Verum ipsum factum.

—— *Giambattista Vico*

1 马上动手写一个最小的“操作系统”

虽说万事开头难，但有时也未必。比如说，写一个有实用价值的操作系统是一项艰巨的工作，但一个最小的操作系统或许很容易就实现了。现在我们就来实现一个小得无法再小的“操作系统”，建议你跟随书中的介绍一起动手来做，你会发现不但很容易，而且很有趣。

1.1 准备工作

对于写程序，准备工作无非就是硬件和软件两方面，我们来看一下：

1. 硬件
   - 一台计算机（Linux 操作系统[1]或 Windows 操作系统均可）
   - 一张空白软盘
2. 软件
   - 汇编编译器 NASM

     NASM 最新版本可以从其官方网站获得[2]。此刻你可能会有疑问：这么多汇编编译器中，为什么选择 NASM？对于这一点本书后面会有解释。
   - 软盘绝对扇区读写工具

     在 Linux 下可使用 dd 命令，在 Windows 下则需要额外下载一个工具比如 rawrite[3]或者图形界面的 rawwritewin[4]。当然如果你愿意，也可以自己动手写一个“能用就好”的工具，并不是很复杂[5]。

[1]实际上 Linux 并非一种操作系统，而仅仅是操作系统的内核。在这里比较准确的说法是 GNU/Linux，本书提到的 Linux 泛指所有以 Linux 为内核的 GNU/Linux 操作系统。GNU 经常被人们遗忘，但它的贡献无论怎样夸大都不过分，这不仅仅是因为在你所使用的 GNU/Linux 中，GNU 软件的代码量是 Linux 内核的十倍，更加因为，如果没有以 Richard Stallman 为首的 GNU 项目倡导自由软件文化并为之付出艰辛努力，如今你我可能根本没有自由软件可用。本书将 GNU/Linux 简化为 Linux，仅仅是为表达方便，绝不是因为 GNU 这一字眼可有可无。

[2]NASM 的官方网站位于 http://sourceforge.net/projects/nasm。

[3]rawrite 可以在许多地方找到，比如http://ftp.debian.org/debian/tools/。

[4]下载地址为http://www.chrysocome.net/rawwrite。

[5]我们不需要太强大的软盘读写工具，只要能将 512 字节的数据写入软盘的第一个扇区就足够了。

1.2 十分钟完成的操作系统

你相不相信，一个“操作系统”的代码可以只有不到 20 行？请看代码1.1。

代码 1.1 chapter1/a/boot.asm

```
1         org     07c00h                  ; 告诉编译器程序加载到7c00处
2         mov     ax, cs
3         mov     ds, ax
4         mov     es, ax
5         call    DispStr                 ; 调用显示字符串例程
6         jmp     $                       ; 无限循环
7   DispStr:
8         mov     ax, BootMessage
9         mov     bp, ax                  ; ES:BP = 串地址
10        mov     cx, 16                  ; CX = 串长度
11        mov     ax, 01301h              ; AH = 13,  AL = 01h
12        mov     bx, 000ch               ; 页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)
13        mov     dl, 0
14        int     10h                     ; 10h 号中断
15        ret
16  BootMessage:            db      "Hello, OS world!"
17  times   510-($-$$)      db      0       ; 填充剩下的空间，使生成的二进制代码恰好为512字节
18  dw      0xaa55                          ; 结束标志
```

把这段代码用 NASM 编译一下：

```
▷ nasm boot.asm -o boot.bin
```

我们就得到了一个 512 字节的 boot.bin，让我们使用软盘绝对扇区读写工具将这个文件写到一张空白软盘的第一个扇区。在 Linux 下可以这样做[6]：

```
▷ dd if=boot.bin of=/dev/fd0 bs=512 count=1
```

在 Windows 下可以这样做[7]：

```
▷ rawrite2.exe -f boot.bin -d A
```

好了，你的第一个“操作系统”就已经完成了。这张软盘已经是一张引导盘了。

把它放到你的软驱中重新启动计算机，从软盘引导，你看到了什么？

计算机显示出你的字符串了！红色的“Hello, OS world!”，多么奇妙啊，你的“操作系统”在运行了！

如果使用虚拟机比如 Bochs 的话（下文中将会有关于 Bochs 的详细介绍），你应该能看到如图1.1所示的画面[8]。

这真的是太棒了，虽然你知道它有多么简陋，但是，毕竟你已经制作了一个可以引导的软盘了，而且所有工作都是你亲手独立完成的！

[6]取决于硬件环境和具体的 Linux 发行版，此命令可能稍有不同。
[7]rawrite 有多个版本，此处选用的是 2.0 版。
[8]画面看上去有点乱，因为我们打印字符前并未进行任何的清屏操作。

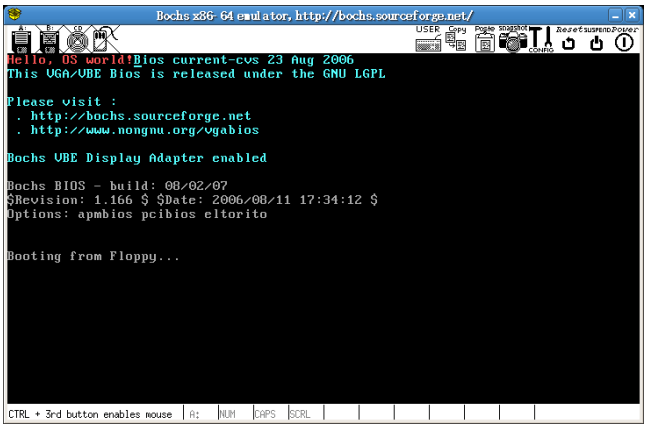

图 1.1 最小的“操作系统”

1.3 引导扇区

你可能还没有从刚刚的兴奋中走出来，可是我不得不告诉你，实际上，你刚刚所完成的并不是一个完整的操作系统，而仅仅是一个最最简单的引导扇区（Boot Sector）。然而不管我们完成的是什么，至少，它是直接在裸机上运行的，不依赖于任何其他软件，所以，这和我们平时所编写的应用软件有本质的区别。它不是操作系统，但已经具备了操作系统的一个特性。

我们知道，当计算机电源被打开时，它会先进行加电自检（POST），然后寻找启动盘，如果是选择从软盘启动，计算机就会检查软盘的 0 面 0 磁道 1 扇区，如果发现它以 0xAA55[9]结束，则 BIOS 认为它是一个引导扇区。当然，一个正确的引导扇区除了以 0xAA55 结束之外，还应该包含一段少于 512 字节的执行码。

好了，一旦 BIOS 发现了引导扇区，就会将这 512 字节的内容装载到内存地址 0000:7c00 处，然后跳转到 0000:7c00 处将控制权彻底交给这段引导代码。到此为止，计算机不再由 BIOS 中固有的程序来控制，而变成由操作系统的一部分来控制。

现在，你可能明白了为什么在那段代码的第一行会出现“`org 07c00`”这样的代码。没错，这行代码就是告诉编译器，将来我们的这段程序要被加载到内存偏移地址 0x7c00 处。好了，下面将对代码的其他部分进行详细解释。

1.4 代码解释

其实程序的主体框架只是第2行到第6行这么一点点而已，其中调用了一个显示字符串的子程序。程序的第2、3、4行是 3 个 mov 指令，使 ds 和 es 两个段

[9]假如把此扇区看做一个字符数组`sector[]`，那么此结束标志相当于`sector[510]==0x55`，且`sector[511]==0xAA`。

寄存器指向与 cs 相同的段，以便在以后进行数据操作的时候能定位到正确的位置。第5行调用子程序显示字符串，然后`jmp $`让程序无限循环下去。

可能有很多人开始学汇编时用的都是 MASM，其实 NASM 的格式跟 MASM 总体上是差不多的，在这段程序中，值得说明的地方有以下几点：

1. 方括号 [] 的使用

   在 NASM 中，任何不被方括号 [] 括起来的标签或变量名都被认为是地址，访问标签中的内容必须使用 []。所以，

   ```
   mov ax, BootMessage
   ```

   将会把“`Hello, OS world!`”这个字符串的首地址传给寄存器 ax。又比如，如果有：

   ```
   foo dw 1
   ```

   则`mov ax, foo`将把`foo`的地址传给`ax`，而 `mov bx, [foo]`将把`bx`的值赋为 1。

   实际上，在 NASM 中，变量和标签是一样的，也就是说：

   ```
   foo dw 1 等价于 foo: dw 1
   ```

   而且你会发现，Offset 这个关键字在 NASM 也是不需要的。因为不加方括号时表示的就是 Offset。

   笔者认为这是 NASM 的一大优点，要地址就不加方括号，也不必额外地用什么 Offset，想要访问地址中的内容就必须加上方括号。代码规则非常鲜明，一目了然。

2. 关于 `$` 和 `$$`

   `$` 表示当前行被汇编后的地址。这好像不太容易理解，不要紧，我们把刚刚生成的二进制代码文件反汇编来看看：

   ```
   ▷ ndisasmw -o 0x7c00 boot.bin >> disboot.asm
   ```

   打开 disboot.asm，你会发现这样一行：

   ```
   00007C09  EBFE                jmp short 0x7c09
   ```

   明白了吧，`$` 在这里的意思原来就是 0x7c09。

   那么 `$$` 表示什么呢？它表示一个节（section[10]）的开始处被汇编后的地址。在这里，我们的程序只有 1 个节，所以，`$$` 实际上就表示程序被编译后的开始地址，也就是 0x7c00。

在写程序的过程中，`$-$$`可能会被经常用到，它表示本行距离程序开始处的相对距离。现在，你应该明白`510-($-$$)`表示什么意思了吧？

`times 510-($-$$) db 0`表示将 0 这个字节重复`510-($-$$)`遍，也就是在剩下的空间中不停地填充 0，直到程序有 510 字节为止。这样，加上结束标志 0xAA55 占用的 2 字节，恰好是 512 字节。

[10]注意：这里的 section 属于 NASM 规范的一部分，表示一段代码，关于它和 `$$` 更详细的注解请参考 NASM 联机技术文档。

1.5 水面下的冰山

即便是非常袖珍的程序，也有可能遇到不能正确运行的情况，对此你一定并不惊讶，谁都可能少写一个标点，或者在一个小小的逻辑问题上犯迷糊。好在我们可以调试，通过调试，可以发现错误，让程序日臻完美。但是对于操作系统这样的特殊程序，我们没有办法用普通的调试工具来调试。可是，哪怕一个小小的引导扇区，我们也没有十足的把握一次就写好，那么，遇到不能正确运行的时候该怎么办呢？在屏幕上没有看到我们所要的东西，甚至于机器一下子重启了，你该如何是好呢？

每一个问题都是一把锁，你要相信，世界上一定存在一把钥匙可以打开这把锁。你也一定能找到这把钥匙。

一个引导扇区代码可能只有 20 行，如果 Copy&Paste 的话，10 秒钟就搞定了，即便自己敲键盘抄一遍下来，也用不了 10 分钟。可是，在遇到一个问题时，如果不小心犯了小错，自己到运行时才发现，你可能不得不花费 10 个 10 分钟甚至更长时间来解决它。笔者把这 20 行的程序称做水面以上的冰山，而把你花了数小时的时间做的工作称做水面下的冰山。

古人云：“授之以鱼，不如授之以渔。”本书将努力将冰山下的部分展示给读者。这些都是笔者经历了痛苦的摸索后的一些心得，这些方法可能不是最好的，但至少可以给你提供一个参考。

好了，就以我们刚刚完成的引导扇区为例，你可以想像得到，将来我们一定会对这 20 行进行扩充，最后得到 200 行甚至更多的代码，我们总得想一个办法，让它调试起来容易一些。

其实很容易，因为有 Bochs，我们前面提到的虚拟机，它本身就可以作为调试器使用[11]。请允许我再次卖一个关子，留待下文分解。但是请相信，调试一个引导扇区——甚至是一个操作系统，在工具的帮助下都不是很困难的事情。只不过跟调试一个普通的应用程序相比，细节上难免有一些不同。

如果你使用的是 Windows，还有个可选的方法能够帮助调试，做法也很简单，就是把“`org 07c00h`”这一行改成“`org 0100h`”就可以编译成一个.COM 文件让它在 DOS 下运行了。我们来试一试，首先把`07c00h`改成`0100h`，编译：

```
▷ nasm boot.asm -o boot.com
```

好了，一个易于执行和调试的引导扇区就制作完毕了。如果你是以 DOS 或者 Windows 为平台学习的编程，那么调试.COM 文件一定是你拿手的工作，比如使用 Turbo Debugger。

调试完之后要放到软盘上进行试验，我们需要再把`0100h`改成`07c00h`，这样改来改去比较麻烦，好在强大的 NASM 给我们提供了预编译宏，把原来的“`org 07c00h`”一行变成许多行即可：

代码 1.2 chapter1/b/boot.asm

```
1 ;%define _BOOT_DEBUG_            ; 制作 Boot Sector 时一定将此行注释掉!
2                                  ; 去掉此行注释后可做成.COM文件易于调试:
```

[11] 如果你实在性急，请翻到第12页第2.1.4节看一下具体怎么调试。

```
3                                   ;   nasm Boot.asm -o Boot.com
4
5   %ifdef  _BOOT_DEBUG_
6           org  0100h              ; 调试状态, 做成 .COM 文件, 可调试
7   %else
8           org  07c00h             ; BIOS 将把 Boot Sector 加载到 0:7C00 处
9   %endif
```

这样一来，如果我们想要调试，就让第一行有效，想要做引导扇区时，将它注释掉就可以了。

这里的预编译命令跟 C 语言差不多，就不用多解释了。

至此，你不但已经学会了如何写一个简单的引导扇区，更知道了如何进行调试。这就好比从石器时代走到了铁器时代，宽阔的道路展现在眼前，运用工具，我们有信心将引导扇区不断扩充，让它变成一个真正的操作系统的一部分。

1.6 回顾

让我们再回过头看看刚才那段代码吧，大部分代码你一定已经读懂了。如果你还是一个 NASM 新手，可能并不是对所有的细节都那么清晰。但是，毕竟你已经发现，原来可以如此容易地迈出写操作系统的第一步。

是啊，这是个并不十分困难的开头，如果你也这样认为，就请带上百倍的信心，以及一直以来想要探索 OS 奥秘的热情，随我一起出发吧！

They who can give up essential liberty to obtain a little temporary safety, deserve neither liberty nor safety.

