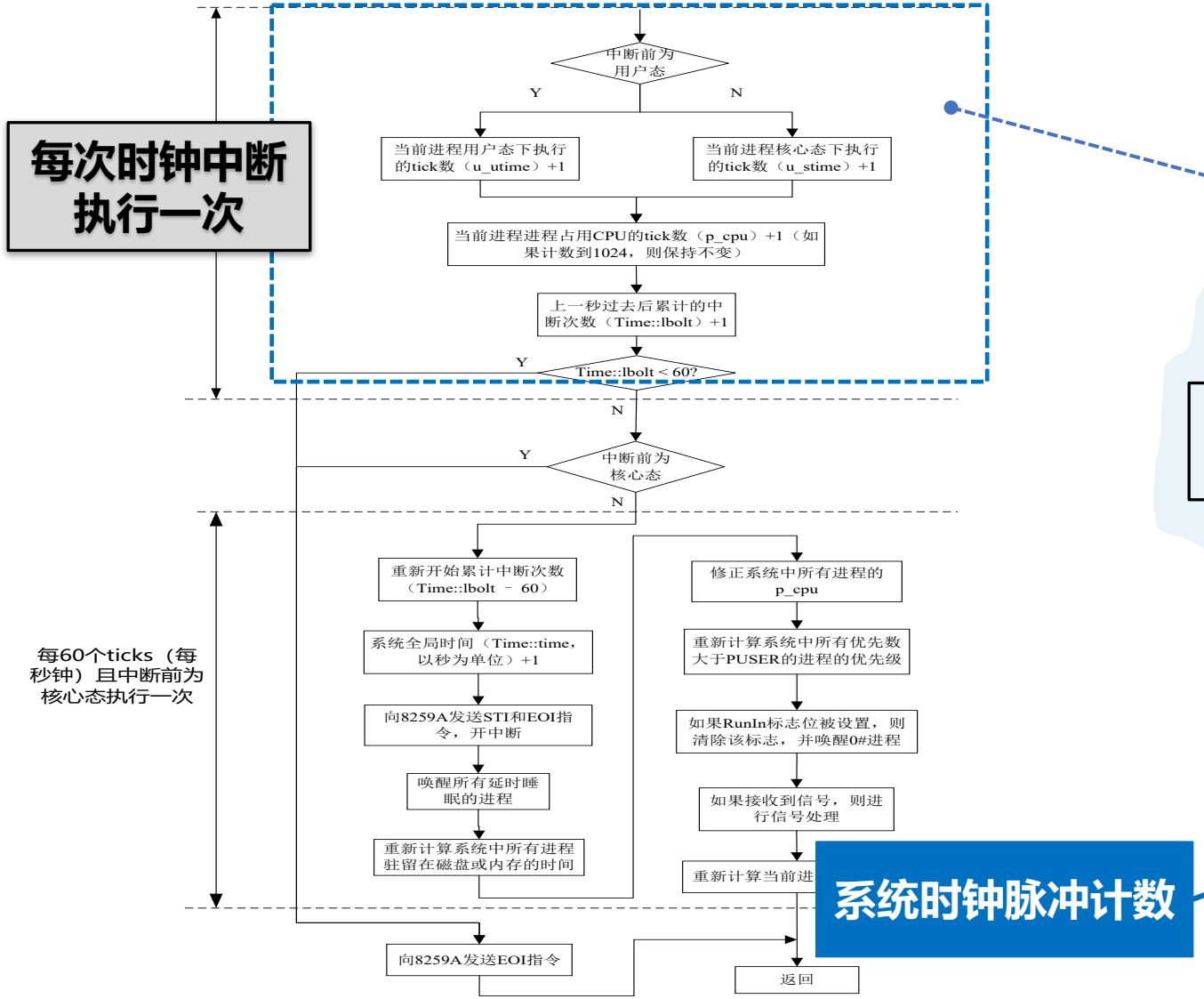


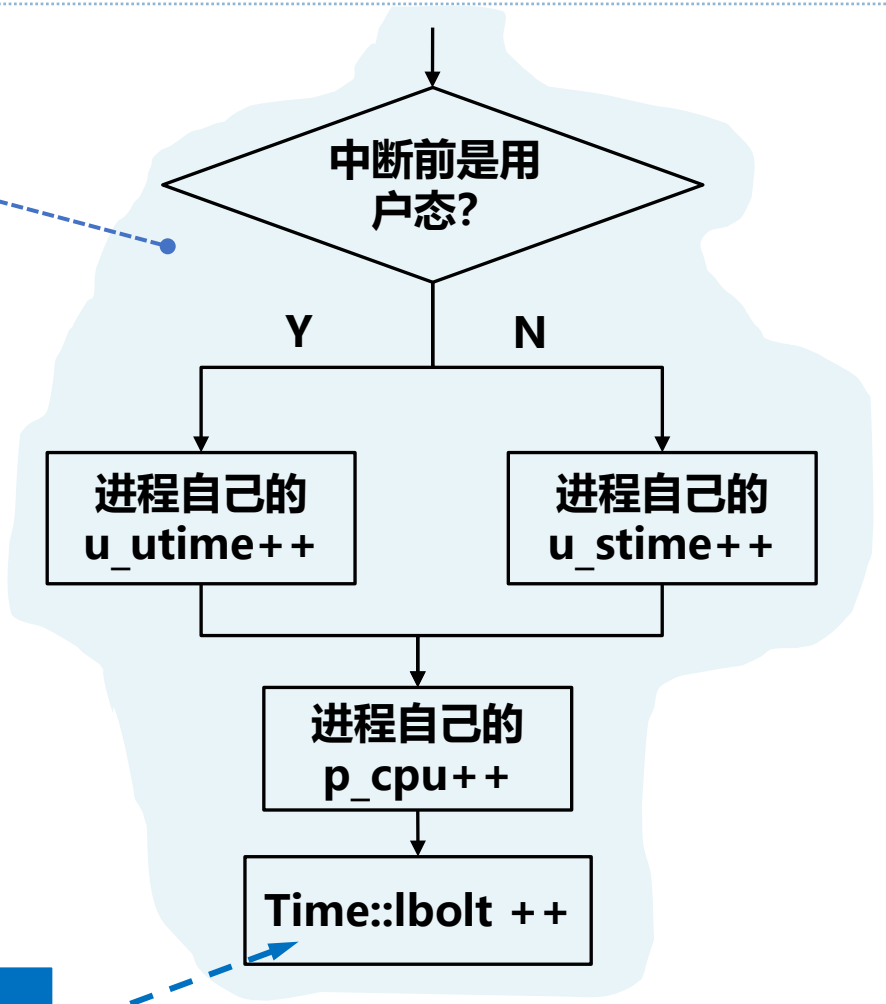
现运行进程在CPU上执行的心跳数p_cpu的计数



UNIX时钟中断

时钟中断处理程序的详细流程

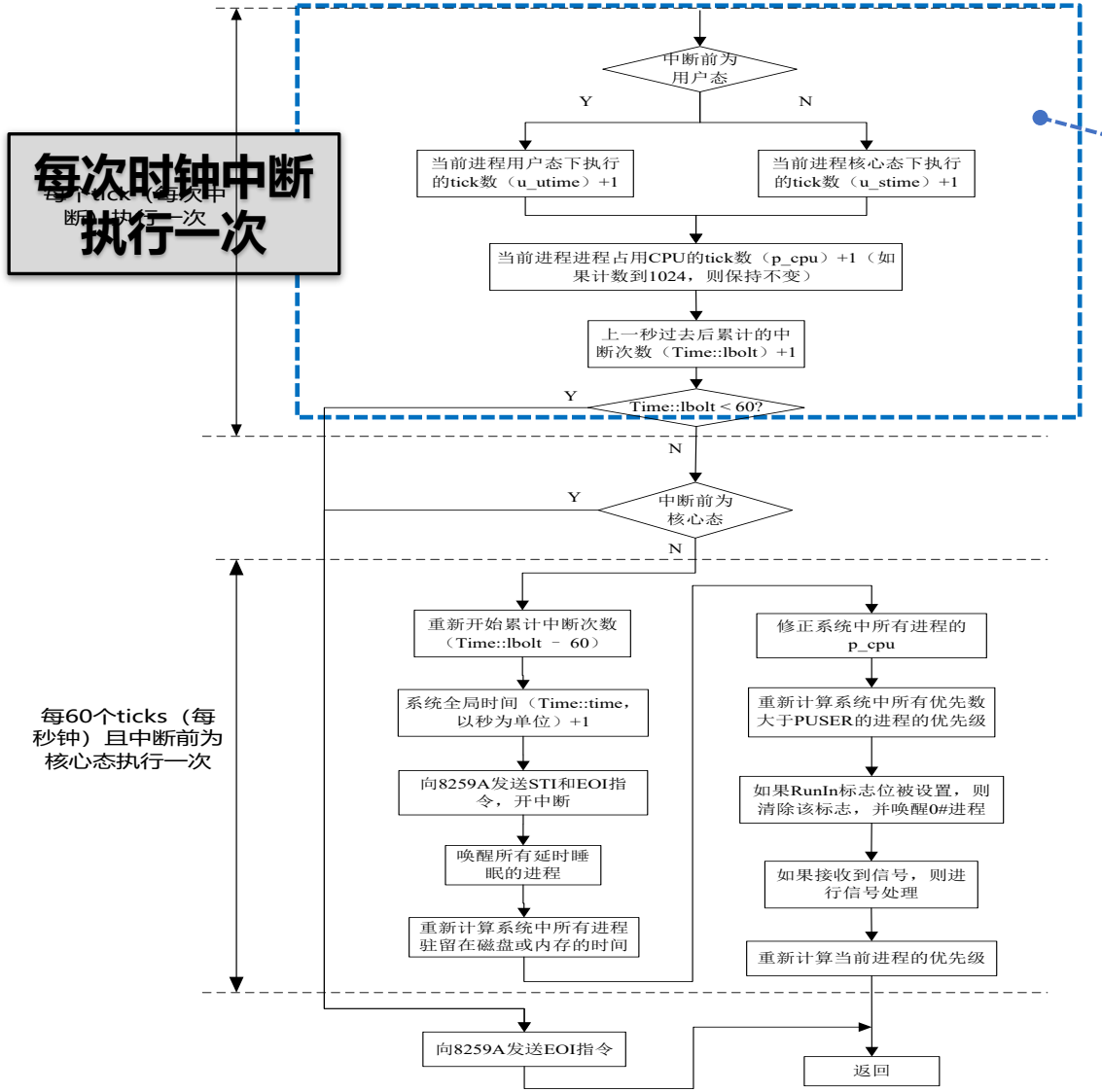


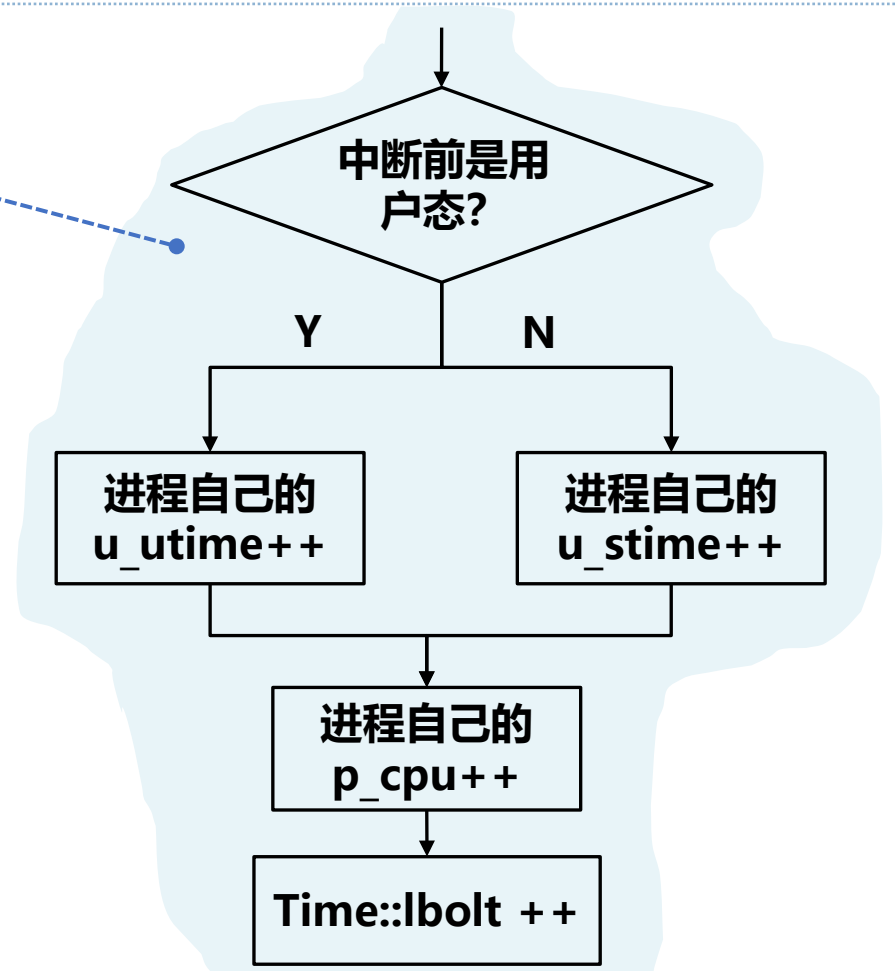


系统时钟脉冲计数

UNIX时钟中断

时钟中断处理程序的详细流程

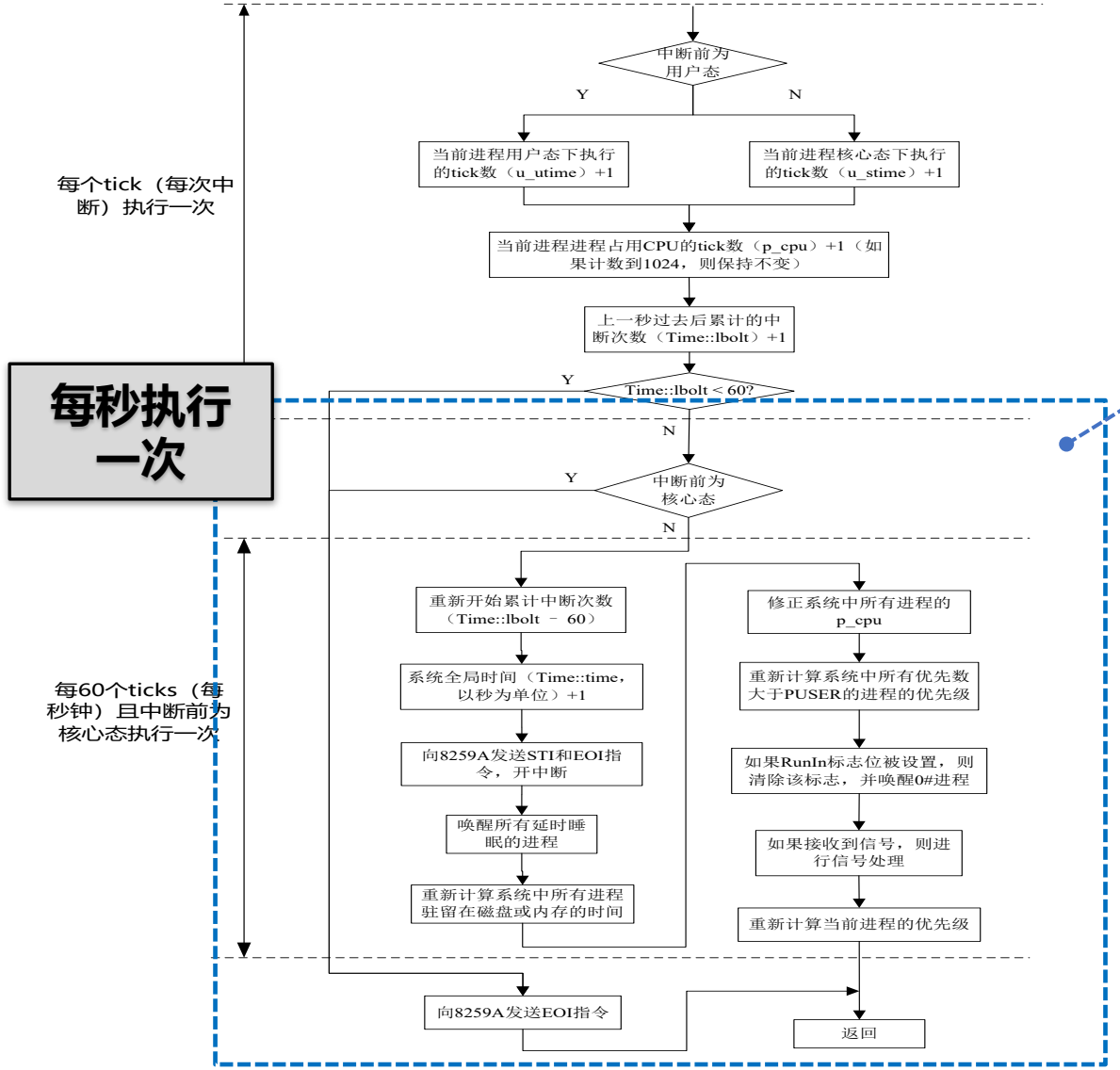


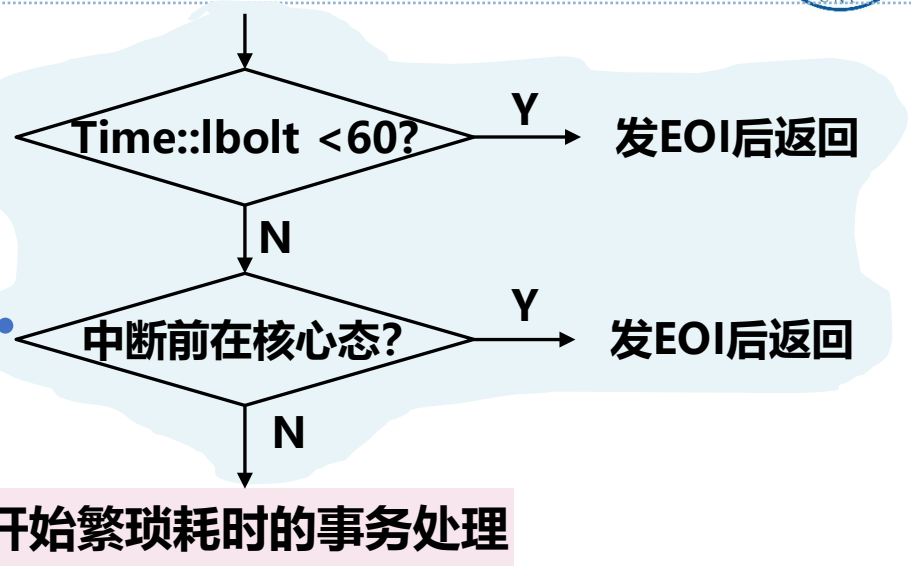


所以，这些参数都是以时钟脉冲数为单位的

UNIX时钟中断

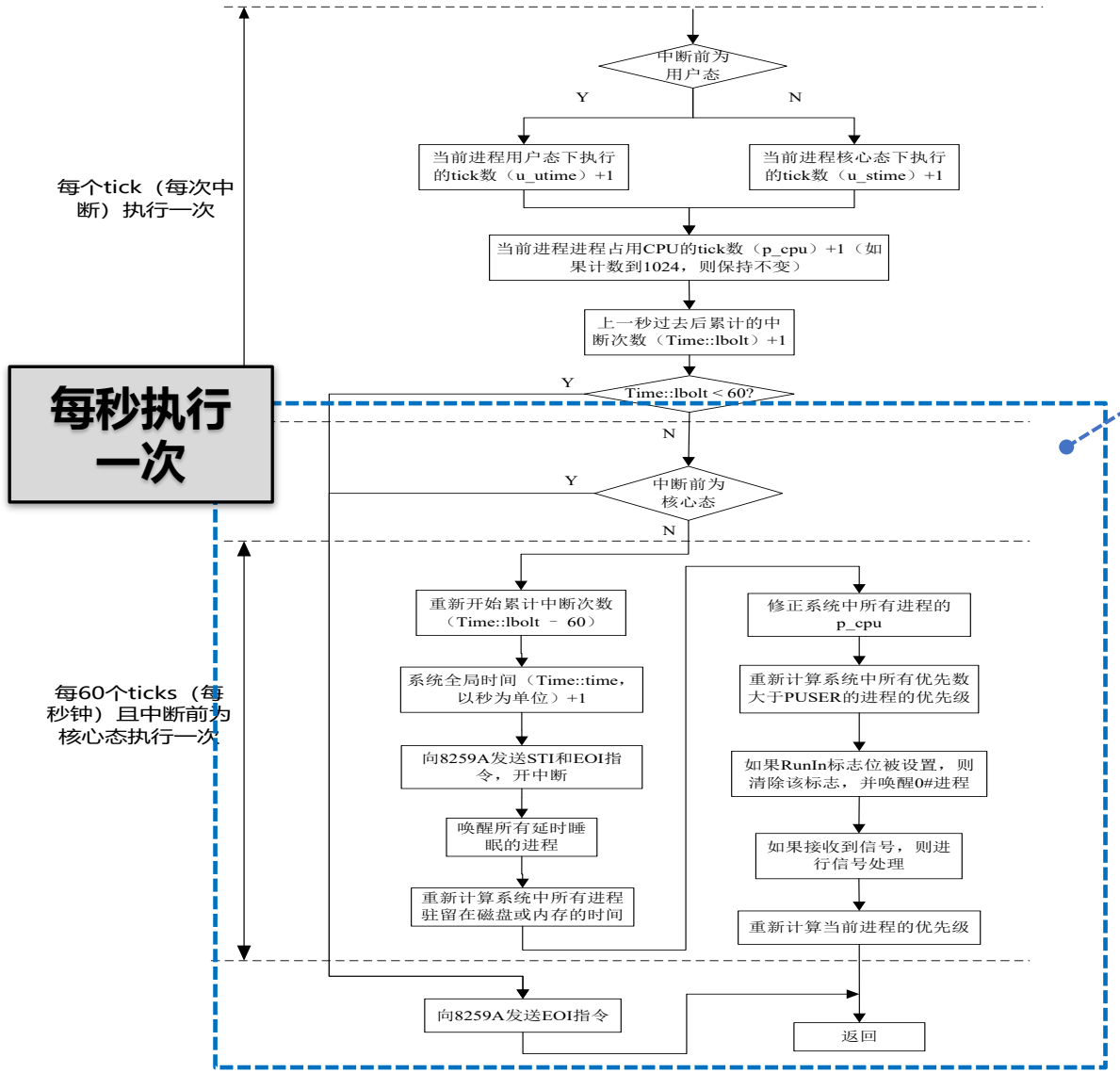
时钟中断处理程序的详细流程

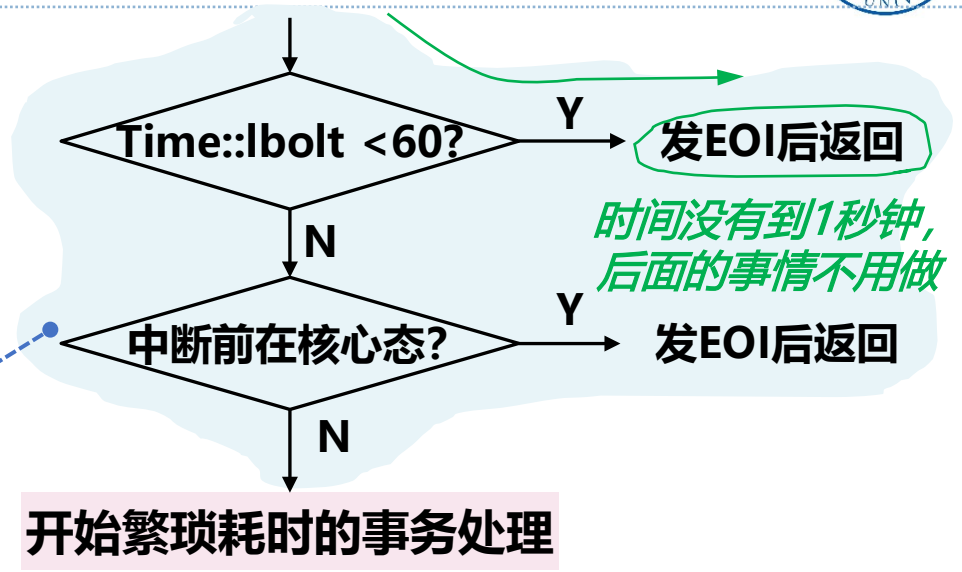




UNIX时钟中断

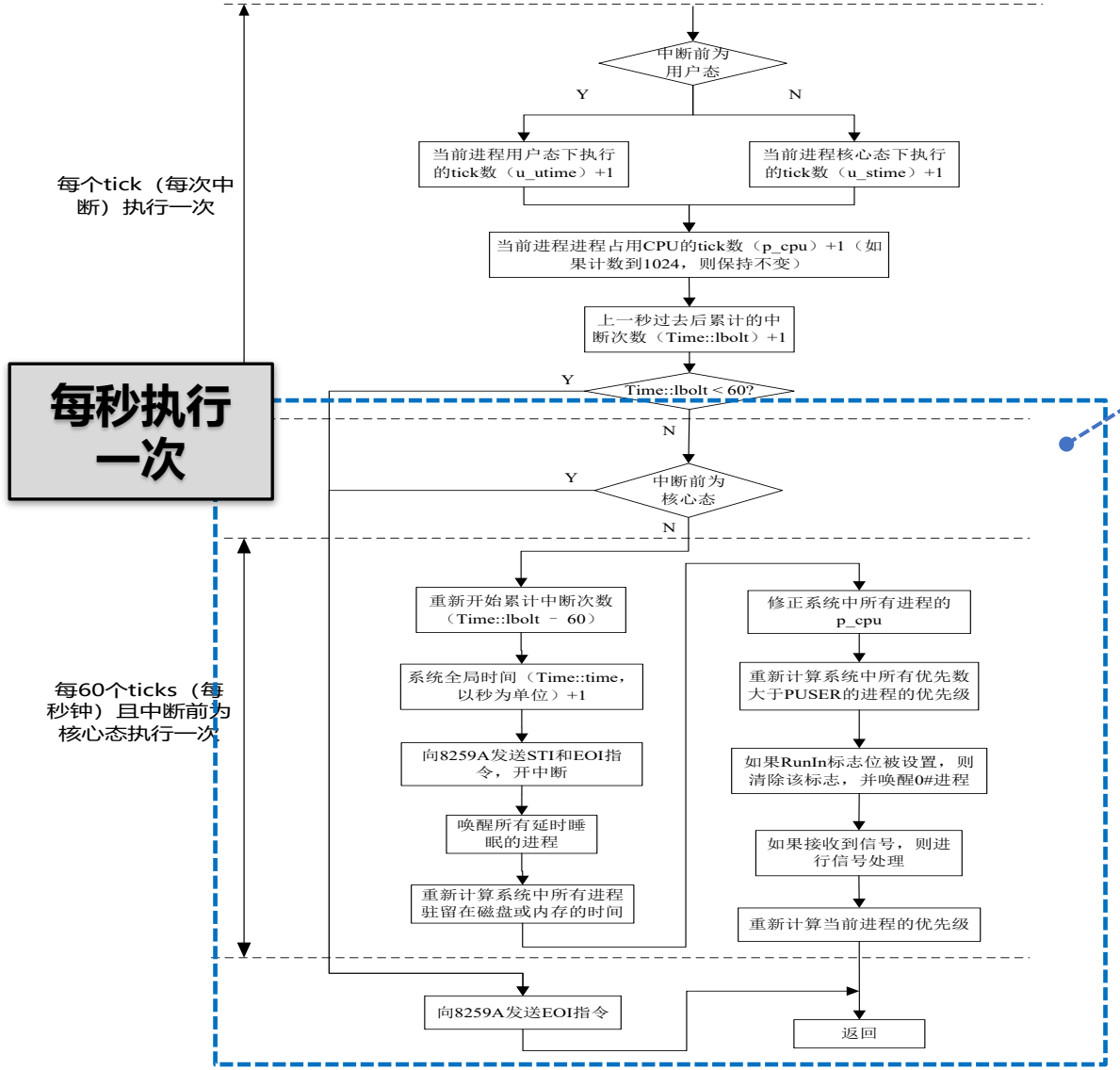
时钟中断处理程序的详细流程

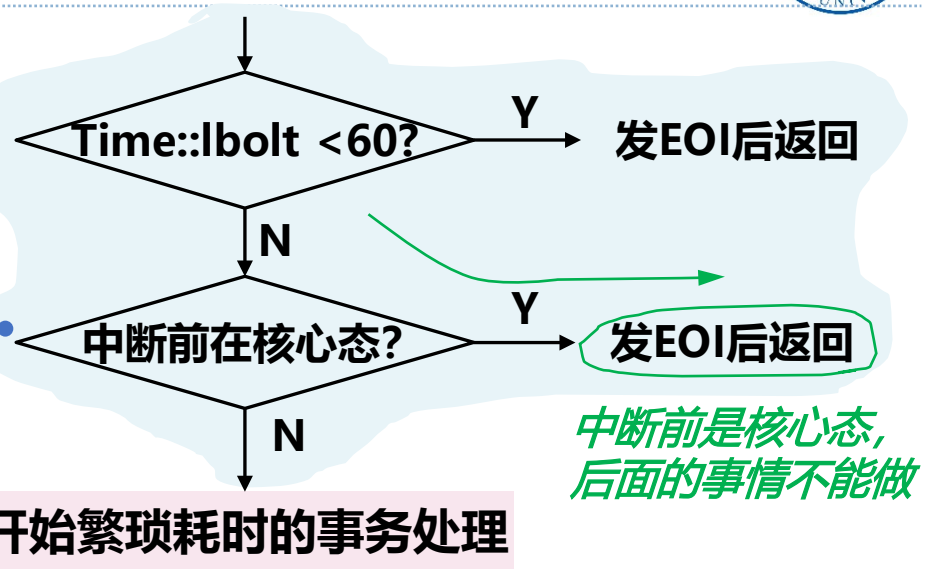




UNIX时钟中断

时钟中断处理程序的详细流程

