深入理解 Linux 内核 第四章中断和异常

Contents

第四章 中断和异常

中断 interrupt 通常被定义为一个事件，改事件改变处理器执行的指令顺序。这样的事件与 CPU 芯片内部外部硬件电路产生的电信号相对应。

中断通常分为同步 synchronous 中断和异步 asyncronous 中断：

- 同步中断是当指令执行时由 CPU 控制单元产生的，之所以称为同步，是因为只有在一条指令终止执行后 CPU 才会发出中断。
- 异步中断是由其他硬件设备依照 CPU 时钟信号随机产生的。

对于 Intel 处理器，把同步和异步中断分别称为异常 exception 和中断 interrupt。

中断是由间隔定时器和 I/O 设备产生的。

异常是由程序的错误产生的，或者是由内核必须处理的异常条件产生的。前者，内核通过发送一个每个 Unix 程序员都熟悉的信号来处理异常；后者，内核执行回复异常需要的所有步骤，例如缺页、或对内核服务的一个请求（通过一条 int 或 sysenter 指令）。

中断信号的作用

中断信号提供了一种特殊的方式，使处理器转而去运行正常控制流之外的代码。当一个中断信号达到时，CPU 必须停止它当前正在做的事情，并且切换到一个新的活动。为了做到这一点，就要在内核态堆栈保存程序计数器的当前值，并把与终端类型相关的一个地址放进程序计数器。

终端处理与进程切换有一个明显的差异：由中断或异常处理程序执行的代码不是一个进程，终端处理程序比一个进程要轻（light，中断的上下文很少，建立或终止终端处理需要的时间很少）。

终端处理是由内核执行的最敏感的任务之一，必须满足下列约束：

- 当内核正打算去完成一些别的事情时，中断随时会到来。内核的目标就是让中断尽可能快地处理完，尽可能把更多的处理向后推迟。内核响应中断后需要进行的操作分为两部分：关键而紧急的部分，内核立即执行；其余推迟的部分，内核随后执行。
- 内核正在处理一个中断时，另一个中断（不同类型）又发生了。终端处理程序必须编写成使相应的内核控制路径能以嵌套的方式的执行。
- 内核代码中存在一些临界区，在临界区中，中断必须被禁止。

中断与异常

中断和异常可分为以下几类：

- 中断
- 可屏蔽中断 maskable interrupt。 IO 设备发出的所有中断请求 IRQ 都产生可屏蔽终端。可屏蔽中断可以处于两种状态：屏蔽的或费屏蔽的的；一个屏蔽的中断只要还是屏蔽的，控制单元就忽略它。
- 非屏蔽中断 nonmaskable interrupt。 只有几个危机事件，如硬件故障，才引起非屏蔽中断。非屏蔽中断总是由 CPU 辨认。
- 异常
- 处理器探测异常 process-detected exception。 当 CPU 执行指令时探测到一个反常条件所产生的异常。可以分为三种：
- 故障 fault。通常可以纠正；一旦纠正，程序就可以在不失连贯性的情况下重新开始。在 eip 寄存器中保存了引起故障的指令地址，当

异常处理程序终止时，该指令会被重新执行。

- 陷阱 trap。在陷阱指令执行后立即报告；内核把控制权返回给程序后就可以继续它的执行而不失连贯性。陷阱的主要的作用是为了调试程序。

- 异常中止 abort。发生了一个严重的错误，控制单元出了问题，不能在 eip 寄存器中保存引起异常的指令所在的位置。异常中止用于报告严重的错误，如硬件故障或系统表中无效的值或不一致的值。异常中止处理程序会强制终止受影响的进程。

- 编程异常 programmed exception。 控制单元把编程异常作为陷阱来处理。编程异常通常也叫做软中断 software interrupt。有两种用途：执行系统调用及给调试程序通报一个特定的事件。

每个中断和异常是由 0-255 之间的一个数来标识，这个 8 位无符号整数叫做一个向量 vector。非屏蔽中断的向量和异常的向量是固定的，可屏蔽中断的向量可以通过对中断控制器的编程来改变。