—— *Benjamin Franklin*

2 搭建你的工作环境

我知道，现在你已经开始摩拳擦掌准备大干一场了，因为你发现，开头并不是那么难的。你可能想到了 Linus，或许他在写出第一个引导扇区并调试成功时也是同样的激动不已；你可能在想，有一天，我也要写出一个 Linux 那样伟大的操作系统！是的，这一切都有可能，因为一切伟大必定是从平凡开始的。我知道此刻你踌躇满志，已经迫不及待要进入操作系统的殿堂。

可是先不要着急，古人云：“工欲善其事，必先利其器”，你可能已经发现，如果每次我们编译好的东西都要写到软盘上，再重启计算机，不但费时费力，对自己的爱机简直是一种蹂躏。你一定不会满足于这样的现状，还好，我们有如此多的工具，比如前面提到过的 Bochs。

在介绍 Bochs 及其他工具之前，需要说明一点，这些工具并不是不可或缺的，介绍它们仅仅是为读者提供一些可供选择的方法，用以搭建自己的工作环境。但是，这并不代表这一章就不重要，因为得心应手的工具不但可以愉悦身心，并且可以起到让工作事半功倍的功效。

下面就从 Bochs 开始介绍。

2.1 虚拟计算机 Bochs

即便没有听说过虚拟计算机，你至少应该听说过磁盘映像。如果经历过 DOS 时代，你可能就曾经用 HD-COPY 把一张软盘做成一个 .IMG 文件，或者把一个 .IMG 文件恢复成一张软盘。虚拟计算机相当于此概念的外延，它与映像文件的关系就相当于计算机与磁盘。简单来讲，它相当于运行在计算机内的小计算机。

2.1.1 Bochs 初体验

我们先来看看 Bochs 是什么样子的，请看图2.1和图2.2这两个屏幕截图。

要看清楚哦，你看到的不是显示器，仅仅是窗口而已。如果你是第一次接触“虚拟机”这个东西的话，一定会感到很惊讶，你会惊叹：“啊，像真的一样！”没错，像真的一样，不过窗口的标题栏一行“Bochs x86-64 emulator”明白

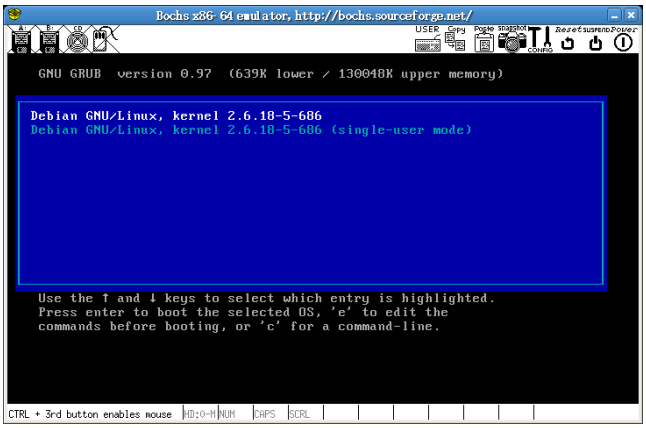

图 2.1 Bochs 中的 grub

图 2.2 Bochs 中的 Linux

无误地告诉我们，这仅仅是个“emulator”——模拟器而已。在本书中我们把这种模拟器称为虚拟机，因为这个词使用得更广泛一些。不管是模拟还是虚拟，我们要的就是它，有了它，我们不再需要频繁地重启计算机，即便程序有严重的问题，也丝毫伤害不到你的爱机。更加方便的是，你可以用这个虚拟机来进行操作系统的调试，在它面前，你就好像是上帝，你可以随时让时间停住，然后钻进这台计算机的内部，CPU 的寄存器、内存、硬盘，一切的一切都尽收眼底。这正是进行操作系统的开发实验所需要的。

好了，既然 Bochs 这么好，我们就来看看如何安装，以及如何使用。

2.1.2 Bochs 的安装

就像大部分软件一样，在不同的操作系统里面安装 Bochs 的过程是不同的，在 Windows 中，最方便的方法就是从 Bochs 的官方网站获取安装程序来安装

（安装时不妨将“DLX Linux Demo”选中，这样你可以参考它的配置文件）。在 Linux 中，不同的发行版（distribution）处理方法可能不同。比如，如果你用的是 Debian GNU/Linux 或其近亲（比如 Ubuntu），可以使用这样的命令：

```
▷ sudo apt-get install vgabios bochs bochs-x bximage
```

敲入这样一行命令，不一会儿就装好了。

很多 Linux 发行版都有自己的包管理机制，比如上面这行命令就是使用了 Debian 的包管理命令，不过这样安装虽然省事，但有个缺点不得不说，就是默认安装的 Bochs 很可能是没有调试功能的，这显然不能满足我们的需要，所以最好的方法还是从源代码安装，源代码同样位于 Bochs 的官方网站[1]，假设你下载的版本是 2.3.5，那么安装过程差不多是这样的：

```
▷ tar vxzf bochs-2.3.5.tar.gz
▷ cd bochs-2.3.5
▷ ./configure --enable-debugger --enable-disasm
▷ make
▷ sudo make install
```

注意“./configure”之后的参数便是打开调试功能的开关。在安装过程中，如果遇到任何困难，不要惊慌，其官方网站上有详细的安装说明。

2.1.3 Bochs 的使用

好了，Bochs 已经安装完毕，是时候来揭晓第 1 章的谜底了，下面我们就一步步来说明图1.1的画面是怎样来的。

在第 1 章我们提到过，硬件方面需要的是一台计算机和一张空白软盘，现在计算机有了——就是刚刚安装好的 Bochs，那么软盘呢？既然计算机都可以“虚拟”，软盘当然也可以。在刚刚装好的 Bochs 组件中，就有一个工具叫做 bximage，它不但可以生成虚拟软盘，还能生成虚拟硬盘，我们也称它们为磁盘映像。创建一个软盘映像的过程如下所示：

```
▷ bximage
========================================================================
                                bximage
                  Disk Image Creation Tool for Bochs
        $Id: bximage.c,v 1.32 2006/06/16 07:29:33 vruppert Exp $
========================================================================

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] fd ↩

Choose the size of floppy disk image to create, in megabytes.
Please type 0.16, 0.18, 0.32, 0.36, 0.72, 1.2, 1.44, 1.68, 1.72, or 2.88.
 [1.44] ↩
I will create a floppy image with
  cyl=80
  heads=2
  sectors per track=18
  total sectors=2880
  total bytes=1474560
```

[1]实际上通过命令行也可以获取源代码，只不过通常不是最新的，在此不做介绍。

```
What should I name the image?
[a.img] ↩

Writing: [] Done.

I wrote 1474560 bytes to a.img.

The following line should appear in your bochsrc:
  floppya: image="a.img", status=inserted
```

凡是有↩记号的地方，都是 bximage 提示输入的地方，如果你想使用默认值，直接按回车键就可以。在这里我们只有一个地方没有使用默认值，就是被问到创建硬盘还是软盘映像的时候，我们输入了“fd”。

完成这一步骤之后，当前目录下就多了一个 a.img，这便是我们的软盘映像了。所谓映像者，你可以理解为原始设备的逐字节复制，也就是说，软盘的第M个字节对应映像文件的第M个字节。

现在我们已经有了“计算机”，也有了“软盘”，是时候将引导扇区写进软盘了。我们使用 dd 命令[2]：

```
▷ dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
```

注意这里多用了一个参数“conv=notrunc”，如果不用它的话软盘映像文件 a.img 会被截断（truncated），因为 boot.bin 比 a.img 要小。第 1 章中我们使用这个命令时不需要此参数，因为真实的软盘不可能被“截断”——真的和假的总是会有一点区别。

现在一切准备就绪，该打开电源启动了。可电源在哪儿呢？不要慌，我们还剩一样重要的东西没有介绍，那就是 Bochs 的配置文件。为什么要有配置文件呢？因为你需要告诉 Bochs，你希望你的虚拟机是什么样子的。比如，内存多大啊、硬盘映像和软盘映像都是哪些文件啊等内容。不用怕，这配置文件也没什么难的，代码2.1就是一个 Linux 下的典型例子。

代码 2.1 bochsrc 示例

```
1  ###############################################################
2  # Configuration file for Bochs
3  ###############################################################
4
5  # how much memory the emulated machine will have
6  megs: 32
7
8  # filename of ROM images
9  romimage: file=/usr/share/bochs/BIOS-bochs-latest
10 vgaromimage: file=/usr/share/vgabios/vgabios.bin
11
12 # what disk images will be used
13 floppya: 1_44=a.img, status=inserted
14
15 # choose the boot disk.
16 boot: floppy
17
18 # where do we send log messages?
19 log: bochsout.txt
20
```

[2]如果你用 Windows，那么使用 Linux 常用命令需要额外一些劳动，比如安装一个 Cygwin，或者下载某个工具的 Windows 版本。在这里你可以简单下载一个“dd for Windows”，其下载地址为http://www.chrysocome.net/dd。

```
# disable the mouse
mouse: enabled=0

# enable key mapping, using US layout as default.
keyboard_mapping: enabled=1, map=/usr/share/bochs/keymaps/x11-pc-us.map
```

可以看到，这个配置文件本来就不长，除去注释之后内容就更少了，而且很容易理解，字面上稍微不容易理解的只有 romimage 和 vgaromimage[3]，它们指定的文件对应的其实就是真实机器的 BIOS 和 VGA BIOS，读者自己操作的时候要确保它们的路径是正确的，不然过一会儿虚拟机启动时可能会被提示“couldn’t open ROM image file”。读者还要注意 floppya 一项，它指定我们使用哪个文件作为软盘映像。

如果你在 Windows 下的话，romimage 和 vgaromimage 两项指定的文件应该是安装目录下的BIOS-bochs-latest和VGABIOS-lgpl-latest。当然，最保险的方法是参考安装程序自带的 DLX linux 的配置文件，将其稍作修改即可。

好了，现在一切准备就绪，是时候启动了，输入命令：

```
▷ bochs -f bochsrc
```

一个回车[4]，你想要的画面就呈现在眼前了。是不是很有趣呢？

顺便告诉你个窍门，如果你输入一个不带任何参数的 Bochs 并执行之，那么 Bochs 将在当前目录顺序寻找以下文件作为默认配置文件：

- .bochsrc
- bochsrc
- bochsrc.txt
- bochsrc.bxrc （仅对 Windows 有效）

所以刚才我们的“`-f bochsrc`”参数其实是可以省略的。读者在给配置文件命名时不妨从这些文件里选一个，这样可以省去许多输入命令的时间。

此外，Bochs 的配置文件还有许多其他选项，读者如果想详细了解的话，可以到其主页上看一看。由于本书中所用到的选项有限，在此不一一介绍。

2.1.4 用 Bochs 调试操作系统

如果单是需要一个虚拟机的话，你有许许多多的选择，本书下文也会对其他虚拟机有所介绍，之所以 Bochs 称为我们的首选，最重要的还在于它的调试功能。

假设你正在运行一个有调试功能的 Bochs，那么启动后，你会看到控制台出现若干选项，默认选项为“6. Begin simulation”，所以直接按回车键，Bochs 就

[3]Bochs 使用的 vgaromimage 来自于 vgabios 项目，如果读者感兴趣，可以去它的主页看看：http://www.nongnu.org/vgabios/。

[4]如果你正在使用的是自己编译的有调试功能的 Bochs，回车后还需要再一次回车，并在出现 Bochs 提示符之后输入“c”，再次回车。不要被这些输入吓怕了，下文有妙计可以让你不必总是这么辛苦。

启动了，不过既然是可调试的，Bochs 并没有急于让虚拟机进入运转状态，而是继续出现一个提示符，等待你的输入，这时，你就可以尽情操纵你的虚拟机了。

还是以我们那个最轻巧的引导扇区为例，假如你想让它一步步地执行，可以先在07c00h处设一个断点——引导扇区就是从这里开始执行的，所以这里就是我们的入口地址——然后单步执行，就好像所有其他调试工具一样。在任何时刻，你都可以查看 CPU 寄存器，或者查看某个内存地址处的内容。下面我就来模拟一下这个过程：

```
... ... ...
Next at t=0
(0) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:1> b 0x7c00↩
<bochs:2> c↩
(0) Breakpoint 1, 0x00007c00 in ?? ()
Next at t=886152
(0) [0x00007c00] 0000:7c00 (unk. ctxt): mov ax, cs                ; 8cc8
<bochs:3> dump_cpu↩
eax:0x0fffaa55, ebx:0x00000000, ecx:0x00120001, edx:0x00000000
ebp:0x00000000, esp:0x0000fffe, esi:0x000088d2, edi:0x0000ffde
eip:0x00007c00, eflags:0x00000282, inhibit_mask:0
cs:s=0x0000, dl=0x0000ffff, dh=0x00009b00, valid=1
ss:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=7
ds:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
es:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
fs:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
gs:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
ldtr:s=0x0000, dl=0x0000ffff, dh=0x00008200, valid=1
tr:s=0x0000, dl=0x0000ffff, dh=0x00008300, valid=1
gdtr:base=0x00000000, limit=0xffff
idtr:base=0x00000000, limit=0xffff
dr0:0x00000000, dr1:0x00000000, dr2:0x00000000
dr3:0x00000000, dr6:0xffff0ff0, dr7:0x00000400
cr0:0x00000010, cr1:0x00000000, cr2:0x00000000
cr3:0x00000000, cr4:0x00000000
done
<bochs:4> x /64xb 0x7c00↩
[bochs]:
0x00007c00 <bogus+   0>:  0x8c  0xc8  0x8e  0xd8  0x8e  0xc0  0xe8  0x02
0x00007c08 <bogus+   8>:  0x00  0xeb  0xfe  0xb8  0x1e  0x7c  0x89  0xc5
0x00007c10 <bogus+  16>:  0xb9  0x10  0x00  0xb8  0x01  0x13  0xbb  0x0c
0x00007c18 <bogus+  24>:  0x00  0xb2  0x00  0xcd  0x10  0xc3  0x48  0x65
0x00007c20 <bogus+  32>:  0x6c  0x6c  0x6f  0x2c  0x20  0x4f  0x53  0x20
0x00007c28 <bogus+  40>:  0x77  0x6f  0x72  0x6c  0x64  0x21  0x00  0x00
0x00007c30 <bogus+  48>:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00007c38 <bogus+  56>:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
<bochs:5> n↩
Next at t=886153
(0) [0x00007c02] 0000:7c02 (unk. ctxt): mov ds, ax                ; 8ed8
<bochs:6> trace-reg on↩
Register-Tracing enabled for CPU 0
<bochs:7> n↩
Next at t=886154
eax: 0x0fff0000 268369920
ecx: 0x00120001 1179649
edx: 0x00000000 0
ebx: 0x00000000 0
esp: 0x0000fffe 65534
ebp: 0x00000000 0
esi: 0x000088d2 35026
edi: 0x0000ffde 65502
eip: 0x00007c04
eflags 0x00000282
IOPL=0 id vip vif ac vm rf nt of df IF tf SF zf af pf cf
```

```
(0) [0x00007c04] 0000:7c04 (unk. ctxt): mov es, ax                ; 8ec0
<bochs:8> c↩
.........
```

以上带有↩符号并以加粗字体显示的是输入，其他均为 Bochs 的输出。如果你用过 GDB，你会觉得这个过程很亲切。没错，它跟用 GDB 调试程序的感觉是很相似的，最大的区别可能就在于在 Bochs 的调试模式下我们需要跟 CPU、内存、机器指令等内容打更多交道。

在上面的演示过程中，最开始的“b 0x7c00”在0x7c00处设置了断点，随后的命令“c”让代码继续执行，一直到我们设置的断点处停止，然后演示的是用“dump_cpu”指令查看 CPU 寄存器以及用“x”指令查看内存。随后用一个“n”指令让代码向下走了一步，“trace-reg on”的功能是让 Bochs 每走一步都显示主要寄存器的值。之所以选择演示这些命令，因为它们基本是调试过程中最常用到的。