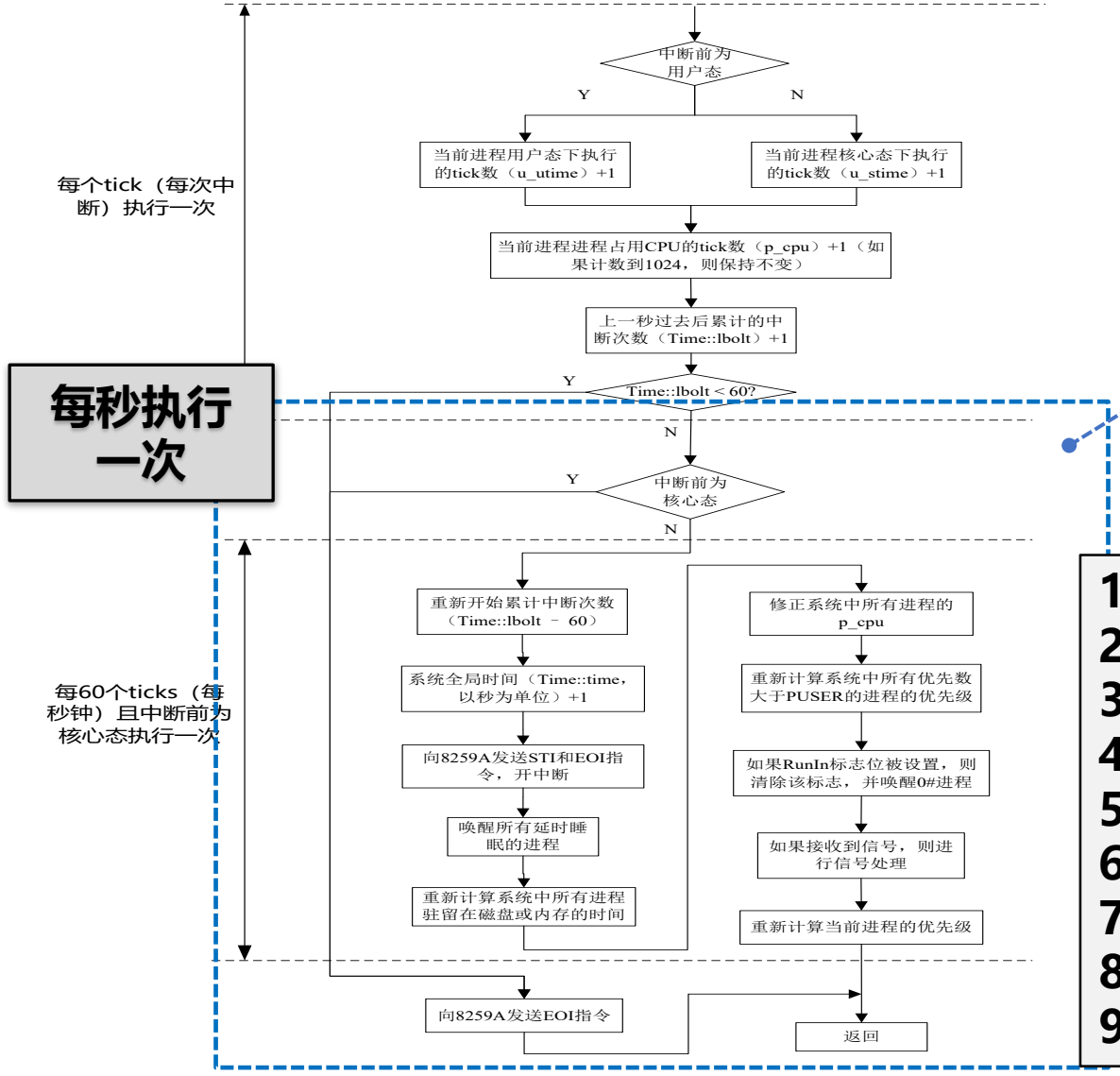


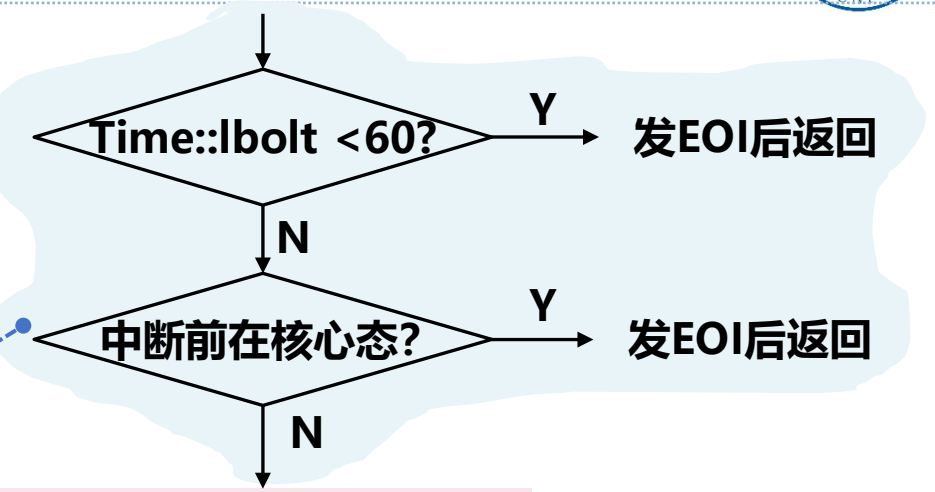


系统不会因为后续冗长的维护工作耽误核心态任务！！！

UNIX时钟中断

时钟中断处理程序的详细流程



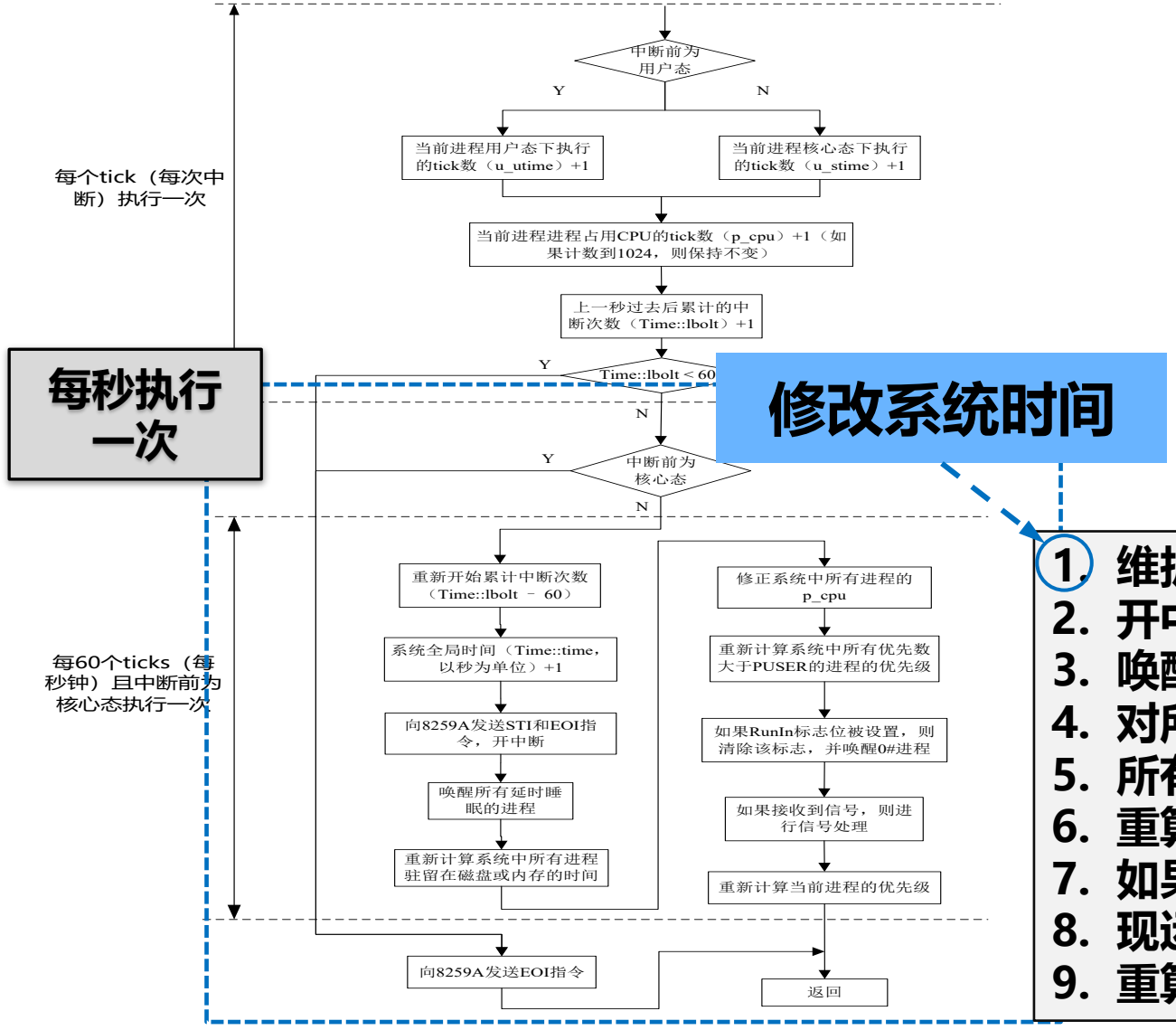


开始繁琐耗时的事务处理

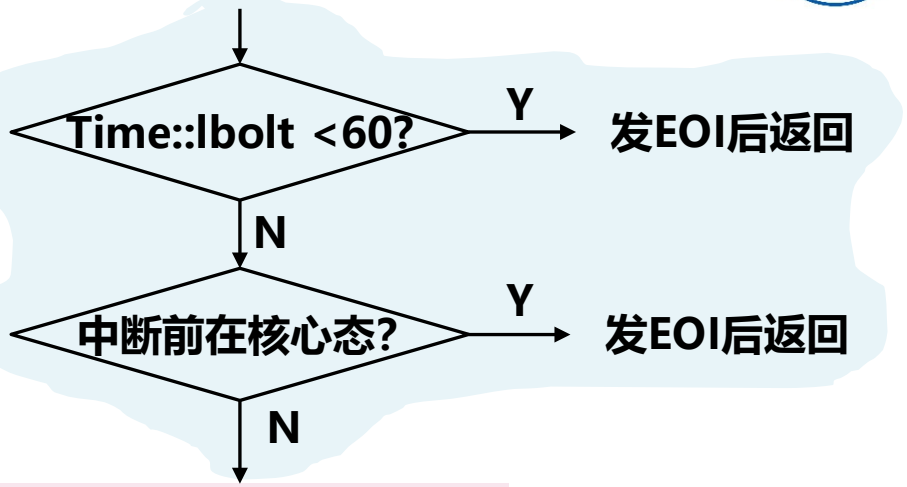
1. 维护系统时间：Time::lbolt -60; 系统时间+1秒；
2. 开中断，向中断控制器发送EOI指令；
3. 唤醒所有延时睡眠的进程；
4. 对所有进程p_time++；
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG);
6. 重算部分进程的优先数；
7. 如果RunIn被设置，唤醒0#进程；
8. 现运行进程进行信号处理；
9. 重算当前进程的优先数。

UNIX时钟中断

时钟中断处理程序的详细流程



修改系统时间

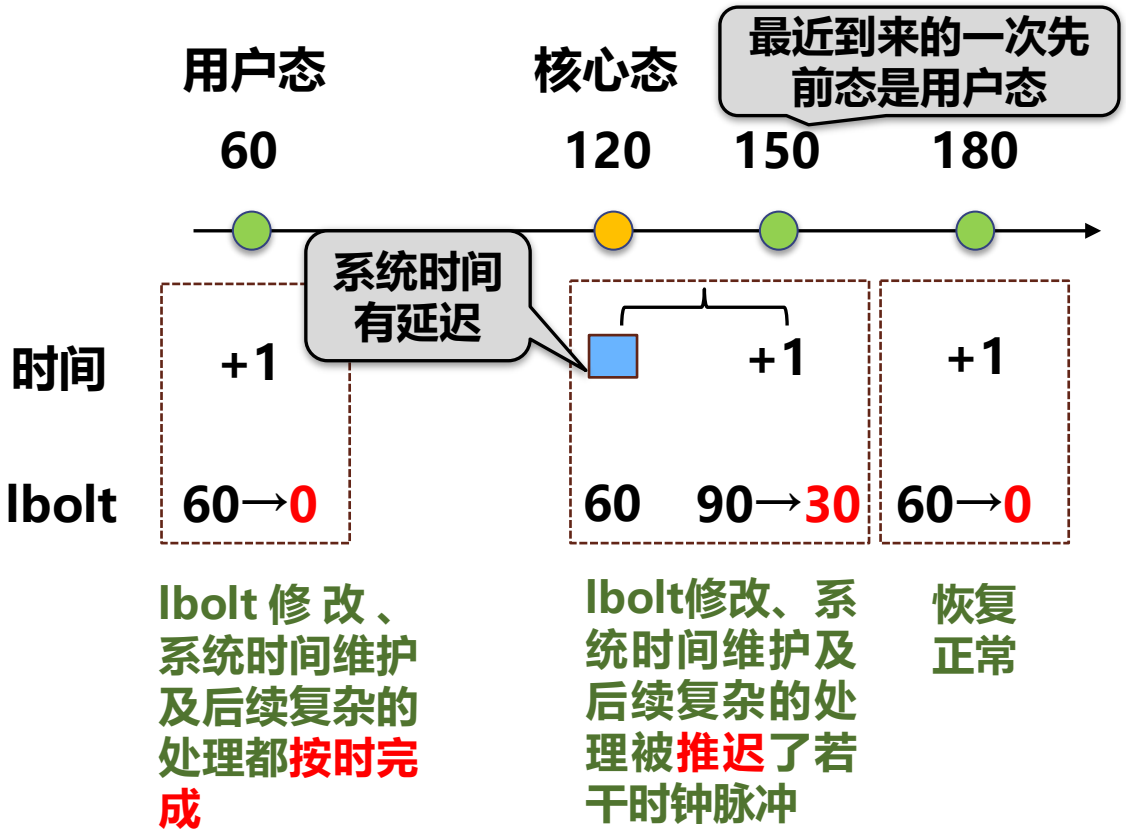


开始繁琐耗时的事务处理

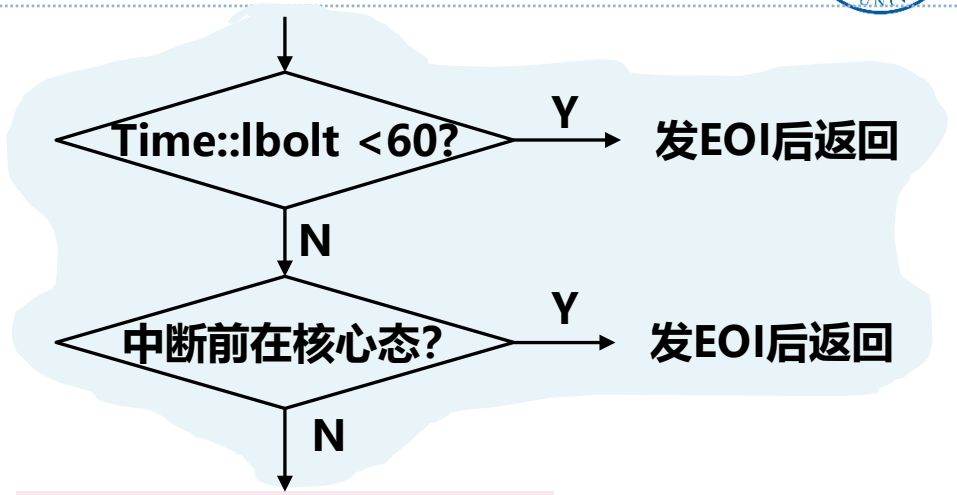
1. 维护系统时间：Time::lbolt -60; 系统时间+1秒；
