Linux 中断解析

摘要:本章将向读者依次解释中断概念，解析 Linux 中的中断实现机理以及 Linux 下中断如何被使用。作为实例我们第一将向《i386 体系结构》一章中打造的系统加入一个时钟中断；第二将为大家注解 RTC 中断，希望通过这两个实例可以帮助读者掌握中断相关的概念、实现和编程方法。

中断是什么

中断的汉语解释是半中间发生阻隔、停顿或故障而断开。那么，在计算机系统中，我们为什么需要“阻隔、停顿和断开”呢？

举个日常生活中的例子，比如说我正在厨房用煤气烧一壶水，这样就只能守在厨房里，苦苦等着水开——如果水溢出来浇灭了煤气，有可能就要发生一场灾难了。等啊等啊，外边突然传来了惊奇的叫声“怎么不关水龙头？”于是我惭愧的发现，刚才接水之后只顾着抱怨这份无聊的差事，居然忘了这事，于是慌慌张张的冲向水管，三下两下关了龙头，声音又传到耳边，“怎么干什么都是这么马虎？”。伸伸舌头，这件小事就这么过去了，我落寞的眼神又落在了水壶上。

门外忽然又传来了铿锵有力的歌声，我最喜欢的古装剧要开演了，真想夺门而出，然而，听着水壶发出“咕嘟咕嘟”的声音，我清楚：除非等到水开，否则没有我享受人生的时候。

这个场景跟中断有什么关系呢？

如果说我专心致志等待水开是一个过程的话，那么叫声、电视里传出的音乐不都让这个过程“半中间发生阻隔、停顿或故障而断开”了吗？这不就是活生生的“中断”吗？

在这个场景中，我是唯一具有处理能力的主体，不管是烧水、关龙头还是看电视，同一个时间点上我只能干一件事情。但是，在我专心致志干一件事情时，总有许多或紧迫或不紧迫的事情突然出现在面前，都需要去关注，有些还需要我停下手头的工作马上去处理。只有在处理完之后，方能回头完成先前的任务，“把一壶水彻底烧开!”

中断机制不仅赋予了我处理意外情况的能力，如果我能充分发挥这个机制的妙用，就可以“同时”完成多个任务了。回到烧水的例子，实际上，无论我在不在厨房，煤气灶总是会把水烧开的，我要做的，只不过是及时关掉煤气灶而已，为了这么一个一秒钟就能完成的动作，却让我死死的守候在厨房里，在 10 分钟的时间里不停的看壶嘴是不是冒蒸汽，怎么说都不划算。我决定安下心来看电视。当然，在有生之年，我都不希望让厨房成为火海，于是我

上了闹钟，10 分钟以后它会发出“尖叫”，提醒我炉子上的水烧开了，那时我再去关煤气也完全来得及。我用一个中断信号——闹铃——换来了 10 分钟的欢乐时光，心里不禁由衷的感叹：中断机制真是个好东西。

正是由于中断机制，我才能有条不紊的“同时”完成多个任务，中断机制实质上帮助我提高了并发“处理”能力。它也能给计算机系统带来同样的好处：如果在键盘按下的时候会得到一个中断信号，CPU 就不必死守着等待键盘输入了；如果硬盘读写完成后发送一个中断信号，CPU 就可以腾出手来集中精力“服务大众”了——无论是人类敲打键盘的指尖还是来回读写介质的磁头，跟 CPU 的处理速度相比，都太慢了。没有中断机制，就像我们苦守厨房一样，计算机谈不上有什么的并行处理能力。

跟人相似，CPU 也一样要面对纷繁芜杂的局面——现实中的意外是无处不在的——有可能是用户等得不耐烦，猛敲键盘；有可能是运算中碰到了 0 除数；还有可能网卡突然接收到了一个新的数据包。这些都需要 CPU 具体情况具体分析，要么马上处理，要么暂缓响应，要么置之不理。无论如何应对，都需要 CPU 暂停“手头”的工作，拿出一种对策，只有在响应之后，方能回头完成先前的使命，“把一壶水彻底烧开！”

先让我们感受一下中断机制对并发处理带来的帮助。

让我们用程序来探讨一下烧水问题，如果没有“中断”（注意，我们这里只是模仿中断的场景，实际上是用异步事件——消息——处理机制来展示中断产生的效果。毕竟，在用户空间没有办法与实际中断产生直接联系，不过操作系统为用户空间提供的异步事件机制，可以看作是模仿中断的产物），设计如下：

```
void StayInKitchen()
{
bool WaterIsBoiled = false;
while ( WaterIsBoiled != true )
{
        bool VaporGavenOff    = false;
                if    (VaporGavenOff )
                        WaterIsBoiled    = true;
else
                WaterIsBoiled    = false;
}
```

```
// 关煤气炉
printf(“Close gas oven.\n”);
// 一切安定下来，终于可以看电视了，10 分钟的宝贵时间啊，逝者如斯夫…
watching_tv();
return;
}
```

可以看出，整个流程如同我们前面描述的一样，所有工作要顺序执行，没有办法完成并发任务。

如果用“中断”，在开始烧水的时候设定一个 10 分钟的“闹铃”，然后让 CPU 去看电视（有点难度，具体实现不在我们关心的范围之内，留给读者自行解决吧：>）。等闹钟响的时候再去厨房关炉子。

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <signal.h>
#include <stdio.h>
// 闹钟到时会执行此程序
void sig_alarm(int signo)
{
        //关煤气炉
        printf(“Close gas oven.\n”);
}
void watching_tv()
{
        while(1)
        {
                // 呵呵，优哉游哉
}
}
int main()
```