如果你在调试过程中忘记了指令的用法，或者根本就忘记了该使用什么指令，可以随时使用help命令，所有命令的列表就呈现在眼前了。你将会发现 Bochs 的调试命令并不多，不需要多久就可以悉数掌握。表2.1列出了常用的指令以及其典型用法。

表 2.1 部分 Bochs 调试指令

行为	指令	举例
在某物理地址设置断点	b addr	b 0x30400
显示当前所有断点信息	info break	info break
继续执行，直到遇上断点	c	c
单步执行	s	s
单步执行（遇到函数则跳过）	n	n
查看寄存器信息	info cpu r fp sreg creg	info cpu r fp sreg creg
查看堆栈	print-stack	print-stack
查看内存物理地址内容	xp /nuf addr	xp /40bx 0x9013e
查看线性地址内容	x /nuf addr	x /40bx 0x13e
反汇编一段内存	u start end	u 0x30400 0x3040D
反汇编执行的每一条指令	trace-on	trace-on
每执行一条指令就打印 CPU 信息	trace-reg	trace-reg on

其中“xp /40bx 0x9013e”这样的格式可能显得有点复杂，读者可以用“help x”这一指令在 Bochs 中亲自看一下它代表的意义。

好了，虽然你可能还无法熟练运用 Bochs 进行调试，但至少你应该知道，即便你的操作系统出现了问题也并不可怕，有强大的工具可以帮助你进行调试。由于 Bochs 是开放源代码的，如果你愿意，你甚至可以通过读 Bochs 的源代码来间接了解计算机的运行过程——因为 Bochs 就是一台计算机。

2.2 QEMU

如果你选择在 Linux 下开发，其实 Bochs 自己就完全够用了。在本书中，今后的大部分例子均使用 Bochs 作为虚拟机来运行。如果你在 Windows 下开发的话，或许你还需要一个运行稍微快一点的虚拟机，它不是用来运行我们自己的操作系统的，而是用来运行 Linux 的，因为我们的代码是用 Linux 来编译的，并且生成的内核代码是 ELF 格式的。

之所以不用 Bochs 来装一个 Linux，是因为 Bochs 速度比较慢，这是由它的运行机制决定的，它完全模拟硬件及一些外围设备，而很多其他虚拟机大都采用一定程度的虚拟化（Virtualization）技术[5]，使得速度大大提高。

我以 QEMU 为例介绍速度较快的虚拟机，但它绝不是唯一的选择，除它之外，Virtual Box、Virtual PC、VM Ware 等都是很有名气的虚拟机。它们各有自己的优缺点，QEMU 的显著优势是它可以模拟较多的硬件平台，这对于一个操作系统爱好者而言是很具有吸引力的。

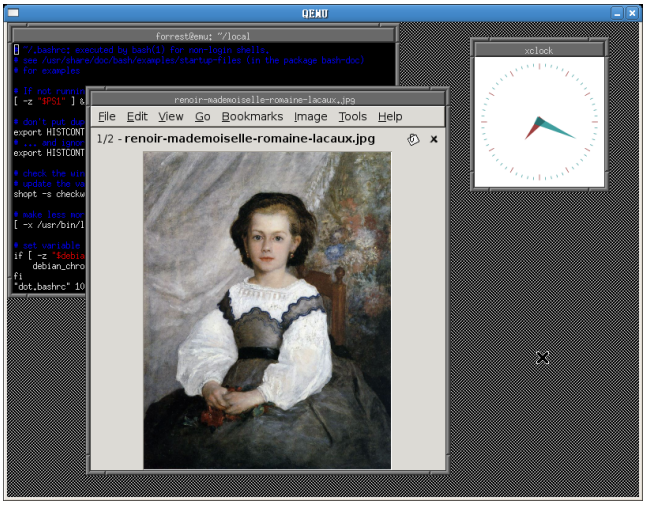

图 2.3 QEMU 中的 Linux

好了，百闻不如一见，图2.3就是 QEMU 上运行 Linux 的抓图。这里我安装的是 Debian 4.0，窗口管理器用的是 Fvwm。在我并不强大的个人电脑上，安装整个系统也没用很久，运行起来也相当顺畅。

与 Bochs 使用配置文件不同，QEMU 运行时需要的参数是通过命令行指定的[6]。比如，如果要用 QEMU 来引导我们之前已经做好的虚拟软盘，可以这样做：

```
▷ qemu -fda a.img
```

[5]读者如果对这一技术感兴趣，可在网上搜索相应资料，比如维基百科上就有个大致的介绍：http://en.wikipedia.org/wiki/Full_virtualization。

[6]实际上 Bochs 也可以用命令行指定参数，详见 Bochs 联机手册。

看起来比 Bochs 还要清爽，不是吗？其实如果你不需要调试的话，在 Linux 下也可以用 QEMU 来运行你的操作系统，这样每次都可以更快地看到结果——这便是我之前说过的“可以让你不必总是这么辛苦”的“妙计”了[7]。

2.3 平台之争：Windows 还是 *nix

读到这里，读者可能发现书中经常出现“如果你用的是 Windows”或者“如果你用的是 Linux”这样的字眼。有时这样的字眼甚至可能影响到你的阅读，如果真的这样请你原谅。我试图照顾尽量多的读者，但是对每一个人来讲，却必须面临一个选择——在什么平台下开发。本书第一版使用的是 Windows 平台，而在第二版中，我投诚了。接下来你会发现，虽然以后的行文会最大限度地兼顾 Windows，但总体是以 Linux 为默认平台的。

其实在什么平台下开发，有时纯粹是口味问题，或者是环境问题——你开始接触计算机时使用什么，很大程度上取决于你周围的人使用什么，而这往往对你的口味产生巨大而深远的影响。然而最早接触的未必是最适合的，在我亲身体会和比较之后，我决定从 Windows 彻底换到 Linux，我想在这里说说为什么。请注意这不是布道会，更不是你开发自己的操作系统必须阅读的章节，我仅仅是谈谈我个人的体会，希望能对你有所启发，同时解释一下为什么第二版会有这样的改动。

在第一版成书的时候，我已经在使用 Linux，但是用得并不多，主要是觉得用不习惯，而现在过了两三年，我已经基本不用 Windows，在 Windows 下我会觉得很不习惯。我的这一经历至少有两点启示：第一是 Linux 不好用是个误解（有一种说法是 Windows 的桌面更好用，这是个复杂的误解），好不好用是习惯问题；第二是如果你有兴趣使用一样你不熟悉的东西，不要因为刚开始的不习惯而放弃。

其实对于 Linux 和 Windows 的误解有很多，我把这种误解归结为操作系统文化上的差异。其实在提起两种系统时，人们往往拿一些具体的事情来做比较。比如比较它们的安装过程、使用方法，甚至是界面。但实际上隐藏在表面背后的是两种完全不同类型的文化，或者称之为不同的理念。

对于 Windows 而言，它的文化植根于微软公司的愿景，“让每个家庭的每个桌面上都有一台电脑”，当然他们希望此电脑内运行的是 Windows 操作系统。这个理想加上 Windows 作为商业软件的性质，决定了 Windows 具有相当程度的亲和力，用户界面显得相当友好。岂止友好，它简直友好到每个人——无论儿童还是老人，受过高等教育还是只念过小学——都能比较容易地开始使用电脑，这无疑是微软对这个社会的巨大贡献。但是界面友好并不一定就完美了，这一点暂且按下不表，我们先来说说 Linux。

Linux 的文化很大程度上来源于 UNIX，UNIX 所倡导和遵循的文化也被称为 UNIX 哲学[8]，其中很重要的一条原则叫做“做一件事并做好”[9]，这听上去

[7]其实妙计不止一条，你也可以在系统内安装两种 Bochs，一种是打开调试功能的，一种是没有打开的，你可以自由选择运行哪一种。

[8]简单的介绍可参见http://en.wikipedia.org/wiki/Unix_philosophy；若想较全面地了解，建议读者阅读 Eric S. Raymond 所著的《UNIX 编程艺术》。

[9]原文作“Do one thing, do it well”。理解这一原则的内涵及外延是理解 UNIX 世界的基本条件。

跟 Windows 的界面友好说的不是一码事，但其实仔细分析起来大有关联。做一件事并做好意味着两件事情，第一件事就是工具之间可以协同作战，不然各人做各人的，无法完成复杂应用；第二件事就是接口要统一，不然无法做到协同。这个统一的接口就是文本流（text stream），这也就意味着，命令行是 UNIX 文化的核心。而 Windows 的做法大有不同，因为要界面友好，于是不能指望用户开始就知道怎么把工具串接在一起，所以 Windows 选择任何应用都自己完成所有功能——至少让用户看起来如此，这使得每个工具都各自为战，从而增加了每个程序的复杂性和开发成本。不仅如此，由于功能都是软件开发者定好的，所以你基本上不能指望大部分的程序具备可扩展性，而在 UNIX 下，大部分的程序都可以跟其他程序协同起来完成程序不曾“设计”的功能。这也是上文我说“界面友好并不一定完美”的原因，友好是有代价的。

那么 UNIX 是一个“不友好”的系统吗？这个问题其实没有看起来那么简单。首先是 UNIX 下流行的桌面环境正在越来越“友好”，你甚至可以将其配置得看上去跟 Windows 别无二致，不过关键点不在于此，而在于长期来看，UNIX 的学习成本并不比 Windows 要高，但收益却要高得多。我们刚刚提到，友好是有代价的，而且代价比想像中要高。对于一个初学者，开始的简单容易使他产生错觉，认为电脑是个简单器械，但实际情况并非如此，一旦遇到麻烦，用户很容易陷入束手无策的境地，一旦有一件事情没有现成的软件可以解决，你马上一筹莫展。而 UNIX 不同，它的学习曲线比较陡峭，但是你一旦入门，就会发现自己的工作可以变得如此轻松而且有趣。在 Windows 中，虽然使用一个工具第一步往往很容易，但很快你就容易迷失在一堆嵌套很深名字晦涩的菜单里面，学习这些菜单可不是一件容易的事情，而且在一个工具里学会的东西到了另一个工具里可能就变了样。如果你想看看程序的帮助，有时也是件困难的事情，因为为了达到“友好”的效果，帮助经常也是一层一层的，很难找到自己需要的内容。而在 UNIX 中，所有的工具都有个手册（Manual），可以通过统一的命令“man”来查看，而且这些手册都是平坦的，你可以一口气从头看到尾，可以随时查看你所要的关键字。此外，除了少数极其复杂的工具，手册基本上是够用的。简而言之，在 UNIX 中，软件使用看起来复杂了，实际上如果你想真正掌握一个东西，用的时间不会比 Windows 中更多。况且，在 Windows 中你很难真正掌握一个东西。

我并非故意贬低 Windows，我说过它对社会的贡献巨大。对于一个平常只用电脑来收收邮件看看电影的用户，它的易用性绝对是巨大的优点，但你我不是这样的用户。我相信阅读本书的人都是程序员，而且都是像我一样喜欢探索的程序员——不喜欢探索的程序员很难有心思写自己的操作系统做消遣。一个程序员的要求和普通用户是不同的，程序员需要了解他的电脑，掌握它，并且可以熟练地让它帮助自己完成工作，从这个角度上讲，UNIX 无疑具有巨大的优势。它里面的每个工具都很锋利，你可以组合着使用，持久地使用，而且许多年都不会过时。

在这里我可以举一个我自己遇到过的例子。在我编写操作系统的文件系统时，需要多次查看某几个扇区的内容，并对其中的数据进行分析。在 Linux 中，我可以很容易地将 od、grep、sed 和 awk 等工具[10]串在一起完成这项工作，我也可以编写一个简单的脚本，将命令放在脚本中方便取用。而在 Windows 中，我

[10]这些都是 UNIX 下的常用工具，读者可以通过联机手册查看它们的用法。更多 UNIX 下的工具介绍可参考http://en.wikipedia.org/wiki/List_of_Unix_utilities。

通常只能在窗口间反复地单击鼠标，费时费力而且效率低下。类似的例子不胜枚举，你一旦熟悉了这些工具，就会发现通过组合它们，你能得到比任何图形界面工具都多的功能。而在 Windows 下就不得不看具体菜单的眼色了。不仅如此，UNIX 下的工具往往学习一次就能长久使用，很少过时。比如刚才提到的几个工具大部分有 20 年以上的历史，到现在它们依然被广泛使用，即便它们学起来会稍微难一点，平摊在 20 年里面，成本也是极低的。这就是 UNIX 哲学，你不需要重复学习，每个工具都好用，而且可以因为跟其他工具结合而发挥多种作用。

大多数 Windows 下的软件都有个毛病，它经常试图隐藏一些东西。它的本意是好的，就是让界面更“友好”，但这对程序员有时是件坏事，因为它让人难以透彻地理解软件的所作所为。或许你会说，如果你想理解，你总能理解的。没错，这跟“在 UNIX 下能做的事情在 Windows 下都能做”是相似的命题，甚至于，只要安装一些额外的软件（比如 Cygwin[11]），你可以在 Windows 下使用 UNIX 的命令。问题是即便能做，你也未必去做，这就是所谓文化的力量。理论上你在任何地方都能读书学习，但效率最高的地方还是教室和书房，在客厅舒服的沙发上，你不自主地就拿起了电视遥控器。

所以以我自己的体会而言，一个程序员最好还是使用类 UNIX 的操作系统。它能在日常生活中帮你提高自己的水平和工作效率。这一点与摄影有点类似，市面上数量最多的是傻瓜相机，但一个专业的摄影师总是会选择功能复杂的专业级设备，不是傻瓜相机不好，而是适应的人群不同，如果你想成为好的摄影师，那么上手容易的傻瓜相机一定不是你的最终选择。不是好不好的问题，是适不适合的问题。

上面论及的是两类操作系统文化上的差异，其实即便是纯粹应用层的，也有诸多误解，比如以下几条：

误解一. Linux 难安装。如果你曾被 Linux 的安装难倒过，我建议你下次试试 Ubuntu[12]。在本书第一版开始写作之时，Ubuntu 的第一个版本还没有发布[13]，但短短几年时间，它已经变成全世界最流行的发行版[14]，这与它的易装和易用性是分不开的。笔者本人大规模地使用 Linux 也是从 Ubuntu 开始的，它的安装过程一点也不比 Windows 的难，而且中文资料也相当丰富，很容易找到志同道合的人。Ubuntu 的另一特点是它的驱动程序很丰富，支持很多的硬件，大部分情况下驱动程序都能自动安装好，甚至不需要用户参与，在这一点上它甚至比 Windows 更“友好”。

误解二. Linux 难学。希望你永远记住，电脑不是个简单器械，无论是硬件、操作系统还是应用软件，都经常比看上去复杂得多。所以易用未必是好事，它肯定向你隐瞒了些什么，花一点时间绝对是值得的，尤其是当你想做一个程序员的时候。况且 Linux 也没那么难学，且不说它的图形界面越来越好用，就是完全用命令行的话，入门也相当容易。而且 Linux 世界的文档齐全且易于检索，更有高度发达的社区文化，在这里你学到的往往比预期的还要多。