2. 开中断，向中断控制器发送EOI指令；
3. 唤醒所有延时睡眠的进程；
4. 对所有进程p_time++；
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG);
6. 重算部分进程的优先数；
7. 如果RunIn被设置，唤醒0#进程；
8. 现运行进程进行信号处理；
9. 重算当前进程的优先数。

UNIX时钟中断

如果整数秒的时钟中断先前态是核心态，是否会漏掉这一秒计时?



不漏掉一次时钟脉冲，但不急于修改系统时间

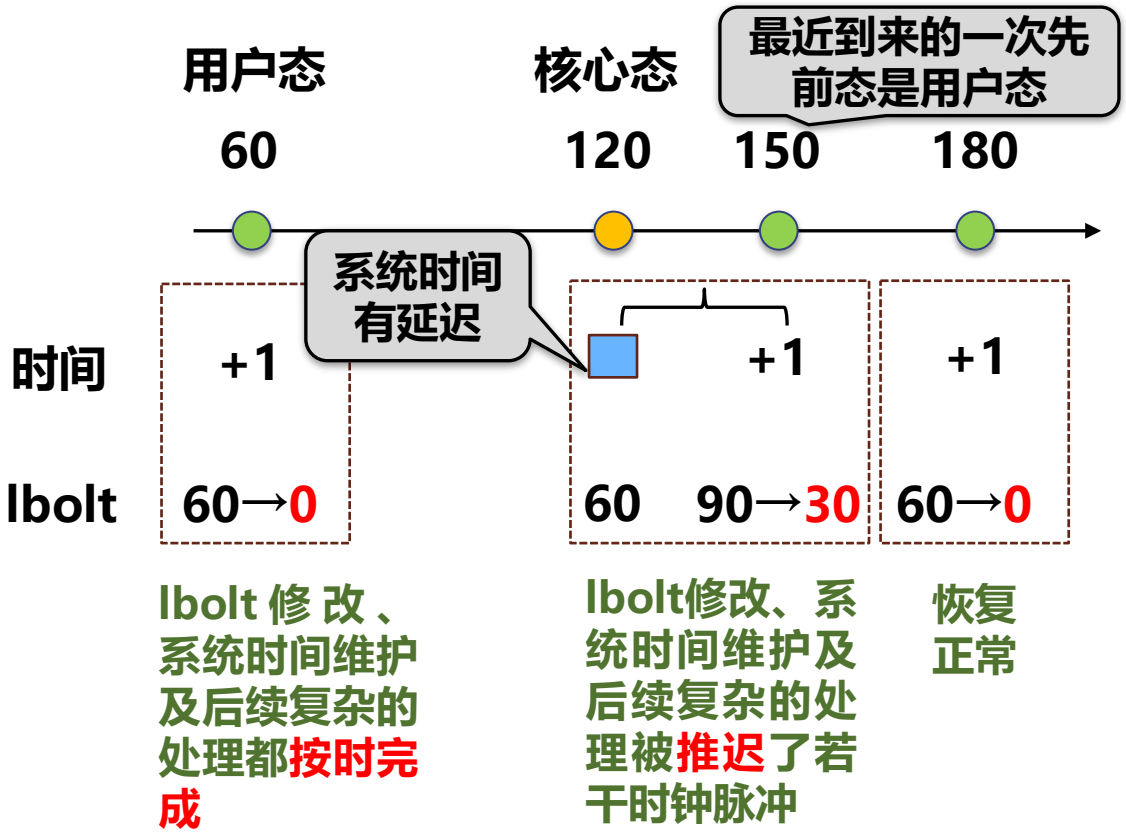


开始繁琐耗时的事务处理

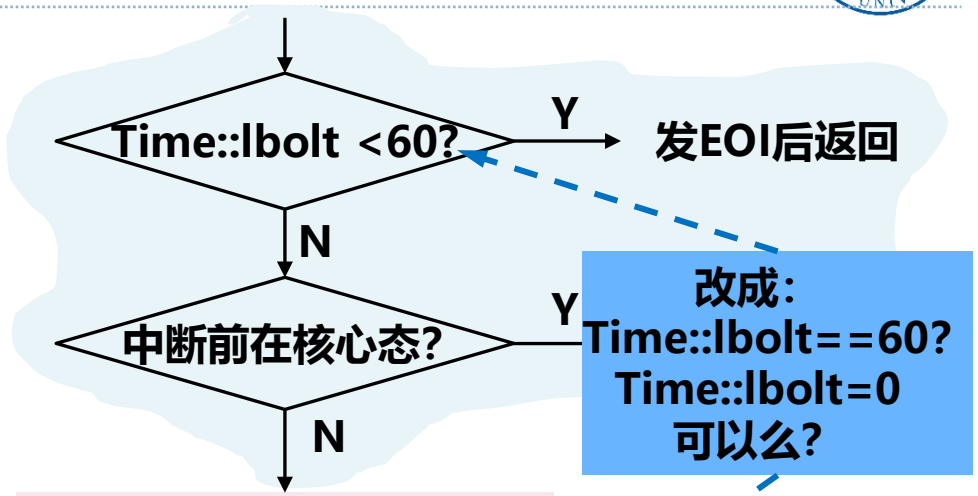
1. 维护系统时间：Time::lbolt -60; 系统时间+1秒;
2. 开中断，向中断控制器发送EOI指令;
3. 唤醒所有延时睡眠的进程;
4. 对所有进程p_time++;
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG);
6. 重算部分进程的优先数;
7. 如果RunIn被设置，唤醒0#进程;
8. 现运行进程进行信号处理;
9. 重算当前进程的优先数。

UNIX时钟中断

如果整数秒的时钟中断先前态是核心态，是否会漏掉这一秒计时?



