保护模式下

8259A 芯片编程及中断处理探究

（上）

Version 0.02

哈尔滨工业大学 并行计算实验室 谢煜波[1]

简介

中断处理是操作系统必须完成的任务，在 IBM PC 中，常用一块中断控制芯片（PIC）——8259A 来辅助 CPU 完成中断管理。在实模式下，中断控制芯片（PIC）8259A 的初始化是由 BIOS 自动完成的，然而在保护模式下却需要我们自行编程初始化。本篇拟从操作系统的编写角度详细描述下笔者在此方向上所做的摸索，并在最后通过 pyos 进行实验验证。此是这部份内容的上篇，将详细描述 8259A 芯片的编程部份，对于操作系统中的中断处理以及程序验证将在下篇里面详细描述。

此文只是我在进行操作系统实验过程中的一点心得体会，记下来，避免自己忘记。对于其中可能出现的错误，欢迎你来信指证。

一、中断概述

相信大家对于中断一点都不陌生，这里也不准备详细介绍中断的所有内容，只简单做下概要介绍，这样使对中断没有概念的朋友能建立起一点概念。

计算机除了 CPU 外，还有很多外围设备，然而我们都知道 CPU 的运行速度是很快的，而外围设备的运行速度却不是很快了。假设我们现在需要从磁盘上读入十个字节，而这需要 10 秒钟（很夸张，但这只是一个例子），那么在这 10 秒钟之内，CPU 就无所事事，不得不等待磁盘如蜗牛般的读入这十个字节，如果在这 10 秒钟之内，CPU 转去运行其它的程序，不就可以防止浪费 CPU 的时间了吗？但是这就出现了一个问题，CPU 怎么知道磁盘已经读完数据了呢？实际上，这时磁盘的控制器会向 CPU 发送一个信号，CPU 收到信号之后，就知道磁盘已经读完数据了，于是它就中断正在运行的程序，重新回到原先等待磁盘输入的程序上来继续执行。这只是一个很简单的例子，也只是中断应用的一个很简单的方面，但基本上可以说明问题。可以这么认为：中断就是外部设备或程序同 CPU 通信的一种方式。CPU 在接收到中断信号时，会中断正在运行的程序，转到对中断的处理上，而这个对中断的处理程序常常称为中断服务程序，当中断服务程序处理完中断后，CPU 再返回到原先被中断的程序上继续执行。整个过程如下图所示：

（图 1）

中断有很多类型，比如可屏蔽中断（顾名思义，对此种中断，CPU 可以不响应）、不可屏蔽中断；软中断（一般由运行中的程序触发）、硬中断……等很多分类方法。中断可以完成的任务也很多，比如设备准备完毕、设备运行故障、程序运行故障……，这许多突发事件都可以以中断的方式通知 CPU 进行处理。

二、认识中断号及 8259A 芯片

我们都知道计算机可以挂接上许多外部设备，比如键盘、磁盘驱动器、鼠标、声卡……等等一系列设备，而这些设备都可能在同一时刻向 CPU 发出中断信号，那么 CPU 到底应当响应哪一个设备的中断信号呢？这都通过另外一个芯片来控制，在 IBM PC 机中，这个芯片常常被称作：可编程中断控制器（PIC）8259A，说它可编程，是因为我们可以通过编程来改变它的功能。比如我可以通过编程设定 CPU 应当优先响应哪一个中断，屏蔽哪些中断等等一系列事件。

一个 8259A 芯片共有中断请求（IRQ）信号线：IRQ0~IRQ7，共 8 根。在 PC 机中，共有两片 8259A 芯片，通过把它们联结起来使用，就有 IRQ0~IRQ15，共 16 根中断信号线，每个外部设备使用一根或多个外部设备共用一根中断信号线，它们通过 IRQ 发送中断请求，8259A 芯片接到中断请求后就对中断进行优先级选定，然后对多个中断中具有最高优先级的中断进行处理，将其所对应的中断向量送上通往 CPU 的数据线，并通知 CPU 有中断到来。

这里出现了一个中断向量的概念，其实它就是一个被送往 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东为什么不直接用 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以及节约规则。

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）？

对于这个问题，则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，因为对于 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，因为 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，这于 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。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，因为中断号是与中断服务程序一一对应的，这一点我们将在随后的内容中详细描述。8259A 将中断号通知 CPU 后，它的任务就完成了，至于 CPU 使用此中断号去调用什么程序它就不管了。下图就是 8259A 芯片的结构：

