操作系统功能概述

Norman Matloff
University of California, Davis

December 1, 2004

zchar@bbs.linuxsir.com 翻译
2005 年 8 月 3 日

Hop Lee 整理
（补上了所有的脚注，修改了若干术语。并用 LaTeX 2ε/CTEX 重新排版。）
mailto:hoplee@bupt.edu.cn
2005 年 12 月 23 日

目录

1 介绍

1.1 仅仅是程序!

首先而且最重要的是，明白操作系统（OS）仅仅是一个程序，虽然它非常庞大、非常复杂，不过它仍然是一个程序。OS 提供对加载和处理其他程序的支持（我们以下将这些程序称为**应用程序**），并且操作系统能够建立某些机制，获取某些特权，这些特权是应用程序所没有的，不过最后还是要提醒你，操作系统仅仅只是一个程序。

举个例子，当你的一个程序，假如是 `a.out`[1] 处在运行状态，不过 OS 没有运行，这样你的 OS 将没有能力在 `a.out` 程序运行时加载、中断该程序 — 因为 OS 没有运行！这是一个关键概念，所以让我们先来了解这个描述是什么意思。

上面我们描述 `a.out` 处于运行状态，那么什么是运行状态？计算机的 CPU 会不间断的执行获取代码/执行代码/获取代码/执行代码 ... 的循环，每一次获取过程，CPU 都会去取得 Program Counter 指针所指向的指令。如果当前的 PC 指针指向你的程序当中的一条指令，那么我们称你的程序将处于**运行状态**。每次你程序中的一条指令执行后，

CPU 的循环机制会更新 PC 指针的值（一般就是加一），使之指向你的程序中的下一条指令（通常情况）或是你程序中的任意位置处的指令（若使用了跳转指令）。

需要注意的是，能使你的程序停止运行的唯一方法是使 PC 指针指向另一个程序，比如指向 OS。它是怎样发生的呢？只有两种方法：

- 你的程序可以自动将 CPU 释放给 OS。通常你的程序会通过系统调用 — 由 OS 提供的一系列实现某种有用功能的函数 — 来达到上述目的。

  比如，假设 `a.out` 是由 C 源程序编译而来，其中调用了 `scanf()` 函数。`scanf()` 函数是 C 语言库函数，它在 `a.out` 源程序编译的过程中链接进了目标文件 `a.out`，但是，`scanf()`这个库函数实际上又调用了 `read()` 函数，而 `read()` 函数正是一个系统调用函数（它被包含在操作系统中）。当 `a.out` 这个程序运行至 `scanf()` 指令时，对 `scanf()` 的调用会导致对 OS 的调用，不过当 OS 读入键盘输入后，它又会将资源返回给 `a.out`[2] 程序。

- 另外一种可能原因是产生硬件中断。它是一种发送给 CPU 的信号 — 实际上是在总线中的中断请求线路上的一个物理电流脉冲，来自诸如键盘之类的 I/O 设备。当 CPU 收到该中断后，就会转入一段我们指定的在启动计算机之后的内存空间中。它会存在于 OS 中，因此这时 OS 就会运行。这时 OS 就会监控 I/O 设备，比如，记录键盘的敲击情况，最后，返回到被中断的应用程序。

  注意，在这个敲击键盘的例子中，敲击不一定是由你来完成。当你的程序运行时，其他用户可能会使用键盘。中断会导致你的程序被挂起；OS 会运行发起中断请求的设备的驱动 — 比如键盘，如果该用户使用一台

[1] 或者是一个不是由你自己编写的程序，比如说 `gcc`.

[2] 一个例外是系统调用 `exit()`，它将在你的程序执行完毕后被调用。如果你没有在你的源程序中调用它，编译器将会替你增加一个对它的调用。

设备的 console 或者 internet 接口，如果该用户是从远程登陆，比如通过 `telnet` — 这会在该用户的缓冲区里记录该用户的击键情况；然后 OS 会执行 iret 指令从该中断程序中返回，这样你的程序就恢复了。

硬件中断还可以由 CPU 自己产生，比如说，如果你的程序有除以零的操作，或者试图获取一段超过你的程序限制的内存空间时，CPU 就会自己产生中断了。

所以，当你的程序处在运行状态时，它就是老大，OS 没有停止它的能力。使你的程序停止运行的唯一方法就是自动交出执行权或者是由 I/O 设备的中断行为强行停止。

1.2 OS 有什么作用?

一台计算机可能没有操作系统，比如在一些嵌入式的应用中。嵌入在洗衣机中的"计算机"只会运行一个程序（在 ROM 中）。所以，不会为它担心文件情况、不会有时钟共享的问题、等等 . . . 所以，它不需要 OS。

不过在一台通用计算机上，我们还是需要操作系统来完成以下功能：

- 加载需要运行的应用程序；
- 提供一些服务，比如：I/O、一些应用程序需要调用的系统函数（比如 `read()` 函数）；
- 提供时钟共享机制，这种机制通过**硬件的协调**使得那些本来独立的程序采用分占时钟、循环执行的方法达到看似同时运行的目的；
- 为应用程序提供虚拟存储器，这一机制可以使内存使用更有弹性，并且加强了安全性，这一机制同样是由**硬件协调**的；
- 维护文件系统（比如记录磁盘中所有文件的位置）；
- 控制 I/O 设备，包括网络。

需要仔细掌握 OS 和硬件之间的相互作用。OS 控制 I/O 设备，因此禁止了应用程序的编制者直接处理这些 I/O 设备。比如，假设你希望你的程序读取在磁盘中的某个文件，那么如果使用处理磁盘物理存储单元获取该文件的方法会是一场灾难，与此相反的是，实际上你只是简单的调用了 `fopen()` 和 `fread()` 函数，接下来细节的工作是由 OS 来完成的。由于文件的创建就是由 OS 来完成的，所以 OS 也知道文件的具体位置；当你想完成读取文件的任务时，OS 会找到该文件的位置，并且记录下它们。

另外，如果硬件允许、OS 设计允许[3]，那么 OS 不仅仅**减轻**程序员直接处理磁盘的工作，并且将会**阻止**他这么做。这是出于安全考虑的，我们不会希望程序员有意或无意删除别人存储在磁盘中的文件。

以下部分会讨论 OS 如何实现这些作用，我们会在 UNIX 系统中进行建模讲解，不过关于这些功能的描述适用于大多数现代操作系统。这里假设读者熟悉基本的 UNIX 命令。如果不熟悉的可以参看该教程：`http://heather.cs.ucdavis.edu/~matloff/unix.html`。

[3]例如：Pentium CPU 拥有这项功能，但是 Windows 95 没有使用它，而 Linux, Windows NT 及 NT 之后的版本使用了它。

2 系统启动

正如将在后面介绍的，当我们希望运行一个应用程序时，操作系统会将它装载进内存。但是操作系统本身是如何被装载进内存并且执行的呢？处理上述过程的数语称为 **bootup**.