```
{
// 点火后设置定时中断
   printf(“Start to boil water, set Alarm”);
if (signal( SIGALRM, sig_alrm ) == SIG_ERR)
{
      perror("signal(SIGALRM) error");
      return -1;
   }
// 然后就可以欣赏电视节目了
   printf(“Watching TV!\n”);
watching_tv();
return 0;
}
```

这两段程序都在用户空间执行。第二段程序跟中断也没有太大的关系，实际上它只用了信号机制而已。但是，通过这两个程序的对比，我们可以清楚地看到异步的事件处理机制是如何提升并发处理能力的。

Alarm 定时器：alarm 相当于系统中的一个定时器，如果我们调用 alarm(5)，那么 5 秒钟后就会“响起一个闹铃”（实际上靠信号机制实现的，我们这里不想深入细节，如果你对此很感兴趣，请参考 Richard Stevens 不朽著作《Unix 环境高级编程》）。在闹铃响起的时候会发生什么呢?系统会执行一个函数，至于到底是什么函数，系统允许程序自行决定。程序员编写一个函数，并调用 signal 对该函数进行注册，这样一旦定时到来，系统就会调用程序员提供的函数（CallBack 函数？没错，不过在这里如何实现并不关键，我们就不引入新的概念和细节了）。上面的例子里我们提供的函数是 sig_alarm，所做的工作很简单，打印“关闭煤气灶”消息。

上面的两个例子很简单，但很能说明问题，首先，它证明采用异步的消息处理机制可以提高系统的并发处理能力。更重要的是，它揭示了这种处理机制的模式。用户根据需要设计处理程序，并可以将该程序和特定的外部事件绑定起来，在外部事件发生时系统自动调用处理程序，完成相关的工作。这种模式给系统带来了统一的管理方法，也带来无尽的功能扩展空间。

计算机系统实现中断机制是非常复杂的一件工作，再怎么说人都是高度智能化的生物，

而计算机作为一个铁疙瘩，没有程序的教导就一事无成。而处理一个中断过程，它受到的限制和需要学习的东西太多了。

首先，计算机能够接收的外部信号形式非常有限。中断是由外部的输入引起的，可以说是一种刺激。在烧水的场景中，这些输入是叫声和电视的音乐，我们这里只以声音为例。其实现实世界中能输入人类 CPU——大脑的信号很多，图像、气味一样能被我们接受，人的信息接口很完善。而计算机则不然，接受外部信号的途径越多，设计实现就越复杂，代价就越高。因此个人计算机（PC）给所有的外部刺激只留了一种输入方式——特定格式的电信号，并对这种信号的格式、接入方法、响应方法、处理步骤都做了规约（具体内容本文后面部分会继续详解），这种信号就是中断或中断信号，而这一整套机制就是中断机制。

其次，计算机不懂得如何应对信号。人类的大脑可以自行处理外部输入，我从来不用去担心闹钟响时会手足无措——走进厨房关煤气，这简直是天经地义的事情，还用大脑想啊，小腿肚子都知道——可惜计算机不行，没有程序，它就纹丝不动。因此，必须有机制保证外部中断信号到来后，有正确的程序在正确的时候被执行。

还有，计算机不懂得如何保持工作的持续性。我在看电视的时候如果去厨房关了煤气，回来以后能继续将电视进行到底，不受太大的影响。而计算机则不然，如果放下手头的工作直接去处理“意外”的中断，那么它就再也没有办法想起来曾经作过什么，做到什么程度了。自然也就没有什么“重操旧业”的机会了。这样的处理方式就不是并发执行，而是东一榔头，西一棒槌了。

那么，通用的计算机系统是如何解决这些问题的呢？它是靠硬件和软件配合来协同实现中断处理的全过程的。我们将通过 Intel X86 架构的实现来介绍这一过程。

中断流程处理

CPU 执行完一条指令后，下一条指令的逻辑地址存放在 cs 和 eip 这对寄存器中。在执行新指令前，控制单元会检查在执行前一条指令的过程中是否有中断或异常发生。如果有，控制单元就会抛下指令，进入下面的流程：

1）. 确定与中断或异常关联的向量 i (0≤i≤255)

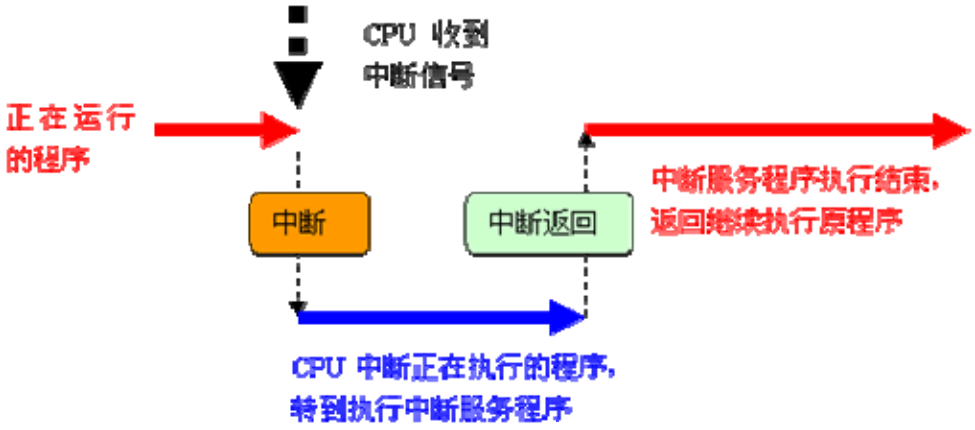
2）. 寻找向量对应的处理程序

3）. 保存当前的“工作现场”，执行中断或异常的处理程序

4）. 处理程序执行完毕后，把控制权交还给控制单元

5）. 控制单元恢复现场，返回继续执行原程序

整个流程如下图所示：



图一：中断处理过程

让我们深入这个流程，看看都有什么问题需要面对。

1、异常是什么概念？

在处理器执行到由于编程失误而导致的错误指令（例如除数是 0）的时候，或者在执行期间出现特殊情况（例如缺页），需要靠操作系统来处理的时候，处理器就会产生一个异常。对大部分处理器体系结构来说，处理异常和处理中断的方式基本是相同的，x86 架构的 CPU 也是如此。异常与中断还是有些区别，异常的产生必须考虑与处理器时钟的同步。实际上，异常往往被称为同步中断。

2、中断向量是什么？

中断向量代表的是中断源——从某种程度上讲，可以看作是中断或异常的类型。中断和异常的种类很多，比如说被 0 除是一种异常，缺页又是一种异常，网卡会产生中断，声卡也会产生中断，CPU 如何区分它们呢？中断向量的概念就是由此引出的，其实它就是一个被送通往 CPU 数据线的一个整数。CPU 给每个 IRQ 分配了一个类型号，通过这个整数 CPU 来识别不同类型的中断。这里可能很多朋友会寻问为什么还要弄个中断向量这么麻烦的东西？为什么不直接用 IRQ0~IRQ15 就完了？比如就让 IRQ0 为 0，IRQ1 为 1……，这不是要简单的多么？其实这里体现了模块化设计规则，及节约规则。

首先我们先谈谈节约规则，所谓节约规则就是所使用的信号线数越少越好，这样如果每个 IRQ 都独立使用一根数据线，如 IRQ0 用 0 号线，IRQ1 用 1 号线……这样，16 个 IRQ 就会用 16 根线，这显然是一种浪费。那么也许马上就有朋友会说：那么只用 4 根线不就行了吗？（2^4=16）。

这个问题，体现了模块设计规则。我们在前面就说过中断有很多类，可能是外部硬件触

