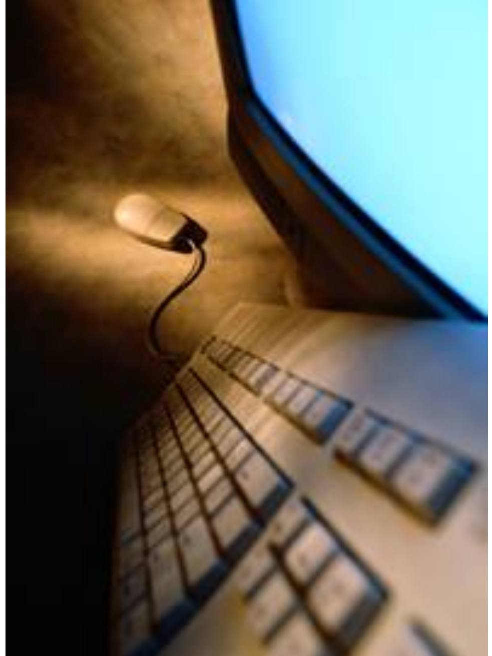
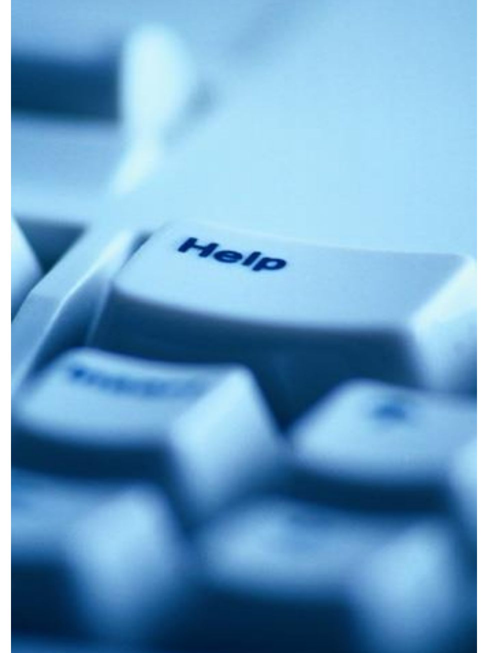
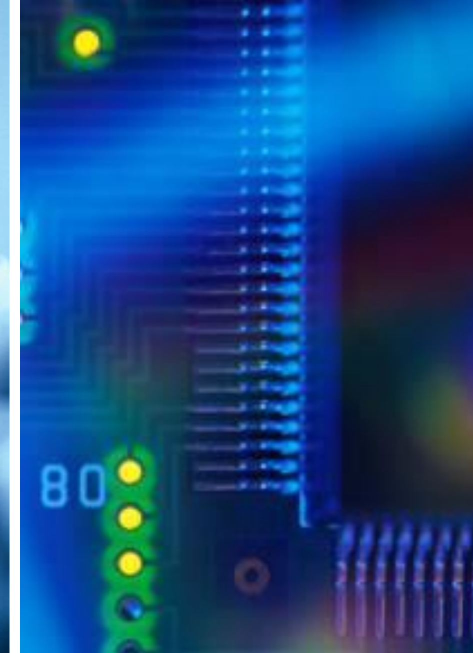






操作系统原理

Operating Systems Principles

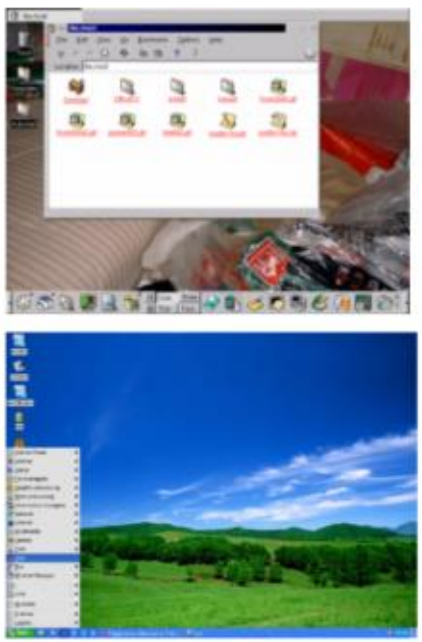
张青
计算机学院

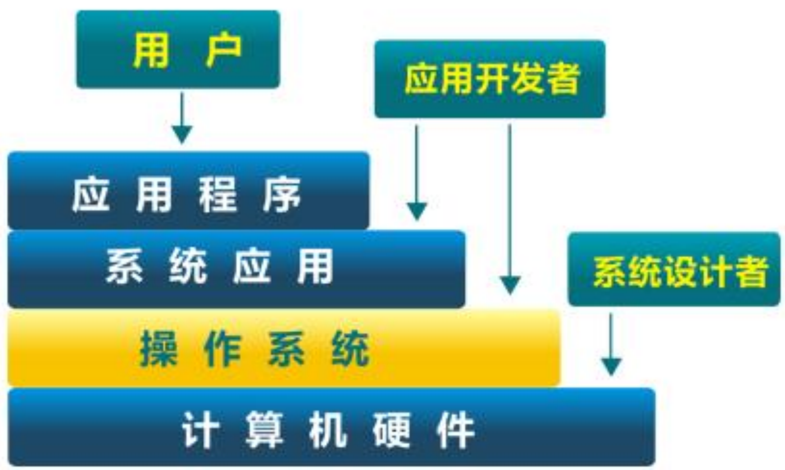
第三讲 —中断、异常

基本概念和原理

中断 (hardware interrupt)