经过设计的 CPU 将会使计算机启动后的 PC 指针指向一个特殊的位置，比如 Intel 的处理器会使 PC 指针指向 `0xfffffff0` 这个内存地址。并且那些**组装**计算机的人（就是将 CPU，内存，总线系统等等装配在一起的人）会把 `0xfffffff0` 这个地址付予一块 ROM 存储器，该 ROM 存储器里就包含了 boot loader 程序。所以，一旦计算器启动，boot loader 程序就会运行。

boot loader 程序的目的是将 OS 从磁盘中装载进内存中，一种比较简单的方式是，boot loader 程序会读取磁盘中的一段特定的区域，将该区域中的内容（就是 OS 了）拷贝进内存中，然后在boot loader 程序的最后执行一条 `JMP` 指令（或类似的跳转指令）跳转到该内存单元 — 此时 OS 便开始运行了。还有一种比较复杂的方式是，boot loader 程序只将操作系统的一部分装载入内存，然后就跳转至该内存单元运行 OS，并且由 OS 自己将它的剩余部分装载进内存。

让我们看看具体的例子，以 Intel 为例，在 ROM 中的程序称为 **BIOS**，它包含了该计算机的部分硬件驱动[4]，并且包含了 boot loader 程序的第一部分。

在 BIOS 当中的 boot loader 程序用来读取某些信息，现在就可以告诉你的是，那些信息存储在物理磁盘的第一个块中[5]。这个块也被称为 **Master Boot Record (MBR)**。

典型的，一个操作系统会定义磁盘的分区，如果假设一块磁盘有 1000 个柱面，那么我们可以将前 200 个柱面作为一个分区，剩下的 800 柱面[6]作为第二个分区。这些分区由 MBR 中的分区表[7]来确定。主分区中的确切的一个将在这个表中被标记为 active，意味着可引导的。

MBR 中还包含了 boot loader 程序的第二部分。在 BIOS 中的 boot loader 将会加载这部分代码至内存，然后跳转过去，因此该程序就会运行。该程序将读取分区表，以决定哪个分区是被 active 的。然后该程序会转到该分区的第一部分，在那里将 boot loader 的第三部分读入内存，然后跳转至该部分程序。

现在，如果计算机最初是安装的 Windows 操作系统，那么在 MBR 中的第二部分 boot loader 代码会将 Windows 分区中的第三部分代码加载入内存。另一方面，如果该机器安装的是 Linux 或是其他的 OS，那么 MBR 中的代码也会做相应变化，以使对应的操作系统能被加载进内存中。

很多 Linux 用户在他们的机器中保留了 Windows 操作系统，并且有一个双启动配置。他们最开始使用 Windows，不过后来又装上了 Linux，这样，这台机器上就同时存在两个 OS。作为 Linux 安装过程的一部分，旧的 MBR 被拷贝至别处，一个新的叫做 LILO(Linux Loader) 的程序将被写入 MBR 中。换句话

[4]这些驱动是否能被用到随操作系统的不同而有所区别。Windows 可以使用这些驱动，而 Linux 就不行。

[5]启动设备可以是硬盘之外的各种存储设备，比如软盘、光盘以及 U 盘等。到底按照一个什么样的顺序来检查启动设备是在 BIOS 中进行设置的。

[6]通常一块硬盘是由若干片圆盘状介质叠在一起构成的。假设某块硬盘有四张盘片、十五个柱面，那么每张盘片上都将会有十五个柱面。而“柱面”这个名字显然是从硬盘结构的几何视角上得到的创意。

[7]请注意，分区之间在物理上并不是分开的，这个分界线仅仅是定义在分区表中的一个逻辑概念。

说，LILO 将会称为 **boot loader** 的第二部分程序。LILO 会让用户选择是启动 Windows 还是 Linux，然后到相应的分区装载被处理第三部分引导程序[8]。

无论怎样，当 OS 被第三部分引导程序装载进内存中后，该引导程序会跳转至 OS 代码的位置，这时 OS 就开始运行了。

3 应用程序的装载

假设你在 Linux 操作系统下编译完成了一个可执行文件 `a.out`，你使用如下命令运行它：

```
% a.out
```

如果这次你使用的是非常简单的设备/OS 组合，没有虚拟内存，那么下面是发生的事情：

- 当你敲击上述命令时，shell 处在运行状态，可能是 `tcsh` 或 `bash`。另外，shell 仅仅是一个程序，它使用 `printf()` 函数输出 % 提示符，并且在一个循环中使用 `scanf()` 函数记录你敲击的命令[9]；
- 然后 shell 会产生一个系统调用 — `execve()`[10]，要求 OS 运行 `a.out` 应用程序。OS 在这时开始运行；
- 该 OS 会查看它的磁盘索引，以决定 `a.out` 在磁盘的哪个位置。它会读取 `a.out` 的最开始部分，该部分包括该应用程序的大小、该程序使用的数据段列表等等；
- OS 会查看它的内存分配表（只是 OS 中的一个数组）以查找一段足够大的未用的内存空间来给 `a.out` 使用。（值得注意的是，OS 正是这样加载了目前所有的正在运行的程序，所以它知道目前有哪些内存已经被使用，哪些内存是空闲的。）我们需要为 `a.out` 的指令（UNIX 术语称为**程序中的 text 部分**）和数据 — 静态数据项（标量和数组变量，在 UNIX 中称为 **数据段**），以及栈空间和堆空间（被 `calloc()` 函数和 `malloc()` 函数所使用）留出足够的空间；
- 接下来 OS 将 `a.out` (包括 text 和 data)装载进上一步所分配出来的内存空间里，然后相应的更改它的内存分配表；
- OS 会检查`a.out` 文件的某一特定部分，在这里留下了链接器对 `a.out` 文件入口点的记录，也就是文件执行的第一条指令；（以我们前面的编译语言为例，这里是 `_start` 指令。）
- 现在OS准备好初始化 `a.out` 的执行了。它会将栈指针指向先前为 `a.out` 分配好的栈空间。（操作系统会保存它自己的寄存器值，包括之前的栈

[8]这个分区不必是“激活的”(active)。术语“active”仅仅适用于 Windows 启动过程的第二部分，这个部分已经被安装在 MBR 中了。

[9]请注意：你每敲一个键都会产生一个中断，OS 会将这些字符缓存起来直到你敲回车键为止，这时 OS 才会“唤醒” shell 程序、`scanf()` 才会被执行。参见后面有关 Sleep 状态和 Run 状态的部分。

[10]它还会调用一个系统调用 — `fork()`，其详细情况已经超出了本文的范围。

指针的值。）然后它会将所有命令行参数送到 a.out 的栈空间里，然后开始执行 a.out，比如执行一条 JMP 指令（或类似功能的）跳转至 a.out 的入口点；