发，也可能是由软件触发，然而对于 CPU 来说中断就是中断，只有一种，CPU 不用管它到底是由外部硬件触发的还是由运行的软件本身触发的，应为对于 CPU 来说，中断处理的过程都是一样的：中断现行程序，转到中断服务程序处执行，回到被中断的程序继续执行。CPU 总共可以处理 256 种中断，而并不知道，也不应当让 CPU 知道这是硬件来的中断还是软件来的中断，这样，就可以使 CPU 的设计独立于中断控制器的设计，这样 CPU 所需完成的工作就很单纯了。CPU 对于其它的模块只提供了一种接口，这就是 256 个中断处理向量，也称为中断号。由这些中断控制器自行去使用这 256 个中断号中的一个与 CPU 进行交互，比如，硬件中断可以使用前 128 个号，软件中断使用后 128 个号，也可以软件中断使用前 128 个号，硬件中断使用后 128 个号，这与 CPU 完全无关了，当你需要处理的时候，只需告诉 CPU 你用的是哪个中断号就行，而不需告诉 CPU 你是来自哪儿的中断。这样也方便了以后的扩充，比如现在机器里又加了一片 8259 芯片，那么这个芯片就可以使用空闲的中断号，看哪一个空闲就使用哪一个，而不是必须要使用第 0 号，或第 1 号中断号了。其实这相当于一种映射机制，把 IRQ 信号映射到不同的中断号上，IRQ 的排列或说编号是固定的，但通过改变映射机制，就可以让 IRQ 映射到不同的中断号，也可以说调用不同的中断服务程序。

3、什么是中断服务程序？

在响应一个特定中断的时候，内核会执行一个函数，该函数叫做中断处理程序（interrupt handler）或中断服务程序（interrupt service routine(ISR)）。产生中断的每个设备都有相应的中断处理程序。例如，由一个函数专门处理来自系统时钟的中断，而另外一个函数专门处理由键盘产生的中断。

一般来说，中断服务程序要负责与硬件进行交互，告诉该设备中断已被接收。此外，还需要完成其他相关工作。比如说网络设备的中断服务程序除了要对硬件应答，还要把来自硬件的网络数据包拷贝到内存，对其进行处理后再交给合适的协议栈或应用程序。每个中断服务程序根据其要完成的任务，复杂程度各不相同。

一般来说，一个设备的中断服务程序是它设备驱动程序（device driver）的一部分——设备驱动程序是用于对设备进行管理的内核代码。

4、隔离变化

不知道您有没有意识到，中断处理前面这部分的设计是何等的简单优美。人是高度智能化的，能够对遇到的各种意外情况做有针对性的处理，计算机相比就差距甚远了，它只能根据预定的程序进行操作。对于计算机来说，硬件支持的，只能是中断这种电信号传播的方式和 CPU 对这种信号的接收方法，而具体如何处理这个中断，必须得靠操作系统实现。操作系

统支持所有事先能够预料到的中断信号，理论上都不存在太大的挑战，但在操作系统安装到计算机设备上以后，肯定会时常有新的外围设备被加入系统，这可能会带来安装系统时根本无法预料的“意外”中断。如何支持这种扩展，是整个系统必须面对的。

而硬件和软件在这里的协作，给我们带来了完美的答案。当新的设备引入新类型的中断时，CPU 和操作系统不用关注如何处理它。CPU 只负责接收中断信号，并引用中断服务程序；而操作系统提供默认的中断服务——一般来说就是不理会这个信号，返回就可以了——并负责提供接口，让用户通过该接口注册根据设备具体功能而编制的中断服务程序。如果用户注册了对应于一个中断的服务程序，那么 CPU 就会在该中断到来时调用用户注册的服务程序。这样，在中断来临时系统需要如何操作硬件、如何实现硬件功能这部分工作就完全独立于 CPU 架构和操作系统的设计了。

而当你需要加入新设备的时候，只需要告诉操作系统该设备占用的中断号、按照操作系统要求的接口格式撰写中断服务程序，用操作系统提供的函数注册该服务程序，设备的中断就被系统支持了。

中断和对中断的处理被解除了耦合。这样，无论是你在需要加入新的中断时，还是在你需要改变现有中断的服务程序时、又或是取消对某个中断支持的时候，CPU 架构和操作系统都无需作改变。

5、保存当前工作“现场”

在中断处理完毕后，计算机一般来说还要回头处理原先手头正做的工作。这给中断的概念带来些额外的“内涵”[1]。“回头”不是指从头再来重新做，而是要接着刚才的进度继续做。这就需要在处理中断信号之前保留工作“现场”。“现场”这个词比较晦涩，其实就是指一个信息集，它能反映某个时间点上任务的状态，并能保证按照这些信息就能恢复任务到该状态，继续执行下去。再直白一点，现场不过就是一组寄存器值。而如何保护现场和恢复场景是中断机制需要考虑的重点之一。

每个中断处理都要经历这个保存和恢复过程，我们可以抽象出其中的步骤：

1）.　　保存现场

2）.　　执行具体的中断服务程序

3）.　　从中断服务返回

4）.　　恢复现场

上面说过了，“现场”看似在不断变化，没有哪个瞬间相同。但实际上组成现场的要素却不会有任何改变。也就是说，这要我们保存了相关的寄存器状态，现场就能保存下来。而恢

复“现场”就是重新载入这些寄存器。换句话说，对于任何一个中断，保护现场和恢复现场所作的都是完全相同的操作。

既然操作相同，实现操作的过程和代码就相同。减少代码的冗余是模块化设计的基本准则，实在没有道理让所有的中断服务程序都重复的实现这样的功能，应该将它作为一种基本的结构由底层的操作系统或硬件完成。而对中断的处理过程需要迅速完成，因此，Intel CPU 的控制器就承担了这个任务，非但如此，上面的所有步骤次序都被固化下来，由控制器驱动完成。保存现场和恢复现场都由硬件自动完成，大大减轻了操作系统和设备驱动程序的负担。

6、硬件对中断支持的细节

下面的部分，本来应该介绍 8259、中断控制器编程、中断描述符表等内容，可是我看到了潇寒写的“保护模式下的 8259A 芯片编程及中断处理探究”，前人之述备矣，读者直接读它好了。

Linux 下的中断

在 Linux 中，中断处理程序看起来就是普普通通的 C 函数。只不过这些函数必须按照特定的类型声明，以便内核能够以标准的方式传递处理程序的信息，在其他方面，它们与一般的函数看起来别无二致。中断处理程序与其它内核函数的真正区别在于，中断处理程序是被内核调用来响应中断的，而它们运行于我们称之为中断上下文的特殊上下文中。关于中断上下文，我们将在后面讨论。

中断可能随时发生，因此中断处理程序也就随时可能执行。所以必须保证中断处理程序能够快速执行，这样才能保证尽可能快地恢复被中断代码的执行。因此，尽管对硬件而言，迅速对其中断进行服务非常重要。但对系统的其它部分而言，让中断处理程序在尽可能短的时间内完成执行也同样重要。

即使最精简版的中断服务程序，它也要与硬件进行交互，告诉该设备中断已被接收。但通常我们不能像这样给中断服务程序随意减负，相反，我们要靠它完成大量的其它工作。作为一个例子，我们可以考虑一下网络设备的中断处理程序面临的挑战。该处理程序除了要对硬件应答，还要把来自硬件的网络数据包拷贝到内存，对其进行处理后再交给合适的协议栈或应用程序。显而易见，这种运动量不会太小。

