第 5 章　插叙：进程 API

补充：插叙

本章将介绍更多系统实践方面的内容，包括特别关注操作系统的 API 及其使用方式。如果不关心实践相关的内容，你可以略过。但是你应该喜欢实践内容，它们通常在实际生活中有用。例如，公司通常不会因为不实用的技能而聘用你。

本章将讨论 UNIX 系统中的进程创建。UNIX 系统采用了一种非常有趣的创建新进程的方式，即通过一对系统调用：fork()和 exec()。进程还可以通过第三个系统调用 wait()，来等待其创建的子进程执行完成。本章将详细介绍这些接口，通过一些简单的例子来激发兴趣。

关键问题：如何创建并控制进程

操作系统应该提供怎样的进程来创建及控制接口？如何设计这些接口才能既方便又实用？

5.1　fork()系统调用

系统调用 fork()用于创建新进程[C63]。但要小心，这可能是你使用过的最奇怪的接口[①]。具体来说，你可以运行一个程序，代码如图 5.1 所示。仔细看这段代码，建议亲自键入并运行！

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <unistd.h>
4
5    int
6    main(int argc, char *argv[])
7    {
8        printf("hello world (pid:%d)\n", (int) getpid());
9        int rc = fork();
10       if (rc < 0) {         // fork failed; exit
11           fprintf(stderr, "fork failed\n");
12           exit(1);
13       } else if (rc == 0) { // child (new process)
14           printf("hello, I am child (pid:%d)\n", (int) getpid());
15       } else {              // parent goes down this path (main)
16           printf("hello, I am parent of %d (pid:%d)\n",
17                   rc, (int) getpid());
```

① 好吧，我们承认我们并不确定。谁知道你在没人的时候调用过什么？但 fork()相当奇怪，不管你的函数调用模式有多不同。

```
18        }
19        return 0;
20    }
```

图 5.1 调用 fork()（p1.c）

运行这段程序（p1.c），将看到如下输出：

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

让我们更详细地理解一下 p1.c 到底发生了什么。当它刚开始运行时，进程输出一条 hello world 信息，以及自己的进程描述符（process identifier，PID）。该进程的 PID 是 29146。在 UNIX 系统中，如果要操作某个进程（如终止进程），就要通过 PID 来指明。到目前为止，一切正常。

紧接着有趣的事情发生了。进程调用了 fork()系统调用，这是操作系统提供的创建新进程的方法。新创建的进程几乎与调用进程完全一样，对操作系统来说，这时看起来有两个完全一样的 p1 程序在运行，并都从 fork()系统调用中返回。新创建的进程称为子进程（child），原来的进程称为父进程（parent）。子进程不会从 main()函数开始执行（因此 hello world 信息只输出了一次），而是直接从 fork()系统调用返回，就好像是它自己调用了 fork()。

你可能已经注意到，子进程并不是完全拷贝了父进程。具体来说，虽然它拥有自己的地址空间（即拥有自己的私有内存）、寄存器、程序计数器等，但是它从 fork()返回的值是不同的。父进程获得的返回值是新创建子进程的 PID，而子进程获得的返回值是 0。这个差别非常重要，因为这样就很容易编写代码处理两种不同的情况（像上面那样）。

你可能还会注意到，它的输出不是确定的（deterministic）。子进程被创建后，我们就需要关心系统中的两个活动进程了：子进程和父进程。假设我们在单个 CPU 的系统上运行（简单起见），那么子进程或父进程在此时都有可能运行。在上面的例子中，父进程先运行并输出信息。在其他情况下，子进程可能先运行，会有下面的输出结果：

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

CPU 调度程序（scheduler）决定了某个时刻哪个进程被执行，我们稍后将详细介绍这部分内容。由于 CPU 调度程序非常复杂，所以我们不能假设哪个进程会先运行。事实表明，这种不确定性（non-determinism）会导致一些很有趣的问题，特别是在多线程程序（multi-threaded program）中。在本书第 2 部分中学习并发（concurrency）时，我们会看到许多不确定性。

5.2 wait()系统调用

到目前为止，我们没有做太多事情：只是创建了一个子进程，打印了一些信息并退出。事实表明，有时候父进程需要等待子进程执行完毕，这很有用。这项任务由 wait()系统调用

（或者更完整的兄弟接口 waitpid()）。图 5.2 展示了更多细节。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6    int
7    main(int argc, char *argv[])
8    {
9        printf("hello world (pid:%d)\n", (int) getpid());
10       int rc = fork();
11       if (rc < 0) {         // fork failed; exit
12           fprintf(stderr, "fork failed\n");
13           exit(1);
14       } else if (rc == 0) { // child (new process)
15           printf("hello, I am child (pid:%d)\n", (int) getpid());
16       } else {     // parent goes down this path (main)
17           int wc = wait(NULL);
18           printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19                   rc, wc, (int) getpid());
20       }
21       return 0;
22   }
```