IRQ 和中断

每个能发出中断请求的硬件控制设备都由一条名为 IRQ Interrupt ReQuest 的输出线。所有的 IRQ 线 IRQ line 都与一个名为可编程中断控制器 Programmable Interrupt Controuer PIC 的硬件电路的输入引脚相连。可编程中断控制器执行下列动作：

1. 监视 IRQ 线，检查产生的信号 raised signal。如果有两条以上的 IRQ 线上产生信号，选择引脚编号较小的 IRQ 线。
2. 如果一个引发信号出现在 IRQ 线上：
- 把收到的信号转换成对应的向量。
- 把向量存放在中断控制器的一个 IO 端口，允许 CPU 通过数据总线读此向量。
- 把引发信号发送到处理器的 INTR 引脚，即产生一个中断。
- 等待，直到 CPU 通过把这个中断信号写进可编程中断控制器的一个 IO 端口来确认；当这种情况发生时，清 INTR 线。
3. 返回第 1 步。

IRQ 线是从 0 开始顺序编号的。如前所述，通过向中断控制器端口发布合适的指令，就可以修改 IRQ 和向量之间的映射。

可以有选择地禁止每条 IRQ 线。禁止的中断是丢失不了的，它们一旦被激活，PIC 就又把它们发送到 CPU。这个特点被大多数中断处理程序使用，因为这允许中断处理程序逐次地处理同一类型的 IRQ。

有选择地激活禁止 IRQ 线不同于可屏蔽中断的全局屏蔽/非屏蔽。当 eflags 寄存器的 IF 标志被清 0 时，由 PIC 发布的每个可屏蔽中断都由 CPU 暂时忽略。

高级可编程中断控制器

对于多处理器来说，为了能够把中断传递给系统中的每个 CPU，引入了 IO 高级可编程控制器 Advanced Programmable Interrupt Controller，APIC。

来自外部硬件设备的中断请求以两种方式在可用 CPU 之间分发：

- 静态分发。 IRQ 信号传递给重定向表响应项中列出的本地 APIC。中断立即传递给一个特定的 CPU，或一组 CPU，或所有 CPU。
- 动态分发。 如果处理器正在执行最低优先级的进程，IRQ 信号就传递给这种处理器的本地 APIC。

如果两个或多个 CPU 共享最低优先级，利用仲裁 arbitration 技术在这些 CPU 之间分配负荷。

多 APIC 系统还允许 CPU 产生处理器间中断 interprocessor interrupt。处理器间终端 IPI，是 SMP 体积结构至关重要的组成部分。

异常

内核必须为每一种异常低通一个专门的异常处理程序。对于某些异常，CPU 控制单元在开始执行异常处理程序前会产生一个硬件出错码 hardware error code，并压入内核态堆栈。

异常处理程序发送的信号

编号	异常	信号
0	Divide error	SIGFPE
1	Debug	SIGTRAP
2	NMI	None
3	Breakpoint	SIGTRAP
4	Overflow	SIGSEGV
5	Bounds check	SIGSEGV
6	Invalid opcode	SIGILL
7	Device not avalibale	None
8	Double fault	None
9	Coprocessor segment	SIGFPE
10	Invalid TSS	SIGSEGV
11	Segment not present	SIGBUS
12	Stack exception	SIGBUS
13	General protection	SIGSEGV
14	Page fault	SIGSEGV
15	Intel reserved	None
16	Floating point error	SIGFPE
17	Alignment check	SIGSEGV
18	Machine check	None
19	SIMD floating point	SIGFPE

中断描述符表

Interrupt Descriptor Table IDT 是一个系统表，它与每一个终端或异常向量相联系，每一个向量在表中有相应的中断或异常处理程序的入口地址。内核在允许中断前，必须适当地初始化 IDT。

IDT 包含三种类型的描述符：

- 任务门 task gate，当中断信号发生时，必须取代当前进程的那个进程的 TSS 选择符存放在任务门中。
- 中断门 interrupt gate，包含段选择符和中断或异常处理程序偏移量。当控制权转移到一个适当的段时，处理器清 IF 标志，从而关闭将来会发生的可屏蔽中断。
- 陷阱门 trap gate，与中断门相似，只是控制权传递到一个适当的段时处理器不修改 IF 标志。

