Operating Systems

Three Easy Pieces

Part I

Vitualization

目　录

第 1 部分　虚拟化

第 2 部分　并发

第 3 部分 持久性

第 1 章　关于本书的对话

教授：欢迎阅读这本书，本书英文书名为《Operating Systems:Three Easy Pieces》，由我来讲授关于操作系统的知识。请做一下自我介绍。

学生：教授，您好，我是学生，您可能已经猜到了，我已经准备好开始学习了！

教授：很好。有问题吗？

学生：有！本书为什么讲“3 个简单部分”？

教授：这很简单。理查德 • 费曼有几本关于物理学的讲义，非常不错……

学生：啊，是《别闹了，费曼先生》的作者吗？那本书很棒！这书也会像那本书一样搞笑吗？

教授：呃……不。那本书的确很棒，很高兴你读过它。我希望这本书更像他关于物理学的讲义。将一些基本内容汇集成一本书，名为《Six Easy Pieces》。他讲的是物理学，而我们将探讨的主题是操作系统的 3 个简单部分。这很合适，因为操作系统的难度差不多是物理学的一半。

学生：懂了，我喜欢物理学。是哪 3 个部分呢？

教授：虚拟化（virtualization）、并发（concurrency）和持久性（persistence）。这是我们要学习的 3 个关键概念。通过学习这 3 个概念，我们将理解操作系统是如何工作的，包括它如何决定接下来哪个程序使用 CPU，如何在虚拟内存系统中处理内存使用过载，虚拟机监控器如何工作，如何管理磁盘上的数据，还会讲一点如何构建在部分节点失败时仍能正常工作的分布式系统。

学生：对于您说的这些，我都没有概念。

教授：好极了，这说明你来对了地方。

学生：我还有一个问题：学习这些内容最好的方法是什么？

教授：好问题！当然，每个人都有适合自己的学习方法，但我的方法是：首先听课，听老师讲解并做好笔记，然后每个周末阅读笔记，以便更好地理解这些概念。过一段时间（比如考试前），再阅读一遍笔记来进一步巩固知识。当然老师也肯定会布置作业和项目，你需要认真完成。特别是做项目，你会编写真正的代码来解决真正的问题，这是将笔记中的概念活学活用。就像孔子说的那样……

学生：我知道！“不闻不若闻之，闻之不若见之，见之不若知之，知之不若行之。”

教授：（惊讶）你怎么知道我要说这个？

学生：这样似乎很连贯。我是孔子的粉丝，更是荀子的粉丝，实际上荀子才是说这句话的人[①]。

---

① 儒家思想家荀子曾说过：“不闻不若闻之，闻之不若见之，见之不若知之，知之不若行之。”后来，不知怎么这句名言归到了孔子头上。感谢 Jiao Dong（Rutgers）告诉我们。

教授：（愕然）我猜我们会相处得很愉快。

学生：教授，我还有一个问题，我们这样的对话有什么用的。我是说如果这仅是一本书，为什么您不直接上来就讲述知识呢？

教授：好问题！我觉得有的时候将自己从叙述中抽离出来，然后进行一些思考会更有用。这些对话就是思考。我们将协作探究所有这些复杂的概念。你是为此而来的吗？

学生：所以我们必须思考？好的，我正是为此而来。不过我还有什么要做的吗？看起来我好像就是为此书而生。

教授：我也是。我们开始学习吧！

第 2 章　操作系统介绍

如果你正在读本科操作系统课程，那么应该已经初步了解了计算机程序运行时做的事情。如果不了解，这本书（和相应的课程）对你来说会很困难：你应该停止阅读本书，或跑到最近的书店，在继续读本书之前快速学习必要的背景知识（包括 Patt / Patel [PP03]，特别是 Bryant / O'Hallaron 的书[BOH10]，都是相当不错的）。

程序运行时会发生什么？

一个正在运行的程序会做一件非常简单的事情：执行指令。处理器从内存中获取（fetch）一条指令，对其进行解码（decode）（弄清楚这是哪条指令），然后执行（execute）它（做它应该做的事情，如两个数相加、访问内存、检查条件、跳转到函数等）。完成这条指令后，处理器继续执行下一条指令，依此类推，直到程序最终完成[①]。

这样，我们就描述了冯 • 诺依曼（Von Neumann）计算模型[②]的基本概念。听起来很简单，对吧？但在这门课中，我们将了解到在一个程序运行的同时，还有很多其他疯狂的事情也在同步进行——主要是为了让系统易于使用。

实际上，有一类软件负责让程序运行变得容易（甚至允许你同时运行多个程序），允许程序共享内存，让程序能够与设备交互，以及其他类似的有趣的工作。这些软件称为操作系统（Operating System，OS）[③]，因为它们负责确保系统既易于使用又正确高效地运行。

关键问题：如何将资源虚拟化

我们将在本书中回答一个核心问题：操作系统如何将资源虚拟化？这是关键问题。为什么操作系统这样做？这不是主要问题，因为答案应该很明显：它让系统更易于使用。因此，我们关注如何虚拟化：操作系统通过哪些机制和策略来实现虚拟化？操作系统如何有效地实现虚拟化？需要哪些硬件支持？

我们将用这种灰色文本框来突出“关键（crux）问题”，以此引出我们在构建操作系统时试图解决的具体问题。因此，在关于特定主题的说明中，你可能会发现一个或多个关键点（是的，cruces 是正确的复数形式），它突出了问题。当然，该章详细地提供了解决方案，或至少是解决方案的基本参数。

要做到这一点，操作系统主要利用一种通用的技术，我们称之为虚拟化（virtualization）。也就是说，操作系统将物理（physical）资源（如处理器、内存或磁盘）转换为更通用、更强大且更易于使用的虚拟形式。因此，我们有时将操作系统称为虚拟机（virtual machine）。

当然，为了让用户可以告诉操作系统做什么，从而利用虚拟机的功能（如运行程序、

① 当然，现代处理器在背后做了许多奇怪而可怕的事情，让程序运行得更快。例如，一次执行多条指令，甚至乱序执行并完成它们！但这不是我们在这里关心的问题。我们只关心大多数程序所假设的简单模型：指令似乎按照有序和顺序的方式逐条执行。

② 冯 • 诺依曼是计算系统的早期先驱之一。他还完成了关于博弈论和原子弹的开创性工作，并在 NBA 打了 6 年球。好吧，其中有一件事不是真的。

③ 操作系统的另一个早期名称是监管程序（super visor），甚至叫主控程序（master control program）。显然，后者听起来有些过分热情（详情请参阅电影《Tron》），因此，谢天谢地，“操作系统”最后胜出。

分配内存或访问文件），操作系统还提供了一些接口（API），供你调用。实际上，典型的操作系统会提供几百个系统调用（system call），让应用程序调用。由于操作系统提供这些调用来运行程序、访问内存和设备，并进行其他相关操作，我们有时也会说操作系统为应用程序提供了一个标准库（standard library）。

最后，因为虚拟化让许多程序运行（从而共享 CPU），让许多程序可以同时访问自己的指令和数据（从而共享内存），让许多程序访问设备（从而共享磁盘等），所以操作系统有时被称为资源管理器（resource manager）。每个 CPU、内存和磁盘都是系统的资源（resource），因此操作系统扮演的主要角色就是管理（manage）这些资源，以做到高效或公平，或者实际上考虑其他许多可能的目标。为了更好地理解操作系统的角色，我们来看一些例子。

2.1 虚拟化 CPU

图 2.1 展示了我们的第一个程序。实际上，它没有太大的作用，它所做的只是调用 Spin() 函数，该函数会反复检查时间并在运行一秒后返回。然后，它会打印出用户在命令行中传入的字符串，并一直重复这样做。

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <sys/time.h>
4    #include <assert.h>
5    #include "common.h"
6
7    int
8    main(int argc, char *argv[])
9    {
10       if (argc != 2) {
11           fprintf(stderr, "usage: cpu <string>\n");
12           exit(1);
13       }
14       char *str = argv[1];
15       while (1) {
16           Spin(1);
17           printf("%s\n", str);
18       }
19       return 0;
20   }
```

图 2.1 简单示例：循环打印的代码（cpu.c）

假设我们将这个文件保存为 cpu.c，并决定在一个单处理器（或有时称为 CPU）的系统上编译和运行它。以下是我们将看到的内容：

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
```

```
A
A
^C
prompt>
```

运行不太有趣：系统开始运行程序时，该程序会重复检查时间，直到一秒钟过去。一秒钟过去后，代码打印用户传入的字符串（在本例中为字母“A”）并继续。注意：该程序将永远运行，只有按下“Control-c”（这在基于 UNIX 的系统上将终止在前台运行的程序），才能停止运行该程序。

现在，让我们做同样的事情，但这一次，让我们运行同一个程序的许多不同实例。图 2.2 展示了这个稍复杂的例子的结果。

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

图 2.2 同时运行许多程序

好吧，现在事情开始变得有趣了。尽管我们只有一个处理器，但这 4 个程序似乎在同时运行！这种魔法是如何发生的？[①]

事实证明，在硬件的一些帮助下，操作系统负责提供这种假象（illusion），即系统拥有非常多的虚拟 CPU 的假象。将单个 CPU（或其中一小部分）转换为看似无限数量的 CPU，从而让许多程序看似同时运行，这就是所谓的虚拟化 CPU（virtualizing the CPU），这是本书第一大部分的关注点。

当然，要运行程序并停止它们，或告诉操作系统运行哪些程序，需要有一些接口（API），你可以利用它们将需求传达给操作系统。我们将在本书中讨论这些 API。事实上，它们是大多数用户与操作系统交互的主要方式。

你可能还会注意到，一次运行多个程序的能力会引发各种新问题。例如，如果两个程

① 请注意我们如何利用& 符号同时运行 4 个进程。这样做会在 tcsh shell 的后台运行一个作业，这意味着用户能够立即发出下一个命令，在这个例子中，是另一个运行的程序。命令之间的分号允许我们在 tcsh 中同时运行多个程序。如果你使用的是不同的 shell（例如 bash），它的工作原理会稍有不同。关于详细信息，请阅读在线文档。

序想要在特定时间运行，应该运行哪个？这个问题由操作系统的策略（policy）来回答。在操作系统的许多不同的地方采用了一些策略，来回答这类问题，所以我们将在学习操作系统实现的基本机制（mechanism）（例如一次运行多个程序的能力）时研究这些策略。因此，操作系统承担了资源管理器（resource manager）的角色。

2.2 虚拟化内存

现在让我们考虑一下内存。现代机器提供的物理内存（physical memory）模型非常简单。内存就是一个字节数组。要读取（read）内存，必须指定一个地址（address），才能访问存储在那里的数据。要写入（write）或更新（update）内存，还必须指定要写入给定地址的数据。

程序运行时，一直要访问内存。程序将所有数据结构保存在内存中，并通过各种指令来访问它们，例如加载和保存，或利用其他明确的指令，在工作时访问内存。不要忘记，程序的每个指令都在内存中，因此每次读取指令都会访问内存。

让我们来看一个程序，它通过调用 malloc()来分配一些内存（见图 2.3）。

```
1    #include <unistd.h>
2    #include <stdio.h>
3    #include <stdlib.h>
4    #include "common.h"
5
6    int
7    main(int argc, char *argv[])
8    {
9        int *p = malloc(sizeof(int));              // a1
10       assert(p != NULL);
11       printf("(%d) memory address of p: %08x\n",
12              getpid(), (unsigned) p);            // a2
13       *p = 0;                                    // a3
14       while (1) {
15           Spin(1);
16           *p = *p + 1;
17            printf("(%d) p: %d\n", getpid(), *p); // a4
18       }
19       return 0;
20   }
```

图 2.3 一个访问内存的程序（mem.c）

该程序的输出如下：

```
prompt> ./mem
(2134) memory address of p: 00200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

该程序做了几件事。首先，它分配了一些内存（a1 行）。然后，打印出内存（a2）的地址，然后将数字 0 放入新分配的内存（a3）的第一个空位中。最后，程序循环，延迟一秒钟并递增 p 中保存的地址值。在每个打印语句中，它还会打印出所谓的正在运行程序的进程标识符（PID）。该 PID 对每个运行进程是唯一的。

同样，第一次的结果不太有趣。新分配的内存地址为 00200000。程序运行时，它慢慢地更新值并打印出结果。

现在，我们再次运行同一个程序的多个实例，看看会发生什么（见图 2.4）。我们从示例中看到，每个正在运行的程序都在相同的地址（00200000）处分配了内存，但每个似乎都独立更新了 00200000 处的值！就好像每个正在运行的程序都有自己的私有内存，而不是与其他正在运行的程序共享相同的物理内存[①]。

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 00200000
(24114) memory address of p: 00200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

图 2.4 多次运行内存程序

实际上，这正是操作系统虚拟化内存（virtualizing memory）时发生的情况。每个进程访问自己的私有虚拟地址空间（virtual address space）（有时称为地址空间，address space），操作系统以某种方式映射到机器的物理内存上。一个正在运行的程序中的内存引用不会影响其他进程（或操作系统本身）的地址空间。对于正在运行的程序，它完全拥有自己的物理内存。但实际情况是，物理内存是由操作系统管理的共享资源。所有这些是如何完成的，也是本书第 1 部分的主题，属于虚拟化（virtualization）的主题。

2.3 并发

本书的另一个主题是并发（concurrency）。我们使用这个术语来指代一系列问题，这些问题在同时（并发地）处理很多事情时出现且必须解决。并发问题首先出现在操作系统本身中。如你所见，在上面关于虚拟化的例子中，操作系统同时处理很多事情，首先运行一个进程，然后再运行一个进程，等等。事实证明，这样做会导致一些深刻而有趣的问题。

遗憾的是，并发问题不再局限于操作系统本身。事实上，现代多线程（multi-threaded）程序也存在相同的问题。我们来看一个多线程程序的例子（见图 2.5）。

① 要让这个例子能工作，需要确保禁用地址空间随机化。事实证明，随机化可以很好地抵御某些安全漏洞。请自行阅读更多的相关资料，特别是如果你想学习如何通过堆栈粉碎黑客对计算机系统的攻击入侵。我们不会推荐这样的东西……

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include "common.h"
4
5    volatile int counter = 0;
6    int loops;
7
8    void *worker(void *arg) {
9        int i;
10       for (i = 0; i < loops; i++) {
11           counter++;
12       }
13       return NULL;
14   }
15
16   int
17   main(int argc, char *argv[])
18   {
19       if (argc != 2) {
20           fprintf(stderr, "usage: threads <value>\n");
21           exit(1);
22       }
23       loops = atoi(argv[1]);
24       pthread_t p1, p2;
25       printf("Initial value : %d\n", counter);
26
27       Pthread_create(&p1, NULL, worker, NULL);
28       Pthread_create(&p2, NULL, worker, NULL);
29       Pthread_join(p1, NULL);
30       Pthread_join(p2, NULL);
31       printf("Final value    : %d\n", counter);
32       return 0;
33   }
```

图 2.5 一个多线程程序（threads.c）

尽管目前你可能完全不理解这个例子（在后面关于并发的部分中，我们将学习更多的内容），但基本思想很简单。主程序利用 Pthread_create()创建了两个线程（thread）[①]。你可以将线程看作与其他函数在同一内存空间中运行的函数，并且每次都有多个线程处于活动状态。在这个例子中，每个线程开始在一个名为 worker()的函数中运行，在该函数中，它只是递增一个计数器，循环 loops 次。

下面是运行这个程序、将变量 loops 的输入值设置为 1000 时的输出结果。loops 的值决定了两个 worker 各自在循环中增加共享计数器的次数。如果 loops 的值设置为 1000 并运行程序，你认为计数器的最终值是多少？

关键问题：如何构建正确的并发程序

如果同一个内存空间中有很多并发执行的线程，如何构建一个正确工作的程序？操作系统需要什么原语？硬件应该提供哪些机制？我们如何利用它们来解决并发问题？

① 实际的调用应该是小写的 pthread_create()。大写版本是我们自己的包装函数，它调用 pthread_create()，并确保返回代码指示调用成功。详情请参阅代码。

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

你可能会猜到，两个线程完成时，计数器的最终值为 2000，因为每个线程将计数增加 1000 次。也就是说，当 loops 的输入值设为 *N* 时，我们预计程序的最终输出为 2*N*。但事实证明，事情并不是那么简单。让我们运行相同的程序，但 loops 的值更高，然后看看会发生什么：

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012     // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298     // what the??
```

在这次运行中，当我们提供 100000 的输入值时，得到的最终值不是 200000，我们得到的是 143012。然后，当我们再次运行该程序时，不仅再次得到了错误的值，还与上次的值不同。事实上，如果你一遍又一遍地使用较高的 loops 值运行程序，可能会发现有时甚至可以得到正确的答案！那么为什么会这样？

事实证明，这些奇怪的、不寻常的结果与指令如何执行有关，指令每次执行一条。遗憾的是，上面的程序中的关键部分是增加共享计数器的地方，它需要 3 条指令：一条将计数器的值从内存加载到寄存器，一条将其递增，另一条将其保存回内存。因为这 3 条指令并不是以原子方式（atomically）执行（所有的指令一次性执行）的，所以奇怪的事情可能会发生。关于这种并发（concurrency）问题，我们将在本书的第 2 部分中详细讨论。

2.4 持久性

本课程的第三个主题是持久性（persistence）。在系统内存中，数据容易丢失，因为像 DRAM 这样的设备以易失（volatile）的方式存储数值。如果断电或系统崩溃，那么内存中的所有数据都会丢失。因此，我们需要硬件和软件来持久地（persistently）存储数据。这样的存储对于所有系统都很重要，因为用户非常关心他们的数据。

硬件以某种输入/输出（Input/Output，I/O）设备的形式出现。在现代系统中，硬盘驱动器（hard drive）是存储长期保存的信息的通用存储库，尽管固态硬盘（Solid-State Drive，SSD）正在这个领域取得领先地位。

操作系统中管理磁盘的软件通常称为文件系统（file system）。因此它负责以可靠和高效的方式，将用户创建的任何文件（file）存储在系统的磁盘上。

不像操作系统为 CPU 和内存提供的抽象，操作系统不会为每个应用程序创建专用的虚拟磁盘。相反，它假设用户经常需要共享（share）文件中的信息。例如，在编写 C 程序时，你可能首先使用编辑器（例如 Emacs[①]）来创建和编辑 C 文件（emacs -nw main.c）。之后，你可以使用编译器将源代码转换为可执行文件（例如，gcc -o main main.c）。再之后，你可

① 你应该用 Emacs。如果用 vi，则可能会出现问题。 如果你用的不是真正的代码编辑器，那更糟糕。

以运行新的可执行文件（例如./main）。因此，你可以看到文件如何在不同的进程之间共享。首先，Emacs 创建一个文件，作为编译器的输入。编译器使用该输入文件创建一个新的可执行文件（可选一门编译器课程来了解细节）。最后，运行新的可执行文件。这样一个新的程序就诞生了！

为了更好地理解这一点，我们来看一些代码。图 2.6 展示了一些代码，创建包含字符串“hello world”的文件（/tmp/file）。