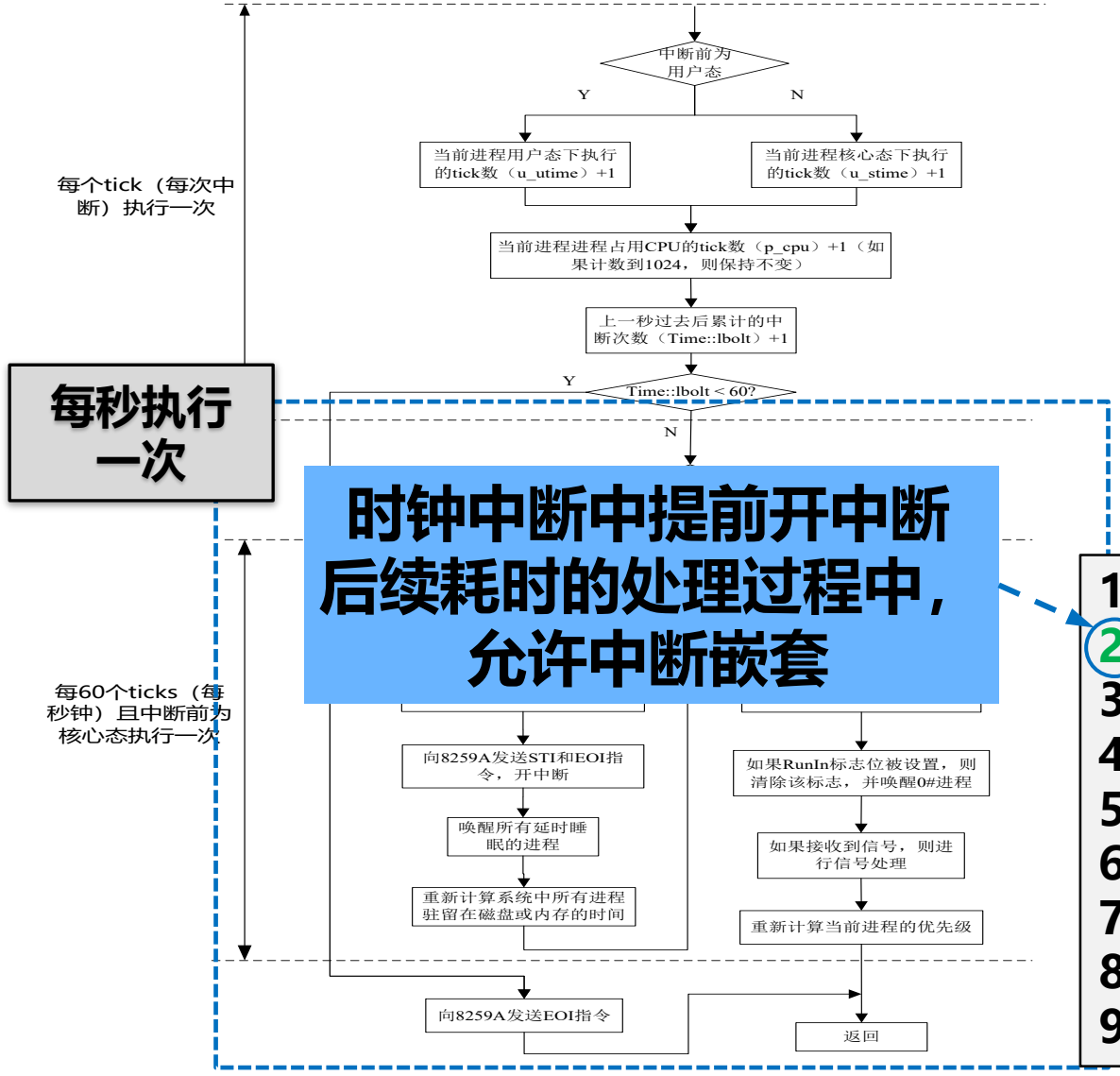
不漏掉一次时钟脉冲，但不急于修改系统时间



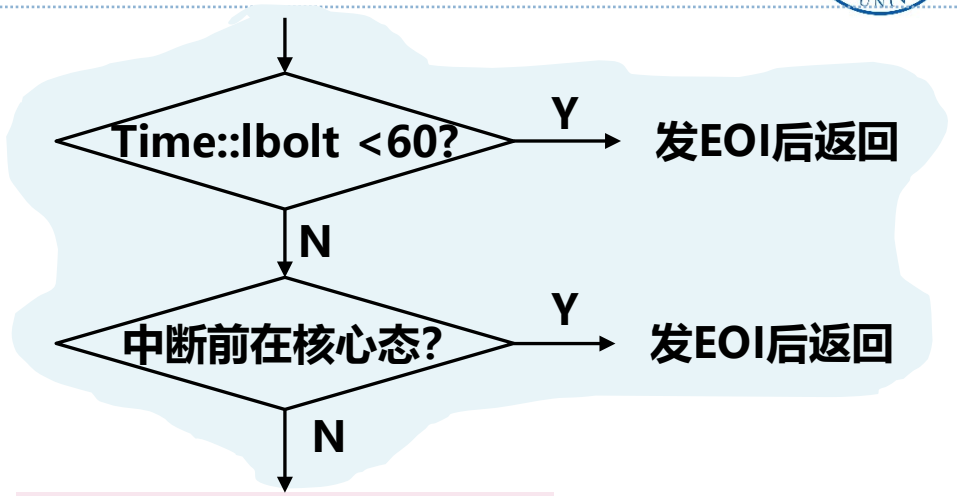
1. 维护系统时间：Time::lbolt -60; 系统时间+1秒;
2. 开中断，向中断控制器发送EOI指令;
3. 唤醒所有延时睡眠的进程;
4. 对所有进程p_time++;
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG);
6. 重算部分进程的优先数;
7. 如果RunIn被设置，唤醒0#进程;
8. 现运行进程进行信号处理;
9. 重算当前进程的优先数。

UNIX时钟中断

时钟中断处理程序的详细流程



时钟中断中提前开中断后续耗时的处理过程中，允许中断嵌套

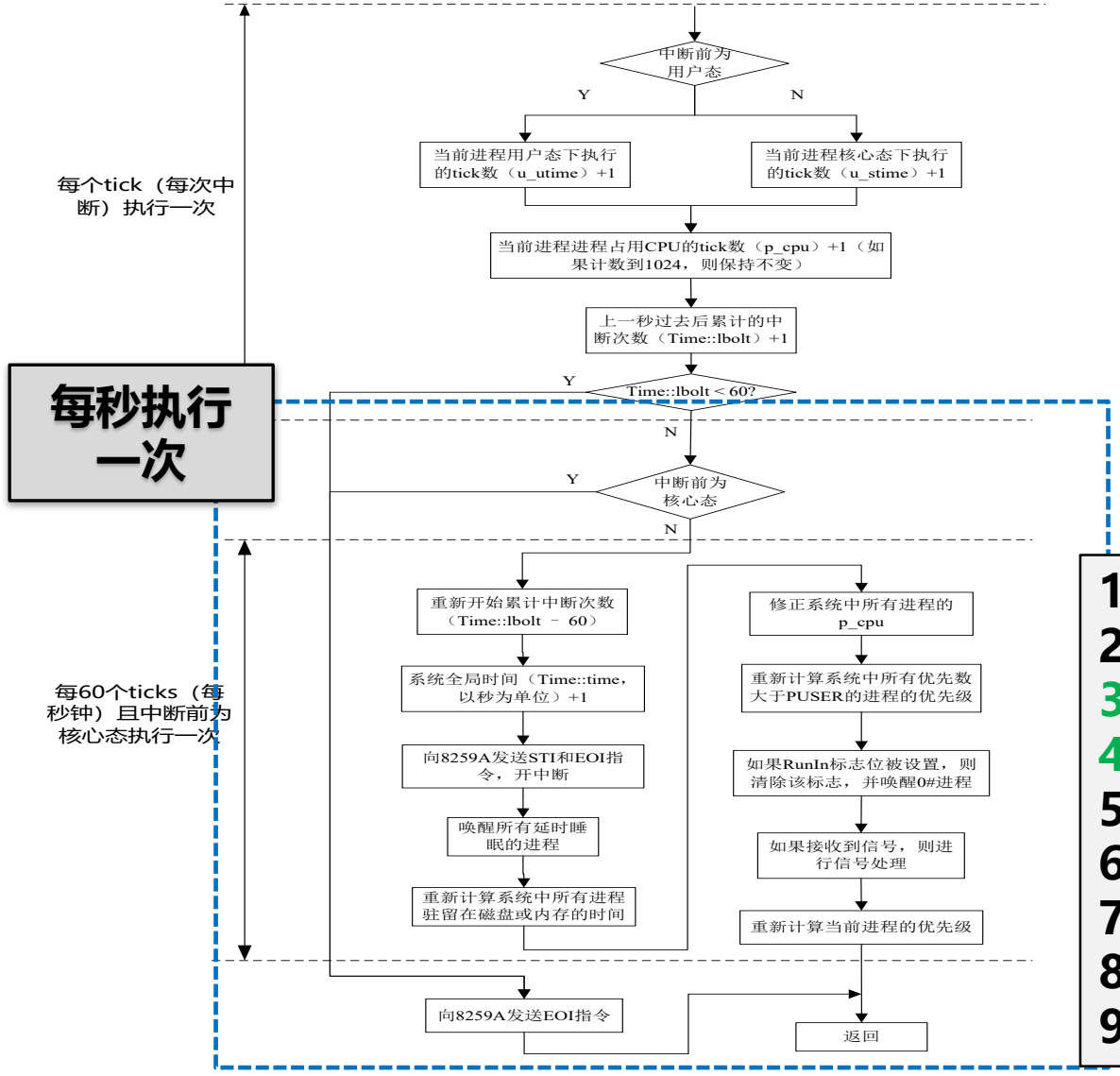


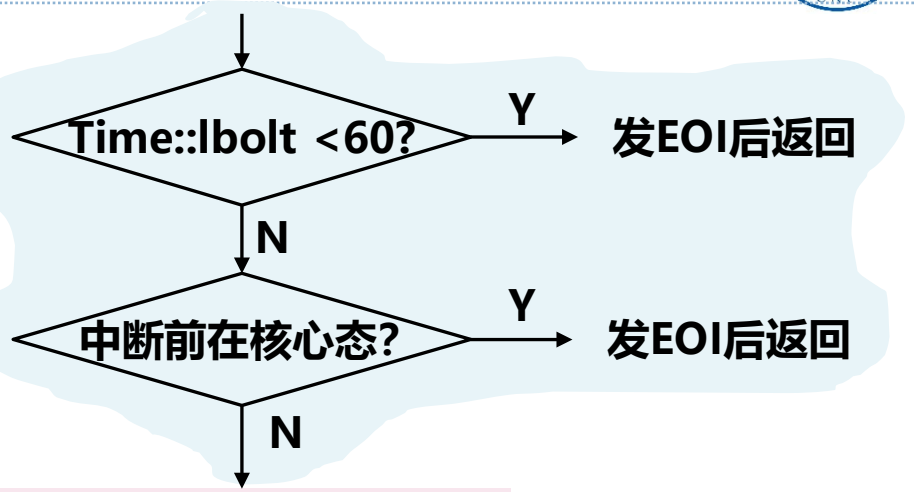
开始繁琐耗时的事务处理

1. 维护系统时间：Time::lbolt -60; 系统时间+1秒；
2. 开中断，向中断控制器发送EOI指令；
3. 唤醒所有延时睡眠的进程；
4. 对所有进程p_time++;
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG);
6. 重算部分进程的优先数；
7. 如果RunIn被设置，唤醒0#进程；
8. 现运行进程进行信号处理；
9. 重算当前进程的优先数。

UNIX时钟中断

时钟中断处理程序的详细流程





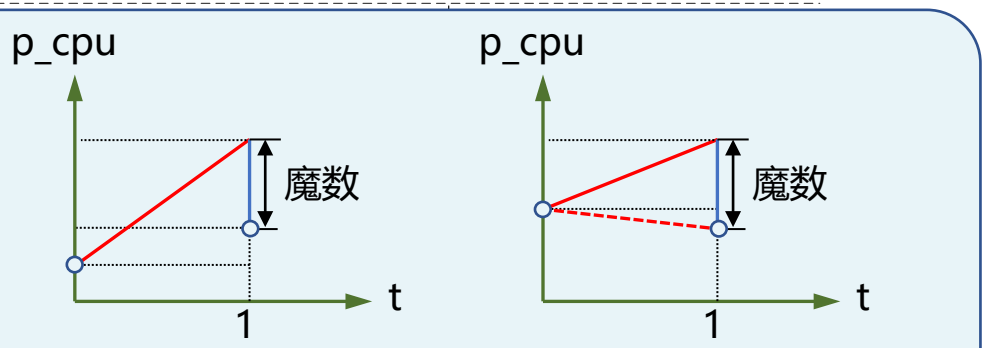
开始繁琐耗时的事务处理

1. 维护系统时间：Time::lbolt -60; 系统时间+1秒；
2. 开中断，向中断控制器发送EOI指令；
3. 唤醒所有延时睡眠的进程；
4. 对所有进程p_time++；
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG);
6. 重算部分进程的优先数；
7. 如果RunIn被设置，唤醒0#进程；
8. 现运行进程进行信号处理；
9. 重算当前进程的优先数。