来自硬件设备（外设, device）的处理请求



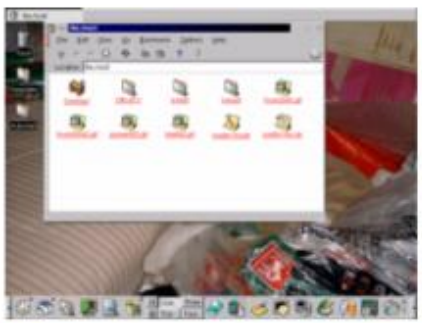


异步：产生原因和当前执行指令无关，如程序被磁盘读打断

基本概念和原理

异常（exception）

非法指令或者其他原因导致**当前指令执行失败**, (如：内存出错) 后的处理请求

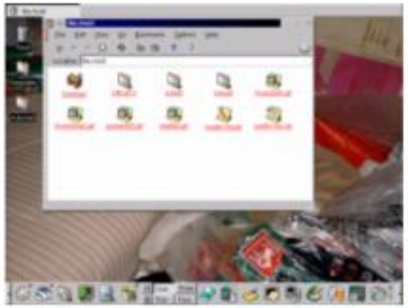


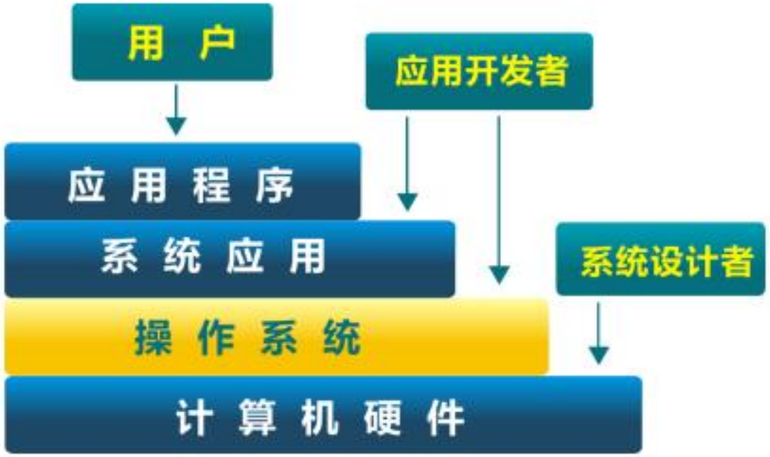
同步：产生和当前执行或试图执行的指令相关

基本概念和原理

系统调用 (system call)