```
1    #include <stdio.h>
2    #include <unistd.h>
3    #include <assert.h>
4    #include <fcntl.h>
5    #include <sys/types.h>
6
7    int
8    main(int argc, char *argv[])
9    {
10       int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11       assert(fd > -1);
12       int rc = write(fd, "hello world\n", 13);
13       assert(rc == 13);
14       close(fd);
15       return 0;
16   }
```

图 2.6 一个进行 I/O 的程序（io.c）

关键问题：如何持久地存储数据

文件系统是操作系统的一部分，负责管理持久的数据。持久性需要哪些技术才能正确地实现？需要哪些机制和策略才能高性能地实现？面对硬件和软件故障，可靠性如何实现？

为了完成这个任务，该程序向操作系统发出 3 个调用。第一个是对 open()的调用，它打开文件并创建它。第二个是 write()，将一些数据写入文件。第三个是 close()，只是简单地关闭文件，从而表明程序不会再向它写入更多的数据。这些系统调用（system call）被转到称为文件系统（file system）的操作系统部分，然后该系统处理这些请求，并向用户返回某种错误代码。

你可能想知道操作系统为了实际写入磁盘而做了什么。我们会告诉你，但你必须答应先闭上眼睛。这是不愉快的。文件系统必须做很多工作：首先确定新数据将驻留在磁盘上的哪个位置，然后在文件系统所维护的各种结构中对其进行记录。这样做需要向底层存储设备发出 I/O 请求，以读取现有结构或更新（写入）它们。所有写过设备驱动程序[①]（device driver）的人都知道，让设备代表你执行某项操作是一个复杂而详细的过程。它需要深入了解低级别设备接口及其确切的语义。幸运的是，操作系统提供了一种通过系统调用来访问设备的标准和简单的方法。因此，OS 有时被视为标准库（standard library）。

当然，关于如何访问设备、文件系统如何在所述设备上持久地管理数据，还有更多细节。出于性能方面的原因，大多数文件系统首先会延迟这些写操作一段时间，希望将其批量分

① 设备驱动程序是操作系统中的一些代码，它们知道如何与特定的设备打交道。我们稍后会详细讨论设备和设备驱动程序。

组为较大的组。为了处理写入期间系统崩溃的问题，大多数文件系统都包含某种复杂的写入协议，如日志（journaling）或写时复制（copy-on-write），仔细排序写入磁盘的操作，以确保如果在写入序列期间发生故障，系统可以在之后恢复到合理的状态。为了使不同的通用操作更高效，文件系统采用了许多不同的数据结构和访问方法，从简单的列表到复杂的 B 树。如果所有这些都不太明白，那很好！在本书的第 3 部分关于持久性（persistence）的讨论中，我们将详细讨论所有这些内容，在其中讨论设备和 I/O，然后详细讨论磁盘、RAID 和文件系统。

2.5 设计目标

现在你已经了解了操作系统实际上做了什么：它取得 CPU、内存或磁盘等物理资源（resources），并对它们进行虚拟化（virtualize）。它处理与并发（concurrency）有关的麻烦且棘手的问题。它持久地（persistently）存储文件，从而使它们长期安全。鉴于我们希望建立这样一个系统，所以要有一些目标，以帮助我们集中设计和实现，并在必要时进行折中。找到合适的折中是建立系统的关键。

一个最基本的目标，是建立一些抽象（abstraction），让系统方便和易于使用。抽象对我们在计算机科学中做的每件事都很有帮助。抽象使得编写一个大型程序成为可能，将其划分为小而且容易理解的部分，用 C[①] 这样的高级语言编写这样的程序不用考虑汇编，用汇编写代码不用考虑逻辑门，用逻辑门来构建处理器不用太多考虑晶体管。抽象是如此重要，有时我们会忘记它的重要性，但在这里我们不会忘记。因此，在每一部分中，我们将讨论随着时间的推移而发展的一些主要抽象，为你提供一种思考操作系统部分的方法。

设计和实现操作系统的一个目标，是提供高性能（performance）。换言之，我们的目标是最小化操作系统的开销（minimize the overhead）。虚拟化和让系统易于使用是非常值得的，但不会不计成本。因此，我们必须努力提供虚拟化和其他操作系统功能，同时没有过多的开销。这些开销会以多种形式出现：额外时间（更多指令）和额外空间（内存或磁盘上）。如果有可能，我们会寻求解决方案，尽量减少一种或两种。但是，完美并非总是可以实现的，我们会注意到这一点，并且（在适当的情况下）容忍它。

另一个目标是在应用程序之间以及在 OS 和应用程序之间提供保护（protection）。因为我们希望让许多程序同时运行，所以要确保一个程序的恶意或偶然的不良行为不会损害其他程序。我们当然不希望应用程序能够损害操作系统本身（因为这会影响系统上运行的所有程序）。保护是操作系统基本原理之一的核心，这就是隔离（isolation）。让进程彼此隔离是保护的关键，因此决定了 OS 必须执行的大部分任务。

操作系统也必须不间断运行。当它失效时，系统上运行的所有应用程序也会失效。由于这种依赖性，操作系统往往力求提供高度的可靠性（reliability）。随着操作系统变得越来越复杂（有时包含数百万行代码），构建一个可靠的操作系统是一个相当大的挑战：事实上，该领域的许多正在进行的研究（包括我们自己的一些工作[BS+09，SS+10]），正是专注于这个问题。

其他目标也是有道理的：在我们日益增长的绿色世界中，能源效率（energy-efficiency）

① 你们中的一些人可能不同意将 C 称为高级语言。不过，请记住，这是一门操作系统课程，我们很高兴不需要一直用汇编语言写程序！

非常重要；安全性（security）（实际上是保护的扩展）对于恶意应用程序至关重要，特别是在这高度联网的时代。随着操作系统在越来越小的设备上运行，移动性（mobility）变得越来越重要。根据系统的使用方式，操作系统将有不同的目标，因此可能至少以稍微不同的方式实现。但是，我们会看到，我们将要介绍的关于如何构建操作系统的许多原则，这在各种不同的设备上都很有用。

2.6 简单历史

在结束本章之前，让我们简单介绍一下操作系统的开发历史。就像任何由人类构建的系统一样，随着时间的推移，操作系统中积累了一些好想法，工程师们在设计中学到了重要的东西。在这里，我们简单介绍一下操作系统的几个发展阶段。更丰富的阐述，请参阅 Brinch Hansen 关于操作系统历史的佳作[BH00]。

早期操作系统：只是一些库

一开始，操作系统并没有做太多事情。基本上，它只是一组常用函数库。例如，不是让系统中的每个程序员都编写低级 I/O 处理代码，而是让“OS”提供这样的 API，这样开发人员的工作更加轻松。

通常，在这些老的大型机系统上，一次运行一个程序，由操作员来控制。这个操作员完成了你认为现代操作系统会做的许多事情（例如，决定运行作业的顺序）。如果你是一个聪明的开发人员，就会对这个操作员很好，这样他们可以将你的工作移动到队列的前端。

这种计算模式被称为批（batch）处理，先把一些工作准备好，然后由操作员以“分批”的方式运行。此时，计算机并没有以交互的方式使用，因为这样做成本太高：让用户坐在计算机前使用它，大部分时间它都会闲置，所以会导致设施每小时浪费数千美元[BH00]。

超越库：保护

在超越常用服务的简单库的发展过程中，操作系统在管理机器方面扮演着更为重要的角色。其中一个重要方面是意识到代表操作系统运行的代码是特殊的。它控制了设备，因此对待它的方式应该与对待正常应用程序代码的方式不同。为什么这样？好吧，想象一下，假设允许任何应用程序从磁盘上的任何地方读取。因为任何程序都可以读取任何文件，所以隐私的概念消失了。因此，将一个文件系统（file system）（管理你的文件）实现为一个库是没有意义的。实际上，还需要别的东西。

因此，系统调用（system call）的概念诞生了，它是 Atlas 计算系统[K+61，L78]率先采用的。不是将操作系统例程作为一个库来提供（你只需创建一个过程调用（procedure call）来访问它们），这里的想法是添加一些特殊的硬件指令和硬件状态，让向操作系统过渡变为更正式的、受控的过程。

系统调用和过程调用之间的关键区别在于，系统调用将控制转移（跳转）到 OS 中，同时提高硬件特权级别（hardware privilege level）。用户应用程序以所谓的用户模式（user mode）

运行，这意味着硬件限制了应用程序的功能。例如，以用户模式运行的应用程序通常不能发起对磁盘的 I/O 请求，不能访问任何物理内存页或在网络上发送数据包。在发起系统调用时 [通常通过一个称为陷阱（trap）的特殊硬件指令]，硬件将控制转移到预先指定的陷阱处理程序（trap handler）（即预先设置的操作系统），并同时将特权级别提升到内核模式（kernel mode）。在内核模式下，操作系统可以完全访问系统的硬件，因此可以执行诸如发起 I/O 请求或为程序提供更多内存等功能。当操作系统完成请求的服务时，它通过特殊的陷阱返回（return-from-trap）指令将控制权交还给用户，该指令返回到用户模式，同时将控制权交还给应用程序，回到应用离开的地方。

多道程序时代

操作系统的真正兴起在大主机计算时代之后，即小型机（minicomputer）时代。像数字设备公司（DEC）的 PDP 系列这样的经典机器，让计算机变得更加实惠。因此，不再是每个大型组织拥有一台主机，而是组织内的一小群人可能拥有自己的计算机。毫不奇怪，这种成本下降的主要影响之一是开发者活动的增加。更聪明的人接触到计算机，从而让计算机系统做出更有趣和漂亮的事情。

特别是，由于希望更好地利用机器资源，多道程序（multiprogramming）变得很普遍。操作系统不是一次只运行一项作业，而是将大量作业加载到内存中并在它们之间快速切换，从而提高 CPU 利用率。这种切换非常重要，因为 I/O 设备很慢。在处理 I/O 时让程序占着 CPU，浪费了 CPU 时间。那么，为什么不切换到另一份工作并运行一段时间？

在 I/O 进行和任务中断时，要支持多道程序和重叠运行。这一愿望迫使操作系统创新，沿着多个方向进行概念发展。内存保护（memory protection）等问题变得重要。我们不希望一个程序能够访问另一个程序的内存。了解如何处理多道程序引入的并发（concurrency）问题也很关键。在中断存在的情况下，确保操作系统正常运行是一个很大的挑战。我们将在本书后面研究这些问题和相关主题。

当时主要的实际进展之一是引入了 UNIX 操作系统，主要归功于贝尔实验室（电话公司）的 Ken Thompson 和 Dennis Ritchie。UNIX 从不同的操作系统获得了许多好的想法（特别是来自 Multics [O72]，还有一些来自 TENEX [B+72]和 Berkeley 分时系统[S+68]等系统），但让它们更简单易用。很快，这个团队就向世界各地的人们发送含有 UNIX 源代码的磁带，其中许多人随后参与并添加到系统中。请参阅补充了解更多细节[①]。

摩登时代

除了小型计算机之外，还有一种新型机器，便宜，速度更快，而且适用于大众：今天我们称之为个人计算机（Personal Computer，PC）。在苹果公司早期的机器（如 Apple II）和 IBM PC 的引领下，这种新机器很快就成为计算的主导力量，因为它们的低成本让每个桌子上都有一台机器，而不是每个工作小组共享一台小型机。

遗憾的是，对于操作系统来说，个人计算机起初代表了一次巨大的倒退，因为早期的系统忘

① 我们将使用补充和其他相关文本框，让你注意到不太适合文本主线的各种内容。有时候，我们甚至会用它们来开玩笑，为什么在这个过程中没有一点乐趣？是的，许多笑话都很糟糕。

记了（或从未知道）小型机时代的经验教训。例如，早期的操作系统，如 DOS（来自微软的磁盘操作系统），并不认为内存保护很重要。因此，恶意程序（或者只是一个编程不好的应用程序）可能会在整个内存中乱写乱七八糟的东西。第一代 macOS（V9 及更早版本）采取合作的方式进行作业调度。因此，意外陷入无限循环的线程可能会占用整个系统，从而导致重新启动。这一代系统中遗漏的操作系统功能造成的痛苦列表很长，太长了，因此无法在此进行全面的讨论。

幸运的是，经过一段时间的苦难后，小型计算机操作系统的老功能开始进入台式机。例如，macOS X 的核心是 UNIX，包括人们期望从这样一个成熟系统中获得的所有功能。Windows 在计算历史中同样采用了许多伟大的思想，特别是从 Windows NT 开始，这是微软操作系统技术的一次巨大飞跃。即使在今天的手机上运行的操作系统（如 Linux），也更像小型机在 20 世纪 70 年代运行的，而不像 20 世纪 80 年代 PC 运行的那种操作系统。很高兴看到在操作系统开发鼎盛时期出现的好想法已经进入现代世界。更好的是，这些想法不断发展，为用户和应用程序提供更多功能，让现代系统更加完善。

补充：UNIX 的重要性

在操作系统的历史中，UNIX 的重要性举足轻重。受早期系统（特别是 MIT 著名的 Multics 系统）的影响，UNIX 汇集了许多了不起的思想，创造了既简单又强大的系统。

最初的“贝尔实验室”UNIX 的基础是统一的原则，即构建小而强大的程序，这些程序可以连接在一起形成更大的工作流。在你输入命令的地方，shell 提供了诸如管道（pipe）之类的原语，来支持这样的元（meta-level）编程，因此很容易将程序串起来完成更大的任务。例如，要查找文本文件中包含单词“foo”的行，然后要计算存在多少行，请键入：grep foo file.txt | wc -l，从而使用 grep 和 wc （单词计数）程序来实现你的任务。

UNIX 环境对于程序员和开发人员都很友好，并为新的 C 编程语言提供了编译器。程序员很容易编写自己的程序并分享它们，这使得 UNIX 非常受欢迎。作为开放源码软件（open-source software）的早期形式，作者向所有请求的人免费提供副本，这可能帮助很大。

代码的可得性和可读性也非常重要。用 C 语言编写的美丽的小内核吸引其他人摆弄内核，添加新的、很酷的功能。例如，由 Bill Joy 领导的伯克利创业团队发布了一个非常棒的发行版（Berkeley Systems Distribution，BSD），该发行版拥有先进的虚拟内存、文件系统和网络子系统。Joy 后来与朋友共同创立了 Sun Microsystems。

遗憾的是，随着公司试图维护其所有权和利润，UNIX 的传播速度有所放慢，这是律师参与其中的不幸（但常见的）结果。许多公司都有自己的变种：Sun Microsystems 的 SunOS、IBM 的 AIX、HP 的 HPUX（又名 H-Pucks）以及 SGI 的 IRIX。AT&T/贝尔实验室和这些其他厂商之间的法律纠纷给 UNIX 带来了阴影，许多人想知道它是否能够存活下来，尤其是 Windows 推出后并占领了大部分 PC 市场……

补充：然后出现了 Linux

幸运的是，对于 UNIX 来说，一位名叫 Linus Torvalds 的年轻芬兰黑客决定编写他自己的 UNIX 版本，该版本严重依赖最初系统背后的原则和思想，但没有借用原来的代码集，从而避免了合法性问题。他征集了世界各地许多其他人的帮助，不久，Linux 就诞生了（同时也开启了现代开源软件运动）。

随着互联网时代的到来，大多数公司（如谷歌、亚马逊、Facebook 和其他公司）选择运行 Linux，因为它是免费的，可以随时修改以适应他们的需求。事实上，如果不存在这样一个系统，很难想象这些

新公司的成功。随着智能手机成为占主导地位的面向用户的平台，出于许多相同的原因，Linux 也在那里找到了用武之地（通过 Android）。史蒂夫·乔布斯将他的基于 UNIX 的 NeXTStep 操作环境带到了苹果公司，从而使得 UNIX 在台式机上非常流行（尽管很多苹果技术用户可能都不知道这一事实）。因此，UNIX 今天比以往任何时候都更加重要。如果你相信有计算之神，那么应该感谢这个美妙的结果。

2.7 小结

至此，我们介绍了操作系统。今天的操作系统使得系统相对易于使用，而且你今天使用的几乎所有操作系统都受到本章讨论的操作系统发展的影响。

由于篇幅的限制，我们在本书中将不会涉及操作系统的一些部分。例如，操作系统中有很多网络代码。我们建议你去上网络课以便更多地学习相关知识。同样，图形设备尤为重要。请参加图形课程以扩展你在这方面的知识。最后，一些操作系统书籍谈论了很多关于安全性的内容。我们会这样做，因为操作系统必须在正在运行的程序之间提供保护，并为用户提供保护文件的能力，但我们不会深入研究安全课程中可能遇到的更深层次的安全问题。

但是，我们将讨论许多重要的主题，包括 CPU 和内存虚拟化的基础知识、并发以及通过设备和文件系统的持久性。别担心！虽然有很多内容要介绍，但其中大部分都很酷。这段旅程结束时，你将会对计算机系统的真实工作方式有一个全新的认识。现在开始吧！

参考资料

[BS+09]“Tolerating File-System Mistakes with EnvyFS”
Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
USENIX ’09, San Diego, CA, June 2009
一篇有趣的文章，讲述同时使用多个文件系统以容忍其中任何一个文件系统出现错误。

[BH00]“The Evolution of Operating Systems”
P. Brinch Hansen
In Classic Operating Systems: From Batch Processing to Distributed Systems Springer-Verlag, New York, 2000
这篇文章介绍了与具有历史意义的系统相关的内容。

[B+72]“TENEX, A Paged Time Sharing System for the PDP-10”
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson CACM, Volume 15, Number 3, March 1972
TENEX 拥有现代操作系统中的许多机制。请阅读更多关于它的信息，看看在 20 世纪 70 年代早期已经有了哪些创新。

[B75]“The Mythical Man-Month” Fred Brooks
Addison-Wesley, 1975

一本关于软件工程的经典教科书，非常值得一读。

[BOH10]“Computer Systems: A Programmer’s Perspective”Randal E. Bryant and David R. O’Hallaron
Addison-Wesley, 2010
关于计算机系统工作原理的另一本卓越的图书，与本书的内容有一点点重叠——所以，如果你愿意，你可以跳过本书的最后几章，或者直接阅读它们，以获取关于某些相同材料的不同观点。毕竟，健全与完善自己知识的一个好方法，就是尽可能多地听取其他观点，然后在此问题上扩展自己的观点和想法。

[K+61]“One-Level Storage System”
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner IRE Transactions on Electronic Computers, April 1962
Atlas 开创了你在现代系统中看到的大部分概念。但是，这篇论文并不是最好的读物。如果你只读一篇文章，可以了解一下下面的历史观点[L78]。

[L78]“The Manchester Mark I and Atlas: A Historical Perspective”
S.H. Lavington
Communications of the ACM archive Volume 21, Issue 1 (January 1978), pages 4-12
关于计算机系统早期发展的历史和 Atlas 的开拓性工作。当然，我们可以自己阅读 Atlas 的论文，但是这篇论文提供了一个对计算机系统的很好的概述，并且增加了一些历史观点。

[O72]“The Multics System: An Examination of its Structure”Elliott Organick, 1972
Multics 的完美概述。这么多好的想法，但它是一个过度设计的系统，目标太多，因此从未真正按预期工作。*Fred Brooks* 是所谓的“第二系统效应”的典型例子[B75]。

[PP03]“Introduction to Computing Systems: From Bits and Gates to C and Beyond”
Yale N. Patt and Sanjay J. Patel
McGraw-Hill, 2003
我们最喜欢的计算系统图书之一。它从晶体管开始讲解，一直讲到 C。书中早期的素材特别好。

[RT74]“The UNIX Time-Sharing System”Dennis M. Ritchie and Ken Thompson
CACM, Volume 17, Number 7, July 1974, pages 365-375
关于 UNIX 的杰出总结，作者撰写此书时，UNIX 正在计算世界里占据统治地位。

[S68]“SDS 940 Time-Sharing System”Scientific Data Systems Inc.
TECHNICAL MANUAL, SDS 90 11168 August 1968
这是我们可以找到的一本不错的技术手册。阅读这些旧的系统文件，能看到在 20 世纪 60 年代后期技术发展的进程，这很有意思。伯克利时分系统（最终成为 SDS 系统）背后的核心构建者之一是 Butler Lampson，后来他因系统贡献而获得图灵奖。

[SS+10]“Membrane: Operating System Support for Restartable File Systems”Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift FAST ’10, San Jose, CA, February 2010
写自己的课程注解的好处是：你可以为自己的研究做广告。但是这篇论文实际上非常简洁。当文件系统遇到错误并崩溃时，Membrane 会自动重新启动它，所有这些都不会导致应用程序或系统的其他部分受到影响。

第 3 章　关于虚拟化的对话

教授：现在我们开始讲操作系统 3 个部分的第 1 部分——虚拟化。

学生：尊敬的教授，什么是虚拟化？

教授：想象我们有一个桃子。

学生：桃子？（不可思议）

教授：是的，一个桃子，我们称之为物理（physical）桃子。但有很多想吃这个桃子的人，我们希望向每个想吃的人提供一个属于他的桃子，这样才能皆大欢喜。我们把给每个人的桃子称为虚拟（virtual）桃子。我们通过某种方式，从这个物理桃子创造出许多虚拟桃子。重要的是，在这种假象中，每个人看起来都有一个物理桃子，但实际上不是。

学生：所以每个人都不知道他在和别人一起分享一个桃子吗？

教授：是的。

学生：但不管怎么样，实际情况就是只有一个桃子啊。

教授：是的，所以呢？

学生：所以，如果我和别人分享同一个桃子，我一定会发现这个问题。

教授：是的！你说得没错。但吃的人多才有这样的问题。多数时间他们都在打盹或者做其他事情，所以，你可以在他们打盹的时候把他手中的桃子拿过来分给其他人，这样我们就创造了有许多虚拟桃子的假象，每人一个桃子！

学生：这听起来就像糟糕的竞选口号。教授，您是在跟我讲计算机知识吗？

教授：年轻人，看来需要给你一个更具体的例子。以最基本的计算机资源 CPU 为例，假设一个计算机只有一个 CPU（尽管现代计算机一般拥有 2 个、4 个或者更多 CPU），虚拟化要做的就是将这个 CPU 虚拟成多个虚拟 CPU 并分给每一个进程使用，因此，每个应用都以为自己在独占 CPU，但实际上只有一个 CPU。这样操作系统就创造了美丽的假象——它虚拟化了 CPU。

学生：听起来好神奇，能再多讲一些吗？它是如何工作的？

教授：问得很及时，听上去你已经做好开始学习的准备了。

学生：是的，不过我还真有点担心您又要讲桃子的事情了。

教授：不用担心，毕竟我也不喜欢吃桃子。那我们开始学习吧……

第 4 章　抽象：进程

本章讨论操作系统提供的基本的抽象——进程。进程的非正式定义非常简单：进程就是运行中的程序。程序本身是没有生命周期的，它只是存在磁盘上面的一些指令（也可能是一些静态数据）。是操作系统让这些字节运行起来，让程序发挥作用。

事实表明，人们常常希望同时运行多个程序。比如：在使用计算机或者笔记本的时候，我们会同时运行浏览器、邮件、游戏、音乐播放器，等等。实际上，一个正常的系统可能会有上百个进程同时在运行。如果能实现这样的系统，人们就不需要考虑这个时候哪一个 CPU 是可用的，使用起来非常简单。因此我们的挑战是:

关键问题：如何提供有许多 CPU 的假象？

虽然只有少量的物理 CPU 可用，但是操作系统如何提供几乎有无数个 CPU 可用的假象？

操作系统通过虚拟化（virtualizing）CPU 来提供这种假象。通过让一个进程只运行一个时间片，然后切换到其他进程，操作系统提供了存在多个虚拟 CPU 的假象。这就是时分共享（time sharing）CPU 技术，允许用户如愿运行多个并发进程。潜在的开销就是性能损失，因为如果 CPU 必须共享，每个进程的运行就会慢一点。

要实现 CPU 的虚拟化，要实现得好，操作系统就需要一些低级机制以及一些高级智能。我们将低级机制称为机制（mechanism）。机制是一些低级方法或协议，实现了所需的功能。例如，我们稍后将学习如何实现上下文切换（context switch），它让操作系统能够停止运行一个程序，并开始在给定的 CPU 上运行另一个程序。所有现代操作系统都采用了这种分时机制。

提示：使用时分共享（和空分共享）

时分共享（time sharing）是操作系统共享资源所使用的最基本的技术之一。通过允许资源由一个实体使用一小段时间，然后由另一个实体使用一小段时间，如此下去，所谓的资源（例如，CPU 或网络链接）可以被许多人共享。时分共享的自然对应技术是空分共享，资源在空间上被划分给希望使用它的人。例如，磁盘空间自然是一个空分共享资源，因为一旦将块分配给文件，在用户删除文件之前，不可能将它分配给其他文件。

在这些机制之上，操作系统中有一些智能以策略（policy）的形式存在。策略是在操作系统内做出某种决定的算法。例如，给定一组可能的程序要在 CPU 上运行，操作系统应该运行哪个程序？操作系统中的调度策略（scheduling policy）会做出这样的决定，可能利用历史信息（例如，哪个程序在最后一分钟运行得更多？）、工作负载知识（例如，运行什么类型的程序？）以及性能指标 （例如，系统是否针对交互式性能或吞吐量进行优化？）来做出决定。

4.1 抽象：进程

操作系统为正在运行的程序提供的抽象，就是所谓的进程（process）。正如我们上面所说的，一个进程只是一个正在运行的程序。在任何时刻，我们都可以清点它在执行过程中访问或影响的系统的不同部分，从而概括一个进程。

为了理解构成进程的是什么，我们必须理解它的机器状态（machine state）：程序在运行时可以读取或更新的内容。在任何时刻，机器的哪些部分对执行该程序很重要？

进程的机器状态有一个明显组成部分，就是它的内存。指令存在内存中。正在运行的程序读取和写入的数据也在内存中。因此进程可以访问的内存（称为地址空间，address space）是该进程的一部分。

进程的机器状态的另一部分是寄存器。许多指令明确地读取或更新寄存器，因此显然，它们对于执行该进程很重要。

请注意，有一些非常特殊的寄存器构成了该机器状态的一部分。例如，程序计数器（Program Counter，PC）（有时称为指令指针，Instruction Pointer 或 IP）告诉我们程序当前正在执行哪个指令；类似地，栈指针（stack pointer）和相关的帧指针（frame pointer）用于管理函数参数栈、局部变量和返回地址。

提示：分离策略和机制

在许多操作系统中，一个通用的设计范式是将高级策略与其低级机制分开[L+75]。你可以将机制看成为系统的“如何（how）”问题提供答案。例如，操作系统如何执行上下文切换？策略为“哪个（which）”问题提供答案。例如，操作系统现在应该运行哪个进程？将两者分开可以轻松地改变策略，而不必重新考虑机制，因此这是一种模块化（modularity）的形式，一种通用的软件设计原则。

最后，程序也经常访问持久存储设备。此类 I/O 信息可能包含当前打开的文件列表。

4.2 进程 API

虽然讨论真实的进程 API 将推迟到第 5 章讲解，但这里先介绍一下操作系统的所有接口必须包含哪些内容。所有现代操作系统都以某种形式提供这些 API。

- **创建（create）**：操作系统必须包含一些创建新进程的方法。在 shell 中键入命令或双击应用程序图标时，会调用操作系统来创建新进程，运行指定的程序。
- **销毁（destroy）**：由于存在创建进程的接口，因此系统还提供了一个强制销毁进程的接口。当然，很多进程会在运行完成后自行退出。但是，如果它们不退出，用户可能希望终止它们，因此停止失控进程的接口非常有用。
- **等待（wait）**：有时等待进程停止运行是有用的，因此经常提供某种等待接口。
- **其他控制（miscellaneous control）**：除了杀死或等待进程外，有时还可能有其他

控制。例如，大多数操作系统提供某种方法来暂停进程（停止运行一段时间），然后恢复（继续运行）。

- **状态（statu）**：通常也有一些接口可以获得有关进程的状态信息，例如运行了多长时间，或者处于什么状态。

4.3 进程创建：更多细节

我们应该揭开一个谜，就是程序如何转化为进程。具体来说，操作系统如何启动并运行一个程序？进程创建实际如何进行？

操作系统运行程序必须做的第一件事是将代码和所有静态数据（例如初始化变量）加载（load）到内存中，加载到进程的地址空间中。程序最初以某种可执行格式驻留在磁盘上（disk，或者在某些现代系统中，在基于闪存的 SSD 上）。因此，将程序和静态数据加载到内存中的过程，需要操作系统从磁盘读取这些字节，并将它们放在内存中的某处（见图 4.1）。

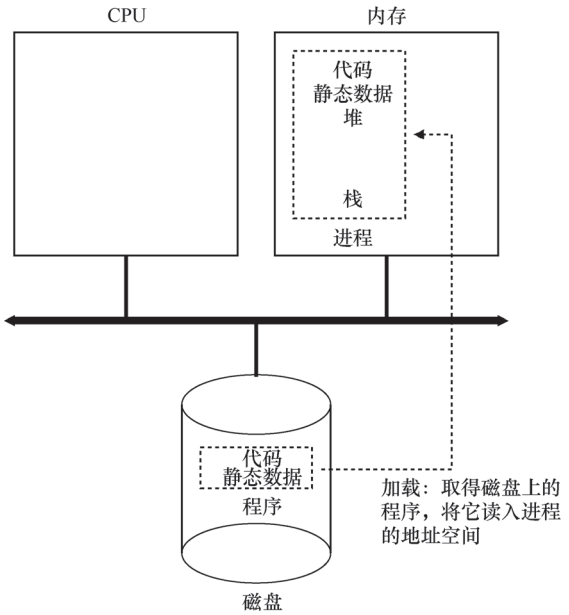


图 4.1 加载：从程序到进程

在早期的（或简单的）操作系统中，加载过程尽早（eagerly）完成，即在运行程序之前全部完成。现代操作系统惰性（lazily）执行该过程，即仅在程序执行期间需要加载的代码或数据片段，才会加载。要真正理解代码和数据的惰性加载是如何工作的，必须更多地了解分页和交换的机制，这是我们将来讨论内存虚拟化时要涉及的主题。现在，只要记住在运行任何程序之前，操作系统显然必须做一些工作，才能将重要的程序字节从磁盘读入内存。

将代码和静态数据加载到内存后，操作系统在运行此进程之前还需要执行其他一些操作。必须为程序的运行时栈（run-time stack 或 stack）分配一些内存。你可能已经知道，C

程序使用栈存放局部变量、函数参数和返回地址。操作系统分配这些内存，并提供给进程。操作系统也可能会用参数初始化栈。具体来说，它会将参数填入 main()函数，即 argc 和 argv 数组。

操作系统也可能为程序的堆（heap）分配一些内存。在 C 程序中，堆用于显式请求的动态分配数据。程序通过调用 malloc()来请求这样的空间，并通过调用 free()来明确地释放它。数据结构（如链表、散列表、树和其他有趣的数据结构）需要堆。起初堆会很小。随着程序运行，通过 malloc()库 API 请求更多内存，操作系统可能会参与分配更多内存给进程，以满足这些调用。

操作系统还将执行一些其他初始化任务，特别是与输入/输出（I/O）相关的任务。例如，在 UNIX 系统中，默认情况下每个进程都有 3 个打开的文件描述符（file descriptor），用于标准输入、输出和错误。这些描述符让程序轻松读取来自终端的输入以及打印输出到屏幕。在本书的第 3 部分关于持久性（persistence）的知识中，我们将详细了解 I/O、文件描述符等。

通过将代码和静态数据加载到内存中，通过创建和初始化栈以及执行与 I/O 设置相关的其他工作，OS 现在（终于）为程序执行搭好了舞台。然后它有最后一项任务：启动程序，在入口处运行，即 main()。通过跳转到 main()例程（第 5 章讨论的专门机制），OS 将 CPU 的控制权转移到新创建的进程中，从而程序开始执行。

4.4 进程状态

既然已经了解了进程是什么（但我们会继续改进这个概念），以及（大致）它是如何创建的，让我们来谈谈进程在给定时间可能处于的不同状态（state）。在早期的计算机系统[DV66，V+65]中，出现了一个进程可能处于这些状态之一的概念。简而言之，进程可以处于以下 3 种状态之一。

- **运行（running）**：在运行状态下，进程正在处理器上运行。这意味着它正在执行指令。
- **就绪（ready）**：在就绪状态下，进程已准备好运行，但由于某种原因，操作系统选择不在此时运行。
- **阻塞（blocked）**：在阻塞状态下，一个进程执行了某种操作，直到发生其他事件时才会准备运行。一个常见的例子是，当进程向磁盘发起 I/O 请求时，它会被阻塞，因此其他进程可以使用处理器。

如果将这些状态映射到一个图上，会得到图 4.2。如图 4.2 所示，可以根据操作系统的载量，让进程在就绪状态和运行状态之间转换。从就绪到运行意味着该进程已经被调度（scheduled）。从运行转移到就绪意味着该进程已经取消调度（descheduled）。一旦进程被阻塞（例如，通过发起 I/O 操作），OS 将保持进程的这种状态，直到发生某种事件（例如，I/O 完成）。此时，进程再次转入就绪状态（也可能立即再次运行，如果操作系统这样决定）。

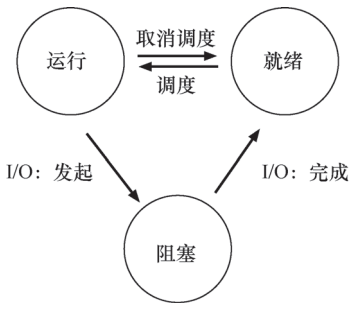


图 4.2 进程：状态转换

我们来看一个例子，看两个进程如何通过这些状态转换。首先，想象两个正在运行的进程，每个进程只使用 CPU（它们没有 I/O）。在这种情况下，每个进程的状态可能如表 4.1 所示。

表 4.1 跟踪进程状态：只看 CPU

时间	Process0	Process1	注
1	运行	就绪	
2	运行	就绪	
3	运行	就绪	
4	运行	就绪	Process0 现在完成
5	—	运行	
6	—	运行	
7	—	运行	
8	—	运行	Process1 现在完成

在下一个例子中，第一个进程在运行一段时间后发起 I/O 请求。此时，该进程被阻塞，让另一个进程有机会运行。表 4.2 展示了这种场景。

表 4.2 跟踪进程状态：CPU 和 I/O

时间	Process0	Process1	注
1	运行	就绪	
2	运行	就绪	
3	运行	就绪	Process0 发起 I/O
4	阻塞	运行	Process0 被阻塞
5	阻塞	运行	所以 Process1 运行
6	阻塞	运行	
7	就绪	运行	I/O 完成
8	就绪	运行	Process1 现在完成
9	运行	—	
10	运行	—	Process0 现在完成

更具体地说，Process0 发起 I/O 并被阻塞，等待 I/O 完成。例如，当从磁盘读取数据或等待网络数据包时，进程会被阻塞。OS 发现 Process0 不使用 CPU 并开始运行 Process1。当 Process1 运行时，I/O 完成，将 Process0 移回就绪状态。最后，Process1 结束，Process0 运行，然后完成。

请注意，即使在这个简单的例子中，操作系统也必须做出许多决定。首先，系统必须决定在 Process0 发出 I/O 时运行 Process1。这样做可以通过保持 CPU 繁忙来提高资源利用率。其次，当 I/O 完成时，系统决定不切换回 Process0。目前还不清楚这是不是一个很好的决定。你怎么看？这些类型的决策由操作系统调度程序完成，这是我们在未来几章讨论的主题。

4.5 数据结构

操作系统是一个程序，和其他程序一样，它有一些关键的数据结构来跟踪各种相关的信息。例如，为了跟踪每个进程的状态，操作系统可能会为所有就绪的进程保留某种进程列表（process list），以及跟踪当前正在运行的进程的一些附加信息。操作系统还必须以某种方式跟踪被阻塞的进程。当 I/O 事件完成时，操作系统应确保唤醒正确的进程，让它准备好再次运行。

图 4.3 展示了 OS 需要跟踪 xv6 内核中每个进程的信息类型[CK+08]。“真正的”操作系统中存在类似的进程结构，如 Linux、macOS X 或 Windows。查看它们，看看有多复杂。

从图 4.3 中可以看到，操作系统追踪进程的一些重要信息。对于停止的进程，寄存器上下文将保存其寄存器的内容。当一个进程停止时，它的寄存器将被保存到这个内存位置。通过恢复这些寄存器（将它们的值放回实际的物理寄存器中），操作系统可以恢复运行该进程。我们将在后面的章节中更多地了解这种技术，它被称为上下文切换（context switch）。

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
```

```
  struct inode *cwd;           // Current directory
  struct context context;      // Switch here to run process
  struct trapframe *tf;        // Trap frame for the
                               // current interrupt
};
```

图 4.3　xv6 的 proc 结构

从图 4.3 中还可以看到，除了运行、就绪和阻塞之外，还有其他一些进程可以处于的状态。有时候系统会有一个初始（initial）状态，表示进程在创建时处于的状态。另外，一个进程可以处于已退出但尚未清理的最终（final）状态（在基于 UNIX 的系统中，这称为僵尸状态[①]）。这个最终状态非常有用，因为它允许其他进程（通常是创建进程的父进程）检查进程的返回代码，并查看刚刚完成的进程是否成功执行（通常，在基于 UNIX 的系统中，程序成功完成任务时返回零，否则返回非零）。完成后，父进程将进行最后一次调用（例如，wait()），以等待子进程的完成，并告诉操作系统它可以清理这个正在结束的进程的所有相关数据结构。

补充：数据结构——进程列表

操作系统充满了我们将在这些讲义中讨论的各种重要数据结构（data structure）。进程列表（process list）是第一个这样的结构。这是比较简单的一种，但是，任何能够同时运行多个程序的操作系统当然都会有类似这种结构的东西，以便跟踪系统中正在运行的所有程序。有时候人们会将存储关于进程的信息的个体结构称为进程控制块（Process Control Block，PCB），这是谈论包含每个进程信息的 C 结构的一种方式。

4.6　小结

我们已经介绍了操作系统的最基本抽象：进程。它很简单地被视为一个正在运行的程序。有了这个概念，接下来将继续讨论具体细节：实现进程所需的低级机制和以智能方式调度这些进程所需的高级策略。结合机制和策略，我们将加深对操作系统如何虚拟化 CPU 的理解。