UNIX时钟中断

时钟中断处理程序的详细流程

每个tick（每次时钟中断）执行一次

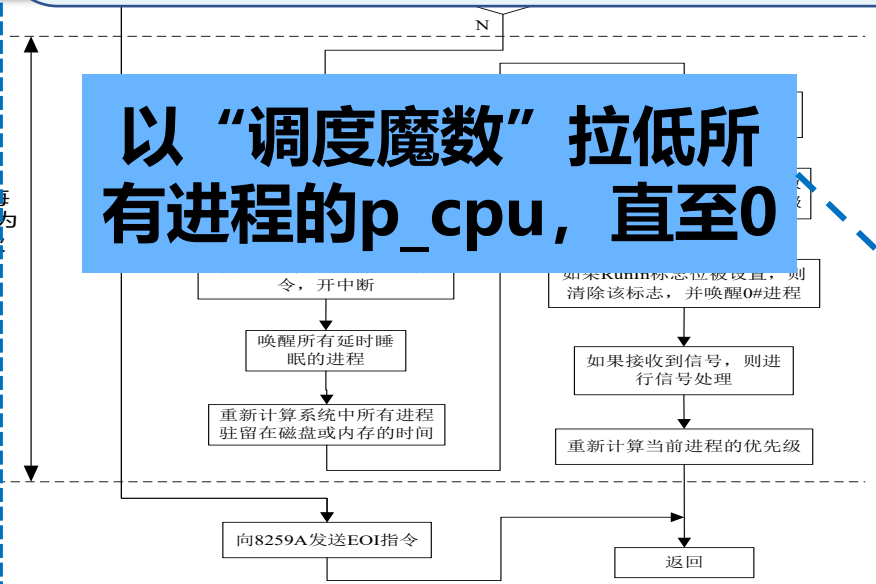


如果：1秒内p_cpu增加的量>调度魔数
则：1秒到时p_cpu增加

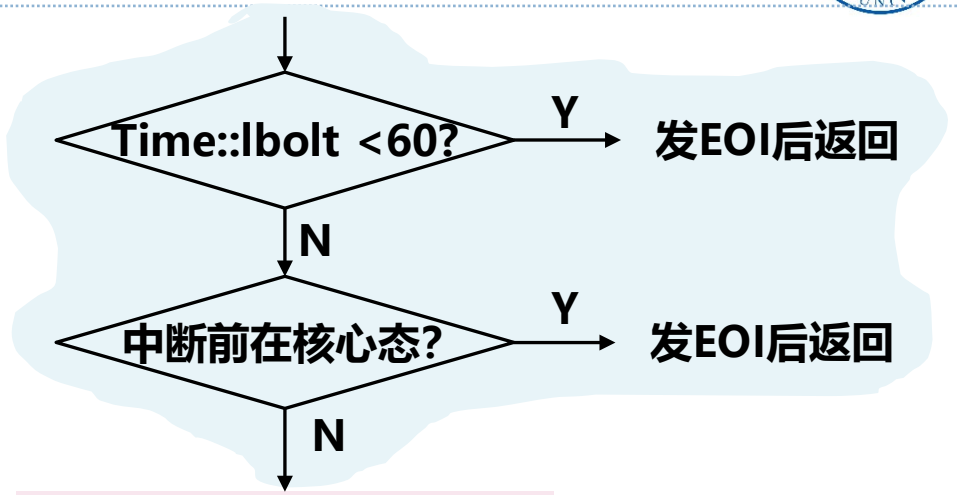
如果：1秒内p_cpu增加的量<调度魔数
则：1秒到时p_cpu减小

每秒执行一次

每60个ticks（每秒钟）且中断前为核心态执行一次



以“调度魔数”拉低所有进程的p_cpu，直至0

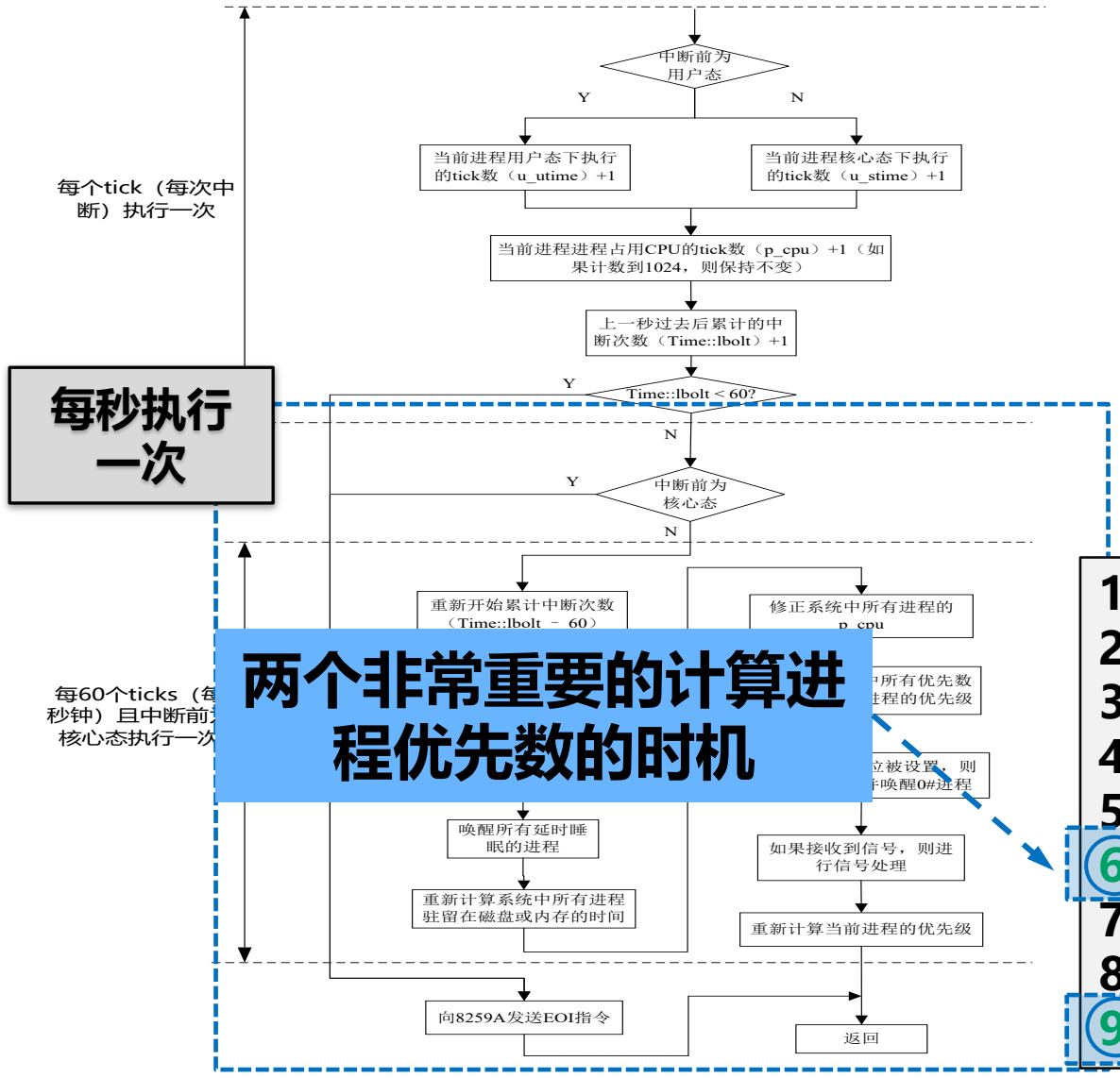


开始繁琐耗时的事务处理

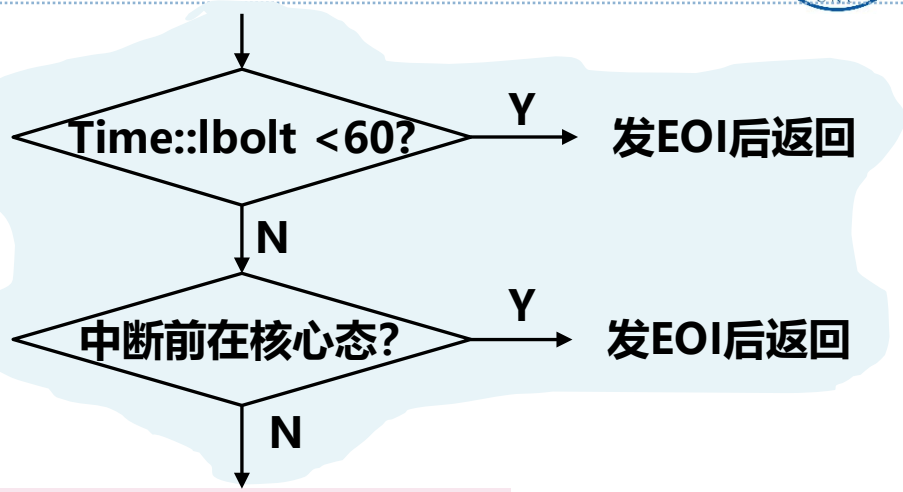
1. 维护系统时间：Time::lbolt -60; 系统时间+1秒；
2. 开中断，向中断控制器发送EOI指令；
3. 唤醒所有延时睡眠的进程；
4. 对所有进程p_time++；
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG)；
6. 重算部分进程的优先数；
7. 如果RunIn被设置，唤醒0#进程；
8. 现运行进程进行信号处理；
9. 重算当前进程的优先数。

UNIX时钟中断

时钟中断处理程序的详细流程



两个非常重要的计算进程优先数的时机



开始繁琐耗时的事务处理

1. 维护系统时间：Time::lbolt -60; 系统时间+1秒；
2. 开中断，向中断控制器发送EOI指令；
3. 唤醒所有延时睡眠的进程；
4. 对所有进程p_time++；
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG);
6. 重算部分进程的优先数； 1
7. 如果RunIn被设置，唤醒0#进程；
8. 现运行进程进行信号处理；
9. 重算当前进程的优先数。 2

UNIX时钟中断

进程优先数的计算方法

值越小，进程的优先级越高

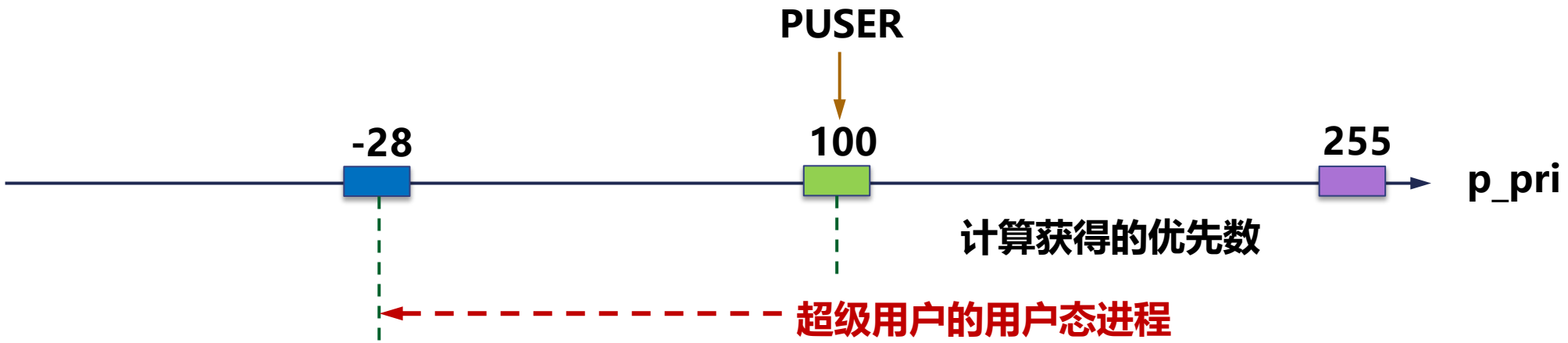
$$p\_pri = \min\{ 255, (p\_cpu / 16 + PUSER + p\_nice) \}$$

PUSER：固定偏置常数

static const int PUSER = 100;

p_nice： 相对优先程度

允许用户通过系统调用设置
超级用户：-128 ~ 20
普通用户：0 ~20

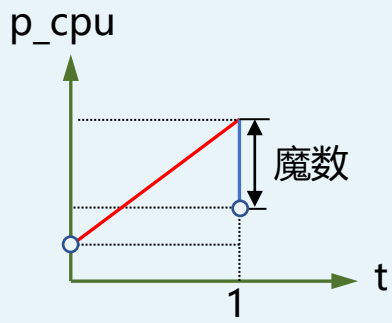


UNIX时钟中断

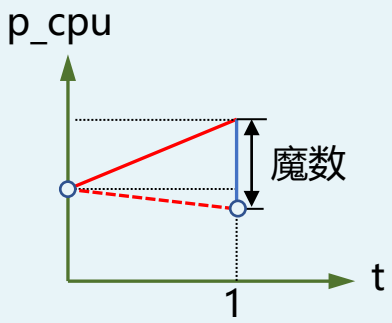
进程优先数的计算方法

值越小，进程的优先级越高

$$p\_pri = \min\{ 255, (p\_cpu / 16 + PUSER + p\_nice) \}$$



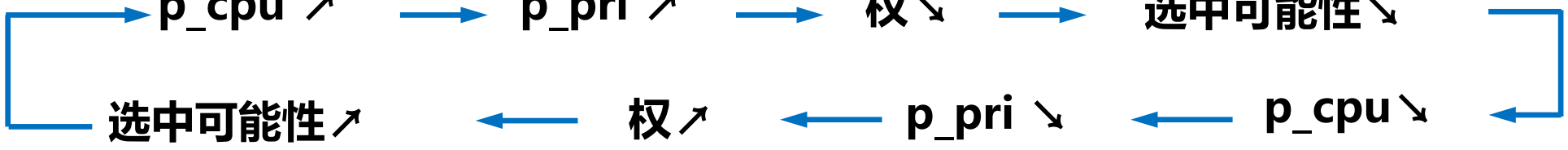
如果：1秒内p_cpu增加的量>调度魔数
则：1秒到时p_cpu增加



如果：1秒内p_cpu增加的量<调度魔数
则：1秒到时p_cpu减小

1. 连续占用处理机较长时间的进程：
   p_cpu ↗，优先数↗，优先权↘
2. 较长时间内未使用处理机的进程：
   p_cpu ↘，优先数↘，优先权↗；
3. 虽频繁使用处理机，但每次时间很短的进程：
   p_cpu ↘，优先数↘，优先权↗。

防止高者恒高，低者“饥饿”