应用程序**主动**向操作系统发出的服务请求

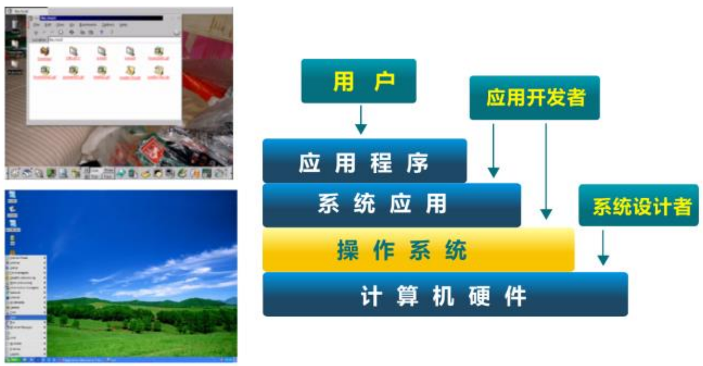




不同体系结构中的中断和异常

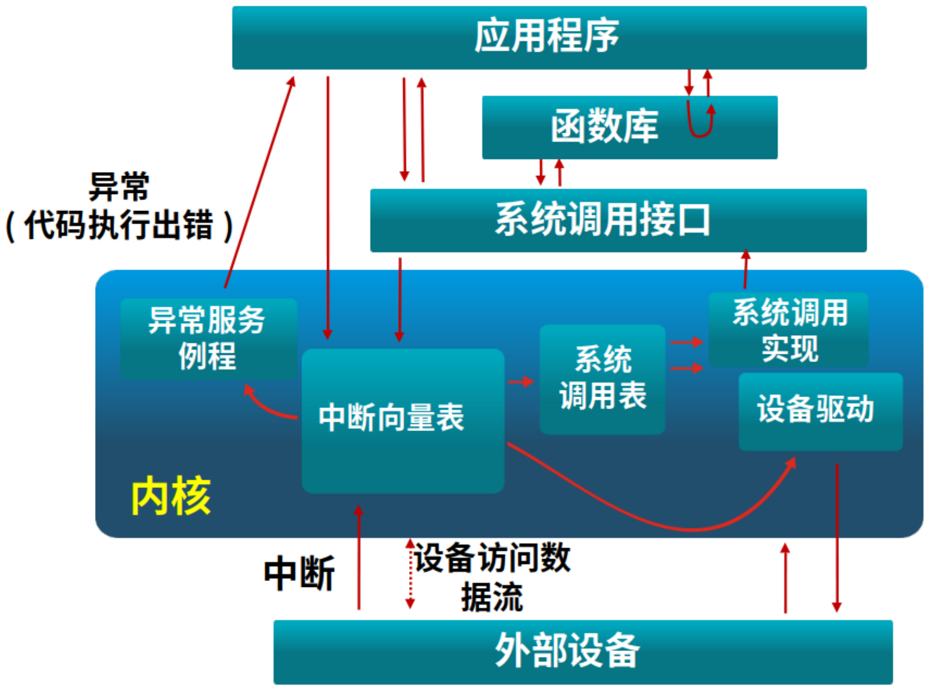
<table>
<tr><th>通用概念</th><th>产生原因</th><th colspan="2">AArch64</th><th>x86-64</th></tr>
<tr><td>中断</td><td>硬件
异步</td><td rowspan="2">异常</td><td>异步异常
（重置/中断）</td><td>中断
（可屏蔽/不可屏蔽）</td></tr>
<tr><td>异常</td><td>软件
同步</td><td>同步异常
（终止/异常指令）</td><td>异常
（Fault/Trap/Abort）</td></tr>
</table>

为什么需要



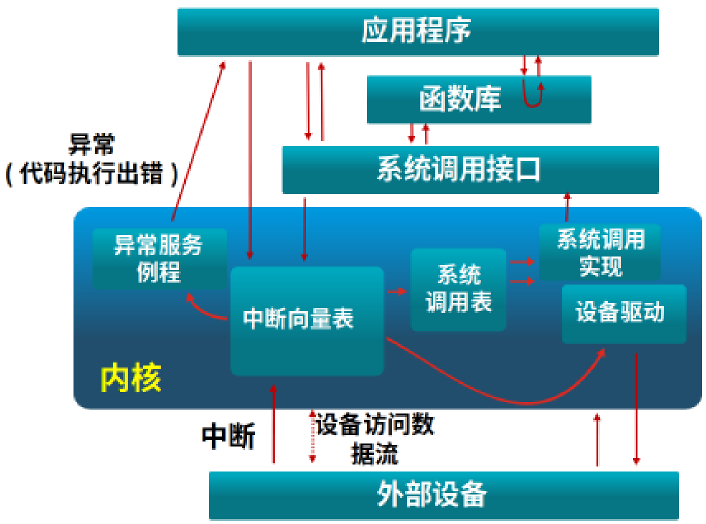
- 为什么需要中断、异常和系统调用
  - OS 内核是被信任的第三方
  - OS 内核可以执行特权指令，管理硬件
  - OS 内核提供了各种 Service
- 中断希望解决的问题
  - 当外设连接计算机时，会出现什么现象?
- 异常希望解决的问题
  - 当应用程序处理意想不到的行为时，会出现什么现象?
- 系统调用希望解决的问题
  - 用户应用程序是如何得到系统服务?