参考资料

[BH70]“The Nucleus of a Multiprogramming System”Per Brinch Hansen
Communications of the ACM, Volume 13, Number 4, April 1970
本文介绍了 Nucleus，它是操作系统历史上的第一批微内核（microkernel）之一。体积更小、系统更小的想法，在操作系统历史上是不断重复的主题。这一切都始于 Brinch Hansen 在这里描述的工作。

① 是的，僵尸状态。 就像真正的僵尸一样，这些“僵尸”相对容易杀死。 但是，通常建议使用不同的技术。

[CK+08]“The xv6 Operating System”
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich
xv6 是世界上颇有魅力的、真实的小型操作系统。请下载并利用它来了解更多关于操作系统实际工作方式的细节。

[DV66]“Programming Semantics for Multiprogrammed Computations”Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
本文定义了构建多道程序系统的许多早期术语和概念。

[L+75]“Policy/mechanism separation in Hydra”
R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf SOSP 1975
一篇关于如何在名为 Hydra 的研究操作系统中构建一些操作系统的早期论文。虽然 Hydra 从未成为主流操作系统，但它的一些想法影响了操作系统设计人员。

[V+65]“Structure of the Multics Supervisor”
V.A. Vyssotsky, F. J. Corbato, R. M. Graham Fall Joint Computer Conference, 1965
一篇关于 Multics 的早期论文，描述了我们在现代系统中看到的许多基本概念和术语。计算作为实用工具，这背后的一些愿景终于在现代云系统中得以实现。

作业

补充：模拟作业

模拟作业以模拟器的形式出现，你运行它以确保理解某些内容。模拟器通常是 Python 程序，它们让你能够生成不同的问题（使用不同的随机种子），也让程序为你解决问题（带-c 标志），以便你检查答案。使用-h 或--help 标志运行任何模拟器，将提供有关模拟器所有选项的更多信息。

每个模拟器附带的 README 文件提供了有关如何运行它的更多详细信息，其中详细描述了每个标志。

程序 process-run.py 让你查看程序运行时进程状态如何改变，是在使用 CPU（例如，执行相加指令）还是执行 I/O（例如，向磁盘发送请求并等待它完成）。详情请参阅 README 文件。

问题

1．用以下标志运行程序：./process-run.py -l 5:100,5:100。CPU 利用率（CPU 使用时间的百分比）应该是多少？为什么你知道这一点？利用 -c 标记查看你的答案是否正确。

2．现在用这些标志运行：./process-run.py -l 4:100,1:0。这些标志指定了一个包含 4 条指

令的进程（都要使用 CPU），并且只是简单地发出 I/O 并等待它完成。完成这两个进程需要多长时间？利用-c 检查你的答案是否正确。

3．现在交换进程的顺序：./process-run.py -l 1:0,4:100。现在发生了什么？交换顺序是否重要？为什么？同样，用-c 看看你的答案是否正确。

4．现在探索另一些标志。一个重要的标志是-S，它决定了当进程发出 I/O 时系统如何反应。将标志设置为 SWITCH_ON_END，在进程进行 I/O 操作时，系统将不会切换到另一个进程，而是等待进程完成。当你运行以下两个进程时，会发生什么情况？一个执行 I/O，另一个执行 CPU 工作。（-l 1:0,4:100 -c -S SWITCH_ON_END）

5．现在，运行相同的进程，但切换行为设置，在等待 I/O 时切换到另一个进程（-l 1:0,4:100 -c -S SWITCH_ON_IO）。现在会发生什么？利用-c 来确认你的答案是否正确。

6．另一个重要的行为是 I/O 完成时要做什么。利用-I IO_RUN_LATER，当 I/O 完成时，发出它的进程不一定马上运行。相反，当时运行的进程一直运行。当你运行这个进程组合时会发生什么？（./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p）系统资源是否被有效利用？

7．现在运行相同的进程，但使用-I IO_RUN_IMMEDIATE 设置，该设置立即运行发出 I/O 的进程。这种行为有何不同？为什么运行一个刚刚完成 I/O 的进程会是一个好主意？

8．现在运行一些随机生成的进程，例如-s 1 -l 3:50,3:50, -s 2 -l 3:50,3:50, -s 3 -l 3:50,3:50。看看你是否能预测追踪记录会如何变化？当你使用-I IO_RUN_IMMEDIATE 与-I IO_RUN_LATER 时会发生什么？当你使用-S SWITCH_ON_IO 与-S SWITCH_ON_END 时会发生什么？

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。

表 6.3 展示了整个过程的时间线。在这个例子中，进程 A 正在运行，然后被中断时钟中断。硬件保存它的寄存器（在内核栈中），并进入内核（切换到内核模式）。在时钟中断处理程序中，操作系统决定从正在运行的进程 A 切换到进程 B。此时，它调用 switch()例程，该例程仔细保存当前寄存器的值（保存到 A 的进程结构），恢复寄存器进程 B（从它的进程结构），然后切换上下文（switch context），具体来说是通过改变栈指针来使用 B 的内核栈（而不是 A 的）。最后，操作系统从陷阱返回，恢复 B 的寄存器并开始运行它。

表 6.3　　受限直接执行协议（时钟中断）

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住以下地址： 　系统调用处理程序 　时钟处理程序	
启动中断时钟		
	启动时钟 每隔 *x* ms 中断 CPU	
操作系统@运行（内核模式）	硬件	程序（应用模式）
		进程 A……
	时钟中断 将寄存器（A）保存到内核栈（A） 转向内核模式 跳到陷阱处理程序	
处理陷阱 调用 switch()例程 　将寄存器（A）保存到进程结构（A） 　将进程结构（B）恢复到寄存器（B） 从陷阱返回（进入 B）		
	从内核栈（B）恢复寄存器（B） 转向用户模式 跳到 B 的程序计数器	
		进程 B……

请注意，在此协议中，有两种类型的寄存器保存/恢复。第一种是发生时钟中断的时候。在这种情况下，运行进程的用户寄存器由硬件隐式保存，使用该进程的内核栈。第二种是当操作系统决定从 A 切换到 B。在这种情况下，内核寄存器被软件（即 OS）明确地保存，但这次被存储在该进程的进程结构的内存中。后一个操作让系统从好像刚刚由 A 陷入内核，变成好像刚刚由 B 陷入内核。

为了让你更好地了解如何实现这种切换，图 6.1 给出了 xv6 的上下文切换代码。看看你是否能理解它（你必须知道一点 x86 和一点 xv6）。context 结构 old 和 new 分别在老的和新的进程的进程结构中。

```
1    # void swtch(struct context **old, struct context *new);
2    #
3    # Save current register context in old
4    # and then load register context from new.
5    .globl swtch
```

```
6    swtch:
7      # Save old registers
8      movl 4(%esp), %eax  # put old ptr into eax
9      popl 0(%eax)        # save the old IP
10     movl %esp, 4(%eax)  # and stack
11     movl %ebx, 8(%eax)  # and other registers
12     movl %ecx, 12(%eax)
13     movl %edx, 16(%eax)
14     movl %esi, 20(%eax)
15     movl %edi, 24(%eax)
16     movl %ebp, 28(%eax)
17
18     # Load new registers
19     movl 4(%esp), %eax  # put new ptr into eax
20     movl 28(%eax), %ebp # restore other registers
21     movl 24(%eax), %edi
22     movl 20(%eax), %esi
23     movl 16(%eax), %edx
24     movl 12(%eax), %ecx
25     movl 8(%eax), %ebx
26     movl 4(%eax), %esp  # stack is switched here
27     pushl 0(%eax)       # return addr put in place
28     ret                 # finally return into new ctxt
```

图 6.1 xv6 的上下文切换代码

6.4 担心并发吗

作为细心周到的读者，你们中的一些人现在可能会想：“呃……在系统调用期间发生时钟中断时会发生什么？”或“处理一个中断时发生另一个中断，会发生什么？这不会让内核难以处理吗？”好问题——我们真的对你抱有一点希望！

答案是肯定的，如果在中断或陷阱处理过程中发生另一个中断，那么操作系统确实需要关心发生了什么。实际上，这正是本书第 2 部分关于并发的主题。那时我们将详细讨论。

补充：上下文切换要多长时间

你可能有一个很自然的问题：上下文切换需要多长时间？甚至系统调用要多长时间？如果感到好奇，有一种称为 lmbench [MS96]的工具，可以准确衡量这些事情，并提供其他一些可能相关的性能指标。

随着时间的推移，结果有了很大的提高，大致跟上了处理器的性能提高。例如，1996 年在 200-MHz P6 CPU 上运行 Linux 1.3.37，系统调用花费了大约 4μs，上下文切换时间大约为 6μs[MS96]。现代系统的性能几乎可以提高一个数量级，在具有 2 GHz 或 3 GHz 处理器的系统上的性能可以达到亚微秒级。

应该注意的是，并非所有的操作系统操作都会跟踪 CPU 的性能。正如 Ousterhout 所说的，许多操作系统操作都是内存密集型的，而随着时间的推移，内存带宽并没有像处理器速度那样显著提高[O90]。因此，根据你的工作负载，购买最新、性能好的处理器可能不会像你希望的那样加速操作系统。

为了让你开开胃，我们只是简单介绍了操作系统如何处理这些棘手的情况。操作系统可能简单地决定，在中断处理期间禁止中断（disable interrupt）。这样做可以确保在处理一个中断时，不会将其他中断交给 CPU。当然，操作系统这样做必须小心。禁用中断时间过长可能导致丢失中断，这（在技术上）是不好的。

操作系统还开发了许多复杂的加锁（locking）方案，以保护对内部数据结构的并发访问。这使得多个活动可以同时在内核中进行，特别适用于多处理器。我们在本书下一部分关于并发的章节中将会看到，这种锁可能会变得复杂，并导致各种有趣且难以发现的错误。

6.5 小结

我们已经描述了一些实现 CPU 虚拟化的关键底层机制，并将其统称为受限直接执行（limited direct execution）。基本思路很简单：就让你想运行的程序在 CPU 上运行，但首先确保设置好硬件，以便在没有操作系统帮助的情况下限制进程可以执行的操作。

这种一般方法也在现实生活中采用。例如，那些有孩子或至少听说过孩子的人可能会熟悉宝宝防护（baby proofing）房间的概念——锁好包含危险物品的柜子，并掩盖电源插座。当这些都准备妥当时，你可以让宝宝自由行动，确保房间最危险的方面受到限制。

提示：重新启动是有用的

之前我们指出，在协作式抢占时，无限循环（以及类似行为）的唯一解决方案是重启（reboot）机器。虽然你可能会嘲笑这种粗暴的做法，但研究表明，重启（或在通常意义上说，重新开始运行一些软件）可能是构建强大系统的一个非常有用的工具[C+04]。

具体来说，重新启动很有用，因为它让软件回到已知的状态，很可能是经过更多测试的状态。重新启动还可以回收旧的或泄露的资源（例如内存），否则这些资源可能很难处理。最后，重启很容易自动化。由于所有这些原因，在大规模集群互联网服务中，系统管理软件定期重启一些机器，重置它们并因此获得以上好处，这并不少见。

因此，下次重启时，要相信自己不是在进行某种丑陋的粗暴攻击。实际上，你正在使用经过时间考验的方法来改善计算机系统的行为。干得漂亮！

通过类似的方式，OS 首先（在启动时）设置陷阱处理程序并启动时钟中断，然后仅在受限模式下运行进程，以此为 CPU 提供“宝宝防护”。这样做，操作系统能确信进程可以高效运行，只在执行特权操作，或者当它们独占 CPU 时间过长并因此需要切换时，才需要操作系统干预。

至此，我们有了虚拟化 CPU 的基本机制。但一个主要问题还没有答案：在特定时间，我们应该运行哪个进程？调度程序必须回答这个问题，因此这也是我们研究的下一个主题。

参考资料

[A79]“Alto User’s Handbook”
Xerox Palo Alto Research Center, September 1979

这是一个惊人的系统，其影响远超它的预期。之所以出名，是因为史蒂夫·乔布斯读过它，他记了笔记，由此创建了 Lisa，最终将其变成了 Mac。

[C+04]“Microreboot — A Technique for Cheap Recovery”
George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, Armando Fox OSDI ’04, San Francisco, CA, December 2004
一篇优秀的论文，指出了在建立更健壮的系统时重启可以做到什么程度。

[I11]“Intel 64 and IA-32 Architectures Software Developer’s Manual”Volume 3A and 3B: System Programming Guide
Intel Corporation, January 2011

[K+61]“One-Level Storage System”
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner IRE Transactions on Electronic Computers, April 1962
Atlas 开创了你在现代系统中看到的大部分技术。但是，这篇论文并不是最好的一篇。如果你只打算阅读一篇，不妨看看其中历史观点[L78]。

[L78]“The Manchester Mark I and Atlas: A Historical Perspective”
S. H. Lavington
Communications of the ACM, 21:1, January 1978
计算机早期发展的历史和 Atlas 的开拓性工作。

[M+63]“A Time-Sharing Debugging System for a Small Computer”
J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider AFIPS ’63 (Spring), May, 1963, New York, USA
关于分时共享的早期文章，提出使用时钟中断。这篇文章讨论了这个问题：“通道 17 时钟例程的基本任务，是决定是否将当前用户从核心中移除，如果移除，则决定在移除它时换成哪个用户程序。”

[MS96]“lmbench: Portable tools for performance analysis”Larry McVoy and Carl Staelin
USENIX Annual Technical Conference, January 1996
一篇有趣的文章，关于如何测量关于操作系统及其性能的许多不同指标。请下载 lmbench 并试一试。

[M11]“macOS 9”
January 2011

[O90]“Why Aren’t Operating Systems Getting Faster as Fast as Hardware?”
J. Ousterhout
USENIX Summer Conference, June 1990
一篇关于操作系统性能本质的经典论文。

[P10]“The Single UNIX Specification, Version 3”The Open Group, May 2010
该文读起来晦涩难懂，不建议阅读。

[S07]“The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”Hovav Shacham

CCS ’07, October 2007

有些文章会让你在研读过程中不时看到一些令人惊叹、令人兴奋的想法，这就是其中之一。作者告诉你，如果你可以随意跳入代码，就可以将你喜欢的任何代码序列（给定一个大代码库）进行基本拼接。请阅读文中的细节。这项技术使得抵御恶意攻击更难。

作业（测量）

补充：测量作业

测量作业是小型练习。你可以编写代码在真实机器上运行，从而测量操作系统或硬件性能的某些方面。这样的作业背后的想法是给你一点实际操作系统的实践经验。

在这个作业中，你将测量系统调用和上下文切换的成本。测量系统调用的成本相对容易。例如，你可以重复调用一个简单的系统调用（例如，执行 0 字节读取）并记下所花的时间。将时间除以迭代次数，就可以估计系统调用的成本。

你必须考虑的一件事是时钟的精确性和准确性。你可以使用的典型时钟是 gettimeofday()。详细信息请阅读手册页。你会看到，gettimeofday()返回自 1970 年以来的微秒时间。然而，这并不意味着时钟精确到微秒。测量 gettimeofday()的连续调用，以了解时钟的精确度。这会告诉你为了获得一个好的测量结果，需要让空系统调用测试的迭代运行多少次。如果 gettimeofday()对你来说不够精确，可以考虑利用 x86 机器提供的 rdtsc 指令。

测量上下文切换的成本有点棘手。lmbench 基准测试的实现方法，是在单个 CPU 上运行两个进程并在它们之间设置两个 UNIX 管道。管道只是 UNIX 系统中的进程可以相互通信的许多方式之一。第一个进程向第一个管道写入数据，然后等待第二个数据的读取。由于看到第一个进程等待从第二个管道读取的内容，OS 将第一个进程置于阻塞状态，并切换到另一个进程，该进程从第一个管道读取数据，然后写入第二个管理。当第二个进程再次尝试从第一个管道读取时，它会阻塞，从而继续进行通信的往返循环。通过反复测量这种通信的成本，lmbench 可以很好地估计上下文切换的成本。你可以尝试使用管道或其他通信机制（例如 UNIX 套接字），重新创建类似的东西。

在具有多个 CPU 的系统中，测量上下文切换成本有一点困难。在这样的系统上，你需要确保你的上下文切换进程处于同一个处理器上。幸运的是，大多数操作系统都会提供系统调用，让一个进程绑定到特定的处理器。例如，在 Linux 上，sched_setaffinity()调用就是你要查找的内容。通过确保两个进程位于同一个处理器上，你就能确保在测量操作系统停止一个进程并在同一个 CPU 上恢复另一个进程的成本。

第 7 章　进程调度：介绍

现在，运行进程的底层机制（mechanism）（如上下文切换）应该清楚了。如果还不清楚，请往回翻一两章，再次阅读这些工作原理的描述。然而，我们还不知道操作系统调度程序采用的上层策略（policy）。接下来会介绍一系列的调度策略（sheduling policy，有时称为 discipline），它们是许多聪明又努力的人在过去这些年里开发的。

事实上，调度的起源早于计算机系统。早期调度策略取自于操作管理领域，并应用于计算机。对于这个事实不必惊讶：装配线以及许多人类活动也需要调度，而且许多关注点是一样的，包括像激光一样清楚的对效率的渴望。因此，我们的问题如下。

关键问题：如何开发调度策略

我们该如何开发一个考虑调度策略的基本框架？什么是关键假设？哪些指标非常重要？哪些基本方法已经在早期的系统中使用？

7.1　工作负载假设

探讨可能的策略范围之前，我们先做一些简化假设。这些假设与系统中运行的进程有关，有时候统称为工作负载（workload）。确定工作负载是构建调度策略的关键部分。工作负载了解得越多，你的策略就越优化。

我们这里做的工作负载的假设是不切实际的，但这没问题（目前），因为我们将来会放宽这些假定，并最终开发出我们所谓的……（戏剧性的暂停）……

一个完全可操作的调度准则（a fully-operational scheduling discipline）[①]。

我们对操作系统中运行的进程（有时也叫工作任务）做出如下的假设：

1．每一个工作运行相同的时间。

2．所有的工作同时到达。

3．一旦开始，每个工作保持运行直到完成。

4．所有的工作只是用 CPU（即它们不执行 IO 操作）。

5．每个工作的运行时间是已知的。

我们说这些假设中许多是不现实的，但正如在奥威尔的《动物农场》[045]中一些动物比其他动物更平等，本章中的一些假设比其他假设更不现实。特别是，你会很诧异每一个工作的运行时间是已知的——这会让调度程序无所不知，尽管这样很了不起（也许），但最近不太可能发生。

① 讲这句话的方式和你讲“A fully-operational Death Star.”的方式一样。

7.2 调度指标

除了做出工作负载假设之外，还需要一个东西能让我们比较不同的调度策略：调度指标。指标是我们用来衡量某些东西的东西，在进程调度中，有一些不同的指标是有意义的。

现在，让我们简化一下生活，只用一个指标：周转时间（turnaround time）。任务的周转时间定义为任务完成时间减去任务到达系统的时间。更正式的周转时间定义 $T_{周转时间}$ 是：

$$T_{周转时间} = T_{完成时间} - T_{到达时间} \tag{7.1}$$

因为我们假设所有的任务在同一时间到达，那么 $T_{到达时间} = 0$，因此 $T_{周转时间} = T_{完成时间}$。随着我们放宽上述假设，这个情况将改变。

你应该注意到，周转时间是一个性能（performance）指标，这将是本章的首要关注点。另一个有趣的指标是公平（fairness），比如 Jian's Fairness Index[J91]。性能和公平在调度系统中往往是矛盾的。例如，调度程序可以优化性能，但代价是以阻止一些任务运行，这就降低了公平。这个难题也告诉我们，生活并不总是完美的。

7.3 先进先出（FIFO）

我们可以实现的最基本的算法，被称为先进先出（First In First Out 或 FIFO）调度，有时候也称为先到先服务（First Come First Served 或 FCFS）。

FIFO 有一些积极的特性：它很简单，而且易于实现。而且，对于我们的假设，它的效果很好。

我们一起看一个简单的例子。想象一下，3 个工作 A、B 和 C 在大致相同的时间（$T_{到达时间} = 0$）到达系统。因为 FIFO 必须将某个工作放在前面，所以我们假设当它们都同时到达时，A 比 B 早一点点，然后 B 比 C 早到达一点点。假设每个工作运行 10s。这些工作的平均周转时间（average turnaround time）是多少？

从图 7.1 可以看出，A 在 10s 时完成，B 在 20s 时完成，C 在 30s 时完成。因此，这 3 个任务的平均周转时间就是（10 + 20 + 30）/ 3 = 20。计算周转时间就这么简单。

现在让我们放宽假设。具体来说，让我们放宽假设 1，因此不再认为每个任务的运行时间相同。FIFO 表现如何？你可以构建什么样的工作负载来让 FIFO 表现不好？

（在继续往下读之前，请认真想一下……接着想……想到了么？！）

你现在应该已经弄清楚了，但是以防万一，让我们举个例子来说明不同长度的任务如何导致 FIFO 调度的问题。具体来说，我们再次假设 3 个任务（A、B 和 C），但这次 A 运行 100s，而 B 和 C 运行 10s。

如图 7.2 所示，A 先运行 100s，B 或 C 才有机会运行。因此，系统的平均周转时间是比较高的：令人不快的 110s（（100 + 110 + 120）/ 3 = 110）。

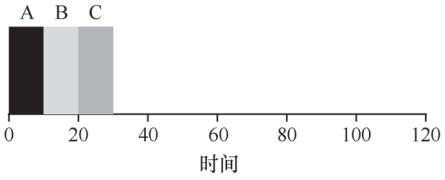


图 7.1 FIFO 的简单例子

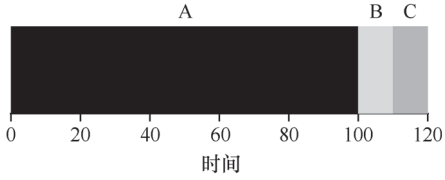


图 7.2 为什么 FIFO 没有那么好

这个问题通常被称为护航效应（convoy effect）[B+79]，一些耗时较少的潜在资源消费者被排在重量级的资源消费者之后。这个调度方案可能让你想起在杂货店只有一个排队队伍的时候，如果看到前面的人装满 3 辆购物车食品并且掏出了支票本，你感觉如何？这会等很长时间[①]。

提示：SJF 原则

最短任务优先代表一个总体调度原则，可以应用于所有系统，只要其中平均客户（或在我们案例中的任务）周转时间很重要。想想你等待的任何队伍：如果有关的机构关心客户满意度，他们可能会考虑到 SJF。例如，大超市通常都有一个“零散购物”的通道，以确保仅购买几件东西的购物者，不会堵在为即将到来的冬天而大量购物以做准备的家庭后面。

那么我们该怎么办？如何开发一种更好的算法来处理任务实际运行时间不一样的场景？先考虑一下，然后继续阅读。

7.4 最短任务优先（SJF）

事实证明，一个非常简单的方法解决了这个问题。实际上这是从运筹学中借鉴的一个想法[C54，PV56]，然后应用到计算机系统的任务调度中。这个新的调度准则被称为最短任务优先（Shortest Job First，SJF），该名称应该很容易记住，因为它完全描述了这个策略：先运行最短的任务，然后是次短的任务，如此下去。

我们用上面的例子，但以 SJF 作为调度策略。图 7.3 展示的是运行 A、B 和 C 的结果。它清楚地说明了为什么在考虑平均周转时间的情况下，SJF 调度策略更好。仅通过在 A 之前运行 B 和 C，SJF 将平均周转时间从 110s 降低到 50s（（10 + 20 + 120）/3 = 50）。

事实上，考虑到所有工作同时到达的假设，我们可以证明 SJF 确实是一个最优（optimal）调度算法。但是，你是在上操作系统课，而不是研究理论，所以，这里允许没有证明。

补充：抢占式调度程序

在过去的批处理计算中，开发了一些非抢占式（non-preemptive）调度程序。这样的系统会将每项工作做完，再考虑是否运行新工作。几乎所有现代化的调度程序都是抢占式的（preemptive），非常愿意停止一个进程以运行另一个进程。这意味着调度程序采用了我们之前学习的机制。特别是调度程序可以进行上下文切换，临时停止一个运行进程，并恢复（或启动）另一个进程。

① 在这种情况下建议采取的措施：要么快速切换到另一个队伍，要么深呼吸并放松。没错，呼气，吸气。这样会变好的，不要担心。

因此，我们找到了一个用 SJF 进行调度的好方法，但是我们的假设仍然是不切实际的。让我们放宽另一个假设。具体来说，我们可以针对假设 2，现在假设工作可以随时到达，而不是同时到达。这导致了什么问题？

（再次停下来想想……你在想吗？加油，你可以做到）

在这里我们可以再次用一个例子来说明问题。现在，假设 A 在 $t = 0$ 时到达，且需要运行 100s。而 B 和 C 在 $t = 10$ 到达，且各需要运行 10s。用纯 SJF，我们可以得到如图 7.4 所示的调度。

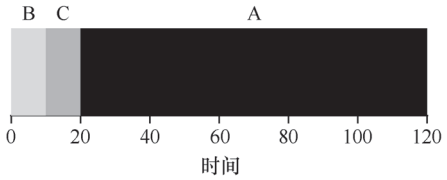


图 7.3 SJF 的简单例子

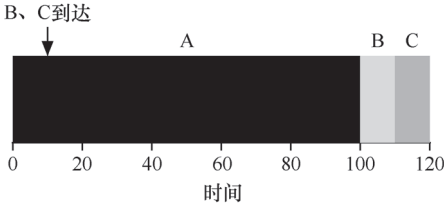


图 7.4 B 和 C 晚到时的 SJF

从图中可以看出，即使 B 和 C 在 A 之后不久到达，它们仍然被迫等到 A 完成，从而遭遇同样的护航问题。这 3 项工作的平均周转时间为 103.33s，即（100+（110−10）+（120−10））/3。

7.5 最短完成时间优先（STCF）

为了解决这个问题，需要放宽假设条件（工作必须保持运行直到完成）。我们还需要调度程序本身的一些机制。你可能已经猜到，鉴于我们先前关于时钟中断和上下文切换的讨论，当 B 和 C 到达时，调度程序当然可以做其他事情：它可以抢占（preempt）工作 A，并决定运行另一个工作，或许稍后继续工作 A。根据我们的定义，SJF 是一种非抢占式（non-preemptive）调度程序，因此存在上述问题。

幸运的是，有一个调度程序完全就是这样做的：向 SJF 添加抢占，称为最短完成时间优先（Shortest Time-to-Completion First，STCF）或抢占式最短作业优先（Preemptive Shortest Job First ，PSJF）调度程序[CK68]。每当新工作进入系统时，它就会确定剩余工作和新工作中，谁的剩余时间最少，然后调度该工作。因此，在我们的例子中，STCF 将抢占 A 并运行 B 和 C 以完成。只有在它们完成后，才能调度 A 的剩余时间。图 7.5 展示了一个例子。

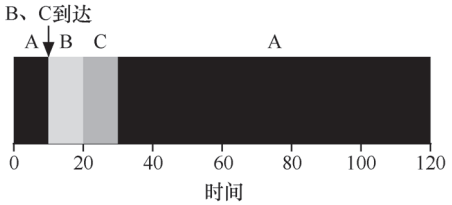


图 7.5 STCF 的简单例子

结果是平均周转时间大大提高：50s（……）。和以前一样，考虑到我们的新假设，STCF 可证明是最优的。考虑到如果所有工作同时到达，SJF 是最优的，那么你应该能够看到 STCF 的最优性是符合直觉的。

7.6 新度量指标：响应时间

因此，如果我们知道任务长度，而且任务只使用 CPU，而我们唯一的衡量是周转时间，STCF 将是一个很好的策略。事实上，对于许多早期批处理系统，这些类型的调度算法有一定的意义。然而，引入分时系统改变了这一切。现在，用户将会坐在终端前面，同时也要求系统的交互性好。因此，一个新的度量标准诞生了：响应时间（response time）。

响应时间定义为从任务到达系统到首次运行的时间。更正式的定义是：

$$T_{\text{响应时间}} = T_{\text{首次运行}} - T_{\text{到达时间}} \tag{7.2}$$

例如，如果我们有上面的调度（A 在时间 0 到达，B 和 C 在时间 10 达到），每个作业的响应时间如下：作业 A 为 0，B 为 0，C 为 10（平均：3.33）。

你可能会想，STCF 和相关方法在响应时间上并不是很好。例如，如果 3 个工作同时到达，第三个工作必须等待前两个工作全部运行后才能运行。这种方法虽然有很好的周转时间，但对于响应时间和交互性是相当糟糕的。假设你在终端前输入，不得不等待 10s 才能看到系统的回应，只是因为其他一些工作已经在你之前被调度：你肯定不太开心。