现在我们来分析一下 Linux 操作系统为了支持中断机制，具体都需要做些什么工作。

首先，操作系统必须保证新的中断能够被支持。计算机系统硬件留给外设的是一个统一的中断信号接口。它固化了中断信号的接入和传递方法，拿 PC 机来说，中断机制是靠两块

8259 和 CPU 协作实现的。外设要做的只是把中断信号发送到 8259 的某个特定引脚上，这样 8259 就会为此中断分配一个标识——也就是通常所说的中断向量，通过中断向量，CPU 就能够在以中断向量为索引的表——中断向量表——里找到中断服务程序，由它决定具体如何处理中断。这是硬件规定的机制，软件只能无条件服从。

因此，操作系统对新中断的支持，说简单点，就是维护中断向量表。新的外围设备加入系统，首先得明确自己的中断向量号是多少，还得提供自身中断的服务程序，然后利用 Linux 的内核调用界面，把〈中断向量号、中断服务程序〉这对信息填写到中断向量表中去。这样 CPU 在接收到中断信号时就会自动调用中断服务程序了。这种注册操作一般是由设备驱动程序完成的。

其次，操作系统必须提供给程序员简单可靠的编程界面来支持中断。中断的基本流程前面已经讲了，它会打断当前正在进行的工作去执行中断服务程序，然后再回到先前的任务继续执行。这中间有大量需要解决问题：如何保护现场、嵌套中断如何处理等等，操作系统要一一化解。程序员，即使是驱动程序的开发人员，在写中断服务程序的时候也很少需要对被打断的进程心存怜悯。（当然，出于提高系统效率的考虑，编写驱动程序要比编写用户级程序多一些条条框框，谁让我们顶着系统程序员的光环呢？）

操作系统为我们屏蔽了这些与中断相关硬件机制打交道的细节，提供了一套精简的接口，让我们用极为简单的方式实现对实际中断的支持，Linux 是怎么完美的做到这一点的呢？

CPU 对中断处理的流程

我们首先必须了解 CPU 在接收到中断信号时会做什么。没办法，操作系统必须了解硬件的机制，不配合硬件就寸步难行。现在我们假定内核已被初始化，CPU 在保护模式下运行。

CPU 执行完一条指令后，下一条指令的逻辑地址存放在 cs 和 eip 这对寄存器中。在执行新指令前，控制单元会检查在执行前一条指令的过程中是否有中断或异常发生。如果有，控制单元就会抛下指令，进入下面的流程：

1.确定与中断或异常关联的向量 i (0≤i≤255)。

2.籍由 idtr 寄存器从 IDT 表中读取第 i 项（在下面的描述中，我们假定该 IDT 表项中包含的是一个中断门或一个陷阱门）。

3.从 gdtr 寄存器获得 GDT 的基地址，并在 GDT 表中查找，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。

4.确信中断是由授权的（中断）发生源发出的。首先将当前特权级 CPL（存放在 cs 寄存器的低两位）与段描述符（存放在 GDT 中）的描述符特权级 DPL 比较，如果 CPL 小于 DPL，

就产生一个“通用保护”异常，因为中断处理程序的特权不能低于引起中断的程序的特权。对于编程异常，则做进一步的安全检查：比较 CPL 与处于 IDT 中的门描述符的 DPL，如果 DPL 小于 CPL，就产生一个“通用保护”异常。这最后一个检查可以避免用户应用程序访问特殊的陷阱门或中断门。

5.检查是否发生了特权级的变化，也就是说，CPL 是否不同于所选择的段描述符的 DPL。如果是，控制单元必须开始使用与新的特权级相关的栈。通过执行以下步骤来做到这点：

a.读 tr 寄存器，以访问运行进程的 TSS 段。

b.用与新特权级相关的栈段和栈指针的正确值装载 ss 和 esp 寄存器。这些值可以在 TSS 中找到（参见第三章的“任务状态段”一节）。

c.在新的栈中保存 ss 和 esp 以前的值，这些值定义了与旧特权级相关的栈的逻辑地址。

6.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。

7.在栈中保存 eflag、cs 及 eip 的内容。

8.如果异常产生了一个硬错误码，则将它保存在栈中。

9.装载 cs 和 eip 寄存器，其值分别是 IDT 表中第 i 项门描述符的段选择符和偏移量域。这些值给出了中断或者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者异常处理程序。换句话说，处理完中断信号后, 控制单元所执行的指令就是被选中处理程序的第一条指令。

中断或异常被处理完后，相应的处理程序必须 0x20/0x21/0xa0/0xa1

产生一条 iret 指令，把控制权转交给被中断的进程，这将迫使控制单元：

1.用保存在栈中的值装载 cs、eip、或 eflag 寄存器。如果一个硬错误码曾被压入栈中，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错误码。

2.检查处理程序的 CPL 是否等于 cs 中最低两位的值（这意味着被中断的进程与处理程序运行在同一特权级）。如果是，iret 终止执行；否则，转入下一步。

3. 从栈中装载 ss 和 esp 寄存器，因此，返回到与旧特权级相关的栈。

4. 检查 ds、es、fs 及 gs 段寄存器的内容，如果其中一个寄存器包含的选择符是一个段描述符，并且其 DPL 值小于 CPL，那么，清相应的段寄存器。控制单元这么做是为了禁止用户态的程序（CPL=3）利用内核以前所用的段寄存器（DPL=0）。如果不清这些寄存器，怀有恶意的用户程序就可能利用它们来访问内核地址空间。

再次，操作系统必须保证中断信息能够高效可靠的传递

实例一——为自己的操作系统中加入中断

中断机制的实现

在这个部分，我将为大家详细介绍 SagaLinux_irq 中是如何处理中断的。为了更好的演示软硬件交互实现中断机制的过程，我将在前期实现的 SagaLinux 上加入对一个新中断-——定时中断——的支持。

首先，让我介绍一下 SagaLinux_irq 中涉及中断的各部分代码。这些代码主要包含在 kernel 目录下，包括 idt.c，irq.c，i8259.s，boot 目录下的 setup.s 也和中断相关，下面将对他们进行讨论。

1、boot/setup.s

setup.s 中相关于中断的部分主要集中在 pic_init 小结，该部分完成了对中断控制器的初始化。对 8259A 的编程是通过向其相应的端口发送一系列的 ICW（初始化命令字）完成的。总共需要发送四个 ICW，它们都分别有自己独特的格式，而且必须按次序发送，并且必须发送到相应的端口，具体细节请查阅相关资料。