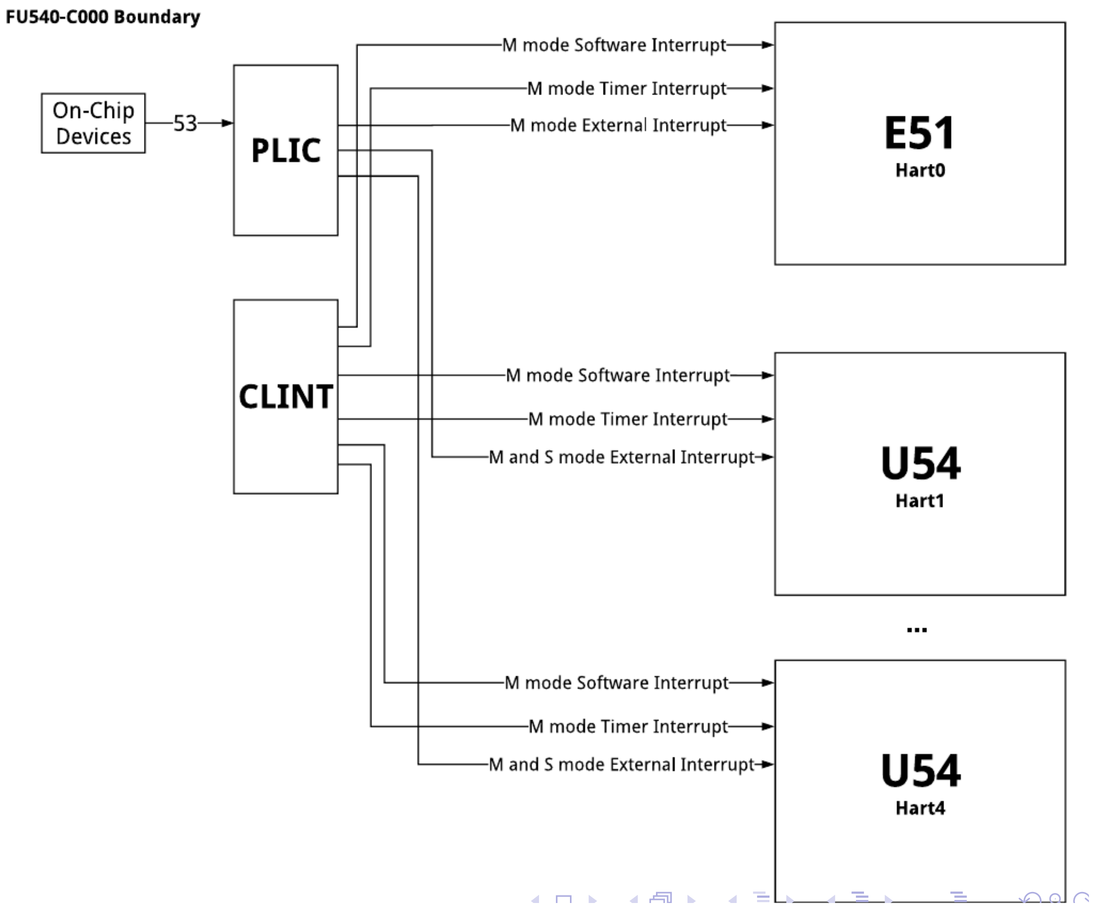
基本概念和原理



- 源头
  - 中断：外设
  - 异常：应用程序意想不到的行为
  - 系统调用：应用程序请求 OS 服务
- 响应方式
  - 中断：异步
  - 异常：同步
  - 系统调用：异步或同步
- 处理机制
  - 中断：持续，对应用程序透明
  - 异常：杀死或者重新执行
  - 系统调用：等待和持续

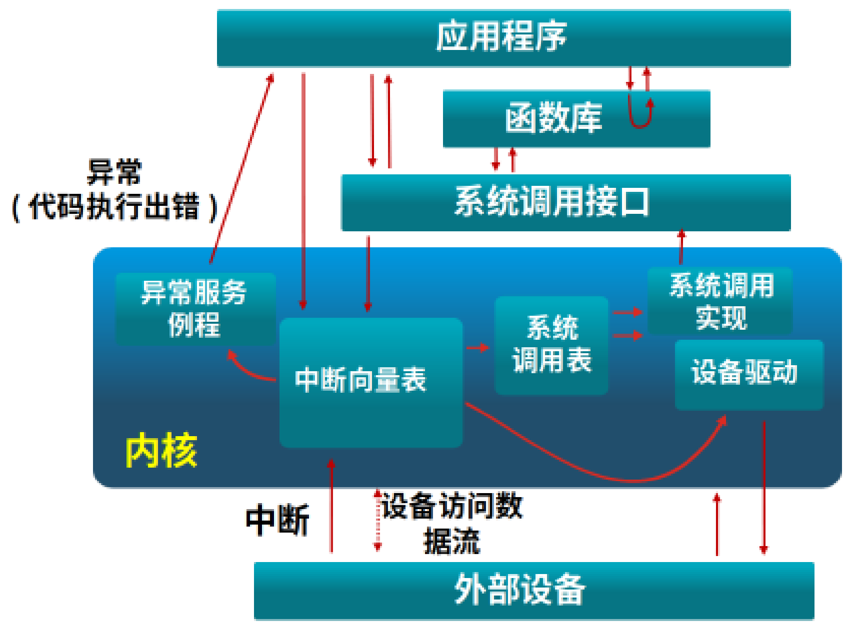
硬件支持

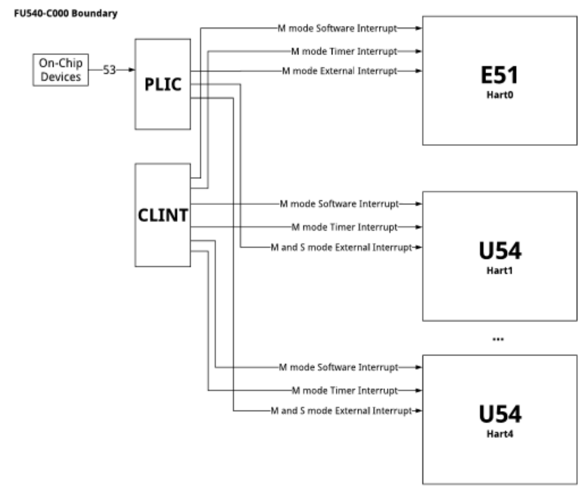




- Core Local Interruptor (CLINT)
- Platform-Level Interrupt Controller (PLIC)