误解三. Linux 难用。再强调一下，Linux 的学习曲线是陡峭的，然而一旦你度过了开始的适应期，就会发现原来命令行可以这么好用，原来有这么丰富的

[11] Cygwin 官方网站为 http://www.cygwin.com/。
[12] Ubuntu 官方网站为http://www.ubuntu.com/。
[13] Ubuntu 的第一个版本（代号 Warty Warthog）发行于 2004 年 10 月。
[14] 根据 2008 年 8 月的数据。及时情况可参考http://distrowatch.com/。

工具来提高效率，而且这些好用的工具居然都是自由的[15]！你不需要向作者付费，甚至他们鼓励你使用和传播，你甚至可以随便修改这些工具的源代码，遇到问题可以发一封邮件反馈给作者。这在 Windows 下都是很难做到的，在那片土地上你很难体会什么叫“自由”。不仅如此，当你熟悉之后会发现，同样一件事情，其实 Linux 下的解决方案往往比 Windows 下要简单。就比如我们提到过的安装 Bochs 一例吧，在 Windows 下你通常需要先到 Bochs 网站，在数次单击之后找到下载链接，然后下载，再然后是双击安装程序来安装。在 Linux 下呢，你看到了，只需要一个命令行就可以了，即便你打字的速度再慢，也比那些鼠标单击操作要快。

我在此并非贬低鼠标的好处，我每天都使用鼠标，它绝对是个伟大的发明。但是我们应该只在需要它的时候使用它，而不是试图用它来解决所有事情。这就好比图形界面是个好东西，如果你的工作是图形图像处理，很难想像没有图形界面该怎么做，但并不是图形界面在任何时候都是好的。我们应该分辨每一类工作最适合的工具是什么，而不是用一种思维解决所有问题。这也是我说“Windows 桌面好用是个复杂的误解”的原因，图形界面是个好东西，Windows 把它用得极端了。

误解四. Linux 下软件少。这是最大的一个误区，事实上很少有事情你在 Linux 下是做不到的。而且 Linux 的发行版通常都有发达的包管理工具，无论是关键字查找、安装、更新还是卸载都可以用一组统一的命令来完成。这使得你在需要某种软件时，使用简单的命令就可以找到它，很多时候你能找到不止一种。如果你想看看除了 od 之外还有哪些二进制查看器，在 Ubuntu 或者 Debian 下通过一个 `apt-cache search 'hex.*(view|edit)'`[16]命令就能找到十几种，这些都是自由软件，有命令行的也有图形界面的，而在 Windows 下，怕是又要在浩瀚的互联网中搜索了。

在 Linux 中找一些 Windows 下软件的替代品是很容易的，虽然这种对应有时并非必要。比如字处理软件就有 OpenOffice.org、KOffice、AbiWord 等选择；图像处理软件有 GIMP；多媒体播放软件有 MPlayer、Totem 等。如果你喜欢玩游戏，Linux 下的游戏数量也会让你大吃一惊，不信你可以来这里看一看：http://en.wikipedia.org/wiki/Category:Linux_games。

其实 Linux 的好处还远不止这些，众所周知的一个优点是它基本没有病毒的烦恼。不是 Linux 中开发不出病毒来[17]，而是因为 Linux 系统有自身的权限机制作保障，加上软件来源都可信赖，且大部分都是源代码开放的（其中相当一部分都是自由软件），所以说 Linux 下没有病毒烦恼并非夸张。想想你在与病毒做斗争的过程中浪费了多少时间吧，我已经很久没有这种烦恼了。Windows 或许正变得越来越稳定，但 Linux 一直都很稳定，而且你不需要整天重启你的电脑，笔者的电脑就有时几十天不重启。除非你要升级内核，否则没有很多关机和重启的理由（不管是安装还是卸载，或是对系统内的包进行升级，都不需要重启电脑）。

作为一个操作系统爱好者，使用 Linux 的理由还有一条，那就是 Linux 的内核是“自由”的，注意它不仅仅是“开放源代码”的，你不仅可以获取其源代

[15]注意这里没用“免费”这个词。Free Software 的 Free 是“自由”之意，它比“免费”一词包含了更多意义。欲获得更详细的内容请访问http://www.fsf.org/。

[16]apt-cache 是个 Debian 家族中常用的包管理命令，可以使用正则表达式来搜索软件包。

[17]关于 Linux 系统下的病毒，读者可以参考：http://en.wikipedia.org/wiki/Linux_malware。

码，而且可以自由地复制、修改、传播它，当然也包括学习它。如果你也想加入到内核黑客[18]的队伍，那么就先从使用它开始吧。Linux 不是完美的，它的问题有很多，但每个问题都是你参与的机会，而这种参与可能是你成为顶尖高手的开始。

笔者本人完全使用 Linux 来工作的时间其实很短，几年而已，但我已经深深体会到它给我带来的好处。我并非想说服你，人不能被说服，除非他自己愿意相信。我只是希望你能尝试着去用一用 Linux，或者 UNIX 的其他变种，然后用自己的判断去选择。

如果你坚持使用 Windows，没问题，两者之中都可以很容易地搭建起开发环境，本章后面的部分将会就 Linux 和 Windows 分别来做介绍。

2.4 GNU/Linux 下的开发环境

在工作环境中，虚拟机是个重头戏，所以在本章的前面单独做了介绍。除了虚拟机之外，还有几样重要的东西，分别是编辑器、编译器和自动化工具 GNU Make。

许多在 Linux 下工作的人会使用 Vi 或者 Emacs 作为编辑器。如果你有兴趣尝试，那么还是那句建议，“不要因为刚开始的不习惯而放弃”，因为它们的确是编辑器中的经典，而且和 Linux 一样，具有陡峭的学习曲线。许多人一旦学会使用就爱上它们，这其中也包括笔者自己。当然，学习它们并不是必需的，而且你的选择范围比操作系统要大多了，相信会有一款能让你满意。

对于编译器，我们选择 GCC 和 NASM 分别来编译 C 代码和汇编代码。选择 GCC 的原因很简单，它是 Linux 世界编译器的事实标准。GCC 的全称是 GNU Compiler Collection，在这里我们只用到其中的 C 编译器，所以对我们而言它的全部意义仅为 GNU C Compiler——这也正是它原先的名字。之前提到过，我们使用的 Linux 其实应该叫做 GNU/Linux，所以使用 GCC 是比较顺理成章的，那么为什么不能使用 GCC 来编译我们的汇编代码呢？何苦再用个 NASM 呢？原因在于 GCC 要求汇编代码是 AT&T 格式的，它的语法对于习惯了 IBMPC 汇编的读者而言会显得很奇怪，我猜大部分读者可能都跟我一样，学习汇编语言时使用的教材里介绍的是 IBMPC 汇编。NASM 的官方网站位于http://nasm.sourceforge.net/，你还可以在上面找到详细的文档资料。

关于 GNU Make 的介绍见本书第 5 章。

还是以 Debian 作为示例，安装 GCC 和 NASM 可以通过以下命令来完成：

```
▷ sudo apt-get install build-essential nasm
```

注意这里的 build-essential 软件包中包含 GCC 和 GNU Make。

好了，现在可以总结一下了，如果你想要搭建一个基于 Linux 的开发环境，那么你需要做的工作有以下这些：

- 安装一个 Linux 发行版，如果你对 Linux 不甚熟悉，推荐使用 Ubuntu。

[18] 如果你对成为黑客感兴趣，或许可以读一读 Eric S. Raymond 的“How To Become A Hacker”。

- 通过 Linux 发行版的包管理工具或者通过下载源代码手工操作的方式来安装以下内容：
  - 一个你喜欢的编辑器，比如 Emacs。
  - 用于编译 C 语言代码的 GCC。
  - 用于编译汇编代码的 NASM。
  - 用于自动化编译和链接的 GNU Make。
  - 一个用于运行我们的操作系统的虚拟机，推荐使用 Bochs。

再次强调，如果你在安装或使用它们时遇到困难，不要着急，也不要气馁，因为一帆风顺的情形在现实生活中着实很少见。你或许可以试试以下的解决方案（这些方法也适用于其他在自由软件的安装配置及使用等方面的问题）：

- 向身边的朋友求助。
- 使用搜索引擎看看是不是有人遇到类似的问题，那里或许已经给出解决方案。
- 仔细阅读相应资料（不要怕英文），比如安装说明，或是 FAQ。
- 订阅相应的邮件列表（Mailing List），只要能将问题描述清楚[19]，通常你能在几小时内得到答复。
- 到论坛提问。
- 如果实在是疑难杂症，你可以试着联系软件的开发者，通常也是通过邮件列表的方式（同一个项目可能有多个邮件列表，开发者邮件列表通常与其他分离）。
- 自己阅读源代码并独立解决——这或许是个挑战，然而一旦解决了问题，你将获得知识、经验以及成功的喜悦。

将来，如果一切顺利的话，你编写操作系统时的步骤很可能是这样的：

1. 用编辑器编写代码。
2. 用 Make 调用 GCC、NASM 及其他 Linux 下的工具来生成内核并写入磁盘映像。
3. 用 Bochs 来运行你的操作系统。
4. 如果有问题的话
   (a) 用各种方法来调试，比如用 Bochs；
   (b) 返回第 1 步。

[19]关于提问的技巧，请参考 Eric S. Raymond 的“How To Ask Questions The Smart Way”。

2.5 Windows 下的开发环境

我们在介绍 QEMU 时提到过，在 Windows 下你需要一个虚拟的 Linux 来帮你编译操作系统的源代码。将操作系统内核编译链接成 ELF 格式有诸多好处，我们不但可以用 Linux 下现成的工具[20]来分析编译好的内核，还可以在必要时参考 Linux 内核的源代码来帮助我们自己的开发，总之这拉近了我们与 Linux 之间的距离。所以不要因为在 Windows 下也离不开 Linux 这件事而沮丧，况且装一个 Linux 是件很容易的事情。

不过装一个*虚拟的* Linux 跟装一个*真实的* Linux 还是有所不同，主要在于两点。一是我们仅仅想用这个 Linux 来做编译链接的工作，所以在选择组件的时候尽量去除不必要的内容，这样可以节省时间和空间；二是要确保你选择的虚拟机容易跟宿主机进行网络通信，因为你需要将宿主机上的源代码拿给虚拟机来编译。

安装方法可以有多种选择，比较简单的方法是通过光盘安装，当然这个光盘也可以是“虚拟”的，也就是一个光盘映像。首先到你所中意的 Linux 发行版的官方网站下载一个安装光盘的映像，有些发行版还提供免费或付费的邮寄服务，读者可以根据自己的喜欢自行选择。这里假设你得到的是光盘映像，文件名为 inst.iso。

有了光盘映像，我们还缺少一个硬盘映像，读者可以用前文提到过的 bximage 来生成它，也可以使用下面的命令：

```
▷ qemu-img create hd.img 1500M
```

这样就能生成一个大小约为 1.5GB 的硬盘映像了。

接下来就可以进行安装了：

```
▷ qemu -cdrom inst.iso -hda hd.img -boot d
```

安装过程从略，注意尽量精简你的组件，不要安装太多无用的东西。这些组件对我们是必需的：GCC、GNU Make、NASM、Samba。如果它们在安装时默认没有装上，那么你需要在系统安装结束后将它们安装上。由于目前大多数虚拟机都具有好用的网络功能，所以安装它们并非难事。

装完之后，我们还需要解决让宿主机和虚拟机通信的问题。其实你可以把它们看成是局域网中的两台机器，局域网中适用的方法这里同样适用，所以 Samba 就很适合。

首先在 Windows 中以可读写方式共享一个文件夹，假设叫做 OrangeS，然后在虚拟的 Linux 上运行下面这条命令：

```
▷ sudo mount -t smbfs -o username=user,password=blah \
//10.0.2.2/OrangeS /mnt
```

其中假设你的宿主机 IP 地址为 10.0.2.2。这样在 Linux 的 /mnt 目录下就能看到 Windows 共享文件夹下的内容了，你可以在虚拟机中随意读写，就像对待本地文件一样。

[20] 比如 readelf。

这样一来，你的编译环境就安装完成了，接下来，如同在 Linux 下一样，你还需要一个编辑器。据说始终有一部分人使用记事本（notepad）来编写代码，不管基于何种理由，希望你不要这样做，因为你可以找到许多比 notepad 更适合编写代码的编辑器，有收费的，也有免费的，它们通常都具备关键字颜色，自动缩进等方便开发者的功能，可以大大提高工作效率。

总结一下的话，搭建一个 Windows 下的开发环境，你需要做以下工作：

- 安装 Windows。
- 安装 Bochs（安装程序可到其官方网站获取）。
- 安装一个你喜欢的编辑器用来编写代码。
- 安装一个速度较快的虚拟机，如 QEMU（安装程序可到其官方网站获取[21]）。
- 在速度较快的虚拟机上安装一个 Linux。
- 在虚拟的 Linux 中安装 GCC、GNU Make、NASM、Samba——如果它们没有默认被安装上的话。
- 在虚拟的 Linux 和宿主机之间共享一个可读写的文件夹。

将来你的开发过程看起来很可能是这样的：

1. 在 Windows 中用编辑器编写代码。
2. 在虚拟 Linux 中用 Make 调用 GCC、NASM 及其他工具来生成内核并写入磁盘映像。
3. 在 Windows 中用 Bochs 来运行你的操作系统。
4. 如果有问题的话。
   (a) 用各种方法来调试，比如用 Bochs；
   (b) 返回第 1 步。

2.6 总结

好了，到这里相信读者已经知道如何搭建自己的开发环境了，说白了它跟开发一个普通的软件区别基本就在一个虚拟机上。它既是我们的“硬件”，又是我们的调试器，有了它我们安心多了。那是不是马上就可以开始我们的操作系统开发之旅了呢？很遗憾，还不能那么着急，因为你知道，操作系统是跟硬件紧密相连的，如果想实现一个运行在使用 IA32 架构的 IBM PC 上的操作系统，免不了要具备相关的知识。其中的重头戏就是 32 位 Intel CPU 的运行机制，毕竟 CPU 是一台计算机的大脑，也是整个计算机体系的核心。

[21] QEMU 的官方网站位于http://bellard.org/qemu/。

所以紧接着我们要学习的，就是要了解 IA32 保护模式。掌握了保护模式，我们才知道 Intel 的 CPU 如何运行在 32 位模式之下，从而才有可能写出一个 32 位的操作系统。

如果读者已经掌握了保护模式的内容，可以直接跳到第 4 章。

Love your neighbor, yet don't pull down your hedge.