- 现在，a.out 就开始运行了！

需要注意的是，a.out 可能会调用 execve() 来运行其他程序。比如，gcc[11] 就这么做。它先运行 cpp C 预处理器（它会预编译你的 C 源文件中的 #include, #define语句）。然后 gcc 会运行真正的编译器 — cc1（正如你所见的，gcc 本身只是运行其他组件的管理者），这一过程会产生一个汇编语言文件[12]。然后 gcc 会运行 as — 汇编语言编译器 — 来生成 a.o 机器代码文件。后者必须被某些代码，比如 /usr/lib/crt1.o，和一些其他的构成 main() 结构的文件，比如 argv 命令行参数的入口，所所链接，也要链接到 C 库，/lib/libc.so.6；所以，gcc 会运行链接器 — ld。以上所有例子中，gcc 都是通过调用 execve() 系统调用来运行这些程序的。

3.1 巩固上述概念：自己尝试这个命令

UNIX 系统命令 strace 会报告你执行的应用程序的系统调用。使用方法如下：

```
% strace a.out
```

你将会看到在一系列不同的系统调用之前有 execve() 的身影，正是由它来启动 a.out 程序的。

4 分时

4.1 许多进程，那么轮换吧

分时是一种使许多正在运行的程序[13]看上去是同时运行一样的方法。因为系统仅仅拥有一块 CPU（在这里我们排除多处理器系统的特例），所以这种“同时”只是一种假象，因为在一个给定的时间里，只能运行一个程序，不过正如我们将要看到的，这是一种很有价值的假象。

第一个问题是，我们如果获得这种假象呢？答案是，我们让所有程序轮换着运行，使每个程序能占用的轮循时间 — 称为**量子**或**时间片** — 都很短，比如 50 毫秒。假设我们马上就要运行四个程序，u，v，x 和 y，想想下面这样会造成什么：首先 u 运行 50 毫秒，接着 u 被挂起、v 运行 50 毫秒，然后 v 被挂起、x 运行 50 毫秒...就象这样继续（当 y 完成它的 50 毫秒后，u 开始第二次运行）。因为轮循（一般称为**上下文切换**）发生的如此之快（每 50 毫秒一次），所以我们人类会觉得每个程序都是同时在运行的（虽然只有四分之一的实际速度），而不会查觉到程序的运行、挂起、运行挂起...

[11] 请记住：编译器也是一个程序。虽然它非常巨大、非常复杂，但是从本质上说它和你自己编写的程序没有什么区别。

[12] 你可以通过运行带有 -S 选项的 gcc 命令来浏览这个汇编文件。当你需要编写一个被 C 代码调用的汇编子程序时，这个手段将会显得非常方便，因为你可以直观地看到编译器是如何处理调用汇编子程序时的参数的。

[13] 这些程序可以来自于同一个用户也可以来自于不同的用户。

但是 OS 是如何切换这些时间片的呢？举例来说，OS 如何使上面的程序 u 在运行 50 毫秒之后准时停止呢？结合前面的想想，答案是：“操作系统不能，因为在 u 运行的时候，OS 并没有运行！”取而代之的，这些轮换是通过一个定时器来实现的，这个设备会在合适时间发送一个硬件中断。比如，我们可以让这个设备每 50 毫秒发出一个中断，我们可以编写一个定时器的驱动，并将它并入 OS 中[14]。

计数设备驱动可以储存 u 程序的所有寄存器值，包括它的 PC 指针值和 EFLAGS 寄存器值。稍后，当 u 程序获得执行机会时，这些值会恢复，这样 u 程序就可以看似连续的执行下去了。不过现在，按照 OS 的计划，最后将恢复程序 u 先前的所有寄存器值，特别要保证恢复上一轮的 PC 值，这最后一步会强制执行一个从 OS 到程序 v 的跳转，跳转到它上一次时间片挂起的正确位置。（另外，CPU “只操心自己的事情”，而并不会知道 OS 已经把控制权交给另一个程序 v 了；CPU 只是不断循环获取 PC 指针指向的数据并且处理这些数据罢了。）

最新的 CPU 会运行于两个或多个特权级别。我们的例子由于安全原因不会涉及到普通的存取 I/O 设备的应用程序，比如磁盘驱动。因此 CPU 被设计为让一定的指令 — -比如那些处理 I/O 的 — 只能在运行在高特权级别上，这个级别称为**内核模式**（内核这个术语来自于 OS）。除了别的之外，计数器发出的中断就会将 CPU 置于内核模式，所以该中断不仅使 OS 得以运行，还是 OS 获得它需要的权限。

在任何给定时间，内存中都会有许多不同的进程。它们是程序执行的实例。如果现在有三个用户在同一台给定设备上运行 gcc 编译程序，那么会对应于一个程序产生三个进程。

4.2 OS 代码例子：Linux For Intel CPUs

这里是一个在 Intel 架构设备、Linux 操作系统下的关于上下文切换的例子[15]。

操作系统会维护一张由结构数据量数组构成的进程表，每一个结构数据量都为一个进程存在。这个结构数据量称为**对应进程的 TSS(Task State Segment)**，并且存储了该进程的不同信息，比如对应程序最后一次执行完成后的寄存器值。

作为描述上述行为的例子，并且为了实实在在的向你说明 OS 就是由真实代码组成的程序，下面是一段典型的代码摘录：

```
pushl %esi
pushl %edi
pushl %ebp
movl %esp, 532(%ebx)
...
movl 532(%ecx), %esp
```

[14]我们在这里将作这样一个假设：通常的情形是将定时器的中断间隔设置得略小于时间片的长度。在普通的 PC 机上，8253 定时器每秒钟将产生 100 次中断，而 Linux 操作系统每收到六次定时器中断就进行一次上下文切换，这样就可以得到长度为 60 毫秒的时间片长度。通过改变触发上下文切换所需的定时器中断次数，我们可以方便地更改时间片的长度。

[15]这个例子是基于版本为 2.3 的 Linux 内核的。当然，为了阅读更清晰，对代码进行了一些简化。

```
...
popl %ebp
popl %edi
popl %esi
...
iret
```

这段代码是计时器的 ISR（中断服务例程）[16]。紧接着入口的背景是 u 程序刚刚执行完，并且被计时器所中断。OS 已经指向了刚刚结束的进程 u，以及即将开始的进程 v 的 TSS 中的寄存器 EBX 和 ECX[17]。

下面是这段代码的作用。Linux 的源代码包括一个变量 tss，它是目前进程的 TSS 结构变量。在该结构中是一个名为 esp 的字段。因此，tss.esp 包含了这个进程以前存储的 ESP 值、栈指针；这一字段恰巧占据 TSS 的头 532 个字节[18]。

现在，在进入上面这段 OS 代码之前，ESP 仍然指向 u 程序的栈，所有，前面三条 PUSH 指令将 u 程序的 ESI、EDI、EBP 寄存器值存储到它自己的栈空间中。其他 u 程序的寄存器值也必须被存储[19]，包括它的 ESP 的值。后者由 MOV 指令完成，该指令将目前的 ESP 值 — 也就是 u 的 ESP 值 — 复制到 u 程序的 TSS 结构中的 tss.esp 中。其他寄存器值的存储方式差不多，所以不在这里介绍了。