（图 2 来源《Linux 0.11 内核完全注释》）

上图就是 PC 机中两片 8259A 的联结及 IRQ 分配示意图。从图中我们可以看到，两片 8259A 芯片是级联工作的，一个为主片，一个为从片，从片的 INT 端口与主片的 IRQ2 相连。主片通过 0x20 及 0x21 两个端口访问，而从片通过 0xA0 及 0xA1 这两个端口访问。

至于为什么从片的 INT 需要与主片的 IRQ2 相连而不是与其它的 IRQ 相联，很遗憾，我目前无法回答这个问题：(，如果你知道答案，非常希望你能来信指教！不过幸运的是，我们只要知道计算机是的确是这样联的，并且这样连它就可以正常工作就行了！

三、8259A 的编程

8259A 常常称之为 PIC（可编程中断控制器），因此，在使用的时候我们必须通过编程对它进行初始化，需要完成的工作是指定主片与从片怎样相连，怎样工作，怎样分配中断号。在实模式下，也就是计算机加电或重启时，这是由 BIOS 自动完成的，然而当转到保护模式下后，我们却不得不对它进行编程重新设定，这都是由该死的 IBM 与 Intel 为维护兼容性而搞出来的麻烦：(。

在 BIOS 初始化 PIC 的时候，IRQ0~IRQ7 被分配了 0x8~0xF 的中断号，然而当 CPU 转到保护模式下工作的时候，0x8~0xF 的中断号却被 CPU 用来处理错误！一点不奇怪，CPU 是 Intel 生产的，而计算机却是由 IBM 生产的，两家公司没有协调好：（。

因此，我们不得不在保护模式下，重新对 PIC 进行编程，主要的目的就是重新分配中断号。幸好这不是一件太难的工作。

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，下面我们先来看看第一个 ICW1 的结构：

ICW1：发送到 0x20（主片）及 0xa0（从片）端口

7	6	5	4	3	2	1	0

0	0	0	1	M	0	C	I

I 位：若置位，表示 ICW4 会被发送。（ICW4 等下解释）

C 位：若清零，表示工作在级联环境下。

M 位：指出中断请求的电平触发模式，在 PC 机中，它应当被置零，表示采用“边沿触发模式”。

ICW2：发送到 0x21（主片）及 0xa1（从片）端口

7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	0	0	0

ICW2 用来指示出 IRQ0 使用的中断号是什么，因为最后三位均是零，因此要求 IRQ0 的中断号必须是 8 的倍数，这又是一个很巧妙的设计。因为 IRQ1 的中断号就是 IRQ0 的中断号+1，IRQ2 的中断号就是 IRQ0 的中断号+2，……，IRQ7 的中断号就是 IRQ0 的中断号+7，刚好填满一个 8 个的中断向量号空间。

ICW3：发送到 0x21（主片）及 0xa1（从片）端口

ICW3 只有在级联工作的时候才会被发送，它主要用来建立两个 PIC 之间的连接，对于主片与从片，它结构是不一样的。

（主片结构：）

7	6	5	4	3	2	1	0
IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0

上面，如果相应的位被置 1，则相应的 IRQ 线就被用于与从片连接，若清零则表示被连接到外围设备。

（从片结构：）

7	6	5	4	3	2	1	0
0	0	0	0	0	IRQ		

上面的 IRQ 位指出了是主片的哪一个 IRQ 连到了从片，这需要同主片上发送的上面的主片结构字一致。

ICW4：发送到 0x21（主片）及 0xa1（从片）端口

7	6	5	4	3	2	1	0
0	0	0	0	0	0	EOI	80x86

80x86 位：若置位则表示工作在 80x86 架构下。

EOI 位：若置位则表示自动清除中断请求信号。在 PC 上这位需要被清零。要理解这一位，我们需要详细了解一下 8259A 的内部中断处理流程。

三、8259A 的内部中断处理流程

下面我们就来从一个系统程序员（System Programmer）的角度看看 8259A 的内部结构。

（图 3）

首先，一个外部中断请求信号通过中断请求线 IRQ，传输到 IMR（中断屏蔽寄存器），IMR 根据所设定的中断屏蔽字（OCW1），决定是将其丢弃还是接受。如果可以接受，则 8259A 将 IRR（中断请求暂存寄存器）中代表此 IRQ 的位置位，以表示此 IRQ 有中断请求信号，并同时向 CPU 的 INTR（中断请求）管脚发送一个信号，但 CPU 这时可能正在执行一条指令，因此 CPU 不会立即响应，而当这 CPU 正忙着执行某条指令时，还有可能有其余的 IRQ 线送来中断请求，这些请求都会接受 IMR 的挑选，如果没有被屏蔽，那么这些请求也会被放到 IRR 中，也即 IRR 中代表它们的 IRQ 的相应位会被置 1。