因此，我们还有另一个问题：如何构建对响应时间敏感的调度程序？

7.7 轮转

为了解决这个问题，我们将介绍一种新的调度算法，通常被称为轮转（Round-Robin，RR）调度[K64]。基本思想很简单：RR 在一个时间片（time slice，有时称为调度量子，scheduling quantum）内运行一个工作，然后切换到运行队列中的下一个任务，而不是运行一个任务直到结束。它反复执行，直到所有任务完成。因此，RR 有时被称为时间切片（time-slicing）。请注意，时间片长度必须是时钟中断周期的倍数。因此，如果时钟中断是每 10ms 中断一次，则时间片可以是 10ms、20ms 或 10ms 的任何其他倍数。

为了更详细地理解 RR，我们来看一个例子。假设 3 个任务 A、B 和 C 在系统中同时到达，并且它们都希望运行 5s。SJF 调度程序必须运行完当前任务才可运行下一个任务（见图 7.6）。相比之下，1s 时间片的 RR 可以快速地循环工作（见图 7.7）。

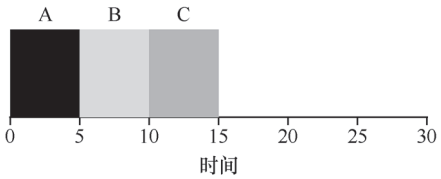


图 7.6 又是 SJF（响应时间不好）

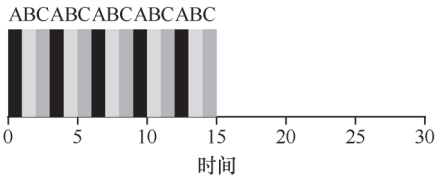


图 7.7 轮转（响应时间好）

RR 的平均响应时间是：（0 + 1 + 2）/3 = 1; SJF 算法平均响应时间是：（0 + 5 + 10）/ 3 = 5。

如你所见，时间片长度对于 RR 是至关重要的。越短，RR 在响应时间上表现越好。然

而，时间片太短是有问题的：突然上下文切换的成本将影响整体性能。因此，系统设计者需要权衡时间片的长度，使其足够长，以便摊销（amortize）上下文切换成本，而又不会使系统不及时响应。

提示：摊销可以减少成本

当系统某些操作有固定成本时，通常会使用摊销技术（amortization）。通过减少成本的频度（即执行较少次的操作），系统的总成本就会降低。例如，如果时间片设置为 10ms，并且上下文切换时间为 1ms，那么浪费大约 10%的时间用于上下文切换。如果要摊销这个成本，可以把时间片增加到 100ms。在这种情况下，不到 1%的时间用于上下文切换，因此时间片带来的成本就被摊销了。

请注意，上下文切换的成本不仅仅来自保存和恢复少量寄存器的操作系统操作。程序运行时，它们在 CPU 高速缓存、TLB、分支预测器和其他片上硬件中建立了大量的状态。切换到另一个工作会导致此状态被刷新，且与当前运行的作业相关的新状态被引入，这可能导致显著的性能成本[MB91]。

如果响应时间是我们的唯一指标，那么带有合理时间片的 RR，就会是非常好的调度程序。但是我们老朋友的周转时间呢？再来看看我们的例子。A、B 和 C，每个运行时间为 5s，同时到达，RR 是具有（长）1s 时间片的调度程序。从图 7.7 可以看出，A 在 13 完成，B 在 14，C 在 15，平均 14。相当可怕！

这并不奇怪，如果周转时间是我们的指标，那么 RR 确实是最糟糕的策略之一。直观地说，这应该是有意义的：RR 所做的正是延伸每个工作，只运行每个工作一小段时间，就转向下一个工作。因为周转时间只关心作业何时完成，RR 几乎是最差的，在很多情况下甚至比简单的 FIFO 更差。

更一般地说，任何公平（fair）的政策（如 RR），即在小规模的时间内将 CPU 均匀分配到活动进程之间，在周转时间这类指标上表现不佳。事实上，这是固有的权衡：如果你愿意不公平，你可以运行较短的工作直到完成，但是要以响应时间为代价。如果你重视公平性，则响应时间会较短，但会以周转时间为代价。这种权衡在系统中很常见。你不能既拥有你的蛋糕，又吃它[①]。

我们开发了两种调度程序。第一种类型（SJF、STCF）优化周转时间，但对响应时间不利。第二种类型（RR）优化响应时间，但对周转时间不利。我们还有两个假设需要放宽：假设 4（作业没有 I/O）和假设 5（每个作业的运行时间是已知的）。接下来我们来解决这些假设。

提示：重叠可以提高利用率

如有可能，重叠（overlap）操作可以最大限度地提高系统的利用率。重叠在许多不同的领域很有用，包括执行磁盘 I/O 或将消息发送到远程机器时。在任何一种情况下，开始操作然后切换到其他工作都是一个好主意，这也提高了系统的整体利用率和效率。

① 这是一个迷惑人的说法，因为它应该是 “你不能保留你的蛋糕，又吃它 ”（这很明显，不是吗？）。令人惊讶的是，这个说法有一个维基百科页面。请自行查阅。

7.8 结合 I/O

首先，我们将放宽假设 4：当然所有程序都执行 I/O。想象一下没有任何输入的程序：每次都会产生相同的输出。设想一个没有输出的程序：它就像谚语所说的森林里倒下的树，没有人看到它。它的运行并不重要。

调度程序显然要在工作发起 I/O 请求时做出决定，因为当前正在运行的作业在 I/O 期间不会使用 CPU，它被阻塞等待 I/O 完成。如果将 I/O 发送到硬盘驱动器，则进程可能会被阻塞几毫秒或更长时间，具体取决于驱动器当前的 I/O 负载。因此，这时调度程序应该在 CPU 上安排另一项工作。

调度程序还必须在 I/O 完成时做出决定。发生这种情况时，会产生中断，操作系统运行并将发出 I/O 的进程从阻塞状态移回就绪状态。当然，它甚至可以决定在那个时候运行该项工作。操作系统应该如何处理每项工作？

为了更好地理解这个问题，让我们假设有两项工作 A 和 B，每项工作需要 50ms 的 CPU 时间。但是，有一个明显的区别：A 运行 10ms，然后发出 I/O 请求（假设 I/O 每个都需要 10ms），而 B 只是使用 CPU 50ms，不执行 I/O。调度程序先运行 A，然后运行 B（见图 7.8）。

假设我们正在尝试构建 STCF 调度程序。这样的调度程序应该如何考虑到这样的事实，即 A 分解成 5 个 10ms 子工作，而 B 仅仅是单个 50ms CPU 需求？显然，仅仅运行一个工作，然后运行另一个工作，而不考虑如何考虑 I/O 是没有意义的。

一种常见的方法是将 A 的每个 10ms 的子工作视为一项独立的工作。因此，当系统启动时，它的选择是调度 10ms 的 A，还是 50ms 的 B。对于 STCF，选择是明确的：选择较短的一个，在这种情况下是 A。然后，A 的工作已完成，只剩下 B，并开始运行。然后提交 A 的一个新子工作，它抢占 B 并运行 10ms。这样做可以实现重叠（overlap），一个进程在等待另一个进程的 I/O 完成时使用 CPU，系统因此得到更好的利用（见图 7.9）。

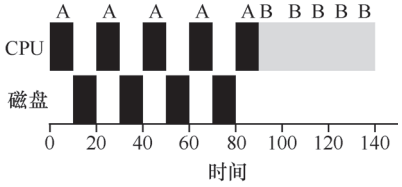


图 7.8 资源的糟糕使用

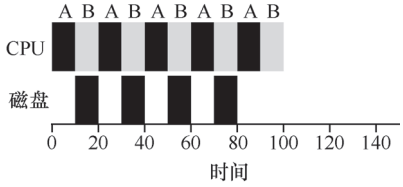


图 7.9 重叠可以更好地使用资源

这样我们就看到了调度程序可能如何结合 I/O。通过将每个 CPU 突发作为一项工作，调度程序确保“交互”的进程经常运行。当这些交互式作业正在执行 I/O 时，其他 CPU 密集型作业将运行，从而更好地利用处理器。

7.9 无法预知

有了应对 I/O 的基本方法，我们来到最后的假设：调度程序知道每个工作的长度。如前

所述，这可能是可以做出的最糟糕的假设。事实上，在一个通用的操作系统中（比如我们所关心的操作系统），操作系统通常对每个作业的长度知之甚少。因此，我们如何建立一个没有这种先验知识的 SJF/STCF？更进一步，我们如何能够将已经看到的一些想法与 RR 调度程序结合起来，以便响应时间也变得相当不错？

7.10 小结

我们介绍了调度的基本思想，并开发了两类方法。第一类是运行最短的工作，从而优化周转时间。第二类是交替运行所有工作，从而优化响应时间。但很难做到“鱼与熊掌兼得”，这是系统中常见的、固有的折中。我们也看到了如何将 I/O 结合到场景中，但仍未解决操作系统根本无法看到未来的问题。稍后，我们将看到如何通过构建一个调度程序，利用最近的历史预测未来，从而解决这个问题。这个调度程序称为多级反馈队列，是第 8 章的主题。

参考资料

[B+79]“The Convoy Phenomenon”
M. Blasgen, J. Gray, M. Mitoma, T. Price
ACM Operating Systems Review, 13:2, April 1979
也许是第一次在数据库和操作系统中提到护航效应。

[C54]“Priority Assignment in Waiting Line Problems”
A. Cobham
Journal of Operations Research, 2:70, pages 70–76, 1954
关于使用 SJF 方法调度修理机器的开创性论文。

[K64]“Analysis of a Time-Shared Processor”Leonard Kleinrock
Naval Research Logistics Quarterly, 11:1, pages 59–73, March 1964
该文可能是第一次提到轮转调度算法，当然是调度时分共享系统方法的最早分析之一。

[CK68]“Computer Scheduling Methods and their Countermeasures”Edward G. Coffman and Leonard Kleinrock
AFIPS ’68 (Spring), April 1968
一篇很好的早期文章，其中还分析了一些基本调度准则。

[J91]“The Art of Computer Systems Performance Analysis:
Techniques for Experimental Design, Measurement, Simulation, and Modeling”
R. Jain
Interscience, New York, April 1991
计算机系统测量的标准教科书。当然，这对你的库是一个很好的参考。

[PV56]“Machine Repair as a Priority Waiting-Line Problem”Thomas E. Phipps Jr. and W. R. Van Voorhis
Operations Research, 4:1, pages 76–86, February 1956
有关后续工作，概括了来自 Cobham 最初工作的机器修理 SJF 方法，也假定了在这样的环境中 STCF 方法的效用。具体来说，“有一些类型的修理工作，……涉及很多拆卸，地上满是螺母和螺栓，一旦进行就不应该中断。在其他情况下，如果有一个或多个短工作可做，继续做长工作是不可取的（第 81 页）。”

[MB91]“The effect of context switches on cache performance”Jeffrey C. Mogul and Anita Borg
ASPLOS, 1991
关于缓存性能如何受上下文切换影响的一项很好的研究。在今天的系统中问题比较小，如今处理器每秒钟发出数十亿条指令，但上下文切换仍发生在毫秒的时间级别。

作业

scheduler.py 这个程序允许你查看不同调度程序在调度指标（如响应时间、周转时间和总等待时间）下的执行情况。详情请参阅 README 文件。

问题

1．使用 SJF 和 FIFO 调度程序运行长度为 200 的 3 个作业时，计算响应时间和周转时间。

2．现在做同样的事情，但有不同长度的作业，即 100、200 和 300。

3．现在做同样的事情，但采用 RR 调度程序，时间片为 1。

4．对于什么类型的工作负载，SJF 提供与 FIFO 相同的周转时间？

5．对于什么类型的工作负载和量子长度，SJF 与 RR 提供相同的响应时间？

6．随着工作长度的增加，SJF 的响应时间会怎样？你能使用模拟程序来展示趋势吗？

7．随着量子长度的增加，RR 的响应时间会怎样？你能写出一个方程，计算给定 N 个工作时，最坏情况的响应时间吗？

第 8 章　调度：多级反馈队列

本章将介绍一种著名的调度方法——多级反馈队列（Multi-level Feedback Queue，MLFQ）。1962 年，Corbato 首次提出多级反馈队列[C+62]，应用于兼容时分共享系统（CTSS）。Corbato 因在 CTSS 中的贡献和后来在 Multics 中的贡献，获得了 ACM 颁发的图灵奖（Turing Award）。该调度程序经过多年的一系列优化，出现在许多现代操作系统中。

多级反馈队列需要解决两方面的问题。首先，它要优化周转时间。在第 7 章中我们看到，这通过先执行短工作来实现。然而，操作系统通常不知道工作要运行多久，而这又是 SJF（或 STCF）等算法所必需的。其次，MLFQ 希望给交互用户（如用户坐在屏幕前，等着进程结束）很好的交互体验，因此需要降低响应时间。然而，像轮转这样的算法虽然降低了响应时间，周转时间却很差。所以这里的问题是：通常我们对进程一无所知，应该如何构建调度程序来实现这些目标？调度程序如何在运行过程中学习进程的特征，从而做出更好的调度决策？

关键问题：没有完备的知识如何调度？

没有工作长度的先验（priori）知识，如何设计一个能同时减少响应时间和周转时间的调度程序？

提示：从历史中学习

多级反馈队列是用历史经验预测未来的一个典型的例子，操作系统中有很多地方采用了这种技术（同样存在于计算机科学领域的很多其他地方，比如硬件的分支预测及缓存算法）。如果工作有明显的阶段性行为，因此可以预测，那么这种方式会很有效。当然，必须十分小心地使用这种技术，因为它可能出错，让系统做出比一无所知的时候更糟的决定。

8.1　MLFQ：基本规则

为了构建这样的调度程序，本章将介绍多级消息队列背后的基本算法。虽然它有许多不同的实现[E95]，但大多数方法是类似的。

MLFQ 中有许多独立的队列（queue），每个队列有不同的优先级（priority level）。任何时刻，一个工作只能存在于一个队列中。MLFQ 总是优先执行较高优先级的工作（即在较高级队列中的工作）。

当然，每个队列中可能会有多个工作，因此具有同样的优先级。在这种情况下，我们就对这些工作采用轮转调度。

因此，MLFQ 调度策略的关键在于如何设置优先级。MLFQ 没有为每个工作指定不变

的优先情绪而已，而是根据观察到的行为调整它的优先级。例如，如果一个工作不断放弃 CPU 去等待键盘输入，这是交互型进程的可能行为，MLFQ 因此会让它保持高优先级。相反，如果一个工作长时间地占用 CPU，MLFQ 会降低其优先级。通过这种方式，MLFQ 在进程运行过程中学习其行为，从而利用工作的历史来预测它未来的行为。

至此，我们得到了 MLFQ 的两条基本规则。

- **规则 1**：如果 A 的优先级 ＞B 的优先级，运行 A（不运行 B）。
- **规则 2**：如果 A 的优先级 ＝B 的优先级，轮转运行 A 和 B。

如果要在某个特定时刻展示队列，可能会看到如下内容（见图 8.1）。图 8.1 中，最高优先级有两个工作（A 和 B），工作 C 位于中等优先级，而 D 的优先级最低。按刚才介绍的基本规则，由于 A 和 B 有最高优先级，调度程序将交替的调度他们，可怜的 C 和 D 永远都没有机会运行，太气人了！

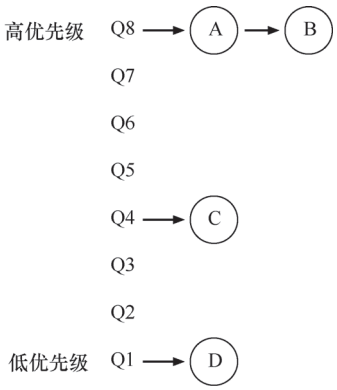


图 8.1 MLFQ 的例子

当然，这只是展示了一些队列的静态快照，并不能让你真正明白 MLFQ 的工作原理。我们需要理解工作的优先级如何随时间变化。初次拿起本书阅读一章的人可能会吃惊，这正是我们接下来要做的事。

8.2 尝试 1：如何改变优先级

我们必须决定，在一个工作的生命周期中，MLFQ 如何改变其优先级（在哪个队列中）。要做到这一点，我们必须记得工作负载：既有运行时间很短、频繁放弃 CPU 的交互型工作，也有需要很多 CPU 时间、响应时间却不重要的长时间计算密集型工作。下面是我们第一次尝试优先级调整算法。

- **规则 3**：工作进入系统时，放在最高优先级（最上层队列）。
- **规则 4a**：工作用完整个时间片后，降低其优先级（移入下一个队列）。
- **规则 4b**：如果工作在其时间片以内主动释放 CPU，则优先级不变。

实例 1：单个长工作

我们来看一些例子。首先，如果系统中有一个需要长时间运行的工作，看看会发生什么。图 8.2 展示了在一个有 3 个队列的调度程序中，随着时间的推移，这个工作的运行情况。

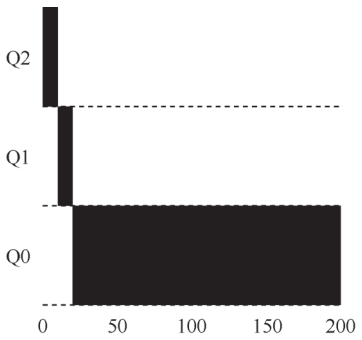


图 8.2 长时间工作随时间的变化

从这个例子可以看出，该工作首先进入最高优先级（Q2）。执行一个 10ms 的时间片后，调度程序将工作的优先级减 1，因此进入 Q1。在 Q1 执行一个时间片后，最终降低优先级进入系统的最低优先级（Q0），一直留在那里。相当简单，不是吗？

实例 2：来了一个短工作

再看一个较复杂的例子，看看 MLFQ 如何近似 SJF。在这个例子中，有两个工作：A 是一个长时间运行的 CPU 密集型工作，B 是一个运行时间很短的交互型工作。假设 A 执行一段时间后 B 到达。会发生什么呢？对 B 来说，MLFQ 会近似于 SJF 吗？

图 8.3 展示了这种场景的结果。A（用黑色表示）在最低优先级队列执行（长时间运行的 CPU 密集型工作都这样）。B（用灰色表示）在时间 T=100 时到达，并被加入最高优先级队列。由于它的运行时间很短（只有 20ms），经过两个时间片，在被移入最低优先级队列之前，B 执行完毕。然后 A 继续运行（在低优先级）。

通过这个例子，你大概可以体会到这个算法的一个主要目标：如果不知道工作是短工作还是长工作，那么就在开始的时候假设其是短工作，并赋予最高优先级。如果确实是短工作，则很快会执行完毕，否则将被慢慢移入低优先级队列，而这时该工作也被认为是长工作了。通过这种方式，MLFQ 近似于 SJF。

实例 3：如果有 I/O 呢

看一个有 I/O 的例子。根据上述规则 4b，如果进程在时间片用完之前主动放弃 CPU，则保持它的优先级不变。这条规则的意图很简单：假设交互型工作中有大量的 I/O 操作（比如等待用户的键盘或鼠标输入），它会在时间片用完之前放弃 CPU。在这种情况下，我们不想处罚它，只是保持它的优先级不变。

图 8.4 展示了这个运行过程，交互型工作 B（用灰色表示）每执行 1ms 便需要进行 I/O 操作，它与长时间运行的工作 A（用黑色表示）竞争 CPU。MLFQ 算法保持 B 在最高优先级，因为 B 总是让出 CPU。如果 B 是交互型工作，MLFQ 就进一步实现了它的目标，让交互型工作快速运行。

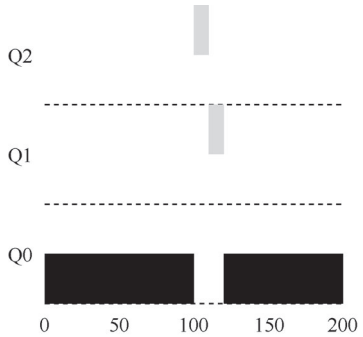


图 8.3 一个交互型工作

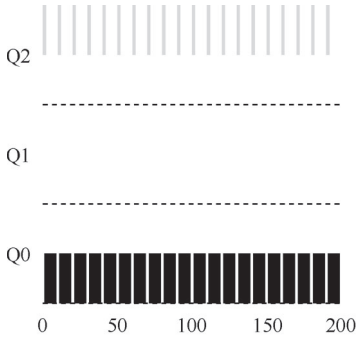


图 8.4 混合 I/O 密集型和 CPU 密集型工作负载

当前 MLFQ 的一些问题

至此，我们有了基本的 MLFQ。它看起来似乎相当不错，长工作之间可以公平地分享 CPU，又能给短工作或交互型工作很好的响应时间。然而，这种算法有一些非常严重的缺点。

你能想到吗？

（暂停一下，尽量让脑筋转转弯）

首先，会有饥饿（starvation）问题。如果系统有“太多”交互型工作，就会不断占用 CPU，导致长工作永远无法得到 CPU（它们饿死了）。即使在这种情况下，我们希望这些长工作也能有所进展。

其次，聪明的用户会重写程序，愚弄调度程序（game the scheduler）。愚弄调度程序指的是用一些卑鄙的手段欺骗调度程序，让它给你远超公平的资源。上述算法对如下的攻击束手无策：进程在时间片用完之前，调用一个 I/O 操作（比如访问一个无关的文件），从而主动释放 CPU。如此便可以保持在高优先级，占用更多的 CPU 时间。做得好时（比如，每运行 99%的时间片时间就主动放弃一次 CPU），工作可以几乎独占 CPU。

最后，一个程序可能在不同时间表现不同。一个计算密集的进程可能在某段时间表现为一个交互型的进程。用我们目前的方法，它不会享受系统中其他交互型工作的待遇。

8.3 尝试 2：提升优先级

让我们试着改变之前的规则，看能否避免饥饿问题。要让 CPU 密集型工作也能取得一些进展（即使不多），我们能做些什么？

一个简单的思路是周期性地提升（boost）所有工作的优先级。可以有很多方法做到，但我们就用最简单的：将所有工作扔到最高优先级队列。于是有了如下的新规则。

- **规则 5**：经过一段时间 *S*，就将系统中所有工作重新加入最高优先级队列。

新规则一下解决了两个问题。首先，进程不会饿死——在最高优先级队列中，它会以轮转的方式，与其他高优先级工作分享 CPU，从而最终获得执行。其次，如果一个 CPU 密集型工作变成了交互型，当它优先级提升时，调度程序会正确对待它。

我们来看一个例子。在这种场景下，我们展示长工作与两个交互型短工作竞争 CPU 时的行为。图 8.5 包含两张图。左边没有优先级提升，长工作在两个短工作到达后被饿死。右边每 50ms 就有一次优先级提升（这里只是举例，这个值可能过小），因此至少保证长工作会有一些进展，每过 50ms 就被提升到最高优先级，从而定期获得执行。

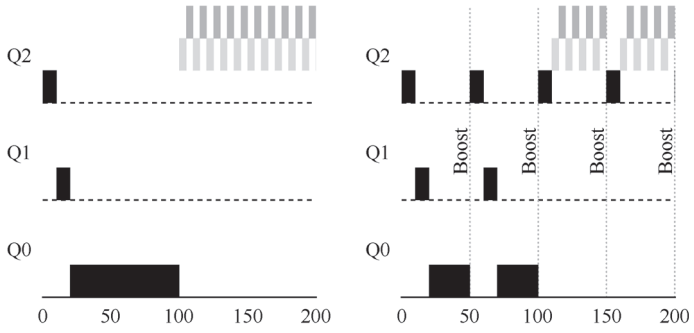


图 8.5 不采用优先级提升（左）和采用（右）

当然，添加时间段 *S* 导致了明显的问题：*S* 的值应该如何设置？德高望重的系统研究员

John Ousterhout[O11]曾将这种值称为“巫毒常量（voo-doo constant）”，因为似乎需要一些黑魔法才能正确设置。如果 *S* 设置得太高，长工作会饥饿；如果设置得太低，交互型工作又得不到合适的 CPU 时间比例。

8.4 尝试 3：更好的计时方式

现在还有一个问题要解决：如何阻止调度程序被愚弄？可以看出，这里的元凶是规则 4a 和 4b，导致工作在时间片以内释放 CPU，就保留它的优先级。那么应该怎么做？

这里的解决方案，是为 MLFQ 的每层队列提供更完善的 CPU 计时方式（accounting）。调度程序应该记录一个进程在某一层中消耗的总时间，而不是在调度时重新计时。只要进程用完了自己的配额，就将它降到低一优先级的队列中去。不论它是一次用完的，还是拆成很多次用完。因此，我们重写规则 4a 和 4b。

- **规则 4：**一旦工作用完了其在某一层中的时间配额（无论中间主动放弃了多少次 CPU），就降低其优先级（移入低一级队列）。

来看一个例子。图 8.6 对比了在规则 4a、4b 的策略下（左图），以及在新的规则 4（右图）的策略下，同样试图愚弄调度程序的进程的表现。没有规则 4 的保护时，进程可以在每个时间片结束前发起一次 I/O 操作，从而垄断 CPU 时间。有了这样的保护后，不论进程的 I/O 行为如何，都会慢慢地降低优先级，因而无法获得超过公平的 CPU 时间比例。

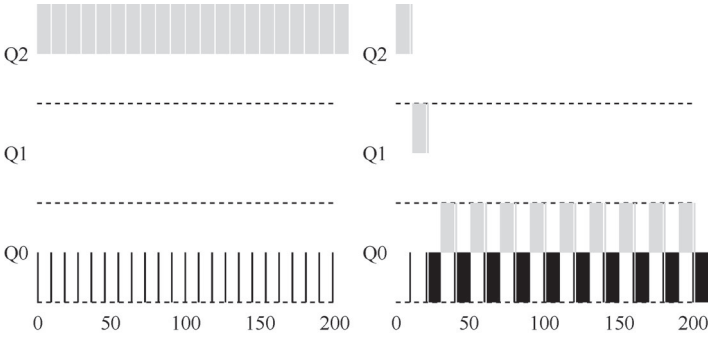


图 8.6 不采用愚弄反制（左）和采用（右）

8.5 MLFQ 调优及其他问题

关于 MLFQ 调度算法还有一些问题。其中一个大问题是如何配置一个调度程序，例如，配置多少队列？每一层队列的时间片配置多大？为了避免饥饿问题以及进程行为改变，应该多久提升一次进程的优先级？这些问题都没有显而易见的答案，因此只有利用对工作负载的经验，以及后续对调度程序的调优，才会导致令人满意的平衡。

例如，大多数的 MLFQ 变体都支持不同队列可变的时间片长度。高优先级队列通常只有较短的时间片（比如 10ms 或者更少），因而这一层的交互工作可以更快地切换。相反，

低优先级队列中更多的是 CPU 密集型工作，配置更长的时间片会取得更好的效果。图 8.7 展示了一个例子，两个长工作在高优先级队列执行 10ms，中间队列执行 20ms，最后在最低优先级队列执行 40ms。

提示：避免巫毒常量（Ousterhout 定律）

尽可能避免巫毒常量是个好主意。然而，从上面的例子可以看出，这通常很难。当然，我们也可以让系统自己去学习一个很优化的值，但这同样也不容易。因此，通常我们会有一个写满各种参数值默认值的配置文件，使得系统管理员可以方便地进行修改调整。然而，大多数使用者并不会去修改这些默认值，这时就寄希望于默认值合适了。这个提示是由资深的 OS 教授 John Ousterhout 提出的，因此称为 Ousterhout 定律（Ousterhout's Law）。

Solaris 的 MLFQ 实现（时分调度类 TS）很容易配置。它提供了一组表来决定进程在其生命周期中如何调整优先级，每层的时间片多大，以及多久提升一个工作的优先级[AD00]。管理员可以通过这些表，让调度程序的行为方式不同。该表默认有 60 层队列，时间片长度从 20ms（最高优先级），到几百 ms（最低优先级），每一秒左右提升一次进程的优先级。

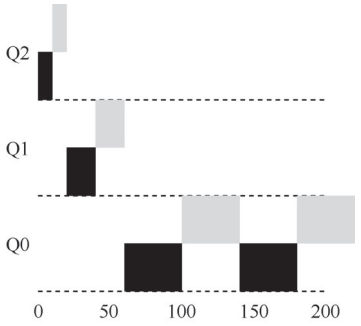


图 8.7 优先级越低，时间片越长

其他一些 MLFQ 调度程序没用表，甚至没用本章中讲到的规则，有些采用数学公式来调整优先级。例如，FreeBSD 调度程序（4.3 版本），会基于当前进程使用了多少 CPU，通过公式计算某个工作的当前优先级[LM+89]。另外，使用量会随时间衰减，这提供了期望的优先级提升，但与这里描述方式不同。阅读 Epema 的论文，他漂亮地概括了这种使用量衰减（decay-usage）算法及其特征[E95]。

最后，许多调度程序有一些我们没有提到的特征。例如，有些调度程序将最高优先级队列留给操作系统使用，因此通常的用户工作是无法得到系统的最高优先级的。有些系统允许用户给出优先级设置的建议（advice），比如通过命令行工具 nice，可以增加或降低工作的优先级（稍微），从而增加或降低它在某个时刻运行的机会。更多信息请查看 man 手册。