现在，OS 必须准备开始执行这一轮的 v 程序进程了。因此 v 程序上一轮的寄存器值必须被重载入寄存器中。为了理解这是如何完成的，你必须清楚在 v 的上一轮结束时，也执行了和上面相同的指令。因此，v 程序的 ESP 值就存储在它自己的 TSS 的 tss.esp 中，上面代码的第二条 MOV 指令正式将这个值重新赋予 ESP，因此，我们现在使用的就是 v 程序的栈空间了。

下一步，注意到在 v 程序的上一轮结尾，它的 ESI、EDI、EBP 值被 push 进了它自己的栈中，并且理所当然的，这几个值仍然在相应栈空间中。所以，后面的三个 POP 指令将这几个值重新装载进相应的寄存器中了。

最后，实际上是什么使 v 程序的新轮循开始的呢？要回答这个问题，需要明白如下机制：导致 v 程序的上一轮结束的是从计数器发出的一个硬件中断信号。那时，FLAGS 寄存器、CS 寄存器、PC 寄存器的值是被 push 进相应栈空间中了的。现在，你看到的 IRET 指令将这些玩意儿全都重新装载回对应的寄存器中了。由于 v 程序上一轮存储的 PC 值重新装入 PC 寄存器中，因此 v 程序就开始运行了。

4.3 进程状态

OS 维护一张进程表（process table），这张表显示了内存中每个进程的状态，其中最主要的是 Run 状态和与之相反的 Sleep 状态。一个进程处于 Run

[16] 更准确地说，这段代码是被 ISR 调用的。

[17] OS 能够记住当前正在处理哪个进程，因此它知道应该将 EBX 指向何处。OS 还知道下一个应该轮到哪个进程，因此它也能正确地设置 ECX.

[18] 在 C 语言源代码中，这一字段是通过 tss.esp 来引用的，而在汇编语言中，由于 struct 的名字无法得知，我们只能用 532 来引用这一字段

[19] 请注意：这个操作只在汇编语言中是必要的，因为 C 语言没有提供直接访问寄存器的手段。Linux 的绝大部分是用 C 语言编写的，但是涉及道与硬件相关的操作 — 现在的情形 — 则需要用汇编语言来实现。

状态意味着它已经作好了运行的准备只待下一轮循的到来了。OS 会循环的检查进程表，并且将处于 Run 状态的进程调入运行轮循，然后忽略掉那些处于 Sleep 的进程。处于 Sleep 状态的进程一直在等待某些触发物，典型的如一次 I/O 操作，因此它们都暂时不能进入轮循。所以，每次轮循结束后，OS 都会浏览它的进程表，寻找一个处在 Run 状态的进程并且让它进入运行轮循。

假设我们上面的 u 程序包括一个 scanf() 函数来记录键盘输入。回忆起 scanf() 函数调用了系统函数 read() 来完成它的功能，后者会检查在键盘缓冲器中是否有任何准备输入的字符。一般情况不会有任何准备好的字符，因为用户还没开始输入。那么此时，OS 就会将该进程置为 Sleep 状态，并且开始另一进程。

那么一个进程如何从 Sleep 状态转换为 Run 状态呢？假设如上面所说，u 程序由于等待用户输入而正处于 Sleep 状态（假设它只用等待一个字符的输入）。正如前面所解释的，当用户敲打键盘时，会产生一个硬件中断信号，该中断将会强制一个到 OS 的跳转。假设当时 v 程序的进程正好处于时间片的中间，CPU 将会暂时将 v 程序挂起，然后跳转执行 OS 中的键盘驱动程序，OS 会发现 u 进程目前处于等待键盘输入的 Sleep 状态，因此 OS 将会把 u 进程转至 Run 状态。

需要注意的是，尽管 OS 将 u 进程置于 Run 状态了，但是 u 的轮循并没有开始，u 只是简单的处在可以运行的状态了。请回忆，每当一轮进程结束时，OS 会从处于 Run 状态的进程中选择进入下一轮的进程，而 u 就处于该状态了。

为了巩固上面所述的，让我们看个例子，假如 u 程序是运行 vi 编辑器， v 是运行一个冗长的运算程序，并且用户 w 也在运行一个大计算量的程序 w 。记住，vi 是一个程序，它的源程序中也许会包含以下代码片段：

```
while (1) {
  scanf("%c", &KeyStroke);
  if (KeyStroke == 'x')
    DeleteChar();
  else if (KeyStroke = 'j')
    CursorDown();
  else if ...
}
```

这里你可以看到 vi 是如何读入用户输入的，比如 x（删除一个字符）、j（使光标下移一行）等。当然，DeleteChar() 等上面出现的函数的源代码并没有在这里写出。在 u 程序的轮循中，vi 程序会遇到 scanf() 行，后者调用 read() 系统调用，于是此时 OS 就会运行。假设在 u 程序在运行时，用户还没有从键盘输入，这时 OS 就会将 u 进程设置为 Sleep 状态。

接下来 OS 会选择进程表中的一个处于 Run 状态的进程，并且使它进入 CPU 的下一轮处理。这里假设是 v 程序，它会一直运行直到计时器中断的到来。这一电流中断脉冲会导致 CPU 跳转至 OS，这时 OS 就会再次运行，这次，假设程序 w 会被执行。

假如在 w 的运行期间，u 程序终于得到一个键盘输入，那么如上文所说明的那样，这个键盘输入产生的硬件中断会强行导致 CPU 跳转至 OS，OS 会读入 u 程序的键盘输入，需要注意的是，这时 OS 会将进程表中原本处于 Sleep 状

态的 u 进程设置为 Run 状态。接着，OS 会执行 IRET 指令，这会使 w 进程继续运行，不过当下一个计时器中断到来之后，OS 可能会使 u 进程再次运行。

4.4 关于后台作业

假如你有一个名为 a.out 的程序，你想长时间的运行它。而在它运行时，你还想去处里其他事，比如使用 vi 编辑器编辑文件 xyz。在 UNIX 操作系统中，你可以输入以下命令：

```
% a.out &
% vi xyz
```

使用“&”号的意思是：“在后台运行该作业。”这又意味着什么呢？

答案是实际上它对于 OS 来说什么都不意味。在 OS 进程表中 a.out 进程是后台还是前台是不会被说明的，它仅仅就是个简单的进程。

“&”只会对 shell 有意义。我们使用该符号告诉 shell：“请为我运行 a.out 程序，不过不要等到它运行完了才给我“%”提示符，请立即给我该提示符，因为我想做一些其他事情。”