硬件支持





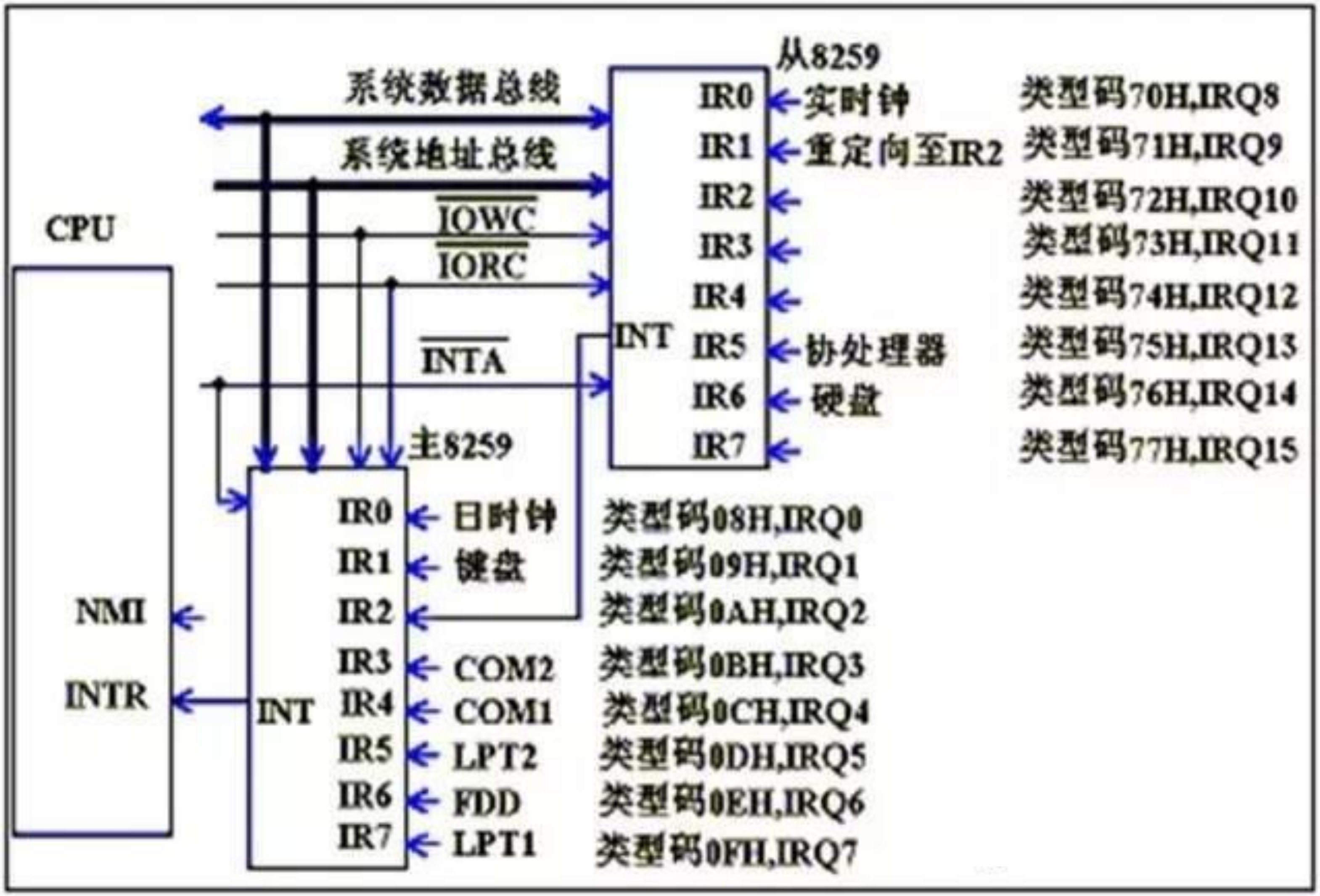
- Core Local Interruptor (CLINT)
- Platform-Level Interrupt Controller (PLIC)

三种标准的中断源:

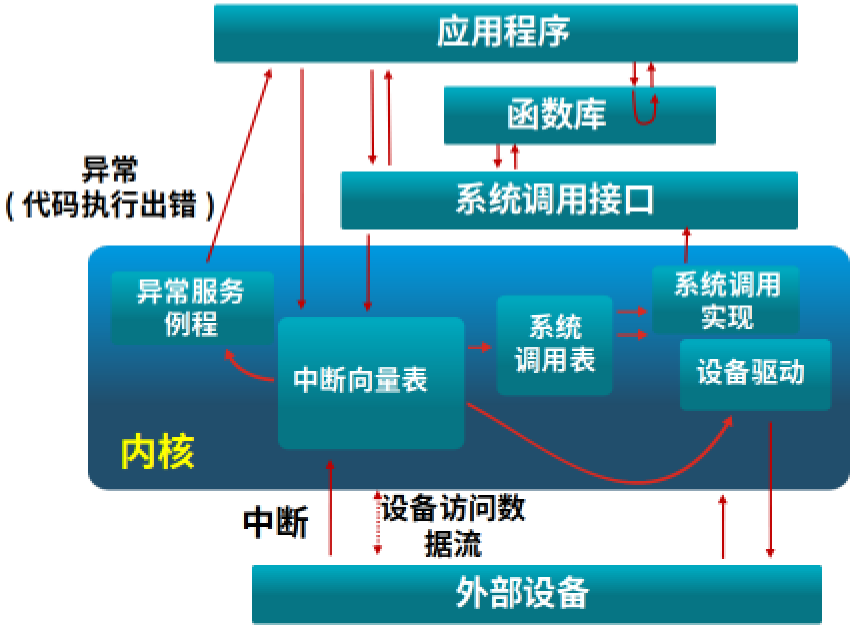
- 软件中断通过向内存映射寄存器中存数来触发，如 IPI
- 时钟中断, 如 stimecmp > stime
- 由平台级中断控制器引发外部中断