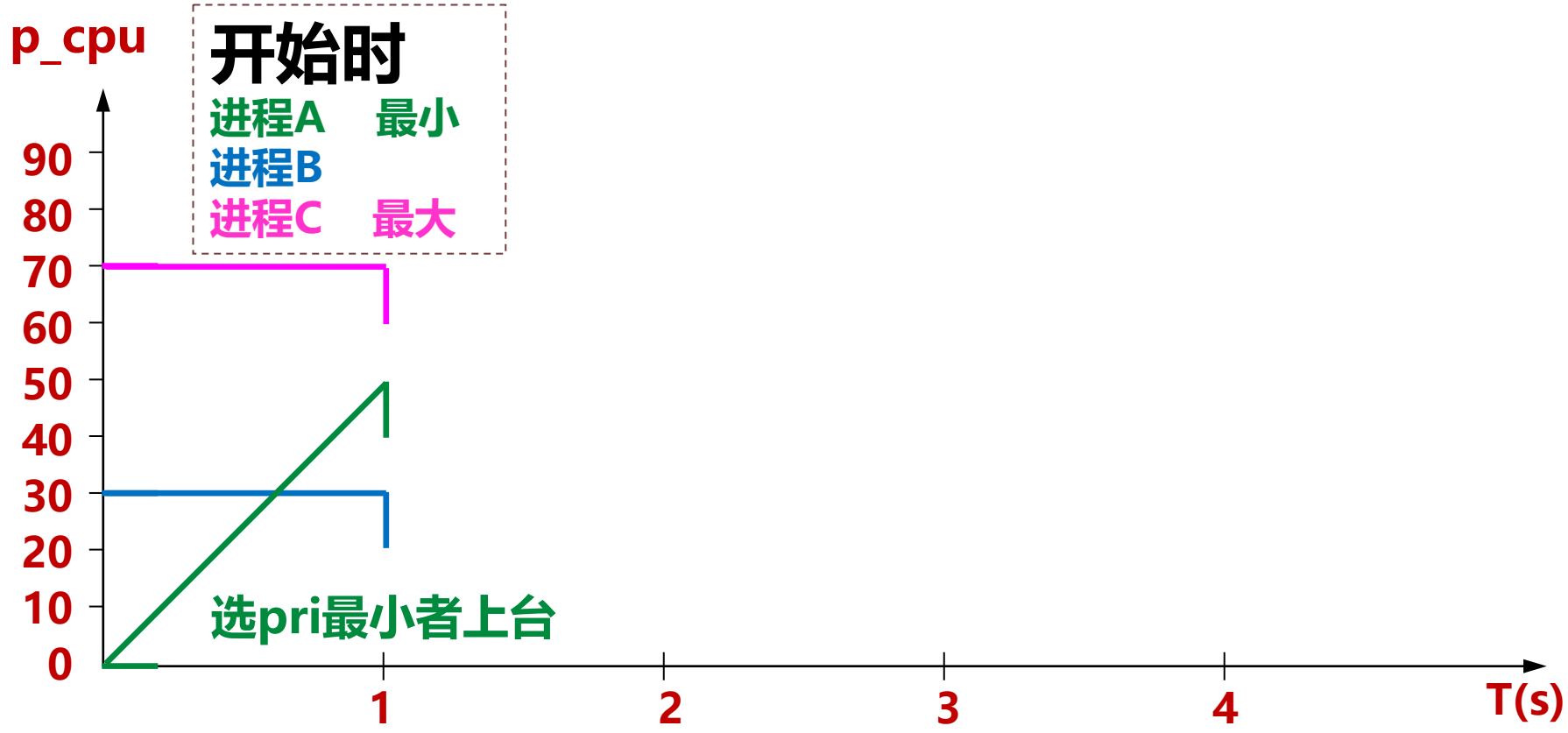


UNIX时钟中断

进程优先数的计算方法

值越小，进程的优先级越高

$$p\_pri = \min\{ 255, (p\_cpu / 16 + PUSER + p\_nice) \}$$

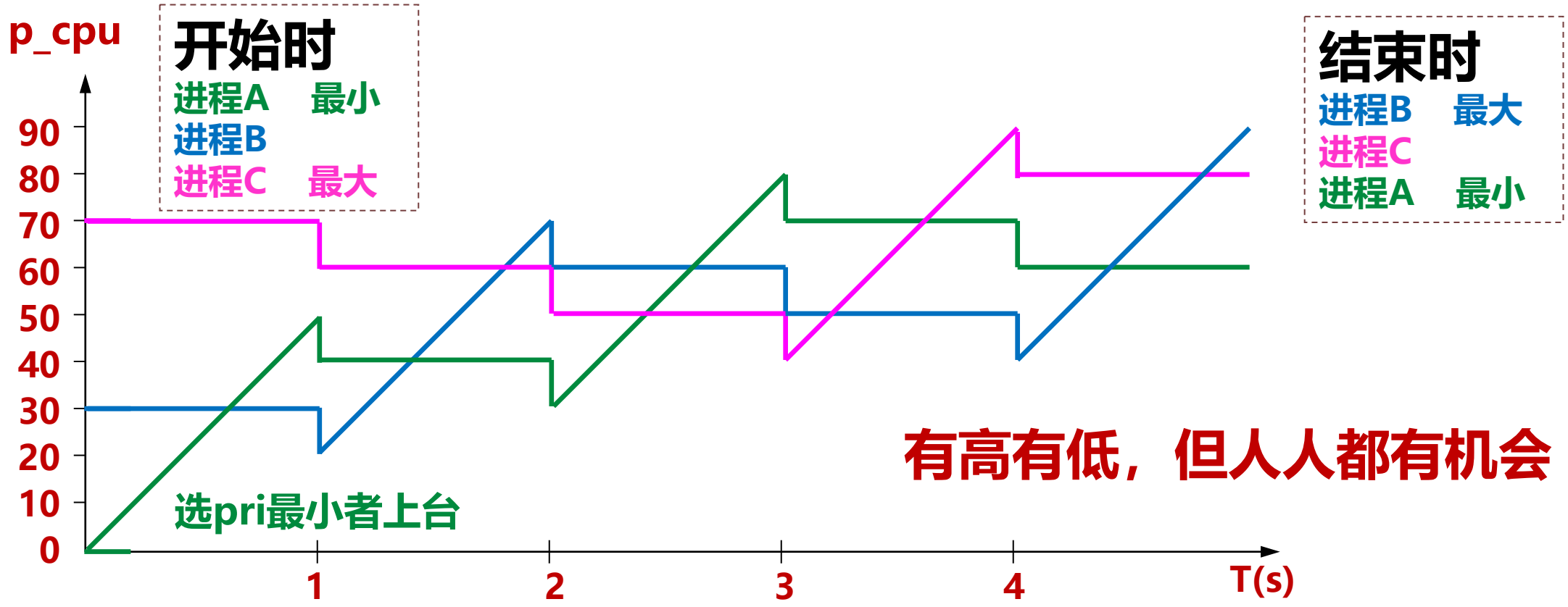


UNIX时钟中断

进程优先数的计算方法

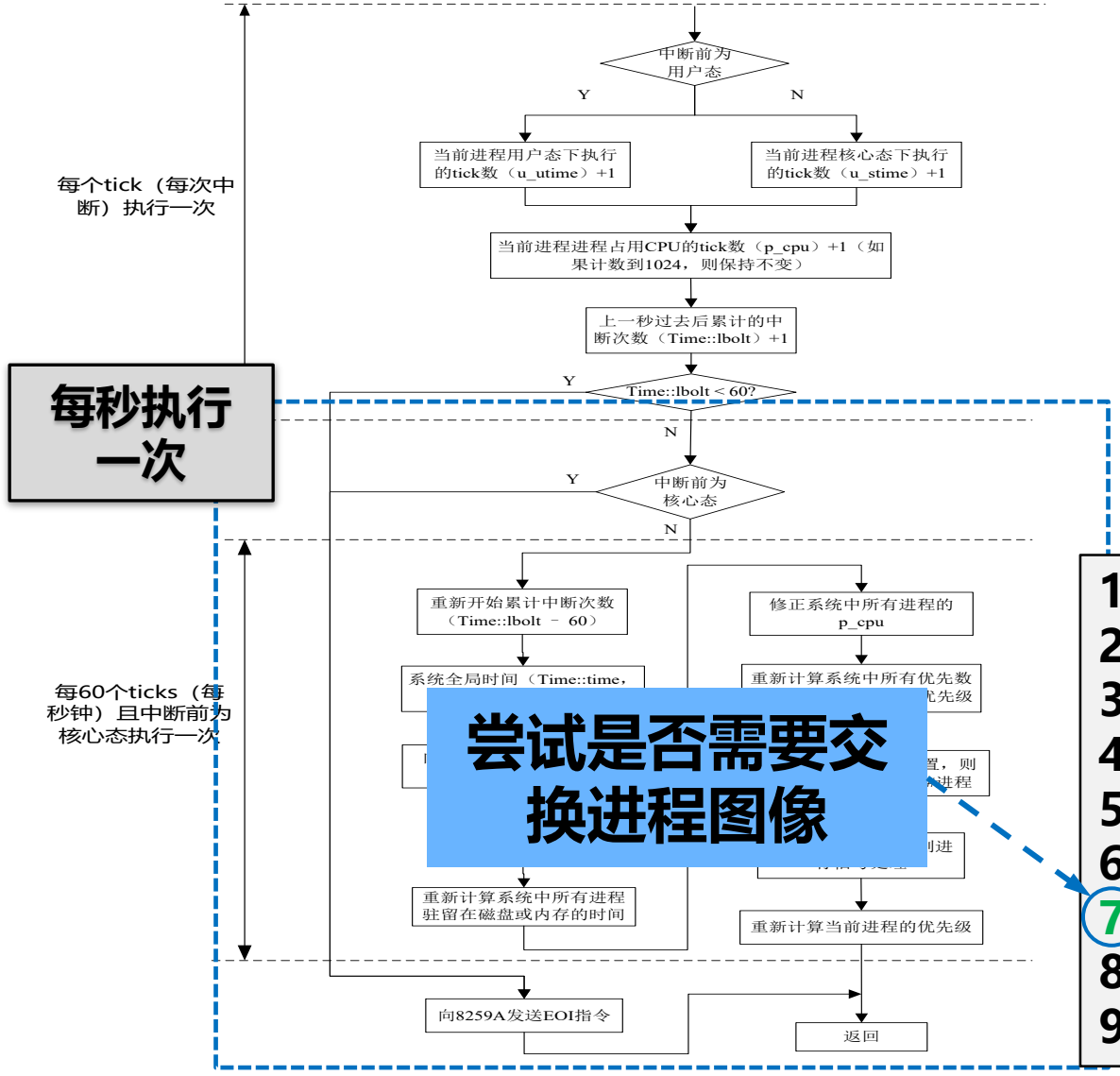
值越小，进程的优先级越高

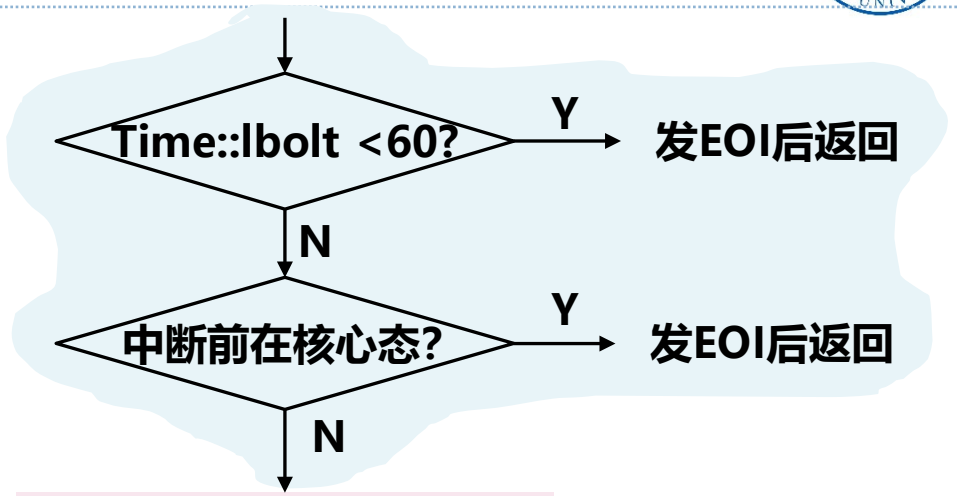
$$p\_pri = \min\{ 255, (p\_cpu / 16 + PUSER + p\_nice) \}$$



UNIX时钟中断

时钟中断处理程序的详细流程





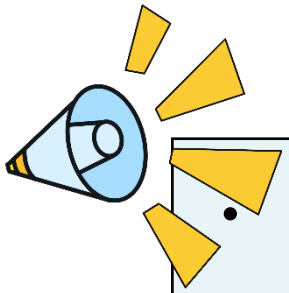
开始繁琐耗时的事务处理

1. 维护系统时间：Time::lbolt -60; 系统时间+1秒；
2. 开中断，向中断控制器发送EOI指令；
3. 唤醒所有延时睡眠的进程；
4. 对所有进程p_time++；
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG);
6. 重算部分进程的优先数；
7. 如果RunIn被设置，唤醒0#进程；
8. 现运行进程进行信号处理；
9. 重算当前进程的优先数。

总结一下时钟中断的主要工作

时钟中断以每秒60次的频率定时自动发生

和进程是否发起IO操作无关



- 对p_cpu的处理
- 进程优先数的计算
- 两次计算优先数

每次心跳一次

- u_utime, u_stime的计数
- p_cpu
- lbolt的计数

简单，迅速完成

每1秒钟一次

- 维护系统时间
- 唤醒所有延时睡眠的进程
- 修改所有进程的p_time
- 调整所有进程的p_cpu
- 重算部分进程的优先级
- 可能唤醒0#进程
- 重算当前进程的优先级

繁琐，耗时

- 先前态是用户态才做
- 提前开中断，EOI

主 要 内 容

4.1 UNIX时钟中断与异常

4.2 UNIX系统调用

4.3 UNIX的进程调度状态

4.4 UNIX进程控制

UNIX异常处理

现代计算机系统中，中断的概念被扩展……

中断源

CPU外

1. I/O完成或出错引起的设备中断

中断

CPU停下正在执行的代码，转去执行一段特定的内核代码

CPU内

2. 断点跟踪
3. 硬件故障（奇、偶校验错、电源）
4. 程序性故障（浮点、除0、地址溢出）

异常/陷入

UNIX异常处理

用户态
User Mode

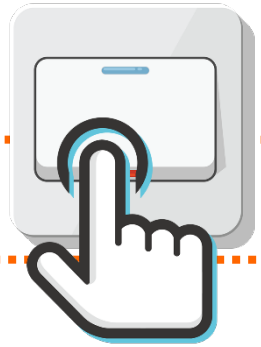
执行用户程序，
提供用户功能

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。

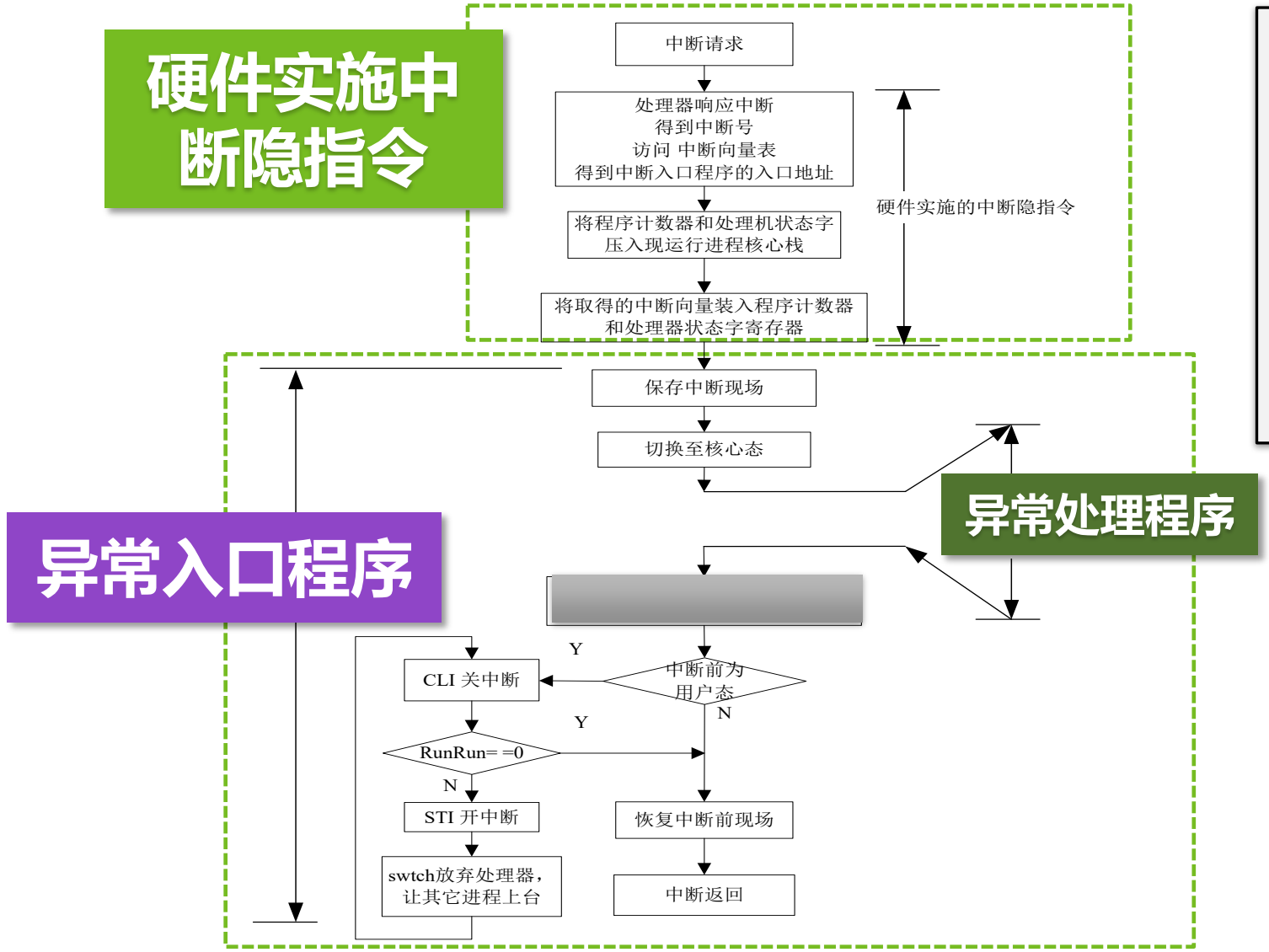


中断、异常

在核心态下执行内核代码的进程完成了内核功能！

UNIX异常处理

概况

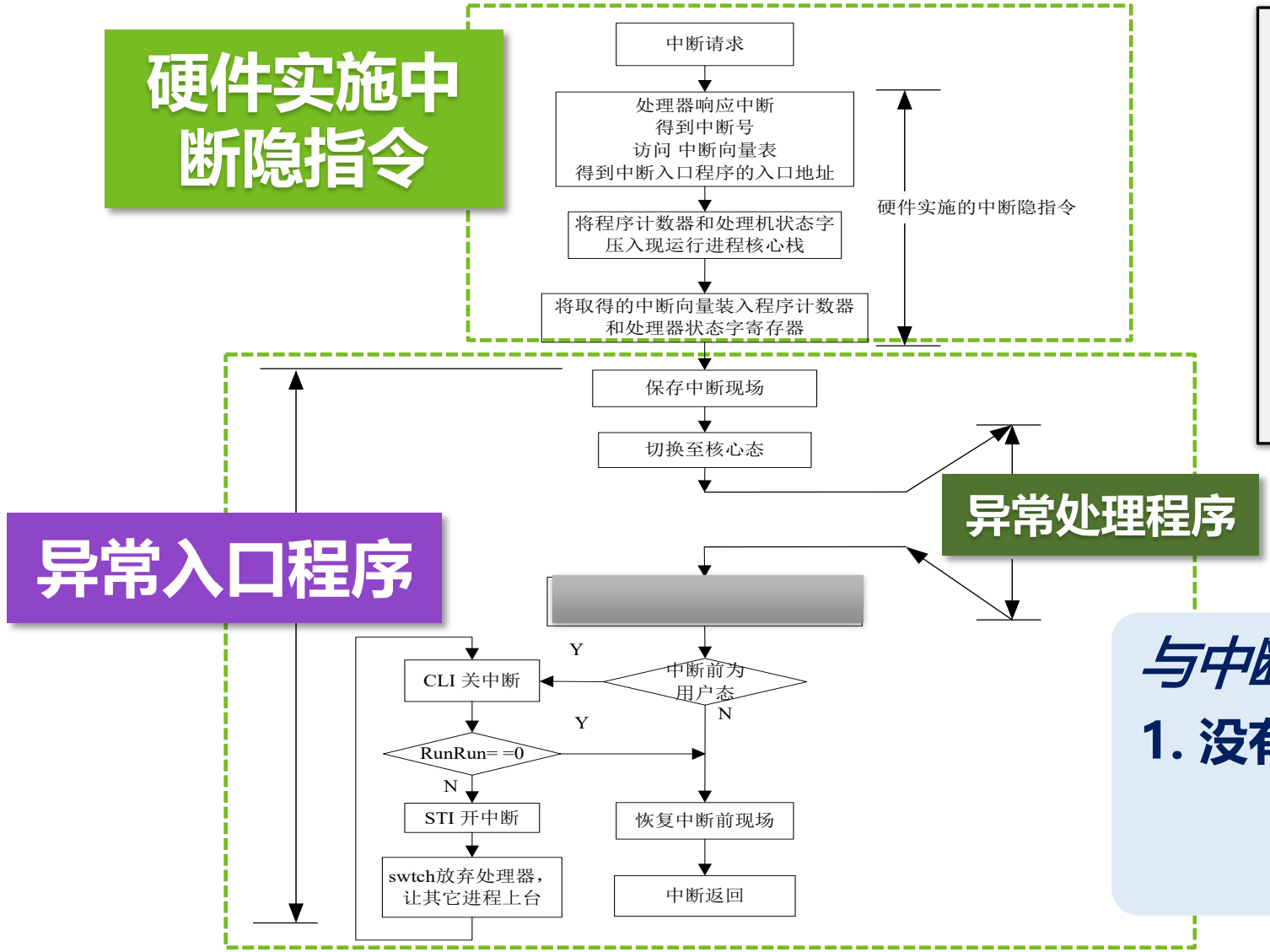



```
void Exception::DivideErrorEntrance()
{
        SaveContext();
        SwitchToKernel();

        CallHandler(Exception, DivideError);

        RestoreContext();
        Leave();
        InterruptReturn();
}
```

以除数为0的异常为例：

UNIX异常处理

概况




```
void Exception::DivideErrorEntrance()
{
        SaveContext();
        SwitchToKernel();

        CallHandler(Exception, DivideError);

        RestoreContext();
        Leave();
        InterruptReturn();
}
```

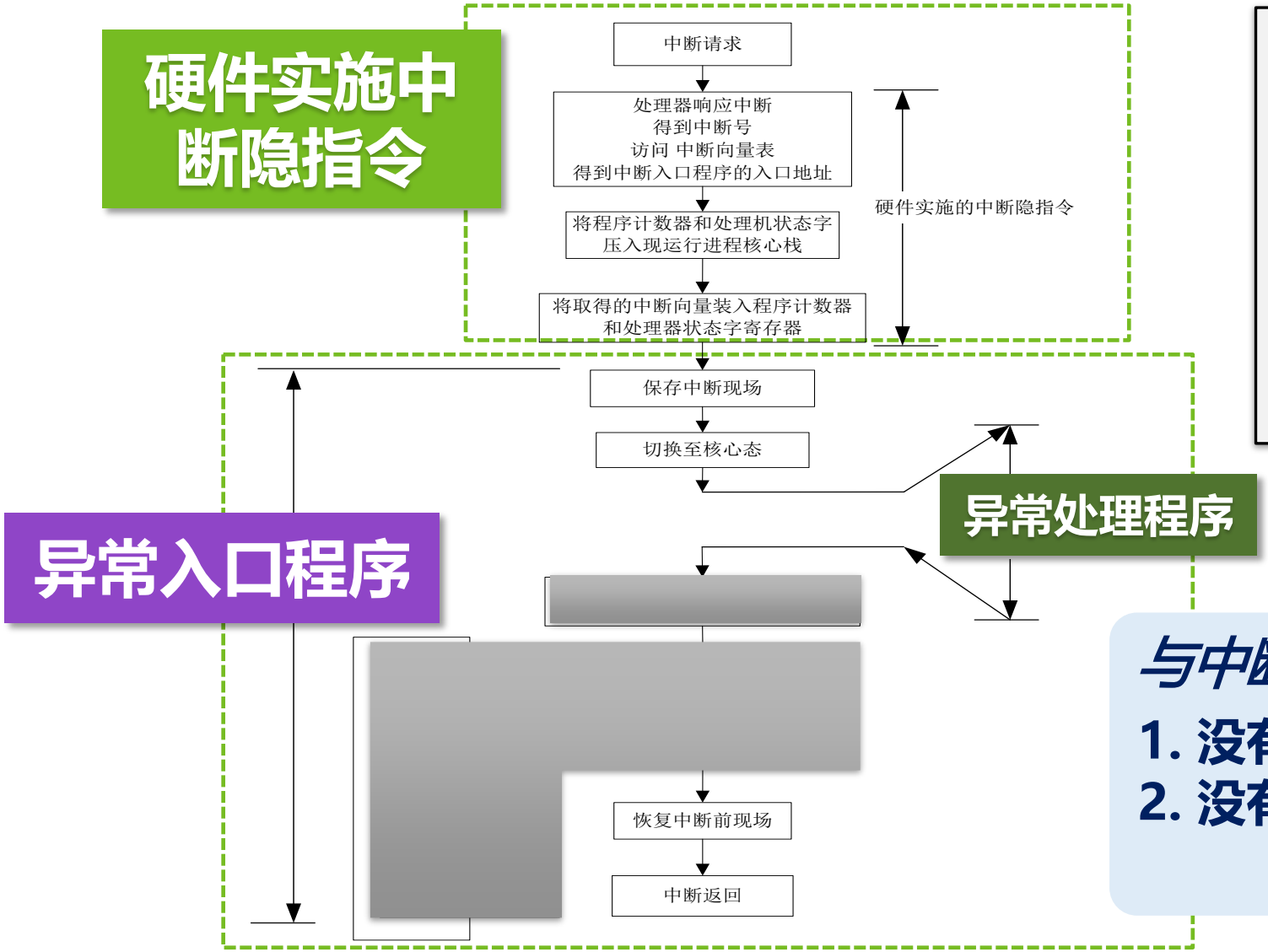
以除数为0的异常为例：

与中断的主要区别：

1. 没有EOI

UNIX异常处理

概况




```
void Exception::DivideErrorEntrance()
{
        SaveContext();
        SwitchToKernel();

        CallHandler(Exception, DivideError);

        RestoreContext();
        Leave();
        InterruptReturn();
}
```