当 CPU 执行完一条指令时后，会检查一下 INTR 管脚是否有信号，如果发现有信号，就会转到中断服务，此时，CPU 会立即向 8259A 芯片的 INTA（中断应答）管脚发送一个信号。当芯片收到此信号后，判优部件开始工作，它在 IRR 中，挑选优先级最高的中断，将中断请求送到 ISR（中断服务寄存器），也即将 ISR 中代表此 IRQ 的位置位，并将 IRR 中相应位置零，表明此中断正在接受 CPU 的处理。同时，将它的编号写入中断向量寄存器 IVR 的低三位（IVR 正是由 ICW2 所指定的，不知你是否还记得 ICW2 的最低三位在指定时都是 0，而在

这里，它们被利用了！）这时，CPU 还会送来第二个 INTA 信号，当收到此信号后，芯片将 IVR 中的内容，也就是此中断的中断号送上通向 CPU 的数据线。

这个内容看起来仿佛十分复杂，但如果我们用一个很简单的比喻来解释就好理解了。CPU 就相当于一个公司的老总，而 8259A 芯 片就相当于这个老总的秘书，现在有很多人想见老总，但老总正在打电话，于是交由秘书先行接待。每个想见老总的人都需要把自己的名片交给秘书，秘书首先看看 名片，有没有老总明确表示不愿见到的人，如果没有就把它放到一个盒子里面，这时老总的电话还没打完，但不停的有人递上名片求见老总，秘书就把符合要求的名 片全放在盒子里了。这时，老总打完电话了，探出头来问秘书：有人想见我吗？这时，秘书就从盒子里挑选一个级别最高的，并把他的名片交给老总。

这里需要理解的是中断屏蔽与优先级判定并不是一回事，如果被屏蔽了，那么参加判定的机会也都没了。在默认情况下，IRQ0 的优先级最高，IRQ7 最低。当然我们可以更改这个设定，这样在下面有详细描述。

言归正传，当芯片把中断号送上通往 CPU 的数据线后，就会检测 ICW4 中的 EOI 是否被置位。如果 EOI 被置位表示需要自动清除中断请求信号，则芯片会自动将 ISR 中的相应位清零。如果 EOI 没有被置位，则需要中断处理程序向芯片发送 EOI 消息，芯片收到 EOI 消息后才会将 ISR 中的相应位清零。

这里的机关存在于这样一个地方。优先权判定是存在于 8259A 芯片中的，假如 CPU 正在处理 IRQ1 线来的中断，这时 ISR 中 IRQ1 所对应的位是置 1 的。这时来了一个 IRQ2 的中断请求，8259A 会将其同 ISR 中的位进行比较，发现比它

高的 IRQ1 所对应的位被置位，于是 8259A 会很遗憾的告诉 IRQ2：你先在 IRR 中等等。而如果这时来的是 IRQ0，芯片会马上让其进入 ISR，即将 ISR 中的 IRQ0 所对应的位置位，并向 CPU 发送中断请求。这时由于 IRQ1 还在被 CPU 处理，所以 ISR 中 IRQ1 的位也还是被置位的，但由于 IRQ0 的优先级高，所以 IRQ0 的位也会被置位，并向 CPU 发送新的中断请求。此时 ISR 中 IRQ0 与 IRQ1 的位都是被置位的，这种情况在多重中断时常常发生，非常正常。

如果 EOI 被设为自动的，那么 ISR 中的位总是被清零的（在 EOI 被置位的情况下，8259A 只要向 CPU 发送了中断号就会将 ISR 中的相应位清零），也就是如果有中断来，芯片就会马上再向 CPU 发出中断请求，即使 CPU 正在处理 IRQ0 的中断，CPU 并不知道谁的优先级高，它只会简单的响应 8259A 送来的中断，因此，这种情况下低优先级的中断就可能会中断高优先级的中断服务程序。所以在 PC 中，我们总是将 EOI 位清零，而在中断服务程序结束的时候才发送 EOI 消息。

四、EOI 消息及 OCW 操作命令字

上面所描述的内容几乎全与 EOI 消息有关，我们现在也已经知道了，应当在中断服务程序结束的时候发送 EOI 消息给芯片，让芯片在这个时候将其相应的位清零。那让我们现在来揭开 EOI 消息的神秘面纱。

要认识 EOI 消息，我们需要先行了解 OCW（操作命令字），它们用来操作 8259A 的优先级、中断屏蔽及中断结束等控制。总共有三个 OCW，它们也都有自己很独特的格式，不过它们的发送却不须按固定的顺序进行。