图 5.2 调用 fork()和 wait()（p2.c）

在 p2.c 的例子中，父进程调用 wait()，延迟自己的执行，直到子进程执行完毕。当子进程结束时，wait()才返回父进程。

上面的代码增加了 wait()调用，因此输出结果也变得确定了。这是为什么呢？想想看。（等你想想看……好了）

下面是输出结果：

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

通过这段代码，现在我们知道子进程总是先输出结果。为什么知道？好吧，它可能只是碰巧先运行，像以前一样，因此先于父进程输出结果。但是，如果父进程碰巧先运行，它会马上调用 wait()。该系统调用会在子进程运行结束后才返回[①]。因此，即使父进程先运行，它也会礼貌地等待子进程运行完毕，然后 wait()返回，接着父进程才输出自己的信息。

5.3 最后是 exec()系统调用

最后是 exec()系统调用，它也是创建进程 API 的一个重要部分[②]。这个系统调用可以让

① 有些情况下，wait()在子进程退出之前返回。像往常一样，请阅读 man 手册获取更多细节。小心本书中绝对的、无条件的陈述，比如“子进程总是先输出结果”或“UNIX 是世界上最好的东西，甚至比冰淇淋还要好”。

② 实际上，exec()有几种变体：execl()、execle()、execlp()、execv()和 execvp()。请阅读 man 手册以了解更多信息。

子进程执行与父进程不同的程序。例如，在 p2.c 中调用 fork()，这只是在你想运行相同程序的拷贝时有用。但时，我们常常想运行不同的程序，exec()正好做这样的事（见图 5.3）。

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107   1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <unistd.h>
4    #include <string.h>
5    #include <sys/wait.h>
6
7    int
8    main(int argc, char *argv[])
9    {
10       printf("hello world (pid:%d)\n", (int) getpid());
11       int rc = fork();
12       if (rc < 0) {         // fork failed; exit
13           fprintf(stderr, "fork failed\n");
14           exit(1);
15       } else if (rc == 0) { // child (new process)
16           printf("hello, I am child (pid:%d)\n", (int) getpid());
17           char *myargs[3];
18           myargs[0] = strdup("wc");   // program: "wc" (word count)
19           myargs[1] = strdup("p3.c"); // argument: file to count
20           myargs[2] = NULL;           // marks end of array
21           execvp(myargs[0], myargs);  // runs word count
22           printf("this shouldn't print out");
23       } else {    // parent goes down this path (main)
24           int wc = wait(NULL);
25           printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26                   rc, wc, (int) getpid());
27       }
28       return 0;
29   }
```

图 5.3 调用 fork()、wait()和 exec()（p3.c）

在这个例子中，子进程调用 execvp()来运行字符计数程序 wc。实际上，它针对源代码文件 p3.c 运行 wc，从而告诉我们该文件有多少行、多少单词，以及多少字节。

fork()系统调用很奇怪，它的伙伴 exec()也不一般。给定可执行程序的名称（如 wc）及需要的参数（如 p3.c）后，exec()会从可执行程序中加载代码和静态数据，并用它覆写自己的代码段（以及静态数据），堆、栈及其他内存空间也会被重新初始化。然后操作系统就执行该程序，将参数通过 argv 传递给该进程。因此，它并没有创建新进程，而是直接将当前运行的程序（以前的 p3）替换为不同的运行程序（wc）。子进程执行 exec()之后，几乎就像 p3.c 从未运行过一样。对 exec()的成功调用永远不会返回。

5.4 为什么这样设计 API

当然，你的心中可能有一个大大的问号：为什么设计如此奇怪的接口，来完成简单的、创建新进程的任务？好吧，事实证明，这种分离 fork()及 exec()的做法在构建 UNIX shell 的时候非常有用，因为这给了 shell 在 fork 之后 exec 之前运行代码的机会，这些代码可以在运行新程序前改变环境，从而让一系列有趣的功能很容易实现。

提示：重要的是做对事（LAMPSON 定律）

Lampson 在他的著名论文《Hints for Computer Systems Design》[L83]中曾经说过：“做对事（Get it right）。抽象和简化都不能替代做对事。”有时你必须做正确的事，当你这样做时，总是好过其他方案。有许多方式来设计创建进程的 API，但 fork()和 exec()的组合既简单又极其强大。因此 UNIX 的设计师们做对了。因为 Lampson 经常“做对事”，所以我们就以他来命名这条定律。

shell 也是一个用户程序[①]，它首先显示一个提示符（prompt），然后等待用户输入。你可以向它输入一个命令（一个可执行程序的名称及需要的参数），大多数情况下，shell 可以在文件系统中找到这个可执行程序，调用 fork()创建新进程，并调用 exec()的某个变体来执行这个可执行程序，调用 wait()等待该命令完成。子进程执行结束后，shell 从 wait()返回并再次输出一个提示符，等待用户输入下一条命令。

fork()和 exec()的分离，让 shell 可以方便地实现很多有用的功能。比如：