```
pic_init:
    cli
    mov al, 0x11                ; initialize PICs
; 给中断寄存器编程
; 发送 ICW1:使用 ICW4，级联工作
    out 0x20, al                ; 8259_MASTER
    out 0xA0, al                ; 8259_SLAVE
    ; 发送 ICW2，中断起始号从 0x20 开始（第一片）及 0x28 开始（第二片）
    mov al,     0x20            ; interrupt start 32
    out 0x21, al
    mov al,     0x28            ; interrupt start 40
    out 0xA1, al
; 发送 ICW3
    mov al,     0x04            ; IRQ 2 of 8259_MASTER
    out 0x21, al
; 发送 ICW4
```

```
        mov al,     0x02                ; to 8259_SLAVE
        out 0xA1, al
; 工作在 80x86 架构下
        mov al,     0x01                ; 8086 Mode
        out 0x21, al
        out 0xA1, al
; 设置中断屏蔽位 OCW1 ，屏蔽所有中断请求
        mov al,     0xFF                ; mask all
        out 0x21, al
        out 0xA1, al
        sti
```

2、kernel/irq.c

irq.c 提供了三个函数 enable_irq、disable_irq 和 request_irq，函数原型如下：

```
void enable_irq(int irq)
void disable_irq(int irq)
void request_irq(int irq, void (*handler)())
```

enable_irq 和 disable_irq 用来开启和关闭右参数 irq 指定的中断，这两个函数直接对 8259 的寄存器进行操作，因此 irq 对应的是实实在在的中断号，比如说 X86 下时钟中断一般为 0 号中断，那么启动时钟中断就需要调用 enable_irq（1），而键盘一般占用 2 号中断，那么关闭键盘中断就需要调用 disable_irq（2）。irq 对应的不是中断向量。

request_irq 用来将中断号和中断服务程序绑定起来，绑定完成后，命令 8259 开始接受中断请求。下面是 request_irq 的实现代码：

```
void request_irq(int irq, void (*handler)())
{
   irq_handler[irq] = handler;
   enable_irq(irq);
}
```

其中 irq_handler 是一个拥有 16 个元素的数组，数组项是指向函数的指针，每个指针可以指向一个中断服务程序。irq_handler[irq] = handler 就是一个给数组项赋值的过程，其中隐藏了中断号向中断向量映射的过程，在初始化 IDT 表的部分，我会介绍相关内容。

3、kernel/i8259.s[2]

i8259.c 负责对外部中断的支持。我们已经讨论过了，8259 芯片负责接收外部设备——如定时器、键盘、声卡等——的中断，两块 8259 共支持 16 个中断。

我们也曾讨论过，在编写操作系统的时候，我们不可能知道每个中断到底对应的是哪个中断服务程序。实际上，通常在这个时候，中断服务程序压根还没有被编写出来。可是，X86 体系规定，在初始化中断向量表的时候，必须提供每个向量对应的服务程序的偏移地址，以便 CPU 在接收到中断时调用相应的服务程序，这该如何是好呢？

巧妇难为无米之炊，此时此刻，我们只有创造所有中断对应的服务程序，才能完成初始化 IDT 的工作，于是我们制造出 16 个函数——__irq0 到__irq15，在注册中断服务程序的时候，我们就把它们填写到 IDT 的描述符中去。（在 SagaLinux 中当前的实现里，我并没有填写完整的 IDT 表，为了让读者看得较为清楚，我只加入了定时器和键盘对应的__irq 和__irq1。但这样一来就带来一个恶果，读者会发现在加入新的中断支持时，需要改动 idt.c 中的 trap_init 函数，用 set_int_gate 对新中断进行支持。完全背离了我们强调的分隔变化的原则。实际上，只要我们在这里填写完整，并提供一个缺省的中断服务函数就可以解决这个问题。我再强调一遍，这不是设计问题，只是为了便于读者观察而做的简化。）

可是，这 16 个函数怎么能对未知的中断进行有针对性的个性化服务呢？当然不能，这 16 个函数只是一个接口，我们可以在其中留下后门，当新的中断需要被系统支持时，它实际的中断服务程序就能被这些函数调用。具体调用关系请参考图 2

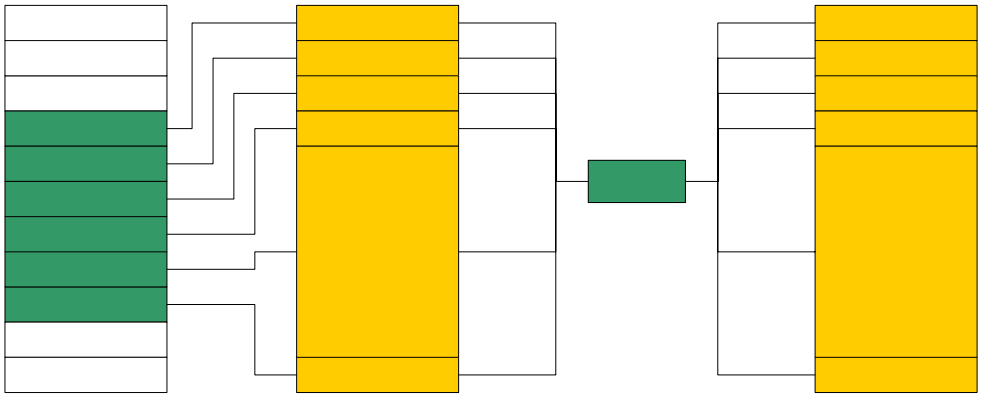


图 2：中断服务程序调用关系

如图 2 所示，__irq0 到__irq15 会被填充到 IDT 从 32 到 47（之所以映射到这个区间是为了模仿 Linux 的做法，其实这部分的整个实现都是在模仿 Linux）这 16 个条目的中断描述符中去，这样中断到来的时候就会调用相应的__irq 函数。所有 irq 函数所作的工作基本相同，

把中断号压入栈中，再调用 do_irq 函数；它们之间唯一区别的地方就在于不同的 irq 函数压入的中断号不同。

do_irq 首先会从栈中取出中断号，然后根据中断号计算该中断对应的中断服务程序在 irq_handler 数组中的位置，并跳到该位置上去执行相应的服务程序。

还记得 irq.c 中介绍的 request_irq 函数吗，该函数绑定中断号和中断服务程序的实现，其实就是把指向中断服务程序的指针填写到中断号对应的 irq_handler 数组中去。现在，你应该明白我们是怎样把一个中断服务程序加入到 SagaLinux 中的了吧——通过一个中间层，我们可以做任何事情。

在上图的实现中，IDT 表格中墨绿色的部分——外部中断对应的部分——可以浮动，也就是说，我们可以任意选择映射的起始位置，比如说，我们让__irq0 映射到 IDT 的第 128 项，只要后续的映射保持连续就可以了。

4、kernel/idt.c

idt.c 当然是用来初始化 IDT 表的了。

在 i8259.s 中我们介绍了操作系统是如何支持中断服务程序的添加的，但是，有两个部分的内容没有涉及：一是如何把__irq 函数填写到 IDT 表中，另外一个就是中断支持了，那异常怎么支持呢？idt.c 负责解决这两方面的问题。

idt.c 提供了 trap_init 函数来填充 IDT 表。

```
        void trap_init()
{
   int i;
   idtr_t idtr;
   // 填入系统默认的异常，共 17 个
   set_trap_gate(0, (unsigned int)&divide_error);
   set_trap_gate(1, (unsigned int)&debug);
   set_trap_gate(2, (unsigned int)&nmi);
   set_trap_gate(3, (unsigned int)&int3);
   set_trap_gate(4, (unsigned int)&overflow);
   set_trap_gate(5, (unsigned int)&bounds);
   set_trap_gate(6, (unsigned int)&invalid_op);
   set_trap_gate(7, (unsigned int)&device_not_available);
```