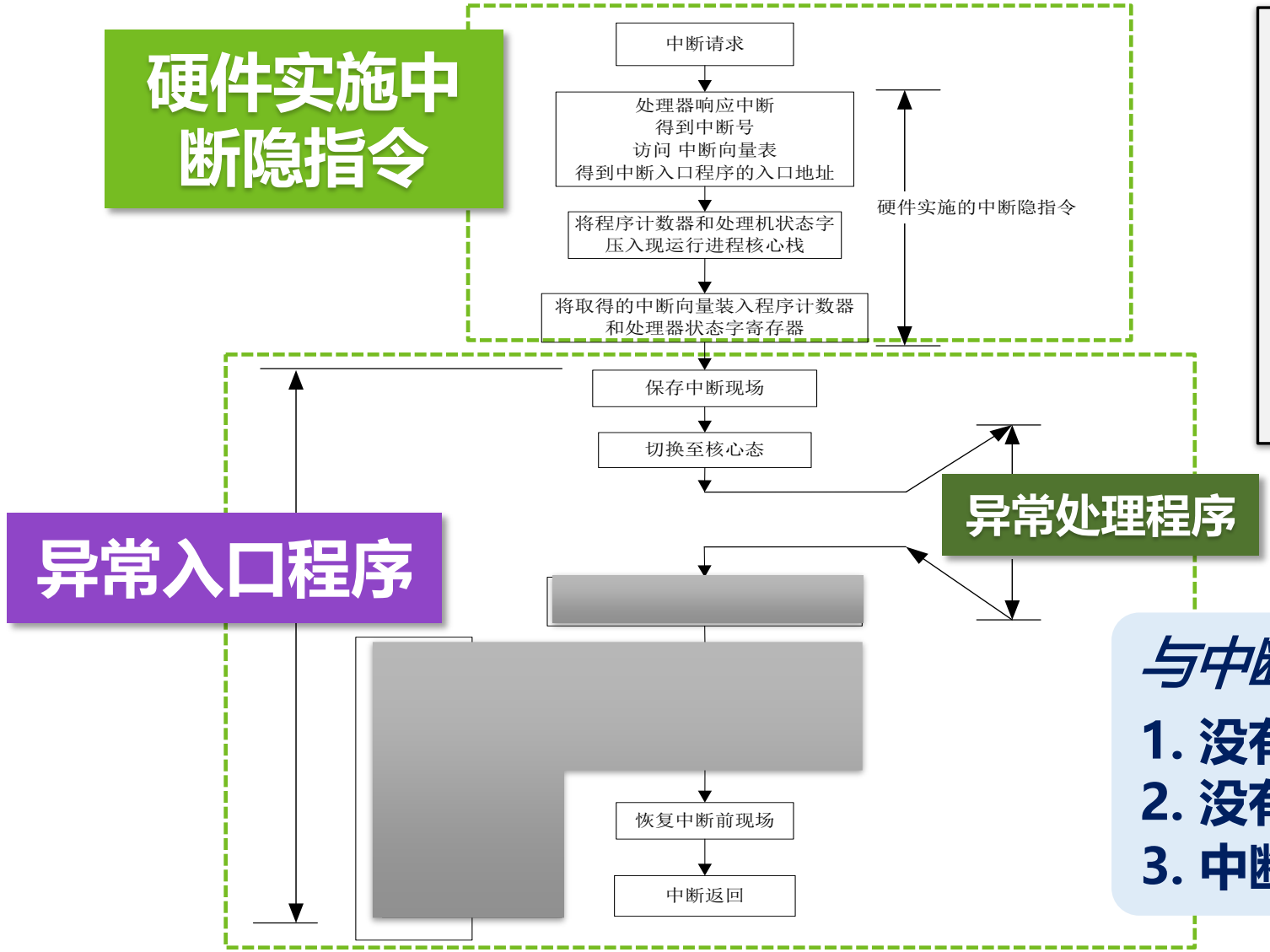
以除数为0的异常为例：

与中断的主要区别：

1. 没有EOI
2. 没有例行调度

UNIX异常处理

概况



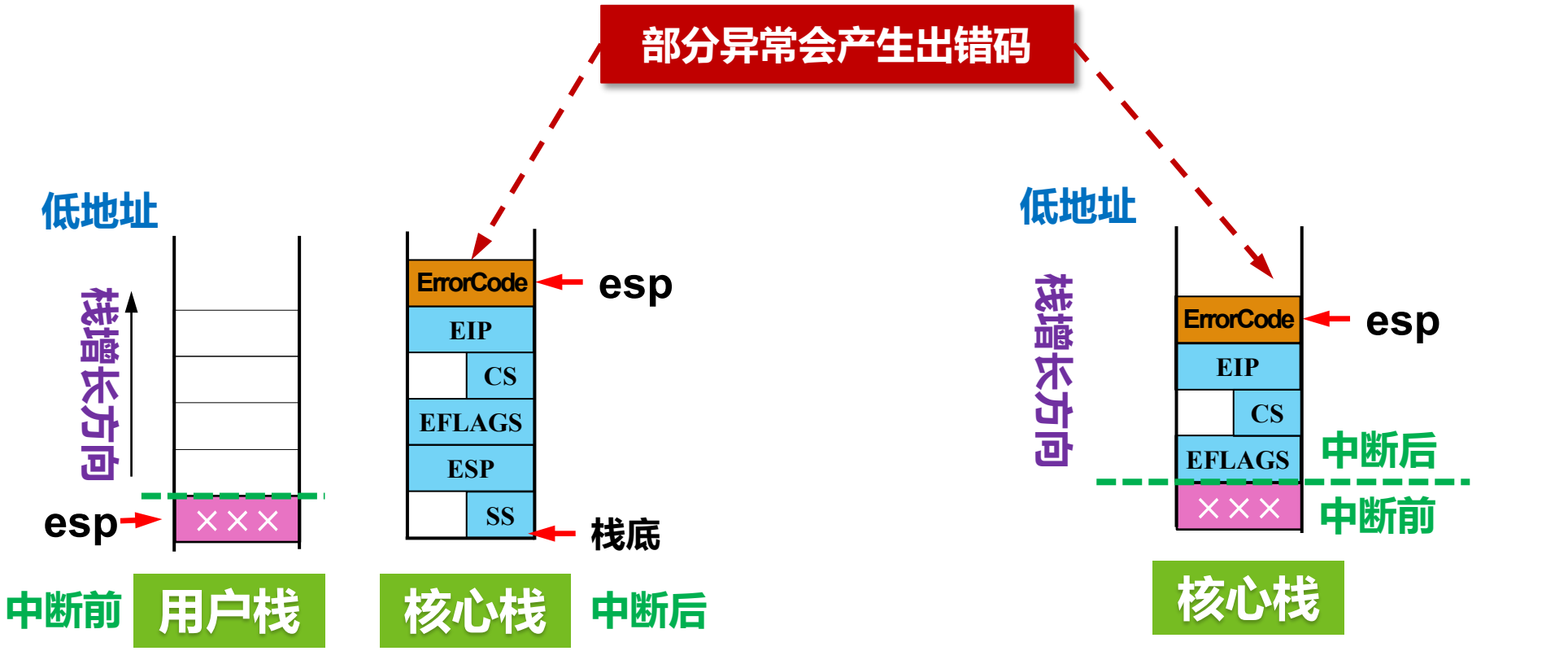

```
void Exception::DivideErrorEntrance()
{
        SaveContext();
        SwitchToKernel();

        CallHandler(Exception, DivideError);

        RestoreContext();
        Leave();
        InterruptReturn();
}
```

以除数为0的异常为例：

与中断的主要区别：

1. 没有EOI
2. 没有例行调度
3. 中断隐指令形成的硬件现场

UNIX异常处理



异常发生时现运行进程在用户态运行

异常发生时现运行进程在核心态运行

中断隐指令实施现场保护后的堆栈状态

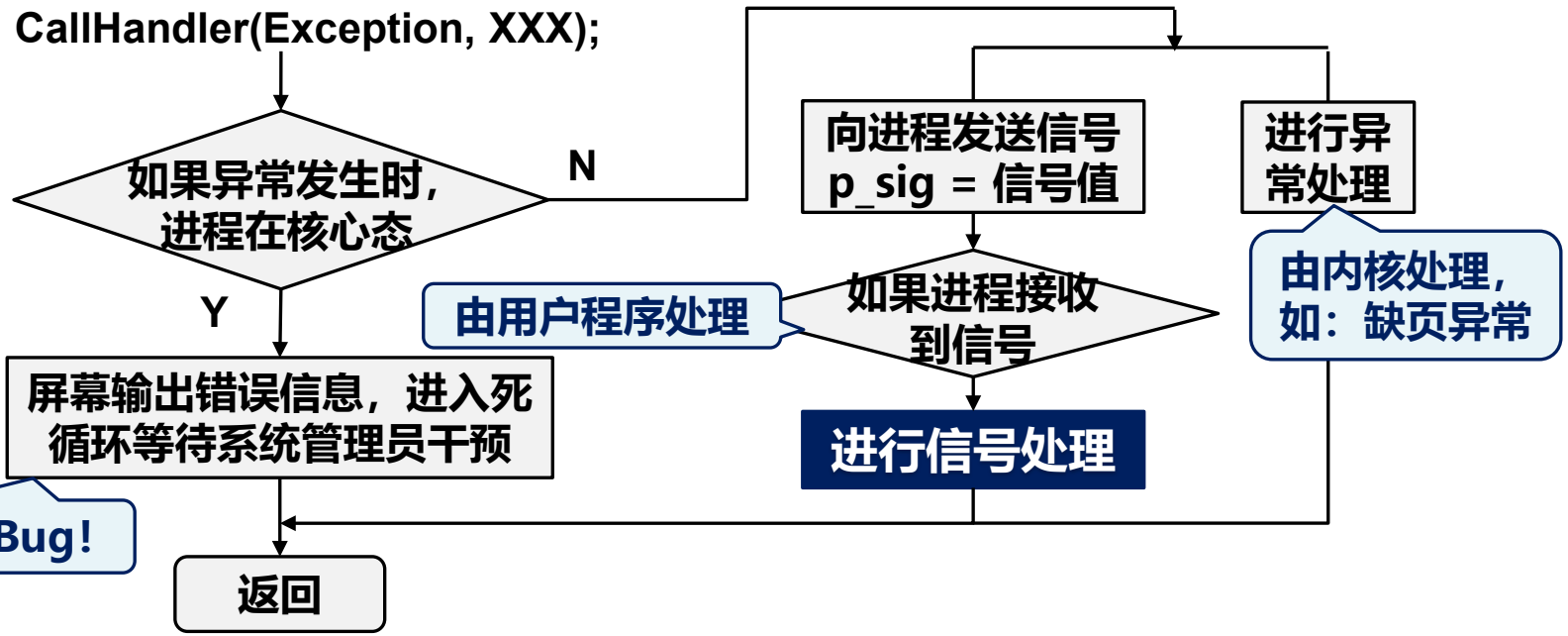
UNIX异常处理

异常处理的详细流程

异常处理的流程

```
void Exception::XXXEntrance()
{
        SaveContext();
        SwitchToKernel();

        CallHandler(Exception, XXX);

        RestoreContext();
        Leave();
        InterruptReturn();
}
```

操作系统内核有Bug!

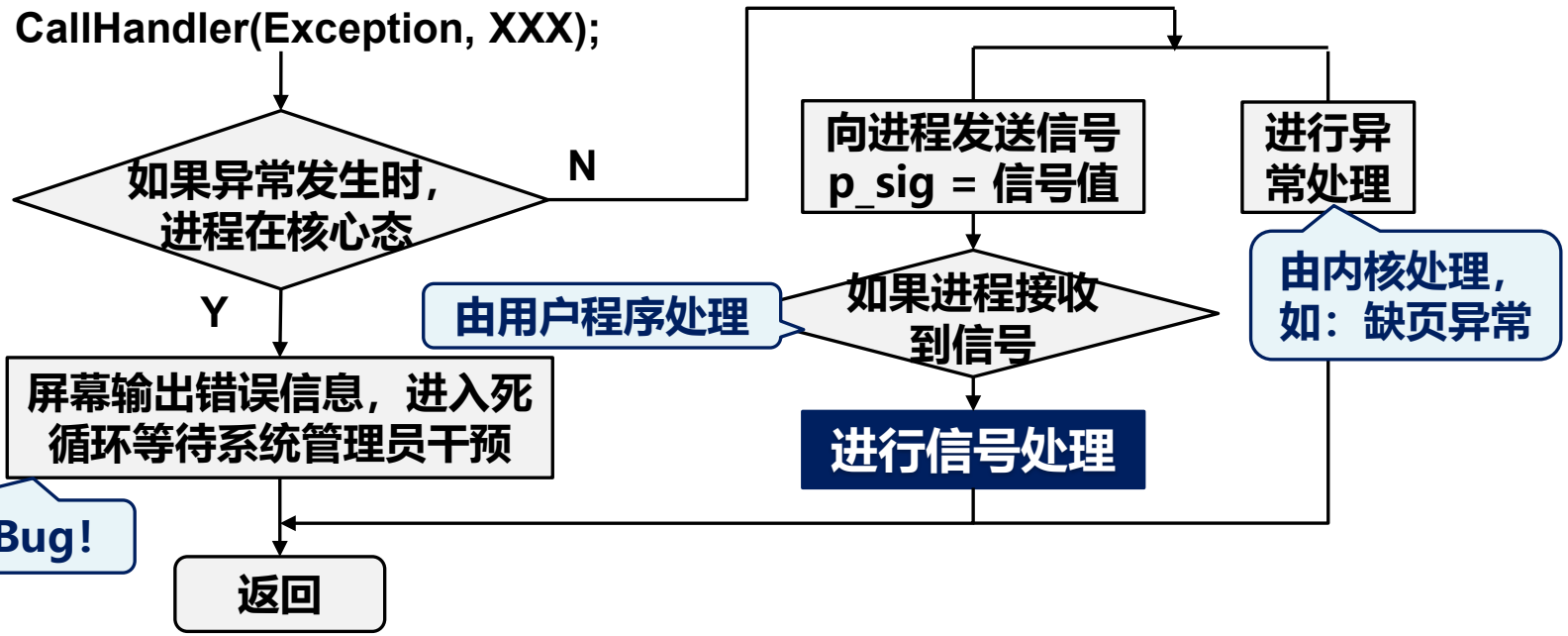


UNIX异常处理

异常处理的流程

```
void Exception::XXXEntrance()
{
    SaveContext();
    SwitchToKernel();

    CallHandler(Exception, XXX);

    RestoreContext();
    Leave();
    InterruptReturn();
}
```

操作系统内核有Bug!



User::u_signal[NSIG]

0	0	→ 终止本进程
1	X	→ 信号处理入口函数
	…	
14	0	

可通过系统调用设置

每个进程最多可接收15个不同的信号

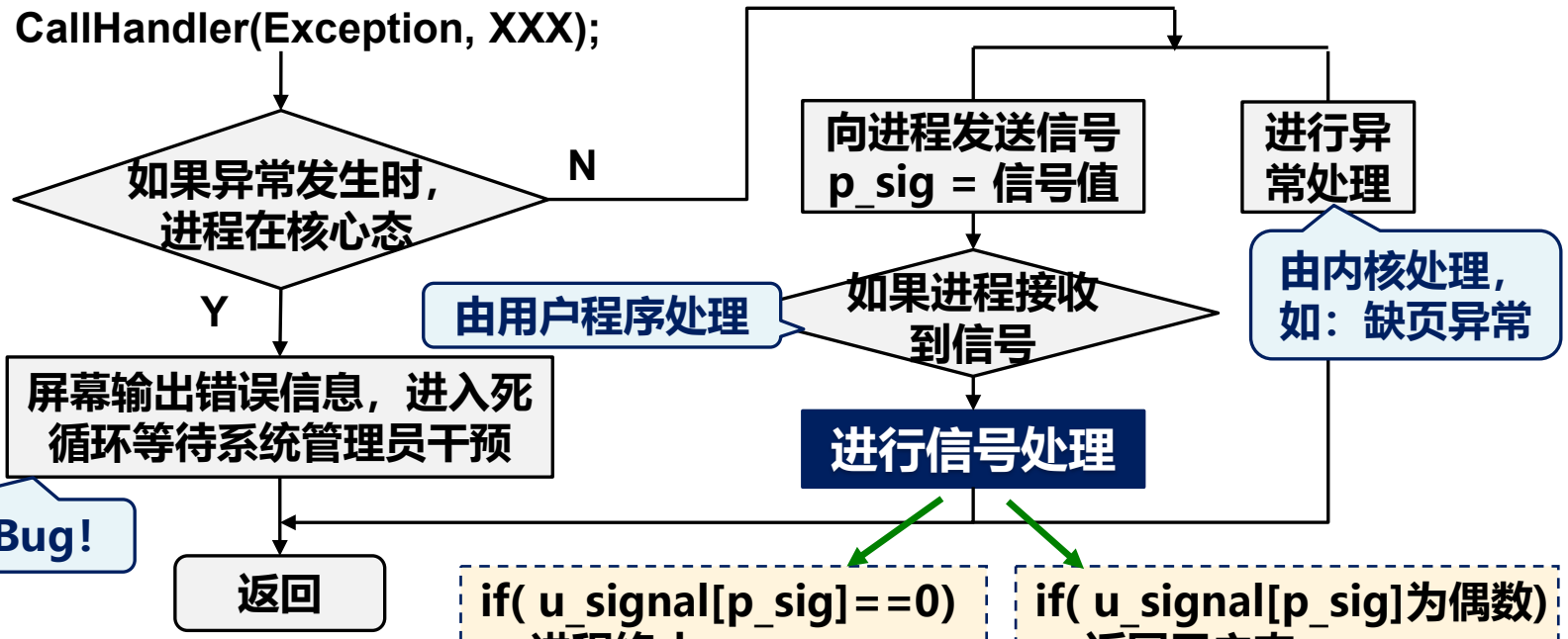
UNIX异常处理

异常处理的详细流程

异常处理的流程

```
void Exception::XXXEntrance()
{
    SaveContext();
    SwitchToKernel();

    CallHandler(Exception, XXX);

    RestoreContext();
    Leave();
    InterruptReturn();
}
```

操作系统内核有Bug!