4.5 巩固上述概念：自己尝试这些命令

首先试试 ps 命令，在 UNIX 系统中，它会告诉你很多关于目前进程的信息，包括：

- 状态（Run，Sleep 等）
- 页使用情况（有多少页、多少页故障等；参看下面的虚拟存储部分）
- 族谱（该给出进程的父进程）

我们极力推荐读者自己试试。你会在看到 ps 命令的输出后更加明白我们上面提到的概念。该命令的格式会因系统不同而不同（在 UNIX 系统上，我建议使用 ps ax），所以请查看 man 手册页获取更多细节，不过使用的选项越多越有可能得到完全的输出信息。

另一个需要尝试的命令是 w，该命令所给出的信息中有一项是在过去几分钟里处于 Run 状态的进程的平均数。这个数字越大，那么用户能查觉的计算机的响应时间就越慢，就象它的程序和更多其他用户的程序一同运行一样。

在 Linux 系统（以及一些其他的 UNIX 系统）上，你也可以试试 pstree 命令，该命令以图示的方式显示了关于每个进程的“家族树”（族谱关系）。举个例子，下面是我的 CSIF PC 机上使用该命令的输出：

```
% pstree
init-+-atd
 |-crond
 |-gpm
 |-inetd---in.rlogind---tcsh---pstree
 |-kdm-+-X
 |     `-kdm---wmaker-+-gnome-terminal-+-gnome-pty-helpe
 |                    |                `-tcsh-+-netscape-commun---netscape-+
```

```
 |                    |                       `-vi
 |                    |-2*[gnome-terminal-+-gnome-pty-helpe]
 |                    |                     `-tcsh]
 |                    |-gnome-terminal-+-gnome-pty-helpe
 |                    |                `-tcsh---vi
 |                    `-wmclock
 |-kerneld
 |-kflushd
 |-klogd
 |-kswapd
 |-lpd
 |-6*[mingetty]
 |-2*[netscape-commun---netscape-commun]
 |-4*[nfsiod]
 |-portmap
 |-rpc.rusersd
 |-rwhod
 |-sendmail
 |-sshd
 |-syslogd
 |-update
 |-xconsole
 |-xntpd
 `-ypbind---ypbind
%
```

如果是 UNIX 系统，当系统启动后 OS 做的第一件事就是开始一个名为 `init` 的进程，该进程会是其他所有进程的父进程（或是爷爷进程、祖父进程等等）。

`init` 进程然后启动一些 OS 守护进程。守护进程是 UNIX 的一种说法，意思是某种服务程序。在 UNIX 中，该种程序以“d”结尾，代表 “daemon”。

举个例子，你能从上面的 `pstree` 输出中发现 `init` 进程启动了 `lpd` 守护进程，这是关于打印机的服务进程[20]。当用户使用 `lpr` 或其他打印命令，OS 会将这些命令交给 `lpd` 守护进程，该进程会实际分配完成打印任务。

守护进程经常产生更进一步的程序。看看下面这条线：

```
|-inetd---in.rlogind---tcsh---pstree
```

`inetd` 是 Internet 请求服务，系统管理员可以运行它来代替那些不怎么使用的网络守护进程。这里该用户（也就是我）已经完成了一个 rlogin，这是一个登陆程序比如 ssh[21]，用来远程登陆到该系统。系统管理员猜想 rlogin 不会频繁使用，所以它们不会在启动之后由 `init` 来启动相应的守护进程，也就是 `in.rlogind`。替代的是，任何不在启动时运行的网络守护进程都会由 `inetd` 来启动，正如你在这里看到的如 `in.rlogind` 守护进程这样的。后者又为登陆上

[20] 另一个常见的 UNIX 打印服务是 `cupsd`.

[21] 但是在 CSIF 中已经不允许了。

来的用户（也就是我）启动一个 shell，然后该用户（我）运行 pstree 命令。再次注意的是，shell 才是 pstree 运行的实体。

5 虚拟存储

5.1 确定明白你的目的

下面我们介绍虚拟存储技术(Virtual Memory)。
VM 实现以下基本目的：

- 克服内存容量的局限：我们想要能够运行一个或一系列占用超过物理内存空间的程序和程序组；
- 在程序运行时释放编译器和链接器的负担，使它们不用知道哪些内存可以使用：我们想促进程序的再装载，意味着编译器和链接器不用再关心程序运行时应该从内存的哪个位置将程序装载；
- 加强安全：我们想要确保一个程序不会有意无意的通过写入其他程序占据的内存空间而破坏后者；
- 开启共享：我们想要能够将一个大程序的文本部分仅仅在内存中保存一份拷贝（比如一个编译器），尽管有一些用户都在运行该程序的实例。

5.2 虚拟自然地址的例子

“虚拟”在这里意味着“看上去的”。看上去一个程序的整体都会驻留在主内存中，但是事实上只有一部分；看上去（从编译器的观点上来看）程序会从内存中的最开始部分进行装载，但并非如此。

（为了简单，在讲解 VM 的时候不会考虑缓存的情况，后者会在稍后部分介绍。）为了使上述概念形象化，假设我们 a.out 这个程序的 C 源程序代码中包含一行声明：

```
int x;
```

并且假设编译器和链接器将地址 200 赋予了 x。换句话说，在我们源程序中如下一行的代码：

```
printf("%d",&x);
```

会打印出 x 的地址：200。那么，在一台 Intel 设备上 a.out 将会包含类似的汇编指令：

```
movl 200, %eax
```

这条指令将内存中 200 地址单元的内容拷贝给了 CPU 寄存器 EAX。

在 a.out 被 OS 载入内存的时候，OS 会将指令部分和数据部分分成块，并且在内存中寻找未被使用的空间放置这些块。这些块被称为**程序的页**，而 OS 寻找到的在内存中的相同大小的空间称为**内存的页**[22]。OS 将会建立一张页表，

[22]栈也会分为页。请注意，这些“内存页”仅仅只是一个概念，两个“内存页”之间并不存在任何物理边界。

这是由 OS 维护的数组，OS 在里面记录相关信息，也就是记录程序的每一页放置在内存的哪一相应页中[23]。

所以，在上述代码中，看似在内存中 200 字空间中的内容可能实际上却在 1024 字空间中。当 CPU 处理该指令时，CPU 会判断“200字空间”实际在哪个位置，通过查询页表。在这个例子中，我们实际上要寻找的内容是在 1024 字单元中，因此，CPU 实际上会从那个地址中读取数据。

在这个例子中，我们称 200 为**虚拟地址**，1024 为**物理地址**。

5.3 关于怎样实现这一目标的概览

让我们根据 5.1 节中的目标来看看如何实现它们：

- 克服内存容量的局限：

  为了保存内存容量，OS 最开始只会将 `a.out` 部分装载入内存中，剩下的留在硬盘中。那些未被装载的程序的页会在页表中被标记为目前未安置，并且在该表中可以查到它们在硬盘中的位置。在程序处理阶段，如果该程序需要某个未安置的页，那么 CPU 将会注意到该页是未安置的（这被称为“ **页错误**”），并会产生一个内部中断[24]。这会导致到OS的跳转，然后 OS 会将该未安置页从硬盘中装入内存，然后跳转回该程序，后者将会开始执行需要读取该页的指令。