8.6 MLFQ：小结

本章介绍了一种调度方式，名为多级反馈队列（MLFQ）。你应该已经知道它为什么叫这个名字——它有多级队列，并利用反馈信息决定某个工作的优先级。以史为鉴：关注进程的一贯表现，然后区别对待。

提示：尽可能多地使用建议

操作系统很少知道什么策略对系统中的单个进程和每个进程算是好的，因此提供接口并允许用户或管理员给操作系统一些提示（hint）常常很有用。我们通常称之为建议（advice），因为操作系统不一定要关注它，但是可能会将建议考虑在内，以便做出更好的决定。这种用户建议的方式在操作系统中的各个领域经常十分有用，包括调度程序（通过 nice）、内存管理（madvise），以及文件系统（通知预取和缓存[P+95]）。

本章包含了一组优化的 MLFQ 规则。为了方便查阅，我们重新列在这里。

- **规则 1**：如果 A 的优先级 ＞B 的优先级，运行 A（不运行 B）。
- **规则 2**：如果 A 的优先级 ＝B 的优先级，轮转运行 A 和 B。
- **规则 3**：工作进入系统时，放在最高优先级（最上层队列）。
- **规则 4**：一旦工作用完了其在某一层中的时间配额（无论中间主动放弃了多少次 CPU），就降低其优先级（移入低一级队列）。
- **规则 5**：经过一段时间 *S*，就将系统中所有工作重新加入最高优先级队列。

MLFQ 有趣的原因是：它不需要对工作的运行方式有先验知识，而是通过观察工作的运行来给出对应的优先级。通过这种方式，MLFQ 可以同时满足各种工作的需求：对于短时间运行的交互型工作，获得类似于 SJF/STCF 的很好的全局性能，同时对长时间运行的 CPU 密集型负载也可以公平地、不断地稳步向前。因此，许多系统使用某种类型的 MLFQ 作为自己的基础调度程序，包括类 BSD UNIX 系统[LM+89，B86]、Solaris[M06]以及 Windows NT 和其后的 Window 系列操作系统。

参考资料

[AD00]“Multilevel Feedback Queue Scheduling in Solaris”Andrea Arpaci-Dusseau
本书的一位作者就 Solaris 调度程序的细节做了一些简短的说明。我们这里的描述可能有失偏颇，但这些讲义还是不错的。

[B86]“The Design of the UNIX Operating System”
M.J. Bach
Prentice-Hall, 1986
关于如何构建真正的 UNIX 操作系统的经典老书之一。对内核黑客来说，这是必读内容。

[C+62]“An Experimental Time-Sharing System”
F. J. Corbato, M. M. Daggett, R. C. Daley IFIPS 1962
有点难读，但这是多级反馈调度中许多首创想法的来源。其中大部分后来进入了 Multics，人们可以争辩说它是有史以来有影响力的操作系统。

[CS97]“Inside Windows NT”
Helen Custer and David A. Solomon Microsoft Press, 1997
如果你想了解 UNIX 以外的东西，来读 NT 书吧！当然，你为什么会想？好吧，我们在开玩笑吧。说不定有一天你会为微软工作。

[E95]“An Analysis of Decay-Usage Scheduling in Multiprocessors”
D.H.J.　Epema SIGMETRICS ’95
一篇关于 20 世纪 90 年代中期调度技术发展状况的优秀论文，概述了使用量衰减调度程序背后的基本方法。

[LM+89]“The Design and Implementation of the 4.3BSD UNIX Operating System”

S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman Addison-Wesley, 1989
另一本操作系统经典图书，由 BSD 背后的 4 个主要人员编写。本书后面的版本虽然更新了，但感觉不如这一版好。

[M06]“Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture”Richard McDougall
Prentice-Hall, 2006
一本关于 Solaris 及其工作原理的好书。

[O11]“John Ousterhout’s Home Page”John Ousterhout
著名的 Ousterhout 教授的主页。本书的两位合著者一起在研究生院学习 Ousterhout 的研究生操作系统课程。事实上，这是两位合著者相互认识的地方，最终他们结了婚、生了孩子，还合著了这本书。因此，你真的可以责怪 Ousterhout，让你陷入这场混乱。

[P+95]“Informed Prefetching and Caching”
R.H．Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka SOSP ’95
关于文件系统中一些非常酷的创意的有趣文章，其中包括应用程序如何向操作系统提供关于它正在访问哪些文件，以及它计划如何访问这些文件的建议。

作业

程序 mlfq.py 允许你查看本章介绍的 MLFQ 调度程序的行为。详情请参阅 README 文件。

问题

1．只用两个工作和两个队列运行几个随机生成的问题。针对每个工作计算 MLFQ 的执行记录。限制每项作业的长度并关闭 I/O，让你的生活更轻松。

2．如何运行调度程序来重现本章中的每个实例？

3．将如何配置调度程序参数，像轮转调度程序那样工作？

4．设计两个工作的负载和调度程序参数，以便一个工作利用较早的规则 4a 和 4b（用 -S 标志打开）来“愚弄”调度程序，在特定的时间间隔内获得 99%的 CPU。

5．给定一个系统，其最高队列中的时间片长度为 10ms，你需要如何频繁地将工作推回到最高优先级级别（带有-B 标志），以保证一个长时间运行（并可能饥饿）的工作得到至少 5%的 CPU？

6．调度中有一个问题，即刚完成 I/O 的作业添加在队列的哪一端。-I 标志改变了这个调度模拟器的这方面行为。尝试一些工作负载，看看你是否能看到这个标志的效果。

第 9 章　调度：比例份额

在本章中，我们来看一个不同类型的调度程序——比例份额（proportional-share）调度程序，有时也称为公平份额（fair-share）调度程序。比例份额算法基于一个简单的想法：调度程序的最终目标，是确保每个工作获得一定比例的 CPU 时间，而不是优化周转时间和响应时间。

比例份额调度程序有一个非常优秀的现代例子，由 Waldspurger 和 Weihl 发现，名为彩票调度（lottery scheduling） [WW94]。但这个想法其实出现得更早[KL88]。基本思想很简单：每隔一段时间，都会举行一次彩票抽奖，以确定接下来应该运行哪个进程。越是应该频繁运行的进程，越是应该拥有更多地赢得彩票的机会。很简单吧？现在，谈谈细节！但还是先看看下面的关键问题。

关键问题：如何按比例分配 CPU

如何设计调度程序来按比例分配 CPU？其关键的机制是什么？效率如何？

9.1　基本概念：彩票数表示份额

彩票调度背后是一个非常基本的概念：彩票数（ticket）代表了进程（或用户或其他）占有某个资源的份额。一个进程拥有的彩票数占总彩票数的百分比，就是它占有资源的份额。

下面来看一个例子。假设有两个进程 A 和 B，A 拥有 75 张彩票，B 拥有 25 张。因此我们希望 A 占用 75%的 CPU 时间，而 B 占用 25%。

通过不断定时地（比如，每个时间片）抽取彩票，彩票调度从概率上（但不是确定的）获得这种份额比例。抽取彩票的过程很简单：调度程序知道总共的彩票数（在我们的例子中，有 100 张）。调度程序抽取中奖彩票，这是从 0 和 99[①]之间的一个数，拥有这个数对应的彩票的进程中奖。假设进程 A 拥有 0 到 74 共 75 张彩票，进程 B 拥有 75 到 99 的 25 张，中奖的彩票就决定了运行 A 或 B。调度程序然后加载中奖进程的状态，并运行它。

提示：利用随机性

彩票调度最精彩的地方在于利用了随机性（randomness）。当你需要做出决定时，采用随机的方式常常是既可靠又简单的选择。

随机方法相对于传统的决策方式，至少有 3 点优势。第一，随机方法常常可以避免奇怪的边角情况，

① 计算机科学家总是从 0 开始计数。对于非计算机类型的人来说，这非常奇怪，所以著名人士不得不撰文说明这样做的原因[D82]。

较传统的算法可能在处理这些情况时遇到麻烦。例如 LRU 替换策略（稍后会在虚拟内存的章节详细介绍）。虽然 LRU 通常是很好的替换算法，但在有重复序列的负载时表现非常差。但随机方法就没有这种最差情况。

第二，随机方法很轻量，几乎不需要记录任何状态。在传统的公平份额调度算法中，记录每个进程已经获得了多少的 CPU 时间，需要对每个进程计时，这必须在每次运行结束后更新。而采用随机方式后每个进程只需要非常少的状态（即每个进程拥有的彩票号码）。

第三，随机方法很快。只要能很快地产生随机数，做出决策就很快。因此，随机方式在对运行速度要求高的场景非常适用。当然，越是需要快的计算速度，随机就会越倾向于伪随机。

下面是彩票调度程序输出的中奖彩票：

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 49
```

下面是对应的调度结果:

```
A   A A     A A A A A A     A     A A A A A A
  B       B                 B     B
```

从这个例子中可以看出，彩票调度中利用了随机性，这导致了从概率上满足期望的比例，但并不能确保。在上面的例子中，工作 B 运行了 20 个时间片中的 4 个，只是占了 20%，而不是期望的 25%。但是，这两个工作运行得时间越长，它们得到的 CPU 时间比例就会越接近期望。

提示：用彩票来表示份额

彩票（步长）调度的设计中，最强大（且最基本）的机制是彩票。在这些例子中，彩票用于表示一个进程占有 CPU 的份额，但也可以用在更多的地方。比如在虚拟机管理程序的虚存管理的最新研究工作中，Waldspurger 提出了用彩票来表示用户占用操作系统内存份额的方法[W02]。因此，如果你需要通过什么机制来表示所有权比例，这个概念可能就是彩票。

9.2 彩票机制

彩票调度还提供了一些机制，以不同且有效的方式来调度彩票。一种方式是利用彩票货币（ticket currency）的概念。这种方式允许拥有一组彩票的用户以他们喜欢的某种货币，将彩票分给自己的不同工作。之后操作系统再自动将这种货币兑换为正确的全局彩票。

比如，假设用户 A 和用户 B 每人拥有 100 张彩票。用户 A 有两个工作 A1 和 A2，他以自己的货币，给每个工作 500 张彩票（共 1000 张）。用户 B 只运行一个工作，给它 10 张彩票（总共 10 张）。操作系统将进行兑换，将 A1 和 A2 拥有的 A 的货币 500 张，兑换成全局货币 50 张。类似地，兑换给 B1 的 10 张彩票兑换成 100 张。然后会对全局彩票货币（共 200 张）举行抽奖，决定哪个工作运行。

```
User A -> 500 (A's currency) to A1 -> 50 (global currency)
       -> 500 (A's currency) to A2 -> 50 (global currency)
User B -> 10 (B's currency) to B1 -> 100 (global currency)
```

另一个有用的机制是彩票转让（ticket transfer）。通过转让，一个进程可以临时将自己的彩票交给另一个进程。这种机制在客户端/服务端交互的场景中尤其有用，在这种场景中，客户端进程向服务端发送消息，请求其按自己的需求执行工作，为了加速服务端的执行，客户端可以将自己的彩票转让给服务端，从而尽可能加速服务端执行自己请求的速度。服务端执行结束后会将这部分彩票归还给客户端。

最后，彩票通胀（ticket inflation）有时也很有用。利用通胀，一个进程可以临时提升或降低自己拥有的彩票数量。当然在竞争环境中，进程之间互相不信任，这种机制就没什么意义。一个贪婪的进程可能给自己非常多的彩票，从而接管机器。但是，通胀可以用于进程之间相互信任的环境。在这种情况下，如果一个进程知道它需要更多 CPU 时间，就可以增加自己的彩票，从而将自己的需求告知操作系统，这一切不需要与任何其他进程通信。

9.3 实现

彩票调度中最不可思议的，或许就是实现简单。只需要一个不错的随机数生成器来选择中奖彩票和一个记录系统中所有进程的数据结构（一个列表），以及所有彩票的总数。

假定我们用列表记录进程。下面的例子中有 A、B、C 这 3 个进程，每个进程有一定数量的彩票。

在做出调度决策之前，首先要从彩票总数 400 中选择一个随机数（中奖号码）[①]。假设选择了 300。然后，遍历链表，用一个简单的计数器帮助我们找到中奖者（见图 9.1）。

```
1   // counter: used to track if we've found the winner yet
2   int counter = 0;
3
4   // winner: use some call to a random number generator to
5   //         get a value, between 0 and the total # of tickets
6   int winner = getrandom(0, totaltickets);
7
8   // current: use this to walk through the list of jobs
9   node_t *current = head;
10
11  // loop until the sum of ticket values is > the winner
12  while (current) {
13      counter = counter + current->tickets;
14      if (counter > winner)
15          break; // found the winner
16      current = current->next;
17  }
18  // 'current' is the winner: schedule it...
```

图 9.1 彩票调度决定代码

① 令人惊讶的是，正如 Björn Lindberg 所指出的那样，要做对，这可能是一个挑战。

这段代码从前向后遍历进程列表，将每张票的值加到 counter 上，直到值超过 winner。这时，当前的列表元素所对应的进程就是中奖者。在我们的例子中，中奖彩票是 300。首先，计 A 的票后，counter 增加到 100。因为 100 小于 300，继续遍历。然后 counter 会增加到 150（B 的彩票），仍然小于 300，继续遍历。最后，counter 增加到 400（显然大于 300），因此退出遍历，current 指向 C（中奖者）。

要让这个过程更有效率，建议将列表项按照彩票数递减排序。这个顺序并不会影响算法的正确性，但能保证用最小的迭代次数找到需要的节点，尤其当大多数彩票被少数进程掌握时。

9.4 一个例子

为了更好地理解彩票调度的运行过程，我们现在简单研究一下两个互相竞争工作的完成时间，每个工作都有相同数目的 100 张彩票，以及相同的运行时间 *R*（稍后会改变）。

这种情况下，我们希望两个工作在大约同时完成，但由于彩票调度算法的随机性，有时一个工作会先于另一个完成。为了量化这种区别，我们定义了一个简单的不公平指标 *U*（unfairness metric），将两个工作完成时刻相除得到 *U* 的值。比如，运行时间 *R* 为 10，第一个工作在时刻 10 完成，另一个在 20，*U*=10/20=0.5。如果两个工作几乎同时完成，*U* 的值将很接近于 1。在这种情况下，我们的目标是：完美的公平调度程序可以做到 *U*=1。

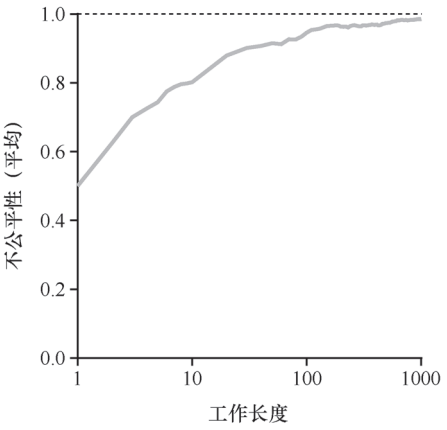


图 9.2 彩票公平性研究

图 9.2 展示了当两个工作的运行时间从 1 到 1000 变化时，30 次试验的平均 *U* 值（利用本章末尾的模拟器产生的结果）。可以看出，当工作执行时间很短时，平均不公平度非常糟糕。只有当工作执行非常多的时间片时，彩票调度算法才能得到期望的结果。

9.5 如何分配彩票

关于彩票调度，还有一个问题没有提到，那就是如何为工作分配彩票？这是一个非常棘手的问题，系统的运行严重依赖于彩票的分配。假设用户自己知道如何分配，因此可以给每个用户一定量的彩票，由用户按照需要自主分配给自己的工作。然而这种方案似乎什么也没有解决——还是没有给出具体的分配策略。因此对于给定的一组工作，彩票分配的问题依然没有最佳答案。

9.6 为什么不是确定的

你可能还想知道，究竟为什么要利用随机性？从上面的内容可以看出，虽然随机方式可以使得调度程序的实现简单（且大致正确），但偶尔并不能产生正确的比例，尤其在工作运行时间很短的情况下。由于这个原因，Waldspurger 提出了步长调度（stride scheduling），一个确定性的公平分配算法[W95]。

步长调度也很简单。系统中的每个工作都有自己的步长，这个值与票数值成反比。在上面的例子中，A、B、C 这 3 个工作的票数分别是 100、50 和 250，我们通过用一个大数分别除以他们的票数来获得每个进程的步长。比如用 10000 除以这些票数值，得到了 3 个进程的步长分别为 100、200 和 40。我们称这个值为每个进程的步长（stride）。每次进程运行后，我们会让它的计数器 [称为行程（pass）值] 增加它的步长，记录它的总体进展。

之后，调度程序使用进程的步长及行程值来确定调度哪个进程。基本思路很简单：当需要进行调度时，选择目前拥有最小行程值的进程，并且在运行之后将该进程的行程值增加一个步长。下面是 Waldspurger[W95]给出的伪代码：

```
current = remove_min(queue);       // pick client with minimum pass
schedule(current);                 // use resource for quantum
current->pass += current->stride;  // compute next pass using stride
insert(queue, current);            // put back into the queue
```

在我们的例子中，3 个进程（A、B、C）的步长值分别为 100、200 和 40，初始行程值都为 0。因此，最初，所有进程都可能被选择执行。假设选择 A（任意的，所有具有同样低的行程值的进程，都可能被选中）。A 执行一个时间片后，更新它的行程值为 100。然后运行 B，并更新其行程值为 200。最后执行 C，C 的行程值变为 40。这时，算法选择最小的行程值，是 C，执行并增加为 80（C 的步长是 40）。然后 C 再次运行（依然行程值最小），行程值增加到 120。现在运行 A，更新它的行程值为 200（现在与 B 相同）。然后 C 再次连续运行两次，行程值也变为 200。此时，所有行程值再次相等，这个过程会无限地重复下去。表 9.1 展示了一段时间内调度程序的行为。

表 9.1 步长调度：记录

行程值（A）（步长=100）	行程值（B）（步长=200）	行程值（C）（步长=40）	谁运行
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	……

可以看出，C 运行了 5 次、A 运行了 2 次，B 一次，正好是票数的比例——200、100 和 50。彩票调度算法只能一段时间后，在概率上实现比例，而步长调度算法可以在每个调度周期后做到完全正确。

你可能想知道，既然有了可以精确控制的步长调度算法，为什么还要彩票调度算法呢？好吧，彩票调度有一个步长调度没有的优势——不需要全局状态。假如一个新的进程在上面的步长调度执行过程中加入系统，应该怎么设置它的行程值呢？设置成 0 吗？这样的话，它就独占 CPU 了。而彩票调度算法不需要对每个进程记录全局状态，只需要用新进程的票数更新全局的总票数就可以了。因此彩票调度算法能够更合理地处理新加入的进程。

9.7 小结

本章介绍了比例份额调度的概念，并简单讨论了两种实现：彩票调度和步长调度。彩票调度通过随机值，聪明地做到了按比例分配。步长调度算法能够确定的获得需要的比例。虽然两者都很有趣，但由于一些原因，并没有作为 CPU 调度程序被广泛使用。一个原因是这两种方式都不能很好地适合 I/O[AC97]；另一个原因是其中最难的票数分配问题并没有确定的解决方式，例如，如何知道浏览器进程应该拥有多少票数？通用调度程序（像前面讨论的 MLFQ 及其他类似的 Linux 调度程序）做得更好，因此得到了广泛的应用。

结果，比例份额调度程序只有在这些问题可以相对容易解决的领域更有用（例如容易确定份额比例）。例如在虚拟（virtualized）数据中心中，你可能会希望分配 1/4 的 CPU 周期给 Windows 虚拟机，剩余的给 Linux 系统，比例分配的方式可以更简单高效。详细信息请参考 Waldspurger [W02]，该文介绍了 VMWare 的 ESX 系统如何用比例分配的方式来共享内存。

参考资料

[AC97]“Extending Proportional-Share Scheduling to a Network of Workstations”Andrea C. Arpaci-Dusseau and David E. Culler

PDPTA’97, June 1997

这是本书的一位作者撰写的论文，关于如何扩展比例共享调度，从而在群集环境中更好地工作。

[D82]“Why Numbering Should Start At Zero”
Edsger Dijkstra, August 1982
来自计算机科学先驱之一 E. Dijkstra 的简短讲义。在关于并发的部分，我们会听到更多关于 E. Dijkstra 的信息。与此同时，请阅读这份讲义，其中有一句激励人心的话：“我的一个同事（不是一个计算科学家）指责一些年轻的计算科学家‘卖弄学问’，因为他们从零开始编号。”该讲义解释了为什么这样做是合理的。

[KL88]“A Fair Share Scheduler”
J．Kay and P. Lauder
CACM, Volume 31 Issue 1, January 1988
关于公平份额调度程序的早期参考文献。

[WW94]“Lottery Scheduling: Flexible Proportional-Share Resource Management”Carl A. Waldspurger and William E. Weihl
OSDI ’94, November 1994
关于彩票调度的里程碑式的论文，让调度、公平分享和简单随机算法的力量在操作系统社区重新焕发了活力。

[W95]“Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management”Carl A. Waldspurger
Ph.D. Thesis, MIT, 1995
Waldspurger 的获奖论文，概述了彩票和步长调度。如果你想写一篇博士论文，总应该有一个很好的例子，让你有个努力的方向：这是一个很好的例子。

[W02]“Memory Resource Management in VMware ESX Server”Carl A. Waldspurger
OSDI ’02, Boston, Massachusetts
关于 VMM（虚拟机管理程序）中的内存管理的文章。除了相对容易阅读之外，该论文还包含许多有关新型 VMM 层面内存管理的很酷的想法。

作业

lottery.py 这个程序允许你查看彩票调度程序的工作原理。详情请参阅 README 文件。

问题

1．计算 3 个工作在随机种子为 1、2 和 3 时的模拟解。

2．现在运行两个具体的工作：每个长度为 10，但是一个（工作 0）只有一张彩票，另一个（工作 1）有 100 张（−l 10：1,10：100）。

彩票数量如此不平衡时会发生什么？在工作 1 完成之前，工作 0 是否会运行？多久？一般来说，这种彩票不平衡对彩票调度的行为有什么影响？

3．如果运行两个长度为 100 的工作，都有 100 张彩票（–l100：100,100：100），调度程序有多不公平？运行一些不同的随机种子来确定（概率上的）答案。不公平性取决于一项工作比另一项工作早完成多少。

4．随着量子规模（-q）变大，你对上一个问题的答案如何改变？

5．你可以制作类似本章中的图表吗？

还有什么值得探讨的？用步长调度程序，图表看起来如何？

第 10 章　多处理器调度（高级）

本章将介绍多处理器调度（multiprocessor scheduling）的基础知识。由于本章内容相对较深，建议认真学习并发相关的内容后再读。

过去很多年，多处理器（multiprocessor）系统只存在于高端服务器中。现在，它们越来越多地出现在个人 PC、笔记本电脑甚至移动设备上。多核处理器（multicore）将多个 CPU 核组装在一块芯片上，是这种扩散的根源。由于计算机的架构师们当时难以让单核 CPU 更快，同时又不增加太多功耗，所以这种多核 CPU 很快就变得流行。现在，我们每个人都可以得到一些 CPU，这是好事，对吧？

当然，多核 CPU 带来了许多困难。主要困难是典型的应用程序（例如你写的很多 C 程序）都只使用一个 CPU，增加了更多的 CPU 并没有让这类程序运行得更快。为了解决这个问题，不得不重写这些应用程序，使之能并行（parallel）执行，也许使用多线程（thread，本书的第 2 部分将用较多篇幅讨论）。多线程应用可以将工作分散到多个 CPU 上，因此 CPU 资源越多就运行越快。

补充：高级章节

需要阅读本书的更多内容才能真正理解高级章节，但这些内容在逻辑上放在一章里。例如，本章是关于多处理器调度的，如果先学习了中间部分的并发知识，会更有意思。但是，从逻辑上它属于本书中虚拟化（一般）和 CPU 调度（具体）的部分。因此，建议不按顺序学习这些高级章节。对于本章，建议在本书第 2 部分之后学习。

除了应用程序，操作系统遇到的一个新的问题是（不奇怪！）多处理器调度（multiprocessor scheduling）。到目前为止，我们讨论了许多单处理器调度的原则，那么如何将这些想法扩展到多处理器上呢？还有什么新的问题需要解决？因此，我们的问题如下。

关键问题：如何在多处理器上调度工作

操作系统应该如何在多 CPU 上调度工作？会遇到什么新问题？已有的技术依旧适用吗？是否需要新的思路？

10.1　背景：多处理器架构

为了理解多处理器调度带来的新问题，必须先知道它与单 CPU 之间的基本区别。区别的核心在于对硬件缓存（cache）的使用（见图 10.1），以及多处理器之间共享数据的方式。本章将在较高层面讨论这些问题。更多信息可以其他地方找到[CSG99]，尤其是在高年级或

研究生计算机架构课程中。

在单 CPU 系统中，存在多级的硬件缓存（hardware cache），一般来说会让处理器更快地执行程序。缓存是很小但很快的存储设备，通常拥有内存中最热的数据的备份。相比之下，内存很大且拥有所有的数据，但访问速度较慢。通过将频繁访问的数据放在缓存中，系统似乎拥有又大又快的内存。

举个例子，假设一个程序需要从内存中加载指令并读取一个值，系统只有一个 CPU，拥有较小的缓存（如 64KB）和较大的内存。

程序第一次读取数据时，数据在内存中，因此需要花费较长的时间（可能数十或数百纳秒）。处理器判断该数据很可能会被再次使用，因此将其放入 CPU 缓存中。如果之后程序再次需要使用同样的数据，CPU 会先查找缓存。因为在缓存中找到了数据，所以取数据快得多（比如几纳秒），程序也就运行更快。

缓存是基于局部性（locality）的概念，局部性有两种，即时间局部性和空间局部性。时间局部性是指当一个数据被访问后，它很有可能会在不久的将来被再次访问，比如循环代码中的数据或指令本身。而空间局部性指的是，当程序访问地址为 x 的数据时，很有可能会紧接着访问 x 周围的数据，比如遍历数组或指令的顺序执行。由于这两种局部性存在于大多数的程序中，硬件系统可以很好地预测哪些数据可以放入缓存，从而运行得很好。

有趣的部分来了：如果系统有多个处理器，并共享同一个内存，如图 10.2 所示，会怎样呢？

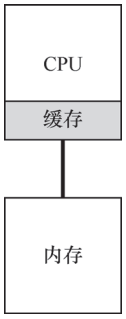


图 10.1 带缓存的单 CPU

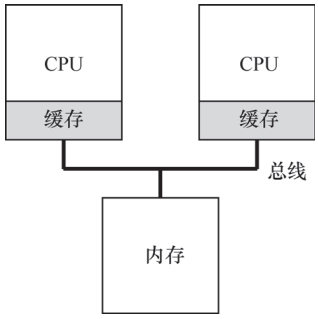


图 10.2 两个有缓存的 CPU 共享内存

事实证明，多 CPU 的情况下缓存要复杂得多。例如，假设一个运行在 CPU 1 上的程序从内存地址 A 读取数据。由于不在 CPU 1 的缓存中，所以系统直接访问内存，得到值 D。程序然后修改了地址 A 处的值，只是将它的缓存更新为新值 D'。将数据写回内存比较慢，因此系统（通常）会稍后再做。假设这时操作系统中断了该程序的运行，并将其交给 CPU 2，重新读取地址 A 的数据，由于 CPU 2 的缓存中并没有该数据，所以会直接从内存中读取，得到了旧值 D，而不是正确的值 D'。哎呀！

这一普遍的问题称为缓存一致性（cache coherence）问题，有大量的研究文献描述了解决这个问题时的微妙之处[SHW11]。这里我们会略过所有的细节，只提几个要点。选一门计算机体系结构课（或 3 门），你可以了解更多。

硬件提供了这个问题的基本解决方案：通过监控内存访问，硬件可以保证获得正确的数据，并保证共享内存的唯一性。在基于总线的系统中，一种方式是使用总线窥探（bus

snooping）[G83]。每个缓存都通过监听链接所有缓存和内存的总线，来发现内存访问。如果 CPU 发现对它放在缓存中的数据的更新，会作废（invalidate）本地副本（从缓存中移除），或更新（update）它（修改为新值）。回写缓存，如上面提到的，让事情更复杂（由于对内存的写入稍后才会看到），你可以想想基本方案如何工作。

10.2 别忘了同步

既然缓存已经做了这么多工作来提供一致性，应用程序（或操作系统）还需要关心共享数据的访问吗？依然需要！本书第 2 部分关于并发的描述中会详细介绍。虽然这里不会详细讨论，但我们会简单介绍（或复习）下其基本思路（假设你熟悉并发相关内容）。

跨 CPU 访问（尤其是写入）共享数据或数据结构时，需要使用互斥原语（比如锁），才能保证正确性（其他方法，如使用无锁（lock-free）数据结构，很复杂，偶尔才使用。详情参见并发部分关于死锁的章节）。例如，假设多 CPU 并发访问一个共享队列。如果没有锁，即使有底层一致性协议，并发地从队列增加或删除元素，依然不会得到预期结果。需要用锁来保证数据结构状态更新的原子性。

为了更具体，我们设想这样的代码序列，用于删除共享链表的一个元素，如图 10.3 所示。假设两个 CPU 上的不同线程同时进入这个函数。如果线程 1 执行第一行，会将 head 的当前值存入它的 tmp 变量。如果线程 2 接着也执行第一行，它也会将同样的 head 值存入它自己的私有 tmp 变量（tmp 在栈上分配，因此每个线程都有自己的私有存储）。因此，两个线程会尝试删除同一个链表头，而不是每个线程移除一个元素，这导致了各种问题（比如在第 4 行重复释放头元素，以及可能两次返回同一个数据）。