中断处理机制

X86平台 8259中断处理



中断处理机制



- Core Local Interruptor (CLINT)
- Platform-Level Interrupt Controller (PLIC)

建立中断机制

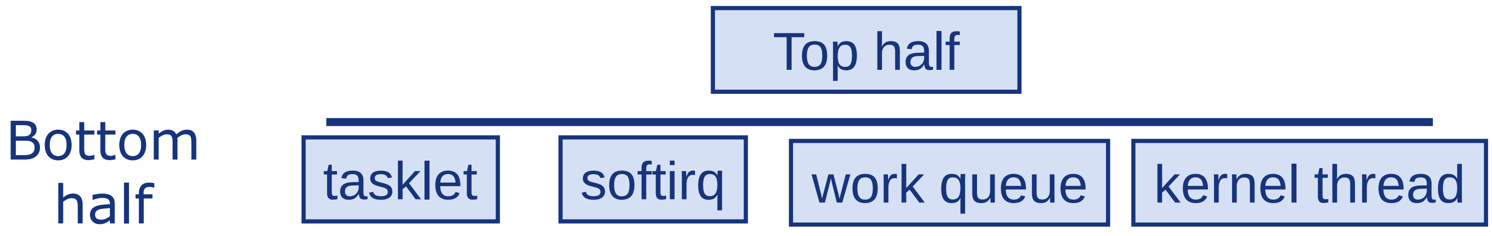
- 建立中断服务例程
- 让 CPU 能响应中断
- 响应并处理中断
- 保存/恢复现场

Linux系统中的中断

- **中断（设备产生、异步）**
  - 可屏蔽：设备产生的信号，通过中断控制器与处理器相连，可被暂时屏蔽（如，键盘、网络事件）
  - 不可屏蔽：一些关键硬件的崩溃（如，内存校验错误）
- **异常（软件产生、同步）**
  - **错误（Fault）:** 如缺页异常（可恢复）、段错误（不可恢复）等
  - **陷阱（Trap）:** 无需恢复，如断点（int 3）、系统调用（int 80）
  - **中止（Abort）:** 严重的错误，不可恢复（机器检查）

Linux系统中的中断

- 在中断处理中做尽量少的事
- 推迟非关键行为
- 结构：Top half & Bottom half
  - Top half ：做最少的工作后返回
  - Bottom half ：推迟处理（softirq，tasklets，工作队列，内核线程）



Top Half：马上做

- 最小的、公共行为
  - 保存寄存器、屏蔽其他中断
  - 恢复寄存器，返回原来场景
- 最重要：调用合适的由硬件驱动提供的中断处理handler
- 因为中断被屏蔽，所以不要做太多事情（时间、空间）
- 使用将请求放入队列，或者设置标志位将其他处理推迟到bottom half

Top Half：找到handler

- 现代处理器中，多个I/O设备共享一个IRQ和中断向量
- 多个ISR (interrupt service routines)可以结合在一个向量上
- 调用每个设备对应该IRQ的ISR

Bottom Half：延迟完成

- 提供一些推迟完成任务的机制
  - softirqs
  - tasklets （建立在softirqs之上）
  - 工作队列
  - 内核线程
- 这些工作可以被中断