```
prompt> wc p3.c > newfile.txt
```

在上面的例子中，wc 的输出结果被重定向（redirect）到文件 newfile.txt 中（通过 newfile.txt 之前的大于号来指明重定向）。shell 实现结果重定向的方式也很简单，当完成子进程的创建后，shell 在调用 exec()之前先关闭了标准输出（standard output），打开了文件 newfile.txt。这样，即将运行的程序 wc 的输出结果就被发送到该文件，而不是打印在屏幕上。

图 5.4 展示了这样做的一个程序。重定向的工作原理，是基于对操作系统管理文件描述符方式的假设。具体来说，UNIX 系统从 0 开始寻找可以使用的文件描述符。在这个例子中，STDOUT_FILENO 将成为第一个可用的文件描述符，因此在 open()被调用时，得到赋值。然后子进程向标准输出文件描述符的写入（例如通过 printf()这样的函数），都会被透明地转向新打开的文件，而不是屏幕。

下面是运行 p4.c 的结果：

```
prompt> ./p4
prompt> cat p4.output
      32     109     846 p4.c
prompt>
1    #include <stdio.h>
2    #include <stdlib.h>
```

① 有许多 shell，如 tcsh、bash 和 zsh 等。你应该选择一个，阅读它的 man 手册，了解更多信息。所有 UNIX 专家都这样做。