```
1    typedef struct __Node_t {
2        int value;
3        struct __Node_t *next;
4    } Node_t;
5
6    int List_Pop() {
7        Node_t *tmp = head;       // remember old head ...
8        int value = head->value;  // ... and its value
9        head = head->next;        // advance head to next pointer
10       free(tmp);                // free old head
11       return value;             // return value at head
12   }
```

图 10.3 简单的链表删除代码

当然，让这类函数正确工作的方法是加锁（locking）。这里只需要一个互斥锁（即 pthread_mutex_t m;），然后在函数开始时调用 lock(&m)，在结束时调用 unlock(&m)，确保代码的执行如预期。我们会看到，这里依然有问题，尤其是性能方面。具体来说，随着 CPU 数量的增加，访问同步共享的数据结构会变得很慢。

10.3 最后一个问题：缓存亲和度

在设计多处理器调度时遇到的最后一个问题，是所谓的缓存亲和度（cache affinity）。这个概念很简单：一个进程在某个 CPU 上运行时，会在该 CPU 的缓存中维护许多状态。下次该进程在相同 CPU 上运行时，由于缓存中的数据而执行得更快。相反，在不同的 CPU 上执行，会由于需要重新加载数据而很慢（好在硬件保证的缓存一致性可以保证正确执行）。因此多处理器调度应该考虑到这种缓存亲和性，并尽可能将进程保持在同一个 CPU 上。

10.4 单队列调度

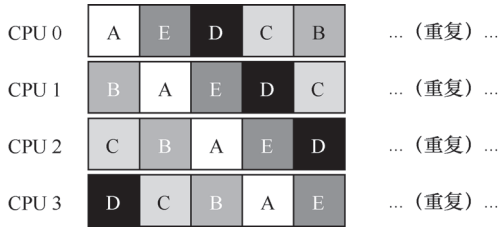
上面介绍了一些背景，现在来讨论如何设计一个多处理器系统的调度程序。最基本的方式是简单地复用单处理器调度的基本架构，将所有需要调度的工作放入一个单独的队列中，我们称之为单队列多处理器调度（Single Queue Multiprocessor Scheduling，SQMS）。这个方法最大的优点是简单。它不需要太多修改，就可以将原有的策略用于多个 CPU，选择最适合的工作来运行（例如，如果有两个 CPU，它可能选择两个最合适的工作）。

然而，SQMS 有几个明显的短板。第一个是缺乏可扩展性（scalability）。为了保证在多 CPU 上正常运行，调度程序的开发者需要在代码中通过加锁（locking）来保证原子性，如上所述。在 SQMS 访问单个队列时（如寻找下一个运行的工作），锁确保得到正确的结果。

然而，锁可能带来巨大的性能损失，尤其是随着系统中的 CPU 数增加时[A91]。随着这种单个锁的争用增加，系统花费了越来越多的时间在锁的开销上，较少的时间用于系统应该完成的工作（哪天在这里加上真正的测量数据就好了）。

SQMS 的第二个主要问题是缓存亲和性。比如，假设我们有 5 个工作（A、B、C、D、E）和 4 个处理器。调度队列如下：

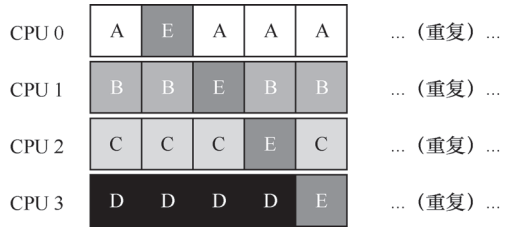
一段时间后，假设每个工作依次执行一个时间片，然后选择另一个工作，下面是每个 CPU 可能的调度序列：



由于每个 CPU 都简单地从全局共享的队列中选取下一个工作执行，因此每个工作都不断在不同 CPU 之间转移，这与缓存亲和的目标背道而驰。

为了解决这个问题，大多数 SQMS 调度程序都引入了一些亲和度机制，尽可能让进程

在同一个 CPU 上运行。保持一些工作的亲和度的同时，可能需要牺牲其他工作的亲和度来实现负载均衡。例如，针对同样的 5 个工作的调度如下：



这种调度中，A、B、C、D 这 4 个工作都保持在同一个 CPU 上，只有工作 E 不断地来回迁移（migrating），从而尽可能多地获得缓存亲和度。为了公平起见，之后我们可以选择不同的工作来迁移。但实现这种策略可能很复杂。

我们看到，SQMS 调度方式有优势也有不足。优势是能够从单 CPU 调度程序很简单地发展而来，根据定义，它只有一个队列。然而，它的扩展性不好（由于同步开销有限），并且不能很好地保证缓存亲和度。

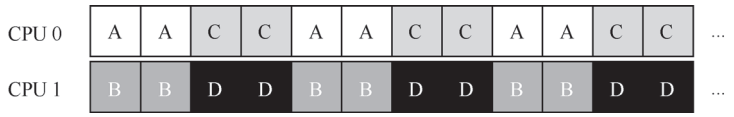
10.5 多队列调度

正是由于单队列调度程序的这些问题，有些系统使用了多队列的方案，比如每个 CPU 一个队列。我们称之为多队列多处理器调度（Multi-Queue Multiprocessor Scheduling，MQMS）

在 MQMS 中，基本调度框架包含多个调度队列，每个队列可以使用不同的调度规则，比如轮转或其他任何可能的算法。当一个工作进入系统后，系统会依照一些启发性规则（如随机或选择较空的队列）将其放入某个调度队列。这样一来，每个 CPU 调度之间相互独立，就避免了单队列的方式中由于数据共享及同步带来的问题。

例如，假设系统中有两个 CPU（CPU 0 和 CPU 1）。这时一些工作进入系统：A、B、C 和 D。由于每个 CPU 都有自己的调度队列，操作系统需要决定每个工作放入哪个队列。可能像下面这样做：

根据不同队列的调度策略，每个 CPU 从两个工作中选择，决定谁将运行。例如，利用轮转，调度结果可能如下所示：



MQMS 比 SQMS 有明显的优势，它天生更具有可扩展性。队列的数量会随着 CPU 的增加而增加，因此锁和缓存争用的开销不是大问题。此外，MQMS 天生具有良好的缓存亲和度。所有工作都保持在固定的 CPU 上，因而可以很好地利用缓存数据。

但是，如果稍加注意，你可能会发现有一个新问题（这在多队列的方法中是根本的），即负载不均（load imbalance）。假定和上面设定一样（4 个工作，2 个 CPU），但假设一个工

作（如 C）这时执行完毕。现在调度队列如下：

如果对系统中每个队列都执行轮转调度策略，会获得如下调度结果：

从图中可以看出，A 获得了 B 和 D 两倍的 CPU 时间，这不是期望的结果。更糟的是，假设 A 和 C 都执行完毕，系统中只有 B 和 D。调度队列看起来如下：

因此 CPU 使用时间线看起来令人难过：

所以可怜的多队列多处理器调度程序应该怎么办呢？怎样才能克服潜伏的负载不均问题，打败邪恶的……霸天虎军团[①]？如何才能不要问这些与这本好书几乎无关的问题？

关键问题：如何应对负载不均

多队列多处理器调度程序应该如何处理负载不均问题，从而更好地实现预期的调度目标？

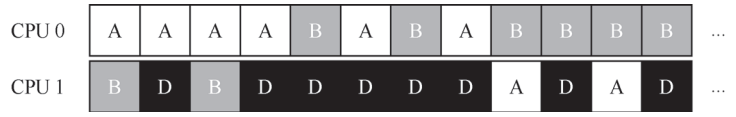
最明显的答案是让工作移动，这种技术我们称为迁移（migration）。通过工作的跨 CPU 迁移，可以真正实现负载均衡。

来看两个例子就更清楚了。同样，有一个 CPU 空闲，另一个 CPU 有一些工作。

在这种情况下，期望的迁移很容易理解：操作系统应该将 B 或 D 迁移到 CPU0。这次工作迁移导致负载均衡，皆大欢喜。

更棘手的情况是较早一些的例子，A 独自留在 CPU 0 上，B 和 D 在 CPU 1 上交替运行。

在这种情况下，单次迁移并不能解决问题。应该怎么做呢？答案是不断地迁移一个或多个工作。一种可能的解决方案是不断切换工作，如下面的时间线所示。可以看到，开始的时候 A 独享 CPU 0，B 和 D 在 CPU 1。一些时间片后，B 迁移到 CPU 0 与 A 竞争，D 则独享 CPU 1 一段时间。这样就实现了负载均衡。



① 一个鲜为人知的事实是，变形金刚的家乡塞伯坦星球被糟糕的 CPU 调度决策所摧毁。

当然，还有其他不同的迁移模式。但现在是最棘手的部分：系统如何决定发起这样的迁移？

一个基本的方法是采用一种技术，名为工作窃取（work stealing）[FLR98]。通过这种方法，工作量较少的（源）队列不定期地“偷看”其他（目标）队列是不是比自己的工作多。如果目标队列比源队列（显著地）更满，就从目标队列“窃取”一个或多个工作，实现负载均衡。

当然，这种方法也有让人抓狂的地方——如果太频繁地检查其他队列，就会带来较高的开销，可扩展性不好，而这是多队列调度最初的全部目标！相反，如果检查间隔太长，又可能会带来严重的负载不均。找到合适的阈值仍然是黑魔法，这在系统策略设计中很常见。

10.6 Linux 多处理器调度

有趣的是，在构建多处理器调度程序方面，Linux 社区一直没有达成共识。一直以来，存在 3 种不同的调度程序：O(1)调度程序、完全公平调度程序（CFS）以及 BF 调度程序（BFS）[①]。从 Meehean 的论文中可以找到对这些不同调度程序优缺点的对比总结[M11]。这里我们只总结一些基本知识。

O(1) CFS 采用多队列，而 BFS 采用单队列，这说明两种方法都可以成功。当然它们之间还有很多不同的细节。例如，O(1)调度程序是基于优先级的（类似于之前介绍的 MLFQ），随时间推移改变进程的优先级，然后调度最高优先级进程，来实现各种调度目标。交互性得到了特别关注。与之不同，CFS 是确定的比例调度方法（类似之前介绍的步长调度）。BFS 作为三个算法中唯一采用单队列的算法，也基于比例调度，但采用了更复杂的方案，称为最早最合适虚拟截止时间优先算法（EEVEF）[SA96]读者可以自己去了解这些现代操作系统的调度算法，现在应该能够理解它们的工作原理了！

10.7 小结

本章介绍了多处理器调度的不同方法。其中单队列的方式（SQMS）比较容易构建，负载均衡较好，但在扩展性和缓存亲和度方面有着固有的缺陷。多队列的方式（MQMS）有很好的扩展性和缓存亲和度，但实现负载均衡却很困难，也更复杂。无论采用哪种方式，都没有简单的答案：构建一个通用的调度程序仍是一项令人生畏的任务，因为即使很小的代码变动，也有可能导致巨大的行为差异。除非很清楚自己在做什么，或者有人付你很多钱，否则别干这种事。

参考资料

[A90]“The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors”Thomas E. Anderson IEEE TPDS Volume 1:1, January 1990

① 自己去查 BF 代表什么。预先警告，小心脏可能受不了。

这是一篇关于不同加锁方案扩展性好坏的经典论文。Tom Anderson 是非常著名的系统和网络研究者，也是一本非常好的操作系统教科书的作者。

[B+10]“An Analysis of Linux Scalability to Many Cores Abstract”
Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich
OSDI ’10, Vancouver, Canada, October 2010
关于将 Linux 扩展到多核的很好的现代论文。

[CSG99]“Parallel Computer Architecture: A Hardware/Software Approach”David E. Culler, Jaswinder Pal Singh, and Anoop Gupta
Morgan Kaufmann, 1999
其中充满了并行机器和算法细节的宝藏。正如 Mark Hill 幽默地在书的护封上说的——这本书所包含的信息比大多数研究论文都多。

[FLR98]“The Implementation of the Cilk-5 Multithreaded Language”Matteo Frigo, Charles E. Leiserson, Keith Randall
PLDI ’98, Montreal, Canada, June 1998
Cilk 是用于编写并行程序的轻量级语言和运行库，并且是工作窃取范式的极好例子。

[G83]“Using Cache Memory To Reduce Processor-Memory Traffic”James R. Goodman
ISCA ’83, Stockholm, Sweden, June 1983
关于如何使用总线监听，即关注总线上看到的请求，构建高速缓存一致性协议的开创性论文。Goodman 在威斯康星的多年研究工作充满了智慧，这只是一个例子。

[M11]“Towards Transparent CPU Scheduling”Joseph T. Meehean
Doctoral Dissertation at University of Wisconsin—Madison, 2011
一篇涵盖了现代 Linux 多处理器调度如何工作的许多细节的论文。非常棒！但是，作为 Joe 的联合导师，我们可能在这里有点偏心。

[SHW11]“A Primer on Memory Consistency and Cache Coherence”Daniel J. Sorin, Mark D. Hill, and David A. Wood
Synthesis Lectures in Computer Architecture
Morgan and Claypool Publishers, May 2011
内存一致性和多处理器缓存的权威概述。对于喜欢对该主题深入了解的人来说，这是必读物。

[SA96]“Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Pro- portional Share Resource Allocation”
Ion Stoica and Hussein Abdel-Wahab
Technical Report TR-95-22, Old Dominion University, 1996
来自 Ion Stoica 的一份技术报告，其中介绍了很酷的调度思想。他现在是 U.C.伯克利大学的教授，也是网络、分布式系统和其他许多方面的世界级专家。

第 11 章　关于 CPU 虚拟化的总结对话

教授：那么，同学，你学到了什么？

学生：教授，这似乎是一个既定答案的问题。我想你只想让我说“是的，我学到了”。

教授：确实。但这也还是一个诚实的问题。来吧，让教授休息一下，好吗？

学生：好的，好的。我想我确实学到了一些知识。首先，我了解了操作系统如何虚拟化 CPU。为了理解这一点，我必须了解一些重要的机制（mechanism）：陷阱和陷阱处理程序，时钟中断以及操作系统和硬件在进程间切换时如何谨慎地保存和恢复状态。

教授：很好，很好！

学生：虽然所有这些交互似乎有点复杂，但我怎样才能学到更多的内容？

教授：好的，这是一个很好的问题。我认为没有办法可以替代动手。仅阅读这些内容并不能给你正确的理解。做课堂项目，我敢保证，它会对你有所帮助。

学生：听起来不错。我还能告诉你什么？

教授：那么，你在寻求理解操作系统的基本机制时，是否了解了操作系统的哲学？

学生：嗯……我想是的。似乎操作系统相当偏执。它希望确保控制机器。虽然它希望程序能够尽可能高效地运行 [因此也是受限直接执行（limited direct execution）背后的全部逻辑]，但操作系统也希望能够对错误或恶意的程序说“啊！别那么快，我的朋友”。偏执狂全天控制，并且确保操作系统控制机器。也许这就是我们将操作系统视为资源管理器的原因。

教授：是的，听起来你开始融会贯通了！干得漂亮！

学生：谢谢。

教授：那些机制之上的策略呢？有什么有趣的经验吗？

学生：当然能从中学到一些经验。也许有点明显，但明显也可以是很好。比如将短工作提升到队列前面的想法：自从有一次我在商店买一些口香糖，我就知道这是一个好主意，而且我面前的那个人有一张无法支付的信用卡。我要说的是，他不是“短工作”。

教授：这听起来对那个可怜的家伙有点过分。还有什么吗？

学生：好吧，你可以建立一个聪明的调度程序，试图既像 SJF 又像 RR——MLFQ 相当漂亮。构建真正的调度程序似乎很难。

教授：的确如此。这就是对使用哪个调度程序至今仍有争议的原因。例如，请参阅 CFS、BFS 和 O（1）调度程序之间的 Linux 战斗。不，我不会说出 BFS 的全名。

学生：我不会要求你说！这些策略战争看起来好像可以永远持续下去，真的有一个正确的答案吗？

教授：可能没有。毕竟，即使我们自己的度量指标也不一致。如果你的调度程序周转时间好，那么在响应时间就会很糟糕，反之亦然。正如 Lampson 说的，也许目标不是找到最好的解决方案，而是为了避免灾难。

学生：这有点令人沮丧。

教授：好的工程可以这样。它也可以令人振奋！这只是你的观点，真的。我个人认为，务实是一件好事，实用主义者意识到并非所有问题都有简洁明了的解决方案。你还喜欢什么？

学生：我非常喜欢操控调度程序的概念。我下次在亚马逊的 EC2 服务上运行一项工作时，看起来这可能是需要考虑的事情。也许我可以从其他一些毫无戒心的（更重要的是，对操作系统一无所知的）客户那里窃取一些时间周期！

教授：看起来我可能创造了一个“怪物”！你知道，我可不想被人称为弗兰肯斯坦教授。

学生：但你不就是这样想的吗？让我们对某件事感到兴奋，这样我们就会自己对它进行研究？点燃火，仅此而已？

教授：我想是的。但我不认为这会成功！

第 12 章　关于内存虚拟化的对话

学生：那么，虚拟化讲完了吗？

教授：没有！

学生：嘿，没理由这么激动，我只是在问一个问题。学生就应该问问题，对吧？

教授：好吧，教授们总是这样说，但实际上他们的意思：提出问题，仅当它们是好问题，而且你实际上已经对这些问题进行了一些思考。

学生：好吧，那肯定会让我失去动力。

教授：我得逞了。不管怎么说，我们离讲完虚拟化还有一段时间！相反，你刚看到了如何虚拟化 CPU，但是真的有一个巨大的“怪物”—— 内存在壁橱里等着你。虚拟内存很复杂，需要我们理解关于硬件和操作系统交互方式的更多复杂细节。

学生：听起来很酷。为什么这很难？

教授：好吧，有很多细节，你必须牢记它们，才能真正对发生的事情建立一个思维模型。我们将从简单的开始，使用诸如基址/界限等非常基本的技术，并慢慢增加复杂性以应对新的挑战，包括有趣的主题，如 TLB 和多级页表。最终，我们将能够描述一个全功能的现代虚拟内存管理程序的工作原理。

学生：漂亮！对我这个可怜的学生有什么提示吗？会被这些信息淹没，并且一般都会睡眠不足？

教授：对于睡眠不足的人来说，这很简单：多睡一会儿（少一点派对）。对于理解虚拟内存，从这里开始：用户程序生成的每个地址都是虚拟地址（every address generated by a user program is a virtual address）。操作系统只是为每个进程提供一个假象，具体来说，就是它拥有自己的大量私有内存。在一些硬件帮助下，操作系统会将这些假的虚拟地址变成真实的物理地址，从而能够找到想要的信息。

学生：好的，我想我可以记住……（自言自语）用户程序中的每个地址都是虚拟的，用户程序中的每个地址都是虚拟的，每个地址都是……

教授：你在嘟囔什么？

学生：哦，没什么……（尴尬的停顿）……但是，操作系统为什么又要提供这种假象？

教授：主要是为了易于使用（ease of use）。操作系统会让每个程序觉得，它有一个很大的连续地址空间（address space）来放入其代码和数据。因此，作为一名程序员，您不必担心诸如“我应该在哪里存储这个变量？”这样的事情，因为程序的虚拟地址空间很大，有很多空间可以存代码和数据。对于程序员来说，如果必须操心将所有的代码数据放入一个小而拥挤的内存，那么生活会变得痛苦得多。

学生：为什么呢？

教授：好吧，隔离（isolation）和保护（protection）也是大事。我们不希望一个错误的程序能够读取或者覆写其他程序的内存，对吗？

学生：可能不希望。除非它是由你不喜欢的人编写的程序。

教授：嗯……我想可能需要在下个学期为你安排一门道德与伦理课程。也许操作系统课程没有传递正确的信息。

学生：也许应该。但请记住，不是我对大家说，对于错误的进程行为，正确的操作系统反应是要“杀死”违规进程！

第 13 章　抽象：地址空间

早期，构建计算机操作系统非常简单。你可能会问，为什么？因为用户对操作系统的期望不高。然而一些烦人的用户提出要“易于使用”“高性能”“可靠性”等，这导致了所有这些令人头痛的问题。下次你见到这些用户的时候，应该感谢他们，他们是这些问题的根源。

13.1　早期系统

从内存来看，早期的机器并没有提供多少抽象给用户。基本上，机器的物理内存看起来如图 13.1 所示。

操作系统曾经是一组函数（实际上是一个库），在内存中（在本例中，从物理地址 0 开始），然后有一个正在运行的程序（进程），目前在物理内存中（在本例中，从物理地址 64KB 开始），并使用剩余的内存。这里几乎没有抽象，用户对操作系统的要求也不多。那时候，操作系统开发人员的生活确实很容易，不是吗？

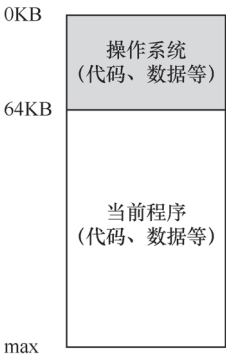


图 13.1　操作系统：早期

13.2　多道程序和时分共享

过了一段时间，由于机器昂贵，人们开始更有效地共享机器。因此，多道程序（multiprogramming）系统时代开启[DV66]，其中多个进程在给定时间准备运行，比如当有一个进程在等待 I/O 操作的时候，操作系统会切换这些进程，这样增加了 CPU 的有效利用率（utilization）。那时候，效率（efficiency）的提高尤其重要，因为每台机器的成本是数十万美元甚至数百万美元（现在你觉得你的 Mac 很贵！）

但很快，人们开始对机器要求更多，分时系统的时代诞生了[S59，L60，M62，M83]。具体来说，许多人意识到批量计算的局限性，尤其是程序员本身[CV65]，他们厌倦了长时间的（因此也是低效率的）编程—调试循环。交互性（interactivity）变得很重要，因为许多用户可能同时在使用机器，每个人都在等待（或希望）他们执行的任务及时响应。

一种实现时分共享的方法，是让一个进程单独占用全部内存运行一小段时间（见图 13.1），然后停止它，并将它所有的状态信息保存在磁盘上（包含所有的物理内存），加载其他进程的状态信息，再运行一段时间，这就实现了某种比较粗糙的机器共享[M+63]。

遗憾的是，这种方法有一个问题：太慢了，特别是当内存增长的时候。虽然保存和恢

复寄存器级的状态信息（程序计数器、通用寄存器等）相对较快，但将全部的内存信息保存到磁盘就太慢了。因此，在进程切换的时候，我们仍然将进程信息放在内存中，这样操作系统可以更有效率地实现时分共享（见图 13.2）。

图 13.2 3 个进程：共享内存

在图 13.2 中，有 3 个进程（A、B、C），每个进程拥有从 512KB 物理内存中切出来给它们的一小部分内存。假定只有一个 CPU，操作系统选择运行其中一个进程（比如 A），同时其他进程（B 和 C）则在队列中等待运行。

随着时分共享变得更流行，人们对操作系统又有了新的要求。特别是多个程序同时驻留在内存中，使保护（protection）成为重要问题。人们不希望一个进程可以读取其他进程的内存，更别说修改了。

13.3 地址空间

然而，我们必须将这些烦人的用户的需求放在心上。因此操作系统需要提供一个易用（easy to use）的物理内存抽象。这个抽象叫作地址空间（address space），是运行的程序看到的系统中的内存。理解这个基本的操作系统内存抽象，是了解内存虚拟化的关键。

一个进程的地址空间包含运行的程序的所有内存状态。比如：程序的代码（code，指令）必须在内存中，因此它们在地址空间里。当程序在运行的时候，利用栈（stack）来保存当前的函数调用信息，分配空间给局部变量，传递参数和函数返回值。最后，堆（heap）用于管理动态分配的、用户管理的内存，就像你从 C 语言中调用 malloc()或面向对象语言（如 C ++ 或 Java）中调用 new 获得内存。当然，还有其他的东西（例如，静态初始化的变量），但现在假设只有这 3 个部分：代码、栈和堆。

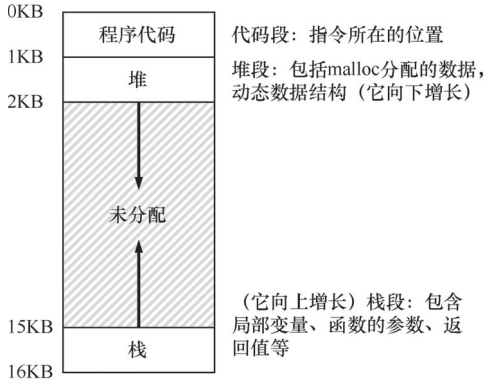


图 13.3 地址空间的例子

在图 13.3 的例子中，我们有一个很小的地址空间[①]（只有 16KB）。程序代码位于地址空间的顶部（在本例中从 0 开始，并且装入到地址空间的前 1KB）。代码是静态的（因此很容易放在内存中），所以可以将它放在地址空间的顶部，我们知道程序运行时不再需要新的空间。

接下来，在程序运行时，地址空间有两个区域可能增长（或者收缩）。它们就是堆（在顶部）和栈（在底部）。把它们放在那里，是因为它们都希望能够增长。通过将它们放在地址空间的两端，我们可以允许这样的增长：它们只需要在相反的方向增长。因此堆在代码（1KB）之下开始并向下增长（当用户通过 malloc()请求更多内存时），栈从 16KB 开始并向上增长

① 我们通常会使用这样的小例子，原因有二：①表示 32 位地址空间是一种痛苦；②数学计算更难。我们喜欢简单的数学。

（当用户进行程序调用时）。然而，堆栈和堆的这种放置方法只是一种约定，如果你愿意，可以用不同的方式安排地址空间 [稍后我们会看到，当多个线程（threads）在地址空间中共存时，就没有像这样分配空间的好办法了]。

当然，当我们描述地址空间时，所描述的是操作系统提供给运行程序的抽象（abstract）。程序不在物理地址 0～16KB 的内存中，而是加载在任意的物理地址。回顾图 13.2 中的进程 A、B 和 C，你可以看到每个进程如何加载到内存中的不同地址。因此问题来了：

关键问题：如何虚拟化内存

操作系统如何在单一的物理内存上为多个运行的进程（所有进程共享内存）构建一个私有的、可能很大的地址空间的抽象？

当操作系统这样做时，我们说操作系统在虚拟化内存（virtualizing memory），因为运行的程序认为它被加载到特定地址（例如 0）的内存中，并且具有非常大的地址空间（例如 32 位或 64 位）。现实很不一样。

例如，当图 13.2 中的进程 A 尝试在地址 0（我们将称其为虚拟地址，virtual address）执行加载操作时，然而操作系统在硬件的支持下，出于某种原因，必须确保不是加载到物理地址 0，而是物理地址 320KB（这是 A 载入内存的地址）。这是内存虚拟化的关键，这是世界上每一个现代计算机系统的基础。

提示：隔离原则

隔离是建立可靠系统的关键原则。如果两个实体相互隔离，这意味着一个实体的失败不会影响另一个实体。操作系统力求让进程彼此隔离，从而防止相互造成伤害。通过内存隔离，操作系统进一步确保运行程序不会影响底层操作系统的操作。一些现代操作系统通过将某些部分与操作系统的其他部分分离，实现进一步的隔离。这样的微内核（microkernel）[BH70，R+89，S+03] 可以比整体内核提供更大的可靠性。

13.4 目标

在这一章中，我们触及操作系统的工作——虚拟化内存。操作系统不仅虚拟化内存，还有一定的风格。为了确保操作系统这样做，我们需要一些目标来指导。以前我们已经看过这些目标（想想本章的前言），我们会再次看到它们，但它们肯定是值得重复的。

虚拟内存（VM）系统的一个主要目标是透明（transparency）[①]。操作系统实现虚拟内存的方式，应该让运行的程序看不见。因此，程序不应该感知到内存被虚拟化的事实，相反，程序的行为就好像它拥有自己的私有物理内存。在幕后，操作系统（和硬件）完成了所有的工作，让不同的工作复用内存，从而实现这个假象。

① 透明的这种用法有时令人困惑。一些学生认为“变得透明”意味着把所有事情都公之于众。在这里，“变得透明”意味着相反的情况：操作系统提供的假象不应该被应用程序看破。因此，按照通常的用法，透明系统是一个很难注意到的系统。

虚拟内存的另一个目标是效率（efficiency）。操作系统应该追求虚拟化尽可能高效（efficient），包括时间上（即不会使程序运行得更慢）和空间上（即不需要太多额外的内存来支持虚拟化）。在实现高效率虚拟化时，操作系统将不得不依靠硬件支持，包括 TLB 这样的硬件功能（我们将在适当的时候学习）。

最后，虚拟内存第三个目标是保护（protection）。操作系统应确保进程受到保护（protect），不会受其他进程影响，操作系统本身也不会受进程影响。当一个进程执行加载、存储或指令提取时，它不应该以任何方式访问或影响任何其他进程或操作系统本身的内存内容（即在它的地址空间之外的任何内容）。因此，保护让我们能够在进程之间提供隔离（isolation）的特性，每个进程都应该在自己的独立环境中运行，避免其他出错或恶意进程的影响。

补充：你看到的所有地址都不是真的

写过打印出指针的 C 程序吗？你看到的值（一些大数字，通常以十六进制打印）是虚拟地址（virtual address）。有没有想过你的程序代码在哪里找到？你也可以打印出来，是的，如果你可以打印它，它也是一个虚拟地址。实际上，作为用户级程序的程序员，可以看到的任何地址都是虚拟地址。只有操作系统，通过精妙的虚拟化内存技术，知道这些指令和数据所在的物理内存的位置。所以永远不要忘记：如果你在一个程序中打印出一个地址，那就是一个虚拟的地址。虚拟地址只是提供地址如何在内存中分布的假象，只有操作系统（和硬件）才知道物理地址。

这里有一个小程序，打印出 main() 函数（代码所在地方）的地址，由 malloc()返回的堆空间分配的值，以及栈上一个整数的地址：

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    int main(int argc, char *argv[]) {
4        printf("location of code  : %p\n", (void *) main);
5        printf("location of heap  : %p\n", (void *) malloc(1));
6        int x = 3;
7        printf("location of stack : %p\n", (void *) &x);
8        return x;
9    }
```