OCW1：中断屏蔽，发送到 0x21(主片)或 0xa1（从片）端口

7	6	5	4	3	2	1	0
IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0

如果相应的位置 1，则表示屏蔽相应的 IRQ 请求。

OCW2：优先权控制及中断结束命令，发送到 0x20（主片）及 0xa0（从片）端口

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

先来看看中断结束消息（EOI）

EOI 也是 OCW2 型命令中的一种，当 EOI 位被置 1，这就是一个 EOI 消息 。SL 是指定级别位，如果 SL 被置位，则表明这是一个指定级别的 EOI 消息，这个消息可以指定将 ISR 中的哪一位清零，即告诉 8259A 应当清除哪一个 IRQ 信号。L2、L1、L0 就用来指定 IRQ 的编号。而在实际运用中我们却将 SL 及 L2、L1、L0 全置零。

SL 置零表示这是一个不指定级别的 EOI 消息，则 8259A 芯片自动将 ISR 中所有被置位的 IRQ 里优先级最高的清零，因为它是正在被处理及等待处理的中断中优先级最高的，也就一定是 CPU 正在处理的中断。

现在再来看看优先级控制命令。

当 R 为 0 时，表明这是一个固定优先权方式，IRQ0 最高，IRQ7 最低。

当 R 为 1 时，表明这是一个循环优先权，比如，如指定 IRQ2 最低，则优先级顺序就为：

IRQ2 < IRQ1 < IRQ0 < IRQ7 < IRQ6 < IRQ5 < IRQ4 < IRQ3

（编者注：可以将其记忆为“IRQ7<IRQ6<IRQ5<IRQ4<IRQ3<IRQ2<IRQ1<IRQ0”进行循环左移后的结果）

也即，如果 IRQ(i)最低，那么 IRQ(i + 1)就最高。

所以，在这种方式下需要先行指定一个最低优先级。如果 SL 被清零，则表示使用自动选择方式，那么正在被处理的中断服务在下一次，就被自动指定为最低优先级。如果 SL 被置位，那么 L2、L1、L0 所指定的 IRQ 就被指定为最低优先级。

在指定优先级的时候，也可通过置位 EOI，即可以将指定优先级命令与中断结束消息同时发送。

上面的描述看起来很复杂，其实对于一般的操作系统编写来说大多就使用一种形式：0010 0000，我也很乐意将其称为 EOI 消息。

还有一个 OCW3 命令字，可以指定特殊的屏蔽方式及读出 IRR 与 ISR 寄存器，不过一般在操作系统中并不需要这样的操作。在操作系统编写中一般只用到前面两个命令字的格式（至少 pyos 是这样：））由于本文并非硬件手册，只想为操作系统的编写提供一点帮助，因此，如果你想了解完整的 OCW3 命令字格式，请查阅相应的硬件手册，或本文的参考资料 1。

五、上篇结束语

在这一篇中，较为详细的描述了对 8259A 中断控制器芯片编程所需具备的基础知识，在编写操作系统的过程中，我们就需要向相应端口发送相应的 ICW 或 OCW，完成对 8259A 的操作，具体的代码将在下篇中描述。

在下篇中将更主要的描述操作系统对中断服务程序的处理及安排，并以开发中的 pyos 做为例子进行实验。

在上篇中，我们详细讲述了保护模式下对于中断的基本原理已及对可编程中断控制器 8259A 的编程方法。如果说上一篇更偏重有原理及特定的硬件编程方法， 那么本篇就会偏软一点，将详细描述怎样编写操作系统中的中断处理程序，并将通过 pyos 进行验证。在此篇中，你将会详细了解到操作系统是怎样处理中断的， 中断处理程序是怎样编写的，操作系统又是怎样调用中断处理程序的。

希望本篇可以使你对上述问题有个比较清晰的认识。

本篇是独立的，当然，如果你阅读了上篇，那么对于理解本篇中所描述的内容无疑是有巨大帮助的。

pyos 是一个实验性的架构系统，阅读本篇之后，你可以尝试着改动 pyos 中的中断处理部份，这样你将更可以详细而深入的理解多重中断，现场保护等内 容，本篇在最后也将对于怎样进行这样的自我实验做些许描述。如果你在学习“操作系统”或“组成原理”的过程中，对于书中描述的内容感到不太直观，你可以试 试用 pyos 去验证你所学习的知识。

再次声明：此文只是我在进行操作系统实验过程中的一点心得体会，记下来，避免自己忘记。对于其中可能出现的错误，欢迎你来信指证。

一、操作系统中断服务概述

现代计算机如果从纯硬件角度，我个人更倾向于将它理解为是利用的一种所谓的“中断驱动”机制，就相当于我们常常津津乐道的 Windows 的“消息驱动”机制一样。CPU 在正常情况下按顺序执行程序，一旦有外部中断到来，CPU 将会中断现行程序的运行，转到中断服务程序进行中断处理，当中断处理完成之后，CPU 再回到原来执行程序被中断的地方继续执行，并等待下一个中断的来临。