```
3   #include <unistd.h>
4   #include <string.h>
5   #include <fcntl.h>
6   #include <sys/wait.h>
7
8   int
9   main(int argc, char *argv[])
10  {
11      int rc = fork();
12      if (rc < 0) {     // fork failed; exit
13          fprintf(stderr, "fork failed\n");
14          exit(1);
15      } else if (rc == 0) { // child: redirect standard output to a file
16          close(STDOUT_FILENO);
17          open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18
19          // now exec "wc"...
20          char *myargs[3];
21          myargs[0] = strdup("wc");    // program: "wc" (word count)
22          myargs[1] = strdup("p4.c");  // argument: file to count
23          myargs[2] = NULL;            // marks end of array
24          execvp(myargs[0], myargs);   // runs word count
25      } else {                         // parent goes down this path (main)
26          int wc = wait(NULL);
27      }
28      return 0;
29  }
```

图 5.4 之前所有的工作加上重定向（p4.c）

关于这个输出，你（至少）会注意到两个有趣的地方。首先，当运行 p4 程序后，好像什么也没有发生。shell 只是打印了命令提示符，等待用户的下一个命令。但事实并非如此，p4 确实调用了 fork 来创建新的子进程，之后调用 execvp()来执行 wc。屏幕上没有看到输出，是由于结果被重定向到文件 p4.output。其次，当用 cat 命令打印输出文件时，能看到运行 wc 的所有预期输出。很酷吧？

UNIX 管道也是用类似的方式实现的，但用的是 pipe()系统调用。在这种情况下，一个进程的输出被链接到了一个内核管道（pipe）上（队列），另一个进程的输入也被连接到了同一个管道上。因此，前一个进程的输出无缝地作为后一个进程的输入，许多命令可以用这种方式串联在一起，共同完成某项任务。比如通过将 grep、wc 命令用管道连接可以完成从一个文件中查找某个词，并统计其出现次数的功能：grep -o foo file | wc -l。

最后，我们刚才只是从较高的层面上简单介绍了进程 API，关于这些系统调用的细节，还有更多需要学习和理解。例如，在本书第 3 部分介绍文件系统时，我们会学习更多关于文件描述符的知识。现在，知道 fork()和 exec()组合在创建和操作进程时非常强大就足够了。

补充：RTFM——阅读 man 手册

很多时候，本书提到某个系统调用或库函数时，会建议阅读 man 手册。man 手册是 UNIX 系统中最原生的文档，要知道它的出现甚至早于网络（Web）。

花时间阅读 man 手册是系统程序员成长的必经之路。手册里有许多有用的隐藏彩蛋。尤其是你正在使用的 shell（如 tcsh 或 bash），以及程序中需要使用的系统调用（以便了解返回值和异常情况）。

最后，阅读 man 手册可以避免尴尬。当你询问同事某个 fork 细节时，他可能会回复："RTFM"。这是他在有礼貌地督促你阅读 man 手册（Read the Man）。RTFM 中的 F 只是为这个短语增加了一点色彩……

5.5 其他 API

除了上面提到的 fork()、exec()和 wait()之外，在 UNIX 中还有其他许多与进程交互的方式。比如可以通过 kill()系统调用向进程发送信号（signal），包括要求进程睡眠、终止或其他有用的指令。实际上，整个信号子系统提供了一套丰富的向进程传递外部事件的途径，包括接受和执行这些信号。

此外还有许多非常有用的命令行工具。比如通过 ps 命令来查看当前在运行的进程，阅读 man 手册来了解 ps 命令所接受的参数。工具 top 也很有用，它展示当前系统中进程消耗 CPU 或其他资源的情况。有趣的是，你常常会发现 top 命令自己就是最占用资源的，它或许有一点自大狂。此外还有许多 CPU 检测工具，让你方便快速地了解系统负载。比如，我们总是让 MenuMeters（来自 Raging Menace 公司）运行在 Mac 计算机的工具栏上，这样就能随时了解当前的 CPU 利用率。一般来说，对现状了解得越多越好。