在 64 位的 Mac 上面运行时，我们得到以下输出：

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack : 0x7fff691aea64
```

从这里，你可以看到代码在地址空间开头，然后是堆，而栈在这个大型虚拟地址空间的另一端。所有这些地址都是虚拟的，并且将由操作系统和硬件翻译成物理地址，以便从真实的物理位置获取该地址的值。

在接下来的章节中，我们将重点介绍虚拟化内存所需的基本机制（mechanism），包括硬件和操作系统的支持。我们还将研究一些较相关的策略（policy），你会在操作系统中遇到它们，包括如何管理可用空间，以及在空间不足时哪些页面该释放。通过这些内容，你会逐渐理解现代虚拟内存系统真正的工作原理[①]。

① 或者，我们会说服你放弃课程。但请坚持下去，如果你坚持学完虚拟内存系统，很可能会坚持到底！

13.5 小结

我们介绍了操作系统的一个重要子系统：虚拟内存。虚拟内存系统负责为程序提供一个巨大的、稀疏的、私有的地址空间的假象，其中保存了程序的所有指令和数据。操作系统在专门硬件的帮助下，通过每一个虚拟内存的索引，将其转换为物理地址，物理内存根据获得的物理地址去获取所需的信息。操作系统会同时对许多进程执行此操作，并且确保程序之间互相不会受到影响，也不会影响操作系统。整个方法需要大量的机制（很多底层机制）和一些关键的策略。我们将自底向上，先描述关键机制。我们继续吧！

参考资料

[BH70]“The Nucleus of a Multiprogramming System”Per Brinch Hansen
Communications of the ACM, 13:4, April 1970
第一篇建议 OS 或内核应该是构建定制操作系统的最小且灵活的基础的论文，这个主题将在整个 OS 研究历史中重新被关注。

[CV65]“Introduction and Overview of the Multics System”
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
一篇卓越的早期 Multics 论文。下面是关于时分共享的一句名言：“时分共享的动力首先来自专业程序员，因为他们在批处理系统中调试程序时经常感到沮丧。因此，时分共享计算机最初的目标，是以允许几个人同时使用，并为他们每个人提供使用整台机器的假象。”

[DV66]“Programming Semantics for Multiprogrammed Computations”Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
关于多道程序系统的早期论文（但不是第一篇）。

[L60]“Man-Computer Symbiosis”
J. C. R. Licklider
IRE Transactions on Human Factors in Electronics, HFE-1:1, March 1960
一篇关于计算机和人类如何进入共生时代的趣味论文，显然超越了它的时代，但仍然令人着迷。

[M62]“Time-Sharing Computer Systems”
J. McCarthy
Management and the Computer of the Future, MIT Press, Cambridge, Mass, 1962
可能是 McCarthy 最早的关于时分共享的论文。然而，在另一篇论文[M83]中，他声称自 1957 年以来一直在思考这个想法。McCarthy 离开了系统领域，并在斯坦福大学成为人工智能领域的巨人，其工作包括创建

LISP 编程语言。查看 McCarthy 的主页可以了解更多信息。

[M+63]“A Time-Sharing Debugging System for a Small Computer”
J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider AFIPS ’63 (Spring), New York, NY, May 1963
这是一个很好的早期系统例子，当程序没有运行时将程序存储器交换到“鼓”，然后在运行时回到“核心”存储器。

[M83]“Reminiscences on the History of Time Sharing”John McCarthy
Winter or Spring of 1983
关于时分共享思想可能来自何处的一个了不起的历史记录，包括针对那些引用 Strachey 的作品[S59]作为这一领域开拓性工作的人的一些怀疑。

[NS07]“Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”Nicholas Nethercote and Julian Seward
PLDI 2007, San Diego, California, June 2007
对于那些使用 C 这样的不安全语言的人来说，Valgrind 是程序的救星。阅读本文以了解其非常酷的二进制探测技术——这真是令人印象深刻。

[R+89]“Mach: A System Software kernel”
Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones
COMPCON 89, February 1989
尽管这不是微内核的第一个项目，但 CMU 的 Mach 项目是众所周知的、有影响力的。它仍然深深扎根于 macOS X 的深处。

[S59]“Time Sharing in Large Fast Computers”
C. Strachey
Proceedings of the International Conference on Information Processing, UNESCO, June 1959
关于时分共享的最早参考文献之一。

[S+03]“Improving the Reliability of Commodity Operating Systems”Michael M. Swift, Brian N. Bershad, Henry M. Levy
SOSP 2003
第一篇介绍微内核思想如何提高操作系统可靠性的论文。

第 14 章　插叙：内存操作 API

在本章中，我们将介绍 UNIX 操作系统的内存分配接口。操作系统提供的接口非常简洁，因此本章简明扼要[①]。本章主要关注的问题是：

关键问题：如何分配和管理内存

在 UNIX/C 程序中，理解如何分配和管理内存是构建健壮和可靠软件的重要基础。通常使用哪些接口？哪些错误需要避免？

14.1　内存类型

在运行一个 C 程序的时候，会分配两种类型的内存。第一种称为栈内存，它的申请和释放操作是编译器来隐式管理的，所以有时也称为自动（automatic）内存。

C 中申请栈内存很容易。比如，假设需要在 func()函数中为一个整形变量 *x* 申请空间。为了声明这样的一块内存，只需要这样做：

```
void func() {
    int x; // declares an integer on the stack
    ...
}
```

编译器完成剩下的事情，确保在你进入 func() 函数的时候，在栈上开辟空间。当你从该函数退出时，编译器释放内存。因此，如果你希望某些信息存在于函数调用之外，建议不要将它们放在栈上。

就是这种对长期内存的需求，所以我们才需要第二种类型的内存，即所谓的堆（heap）内存，其中所有的申请和释放操作都由程序员显式地完成。毫无疑问，这是一项非常艰巨的任务！这确实导致了很多缺陷。但如果小心并加以注意，就会正确地使用这些接口，没有太多的麻烦。下面的例子展示了如何在堆上分配一个整数，得到指向它的指针：

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

关于这一小段代码有两点说明。首先，你可能会注意到栈和堆的分配都发生在这一行：首先编译器看到指针的声明（int * x）时，知道为一个整型指针分配空间，随后，当程序调

① 实际上，我们希望所有章节都简明扼要！但我们认为，本章更简明、更扼要。

用 malloc()时，它会在堆上请求整数的空间，函数返回这样一个整数的地址（成功时，失败时则返回 NULL），然后将其存储在栈中以供程序使用。

因为它的显式特性，以及它更富于变化的用法，堆内存对用户和系统提出了更大的挑战。所以这也是我们接下来讨论的重点。

14.2 malloc()调用

malloc 函数非常简单：传入要申请的堆空间的大小，它成功就返回一个指向新申请空间的指针，失败就返回 NULL[①]。

man 手册展示了使用 malloc 需要怎么做，在命令行输入 man malloc，你会看到：

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

从这段信息可以看到，只需要包含头文件 stdlib.h 就可以使用 malloc 了。但实际上，甚至都不需这样做，因为 C 库是 C 程序默认链接的，其中就有 mallock()的代码，加上这个头文件只是让编译器检查你是否正确调用了 malloc()（即传入参数的数目正确且类型正确）。

malloc 只需要一个 size_t 类型参数，该参数表示你需要多少个字节。然而，大多数程序员并不会直接传入数字（比如 10）。实际上，这样做会被认为是不太好的形式。替代方案是使用各种函数和宏。例如，为了给双精度浮点数分配空间，只要这样：

```
double *d = (double *) malloc(sizeof(double));
```

提示：如果困惑，动手试试

如果你不确定要用的一些函数或者操作符的行为，唯一的办法就是试一下，确保它的行为符合你的期望。虽然读手册或其他文档是有用的，但在实际中如何使用更为重要。实际上，我们正是通过这样做，来确保关于 sizeof()我们所说的都是真的！

啊，好多 double！对 malloc()的调用使用 sizeof() 操作符去申请正确大小的空间。在 C 中，这通常被认为是编译时操作符，意味着这个大小是在编译时就已知道，因此被替换成一个数（在本例中是 8，对于 double），作为 malloc()的参数。出于这个原因，sizeof() 被正确地认为是一个操作符，而不是一个函数调用（函数调用在运行时发生）。

你也可以传入一个变量的名字（而不只是类型）给 sizeof()，但在一些情况下，可能得不到你要的结果，所以要小心使用。例如，看看下面的代码片段：

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

在第一行，我们为 10 个整数的数组声明了空间，这很好，很漂亮。但是，当我们在下一行使用 sizeof()时，它将返回一个较小的值，例如 4（在 32 位计算机上）或 8（在 64 位计

① 请注意，C 中的 NULL 实际上并不是什么特别的东西，只是一个值为 0 的宏。

算机上）。原因是在这种情况下，sizeof()认为我们只是问一个整数的指针有多大，而不是我们动态分配了多少内存。但是，有时 sizeof()的确如你所期望的那样工作：

```
int x[10];
printf("%d\n", sizeof(x));
```

在这种情况下，编译器有足够的静态信息，知道已经分配了 40 个字节。

另一个需要注意的地方是使用字符串。如果为一个字符串声明空间，请使用以下习惯用法：malloc(strlen(s) + 1)，它使用函数 strlen()获取字符串的长度，并加上 1，以便为字符串结束符留出空间。这里使用 sizeof()可能会导致麻烦。

你也许还注意到 malloc()返回一个指向 void 类型的指针。这样做只是 C 中传回地址的方式，让程序员决定如何处理它。程序员将进一步使用所谓的强制类型转换（cast），在我们上面的示例中，程序员将返回类型的 malloc()强制转换为指向 double 的指针。强制类型转换实际上没干什么事，只是告诉编译器和其他可能正在读你的代码的程序员：“是的，我知道我在做什么。”通过强制转换 malloc()的结果，程序员只是在给人一些信心，强制转换不是程序正确所必须的。

14.3 free()调用

事实证明，分配内存是等式的简单部分。知道何时、如何以及是否释放内存是困难的部分。要释放不再使用的堆内存，程序员只需调用 free()：

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

该函数接受一个参数，即一个由 malloc()返回的指针。

因此，你可能会注意到，分配区域的大小不会被用户传入，必须由内存分配库本身记录追踪。

14.4 常见错误

在使用 malloc()和 free()时会出现一些常见的错误。以下是我们在教授本科操作系统课程时反复看到的情形。所有这些例子都可以通过编译器的编译并运行。对于构建一个正确的 C 程序来说，通过编译是必要的，但这远远不够，你会懂的（通常在吃了很多苦头之后）。

实际上，正确的内存管理就是这样一个问题，许多新语言都支持自动内存管理（automatic memory management）。在这样的语言中，当你调用类似 malloc()的机制来分配内存时（通常用 new 或类似的东西来分配一个新对象），你永远不需要调用某些东西来释放空间。实际上，垃圾收集器（garbage collector）会运行，找出你不再引用的内存，替你释放它。

忘记分配内存

许多例程在调用之前，都希望你为它们分配内存。例如，例程 strcpy(dst, src)将源字符串中的字符串复制到目标指针。但是，如果不小心，你可能会这样做：

```
char *src = "hello";
char *dst;        // oops! unallocated
strcpy(dst, src); // segfault and die
```

运行这段代码时，可能会导致段错误（segmentation fault）[①]，这是一个很奇怪的术语，表示“你对内存犯了一个错误。你这个愚蠢的程序员。我很生气。”

提示：它编译过了或它运行了!=它对了

仅仅因为程序编译过了甚至正确运行了一次或多次，并不意味着程序是正确的。许多事件可能会让你相信它能工作，但是之后有些事情会发生变化，它停止了。学生常见的反应是说（或者叫喊）“但它以前是好的!”，然后责怪编译器、操作系统、硬件，甚至是（我们敢说）教授。但是，问题通常就像你认为的那样，在你的代码中。在指责别人之前，先撸起袖子调试一下。

在这个例子中，正确的代码可能像这样：

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

或者你可以用 strdup()，让生活更加轻松。阅读 strdup 的 man 手册页，了解更多信息。

没有分配足够的内存

另一个相关的错误是没有分配足够的内存，有时称为缓冲区溢出（buffer overflow）。在上面的例子中，一个常见的错误是为目标缓冲区留出“几乎”足够的空间。

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

奇怪的是，这个程序通常看起来会正确运行，这取决于如何实现 malloc 和许多其他细节。在某些情况下，当字符串拷贝执行时，它会在超过分配空间的末尾处写入一个字节，但在某些情况下，这是无害的，可能会覆盖不再使用的变量。在某些情况下，这些溢出可能具有令人难以置信的危害，实际上是系统中许多安全漏洞的来源[W06]。在其他情况下，malloc 库总是分配一些额外的空间，因此你的程序实际上不会在其他某个变量的值上涂写，并且工作得很好。还有一些情况下，该程序确实会发生故障和崩溃。因此，我们学到了另一个宝贵的教训：即使它正确运行过一次，也不意味着它是正确的。

① 尽管听起来很神秘，但你很快就会明白为什么这种非法的内存访问被称为段错误。如果这都不能刺激你继续读下去，那什么能呢？

忘记初始化分配的内存

在这个错误中，你正确地调用 malloc()，但忘记在新分配的数据类型中填写一些值。不要这样做！如果你忘记了，你的程序最终会遇到未初始化的读取（uninitialized read），它从堆中读取了一些未知值的数据。谁知道那里可能会有什么？如果走运，读到的值使程序仍然有效（例如，零）。如果不走运，会读到一些随机和有害的东西。

忘记释放内存

另一个常见错误称为内存泄露（memory leak），如果忘记释放内存，就会发生。在长时间运行的应用程序或系统（如操作系统本身）中，这是一个巨大的问题，因为缓慢泄露的内存会导致内存不足，此时需要重新启动。因此，一般来说，当你用完一段内存时，应该确保释放它。请注意，使用垃圾收集语言在这里没有什么帮助：如果你仍然拥有对某块内存的引用，那么垃圾收集器就不会释放它，因此即使在较现代的语言中，内存泄露仍然是一个问题。

在某些情况下，不调用 free()似乎是合理的。例如，你的程序运行时间很短，很快就会退出。在这种情况下，当进程死亡时，操作系统将清理其分配的所有页面，因此不会发生内存泄露。虽然这肯定“有效”（请参阅后面的补充），但这可能是一个坏习惯，所以请谨慎选择这样的策略。长远来看，作为程序员的目标之一是养成良好的习惯。其中一个习惯是理解如何管理内存，并在 C 这样的语言中，释放分配的内存块。即使你不这样做也可以逃脱惩罚，建议还是养成习惯，释放显式分配的每个字节。

在用完之前释放内存

有时候程序会在用完之前释放内存，这种错误称为悬挂指针（dangling pointer），正如你猜测的那样，这也是一件坏事。随后的使用可能会导致程序崩溃或覆盖有效的内存（例如，你调用了 free()，但随后再次调用 malloc()来分配其他内容，这重新利用了错误释放的内存）。

反复释放内存

程序有时还会不止一次地释放内存，这被称为重复释放（double free）。这样做的结果是未定义的。正如你所能想象的那样，内存分配库可能会感到困惑，并且会做各种奇怪的事情，崩溃是常见的结果。

错误地调用 free()

我们讨论的最后一个问题是 free()的调用错误。毕竟，free()期望你只传入之前从 malloc()得到的一个指针。如果传入一些其他的值，坏事就可能发生（并且会发生）。因此，这种无效的释放（invalid free）是危险的，当然也应该避免。

补充：为什么在你的进程退出时没有内存泄露

当你编写一个短时间运行的程序时，可能会使用 malloc()分配一些空间。程序运行并即将完成：是否需要在退出前调用几次 free()？虽然不释放似乎不对，但在真正的意义上，没有任何内存会“丢失”。原因很简单：系统中实际存在两级内存管理。

第一级是由操作系统执行的内存管理，操作系统在进程运行时将内存交给进程，并在进程退出（或以其他方式结束）时将其回收。第二级管理在每个进程中，例如在调用 malloc()和 free()时，在堆内管理。即使你没有调用 free()（并因此泄露了堆中的内存），操作系统也会在程序结束运行时，收回进程的所有内存（包括用于代码、栈，以及相关堆的内存页）。无论地址空间中堆的状态如何，操作系统都会在进程终止时收回所有这些页面，从而确保即使没有释放内存，也不会丢失内存。

因此，对于短时间运行的程序，泄露内存通常不会导致任何操作问题（尽管它可能被认为是不好的形式）。如果你编写一个长期运行的服务器（例如 Web 服务器或数据库管理系统，它永远不会退出），泄露内存就是很大的问题，最终会导致应用程序在内存不足时崩溃。当然，在某个程序内部泄露内存是一个更大的问题：操作系统本身。这再次向我们展示：编写内核代码的人，工作是辛苦的……

小结

如你所见，有很多方法滥用内存。由于内存出错很常见，整个工具生态圈已经开发出来，可以帮助你在代码中找到这些问题。请查看 purify [HJ92]和 valgrind [SN05]，在帮助你找到与内存有关的问题的根源方面，两者都非常出色。一旦你习惯于使用这些强大的工具，就会想知道，没有它们时，你是如何活下来的。

14.5 底层操作系统支持

你可能已经注意到，在讨论 malloc()和 free()时，我们没有讨论系统调用。原因很简单：它们不是系统调用，而是库调用。因此，malloc 库管理虚拟地址空间内的空间，但是它本身是建立在一些系统调用之上的，这些系统调用会进入操作系统，来请求更多内存或者将一些内容释放回系统。

一个这样的系统调用叫作 brk，它被用来改变程序分断（break）的位置：堆结束的位置。它需要一个参数（新分断的地址），从而根据新分断是大于还是小于当前分断，来增加或减小堆的大小。另一个调用 sbrk 要求传入一个增量，但目的是类似的。

请注意，你不应该直接调用 brk 或 sbrk。它们被内存分配库使用。如果你尝试使用它们，很可能会犯一些错误。建议坚持使用 malloc()和 free()。

最后，你还可以通过 mmap()调用从操作系统获取内存。通过传入正确的参数，mmap()可以在程序中创建一个匿名（anonymous）内存区域——这个区域不与任何特定文件相关联，而是与交换空间（swap space）相关联，稍后我们将在虚拟内存中详细讨论。这种内存也可以像堆一样对待并管理。阅读 mmap()的手册页以获取更多详细信息。

14.6 其他调用

内存分配库还支持一些其他调用。例如，calloc()分配内存，并在返回之前将其置零。如果你认为内存已归零并忘记自己初始化它，这可以防止出现一些错误（请参阅 14.4 节中“忘记初始化分配的内存”的内容）。当你为某些东西（比如一个数组）分配空间，然后需要添加一些东西时，例程 realloc()也会很有用：realloc()创建一个新的更大的内存区域，将旧区域复制到其中，并返回新区域的指针。

14.7 小结

我们介绍了一些处理内存分配的 API。与往常一样，我们只介绍了基本知识。更多细节可在其他地方获得。请阅读 C 语言的书[KR88]和 Stevens [SR05]（第 7 章）以获取更多信息。有关如何自动检测和纠正这些问题的很酷的现代论文，请参阅 Novark 等人的论文[N+07]。这篇文章还包含了对常见问题的很好的总结，以及关于如何查找和修复它们的一些简洁办法。

参考资料

[HJ92] Purify: Fast Detection of Memory Leaks and Access Errors
R. Hastings and B. Joyce USENIX Winter ’92
很酷的 Purify 工具背后的文章。Purify 现在是商业产品。

[KR88]“The C Programming Language”Brian Kernighan and Dennis Ritchie Prentice-Hall 1988
C 之书，由 C 的开发者编写。读一遍，编一些程序，然后再读一遍，让它成为你的案头手册。

[N+07]“Exterminator: Automatically Correcting Memory Errors with High Probability”Gene Novark, Emery D. Berger, and Benjamin G. Zorn
PLDI 2007
一篇很酷的文章，包含自动查找和纠正内存错误，以及 C 和 C ++程序中许多常见错误的概述。

[SN05]“Using Valgrind to Detect Undefined Value Errors with Bit-precision”
J. Seward and N. Nethercote USENIX ’05
如何使用 valgrind 来查找某些类型的错误。

[SR05]“Advanced Programming in the UNIX Environment”
W. Richard Stevens and Stephen A. Rago Addison-Wesley, 2005
我们之前已经说过了，这里再重申一遍：读这本书很多遍，并在有疑问时将其用作参考。本书的两位作者

总是很惊讶，每次读这本书时都会学到一些新东西，即使具有多年的 C 语言编程经验的程序员。

[W06]“Survey on Buffer Overflow Attacks and Countermeasures”Tim Werthman
一份很好的调查报告，关于缓冲区溢出及其造成的一些安全问题。文中指出了许多著名的漏洞。

作业（编码）

在这个作业中，你会对内存分配有所了解。首先，你会写一些错误的程序（好玩！）。然后，利用一些工具来帮助你找到其中的错误。最后，你会意识到这些工具有多棒，并在将来使用它们，从而使你更加快乐和高效。

你要使用的第一个工具是调试器 gdb。关于这个调试器有很多需要了解的知识，在这里，我们只是浅尝辄止。

你要使用的第二个工具是 valgrind [SN05]。该工具可以帮助查找程序中的内存泄露和其他隐藏的内存问题。如果你的系统上没有安装，请访问 valgrind 网站并安装它。

问题

1．首先，编写一个名为 null.c 的简单程序，它创建一个指向整数的指针，将其设置为 NULL，然后尝试对其进行释放内存操作。把它编译成一个名为 null 的可执行文件。当你运行这个程序时会发生什么？

2．接下来，编译该程序，其中包含符号信息（使用-g 标志）。这样做可以将更多信息放入可执行文件中，使调试器可以访问有关变量名称等的更多有用信息。通过输入 gdb null，在调试器下运行该程序，然后，一旦 gdb 运行，输入 run。gdb 显示什么信息？

3．最后，对这个程序使用 valgrind 工具。我们将使用属于 valgrind 的 memcheck 工具来分析发生的情况。输入以下命令来运行程序：valgrind --leak-check=yes null。当你运行它时会发生什么？你能解释工具的输出吗？

4．编写一个使用 malloc()来分配内存的简单程序，但在退出之前忘记释放它。这个程序运行时会发生什么？你可以用 gdb 来查找它的任何问题吗？用 valgrind 呢（再次使用 --leak-check=yes 标志）？

5．编写一个程序，使用 malloc 创建一个名为 data、大小为 100 的整数数组。然后，将 data[100]设置为 0。当你运行这个程序时会发生什么？当你使用 valgrind 运行这个程序时会发生什么？程序是否正确？

6．创建一个分配整数数组的程序（如上所述），释放它们，然后尝试打印数组中某个元素的值。程序会运行吗？当你使用 valgrind 时会发生什么？

7．现在传递一个有趣的值来释放（例如，在上面分配的数组中间的一个指针）。会发生什么？你是否需要工具来找到这种类型的问题？

8．尝试一些其他接口来分配内存。例如，创建一个简单的向量似的数据结构，以及使用 realloc()来管理向量的相关函数。使用数组来存储向量元素。当用户在向量中添加条目时，请使用realloc()为其分配更多空间。这样的向量表现如何？它与链表相比如何？使用valgrind来帮助你发现错误。

9．花更多时间阅读有关使用 gdb 和 valgrind 的信息。了解你的工具至关重要，花时间学习如何成为 UNIX 和 C 环境中的调试器专家。

第 15 章　机制：地址转换

在实现 CPU 虚拟化时，我们遵循的一般准则被称为受限直接访问（Limited Direct Execution，LDE）。LDE 背后的想法很简单：让程序运行的大部分指令直接访问硬件，只在一些关键点（如进程发起系统调用或发生时钟中断）由操作系统介入来确保“在正确时间，正确的地点，做正确的事”。为了实现高效的虚拟化，操作系统应该尽量让程序自己运行，同时通过在关键点的及时介入（interposing），来保持对硬件的控制。高效和控制是现代操作系统的两个主要目标。

在实现虚拟内存时，我们将追求类似的战略，在实现高效和控制的同时，提供期望的虚拟化。高效决定了我们要利用硬件的支持，这在开始的时候非常初级（如使用一些寄存器），但会变得相当复杂（比如我们会讲到的 TLB、页表等）。控制意味着操作系统要确保应用程序只能访问它自己的内存空间。因此，要保护应用程序不会相互影响，也不会影响操作系统，我们需要硬件的帮助。最后，我们对虚拟内存还有一点要求，即灵活性。具体来说，我们希望程序能以任何方式访问它自己的地址空间，从而让系统更容易编程。所以，关键问题在于：

关键问题：如何高效、灵活地虚拟化内存

如何实现高效的内存虚拟化？如何提供应用程序所需的灵活性？如何保持控制应用程序可访问的内存位置，从而确保应用程序的内存访问受到合理的限制？如何高效地实现这一切？

我们利用了一种通用技术，有时被称为基于硬件的地址转换（hardware-based address translation），简称为地址转换（address translation）。它可以看成是受限直接执行这种一般方法的补充。利用地址转换，硬件对每次内存访问进行处理（即指令获取、数据读取或写入），将指令中的虚拟（virtual）地址转换为数据实际存储的物理（physical）地址。因此，在每次内存引用时，硬件都会进行地址转换，将应用程序的内存引用重定位到内存中实际的位置。

当然，仅仅依靠硬件不足以实现虚拟内存，因为它只是提供了底层机制来提高效率。操作系统必须在关键的位置介入，设置好硬件，以便完成正确的地址转换。因此它必须管理内存（manage memory），记录被占用和空闲的内存位置，并明智而谨慎地介入，保持对内存使用的控制。

同样，所有这些工作都是为了创造一种美丽的假象：每个程序都拥有私有的内存，那里存放着它自己的代码和数据。虚拟现实的背后是丑陋的物理事实：许多程序其实是在同一时间共享着内存，就像 CPU（或多个 CPU）在不同的程序间切换运行。通过虚拟化，操作系统（在硬件的帮助下）将丑陋的机器现实转化成一种有用的、强大的、易于使用的抽象。

15.1 假设

我们对内存虚拟化的第一次尝试非常简单，甚至有点可笑。如果你觉得可笑就笑吧，很快就轮到操作系统嘲笑你了。当你试图理解 TLB 的换入换出、多级页表，和其他技术一样有奇迹之处的时候。不喜欢操作系统嘲笑你？很不幸，但这就是操作系统的运行方式。

具体来说，我们先假设用户的地址空间必须连续地放在物理内存中。同时，为了简单，我们假设地址空间不是很大，具体来说，小于物理内存的大小。最后，假设每个地址空间的大小完全一样。别担心这些假设听起来不切实际，我们会逐步地放宽这些假设，从而得到现实的内存虚拟化。

15.2 一个例子

为了更好地理解实现地址转换需要什么，以及为什么需要，我们先来看一个简单的例子。设想一个进程的地址空间如图 15.1 所示。这里我们要检查一小段代码，它从内存中加载一个值，对它加 3，然后将它存回内存。你可以设想，这段代码的 C 语言形式可能像这样：