  注意到在这种管理机智下页经常在硬盘和内存中移动。每当程序需要一张未安置页，那么该页就从硬盘中读入内存，同时内存中会有一页回到硬盘中以给这个新成员腾个空间。

  一个大问题是，面对在页错误发生后从硬盘中带来的缺失页，OS 使用什么样的算法来决定将哪张内存页移回硬盘（也即是哪页被取代）。这些概念超过了本文档的说明范围，不过要知道，该算法的选择将基于使大多数程序都良好运行这一点。对有些程序来说，不良的算法会导致许多页错误，由于该种算法会造成在很短的时间内被替换的页又需要被载入内存这一问题。

- 在程序运行时释放编译器和链接器的负担，使它们不用知道哪些内存可以使用：

  这一点已经由上面的例子说清楚了，由编译器和链接器为 x 设置的位置 200 在程序被装载时由 OS 改成 1024了。OS 将这一信息记录在页表中，在程序运行过程中，CPU 中的 VM 硬件查询该表以获得正确的地址。

- 加强安全：

  在页表中，每一页都包含一个入口。如前述，该入口包含了该程序页当前在内存中的地址信息，或者如果该页目前还未安置的话，它在硬盘中存储的位置信息。不过更多的，该入口还会罗列相应程序能够获得的这页的权限 — 读、写、执行 — 同文件权限一样。如果程序想获取页未付予它的权限，也就是说，发生了访问破坏，那么 CPU 中的 VM 硬件就会产

[23] 请记住：OS 是一个程序，而页表是这个程序中的一个数组。
[24] 中断号为 `0xe`，即 IDT 中的 `0xe` 入口。

生一个内部中断，导致 OS 运行[25]。OS 将会杀死该进程，也就是将它从进程表中移除。

- 开启共享：

  假设现在一台设备上有两名用户想运行 gcc。它是个巨大的程序，所以，保留内存就相当重要了。OS 会采取的措施显而易见的会包括将 gcc 的指令部分只做一份拷贝，当然，数据部分的拷贝将会分开，因为这两名用户的数据（C 源文件）很可能不同。

  VM 允许我们实现上述目标。每个用户的页表会拥有相同的访问指令部分的入口，因此他们会共享程序的指令部分。

5.4 创建和维护页表

请仔细弄清楚参与者的角色：是软件，也就是 OS 创建和维护页表，但是实际上是硬件使用页表来产生地址、检查页的位置情况、检查安全情况的。

当 OS 产生一个新的进程时，它必须从内存中找到正确的页来分配给新进程的部分程序使用。它会为该进程创建页表，并在该表中记录页的位置（当然也会把那些没有装入内存中的程序在硬盘的位置记录下来）。

相应的硬件拥有一个名为页表寄存器（PTR）的特殊寄存器，用来指向当前进程的页表。当 OS 将该进程置于新的运行轮中，该进程会重新装载上一轮保存的 PTR 的值，使该进程的页表有效[26]。

5.5 关于页表的使用细节

5.5.1 虚拟-物理地址翻译，页表查询

每当运行的程序产生于一个地址时 — 无论该地址是指令地址还是数据地址 — 该地址都是虚拟的。它必须被转换成存储实际内容的物理地址。CPU 中的轮循就被设计来完成该翻译，通过查询页表实现。

地址空间被分成页。为了方便起见，假设页大小为 4096 字节。对每个虚拟地址而言，虚拟页号等于该地址除以页大小，也就是 4096，该页的偏移量等于该地址除以 4096 的余数。因为 4096 等于 2 的 12 次方，这意味着一个 32 位的虚拟地址，它的高 20 位构成页地址，低 12 位是偏移量[27]。

参看下面的 Intel 指令：

```
movl $3, 0x735bca62
```

该指令会将实数 3 拷贝至 0x735bca62 地址单元中（10 进制就是 1935395426）。也即说明了虚拟页号为 0x735bc ，偏移量为 0xca62 。也就是说，我们将把第一个字节写入 0x735bc 这一页的 0xca62 这个虚拟地址字空间中。

[25] 你可能会对为什么要包括执行权限产生疑惑，当我们使用一个指向函数的指针时这项检查将会非常有用，比如说我们忘了初始化这个指针，那么就会产生一个违例，而这正是我们希望的。

[26] 在 Pentium 机上，PTR 的名字是 CR3. 实际上 Pentium 的页表使用了一个两级的层次结构，具体细节我们就不讨论了。

[27] 你可以将页中的偏移量类比为长途区号之后的电话号码。比如说我的电话号码为 (530) 752-1953, 其中，括号、连字符和空格只是为了阅读清晰加上去的，并不是电话号码的一部分，因此我的电话号码实际上是 5307521953.（也可以是 0015307521953，其中 001 是美国的国际区号，但是为了简单，我们就不考虑国际区号了。）它是由区号 530 和此区号范围内的号码 7631953 共同构成的。正如一个内存地址是由某一页中的页偏移量决定的一样。

假设我们页表中的入口为 32 位，也即是每入口占据一个字单元[28]。让我们来标明入口的 0 到 31 位，其中 Bit 31 处在最左边，Bit 0 处在最右。假设入口的格式如下：

- Bit 31 — Bit 12：代表物理页地址如果 Resident 置位，如果 Not 置位，代表硬盘地址；
- Bit 11：如果该位为1代表 Resident 置位，0 代表 Not 置位；
- Bit 10：如果为 1，代表拥有读权限，0 则相反；
- Bit 9：如果为 1，代表拥有写权限，0 则相反；
- Bit 8：如果为 1，代表拥有执行权限，0 则相反；
- Bit 7 — Bit 0：一些其他信息，这里不会涉及。

那么，CPU 处理上面哪条 MOV 指令会发生什么呢？

- 首先，CPU 发现该虚拟页地址为 `0x735bc` 后，会从页表中获得相应的入口，假设 PTR 的内容为 `0x256a1000`，那么实际的位置将是 0x735bc * 4 + 0x256a1000 = 0x2586e6f0。然后，CPU 从该位置获取入口数据，假设是 `0xc2248eac`。
- 然后 CPU 查看 Bit 11-8 位，获得 `0xe`，知道该页 Resident 置位，并且该程序拥有读写权限但是没有可执行权限。该 MOV 指令的需要是获得写权限，由此看来没有问题。
- 接着 CPU 查看 Bit 31-12 位，获得 `0xc2248`，虚拟偏移量为之前确定的 `0xca62`，因此，CPU 会获知虚拟位置 `0x735bca62` 的物理地址为 `0xc2248a62`。CPU 将该地址存入 MAR(内存地址寄存器)，将 3 放入 MDR(内存数据寄存器)，并且从总线中找出写入线，并将 3 写入 `0x2248a62` 这个地址单元中。至此我们就完成了相应任务。