5.6 小结

本章介绍了在 UNIX 系统中创建进程需要的 API：fork()、exec()和 wait()。更多的细节可以阅读 Stevens 和 Rago 的著作 [SR05]，尤其是关于进程控制、进程关系及信号的章节。其中的智慧让人受益良多。

参考资料

[C63]"A Multiprocessor System Design" Melvin E. Conway
AFIPS '63 Fall Joint Computer Conference
New York, USA 1963
早期关于如何设计多处理系统的论文。文中可能首次在讨论创建新进程时使用 fork()术语。

[DV66]"Programming Semantics for Multiprogrammed Computations" Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966

一篇讲述多道程序计算机系统基础知识的经典文章。毫无疑问，它对 Project MAC、Multics 以及最终的 UNIX 都有很大的影响。

[L83]“Hints for Computer Systems Design” Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
Lampson 关于计算机系统如何设计的著名建议。你应该抽时间读一读它。

[SR05]“Advanced Programming in the UNIX Environment”
W. Richard Stevens and Stephen A. Rago Addison-Wesley, 2005
在这里可以找到使用 UNIX API 的所有细节和妙处。买下这本书！阅读它，最重要的是靠它谋生。

补充：编码作业

编码作业是小型练习。你可以编写代码在真正的机器上运行，从而获得一些现代操作系统必须提供的基本 API 的体验。毕竟，你（可能）是一名计算机科学家，因此应该喜欢编码，对吧？当然，要真正成为专家，你必须花更多的时间来破解机器。实际上，要找一切借口来写一些代码，看看它是如何工作的。花时间，成为智者，你可以做到的。

作业（编码）

在这个作业中，你要熟悉一下刚读过的进程管理 API。别担心，它比听起来更有趣！如果你找到尽可能多的时间来编写代码，通常会增加成功的概率[①]，为什么不现在就开始呢？

问题

1．编写一个调用 fork()的程序。在调用 fork()之前，让主进程访问一个变量（例如 *x*）并将其值设置为某个值（例如 100）。子进程中的变量有什么值？当子进程和父进程都改变 *x* 的值时，变量会发生什么？

2．编写一个打开文件的程序（使用 open()系统调用），然后调用 fork()创建一个新进程。子进程和父进程都可以访问 open()返回的文件描述符吗？当它们并发（即同时）写入文件时，会发生什么？

3．使用 fork()编写另一个程序。子进程应打印“hello”，父进程应打印“goodbye”。你应该尝试确保子进程始终先打印。你能否不在父进程调用 wait()而做到这一点呢？

4．编写一个调用 fork()的程序，然后调用某种形式的 exec()来运行程序/bin/ls。看看是否可以尝试 exec()的所有变体，包括 execl()、execle()、execlp()、execv()、execvp()和 execvP()。

① 如果你不喜欢编码，但想成为计算机科学家，这意味着你需要变得非常擅长计算机科学理论，或者也许要重新考虑你一直在说的“计算机科学”这回事。

为什么同样的基本调用会有这么多变种？

5．现在编写一个程序，在父进程中使用 wait()，等待子进程完成。wait()返回什么？如果你在子进程中使用 wait()会发生什么？

6．对前一个程序稍作修改，这次使用 waitpid()而不是 wait()。什么时候 waitpid()会有用？

7．编写一个创建子进程的程序，然后在子进程中关闭标准输出（STDOUT_FILENO）。如果子进程在关闭描述符后调用 printf()打印输出，会发生什么？

8．编写一个程序，创建两个子进程，并使用 pipe()系统调用，将一个子进程的标准输出连接到另一个子进程的标准输入。

第 6 章　机制：受限直接执行

为了虚拟化 CPU，操作系统需要以某种方式让许多任务共享物理 CPU，让它们看起来像是同时运行。基本思想很简单：运行一个进程一段时间，然后运行另一个进程，如此轮换。通过以这种方式时分共享（time sharing）CPU，就实现了虚拟化。

然而，在构建这样的虚拟化机制时存在一些挑战。第一个是性能：如何在不增加系统开销的情况下实现虚拟化？第二个是控制权：如何有效地运行进程，同时保留对 CPU 的控制？控制权对于操作系统尤为重要，因为操作系统负责资源管理。如果没有控制权，一个进程可以简单地无限制运行并接管机器，或访问没有权限的信息。因此，在保持控制权的同时获得高性能，这是构建操作系统的主要挑战之一。