注意：中断处理没有进程上下文

- 中断（和异常相比）和具体的某条指令无关
- 也和中断时正在跑的进程、用户程序无关
- 中断处理handler不能睡眠！

中断处理中的一些约束条件

- 不能睡眠
  - 或者调用可能会睡眠的任务
- 不能调用schedule()调度
- 不能释放信号或调用可能睡眠的操作
- 不能和用户地址空间交换数据

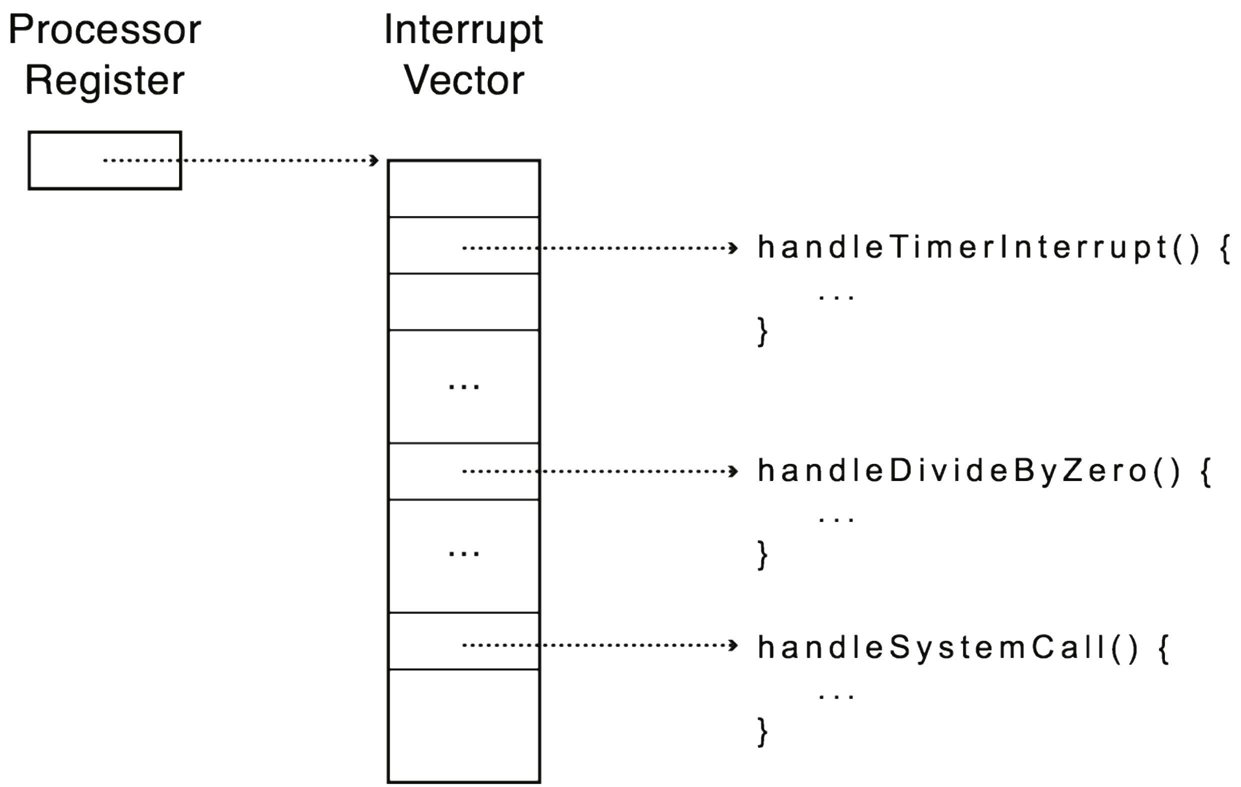
Demo

如何安全执行中断

- **Interrupt Vector Table**
  - Where the processor looks for a handler
  - Limited number of entry points into kernel
  - Stored in RAM at a known address
- **Atomic transfer of control**
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- **Transparent restartable execution**
  - User program does not know interrupt occurred