CPU 需要响应的中断有很多种，比如键盘中断、磁盘中断、CPU 时钟中断等等，每种中断的功能都是不同的，而所需要的中断服务程序也是不同的，CPU 又是怎么识别这种种不同的中断的呢？

二、中断描述符表及中断描述符

在上一篇中，我们知道了 CPU 是通过给这些不同的中断分配不同的中断号来识别的。一种中断就对应一个中断号，而一个中断号就对应一个中断服务程序这 样，当有中断到来的时候，CPU 就会识别出这个中断的中断号，并将这个中断号作为一个索引，在一张表中查找此索引号对应的一个入口地址，而这个入口地址就 是中断服务程序的入口地址，CPU 取得入口地址后，就跳转到这个地址所指示的程序处运行中断服务程序。

这张存放不同中断服务程序的表在系统中 常常称为“中断向量表”。在保护模式下也常常称为“中断描述符表”（IDT），这个表中的每一项就是一个中断描述符，每一个中断描述符都包含中一个中断服 务程序的地址，CPU 通过将中断号做为索引值取得的就是这样一个“中断描述符”，通过“中断描述符”，CPU 就可以得到中断服务程序的地址了，下面，我们 就来看看中断描述符的结构：

< language="JavaScript" type="text/javascript">resizeImage('http://purec.binghua.com/Article/UploadFiles/200447231928452.jpg','image0')

(图一)

上图就是一个“中断描述符”的结构，其中 P 位是存在为，置 1 的时候就是这个描符述可以被使用；DPL 是特权级，可以指定 为 0~3 中的一级；保留位是留给 Inter 将来用的，在现阶段，我们只需要简单的将其置零就可以；偏移量总

共是 32 位，它表示一个中断服务程序在内存中的 位置。由于保护模式下，内存的寻址是由段选择符与偏移量指定的，所以在中断描述符中也分别设定了段选择符与偏移量位，他们共同决定了一个中断服务程序在内 存中的位置。（有关保护模式下内存的寻址方面的描述，可以参考《操作系统引导探究》一文。）

在前面我们描述了，一个中断描述符是放在一张中断 描述符表中的，而中断号就是中断描述符在中断描述符表中的索引或说下标。那么系统又怎么知道中断描述符表是放在什么地方的呢？这在系统中是通过一个称之为“中断描述符表寄存器”（IDTR）实现的，这个寄存器中就存放了“中断描述符表”在内存中的地址。下面，我们也来看看这个寄存器的结构：

< language="JavaScript" type="text/javascript">resizeImage('http://purec.binghua.com/Article/UploadFiles/200447231944170.gif','image1')

(图二)

下面，我们可以比较完整的来欣赏一下 CPU 处理中断的流程：

< language="JavaScript" type="text/javascript">resizeImage('http://purec.binghua.com/Article/UploadFiles/20044723207574.gif','image2')

(图三)

三、pyos 中的中断系统实验

下面，我们将以 pyos 做为例子，用它来实验操作系统中中断系 统的实现。当然，在实际的操作系统中这也许是非常复杂的，但我们现在通过 pyos 也完全可以进行这样的实验。在实验正式开始之前，你许要下载本实验所用到 的源代码，这可以在我们的网站“纯 C 论坛”（http://purec.binghua.com）“操作系统实验专区”中下载，也可以来信向作者索要。对 于实验环境的搭建你也许需要看看“操作系统实验专区”中“When Do We Write Our Chinese Os （1）”对此的描述。

在实验正式开始之前，我们将详细描述一下 pyos 的文件组织形式。Go~~~

3.1 pyos 的文件组织形式

本实验用到的 pyos 目前的文件组织如下：“boot.asm”、“setup.asm”这两个文件是操作系统引导文件，他们负责把 pyos 的内核读入 内存，然后转到内核执行。（这方面的内容可以参见《操作系统引导探究》一文。）“kernel.cpp”就是 pyos 的内核，pyos 是用 c++开发的， 因此它的内核看起来非常简单，也比较有结构，下面就是“kernel.cpp”中的内容：

```
#include "system.h"
#include "video.h"

extern "C" void Pyos_Main()
{
```

```
    /* 系统初始化 */
    class_pyos_System::Init() ;
    class_pyos_Video::ClearScreen() ;
    class_pyos_Video::PrintMessage( "Welcome to Pyos :)" ) ;
    for(;;);
   }
```