Linux 利用中断门处理中断，利用陷阱门处理异常。

中断和异常的硬件处理

当执行一条指令后，cs 和 eip 寄存器包含下一条将要执行的指令的逻辑地址。在处理那条指令之前，控制单元会检查是否已经发生了一个中断或异常。如果发生了，那么：

- 确定与中断或异常关联的向量 i。
- 度由 idtr 寄存器指向的 IDT 表中的第 i 项。
- 从 gdtr 寄存器获得 GDT 的基地址，在 GDT 中查找，以读取 IDT 表项中的选择符所标识的段描述符。这个描述符指定中断或异常处理程序所在段的基地址。
- 确信中断是由授权的中断发生源发出的。
- 检查是否发生了特权级的变化。
- 如果故障已发生，用引起异常的指令地址装载 cs 和 eip 寄存器，从而使得这条指令能再次被执行。
- 在栈中保存 eflags、cs 及 eip 的内容。
- 如果异常发生了一个硬件出错码，则将它保存在栈中。
- 装载 cs 和 eip 寄存器。这些值给出了中断或异常处理程序的第一条指令的逻辑地址。

控制单元所执行的最后一步就是跳转到中断或异常处理程序。中断或异常被处理完后，相应的处理程序必须产生一条 iret 指令，把控制权交回给被中断的进程。处理单元将

- 用保存在栈中的 cs、eip 或 eflags 寄存器。如果一个硬件出错码层被压入栈中，并且在 eip 内容的上面，那么执行 iret 指令前必须先弹出这个硬件出错码。
- 检查处理程序的 CPL 是否等于 cs 中最低两位的值，判断被中断的进程与处理器程序是否运行在同一特权级。如果是，iret 终止执行；否则，转入下一步。
- 从栈中装载 ss 和 esp 寄存器，返回到与旧特权级相关的栈。
- 检查 ds、es、fs 及 gs 段寄存器的内容，防止恶意用户程序利用这些值来访问内核地址空间。

中断和异常处理程序的嵌套执行

允许嵌套执行必须付出代价，就是终端处理程序必须永不阻塞，即中断处理程序运行期间不能发生进程诶切换。因为嵌套的内核控制路径恢复执行时需要的所有数据都存放在内核态堆栈中，这个栈毫无疑义的属于当前进程。

一个中断处理程序既可以抢占其他的中断处理程序，也可以抢占异常处理程序。相反，异常处理程序从不抢占中断处理程序。

基于以下两个主要原因：**linux** 交错执行内核控制路径：

- 为了提高可编程中断控制器和设备控制器的吞吐量。假定设备控制器在一条 IRQ 线上产生了一个信号，PIC 把这个信号转换成一个外

部中断，然后 PIC 和设备控制器保持阻塞，一直到 PIC 从 CPU 处接收到一条应答信息。由于内核控制路径的交错执行，内核即使正在处理前一个中断，也能发送应答。

- 为了实现一种没有优先级的中断模型。

在多处理器上，几个内核控制路径可以并发执行。与异常相关的内核控制路径可以开始在一个 CPU 上执行，并且由于进程切换而转移到另一个 CPU 上执行。

初始化中段描述符表

内核启动中断前，必须把 IDT 标的初始地址装到 idtr 寄存器，并初始化表中的每一项。

中断门、陷阱门及系统门

Linux 使用与 Intel 稍有不同的细目分类和术语：

- 中断门 interrupt gate。Linux 中断处理程序都通过中断门激活，并全部限制在内核态。
- 系统门 system gate。 在用户态下，可以通过系统门来激活三个 Linux 异常处理程序，向量分别是 4，5 及 128。
- 系统中断门 system interrupt gate。用户态可以通过系统中断门来激活与向量 3 相关的异常处理程序。
- 陷阱门 trap gate。 大部分 Linux 异常处理程序都通过陷阱门来激活。
- 任务门 task gate。Linux 对 Double fault 异常的处理程序是由任务门激活的。