—— *B. Franklin*

8 进程间通信

我们提到过，当一个进程需要操作系统的帮助，它可以通过系统调用让内核来替它完成一些工作。迄今为止，我们已经熟悉了系统调用的工作机制，并且已经实现了不止一个系统调用。接下来你会发现，用户进程将会有更多事情依赖于内核。比如我们想实现一个文件系统，最起码读写硬盘的工作要求助于内核。这里我们可以逐渐地增加系统调用，但也可以采用另一种方案，就是将这些工作剥离出来，交给一些系统进程来完成，让内核只负责它必须负责的工作，比如进程调度。这种将内核工作简单化的思想，便是微内核的基本思想。而所有工作通过系统调用扔给内核态的做法，被称为宏内核。

在基于宏内核的操作系统中，完成具体任务时，用户进程通过系统调用让内核来做事，直来直去，我们之前已经很熟悉了。在基于微内核的操作系统中，这个过程稍微复杂一些。在完成具体任务时，内核的角色很像是个中介。就比如我们将要实现的文件系统吧，设想用户进程 P 读取一个文件，首先通过内核告诉进程 FS，然后 FS 再通过内核告诉驱动程序（也是一个独立的进程），驱动程序读取硬盘，返回结果。这样一来，一项工作的完成变得有些曲折，需要多个进程协同工作。于是，进程间通信也就变得至关重要了。

到如今，我们的操作系统慢慢长大，接下来我们要用它来管理磁盘和磁盘上的文件并管理内存等，这些都要向应用程序提供接口，到了必须决定用微内核还是宏内核的时候了。怎么办呢？当然不能抛个硬币了事。我们不妨先找两个具体的例子来看看它们分别是怎么回事，看完了，明白了，再做决定也不迟。

8.1 微内核还是宏内核

微内核和宏内核的例子都非常好找。我们一直拿在手边的 Minix，以及每天在用的 Linux，便是两者的典型例子。Minix 是微内核的，Linux 则是宏内核的。

说起这两个例子，有一段轶事不能不提。那就是当年 Tanenbaum 和 Linus 一老一少的口舌之争。话说 Linus 写了个操作系统叫做 Linux，使用的是宏内核，他把这个消息发在了 comp.os.minix 新闻组上，这时 Tanenbaum 说话了，把 Linux 批评了一通，年轻气盛的 Linus 于是发信回击，这样一来二去，为我们留下一段微内核与宏内核的经典争论。

争论的全部内容在这里我们就不全部转述了，读者感兴趣的话可以用搜索引擎很容易地搜到[1]，我们把其中的重点说一下。在谈到微内核和宏内核时，Andy（Andrew S. Tanenbaum）是这样说的：

> 老一点的操作系统都是宏内核的，也就是说，整个操作系统是一个运行在核心态的单独的 a.out 文件，这个二进制文件包含进程管理、内存管理、文件系统以及其他。具体实例包括 UNIX、MS-DOS、VMS、MVS、OS/360、MULTICS 等。
>
> 另一种便是微内核，在这种系统中操作系统的大部分都运行在单独的进程，而且多数在内核之外。它们之间通过消息传递来通信。内核的任务是处理消息传递、中断处理、底层的进程管理，以及可能的 I/O。这种设计的实例有 RC4000、Amoeba、Chorus、Mach，以及还没有发布的 Windows/NT。
>
> 我完全可以（但不必）再讲述一段关于两者之间相对优势的很长的故事，然而在实际设计操作系统的人中间说说就够了，争论实际上已经结束。微内核已经取得了胜利。对于宏内核而言唯一的争论焦点在于效率，不过已经有足够的证据表明微内核可以像宏内核一样快（比如 Rick Rashid 已经发表了 Mach 3.0 和宏内核系统的比较报告）所以那不过是喊喊而已罢了。
>
> Minix 是微内核的，文件系统和内存管理是单独的进程，它们运行在内核之外。I/O 驱动也是单独的进程（在内核之内，但仅仅是因为 Intel CPU 的糟糕设计使得很难不这样做）。Linux 是个宏内核的系统。这相当于向七十年代倒退了一大步。就好比将一个已存在的工作得很好的 C 程序用 Basic 重写一遍。在我看来，在 1991 年写一个宏内核的系统真不是个好主意。

以上前两段基本上可以被认为是宏内核和微内核的基本概念。从概念上我们不难猜到，宏内核看上去试图包办一切，而微内核恰恰相反，它的任务只是“处理消息传递、中断处理、底层的进程管理，以及可能的 I/O”，而其他事情都交给内核之外单独的进程来完成。

在这段文字中 Andy 不但阐述了宏内核和微内核的概念，摆明了对于这个问题鲜明的观点，而且他也毫不掩饰自己对宏内核的不屑。而且这种不屑让他认为 Linux 简直是技术的倒退。在随后的文字中，对于 Linux 的可移植性 Andy 也做了不客气的批评。也难怪 Linus 对此非常恼火。从 Linus 的第一个回复开始，这场争论开始变得精彩起来。

Linus 的回复是这样开始的：

> 好吧，既然是这么一个主题，我恐怕不能不做回答了。向已经听了太多关于 Linux 的 Minix 使用者们道歉。我很乐意上钩（Andy 说了这些话，好像在引诱 Linus 开始一场争论——笔者注），该是吵一架的时候了！

啊哈，看来 Linus 真的被激怒了，我仿佛看到了他挽起袖子的样子☺。是

[1]或者在维基百科上看一下：http://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate。

啊，看到自己辛辛苦苦的劳动成果被人冠以“过时了”的形容，谁还能平心静气呢？

针对微内核和宏内核之争，他是这样回应的：

> 是的，Linux 是宏内核，我同意微内核更好些。如果不是你使用了具有争论性的主题，我可能会同意你大部分的观点。从理论上（以及美学上）讲 Linux 是输了。如果去年春天 GNU 内核已经做好，我可能不会这么麻烦地开始我的工作：问题是它没有做好而且到现在都没有。在已经实现这一方面 Linux 赢大了。
>
> » Minix 是微内核系统……Linux 则是宏内核的。
>
> 如果这是评价内核好坏的唯一标准，那么你是对的。你没有提到的是 Minix 的微内核实现得并不好，而且（内核内）多任务存在问题。如果我做一个多线程文件系统存在问题的操作系统，我可能不会这么快就声讨别人：事实上我会尽最大努力让别人忘掉我的失败。

这一段我觉得非常重要，因为看得出来，Linus 内心还是承认微内核的优势的，而且他提到了“美学”（aesthetical）这个词，因为的确，微内核的思想更加优雅，这在我们下文中的分析中也可以看到。不过尽管如此，他还是批评了 Minix 本身，认为它的微内核实现的并不令人满意。

在后来谈到可移植性的时候，Linus 的话也颇具初生牛犊不怕虎的劲头：

> 可移植性是为不能编写新程序的人设计的
>
> ——我，现在（使用傲慢的语气）

真的很精彩不是么？我甚至感觉有点像在看武侠片，一老一少，出招拆招，虽是打架，但颇有章法。不难看出，刚刚这句“我，现在（使用傲慢的语气）”甚至带有一丝挑衅意味，看这句话我甚至在想像着 Linus 敲出这行字的时候该是带着怎样傲慢的神色——不过谁在年轻的时候不是这样气盛的呢？呵呵。

看过热闹之后，让我们来实地勘查一下，两种内核看上去是什么样子的。我们就以系统调用作为突破口，看看它们的代码。

8.1.1 Linux 的系统调用

为了简单起见，我们拿 Linux 0.01 作为 Linux 方代表——最新的 Linux 内核代码量太大了，不利于短时间内弄懂。其中的系统调用不止一个，我们以`fork()`为例来分析一下。为了让读者一下子就理清这个系统调用的脉络，对于代码细节这里就不细言了，我们直接来看程序的流程图（如图8.1所示）。相关的具体代码读者可以在 Linux 0.01 的以下文件中找到：

- init/main.c
- include/unistd.h
- kernel/sched.c

- include/linux/sys.h
- kernel/system_call.s

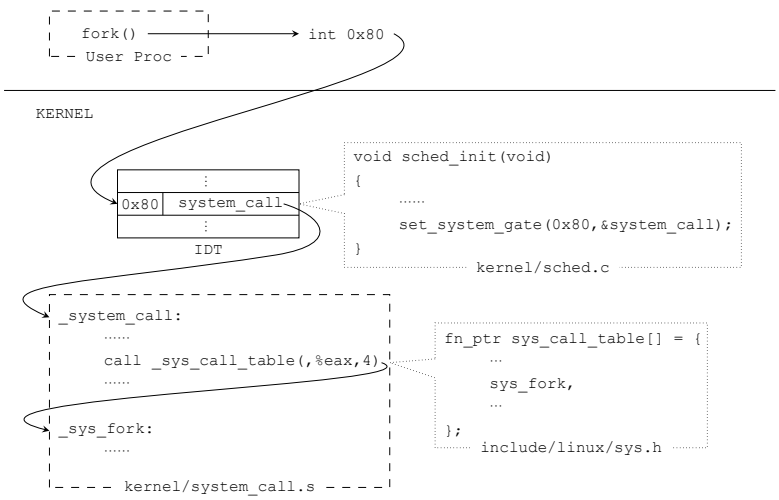

图 8.1 Linux 0.01 的 fork 系统调用

从图8.1中我们可以看出，调用`fork()`实际上是调用了中断`0x80`，通过事先初始化好的 IDT，程序转移到了`_system_call`，最终通过一个函数指针数组`sys_call_table[]`转化成了调用`sys_fork()`。这跟我们实现过的系统调用是很相似的，此处不再赘述。

8.1.2 Minix 的系统调用

Linux 的`fork`系统调用很容易理解，但 Minix 的就不这么简单了，它刚开始甚至可能让你感到迷惑。我们来打开 Minix 代码文件src/kernel/proc.c，看一下函数`sys_call()`的开头（见代码8.1）。

代码 8.1 Minix 的 sys_call()

```
121 PUBLIC int sys_call(function, src_dest, m_ptr)
122 int function;                    /* SEND, RECEIVE, or BOTH */
123 int src_dest;                    /* source to receive from or dest to send to */
124 message *m_ptr;                  /* pointer to message */
125 {
126 /* The only system calls that exist in MINIX are sending and receiving
127  * messages.  These are done by trapping to the kernel with an INT instruction.
128  * The trap is caught and sys_call() is called to send or receive a message
129  * (or both). The caller is always given by proc_ptr.
130  */
--- ...
143 }
```

开头这段注释非常重要，一个“only”道破天机（或者将你搞晕）：在 Minix 中，不再像 Linux 那样有许许多多的系统调用（`sys_call_table[]`中

列出的有几十个），而是仅有发送和接收消息的系统调用。通过sys_call的参数function的注释我们可以知道，系统调用的种类总共有三个，那就是SEND、RECEIVE以及BOTH。

可是系统调用虽少，实现的功能却不能少，那么 Minix 是怎样通过仅仅三个系统调用就实现与以 Linux 为代表的宏内核 OS 一样多的功能呢？我们仍以fork()为例，来看一下 Minix 是怎么做的。

相对于 Linux，Minix 的机制显得有点复杂，我们还是直接来看图8.2。

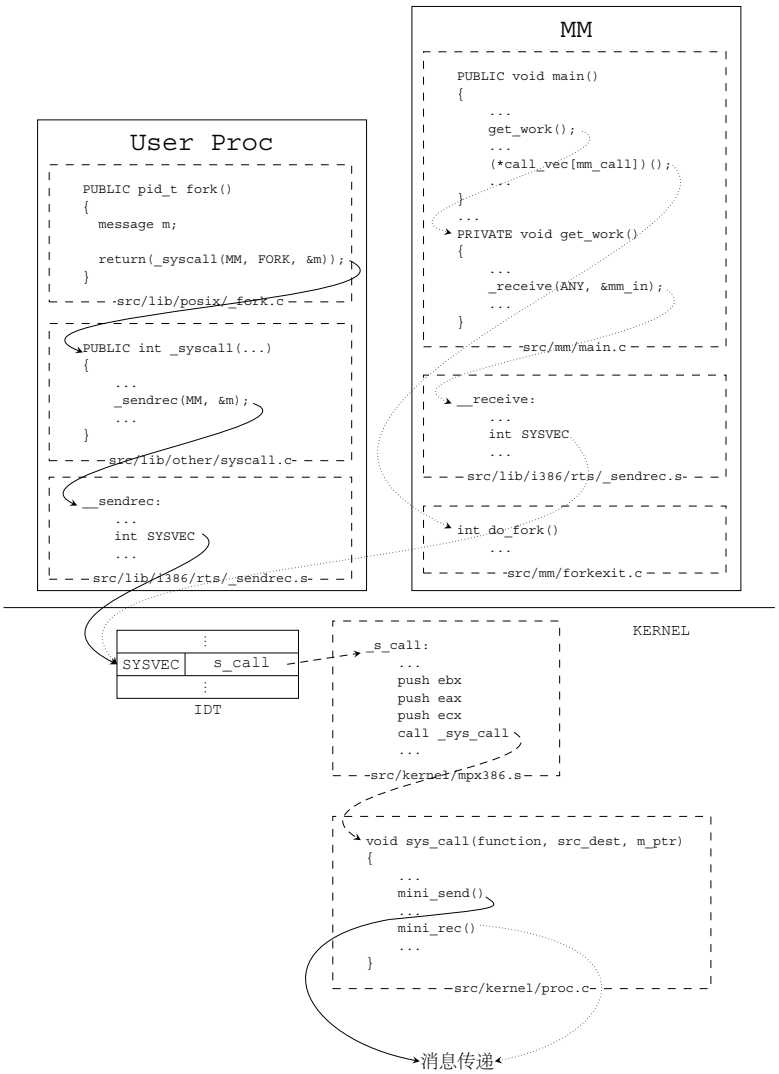

图 8.2 Minix 的 fork“系统调用”

跟图8.1所示的 Linux 不同，这里多出来一个内存管理器（MM），fork()所要做的工作是由它来负责的（如果是另外的系统调用，那么具体工作可能就不是由 MM 来负责，比如系统调用read()就是由 FS 来负责的，跟 MM 类似，

FS是单独运行的另一个进程），那么MM是如何得到用户进程的通知的呢？正是消息机制在进程之间起到了重要作用，它类似于邮政系统，在信封（或包裹单）上写明目的地，消息就送达了。

图中使用了三种箭头，实线（——→）表示消息的发送过程，点线（……→）表示消息的获取过程，虚线（- - -→）表示发送和接收消息都会经历的过程。

用户进程对`fork()`的调用将最终转化成调用内核态的函数`sys_call()`（位于src/kernel/proc.c中），消息（即图中的m）的地址这时已经作为参数被传递进来，`sys_call()`可以据此得知m的内容，并在适当的时候将内容传递给MM。MM的工作其实说起来很简单，它不断地获取并处理消息，所以它能够得到用户进程发送的m，并将其存放在`mm_in`中。当MM通过获得的`mm_in`得知了消息的内容是要进行`fork`操作，它就进一步调用其`do_fork()`完成整个过程。

消息的一送一收之间，`fork()`的大致过程我们就已经基本了解了。其实我们也完全可以猜测出其他系统调用的情况。不外乎是通过调用`_syscall()`转化成发送消息，将来会有相应的进程取出消息进行处理。

说到这里，有一个情况需要说明，就是我们拿Linux中的`fork()`和Minix中的`fork()`来比较是有些不公平的。因为你也一定已经看到，实现方式真正与Linux的`fork()`相同的是Minix的`_sendrec()`和`_receive()`，它们都是通过中断进入内核，在内核中完成任务。而Minix中的`fork()`是通过两个进程分别调用`_sendrec()`和`_receive()`这两个系统调用来实现的，从这个意义上来说，Minix中的`fork()`其实不算系统调用（这也是函数`sys_call()`的开头注释中说Minix的系统调用只有三个的原因），它只是在完成一个紧密依赖系统调用的工作罢了。不过从用户的角度，这种差别是看不到的，而且只要调用`fork()`时能实现需要的功能，这种差别就无关紧要。因此用户完全可以称`fork()`为一个系统调用。