我想，这样的程序也许不用我做什么注释了。“system.h”中定义了一个名为“class_pyos_System”的系统类，用来做系统初始化的 工作，“vide.h”中定义了一个名为“class_pyos_Video”的显卡类，它封装了对 VGA 显卡的操作，从上面的代码中我们可以看见，系统 先通过 class_pyos_System 类的 Init()完成系统初始化，然后调用 class_pyos_Video 类中的 ClearScreen() 进行清屏，最后用 PrintMessage()输出了一条欢迎信息。下面，我就一步一步的来剥开上面看上去很神秘的 Init()——系统初始化函数。对于 显卡类，你可以参看源代码，本篇中将不进行描述，也许以后我会用专门的一篇来详细描述它。当然，你如果看过“When Do We Write Our Chinese OS （3）”的话，对于显卡类的理解就易容反掌了。

3.2 pyos 的系统初始化

下面，我们来看看 pyos 的系统初始化函数：

```
#include "interrupt.h"
/* 系统初始化 */
void class_pyos_System::Init()
```

```
{
 /* 初始化 Gdt 表 */
 InitGdt() ;
 /* 初始化段寄存器 */
 InitSegRegister() ;
 /* 初始化中断 */
 class_pyos_Interrupt::Init() ;
}
```

是的，他就是这么简单，由于 InitGdt 与 InitSegRegister 的内容在《操作系统引导探究》中已经描述过了，这里我们就专注于我们本篇的核心内容——对于中断系统的初始化。

从上面的代码中我们可以看出，系统首先在“interrupt.h”中定义了一个名为“class_pyos_Interrup”的中断类，专门来处理 系统的中断部份。然后，系统调用中断类的 Init()函数，来进行初始化。呵呵，我们马上就去看看这个中断类的初始化函数到底做了些什么：

```
/* 初始化中断服务 */
void class_pyos_Interrupt::Init()
{
 /* 初始化中断可编程控件器 8259A */
 Init8259A() ;
 /* 初始化中断向量表 */
 InitInterruptTable() ;
```

```
 /* 许可键盘中断 */
 class_pyos_System::ToPort( 0x21 , 0xfd ) ;
 /* 汇编指令，开中断 */
 __asm__( "sti" ) ;
}
```

我想，对于这样自说明式的代码，解释是多余的，我们还是抓紧时间来看看它首先是怎样初始化 8259A 的吧：

```
/* 初始化中断控制器 8259A */
void class_pyos_Interrupt::Init8259A()
{
 // 给中断寄存器编程
 // 发送 ICW1：使用 ICW4，级联工作
 class_pyos_System::ToPort( 0x20 , 0x11 ) ;
 class_pyos_System::ToPort( 0xa0 , 0x11 ) ;

 // 发送 ICW2，中断起始号从 0x20 开始（第一片）及 0x28 开始（第二片）
 class_pyos_System::ToPort( 0x21 , 0x20 ) ;
 class_pyos_System::ToPort( 0xa1 , 0x28 ) ;

 // 发送 ICW3
 class_pyos_System::ToPort( 0x21 , 0x4 ) ;
```

```
class_pyos_System::ToPort( 0xa1 , 0x2 ) ;

// 发送 ICW4
class_pyos_System::ToPort( 0x21 , 0x1 ) ;
class_pyos_System::ToPort( 0xa1 , 0x1 ) ;

// 设置中断屏蔽位 OCW1 ，屏蔽所有中断请求
class_pyos_System::ToPort( 0x21 , 0xff ) ;
class_pyos_System::ToPort( 0xa1 , 0xff ) ;
}
```

从上面的代码可以看出程序是通过向 8259A 发送 4 个 ICW 对 8259A 进行初始化的。其中 ToPort 是 class_pyos_System 类中定义的一个成员函数，它的声明如下：

```
/* 写端口 */
void class_pyos_System::ToPort( unsigned short port , unsigned char data )
```

OK，好像又不需要我多解释了，当然，你也许会问，为什么会放送这几个值的数据，而不是其它值的数据。对于这个问题，因为在“上篇”中已经详细描述了，这里就不再浪费大家的时间了。:)