顺便说一下，所有这都是为了完成该 MOV 指令的 c 步骤。步骤 a 也会去做相同的事情。PC 指针会被分成两部分：虚拟页号和偏移量，虚拟页号作为页表的索引，然后将查看页表元素的 10 位和 8 位，以确定是否拥有读和执行该指令的权限；假设这些权限都是够的，那么物理页号会通过页表元素的 31 到 12 位的值来确定；该物理号会和偏移量一起组成物理地址，该物理地址会被放入 MAR ，通过它获取相应的指令。

5.5.2 页错误

假设上面那个例子中页表入口的第 11 位为 0，这代表了需要的页并没有装入内存中，这一事件就被称为**页错误**。如果它发生了，CPU 会产生一个内部中

[28]假如我们去查看 OS 的源代码，就会发现页表是保存为一个很长的 `unsigned int` 类型的数组的，数组中的每一个元素对应一个页表入口。

断[29]，会导致到 OS 的强行跳转。OS 首先会决定将内存中的哪个已置位页[30]写回到硬盘[31]以腾出空间，然后它会将请求的页从硬盘中写入相应位置，此后 OS 会更新页表中的两个入口参数：

- 它会修改被覆盖的页的入口参数，将第 11 位置 0，并重置 31 到 12 位；
- 它会更新新载入的页的入口参数，指明该页已经装载入内存了，并且说明它的位置。

因为读写硬盘的速度远远慢于读写内存的速度，因此，如果页错误出现的太多，程序的运行速度就会非常迟缓。举个例子，假如你家里的 PC 机内存不够，那么你会发现装载大型软件的时候需要等待很长时间，并且硬盘指示灯会闪得很厉害，就是因为 OS 要将许多当前置位的页写回硬盘、并从硬盘中装载入新的页。

5.5.3 访问破坏

另一方面，如果发生一个访问破坏，那么 OS 会声明一个错误 — 在 UNIX 系统中，把这个错误称为**段错误** — 并会导致进程被杀死，也就是从进程表中移除。

举个例子，参看下面的代码：

```
int q[200];

main()
{
  int i;
  for (i = 0; i < 2000; i++){
    q[i] = i;
  }
}
```

注意到编程者很显然在循环结构中犯了个错误，他将叠代数 200 替换成了 2000 。C 编译器在编译过程中会忽略该错误，并且编译出的机器代码在执行过程中也不会检查到数组索引超界了。

如果该程序在非 VM 平台上运行[32]，那么它可以畅快的没有任何明显错误的运行。它会简单的在 q 数组后写入 1800 个字，这能否造成破坏就取决于多余的字数据的目的了。

不过在 VM 平台上，这里就是 UNIX 系统下，实际上会报告一个错误，和一个“ 段错误”的信息。不过，在我们深入理解该错误之前，它出现的时间或许会另你吃惊。这个错误不像是在 `i=200` 时出现的，而像是在此之后很长一段时间才出现的。

[29]CPU 会将导致页错误的指令的 PC 值记录下来，以便页错误恢复后继续执行这条指令。Pentium 中是由 CR2 寄存器来存储这个值的。

[30]在这里采用了“最近被使用的”原则，和 cache 的原理类似。

[31]不过在这里我们将不会作这个假设，大多数的 OS 仅在必要时执行这个写回操作。为此，Bit 0 — 7 中的某个比特将作为 **Dirty bit**. 详细情况我们就不涉及了。

[32]回忆一下，我们在前面提到过：“VM 平台”不仅需要 CPU 具备 VM 功能，而且 OS 也必须有相应的支持。

为了说明它，我选择在 gdb 下运行此程序以便我能够查看 q[199] 的地址[33]。在运行该程序之后，我发现段错误不是出现在 i=200处，而是出现在 i=728 的时候。让我们看看这是为什么。

通过查询 gdb 我发现数组 q 是在地址 0x080497bf 处结束的，也就是说，q[199] 这个最后的元素处于该地址空间中。对于 Intel 的机器，页大小为 4096 字节，所以一个虚拟地址被分成 20 位的页号和 12 位的偏移量，正如在 5.5.1 节中所述的那样。在我们目前这个例子中，q 结束于虚拟页号为 0x8049、十进制为 32841，偏移量为 0x7bf、十进制为 1983 的位置处。所以，在 q[199] 之后，还有 4096-1984=2112 个字节空间在同一个页中可供使用。这写空间如果用于存储 int 型变量，可以存储 2112/4=528 个，也就是说，可以储存从 q[200] 到 q[727] 这 528 个元素。当然，实际上它们并不是 q 数组的元素，不过正如前面讲的，编译器并不会对此提出异议。同样的，硬件也不会，由于我们拥有对页的写权限，因此我们可以把相应的数据写入这些空间中。不过当 i 自增到 728 时，会使程序使用一个新页，该页不会付予我们写权限（或者其他任何权限），这会被硬件发现，并且硬件会触发段错误。

不仅试图读写超过范围的数据项会导致段错误，同样，试图执行超过范围的指令也会导致段错误。举例说明，考虑下面的代码：

```
int f(int x)
{
  return x*x;
}

int (*p)(int);

main()
{
  p = f;
  u = (*p)(5);
  printf("%d\n",u);
}
```

如果我们忘了写如下一行：

```
  u=(*p)(5);
```

那么变量 p 不会指向任何函数，也就是说，我们会企图执行超出我们程序范围的空间中的代码，这就会导致段错误。

5.6 改进优化

当对页表的查询过频时，虚拟内存就会面临巨大的消耗。由于这个原因，典型情况下硬件会包括一个翻译查询缓冲（TLB）。这是一个特殊的缓存，用来报错 CPU 的页表的部分拷贝，这样，就减少了对内存中页表信息的读取次数。

[33]或者通过添加一个 printf() 函数调用来输出相应的信息。需要注意的是无论是在 gdb 下运行还是加入一个 printf() 函数都会改变程序装入的位置，从而得到不同的结果。

5.7 在 VM 机制下缓存扮演什么角色？

直到目前为止，我们都没有涉及缓存[34]。但是事实上很多使用VM机制的设备都拥有缓存，那么在这种情况下，缓存扮演的是什么角色呢？

核心是：速度依然是个问题。CPU 会首先在缓存中寻找它需要的，因为缓存一般都嵌在 CPU 内部、或至少离它最近。如果查询缓存后没有获得 CPU 想要的， CPU 才会查询内存。如果这些数据在内存中，那么整个数据块就会被拷贝至缓存中。如果数据还没载入内存，也就是说发生了一个页错误，那么我们应该从硬盘中相应位置读入正确的页，在一个缓存缺失发生之前。

另外一个问题是，缓存中使用的地址是虚拟地址还是物理地址呢？假设现在要执行的指令来自虚拟地址 200 的位置，假如缓存使用的虚拟地址机制，那么 CPU 会使用 200 作为索引进行缓存查询；如果使用物理地址机制，那么 CPU 首先会将 200 转换为物理地址[35]，然后再基于这个物理地址进行缓存查询。