实现方法的差别源于设计思路的不同。在Minix中，真正的系统调用只有三个，这意味着内核不必事无巨细地处理用户进程要求的所有工作，只需要做好其“邮局”的职能，将消息按照需求来回传送就够了。在Linux中内核所做的工作，在Minix中被交给专门的进程来完成。你可能一下子就明白了，原来微内核的“微”字是让内核功能最简化的意思。

8.1.3 我们的选择

到这里，微内核和宏内核各自的工作原理我想读者已经明白了。同时它们的优缺点也基本上清楚了。宏内核的优势在于其逻辑简单，直截了当，实现起来也容易，而且也因为它的直接，避免了像微内核那样在消息传递时占用资源。而微内核的优势在于，它的逻辑虽相对复杂但非常严谨，结构上显得非常优雅和精致，而且程序更容易模块化，从而更容易移植。

从编程的难易程度上来看，宏内核看上去具有一定优势，因为它很直接，不需要绕弯子，但从长期来看，当内核逐渐变大，微内核的结构会更加清晰。虽然选择微内核意味着有调试起来有些困难的消息机制摆在面前，但从设计理念上来看，微内核更加“摩登”，更“酷”。而且，从学习编程的角度看，搞一个微内核可以为将来架构其他东西作为很有益的参考。基于这些原因，我们选

择微内核。

选择了微内核，首要的任务就比较明显了，那就是实现一个进程间通信（IPC）机制。其实这也没有想像中那么难，我们下面就来看一看。

8.2 IPC

IPC 是 Inter-Process Communication 的缩写，直译为进程间通信，说白了就是进程间发消息。我们在上一节中把这种消息传递比作邮政系统，但实际上这种比喻并不全对。有的消息机制是很像收发邮件的，这种叫做异步 IPC，意思是说，发信者发完就去干别的了，收信者也一样，看看信箱里没信，也不坐在旁边傻等。而有另一种消息机制正好相反，被称为同步 IPC，它不像邮寄，倒像接力赛，发送者一直等到接收者收到消息才肯放手，接收者也一样，接不到就一直等着，不干别的。

当然你可以把同步 IPC 也比作邮寄，只不过寄信的人从把信投到信箱里的那一刻开始，就住在邮局不走了，其他什么也不干了，就等着邮局说：“哥们儿，你的信对方已经收到了，放心回家吧！”这才恋恋不舍地离开。收信的也一样，一旦决定收信，就守在自家信箱前面不走了，一直等，连觉也不睡，望穿秋水，等信拿在手里了，这才回屋，每收一次信，就得瘦个十几斤。

我们都是性情中人，我们选择傻等，或曰同步 IPC。

同步 IPC 有若干的好处，比如：

- 操作系统不需要另外维护缓冲区来存放正在传递的消息；
- 操作系统不需要保留一份消息副本；
- 操作系统不需要维护接收队列（发送队列还是需要的）；
- 发送者和接收者都可在任何时刻清晰且容易地知道消息是否送达；
- 从实现系统调用的角度来看，同步 IPC 更加合理——当使用系统调用时，我们的确需要等待内核返回结果之后再继续。

这些特性读者可能无法一下子全部明白，不要紧，我们接下来写完代码，你就全都明白了。

8.3 实现 IPC

Minix 的 IPC 机制我们已经明白了，它的核心乃在于“`int SYSVEC`”这个软中断以及与之对应的`sys_call()`这个函数。增加一个系统调用对我们来讲已是信手拈来的事，按照表7.6一步一步来就好了。我们把这个新的系统调用起名为`sendrec`。`sendrec`和`sys_sendrec`的函数体分别见代码8.2和代码8.3。

代码 8.2 sendrec（节自 chapter8/a/kernel/syscall.asm）

```
25 sendrec:
```

```
26          mov     eax, _NR_sendrec
27          mov     ebx, [esp + 4]  ; function
28          mov     ecx, [esp + 8]  ; src_dest
29          mov     edx, [esp + 12] ; p_msg
30          int     INT_VECTOR_SYS_CALL
31          ret
```

代码 8.3 sys_sendrec（chapter8/a/kernel/proc.c）

```
53  /*****************************************************************************
54   *                                sys_sendrec
55   *****************************************************************************/
56  /**
57   * <Ring 0> The core routine of system call 'sendrec()'.
58   *
59   * @param function SEND or RECEIVE
60   * @param src_dest To/From whom the message is transferred.
61   * @param m        Ptr to the MESSAGE body.
62   * @param p        The caller proc.
63   *
64   * @return Zero if success.
65   *****************************************************************************/
66  PUBLIC int sys_sendrec(int function, int src_dest, MESSAGE* m, struct proc* p)
67  {
68          assert(k_reenter == 0); /* make sure we are not in ring0 */
69          assert((src_dest >= 0 && src_dest < NR_TASKS + NR_PROCS) ||
70                 src_dest == ANY ||
71                 src_dest == INTERRUPT);
72
73          int ret = 0;
74          int caller = proc2pid(p);
75          MESSAGE* mla = (MESSAGE*)va2la(caller, m);
76          mla->source = caller;
77
78          assert(mla->source != src_dest);
79
80          /**
81           * Actually we have the third message type: BOTH. However, it is not
82           * allowed to be passed to the kernel directly. Kernel doesn't know
83           * it at all. It is transformed into a SEND followed by a RECEIVE
84           * by 'send_recv()'.
85           */
86          if (function == SEND) {
87                  ret = msg_send(p, src_dest, m);
88                  if (ret != 0)
89                          return ret;
90          }
91          else if (function == RECEIVE) {
92                  ret = msg_receive(p, src_dest, m);
93                  if (ret != 0)
94                          return ret;
95          }
96          else {
97                  panic("{sys_sendrec}␣invalid␣function:␣"
98                        "%d␣(SEND:%d,␣RECEIVE:%d).", function, SEND, RECEIVE);
99          }
100
101         return 0;
102 }
```

表7.6的最后一步中提到，如果参数个数与以前的系统调用比有所增加，则需要修改kernel.asm中的`sys_call`。额外要注意，我们新加的参数是通过`edx`这个参数传递的，而`save`这个函数中也用到了寄存器`dx`，所以我们同时需要修改`save`（代码8.4）。

代码 8.4 修改 save 和 sys_call（节自 chapter8/a/kernel/kernel.asm）

```
311  ; ====================================================================================
312  ;                                    save
313  ; ====================================================================================
314  save:
315          pushad          ; `.
316          push    ds      ;  |
317          push    es      ;  | 保存原寄存器值
318          push    fs      ;  |
319          push    gs      ; /
320
321          ;; 注意，从这里开始，一直到 `mov esp, StackTop'，中间坚决不能用 push/pop 指令，
322          ;; 因为当前 esp 指向 proc_table 里的某个位置，push 会破坏掉进程表，导致灾难性后果！
323
⇒ 324          mov     esi, edx        ; 保存 edx，因为 edx 里保存了系统调用的参数
325                                      ;（没用栈，而是用了另一个寄存器 esi）
326          mov     dx, ss
327          mov     ds, dx
328          mov     es, dx
329          mov     fs, dx
330
⇒ 331          mov     edx, esi        ; 恢复 edx
332
333          mov     esi, esp                    ;esi = 进程表起始地址
334
335          inc     dword [k_reenter]           ;k_reenter++;
336          cmp     dword [k_reenter], 0        ;if(k_reenter ==0)
337          jne     .1                          ;{
338          mov     esp, StackTop               ;  mov esp, StackTop <--切换到内核栈
339          push    restart                     ;  push restart
340          jmp     [esi + RETADR - P_STACKBASE];  return;
341  .1:                                         ;} else { 已经在内核栈，不需要再切换
342          push    restart_reenter             ;  push restart_reenter
343          jmp     [esi + RETADR - P_STACKBASE];  return;
344                                              ;}
345
346
347  ; ====================================================================================
348  ;                                  sys_call
349  ; ====================================================================================
350  sys_call:
351          call    save
352
353          sti
354          push    esi
355
356          push    dword [p_proc_ready]
⇒ 357          push    edx
358          push    ecx
359          push    ebx
360          call    [sys_call_table + eax * 4]
⇒ 361          add     esp, 4 * 4
362
363          pop     esi
364          mov     [esi + EAXREG - P_STACKBASE], eax
365          cli
366
367          ret
```

sys_sendrec()这个函数被设计得相当简单，它可以描述为：把SEND消息交给msg_send()处理，把RECEIVE消息交给msg_receive()处理。

msg_send()和msg_receive()这两个函数我们过一会儿细细分解，先来看看之前没出现过的assert()和panic()。这两个函数虽然起的是辅助作用，但绝对不是可有可无，因为在我们接下来要处理的消息收发中，有一些编

程细节还真容易让人迷糊，这时候assert()就大显神威了，它会在错误被放大之前通知你。panic()的作用也类似，用于通知你发生了严重的错误。

8.3.1 assert()和panic()

先来看assert()。你或许早就开始使用这个函数，但之前你使用的都是现成的assert，只要包含一个头文件，就可以方便地使用。如今什么都得自力更生了，不过不用怕，写一个assert函数并非难事，见代码8.5。

代码 8.5 assert（chapter8/a/include/const.h）

```
#define ASSERT
#ifdef ASSERT
void assertion_failure(char *exp, char *file, char *base_file, int line);
#define assert(exp)  if (exp) ; \
        else assertion_failure(#exp, __FILE__, __BASE_FILE__, __LINE__)
#else
#define assert(exp)
#endif
```

注意其中的__FILE__、__BASE_FILE__和__LINE__这三个宏，它们的意义如下[2]：

- **__FILE__**将被展开成当前输入的文件。在这里，它告诉我们哪个文件中产生了异常。
- **__BASE_FILE__**可被认为是传递给编译器的那个文件名。比如你在m.c中包含了n.h，而n.h中的某一个assert函数失败了，则__FILE__为n.h，__BASE_FILE__为m.c。
- **__LINE__**将被展开成当前的行号。

明白了这几个宏的意义，剩下的assertion_failure()这个函数就显得容易了，它的作用就是将错误发生的位置打印出来（代码8.6）。

代码 8.6 assertion_failure（chapter8/a/lib/misc.c）

```
PUBLIC void assertion_failure(char *exp, char *file, char *base_file, int line)
{
        printl("%c  assert(%s) failed: file: %s, base_file: %s, ln%d",
               MAG_CH_ASSERT,
               exp, file, base_file, line);

        /**
         * If assertion fails in a TASK, the system will halt before
         * printl() returns. If it happens in a USER PROC, printl() will
         * return like a common routine and arrive here.
         * @see sys_printx()
         *
         * We use a forever loop to prevent the proc from going on:
         */
        spin("assertion_failure()");

        /* should never arrive here */
        __asm__ __volatile__("ud2");
}
```

[2]更详细的解释可参考 GCC 官方文档之 The C Preprocessor。

注意这里使用了一点点小伎俩，那就是使用了一个改进后的打印函数，叫做printl()，它其实就是一个定义成printf的宏，不过这里的printf跟上一章中的稍有不同，它将调用一个叫做printx的系统调用，并最终调用函数sys_printx()，它位于tty.c中（代码8.7）。

代码 8.7 sys_printx（chapter8/a/kernel/tty.c）

```
PUBLIC int sys_printx(int _unused1, int _unused2, char* s, struct proc* p_proc)
{
        const char * p;
        char ch;

        char reenter_err[] = "? k_reenter is incorrect for unknown reason";
        reenter_err[0] = MAG_CH_PANIC;

        /**
         * @note Code in both Ring 0 and Ring 1~3 may invoke printx().
         * If this happens in Ring 0, no linear-physical address mapping
         * is needed.
         *
         * @attention The value of 'k_reenter' is tricky here. When
         *   -# printx() is called in Ring 0
         *      - k_reenter > 0. When code in Ring 0 calls printx(),
         *        an 'interrupt re-enter' will occur (printx() generates
         *        a software interrupt). Thus 'k_reenter' will be increased
         *        by 'kernel.asm::save' and be greater than 0.
         *   -# printx() is called in Ring 1~3
         *      - k_reenter == 0.
         */
        if (k_reenter == 0)  /* printx() called in Ring<1~3> */
                p = va2la(proc2pid(p_proc), s);
        else if (k_reenter > 0) /* printx() called in Ring<0> */
                p = s;
        else    /* this should NOT happen */
                p = reenter_err;

        /**
         * @note if assertion fails in any TASK, the system will be halted;
         * if it fails in a USER PROC, it'll return like any normal syscall
         * does.
         */
        if ((*p == MAG_CH_PANIC) ||
            (*p == MAG_CH_ASSERT && p_proc_ready < &proc_table[NR_TASKS])) {
                disable_int();
                char * v = (char*)V_MEM_BASE;
                const char * q = p + 1; /* +1: skip the magic char */

                while (v < (char*)(V_MEM_BASE + V_MEM_SIZE)) {
                        *v++ = *q++;
                        *v++ = RED_CHAR;
                        if (!*q) {
                                while (((int)v - V_MEM_BASE) % (SCR_WIDTH * 16)) {
                                        /* *v++ = ' '; */
                                        v++;
                                        *v++ = GRAY_CHAR;
                                }
                                q = p + 1;
                        }
                }

                __asm__ __volatile__("hlt");
        }

        while ((ch = *p++) != 0) {
                if (ch == MAG_CH_PANIC || ch == MAG_CH_ASSERT)
                        continue; /* skip the magic char */
```

```
240
241                 out_char(tty_table[p_proc->nr_tty].p_console, ch);
242         }
243
244         return 0;
245 }
```

容易看到，sys_printx()将首先判断首字符是否为预先设定的“Magic Char”，如果是的话，则做响应的特殊处理。我们的assertion_failure()就使用了MAG_CH_ASSERT作为“Magic Char”。当sys_printx()发现传入字符串的第一个字符是MAG_CH_ASSERT时，会同时判断调用系统调用的进程是系统进程（TASK）还是用户进程（USER PROC），如果是系统进程，则停止整个系统的运转，并将要打印的字符串打印在显存的各处；如果是用户进程，则打印之后像一个普通的printx调用一样返回，届时该用户进程会因为assertion_failure()中对函数spin()的调用而进入死循环。换言之，系统进程的assert失败会导致系统停转，用户进程的失败仅仅使自己停转。

到这里读者应该很清楚assert()函数的实现方法了，我们不妨来试验一下，在系统进程 TTY 中添加一句“assert(0);”，运行，读者将看到如图8.3所示的画面。再在用户进程 TestC 中添加一句“assert(0);”，将看到如图8.4所示的画面。