3.3 初始化 pyos 的中断向量表

从中断初始化的代码中我们可以清楚的看见，pyos 在进行完 8259A 的初始

化后，调用 InitInterruptTable()对中断向量表进行了初始化，这可是本篇的核心内容，我们这就来看看这个核心函数：

```
/* 中断描述符结构 */
struct struct_pyos_InterruptItem{
 unsigned short Offset_0_15 ; // 偏移量的 0~15 位
 unsigned short SegSelector ; // 段选择符
 unsigned char UnUsed ; // 未使用，须设为全零
 unsigned char Saved_1_1_0 : 3 ; // 保留，需设为 110
 unsigned char D : 1 ; // D 位
 unsigned char Saved_0 : 1 ; // 保留，需设为 0
 unsigned char DPL : 2 ; // 特权位
 unsigned char P : 1 ; // P 位
 unsigned short Offset_16_31 ; // 偏移量的 16~31 位
} ;

/* IDTR 所用结构 */
struct struct_pyos_Idtr{
 unsigned short IdtLengthLimit ;
 struct_pyos_InterruptItem* IdtAddr ;
} ;

static struct_pyos_InterruptItem m_Idt[ 256 ] ; // 中断描述符表项
```

```
	static struct_pyos_Idtr m_Idtr ; // 中断描述符寄存器所用对象


	extern "C" void pyos_asm_interrupt_handle_for_default() ; // 默认中断处
理函数
	/* 初始化中断向量表 */
	void class_pyos_Interrupt::InitInterruptTable()
	{
	 /* 设置中断描述符，指向一个哑中断，在需要的时候再填写 */
	 struct_pyos_InterruptItem tmp ;
	 tmp.Offset_0_15 = ( unsigned int )
pyos_asm_interrupt_handle_for_default ;
	 tmp.Offset_16_31 = ( unsigned int )
pyos_asm_interrupt_handle_for_default >> 16 ;
	 tmp.SegSelector = 0x8 ; // 代码段
	 tmp.UnUsed = 0 ;
	 tmp.P = 1 ;
	 tmp.DPL = 0 ;
	 tmp.Saved_1_1_0 = 6 ;
	 tmp.Saved_0 = 0 ;
	 tmp.D = 1 ;

	 for( int i = 0 ; i < 256 ; ++i ){
```

```
m_Idt[ i ] = tmp ;
}

m_Idtr.IdtAddr = m_Idt ;
m_Idtr.IdtLengthLimit = 256 * 8 - 1 ; // 共 256 项，每项占 8 个字节

// 内嵌汇编，载入 ldt
__asm__( "lidt %0" : "=m"( m_Idtr ) ) ; //载入 GDT 表
}
```

程序首先说明了两个结构，一个用来描述中断描述符，一个用来描述中断描述符寄存器。大家可以对照前面的描述看看这两个结构中的成员分别对应硬件系统中的哪一位。

之后，程序建立了一个中断描述符数组 m_Idt，它共有 256 项，这是因为 CPU 可以处理 256 个中断。还建立了一个中断描述符寄存器所用的对象。随后，程序开始为这些变量赋值。

从程序中我们可以看出，pyos 现在是将每个中断描述符都设成一样的，均指向一个相同的中断处理程 序：pyos_asm_interrupt_handle_for_default()，在一个实际的操作系统中，在最初初始化的时候，也常常是这样做 的，这个被称之为“默认中断处理程序”的程序通常是一个什么也不干的“哑中断处理程序”或者是一个只是简单报错的处理程序。而要等到实际需要时，才实用相 应的处理程序替换它。

程序在建立“中断描述符表”后，用 lidt 指令将中断描符述表寄存器所用的内

容载入了中断描述符寄存器（IDTR）中， 对于“中断描述符表”的初始化就完成了，下面我们可以来看看，pyos_asm_interrupt_handle_for_default()这个程序到底做了些什么事：

3.4 中断处理程序的编写

```
pyos_asm_interrupt_handle_for_default:
;保护现场
pushad
;调用相应的 C++处理函数
call pyos_interrupt_handle_for_default
;恢复现场
popad
;告诉硬件中断处理完毕，即发送 EOI 消息
mov al , 0x20
out 0x20 , al
out 0xa0 , al
;返回
iret
```

这个程序是在一个名为“interrupt.asm”的汇编文件中，显然，它是一个汇编语言写的源程序。为什么这里又要用汇编语言编写而不直接用 C++内嵌汇编编写呢，比如写成下面这样：

```
void pyos_asm_interrupt_handle_for_default()
```

```
{
 __asm__( "pushad" ) ;
 /* do something */
 __asm__( "popad" ) ;
 __asm__( "iret" ) ;
}
```

这里我们需要了解这样一个问题。中断服务程序是由 CPU 直接调用的，随后，它使用 iret 指令返回，而不想一般的 c/c++函数由 ret 返回。c /c++的编译器在处理 c/c++语言的函数的时候，会在这个函数的开头与结尾加上很多栈操作，以支持程序调用，比如上边的代码就有可能被 c/c++编译 器处理成如下形式：(其中绿色为编译器自行加上的代码)

```
pusha
pushad
/* do something */
popad
iret
popa
ret
```