需要注意的是，缓存设计整个儿就是一硬件问题。缓存的查询和页表的替换都是由电路中的某些硬件走线实现的。相反的，VM 机制却是硬件和软件的综合机制。硬件完成页表查询、检查页的装载情况和权限问题；不过是由软件 — OS — 来创建和维护页表的，更多的，当发生一个页错误时，完成页替换和更新页表的也是操作系统。

所以，你可能在一台电脑上拥有两个不同版本的 UNIX[36]，使用同样的编译器等，不过它们的页错误发生的频率仍然可能不一样。也许其中一个操作系统对这个程序的页替换机制要比另一个版本的系统优秀。

注意到 OS 能够告诉你你运行的程序发生了多少次页错误（看下面的 `time` 命令）；每次页错误都会导致到 OS 的跳转，也就是说导致 OS 的运行，所以，OS 可以追踪你的程序遭遇了多少此页错误。相反的，OS 不能追踪你程序的缓存缺失有多少次，因为 OS 不能处理它，它完全是由硬件来处理的。

5.8 巩固上述概念：自己尝试这些命令

UNIX 系统命令 `time` 可以告诉你你的程序执行了多少轮、该程序发生了多少页错误等信息。在命令行中输入该命令，后面跟上相应的程序作为参数，假设，你有个名为 `x` 的程序，并且该程序以 12 为参数，那么使用命令：

```
%time x 12
```

来替换命令：

```
%x 12
```

同样的，`top` 命令也可以提供给你很多有用的信息。

[34]不包括 TLB，它实在是太专门了。
[35]通过检查 TLB，然后如果有必要的话再查询页表。
[36]或者 Linux 和 Windows.

6 关于系统调用

回忆前面讲的，OS 给应用程序提供 I/O 服务等[37]。举个例子，当你调用 `printf()` 时，该函数只是在 C 的函数库中，而没有在 OS 的函数库中，不过，接下来它会调用在 OS 库中的 `write()` 函数。这个对 `write()` 函数的调用（你也可以直接把这个函数写在源程序中）就称为**系统调用**。

再回忆，出于安全考虑，我们只想给 OS 授权以通过 Intel 的指令如 in、out 等进行实际 I/O 操作的权利，其他程序都是通过 OS 来访问 I/O 的。因此，硬件的设计会使得这些指令只能以内核模式执行。

由于这个原因，一般不能直接通过普通的子程序 CALL 指令来完成系统调用，因为我们需要一种机制来使设备进入内核模式。（很明显的，不可能用一条指令就将系统设为内核模式了，因为如果这样的话，那么任何一个普通的用户程序都能执行这条指令，然后进入内核模式大加破坏！）另外一个问题是链接器不知道所需的子程序处于 OS 的哪个位置的。

取而代之的是，系统调用是通过一种被称为**软中断**类型的指令来完成的。在 Intel 设备中，这是由 int 指令来实现的，该指令只有一个操作数。

下面的例子都是在 Linux 系统下完成的，并且在例子中，int 的操作数为 `0x80`。换句话说，在你的 C 程序中对 `write()` 的调用(或是对 `printf()` 的调用)将会翻译成如下代码：

```
... # code to put parameters values into designated registers
int 0x80
```

`int` 指令如同硬件中断一样工作，这说明了该指令会产生一个到 OS 的强行跳转，然后将特权等级改变至内核模式，使 OS 能执行它所需要的特权指令。你需要记住的是，这里的“中断”是由“被中断”的程序特意发出的，通过指令 `int`。这和完全与被中断程序无关的硬件中断有很大的不同。

上面的操作数，也就是 `0x80`，是硬件中断设备号的模拟量，CPU 会跳转至由 `c(IDT)+8*0x80`[38] 所指示向量位置。

当中断处理完毕后，OS 会执行一条 iret 指令返回到被中断的应用程序，并且，将把内核模式变回至用户模式。

如上面所讲的，系统调用一般都有参数，就象普通的子程序调用一样。有一个参数几乎所有服务都通用 — 服务类型号，该类型号通过寄存器 EAX 传递给 OS。也可能使用其他寄存器，取决于不同的服务了。

作为一个例子，下面的在 Linux 系统中写的 Intel 汇编源程序会向屏幕输出“ABC/n”，然后退出：

```
.data
hi: .string "ABC\n"
.text
_start:
# write "ABC\n" to the screen
```

[37]现在还不能得到结论说所有的服务都是 I/O 服务。比如一个程序调用 `execve()` 服务来执行另外一个程序，还比如 `getpid()` 服务返回调用者的进程编号，这些都与 I/O 无关。

[38]用户可以通过改变这块内存区域的内容对系统造成损害，因此 OS 将会设置其页表来指定相应的限制。

```
movl $4, %eax # the write() system call, number 4 obtained
# from /usr/include/asm/unistd.h
movl $1, %ebx # 1 = file handle for stdout
movl $hi, %ecx # write from where
movl $4, %edx # write how many bytes
int $0x80 # system call # call exit()
movl $1, %eax # exit() is system call number 1
int $0x80 # system call
```

对于这个特定的OS服务，`write()`函数，参数通过寄存器EBX，ECX，EDX传递（并且，正如前面提到的，EAX指明了我们想要获得什么服务）。

下面是一些其他系统服务的服务类型号（要完成相应的服务，就在`int $0x80`执行之前将类型号写入寄存器EAX中）：

```
read 3
file open 5
execve 11
chdir 12
kill 37
```

7 OS 文件管理

OS会维护一个显示所有文件在硬盘中的起始扇区信息的表。（这个表本身也在硬盘中。）关于该表只需要存储给定文件的起始扇区信息的原因是，该文件的不同扇区能够通过“linked-list”的方式链接起来，换句话说，在一个文件起始扇区的最末部分，OS会存储关于开始跟踪和下一部分文件扇区的信息。

OS也会维护一张关于未用扇区的表。当用户新创建一个文件时，OS会核查这张表以找到一个位置安放该文件，当然，接下来OS会更新该表。

如果用户删除一个文件，OS会同时更新上面两张表，首先删除在第一张表中该文件的入口信息，然后将这个文件占用的空间信息写入第二张表中[39]。

文件的创建和删除动作贯穿于整个设备使用过程中，对未用扇区的设置就像是干一件东拼西凑的活路一样，因为这些未用的扇区任意散落在硬盘中。这对性能有些负面影响，特别是在查询时。因此很多OS中都有类似的碎片整理工具，这些工具用来重新排列硬盘中文件的位置，使得硬盘上每个独立文件的扇区能够彼此靠近。

[39]无论如何，文件的内容依然保存在那里，因此所谓的“undelete”程序才能通过恢复那些释放了的扇区来恢复被用户（不小心）删除的文件。