```
    set_trap_gate(8, (unsigned int)&double_fault);
    set_trap_gate(9, (unsigned int)&coprocessor_segment_overrun);
    set_trap_gate(10,(unsigned int) &invalid_TSS);
    set_trap_gate(11, (unsigned int)&segment_not_present);
    set_trap_gate(12, (unsigned int)&stack_segment);
    set_trap_gate(13, (unsigned int)&general_protection);
    set_trap_gate(14, (unsigned int)&page_fault);
    set_trap_gate(15, (unsigned int)&coprocessor_error);
    set_trap_gate(16, (unsigned int)&alignment_check);
    // 17 到 31 这 15 个异常是 intel 保留的，最好不要占用
    for (i = 17;i<32;i++)
        set_trap_gate(i, (unsigned int)&reserved);
    //  我们只在 IDT 中填入定时器和键盘要用到的两个中断
    set_int_gate(32, (unsigned int)&__irq0);
    set_int_gate(33, (unsigned int)&__irq1);

  //  一共有 34 个中断和异常需要支持
    idtr.limit = 34*8;
    idtr.lowerbase = 0x0000;
    idtr.higherbase = 0x0000;
    cli();
//  载入 IDT 表，新的中断可以用了
    __asm__ __volatile__ ("lidt (%0)"
                    ::"p" (&idtr));
    sti();
}
```

首先我们来看看 set_trap_gate 和 set_int_gate 函数，下面是它们两个的实现

```
void set_trap_gate(int vector, unsigned int handler_offset)
{
    trapgd_t* trapgd = (trapgd_t*) IDT_BASE + vector;
```

```
    trapgd->loffset = handler_offset & 0x0000FFFF;
    trapgd->segment_s = CODESEGMENT;
    trapgd->reserved = 0x00;
    trapgd->options =    0x0F | PRESENT | KERNEL_LEVEL;
    trapgd->hoffset = ((handler_offset & 0xFFFF0000) >> 16);
}
void set_int_gate(int vector,    unsigned int handler_offset)
{
    intgd_t* intgd = (intgd_t*) IDT_BASE + vector;
    intgd->loffset =    handler_offset & 0x0000FFFF;
    intgd->segment_s = CODESEGMENT;
    intgd->reserved = 0x0;
    intgd->options =    0x0E | PRESENT | KERNEL_LEVEL;
    intgd->hoffset = ((handler_offset & 0xFFFF0000) >> 16);
}
```

我们可以发现，它们所作的工作就是根据中断向量号计算出应该把指向中断或异常服务程序的指针放在什么 IDT 表中的什么位置，然后把该指针和中断描述符设置好就行了。同样，中断描述符的格式请查阅有关资料。

现在，来关注一下 set_trap_gate 的参数，又是指向函数的指针。在这里，我们看到每个这样的指针指向一个异常处理函数，如 divide_error、debug 等：

```
void divide_error(void)
{
    sleep("divide error");
}
void debug(void)
{
    sleep("debug");
}
```

每个函数都调用了 sleep，那么 sleep 是有何作用？是不是像——do_irq 一样调用具体异常的中断服务函数呢？

```
// Nooooo ... just sleep :)
void sleep(char* message)
{
   printk("%s",message);
   while(1);
}
```

看样子不是，这个函数就是休眠而已！实际上，我们这里进行了简化，对于 Intel 定义好的前 17 个内部异常，目前 SagaLinux 还不能做有针对性的处理，因此我们直接让系统无限制地进入休眠——跟死机区别不大。因此，当然也不用担心恢复“现场”的问题了，不用考虑栈的影响，所以直接用 C 函数实现。

此外，由于这 17 个异常如何处理在这个时候我们已经确定下来了——sleep，既然没有什么变化，我们也就不用耗尽心思的考虑去如何支持变化了，直接把函数硬编码就可以了。

Intel 规定中断描述符表的第 17-31 项保留，为硬件将来可能的扩展用，因此我们这里将它闲置起来。

```
void reserved(void)
{
sleep("reserved");
}
```

下面的部分是对外部中断的初始化，放在 trap_init 中是否有些名不正言不顺呢？确实如此，这个版本暂时把它放在这里，以后重构的时候再调整吧。注意，这个部分解释了我们是如何把中断服务程序放置到 IDT 中的。此外，可以看出，我们使用手工方式对中断向量号进行了映射，__irq0 对应 32 号中断，而__irq1 对应 33 号中断。能不能映射成别的向量呢？当然可以，可是别忘了修改 setup.s 中的 pic_init 部分，要知道，我们初始化 8259 的时候定义好了外部中断对应的向量，如果你希望从 8259 发来的中断信号能正确的触发相应的中断服务程序，当然要把所有的接收——处理链条上的每个映射关系都改过来。

我们只填充了 34 个表项，每个表项 8 字节长，因此我们把 IDT 表的长度上限设为 34x8，把 IDT 表放置在逻辑地址起始的地方（如果我们没有启用分页机制，那么就是在线性空间起始的地方，也就是物理地址的 0 位置处）。

最后，调用 ldtr 指令启用新的中断处理机制，SagaLinux 的初步中断支持机制就完成了。

扩展新的中断

下面，我们以定时器（timer）设备为例，展示如何通过 SagaLinux 目前提供的中断服务程序接口来支持设备的中断。

IBM PC 兼容机包含了一种时间测量设备，叫做可编程间隔定时器（PIT）。PIT 的作用类似于闹钟，在设定的时间点到来的时候发出中断信号。这种中断叫做定时中断（timer interrupt）。在 Linux 操作系统中，就是它来通知内核又一个时间片断过去了。与闹钟不同，PIT 以某一固定的频率（编程控制）不停地发出中断。每个 IBM PC 兼容机至少都会包含一个 PIT，一般来说，它就是一个使用 0x40~0x43 I/O 端口的 8254CMOS 芯片。

SagaLinux 目前的版本还不支持进程调度，因此定时中断的作用还不明显，不过，作为一个做常见的中断源，我们可以让它每隔一定时间发送一个中断信号，而我们在定时中断的中断服务程序中计算流逝过去的时间数，然后打印出结果，充分体现中断的效果。

我们在 kernel 目录下编写了 timer.c 文件，也在 include 目录下加入了相应的 timer.h，下面就是具体的实现。

```
// 流逝的时间
static volatile ulong_t counter;
        // 中断服务程序
void timer_handler()
{
   // 中断每 10 毫秒一次
   counter += 10;
}
// 初始化硬件和技术器，启用中断
void timer_init()
{
   ushort_t pit_counter = CLOCK_RATE * INTERVAL / SECOND;
   counter = 0;
   outb (SEL_CNTR0|RW_LSB_MSB|MODE2|BINARY_STYLE, CONTROL_REG);
   outb (pit_counter & 0xFF, COUNTER0_REG);
   outb (pit_counter >> 8, COUNTER0_REG);
   // 申请 0 号中断，TIMER 定义为 0
   request_irq(TIMER, timer_handler);
```