关键问题：如何高效、可控地虚拟化 CPU

操作系统必须以高性能的方式虚拟化 CPU，同时保持对系统的控制。为此，需要硬件和操作系统支持。操作系统通常会明智地利用硬件支持，以便高效地实现其工作。

6.1　基本技巧：受限直接执行

为了使程序尽可能快地运行，操作系统开发人员想出了一种技术——我们称之为受限的直接执行（limited direct execution）。这个概念的“直接执行”部分很简单：只需直接在 CPU 上运行程序即可。因此，当 OS 希望启动程序运行时，它会在进程列表中为其创建一个进程条目，为其分配一些内存，将程序代码（从磁盘）加载到内存中，找到入口点（main()函数或类似的），跳转到那里，并开始运行用户的代码。表 6.1 展示了这种基本的直接执行协议（没有任何限制），使用正常的调用并返回跳转到程序的 main()，并在稍后回到内核。

表 6.1　直接运行协议（无限制）

操作系统	程序
在进程列表上创建条目 为程序分配内存 将程序加载到内存中 根据 argc/argv 设置程序栈	
清除寄存器 执行 call main() 方法	
	执行 main() 从 main 中执行 return
释放进程的内存将进程 从进程列表中清除	

听起来很简单，不是吗？但是，这种方法在我们的虚拟化 CPU 时产生了一些问题。第一个问题很简单：如果我们只运行一个程序，操作系统怎么能确保程序不做任何我们不希望它做的事，同时仍然高效地运行它？第二个问题：当我们运行一个进程时，操作系统如何让它停下来并切换到另一个进程，从而实现虚拟化 CPU 所需的时分共享？

下面在回答这些问题时，我们将更好地了解虚拟化 CPU 需要什么。在开发这些技术时，我们还会看到标题中的“受限”部分来自哪里。如果对运行程序没有限制，操作系统将无法控制任何事情，因此会成为“仅仅是一个库”——对于有抱负的操作系统而言，这真是非常令人悲伤的事！

6.2 问题 1：受限制的操作

直接执行的明显优势是快速。该程序直接在硬件 CPU 上运行，因此执行速度与预期的一样快。但是，在 CPU 上运行会带来一个问题——如果进程希望执行某种受限操作（如向磁盘发出 I/O 请求或获得更多系统资源（如 CPU 或内存）），该怎么办？

关键问题：如何执行受限制的操作

一个进程必须能够执行 I/O 和其他一些受限制的操作，但又不能让进程完全控制系统。操作系统和硬件如何协作实现这一点？

提示：采用受保护的控制权转移

硬件通过提供不同的执行模式来协助操作系统。在用户模式（user mode）下，应用程序不能完全访问硬件资源。在内核模式（kernel mode）下，操作系统可以访问机器的全部资源。还提供了陷入（trap）内核和从陷阱返回（return-from-trap）到用户模式程序的特别说明，以及一些指令，让操作系统告诉硬件陷阱表（trap table）在内存中的位置。

对于 I/O 和其他相关操作，一种方法就是让所有进程做所有它想做的事情。但是，这样做导致无法构建许多我们想要的系统。例如，如果我们希望构建一个在授予文件访问权限前检查权限的文件系统，就不能简单地让任何用户进程向磁盘发出 I/O。如果这样做，一个进程就可以读取或写入整个磁盘，这样所有的保护都会失效。

因此，我们采用的方法是引入一种新的处理器模式，称为用户模式（user mode）。在用户模式下运行的代码会受到限制。例如，在用户模式下运行时，进程不能发出 I/O 请求。这样做会导致处理器引发异常，操作系统可能会终止进程。

与用户模式不同的内核模式（kernel mode），操作系统（或内核）就以这种模式运行。在此模式下，运行的代码可以做它喜欢的事，包括特权操作，如发出 I/O 请求和执行所有类型的受限指令。

但是，我们仍然面临着一个挑战——如果用户希望执行某种特权操作（如从磁盘读取），应该怎么做？为了实现这一点，几乎所有的现代硬件都提供了用户程序执行系统调用的能力。系统调用是在 Atlas [K+61，L78]等古老机器上开创的，它允许内核小心地向用户程序暴露某些关键功能，例如访问文件系统、创建和销毁进程、与其他进程通信，以及分配更

多内存。大多数操作系统提供几百个调用（详见 POSIX 标准[P10]）。早期的 UNIX 系统公开了更简洁的子集，大约 20 个调用。

要执行系统调用，程序必须执行特殊的陷阱（trap）指令。该指令同时跳入内核并将特权级别提升到内核模式。一旦进入内核，系统就可以执行任何需要的特权操作（如果允许），从而为调用进程执行所需的工作。完成后，操作系统调用一个特殊的从陷阱返回（return-from-trap）指令，如你期望的那样，该指令返回到发起调用的用户程序中，同时将特权级别降低，回到用户模式。