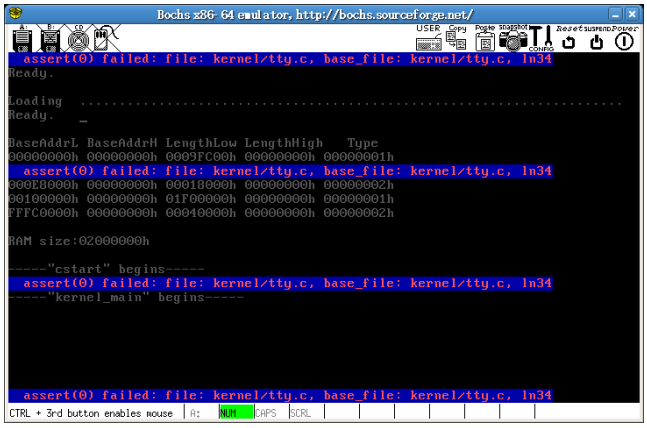

图 8.3 系统任务中 assert 失败

panic()跟assert()类似，也用到了sys_printx()和“Magic Char”，不过它要更简单一些，见代码8.8。

代码 8.8 panic（chapter8/a/kernel/main.c）

```
159 PUBLIC void panic(const char *fmt, ...)
160 {
161         int i;
162         char buf[256];
163
164         /* 4 is the size of fmt in the stack */
165         va_list arg = (va_list)((char*)&fmt + 4);
166
167         i = vsprintf(buf, fmt, arg);
168
```

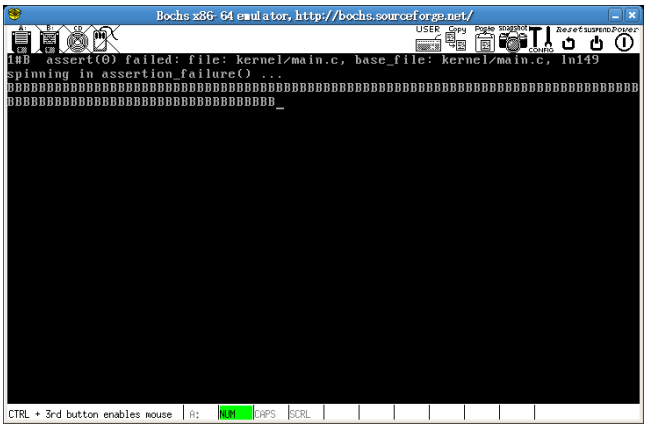

图 8.4 用户进程中 assert 失败

```
169         printl("%c␣!!panic!!␣%s", MAG_CH_PANIC, buf);
170
171         /* should never arrive here */
172         __asm__ __volatile__("ud2");
173 }
```

由于panic只会用在系统任务所处的 Ring1 或 Ring0，所以sys_printx()遇到MAG_CH_PANIC就直接叫停整个系统，因为我们使用panic的时候，必是发生了严重错误的时候。

我们同样可以在 TTY 中试验一下panic的效果，比如添加这么一行：

```
panic("in␣TTY");
```

运行，会看到如图8.5所示的效果。

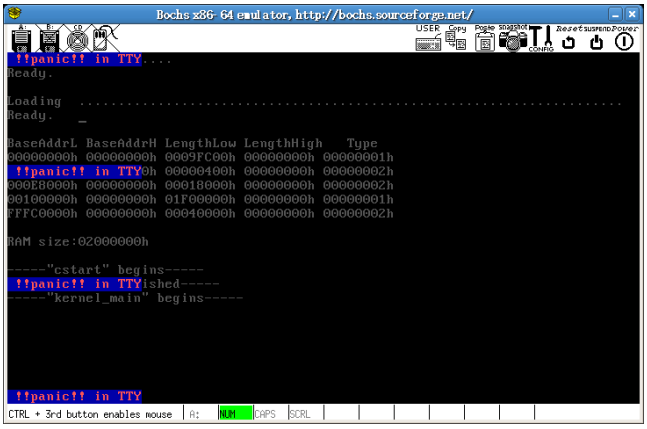

图 8.5 panic

在我们接下来的代码中，很多地方用到了assert()和panic()，其实有些地方完全可以不用这两个函数，而是以返回值的形式向上层函数传递的，但使用assert()和panic()可以减少代码量，并在第一时间通知我们哪里出了问题，作为一个试验性的操作系统，笔者认为这样做比使用某种方法来“消除”错误还要好。

8.3.2 msg_send()和msg_receive()

话题岔开这么久，让我们回到代码8.3，既然关键的函数是msg_send()和msg_receive()，那我们就来看一下，见代码8.9，它们是 IPC 的核心代码。

代码 8.9 msg_send 和 msg_receive（chapter8/a/kernel/proc.c）

```
145 /*****************************************************************************
146  *                                ldt_seg_linear
147  *****************************************************************************/
148 /**
149  * <Ring 0~1> Calculate the linear address of a certain segment of a given
150  * proc.
151  *
152  * @param p   Whose (the proc ptr).
153  * @param idx Which (one proc has more than one segments).
154  *
155  * @return  The required linear address.
156  *****************************************************************************/
157 PUBLIC int ldt_seg_linear(struct proc* p, int idx)
158 {
159         struct descriptor * d = &p->ldts[idx];
160 
161         return d->base_high << 24 | d->base_mid << 16 | d->base_low;
162 }
163 
164 /*****************************************************************************
165  *                                va2la
166  *****************************************************************************/
167 /**
168  * <Ring 0~1> Virtual addr --> Linear addr.
169  *
170  * @param pid  PID of the proc whose address is to be calculated.
171  * @param va   Virtual address.
172  *
173  * @return The linear address for the given virtual address.
174  *****************************************************************************/
175 PUBLIC void* va2la(int pid, void* va)
176 {
177         struct proc* p = &proc_table[pid];
178 
179         u32 seg_base = ldt_seg_linear(p, INDEX_LDT_RW);
180         u32 la = seg_base + (u32)va;
181 
182         if (pid < NR_TASKS + NR_PROCS) {
183                 assert(la == (u32)va);
184         }
185 
186         return (void*)la;
187 }
188 
189 /*****************************************************************************
190  *                                reset_msg
191  *****************************************************************************/
192 /**
```

```
193  * <Ring 0~3> Clear up a MESSAGE by setting each byte to 0.
194  *
195  * @param p  The message to be cleared.
196  *****************************************************************************/
197 PUBLIC void reset_msg(MESSAGE* p)
198 {
199         memset(p, 0, sizeof(MESSAGE));
200 }
201
202 /*****************************************************************************
203  *                                block
204  *****************************************************************************/
205 /**
206  * <Ring 0> This routine is called after 'p_flags' has been set (!= 0), it
207  * calls 'schedule()' to choose another proc as the 'proc_ready'.
208  *
209  * @attention This routine does not change 'p_flags'. Make sure the 'p_flags'
210  * of the proc to be blocked has been set properly.
211  *
212  * @param p The proc to be blocked.
213  *****************************************************************************/
214 PRIVATE void block(struct proc* p)
215 {
216         assert(p->p_flags);
217         schedule();
218 }
219
220 /*****************************************************************************
221  *                                unblock
222  *****************************************************************************/
223 /**
224  * <Ring 0> This is a dummy routine. It does nothing actually. When it is
225  * called, the 'p_flags' should have been cleared (== 0).
226  *
227  * @param p The unblocked proc.
228  *****************************************************************************/
229 PRIVATE void unblock(struct proc* p)
230 {
231         assert(p->p_flags == 0);
232 }
233
234 /*****************************************************************************
235  *                                deadlock
236  *****************************************************************************/
237 /**
238  * <Ring 0> Check whether it is safe to send a message from src to dest.
239  * The routine will detect if the messaging graph contains a cycle. For
240  * instance, if we have procs trying to send messages like this:
241  * A -> B -> C -> A, then a deadlock occurs, because all of them will
242  * wait forever. If no cycles detected, it is considered as safe.
243  *
244  * @param src   Who wants to send message.
245  * @param dest  To whom the message is sent.
246  *
247  * @return Zero if success.
248  *****************************************************************************/
249 PRIVATE int deadlock(int src, int dest)
250 {
251         struct proc* p = proc_table + dest;
252         while (1) {
253                 if (p->p_flags & SENDING) {
254                         if (p->p_sendto == src) {
255                                 /* print the chain */
256                                 p = proc_table + dest;
257                                 printl("=_=%s", p->name);
258                                 do {
259                                         assert(p->p_msg);
```

```
260                                     p = proc_table + p->p_sendto;
261                                     printl("->%s", p->name);
262                             } while (p != proc_table + src);
263                             printl("=_=");
264
265                             return 1;
266                     }
267                     p = proc_table + p->p_sendto;
268             }
269             else {
270                     break;
271             }
272     }
273     return 0;
274 }
275
276 /*****************************************************************************
277  *                                msg_send
278  *****************************************************************************/
279 /**
280  * <Ring 0> Send a message to the dest proc. If dest is blocked waiting for
281  * the message, copy the message to it and unblock dest. Otherwise the caller
282  * will be blocked and appended to the dest's sending queue.
283  *
284  * @param current  The caller, the sender.
285  * @param dest     To whom the message is sent.
286  * @param m        The message.
287  *
288  * @return Zero if success.
289  *****************************************************************************/
290 PRIVATE int msg_send(struct proc* current, int dest, MESSAGE* m)
291 {
292     struct proc* sender = current;
293     struct proc* p_dest = proc_table + dest; /* proc dest */
294
295     assert(proc2pid(sender) != dest);
296
297     /* check for deadlock here */
298     if (deadlock(proc2pid(sender), dest)) {
299         panic(">>DEADLOCK<<␣%s->%s", sender->name, p_dest->name);
300     }
301
302     if ((p_dest->p_flags & RECEIVING) && /* dest is waiting for the msg */
303         (p_dest->p_recvfrom == proc2pid(sender) ||
304          p_dest->p_recvfrom == ANY)) {
305             assert(p_dest->p_msg);
306             assert(m);
307
308             phys_copy(va2la(dest, p_dest->p_msg),
309                       va2la(proc2pid(sender), m),
310                       sizeof(MESSAGE));
311             p_dest->p_msg = 0;
312             p_dest->p_flags &= ~RECEIVING; /* dest has received the msg */
313             p_dest->p_recvfrom = NO_TASK;
314             unblock(p_dest);
315
316             assert(p_dest->p_flags == 0);
317             assert(p_dest->p_msg == 0);
318             assert(p_dest->p_recvfrom == NO_TASK);
319             assert(p_dest->p_sendto == NO_TASK);
320             assert(sender->p_flags == 0);
321             assert(sender->p_msg == 0);
322             assert(sender->p_recvfrom == NO_TASK);
323             assert(sender->p_sendto == NO_TASK);
324     }
325     else { /* dest is not waiting for the msg */
326             sender->p_flags |= SENDING;
```

```
327                 assert(sender->p_flags == SENDING);
328                 sender->p_sendto = dest;
329                 sender->p_msg = m;
330 
331                 /* append to the sending queue */
332                 struct proc * p;
333                 if (p_dest->q_sending) {
334                         p = p_dest->q_sending;
335                         while (p->next_sending)
336                                 p = p->next_sending;
337                         p->next_sending = sender;
338                 }
339                 else {
340                         p_dest->q_sending = sender;
341                 }
342                 sender->next_sending = 0;
343 
344                 block(sender);
345 
346                 assert(sender->p_flags == SENDING);
347                 assert(sender->p_msg != 0);
348                 assert(sender->p_recvfrom == NO_TASK);
349                 assert(sender->p_sendto == dest);
350         }
351 
352         return 0;
353 }
354 
355 
356 /*****************************************************************************
357  *                                msg_receive
358  *****************************************************************************/
359 /**
360  * <Ring 0> Try to get a message from the src proc. If src is blocked sending
361  * the message, copy the message from it and unblock src. Otherwise the caller
362  * will be blocked.
363  *
364  * @param current The caller, the proc who wanna receive.
365  * @param src     From whom the message will be received.
366  * @param m       The message ptr to accept the message.
367  *
368  * @return  Zero if success.
369  *****************************************************************************/
370 PRIVATE int msg_receive(struct proc* current, int src, MESSAGE* m)
371 {
372         struct proc* p_who_wanna_recv = current; /**
373                                                   * This name is a little bit
374                                                   * wierd, but it makes me
375                                                   * think clearly, so I keep
376                                                   * it.
377                                                   */
378         struct proc* p_from = 0; /* from which the message will be fetched */
379         struct proc* prev = 0;
380         int copyok = 0;
381 
382         assert(proc2pid(p_who_wanna_recv) != src);
383 
384         if ((p_who_wanna_recv->has_int_msg) &&
385             ((src == ANY) || (src == INTERRUPT))) {
386                 /* There is an interrupt needs p_who_wanna_recv's handling and
387                  * p_who_wanna_recv is ready to handle it.
388                  */
389 
390                 MESSAGE msg;
391                 reset_msg(&msg);
392                 msg.source = INTERRUPT;
393                 msg.type = HARD_INT;
```

```
394                 assert(m);
395                 phys_copy(va2la(proc2pid(p_who_wanna_recv), m), &msg,
396                           sizeof(MESSAGE));
397
398                 p_who_wanna_recv->has_int_msg = 0;
399
400                 assert(p_who_wanna_recv->p_flags == 0);
401                 assert(p_who_wanna_recv->p_msg == 0);
402                 assert(p_who_wanna_recv->p_sendto == NO_TASK);
403                 assert(p_who_wanna_recv->has_int_msg == 0);
404
405                 return 0;
406         }
407
408
409         /* Arrives here if no interrupt for p_who_wanna_recv. */
410         if (src == ANY) {
411                 /* p_who_wanna_recv is ready to receive messages from
412                  * ANY proc, we'll check the sending queue and pick the
413                  * first proc in it.
414                  */
415                 if (p_who_wanna_recv->q_sending) {
416                         p_from = p_who_wanna_recv->q_sending;
417                         copyok = 1;
418
419                         assert(p_who_wanna_recv->p_flags == 0);
420                         assert(p_who_wanna_recv->p_msg == 0);
421                         assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
422                         assert(p_who_wanna_recv->p_sendto == NO_TASK);
423                         assert(p_who_wanna_recv->q_sending != 0);
424                         assert(p_from->p_flags == SENDING);
425                         assert(p_from->p_msg != 0);
426                         assert(p_from->p_recvfrom == NO_TASK);
427                         assert(p_from->p_sendto == proc2pid(p_who_wanna_recv));
428                 }
429         }
430         else {
431                 /* p_who_wanna_recv wants to receive a message from
432                  * a certain proc: src.
433                  */
434                 p_from = &proc_table[src];
435
436                 if ((p_from->p_flags & SENDING) &&
437                     (p_from->p_sendto == proc2pid(p_who_wanna_recv))) {
438                         /* Perfect, src is sending a message to
439                          * p_who_wanna_recv.
440                          */
441                         copyok = 1;
442
443                         struct proc* p = p_who_wanna_recv->q_sending;
444                         assert(p); /* p_from must have been appended to the
445                                     * queue, so the queue must not be NULL
446                                     */
447                         while (p) {
448                                 assert(p_from->p_flags & SENDING);
449                                 if (proc2pid(p) == src) { /* if p is the one */
450                                         p_from = p;
451                                         break;
452                                 }
453                                 prev = p;
454                                 p = p->next_sending;
455                         }
456
457                         assert(p_who_wanna_recv->p_flags == 0);
458                         assert(p_who_wanna_recv->p_msg == 0);
459                         assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
460                         assert(p_who_wanna_recv->p_sendto == NO_TASK);
```

```
461                         assert(p_who_wanna_recv->q_sending != 0);
462                         assert(p_from->p_flags == SENDING);
463                         assert(p_from->p_msg != 0);
464                         assert(p_from->p_recvfrom == NO_TASK);
465                         assert(p_from->p_sendto == proc2pid(p_who_wanna_recv));
466                 }
467         }
468
469         if (copyok) {
470                 /* It's determined from which proc the message will
471                  * be copied. Note that this proc must have been
472                  * waiting for this moment in the queue, so we should
473                  * remove it from the queue.
474                  */
475                 if (p_from == p_who_wanna_recv->q_sending) { /* the 1st one */
476                         assert(prev == 0);
477                         p_who_wanna_recv->q_sending = p_from->next_sending;
478                         p_from->next_sending = 0;
479                 }
480                 else {
481                         assert(prev);
482                         prev->next_sending = p_from->next_sending;
483                         p_from->next_sending = 0;
484                 }
485
486                 assert(m);
487                 assert(p_from->p_msg);
488                 /* copy the message */
489                 phys_copy(va2la(proc2pid(p_who_wanna_recv), m),
490                           va2la(proc2pid(p_from), p_from->p_msg),
491                           sizeof(MESSAGE));
492
493                 p_from->p_msg = 0;
494                 p_from->p_sendto = NO_TASK;
495                 p_from->p_flags &= ~SENDING;
496                 unblock(p_from);
497         }
498         else {  /* nobody's sending any msg */
499                 /* Set p_flags so that p_who_wanna_recv will not
500                  * be scheduled until it is unblocked.
501                  */
502                 p_who_wanna_recv->p_flags |= RECEIVING;
503
504                 p_who_wanna_recv->p_msg = m;
505
506                 if (src == ANY)
507                         p_who_wanna_recv->p_recvfrom = ANY;
508                 else
509                         p_who_wanna_recv->p_recvfrom = proc2pid(p_from);
510
511                 block(p_who_wanna_recv);
512
513                 assert(p_who_wanna_recv->p_flags == RECEIVING);
514                 assert(p_who_wanna_recv->p_msg != 0);
515                 assert(p_who_wanna_recv->p_recvfrom != NO_TASK);
516                 assert(p_who_wanna_recv->p_sendto == NO_TASK);
517                 assert(p_who_wanna_recv->has_int_msg == 0);
518         }
519
520         return 0;
521 }
```