请注意这样一个实事，当程序运行到 iret 时就返回了，而随后的 popa 就不会被执行，因此这样就破坏了堆栈，于是，我们就只能通过汇编语言编写中断处理程序。

现在再来看看 pyos 中用汇编语言写的中断处理程序，首先它用 pushad 指

令把寄存器中的内容压入堆栈，这常常称之为保护现场，因为之后程序需要返回 被中断的程序中继续运行，因此这些寄存器中的内容也必须在中断处理程序结束时恢复。保护现场完了之后，它调用了一个 c++语言程序 pyos_interrupt_handle_for_default()，然后，它弹出原先保存在堆栈中的寄存器的内容，这常常称为恢复现场，随后，它 发送了 EOI 消息通知 8259A 中断处理完成（关于 EOI 消息，在“上篇”中有详细描述），最后通过 iret 返回被中断的程序处继续执行。

晕！原来 pyos_asm_interrupt_handle_for_default()只是一个汇编的壳，而真正的处理函数是 pyos_interrupt_handle_for_default()！怪不得它会在名字中多个“asm”呢？：），下面，我们不得不来看看真正的中 断处理程序做了什么：

```
extern "C" void pyos_interrupt_handle_for_default()
{
 /* 处理中断 */
 /* 读 0x60 端口，获得键盘扫描码 */
 char ch = class_pyos_System::FromPort( 0x60 ) ;
 /* 显示键盘扫描码 */
 class_pyos_Video::PrintMessage( ch ) ;
}
```

yeah！这才是真正的中断处理程序，不过它的内容很简单，就是读键盘的 0x60 端口，获得键盘的扫描码，然后显示这个扫描码。是不是很简单？:)

3.5 class_pyos_Interrupt::Init()的最后工作

现在让我们重新回到中断类的初始化程序 Init()中吧。Init()在完成了 8259A 及中断描述符表的初始化工作之后，它的工作就也近尾声了。随后，它调用了下面一个程序：

```
/* 许可键盘中断 */
class_pyos_System::ToPort( 0x21 , 0xfd ) ;
```

这是向 8259A 发送中断屏蔽字，十六进制 fd 所对应的二进制为 1111 1101，在“上篇”中我们知道了 1 代表屏蔽，而 0 代表不屏蔽，因此 fd 就表示屏蔽了 IRQ0、IRQ2~IRQ7，而唯 IRQ1 没有屏蔽。通过“上篇” 我们知道 IRQ1 是代表的键盘中断，因此这一语句的意思就是屏蔽掉除键盘中断之外的所有中断，也即：只响应键盘中断。最后，程序用 sti 汇编指令打开了 CPU 的中断请求功能。

3.6 实验结果

想想我们前面所描述的代码，你现在应当知道了本实验最后所应达到的一个实 验结果：pyos 起动后，会响应键盘中断，而中断服务程序的功能是输出键盘的扫描码，也就是说如果你敲击键盘的话，你可以看见计算机的屏蔽上输出键盘的扫 描码。下面我们就能看看我们的实验是否达到了程序所预期的效果。下面，我们在 Virtual PC 中启动我们自己的操作系统——pyos： ）

< language="JavaScript"
type="text/javascript">resizeImage('http://purec.binghua.com/Article/UploadFiles/200447232056719.jpg','image3')

我们可以看见，现在 pyos 已经启动了，现在让我们随便敲击一下键盘：

< language="JavaScript"
type="text/javascript">resizeImage('http://purec.binghua.com/Article/UploadFiles/200447232118778.jpg','image4')

呵呵，屏幕上出现了方才敲键的扫描码。看来我们程序的目的的确是达到了不过怎么有两个字符呢？再敲一次试试：）

< language="JavaScript"
type="text/javascript">resizeImage('http://purec.binghua.com/Article/UploadFiles/200447232137994.jpg','image5')

呵呵，又出来两个字符，看来的确是按一个键发生了两次键盘中断，为什么会这样呢？这个问题留到下一篇再解决吧。：）

3.7 实验的改进

pyos 是一个实验系统，开发他的目的就是为了做实验，检验所学，因此，你完全可以用它来进行实验。比如，你可以少屏蔽两个中断，为每个中断指定不

同的 中断服务程序，以观察多重中断是怎样工作的，中断屏蔽又是怎样起作用的。你还可以用“上篇”所介绍的方法，改变不同中断的优先级，看看中断优先级又是怎样 工作的。本实验中的“中断”二字还体现的不明显，但你马上就可以改改比如在主程序中，不要让它空循环，把原来：

for(;;) ;

改成：

```
for( ;; ){
 class_pyos_Video( '0' ) ;
 }
```

然后，你再敲击键盘，你就会体会到“中断”二字的含义。可以完成的实验很多，只要你愿意去做，也非常欢迎你能来信与笔者交流。 ：)