执行陷阱时，硬件需要小心，因为它必须确保存储足够的调用者寄存器，以便在操作系统发出从陷阱返回指令时能够正确返回。例如，在 x86 上，处理器会将程序计数器、标志和其他一些寄存器推送到每个进程的内核栈（kernel stack）上。从返回陷阱将从栈弹出这些值，并恢复执行用户模式程序（有关详细信息，请参阅英特尔系统手册[I11]）。其他硬件系统使用不同的约定，但基本概念在各个平台上是相似的。

补充：为什么系统调用看起来像过程调用

你可能想知道，为什么对系统调用的调用（如 open()或 read()）看起来完全就像 C 中的典型过程调用。也就是说，如果它看起来像一个过程调用，系统如何知道这是一个系统调用，并做所有正确的事情？原因很简单：它是一个过程调用，但隐藏在过程调用内部的是著名的陷阱指令。更具体地说，当你调用 open()（举个例子）时，你正在执行对 C 库的过程调用。其中，无论是对于 open()还是提供的其他系统调用，库都使用与内核一致的调用约定来将参数放在众所周知的位置（例如，在栈中或特定的寄存器中），将系统调用号也放入一个众所周知的位置（同样，放在栈或寄存器中），然后执行上述的陷阱指令。库中陷阱之后的代码准备好返回值，并将控制权返回给发出系统调用的程序。因此，C 库中进行系统调用的部分是用汇编手工编码的，因为它们需要仔细遵循约定，以便正确处理参数和返回值，以及执行硬件特定的陷阱指令。现在你知道为什么你自己不必写汇编代码来陷入操作系统了，因为有人已经为你写了这些汇编。

还有一个重要的细节没讨论：陷阱如何知道在 OS 内运行哪些代码？显然，发起调用的过程不能指定要跳转到的地址（就像你在进行过程调用时一样），这样做让程序可以跳转到内核中的任意位置，这显然是一个糟糕的主意（想象一下跳到访问文件的代码，但在权限检查之后。实际上，这种能力很可能让一个狡猾的程序员令内核运行任意代码序列[S07]）。因此内核必须谨慎地控制在陷阱上执行的代码。

内核通过在启动时设置陷阱表（trap table）来实现。当机器启动时，它在特权（内核）模式下执行，因此可以根据需要自由配置机器硬件。操作系统做的第一件事，就是告诉硬件在发生某些异常事件时要运行哪些代码。例如，当发生硬盘中断，发生键盘中断或程序进行系统调用时，应该运行哪些代码？操作系统通常通过某种特殊的指令，通知硬件这些陷阱处理程序的位置。一旦硬件被通知，它就会记住这些处理程序的位置，直到下一次重新启动机器，并且硬件知道在发生系统调用和其他异常事件时要做什么（即跳转到哪段代码）。

最后再插一句：能够执行指令来告诉硬件陷阱表的位置是一个非常强大的功能。因此，你可能已经猜到，这也是一项特权（privileged）操作。如果你试图在用户模式下执行这个指令，硬件不会允许，你可能会猜到会发生什么（提示：再见，违规程序）。思考问题：如果可以设置自己的陷阱表，你可以对系统做些什么？你能接管机器吗？

时间线（随着时间的推移向下，在表 6.2 中）总结了该协议。我们假设每个进程都有一个内

核栈，在进入内核和离开内核时，寄存器（包括通用寄存器和程序计数器）分别被保存和恢复。

表 6.2 受限直接运行协议

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住系统调用处理程序的地址	
操作系统@运行（内核模式）	硬件	程序（应用模式）
在进程列表上创建条目 为程序分配内存 将程序加载到内存中 根据 argv 设置程序栈 用寄存器/程序计数器填充内核栈 从陷阱返回		
	从内核栈恢复寄存器 转向用户模式 跳到 main	
		运行 main …… 调用系统调用 陷入操作系统
	将寄存器保存到内核栈 转向内核模式 跳到陷阱处理程序	
处理陷阱 做系统调用的工作 从陷阱返回		
	从内核栈恢复寄存器 转向用户模式 跳到陷阱之后的程序计数器	
		……从 main 返回 陷入（通过 exit()）
释放进程的内存将进程 从进程列表中清除		

LDE 协议有两个阶段。第一个阶段（在系统引导时），内核初始化陷阱表，并且 CPU 记住它的位置以供随后使用。内核通过特权指令来执行此操作（所有特权指令均以粗体突出显示）。第二个阶段（运行进程时），在使用从陷阱返回指令开始执行进程之前，内核设置了一些内容（例如，在进程列表中分配一个节点，分配内存）。这会将 CPU 切换到用户模式并开始运行该进程。当进程希望发出系统调用时，它会重新陷入操作系统，然后再次通过从陷阱返回，将控制权还给进程。该进程然后完成它的工作，并从 main()返回。这通常会返回到一些存根代码，它将正确退出该程序（例如，通过调用 exit()系统调用，这将陷入 OS 中）。此时，OS 清理干净，任务完成了。

6.3 问题 2：在进程之间切换

直接执行的下一个问题是实现进程之间的切换。在进程之间切换应该很简单，对吧？

操作系统应该决定停止一个进程并开始另一个进程。有什么大不了的？但实际上这有点棘手，特别是，如果一个进程在 CPU 上运行，这就意味着操作系统没有运行。如果操作系统没有运行，它怎么能做事情？（提示：它不能）虽然这听起来几乎是哲学，但这是真正的问题——如果操作系统没有在 CPU 上运行，那么操作系统显然没有办法采取行动。因此，我们遇到了关键问题。