```
}
// 返回流逝过去的时间
ulong_t uptime()
{
   return counter;
}
```

timer_init 函数是核心函数，负责硬件的初始化和中断的申请，对 8254 的初始化就不多做纠缠了，请查阅有关资料。我们可以看到，申请中断确实跟预想中的一样容易，调用 request_irq，一行语句就完成了中断的注册。

而中断服务程序非常简单，由于把 8254 设置为每 10 毫秒发送一次中断，因此每次中断到来时都在服务程序中对 counter 加 10，所以 counter 表示的就是流逝的时间。

在 kernel.c 中，我们调用 timer_init 进行初始化，此时定时中断就被激活了，如果我们的中断机制运转顺利，那么流逝时间会不断增加。为了显示出这样的结果，我们编写一个循环不断的调 uptime 函数，并把返回的结果打印在屏幕上。如果打印出的数值越来越大，那就说明我们的中断机制确确实实发挥了作用，定时中断被驱动起来了。

在 kernel.c 中:

```
// 初始化
int i = 0;
timer_init();
i = uptime();
   while(1)
        {
      int temp = uptime();
      // 发生变化才打印，否则看不清楚
      if (temp != i)
   {
                  printk(" %d ", temp);
         i = temp;
      }
```

当 SagaLinux_irq 引导后，你会发现屏幕上开始不停的打印逐渐增大的数字，系统对定时

中断的支持，确实成功了。

为了验证中断支持的一般性，我们又加入了对键盘的支持。这样还可以充分体现中断对并发执行任务带来的帮助，在你按下键盘的时候，定时中断依然不断触发，屏幕上会打印出时间，当然，也会打印出你按下的字符。不过，这里就不对此做进一步描述了。

实例二——从 RTC 设备学习中断

系统实时钟

每台 PC 机都有一个实时钟（Real Time Clock）设备。在你关闭计算机电源的时候，由它维持系统的日期和时间信息。

此外，它还可以用来产生周期信号，频率变化范围从 2Hz 到 8192Hz——当然，频率必须是 2 的倍数。这样该设备就能被当作一个定时器使用，比如我们把频率设定为 4Hz，那么设备启动后，系统实时钟每秒就会向 CPU 发送 4 次定时信号——通过 8 号中断提交给系统（标准 PC 机的 IRQ 8 是如此设定的）。由于系统实时钟是可编程控制的，你也可以把它设成一个警报器，在某个特定的时刻拉响警报——向系统发送 IRQ 8 中断信号。由此看来，IRQ 8 与生活中的闹铃差不多：中断信号代表着报警器或定时器的发作。

在 Linux 操作系统的实现里，上述中断信号可以通过/dev/rtc（主设备号 10，从设备号 135，只读字符设备）设备获得。对该设备执行读（read）操作，会得到 unsigned long 型的返回值，最低的一个字节表明中断的类型（更新完毕 update-done，定时到达 alarm-rang，周期信号 periodic）；其余字节包含上次读操作以来中断到来的次数。如果系统支持/proc 文件系统，/proc/driver/rtc 中也能反映相同的状态信息。

该设备只能由每个进程独占，也就是说，在一个进程打开(open)设备后，在它没有释放前，不允许其它进程再打开它。这样，用户的程序就可以通过对/dev/rtc 执行 read()或 select()系统调用来监控这个中断——用户进程会被阻塞，直到系统接收到下一个中断信号。对于一些高速数据采集程序来说，这个功能非常有用，程序无需死守着反复查询，耗尽所有的 CPU 资源；只要做好设定，以一定频率进行查询就可以了。

```
#include <stdio.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
```

```
#include <unistd.h>
#include <errno.h>
int main(void)
{
   int i, fd, retval, irqcount = 0;
   unsigned long tmp, data;
   struct rtc_time rtc_tm;
   // 打开 RTC 设备
   fd = open ("/dev/rtc", O_RDONLY);
   if (fd ==   -1) {
      perror("/dev/rtc");
      exit(errno);
   }
   fprintf(stderr, "\n\t\t\tEnjoy TV while boiling water.\n\n");
   // 首先是一个报警器的例子，设定 10 分钟后"响铃"
   // 获取 RTC 中保存的当前日期时间信息
   /* Read the RTC time/date */
   retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);
   if (retval == -1) {
      perror("ioctl");
      exit(errno);
   }
   fprintf(stderr, "\n\nCurrent RTC date/time is %d-%d-%d,%02d:
%02d:%02d.\n",
         rtc_tm.tm_mday, rtc_tm.tm_mon + 1, rtc_tm.tm_year + 1900,
         rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);
   // 设定时间的时候要避免溢出
   rtc_tm.tm_min += 10;
   if (rtc_tm.tm_sec >= 60) {
      rtc_tm.tm_sec %= 60;
```

```
    rtc_tm.tm_min++;
}
if    (rtc_tm.tm_min == 60) {
    rtc_tm.tm_min = 0;
    rtc_tm.tm_hour++;
}
if    (rtc_tm.tm_hour == 24)
    rtc_tm.tm_hour = 0;
// 实际的设定工作
retval = ioctl(fd, RTC_ALM_SET, &rtc_tm);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}
// 检查一下，看看是否设定成功
/* Read the current alarm settings */
retval = ioctl(fd, RTC_ALM_READ, &rtc_tm);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}
fprintf(stderr, "Alarm time now set to %02d:%02d:%02d.\n",
        rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);
// 光设定还不成，还要启用 alarm 类型的中断才行
/* Enable alarm interrupts */
retval = ioctl(fd, RTC_AIE_ON, 0);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}
```

```
    // 现在程序可以耐心的休眠了，10 分钟后中断到来的时候它就会被唤醒
    /* This blocks until the alarm ring causes an interrupt */
    retval = read(fd, &data, sizeof(unsigned long));
    if (retval == -1) {
        perror("read");
        exit(errno);
    }
    irqcount++;
    fprintf(stderr, " okay. Alarm rang.\n");
}
```

这个例子稍微显得有点复杂，用到了 open、ioctl、read 等诸多系统调用，初看起来让人眼花缭乱。其实如果简化一下的话，过程还是“烧开水”：设定定时器、等待定时器超时、执行相应的操作（“关煤气灶”）。

读者可能不理解的是：这个例子完全没有表现出中断带来的好处啊，在等待 10 分钟的超时过程中，程序依然什么都不能做，只能休眠啊？

读者需要注意自己的视角，我们所说的中断能够提升并发处理能力，提升的是 CPU 的并发处理能力。在这里，上面的程序可以被看作是烧开水，在烧开水前，闹铃已经被上好，10 分钟后 CPU 会被中断（闹铃声）惊动，过来执行后续的关煤气工作。也就是说，CPU 才是这里唯一具有处理能力的主体，我们在程序中主动利用中断机制来节省 CPU 的耗费，提高 CPU 的并发处理能力。这有什么好处呢？试想如果我们还需要 CPU 烤面包，CPU 就有能力完成相应的工作，其它的工作也一样。这其实是在多任务操作系统环境下程序生存的道德基础——“我为人人，人人为我”。

好了，这段程序其实是我们进入 Linux 中断机制的引子，现在我们就进入 Linux 中断世界。

更详细的内容和其它一些注意事项请参考内核源代码包中 Documentations/rtc.txt

RTC 中断服务程序

RTC 中断服务程序包含在内核源代码树根目录下的 driver/char/rtc.c 文件中，该文件正是 RTC 设备的驱动程序——我们曾经提到过，中断服务程序一般由设备驱动程序提供，实现设备中断特有的操作。

SagaLinux 中注册中断的步骤在 Linux 中同样不能少，实际上，两者的原理区别不大，只是 Linux 由于要解决大量的实际问题（比如 SMP 的支持、中断的共享等）而采用了更复杂的实现方法。

RTC 驱动程序装载时，rtc_init()函数会被调用，对这个驱动程序进行初始化。该函数的一个重要职责就是注册中断处理程序：