IDT 的初步初始化

IDT 存放在 idt_table 表中，有 256 个表项。在内核初始化过程中，被同一个中断门（指向 ignore_int 的中断处理程序）来填充这 256 个表项。该中断处理程序执行下列动作：

- 在栈中保存一些寄存器的内容。
- 调用 printk 函数打印 Unknown interrupt 系统消息。
- 从栈回复寄存器的内容。
- 执行 iret 指令以恢复被中断的程序。

该中断处理程序应该从不被执行，若出现了 Unknown interrupt 的打印，说明出现了一个硬件问题（IO 设备在产生没有预料的中断）或者内核问题（一个中断或异常未被合适地处理）。

紧接着，内核将在 IDT 中进行第二遍初始化，用有意义的陷阱和中断处理程序来替换这个空处理程序。

异常处理

CPU 产生的大部分异常都由 Linux 解释为出错条件。当其中一个异常发生时，内核就向引起异常的进程发送一个信号向它通知一个反常条件。

在两种情况下，Linux 利用 CPU 异常更有效地管理硬件资源。

- Device not availeble 异常与 cr0 寄存器的 TS 标志一起用来把新值装入浮点寄存器。
- Page Fault 异常，推迟给进程分配新的页框，直到不能再推迟为止。

异常处理程序有一个标志的结构：

- 在内核堆栈中保存大多数寄存器的内容。
- 用高级的 C 函数处理异常。
- 通过 ret_from_exception 函数从异常处理程序退出。

中断处理

中断处理依赖于中断类型，下面将讨论三种主要的中断类型：

- IO 中断
- 时钟中断
- 处理器间中断

IO 中断处理

中断处理程序的灵活性是以两种不同的方式实现的

- IRQ 共享。 中断处理程序执行多个终端服务例程 interrupt service routine ISR。每个 ISR 是一个与单独设备相关的函数。由于不能明确哪个特定的设备产生 IRQ，因此每个 ISR 都被执行，以验证它的设备是否需要关注，如果是，就执行需要执行的所有操作。
- IRQ 动态分配。一条 IRQ 线可以同很多设备驱动程序相关联。同一个 IRQ 向量可以由几个设备在不同时刻使用。

Linux 把紧随中断要执行的操作分为三类：

- 紧急的 critical。 如对 PIC 应答中断，对 PIC 或设备控制器重编程，修改由设备和处理器同时访问的**数据结构**。这些能被很快地执行。
- 非紧急的 Noncritical。如修改只有处理器才会访问的数据结构。这些操作也要很快地完成。
- 非紧急可延迟的 Noncritical deferrable。如把缓冲区的内容拷贝到某个进程的地址空间。这些可能被延时较长的时间。

Linux 中的中断向量

范围	用途
0-19	非屏蔽中断和异常
20-31	Intel 保留
32-127	外部中断
128	用于系统调用的可编程异常
129-238	外部中断
239	本地 APIC 时钟中断
240	本地 APIC 高温中断
241-250	由 Linux 留作将来使用
251-253	处理器间中断
254	本地 APIC 错误中断

范围	用途
255	本地 APIC 伪中断

kirqd 内核线程周期性地执行 do_irq_balance 函数，该函数跟踪在最近时间间隔内的每个 CPU 接收的中断次数，如果 CPU 间的 IRQ 负载不平衡很严重，就把 IRQ 从一个 CPU 转移到另一个 CPU。

保存寄存器是中断处理程序做的第一件事情。内核用负数表示所有的中断，正数用来表示系统调用。

软中断和 tasklet 有密切的关系，tasklet 是在软中断之上实现。软中断 softirq 常常表示可延迟函数的所有种类。
软中断的分配是静态的，在编译时定义；tasklet 的分配和初始化可以在运行时进行。
软中断是可重入函数，必须明确地使用自旋锁保护其数据结构。相同类型的 tasklet 总是被串行地执行。

tasklet 是 IO 驱动程序中实现可延迟函数的首选方法。tasklet 建立在两个软中断之上：HI_SOFTIRQ 和 TASKLET_SOFTIRQ。

可延迟函数运行在中断上下文，工作队列中的函数运行在进程上下文。工作队列中的函数是由内核线程来执行的。