围绕msg_send()和msg_receive()，代码中还列出了其他几个必要的函数，它们是：

- **ldt_seg_linear()** 每个进程都有自己的 LDT，位于进程表的中间，这

个函数就是根据 LDT 中描述符的索引来求得描述符所指向的段的基地址。

- **va2la()** 用来由虚拟地址求线性地址，它用到了ldt_seg_linear()。
- **reset_msg()** 用于把一个消息的每个字节清零。
- **block()** 阻塞一个进程。
- **unblock()** 解除一个进程的阻塞。
- **deadlock()** 简单地判断是否发生死锁。方法是判断消息的发送是否构成一个环，如果构成环则意味着发生死锁，比如 A 试图发消息给 B，同时 B 试图给 C，C 试图给 A 发消息，那么死锁就发生了，因为 A、B 和 C 三个进程都将无限等待下去（如图8.6所示）。

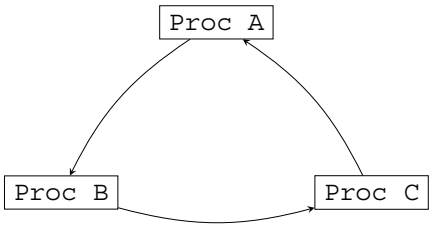

图 8.6 发消息时可能发生的死锁

在block()、unblock()和deadlock()中，都出现了struct proc这个结构体的一个新成员：p_flag。其实增加的新成员还有几个，见代码8.10。

代码 8.10 进程表的新成员（chapter8/a/include/proc.h）

```
31  struct proc {
32          struct stackframe regs;     /* process registers saved in stack frame */
33
34          u16 ldt_sel;                /* gdt selector giving ldt base and limit */
35          struct descriptor ldts[LDT_SIZE]; /* local descs for code and data */
36
37          int ticks;                  /* remained ticks */
38          int priority;
39
40          u32 pid;                    /* process id passed in from MM */
41          char name[16];              /* name of the process */
42
⇒ 43        int  p_flags;               /**
44                                       * process flags.
45                                       * A proc is runnable iff p_flags==0
46                                       */
47
⇒ 48        MESSAGE * p_msg;
⇒ 49        int p_recvfrom;
⇒ 50        int p_sendto;
51
⇒ 52        int has_int_msg;            /**
53                                       * nonzero if an INTERRUPT occurred when
54                                       * the task is not ready to deal with it.
55                                       */
56
```

```
⇒ 57        struct proc * q_sending;    /**
  58                                     * queue of procs sending messages to
  59                                     * this proc
  60                                     */
⇒ 61        struct proc * next_sending;/**
  62                                     * next proc in the sending
  63                                     * queue (q_sending)
  64                                     */
  65
  66        int nr_tty;
  67    };
```

所有增加的这些成员都是跟消息机制有关的。

- **p_flags** 用于标明进程的状态。目前它的取值可以有三种：
  - **0** 进程正在运行或准备运行。
  - **SENDING** 进程处于发送消息的状态。由于消息还未送达，进程被阻塞。
  - **RECEIVING** 进程处于接收消息的状态。由于消息还未收到，进程被阻塞。
- **p_msg** 指向消息体的指针。
- **p_recvfrom** 假设进程 P 想要接收消息，但目前没有进程发消息给它，本成员记录 P 想要从谁那里接收消息。
- **p_sendto** 假设进程 P 想要发送消息，但目前没有进程接收它，本成员记录 P 想要发送消息给谁。
- **has_int_msg** 如果有一个中断需要某进程来处理，或者换句话说，某进程正在等待一个中断发生——比如硬盘驱动可能会等待硬盘中断的发生，系统在得知中断发生后会将此位置为 1。
- **q_sending** 如果有若干进程——比如 A、B 和 C——都向同一个进程 P 发送消息，而 P 此时并未准备接收消息，那么 A、B 和 C 将会排成一个队列。进程 P 的 q_sending 指向第一个试图发送消息的进程。
- **next_sending** 试图发送消息的 A、B 和 C（依时间顺序）三进程排成的队列的实现方式是这样的：目的进程 P 的进程表的`q_sending`指向 A，进程 A 的进程表的`next_sending`指向 B，进程 B 的进程表的`next_sending`指向 C，进程 C 的进程表的`next_sending`指向空。

假设有进程 A 想要向 B 发送消息 M，那么过程将会是这样的：

1. A 首先准备好 M。
2. A 通过系统调用`sendrec`，最终调用`msg_send`。
3. 简单判断是否发生死锁。
4. 判断目标进程 B 是否正在等待来自 A 的消息：

- 如果是：消息被复制给 B，B 被解除阻塞，继续运行；
- 如果否：A 被阻塞，并被加入到 B 的发送队列中。

假设有进程 B 想要接收消息（来自特定进程、中断或者任意进程），那么过程将会是这样的：

1. B 准备一个空的消息结构体 M，用于接收消息。
2. B 通过系统调用`sendrec`，最终调用`msg_receive`。
3. 判断 B 是否有个来自硬件的消息（通过`has_int_msg`），如果是，并且 B 准备接收来自中断的消息或准备接收任意消息，则马上准备一个消息给 B，并返回。
4. 如果 B 想接收来自任意进程的消息，则从自己的发送队列中选取第一个（如果队列非空的话），将其消息复制给 M。
5. 如果 B 是想接收来自特定进程 A 的消息，则先判断 A 是否正在等待向 B 发送消息，若是的话，将其消息复制给 M。
6. 如果此时没有任何进程发消息给 B，B 会被阻塞。

值得说明的是，不管是接收方还是发送方，都各自维护一个消息结构体，只不过发送方的结构体是携带了消息内容的而接收方是空的。由于我们使用同步 IPC，一方的需求——发送或接收——只有被满足之后才会继续运行，所以操作系统不需要维护任何的消息缓冲，实现起来也就相对简单。

8.3.3 增加消息机制之后的进程调度

在上一节中我们提到，如今的每个进程增加了两种可能的状态：`SENDING`和`RECEIVING`。相应的，我们需要在进程调度的时候区别对待了。凡是处于`SENDING`或`RECEIVING`状态的进程，我们就不再让它们获得 CPU 了，也就是说，将它们“阻塞”了。这也解释了为什么`block()`和`unblock()`两个函数本质上没做任何工作——一个进程是否阻塞，已经由进程表中的`p_flags`项决定，我们不需要额外做什么工作。不过我们还是应该保留这两个函数，一方面将来可能要扩展它们，另一方面它们也有助于理清编程的思路。

代码8.11就是修改后的调度函数。

代码 8.11 增加消息机制之后的进程调度（chapter8/a/kernel/proc.c）

```
31  PUBLIC void schedule()
32  {
33          struct proc*    p;
34          int             greatest_ticks = 0;
35
36          while (!greatest_ticks) {
37                  for (p = &FIRST_PROC; p <= &LAST_PROC; p++) {
⇒ 38                          if (p->p_flags == 0) {
39                                  if (p->ticks > greatest_ticks) {
40                                          greatest_ticks = p->ticks;
41                                          p_proc_ready = p;
42                                  }
```

```
43                      }
44              }
45
46              if (!greatest_ticks)
47                      for (p = &FIRST_PROC; p <= &LAST_PROC; p++)
⇒ 48                            if (p->p_flags == 0)
49                                    p->ticks = p->priority;
50      }
51  }
```

可以看到，当且仅当p_flags为零时，一个进程才可能获得运行的机会。

8.4 使用 IPC 来替换系统调用get_ticks

到这里我们的消息机制已经可以用了，如果读者亲自实践的话，别忘了一些细枝末节的东西，比如在初始化进程时给新增加的进程表成员赋值，再比如增加一些必要的函数声明以及修改Makefile等零碎工作。

为验证消息机制是否工作正常，我们还是从最简单的工作着手，删掉原先的系统调用get_ticks，用收发消息的方法重新实现之。

不过且慢，既然是收发消息，必然是有两方参与。想想便知，我们需要一个系统进程来接收用户进程的消息，并且返回ticks值。我们就来建立一个新的系统进程，就叫它“SYSTASK”。

添加一个任务的工作还是按照第6.4.6节中所述步骤进行。它的主循环如代码8.12所示。

代码 8.12 系统进程（chapter8/a/kernel/systask.c）

```
22  /*****************************************************************************
23   *                                task_sys
24   *****************************************************************************/
25  /**
26   * <Ring 1> The main loop of TASK SYS.
27   *
28   *****************************************************************************/
29  PUBLIC void task_sys()
30  {
31          MESSAGE msg;
32          while (1) {
33                  send_recv(RECEIVE, ANY, &msg);
34                  int src = msg.source;
35
36                  switch (msg.type) {
37                  case GET_TICKS:
38                          msg.RETVAL = ticks;
39                          send_recv(SEND, src, &msg);
40                          break;
41                  default:
42                          panic("unknown␣msg␣type");
43                          break;
44                  }
45          }
46  }
```

代码很简单，不过要留心一下其中用到的函数send_recv()，它其实就是把sendrec这个系统调用给封装了一下，见代码8.13。

代码 8.13 send_recv（chapter8/a/kernel/proc.c）

```
/*****************************************************************************
 *                                send_recv
 *****************************************************************************/
/**
 * <Ring 1~3> IPC syscall.
 *
 * It is an encapsulation of 'sendrec',
 * invoking 'sendrec' directly should be avoided
 *
 * @param function  SEND, RECEIVE or BOTH
 * @param src_dest  The caller's proc_nr
 * @param msg       Pointer to the MESSAGE struct
 *
 * @return always 0.
 *****************************************************************************/
PUBLIC int send_recv(int function, int src_dest, MESSAGE* msg)
{
        int ret = 0;

        if (function == RECEIVE)
                memset(msg, 0, sizeof(MESSAGE));

        switch (function) {
        case BOTH:
                ret = sendrec(SEND, src_dest, msg);
                if (ret == 0)
                        ret = sendrec(RECEIVE, src_dest, msg);
                break;
        case SEND:
        case RECEIVE:
                ret = sendrec(function, src_dest, msg);
                break;
        default:
                assert((function == BOTH) ||
                       (function == SEND) || (function == RECEIVE));
                break;
        }

        return ret;
}
```

我们知道，一个完整的系统调用需要一个来回，那就是用户进程向内核请求一个东西，然后内核返回给它。我们用消息机制来实现这个过程同样需要一个来回，这意味着用户进程发送一个消息之后需要马上等待接收一个消息，以便收到内核（其实是某个系统任务）给它的返回值。这个发送然后马上接收的行为被`send_recv()`这个函数包装了一下，并在`SEND`和`RECEIVE`之外又提供了一个叫做`BOTH`的消息类型。今后我们想要收发消息时，就直接使用这个`send_recv()`，而不再直接使用系统调用`sendrec`。

好了，系统进程 SYSTASK 已经就绪，下面就来修改一下函数`get_ticks`（代码8.14）。

代码 8.14 get_ticks（chapter8/a/kernel/main.c）

```
PUBLIC int get_ticks()
{
        MESSAGE msg;
        reset_msg(&msg);
        msg.type = GET_TICKS;
        send_recv(BOTH, TASK_SYS, &msg);
        return msg.RETVAL;
}
```

```
 ...
void TestA()
{
        while (1) {
                printf("<Ticks:%d>", get_ticks());
                milli_delay(200);
        }
}
```

我们以GET_TICKS为消息类型，不夹带其他任何信息地传递给 SYSTASK，SYSTASK 收到这个消息之后，把当前的ticks值放入消息并发给用户进程，用户进程会接收到它，完成整个任务。

我们来运行一下，结果如图8.7所示。

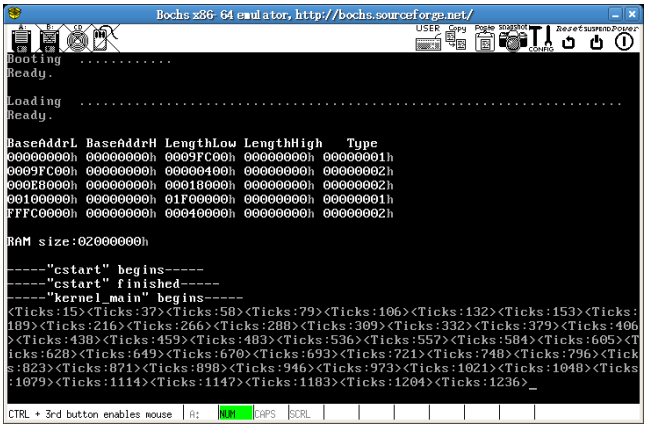

图 8.7 使用 IPC 实现 get_ticks

成功了！进程 TestA 调用get_ticks之后，成功地打印出了它们的值，这表明我们的消息机制工作良好！

8.5 总结

虽然运行结果没有很大改变，但是如今我们的操作系统已经确立了微内核的路线，并且成功地实现了 IPC，即便这算不上是一个质的飞跃，至少我们已经走上了另一个台阶。接下来，基于消息机制，我们将逐步实现硬盘驱动程序、文件系统等内容。而且你将逐步发现微内核的优点，那就是代码相对很独立，结构很清晰，并且内核态的代码今后将很少需要大的改动了。