```
if (request_irq(RTC_IRQ,rtc_interrupt,SA_INTERRUPT,”rtc”,NULL)){
    printk(KERN_ERR “rtc:cannot register IRQ %d\n”,rtc_irq);
        return –EIO;
}
```

这个 request_irq 函数显然要比 SagaLinux 中同名函数复杂很多，光看看参数的个数就知道了。不过头两个参数两者却没有区别，依稀可以推断出：它们的主要功能都是完成中断号与中断服务程序的绑定。

关于 Linux 提供给系统程序员的、与中断相关的函数，很多书籍都给出了详细描述，如“Linux Kernel Development”。我这里就不做重复劳动了，现在集中注意力在中断服务程序本身上。

```
static irqreturn_t rtc_interrupt(int irq, void *dev_id,
struct pt_regs *regs)
{
        /*
         *      Can be an alarm interrupt, update complete interrupt,
         *      or a periodic interrupt. We store the status in the
         *      low byte and the number of interrupts received since
         *      the last read in the remainder of rtc_irq_data.
         */
        spin_lock (&rtc_lock);
        rtc_irq_data += 0x100;
        rtc_irq_data &= ~0xff;
        rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);
        if (rtc_status & RTC_TIMER_ON)
                mod_timer(&rtc_irq_timer,
```

```
jiffies + HZ/rtc_freq   + 2*HZ/100);
        spin_unlock (&rtc_lock);
        /* Now do the rest of the actions */
        spin_lock(&rtc_task_lock);
        if (rtc_callback)
                rtc_callback->func(rtc_callback->private_data);
        spin_unlock(&rtc_task_lock);
        wake_up_interruptible(&rtc_wait);
        kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);
        return IRQ_HANDLED;
}
```

这里先提醒读者注意一个细节：中断服务程序是 static 类型的，也就是说，该函数是本地函数，只能在 rtc.c 文件中调用。这怎么可能呢？根据我们从 SagaLinux 中得出的经验，中断到来的时候，操作系统的中断核心代码一定会调用此函数的，否则该函数还有什么意义？实际上，request_irq 函数会把指向该函数的指针注册到相应的查找表格中（还记得 SagaLinux 中的 irq_handler[]吗？）。static 只能保证 rtc.c 文件以外的代码不能通过函数名字显式的调用函数，而对于指针，它就无法画地为牢了。

程序用到了 spin_lock 函数，它是 Linux 提供的自旋锁相关函数，关于自旋锁的详细情况，我们会在以后的文章中详细介绍。你先记住，自旋锁是用来防止 SMP 结构中的其他 CPU 并发访问数据的，在这里被保护的数据就是 rtc_irq_data。rtc_irq_data 存放有关 RTC 的信息，每次中断时都会更新以反映中断的状态。

接下来，如果设置了 RTC 周期性定时器，就要通过函数 mod_timer()对其更新。定时器是 Linux 操作系统中非常重要的概念，我们会在以后的文章中详加解释。

代码的最后一部分要通过设置自旋锁进行保护，它会执行一个可能被预先设置好的回调函数。RTC 驱动程序允许注册一个回调函数，并在每个 RTC 中断到来时执行。

wake_up_interruptible 是个非常重要的调用，在它执行后，系统会唤醒睡眠的进程，它们等待的 RTC 中断到来了。这部分内容涉及等待队列，我们也会在以后的文章中详加解释。

感受 RTC——最简单的改动

我们来更进一步感受中断，非常简单，我们要在 RTC 的中断服务程序中加入一条 printk

语句，打印什么呢？“I’m coming, interrupt!”。

下面，我们把它加进去：

```
… …
spin_unlock(&rtc_task_lock);
printk(“I’m coming , interrupt!\n”);
wake_up_interruptible(&rtc_wait);
… …
```

没错，就先做这些，请你找到代码树的 drivers\char\rtc.c 文件，在其中 irqreturn_t rtc_interrupt 函数中加入这条 printk 语句。然后重新编译内核模块（当然，你要在配置内核编译选项时包含 RTC，并且以模块形式）现在，当我们插入编译好的 rtc.o 模块，执行前面实时钟部分介绍的用户空间程序，你就会看到屏幕上打印的“I’m coming , interrupt!”信息了。

这是一次实实在在的中断服务过程，如果我们通过 ioctl 改变 RTC 设备的运行方式，设置周期性到来的中断的话，假设我们将频率定位 8HZ，你就会发现屏幕上每秒打印 8 次该信息。

动手修改 RTC 实际上是对中断理解最直观的一种办法，我建议你不但注意中断服务程序，还可以看一下 RTC 驱动中 ioctl 的实现，这样你会更加了解外部设备和驱动程序、中断服务程序之间实际的互动情况。

不仅如此，通过修改 RTC 驱动程序，我完成了不少稀奇古怪的工作，比如说，在高速数据采集过程中，我就是利用高频率的 RTC 中断检查高速 AD 采样板硬件缓冲区使用情况，配合 DMA 共同完成数据采集工作的。当然，在有非常严格时限要求的情况下，这样不一定适用。但是，在两块 12 位 20 兆采样率的 AD 卡交替工作，对每秒 1KHz 的雷达视频数据连续采样的情况下，我的 RTC 跑得相当好。

当然，这可能不是一种美观和标准的做法，但是，我只是一名程序员而不是艺术家，只是了解了这么一点点中断知识，我就完成了工作，我想或许您也希望从系统底层的秘密中获得收益吧，让我们在以后的文章中再见。

[1]那么 PowerOff（关机）算不算中断呢？如果从字面上讲，肯定符合汉语对中断的定义，但是从信号格式、处理方法等方面来看，就很难符合我们的理解了。Intel 怎么说的呢？该中断没有采用通用的中断处理机制。那么到底是不时中断呢？我也说不上来：（

[2]之所以这里使用汇编而不是 C 来实现这些函数，是因为 C 编译器会在函数的实现中推入额外的栈信息。而 CPU 在中断来临时保存和恢复现场都按照严格的格式进行，一个字节的变化都不能有。