User::u_signal[NSIG]

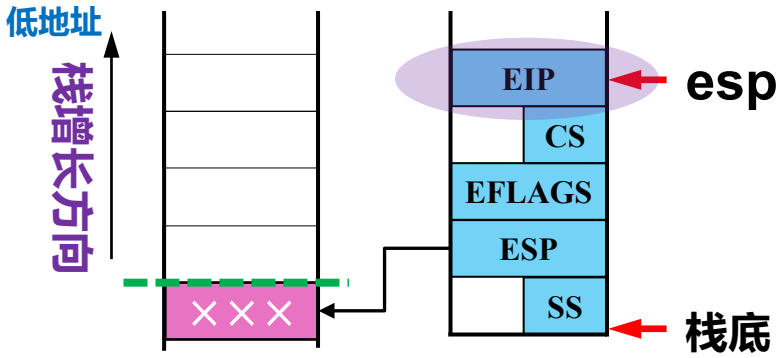
0	0	→ 终止本进程
1	X	→ 信号处理入口函数
	…	
14	0	

可通过系统调用设置

每个进程最多可接收15个不同的信号

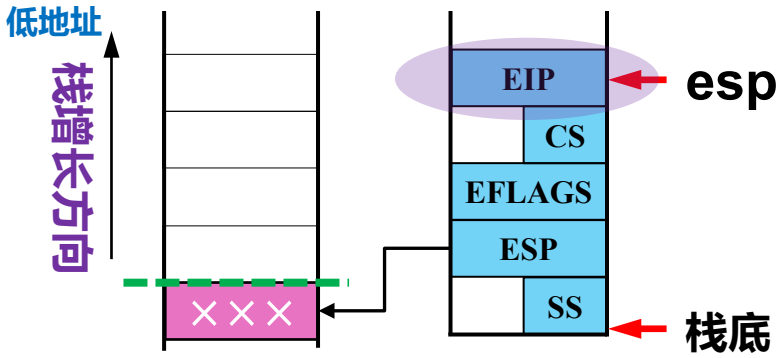
和一般的中断处理函数有什么区别?

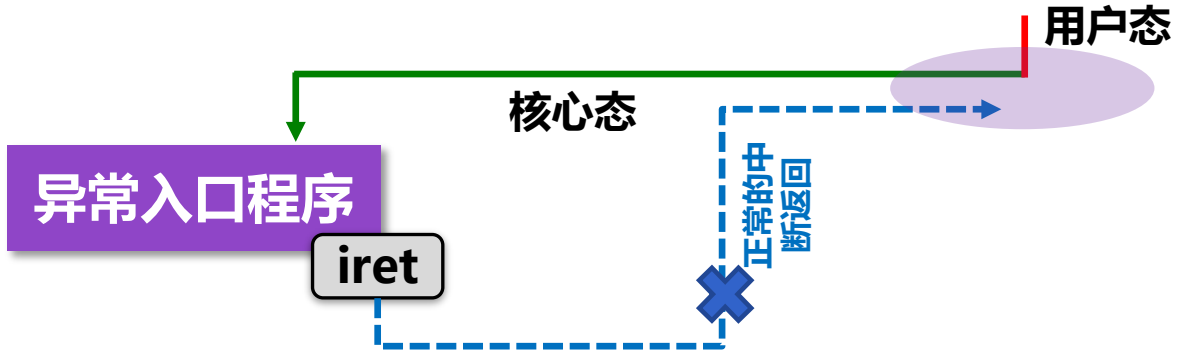
UNIX异常处理



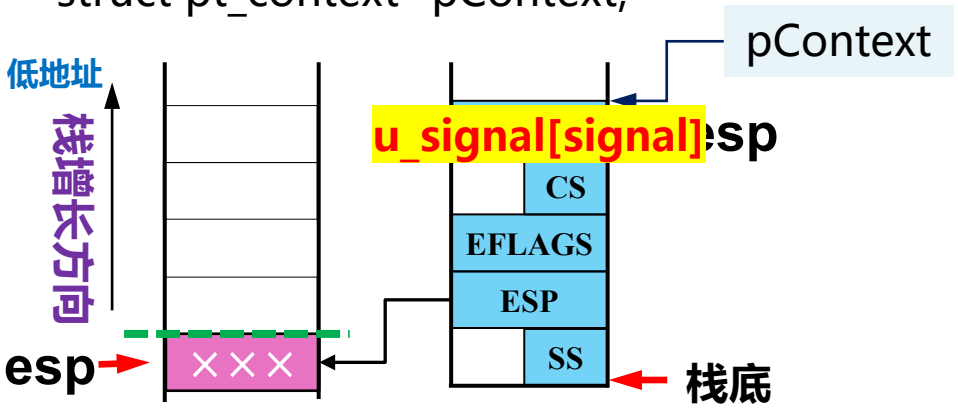
异常处理的详细流程

UNIX异常处理



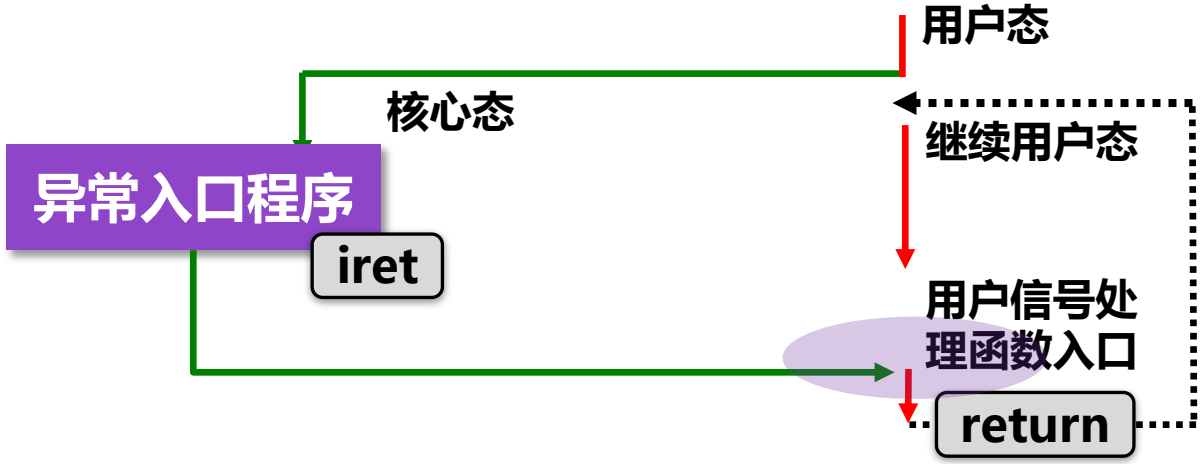


struct pt_context* pContext;

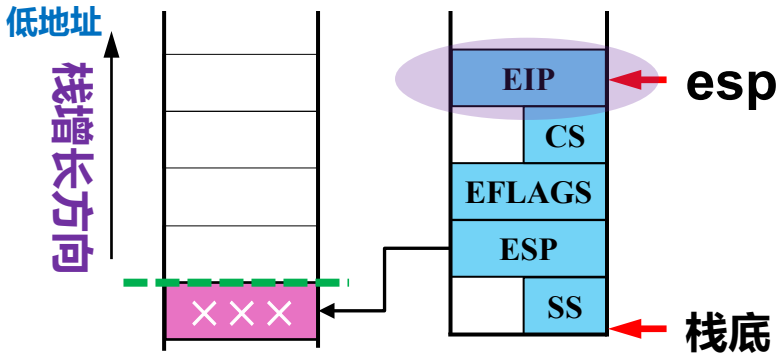


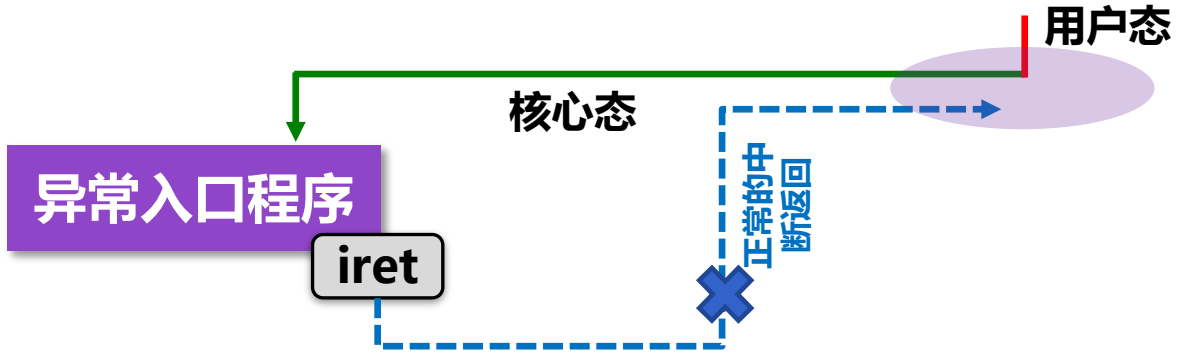
1. unsigned int old_eip = pContext->eip;
2. pContext->eip = u.u_signal[signal];

异常处理结束返回用户态时，回到用户定义的信号处理程序

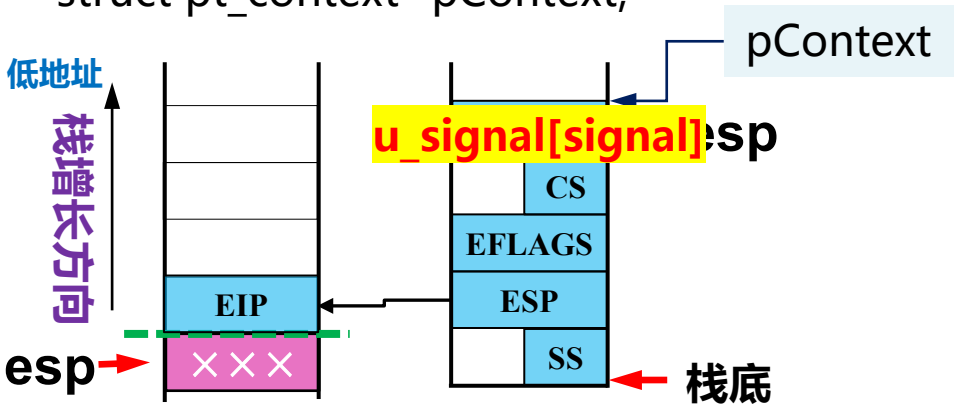


UNIX异常处理



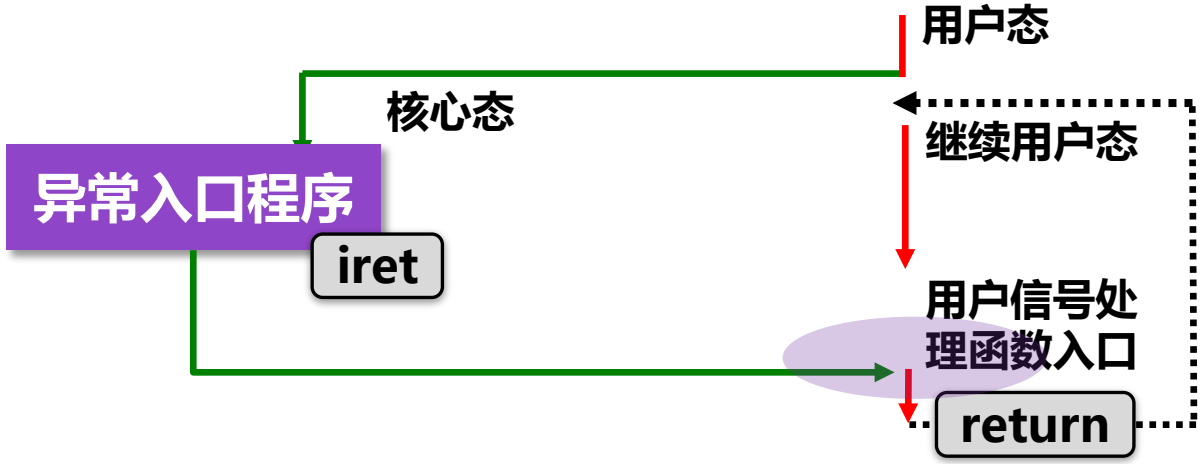


struct pt_context* pContext;



3. pContext->esp -= 4;
4. int* pInt = (int *)pContext->esp;
5. *pInt = old_eip;

异常处理结束返回用户态时，回到用户定义的信号处理程序



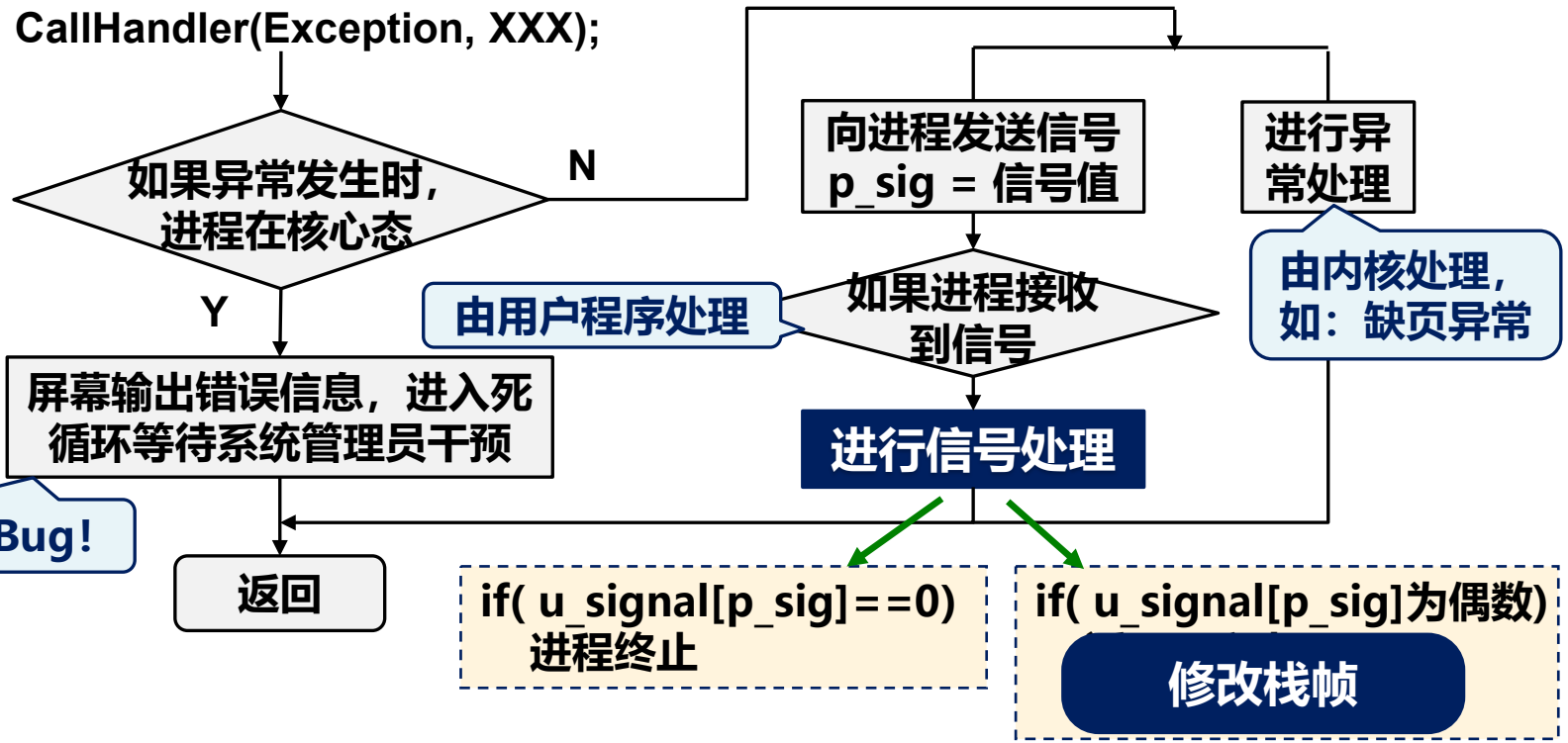
UNIX异常处理

异常处理的详细流程

异常处理的流程

```
void Exception::XXXEntrance()
{
    SaveContext();
    SwitchToKernel();

    CallHandler(Exception, XXX);

    RestoreContext();
    Leave();
    InterruptReturn();
}
```

操作系统内核有Bug!



User::u_signal[NSIG]

0	0	→ 终止本进程
1	X	→ 信号处理入口函数
	…	
14	0	

可通过系统调用设置

每个进程最多可接收15个不同的信号

本节小结

1　中断的基本概念

2　UNIX中断的处理过程

阅读教材：103页 ~ 110页

E10：进程管理（UNIX时钟中断）