中断向量表

Table set up by OS kernel; pointers to code to run on different events

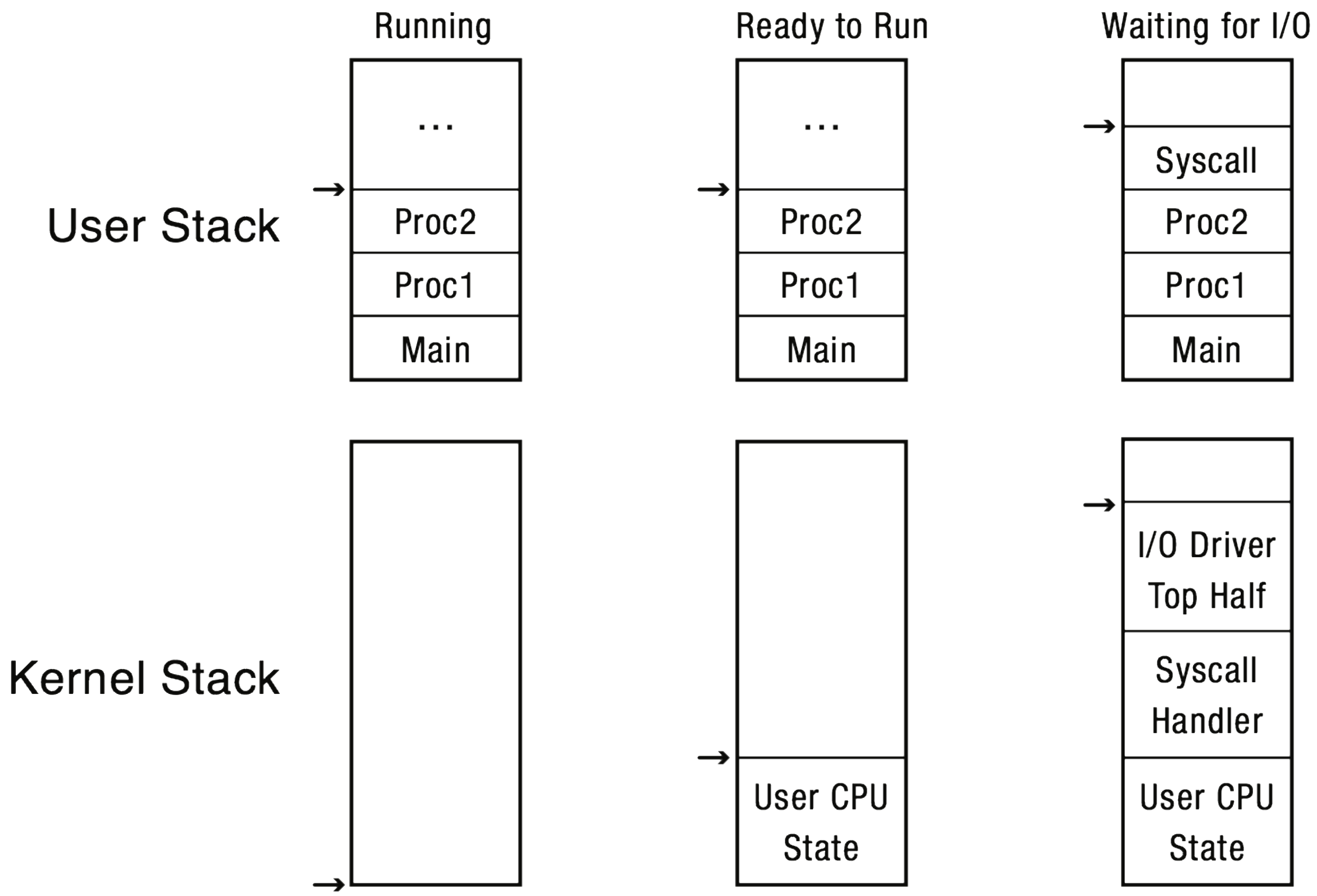


中断向量表

- Per-processor, located in kernel (not user) memory
  - Fun fact! Usually a process/thread has both a kernel and user stack

- **Can the interrupt handler run on the stack of the interrupted user process?**

中断向量表


Running
Ready to Run
Waiting for I/O
User Stack
Kernel Stack
...
Proc2
Proc1
Main
...
Proc2
Proc1
Main
Syscall
Proc2
Proc1
Main
User CPU State
I/O Driver Top Half
Syscall Handler
User CPU State

为什么使用INT

- Hardware devices may need asynchronous and immediate service. For example:
  - Timer interrupt: Timers and time-dependent activities need to be updated with the passage of time at precise intervals
  - Network interrupt: The network card interrupts the CPU when data arrives from the network
  - I/O device interrupt: I/O devices (such as mouse and keyboard) issue hardware interrupts when they have input (e.g., a new character or mouse click)

多核上的中断

- How are interrupts handled on multicore machines?
  - On x86 systems each CPU gets its own local Advanced Programmable Interrupt Controller (APIC). They are wired in a way that allows routing device interrupts to any selected local APIC.
  - The OS can program the APICs to determine which interrupts get routed to which CPUs.
  - The default (unless OS states otherwise) is to route all interrupts to processor 0

其他类型的中断

- Software Interrupts:
  - Interrupts caused by the execution of a software instruction:
    - `INT <interrupt_number>`
  - Used by the system call `interrupt()`
- Initiated by the running (user level) process
- Cause current processing to be interrupted and transfers control to the corresponding interrupt handler in the kernel

其他类型的中断

- Exceptions
  - Initiated by processor hardware itself
  - Example: divide by zero
- Like a software interrupt, they cause a transfer of control to the kernel to handle the exception

其他类型的中断

- Exceptions
  - Initiated by processor hardware itself
  - Example: divide by zero
- Like a software interrupt, they cause a transfer of control to the kernel to handle the exception

其他类型的中断

- HW -> CPU -> Kernel: Classic HW Interrupt
- User -> Kernel: SW Interrupt
- CPU -> Kernel: Exception
- Interrupt Handlers used in all 3 scenarios

中断屏蔽

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run

中断处理程序

Designing an Interrupt Handler:

- Since the interrupt handler must be minimal, all other processing related to the event that caused the interrupt must be deferred
  - Example:
    - Network interrupt causes packet to be copied from network card
    - Other processing on the packet should be deferred until its time comes
- The deferred portion of interrupt processing is called the "Bottom Half"

Bottom half

- Method for deferring portion of interrupt processing
- Globally serialized
  - When one bottom half is executing, no other bottom half can execute (even different type) on any CPU.
- Obvious performance limitations; primarily available for legacy support.
- Note: other mechanisms for deferred work are also sometimes referred to as bottom half mechanisms.

软中断

- Handlers that, like bottom halves, must be statically defined/allocated in the Linux kernel at compile time.
- A hardware interrupt handler (before returning) uses raise_softirq() to mark that a given soft_irq must execute deferred work
- At a later time, when scheduling permits, the marked soft_irq handler is executed
  - When a hardware interrupt is finished
  - When a process makes a system call
  - When a new process is scheduled
- Unlike bottom halves, softirqs are reentrant and can be executed concurrently on several CPUs

谢谢