关键问题：如何重获 CPU 的控制权

操作系统如何重新获得 CPU 的控制权（regain control），以便它可以在进程之间切换？

协作方式：等待系统调用

过去某些系统采用的一种方式（例如，早期版本的 Macintosh 操作系统[M11]或旧的 Xerox Alto 系统[A79]）称为协作（cooperative）方式。在这种风格下，操作系统相信系统的进程会合理运行。运行时间过长的进程被假定会定期放弃 CPU，以便操作系统可以决定运行其他任务。

因此，你可能会问，在这个虚拟的世界中，一个友好的进程如何放弃 CPU？事实证明，大多数进程通过进行系统调用，将 CPU 的控制权转移给操作系统，例如打开文件并随后读取文件，或者向另一台机器发送消息或创建新进程。像这样的系统通常包括一个显式的 yield 系统调用，它什么都不干，只是将控制权交给操作系统，以便系统可以运行其他进程。

提示：处理应用程序的不当行为

操作系统通常必须处理不当行为，这些程序通过设计（恶意）或不小心（错误），尝试做某些不应该做的事情。在现代系统中，操作系统试图处理这种不当行为的方式是简单地终止犯罪者。一击出局！也许有点残酷，但如果你尝试非法访问内存或执行非法指令，操作系统还应该做些什么？

如果应用程序执行了某些非法操作，也会将控制转移给操作系统。例如，如果应用程序以 0 为除数，或者尝试访问应该无法访问的内存，就会陷入（trap）操作系统。操作系统将再次控制 CPU（并可能终止违规进程）。

因此，在协作调度系统中，OS 通过等待系统调用，或某种非法操作发生，从而重新获得 CPU 的控制权。你也许会想：这种被动方式不是不太理想吗？例如，如果某个进程（无论是恶意的还是充满缺陷的）进入无限循环，并且从不进行系统调用，会发生什么情况？那时操作系统能做什么？

非协作方式：操作系统进行控制

事实证明，没有硬件的额外帮助，如果进程拒绝进行系统调用（也不出错），从而将控制权交还给操作系统，那么操作系统无法做任何事情。事实上，在协作方式中，当进程陷入无限循环时，唯一的办法就是使用古老的解决方案来解决计算机系统中的所有问题——重新启动计算机。因此，我们又遇到了请求获得 CPU 控制权的一个子问题。

关键问题：如何在没有协作的情况下获得控制权

即使进程不协作，操作系统如何获得 CPU 的控制权？操作系统可以做什么来确保流氓进程不会占用机器？

答案很简单，许多年前构建计算机系统的许多人都发现了：时钟中断（timer interrupt）[M+63]。时钟设备可以编程为每隔几毫秒产生一次中断。产生中断时，当前正在运行的进程停止，操作系统中预先配置的中断处理程序（interrupt handler）会运行。此时，操作系统重新获得 CPU 的控制权，因此可以做它想做的事：停止当前进程，并启动另一个进程。

提示：利用时钟中断重新获得控制权

即使进程以非协作的方式运行，添加时钟中断（timer interrupt）也让操作系统能够在 CPU 上重新运行。因此，该硬件功能对于帮助操作系统维持机器的控制权至关重要。

首先，正如我们之前讨论过的系统调用一样，操作系统必须通知硬件哪些代码在发生时钟中断时运行。因此，在启动时，操作系统就是这样做的。其次，在启动过程中，操作系统也必须启动时钟，这当然是一项特权操作。一旦时钟开始运行，操作系统就感到安全了，因为控制权最终会归还给它，因此操作系统可以自由运行用户程序。时钟也可以关闭（也是特权操作），稍后更详细地理解并发时，我们会讨论。

请注意，硬件在发生中断时有一定的责任，尤其是在中断发生时，要为正在运行的程序保存足够的状态，以便随后从陷阱返回指令能够正确恢复正在运行的程序。这一组操作与硬件在显式系统调用陷入内核时的行为非常相似，其中各种寄存器因此被保存（进入内核栈），因此从陷阱返回指令可以容易地恢复。

保存和恢复上下文

既然操作系统已经重新获得了控制权，无论是通过系统调用协作，还是通过时钟中断更强制执行，都必须决定：是继续运行当前正在运行的进程，还是切换到另一个进程。这个决定是由调度程序（scheduler）做出的，它是操作系统的一部分。我们将在接下来的几章中详细讨论调度策略。

如果决定进行切换，OS 就会执行一些底层代码，即所谓的上下文切换（context switch）。上下文切换在概念上很简单：操作系统要做的就是为当前正在执行的进程保存一些寄存器的值（例如，到它的内核栈），并为即将执行的进程恢复一些寄存器的值（从它的内核栈）。这样一来，操作系统就可以确保最后执行从陷阱返回指令时，不是返回到之前运行的进程，而是继续执行另一个进程。

为了保存当前正在运行的进程的上下文，操作系统会执行一些底层汇编代码，来保存通用寄存器、程序计数器，以及当前正在运行的进程的内核栈指针，然后恢复寄存器、程序计数器，并切换内核栈，供即将运行的进程使用。通过切换栈，内核在进入切换代码调用时，是一个进程（被中断的进程）的上下文，在返回时，是另一进程（即将执行的进程）的上下文。当操作系统最终执行从陷阱返回指令时，即将执行的进程变成了当前运行的进程。至此上下文切换完成。