```
void func() {
    int x;
    x = x + 3; // this is the line of code we are interested in
```

编译器将这行代码转化为汇编语句，可能像下面这样（x86 汇编）。我们可以用 Linux 的 objdump 或者 Mac 的 otool 将它反汇编：

```
128: movl 0x0(%ebx), %eax   ;load 0+ebx into eax
132: addl $0x03, %eax       ;add 3 to eax register
135: movl %eax, 0x0(%ebx)   ;store eax back to mem
```

这段代码相对简单，它假定 *x* 的地址已经存入寄存器 ebx，之后通过 movl 指令将这个地址的值加载到通用寄存器 eax（长字移动）。下一条指令对 eax 的内容加 3。最后一条指令将 eax 中的值写回到内存的同一位置。

提示：介入（Interposition）很强大

介入是一种很常见又很有用的技术，计算机系统中使用介入常常能带来很好的效果。在虚拟内存中，硬件可以介入到每次内存访问中，将进程提供的虚拟地址转换为数据实际存储的物理地址。但是，一般化的介入技术有更广阔的应用空间，实际上几乎所有良好定义的接口都应该提供功能介入机制，以便增加功能或者在其他方面提升系统。这种方式最基本的优点是透明（transparency），介入完成时通常不需要改动接口的客户端，因此客户端不需要任何改动。

在图 15.1 中，可以看到代码和数据都位于进程的地址空间，3 条指令序列位于地址 128（靠近头部的代码段），变量 *x* 的值位于地址 15KB（在靠近底部的栈中）。如图 15.1 所示，*x* 的初始值是 3000。

如果这 3 条指令执行，从进程的角度来看，发生了以下几次内存访问：

- 从地址 128 获取指令；
- 执行指令（从地址 15KB 加载数据）；
- 从地址 132 获取命令；
- 执行命令（没有内存访问）；
- 从地址 135 获取指令；
- 执行指令（新值存入地址 15KB）。

从程序的角度来看，它的地址空间（address space）从 0 开始到 16KB 结束。它包含的所有内存引用都应该在这个范围内。然而，对虚拟内存来说，操作系统希望将这个进程地址空间放在物理内存的其他位置，并不一定从地址 0 开始。因此我们遇到了如下问题：怎样在内存中重定位这个进程，同时对该进程透明（transparent）？怎么样提供一种虚拟地址空间从 0 开始的假象，而实际上地址空间位于另外某个物理地址？

图 15.2 展示了一个例子，说明这个进程的地址空间被放入物理内存后可能的样子。从图 15.2 中可以看到，操作系统将第一块物理内存留给了自己，并将上述例子中的进程地址空间重定位到从 32KB 开始的物理内存地址。剩下的两块内存空闲（16～32KB 和 48～64KB）。

图 15.1 进程及其地址空间

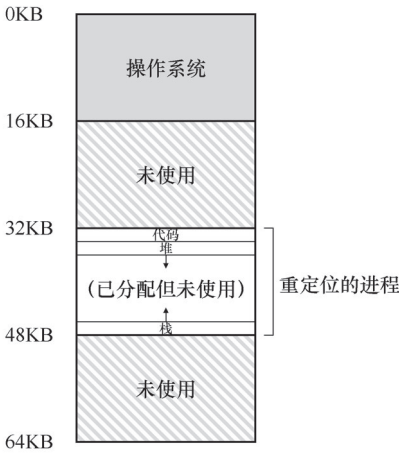


图 15.2 物理内存和单个重定位的进程

15.3 动态（基于硬件）重定位

为了更好地理解基于硬件的地址转换，我们先来讨论它的第一次应用。在 20 世纪 50 年代后期，它在首次出现的时分机器中引入，那时只是一个简单的思想，称为基址加界限机制（base and bound），有时又称为动态重定位（dynamic relocation），我们将互换使用这两个术语[SS74]。

具体来说，每个 CPU 需要两个硬件寄存器：基址（base）寄存器和界限（bound）寄存器，有时称为限制（limit）寄存器。这组基址和界限寄存器，让我们能够将地址空间放在物理内存的任何位置，同时又能确保进程只能访问自己的地址空间。

采用这种方式，在编写和编译程序时假设地址空间从零开始。但是，当程序真正执行时，操作系统会决定其在物理内存中的实际加载地址，并将起始地址记录在基址寄存器中。在上面的例子中，操作系统决定加载在物理地址 32KB 的进程，因此将基址寄存器设置为这个值。

当进程运行时，有趣的事情发生了。现在，该进程产生的所有内存引用，都会被处理器通过以下方式转换为物理地址：

```
physical address = virtual address + base
```

补充：基于软件的重定位

在早期，在硬件支持重定位之前，一些系统曾经采用纯软件的重定位方式。基本技术被称为静态重定位（static relocation），其中一个名为加载程序（loader）的软件接手将要运行的可执行程序，将它的地址重写到物理内存中期望的偏移位置。

例如，程序中有一条指令是从地址 1000 加载到寄存器（即 movl 1000，%eax），当整个程序的地址空间被加载到从 3000（不是程序认为的 0）开始的物理地址中，加载程序会重写指令中的地址（即 movl 4000, %eax），从而完成简单的静态重定位。

然而，静态重定位有许多问题，首先也是最重要的是不提供访问保护，进程中的错误地址可能导致对其他进程或操作系统内存的非法访问，一般来说，需要硬件支持来实现真正的访问保护[WL+93]。静态重定位的另一个缺点是一旦完成，稍后很难将内存空间重定位到其他位置 [M65]。

进程中使用的内存引用都是虚拟地址（virtual address），硬件接下来将虚拟地址加上基址寄存器中的内容，得到物理地址（physical address），再发给内存系统。

为了更好地理解，让我们追踪一条指令执行的情况。具体来看前面序列中的一条指令：

```
128: movl 0x0(%ebx), %eax
```

程序计数器（PC）首先被设置为 128。当硬件需要获取这条指令时，它先将这个值加上基址寄存器中的 32KB(32768)，得到实际的物理地址 32896，然后硬件从这个物理地址获取指令。接下来，处理器开始执行该指令。这时，进程发起从虚拟地址 15KB 的加载，处理器同样将虚拟地址加上基址寄存器内容（32KB），得到最终的物理地址 47KB，从而获得需要的数据。

将虚拟地址转换为物理地址，这正是所谓的地址转换（address translation）技术。也就是说，硬件取得进程认为它要访问的地址，将它转换成数据实际位于的物理地址。由于这

种重定位是在运行时发生的，而且我们甚至可以在进程开始运行后改变其地址空间，这种技术一般被称为动态重定位（dynamic relocation）[M65]。

提示：基于硬件的动态重定位

在动态重定位的过程中，只有很少的硬件参与，但获得了很好的效果。一个基址寄存器将虚拟地址转换为物理地址，一个界限寄存器确保这个地址在进程地址空间的范围内。它们一起提供了既简单又高效的虚拟内存机制。

现在你可能会问，界限（限制）寄存器去哪了？不是基址加界限机制吗？正如你猜测的那样，界限寄存器提供了访问保护。在上面的例子中，界限寄存器被置为 16KB。如果进程需要访问超过这个界限或者为负数的虚拟地址，CPU 将触发异常，进程最终可能被终止。界限寄存器的用处在于，它确保了进程产生的所有地址都在进程的地址“界限”中。

这种基址寄存器配合界限寄存器的硬件结构是芯片中的（每个 CPU 一对）。有时我们将 CPU 的这个负责地址转换的部分统称为内存管理单元（Memory Management Unit，MMU）。随着我们开发更复杂的内存管理技术，MMU 也将有更复杂的电路和功能。

关于界限寄存器再补充一点，它通常有两种使用方式。在一种方式中（像上面那样），它记录地址空间的大小，硬件在将虚拟地址与基址寄存器内容求和前，就检查这个界限。另一种方式是界限寄存器中记录地址空间结束的物理地址，硬件在转化虚拟地址到物理地址之后才去检查这个界限。这两种方式在逻辑上是等价的。简单起见，我们这里假设采用第一种方式。

转换示例

为了更好地理解基址加界限的地址转换的详细过程，我们来看一个例子。设想一个进程拥有 4KB 大小地址空间（是的，小得不切实际），它被加载到从 16KB 开始的物理内存中。一些地址转换结果见表 15.1。

表 15.1 地址转换结果

虚拟地址		物理地址
0	→	16KB
1KB	→	17KB
3000	→	19384
4400	→	错误（越界）

从例子中可以看到，通过基址加虚拟地址（可以看作是地址空间的偏移量）的方式，很容易得到物理地址。虚拟地址“过大”或者为负数时，会导致异常。

补充：数据结构——空闲列表

操作系统必须记录哪些空闲内存没有使用，以便能够为进程分配内存。很多不同的数据结构可以用于这项任务，其中最简单的（也是我们假定在这里采用的）是空闲列表（free list）。它就是一个列表，记录当前没有使用的物理内存的范围。

15.4 硬件支持：总结

我们来总结一下需要的硬件支持（见表 15.2）。首先，正如在 CPU 虚拟化的章节中提到的，我们需要两种 CPU 模式。操作系统在特权模式（privileged mode，或内核模式，kernel mode），可以访问整个机器资源。应用程序在用户模式（user mode）运行，只能做有限的操作。只要一个位，也许保存在处理器状态字（processor status word）中，就能说明当前的 CPU 运行模式。在一些特殊的时刻（如系统调用、异常或中断），CPU 会切换状态。

表 15.2 动态重定位：硬件要求

硬件要求	解释
特权模式	需要，以防用户模式的进程执行特权操作
基址/界限寄存器	每个 CPU 需要一对寄存器来支持地址转换和界限检查
能够转换虚拟地址并检查它是否越界	电路来完成转换和检查界限，在这种情况下，非常简单
修改基址/界限寄存器的特权指令	在让用户程序运行之前，操作系统必须能够设置这些值
注册异常处理程序的特权指令	操作系统必须能告诉硬件，如果异常发生，那么执行哪些代码
能够触发异常	如果进程试图使用特权指令或越界的内存

硬件还必须提供基址和界限寄存器（base and bounds register），因此每个 CPU 的内存管理单元（Memory Management Unit，MMU）都需要这两个额外的寄存器。用户程序运行时，硬件会转换每个地址，即将用户程序产生的虚拟地址加上基址寄存器的内容。硬件也必须能检查地址是否有用，通过界限寄存器和 CPU 内的一些电路来实现。

硬件应该提供一些特殊的指令，用于修改基址寄存器和界限寄存器，允许操作系统在切换进程时改变它们。这些指令是特权（privileged）指令，只有在内核模式下，才能修改这些寄存器。想象一下，如果用户进程在运行时可以随意更改基址寄存器，那么用户进程可能会造成严重破坏[①]。想象一下吧！然后迅速将这些阴暗的想法从你的头脑中赶走，因为它们很可怕，会导致噩梦。

最后，在用户程序尝试非法访问内存（越界访问）时，CPU 必须能够产生异常（exception）。在这种情况下，CPU 应该阻止用户程序的执行，并安排操作系统的“越界”异常处理程序（exception handler）去处理。操作系统的处理程序会做出正确的响应，比如在这种情况下终止进程。类似地，如果用户程序尝试修改基址或者界限寄存器时，CPU 也应该产生异常，并调用“用户模式尝试执行特权指令”的异常处理程序。CPU 还必须提供一种方法，来通知它这些处理程序的位置，因此又需要另一些特权指令。

15.5 操作系统的问题

为了支持动态重定位，硬件添加了新的功能，使得操作系统有了一些必须处理的新问

① 除了“严重破坏（havoc）”还有什么可以“造成（wreaked）”的吗？

题。硬件支持和操作系统管理结合在一起，实现了一个简单的虚拟内存。具体来说，在一些关键的时刻操作系统需要介入，以实现基址和界限方式的虚拟内存，见表 15.3。

第一，在进程创建时，操作系统必须采取行动，为进程的地址空间找到内存空间。由于我们假设每个进程的地址空间小于物理内存的大小，并且大小相同，这对操作系统来说很容易。它可以把整个物理内存看作一组槽块，标记了空闲或已用。当新进程创建时，操作系统检索这个数据结构（常被称为空闲列表，free list），为新地址空间找到位置，并将其标记为已用。如果地址空间可变，那么生活就会更复杂，我们将在后续章节中讨论。

我们来看一个例子。在图 15.2 中，操作系统将物理内存的第一个槽块分配给自己，然后将例子中的进程重定位到物理内存地址 32KB。另两个槽块（16～32KB，48～64KB）空闲，因此空闲列表（free list）就包含这两个槽块。

第二，在进程终止时（正常退出，或因行为不端被强制终止），操作系统也必须做一些工作，回收它的所有内存，给其他进程或者操作系统使用。在进程终止时，操作系统会将这些内存放回到空闲列表，并根据需要清除相关的数据结构。

第三，在上下文切换时，操作系统也必须执行一些额外的操作。每个 CPU 毕竟只有一个基址寄存器和一个界限寄存器，但对于每个运行的程序，它们的值都不同，因为每个程序被加载到内存中不同的物理地址。因此，在切换进程时，操作系统必须保存和恢复基础和界限寄存器。具体来说，当操作系统决定中止当前的运行进程时，它必须将当前基址和界限寄存器中的内容保存在内存中，放在某种每个进程都有的结构中，如进程结构（process structure）或进程控制块（Process Control Block，PCB）中。类似地，当操作系统恢复执行某个进程时（或第一次执行），也必须给基址和界限寄存器设置正确的值。

表 15.3 动态重定位：操作系统的职责

操作系统的要求	解释
内存管理	需要为新进程分配内存 从终止的进程回收内存 一般通过空闲列表（free list）来管理内存
基址/界限管理	必须在上下文切换时正确设置基址/界限寄存器
异常处理	当异常发生时执行的代码，可能的动作是终止犯错的进程

需要注意，当进程停止时（即没有运行），操作系统可以改变其地址空间的物理位置，这很容易。要移动进程的地址空间，操作系统首先让进程停止运行，然后将地址空间拷贝到新位置，最后更新保存的基址寄存器（在进程结构中），指向新位置。当该进程恢复执行时，它的（新）基址寄存器会被恢复，它再次开始运行，显然它的指令和数据都在新的内存位置了。

第四，操作系统必须提供异常处理程序（exception handler），或要一些调用的函数，像上面提到的那样。操作系统在启动时加载这些处理程序（通过特权命令）。例如，当一个进程试图越界访问内存时，CPU 会触发异常。在这种异常产生时，操作系统必须准备采取行动。通常操作系统会做出充满敌意的反应：终止错误进程。操作系统应该尽力保护它运行的机器，因此它不会对那些企图访问非法地址或执行非法指令的进程客气。再见了，行为不端的进程，很高兴认识你。

表 15.4 为按时间线展示了大多数硬件与操作系统的交互。可以看出，操作系统在启动时

做了什么，为我们准备好机器，然后在进程（进程 A）开始运行时发生了什么。请注意，地址转换过程完全由硬件处理，没有操作系统的介入。在这个时候，发生时钟中断，操作系统切换到进程 B 运行，它执行了“错误的加载”（对一个非法内存地址），这时操作系统必须介入，终止该进程，清理并释放进程 B 占用的内存，将它从进程表中移除。从表中可以看出，我们仍然遵循受限直接访问（limited direct execution）的基本方法，大多数情况下，操作系统正确设置硬件后，就任凭进程直接运行在 CPU 上，只有进程行为不端时才介入。

表 15.4　　受限直接执行协议（动态重定位）

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住以下地址： 　系统调用处理程序 　时钟处理程序 　非法内存处理程序 　非常指令处理程序	
开始中断时钟		
	开始时钟，在 *x* ms 后中断	
初始化进程表 初始化空闲列表		
操作系统@运行（核心模式）	硬件	程序（用户模式）
为了启动进程 A： 　在进程表中分配条目 　为进程分配内存 　设置基址/界限寄存器 　从陷阱返回（进入 A）		
	恢复 A 的寄存器 转向用户模式 跳到 A（最初）的程序计数器	
		进程 A 运行 　获取指令
	转换虚拟地址并执行获取	
		执行指令
	如果显式加载/保存 　确保地址不越界 　转换虚拟地址并执行 　加载/保存	
		……
	时钟中断 转向内核模式 跳到中断处理程序	

续表

操作系统@启动（内核模式）	硬件	
处理陷阱 调用 switch()例程 将寄存器（A）保存到进程结构（A） （包括基址/界限） 从进程结构（B）恢复寄存器（B） （包括基址/界限） 从陷阱返回（进入 B）		
	恢复 B 的寄存器 转向用户模式 跳到 B 的程序计数器	
		进程 B 运行 执行错误的加载
	加载越界 转向内核模式 跳到陷阱处理程序	
处理本期报告 决定终止进程 B 回收 B 的内存 移除 B 在进程表中的条目		

15.6 小结

本章通过虚拟内存使用的一种特殊机制，即地址转换（address translation），扩展了受限直接访问的概念。利用地址转换，操作系统可以控制进程的所有内存访问，确保访问在地址空间的界限内。这个技术高效的关键是硬件支持，硬件快速地将所有内存访问操作中的虚拟地址（进程自己看到的内存位置）转换为物理地址（实际位置）。所有的这一切对进程来说都是透明的，进程并不知道自己使用的内存引用已经被重定位，制造了美妙的假象。

我们还看到了一种特殊的虚拟化方式，称为基址加界限的动态重定位。基址加界限的虚拟化方式非常高效，因为只需要很少的硬件逻辑，就可以将虚拟地址和基址寄存器加起来，并检查进程产生的地址没有越界。基址加界限也提供了保护，操作系统和硬件的协作，确保没有进程能够访问其地址空间之外的内容。保护肯定是操作系统最重要的目标之一。没有保护，操作系统不可能控制机器（如果进程可以随意修改内存，它们就可以轻松地做出可怕的事情，比如重写陷阱表并完全接管系统）。

遗憾的是，这个简单的动态重定位技术有效率低下的问题。例如，从图 15.2 中可以看到，

重定位的进程使用了从 32KB 到 48KB 的物理内存，但由于该进程的栈区和堆区并不很大，导致这块内存区域中大量的空间被浪费。这种浪费通常称为内部碎片（internal fragmentation），指的是已经分配的内存单元内部有未使用的空间（即碎片），造成了浪费。在我们当前的方式中，即使有足够的物理内存容纳更多进程，但我们目前要求将地址空间放在固定大小的槽块中，因此会出现内部碎片[①]。所以，我们需要更复杂的机制，以便更好地利用物理内存，避免内部碎片。第一次尝试是将基址加界限的概念稍稍泛化，得到分段（segmentation）的概念，我们接下来将讨论。

参考资料

[M65]“On Dynamic Program Relocation”
W.C. McGee
IBM Systems Journal
Volume 4, Number 3, 1965, pages 184–199
本文对动态重定位的早期工作和静态重定位的一些基础知识进行了很好的总结。

[P90]“Relocating loader for MS-DOS .EXE executable files”Kenneth D. A. Pillay
Microprocessors & Microsystems archive Volume 14, Issue 7 (September 1990)
MS-DOS 重定位加载器的示例。不是第一个，而只是这样的系统如何工作的一个相对现代的例子。

[SS74]“The Protection of Information in Computer Systems”
J. Saltzer and M. Schroeder CACM, July 1974
摘自这篇论文：“在 1957 年至 1959 年间，在 3 个有不同目标的项目中，显然独立出现了基址和界限寄存器和硬件解释描述符的概念。在 MIT，McCarthy 建议将基址和界限的想法作为内存保护系统的一部分，以便让时分共享可行。IBM 独立开发了基本和界限寄存器，作为 Stretch（7030）计算机系统支持可靠多道程序的机制。在 Burroughs，R. Barton 建议硬件解释描述符可以直接支持 B5000 计算机系统中高级语言的命名范围规则。”我们在 Mark Smotherman 的超酷历史页面上找到了这段引用[S04]，更多信息请参见这些页面。

[S04]“System Call Support”
Mark Smotherman, May 2004
系统调用支持的简洁历史。Smotherman 还收集了一些早期历史，包括中断和其他有趣方面的计算历史。可以查看他的网页了解更多详情。

[WL+93]“Efficient Software-based Fault Isolation”
Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham SOSP ’93
关于如何在没有硬件支持的情况下，利用编译器支持限定从程序中引用内存的一篇极好的论文。该论文引

① 另一种解决方案可能会在地址空间内放置一个固定大小的栈，位于代码区域的下方，并在栈下面让堆增长。 但是，这限制了灵活性，让递归和深层嵌套函数调用变得具有挑战，因此我们希望避免这种情况。

发了人们对用于分离内存引用的软件技术的兴趣。

作业

程序 relocation.py 让你看到，在带有基址和边界寄存器的系统中，如何执行地址转换。详情请参阅 README 文件。

问题

1．用种子 1、2 和 3 运行，并计算进程生成的每个虚拟地址是处于界限内还是界限外？如果在界限内，请计算地址转换。

2．使用以下标志运行：-s 0 -n 10。为了确保所有生成的虚拟地址都处于边界内，要将 -l（界限寄存器）设置为什么值？

3．使用以下标志运行：-s 1 -n 10 -l 100。可以设置界限的最大值是多少，以便地址空间仍然完全放在物理内存中？

4．运行和第 3 题相同的操作，但使用较大的地址空间（-a）和物理内存（-p）。

5．作为边界寄存器的值的函数，随机生成的虚拟地址的哪一部分是有效的？画一个图，使用不同随机种子运行，限制值从 0 到最大地址空间大小。

第 16 章　分段

到目前为止，我们一直假设将所有进程的地址空间完整地加载到内存中。利用基址和界限寄存器，操作系统很容易将不同进程重定位到不同的物理内存区域。但是，对于这些内存区域，你可能已经注意到一件有趣的事：栈和堆之间，有一大块“空闲”空间。

从图 16.1 中可知，如果我们将整个地址空间放入物理内存，那么栈和堆之间的空间并没有被进程使用，却依然占用了实际的物理内存。因此，简单的通过基址寄存器和界限寄存器实现的虚拟内存很浪费。另外，如果剩余物理内存无法提供连续区域来放置完整的地址空间，进程便无法运行。这种基址加界限的方式看来并不像我们期望的那样灵活。因此：

关键问题：怎样支持大地址空间

怎样支持大地址空间，同时栈和堆之间（可能）有大量空闲空间？在之前的例子里，地址空间非常小，所以这种浪费并不明显。但设想一个 32 位（4GB）的地址空间，通常的程序只会使用几兆的内存，但需要整个地址空间都放在内存中。

16.1　分段：泛化的基址/界限

为了解决这个问题，分段（segmentation）的概念应运而生。分段并不是一个新概念，它甚至可以追溯到 20 世纪 60 年代初期[H61, G62]。这个想法很简单，在 MMU 中引入不止一个基址和界限寄存器对，而是给地址空间内的每个逻辑段（segment）一对。一个段只是地址空间里的一个连续定长的区域，在典型的地址空间里有 3 个逻辑不同的段：代码、栈和堆。分段的机制使得操作系统能够将不同的段放到不同的物理内存区域，从而避免了虚拟地址空间中的未使用部分占用物理内存。

我们来看一个例子。假设我们希望将图 16.1 中的地址空间放入物理内存。通过给每个段一对基址和界限寄存器，可以将每个段独立地放入物理内存。如图 16.2 所示，64KB 的物理内存中放置了 3 个段（为操作系统保留 16KB）。

从图中可以看到，只有已用的内存才在物理内存中分配空间，因此可以容纳巨大的地址空间，其中包含大量未使用的地址空间（有时又称为稀疏地址空间，sparse address spaces）。

你会想到，需要 MMU 中的硬件结构来支持分断：在这种情况下，需要一组 3 对基址和界限寄存器。表 16.1 展示了上面的例子中的寄存器值，每个界限寄存器记录了一个段的大小。

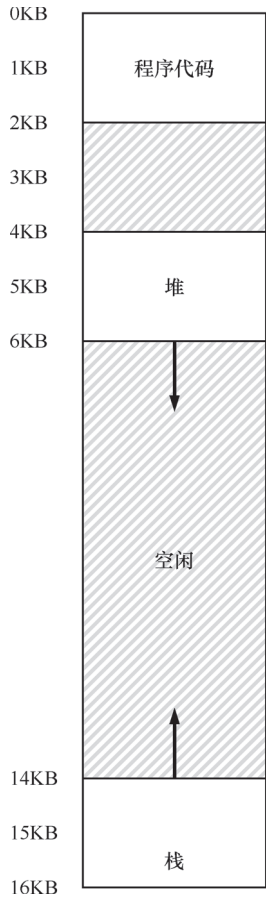


图 16.1 一个地址空间（复习）

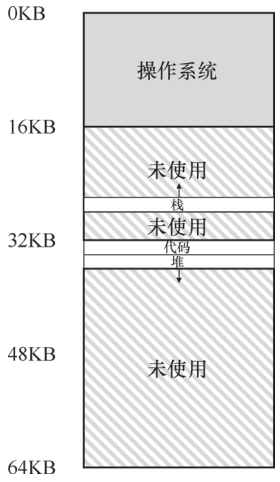


图 16.2 在物理内存中放置段

表 16.1 段寄存器的值

段	基址	大小
代码	32KB	2KB
堆	34KB	2KB
栈	28KB	2KB

如表 16.1 所示，代码段放在物理地址 32KB，大小是 2KB。堆在 34KB，大小也是 2KB。

利用图 16.1 中的地址空间，我们来看一个地址转换的例子。假设现在要引用虚拟地址 100（在代码段中），MMU 将基址值加上偏移量（100）得到实际的物理地址：100 + 32KB = 32868。然后它会检查该地址是否在界限内（100 小于 2KB），发现是的，于是发起对物理地址 32868 的引用。

补充：段错误

段错误指的是在支持分段的机器上发生了非法的内存访问。有趣的是，即使在不支持分段的机器上这个术语依然保留。但如果你弄不清楚为什么代码老是出错，就没那么有趣了。

来看一个堆中的地址，虚拟地址 4200（同样参考图 16.1）。如果用虚拟地址 4200 加上

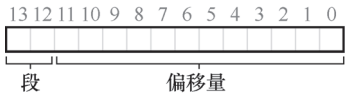
堆的基址（34KB），得到物理地址 39016，这不是正确的地址。我们首先应该先减去堆的偏移量，即该地址指的是这个段中的哪个字节。因为堆从虚拟地址 4K（4096）开始，4200 的偏移量实际上是 4200 减去 4096，即 104，然后用这个偏移量（104）加上基址寄存器中的物理地址（34KB），得到真正的物理地址 34920。

如果我们试图访问非法的地址，例如 7KB，它超出了堆的边界呢？你可以想象发生的情况：硬件会发现该地址越界，因此陷入操作系统，很可能导致终止出错进程。这就是每个 C 程序员都感到恐慌的术语的来源：段异常（segmentation violation）或段错误（segmentation fault）。

16.2 我们引用哪个段

硬件在地址转换时使用段寄存器。它如何知道段内的偏移量，以及地址引用了哪个段？

一种常见的方式，有时称为显式（explicit）方式，就是用虚拟地址的开头几位来标识不同的段，VAX/VMS 系统使用了这种技术[LL82]。在我们之前的例子中，有 3 个段，因此需要两位来标识。如果我们用 14 位虚拟地址的前两位来标识，那么虚拟地址如下所示：



那么在我们的例子中，如果前两位是 00，硬件就知道这是属于代码段的地址，因此使用代码段的基址和界限来重定位到正确的物理地址。如果前两位是 01，则是堆地址，对应地，使用堆的基址和界限。下面来看一个 4200 之上的堆虚拟地址，进行进制转换，确保弄清楚这些内容。虚拟地址 4200 的二进制形式如下：

从图中可以看到，前两位（01）告诉硬件我们引用哪个段。剩下的 12 位是段内偏移：0000 0110 1000（即十六进制 0x068 或十进制 104）。因此，硬件就用前两位来决定使用哪个段寄存器，然后用后 12 位作为段内偏移。偏移量与基址寄存器相加，硬件就得到了最终的物理地址。请注意，偏移量也简化了对段边界的判断。我们只要检查偏移量是否小于界限，大于界限的为非法地址。因此，如果基址和界限放在数组中（每个段一项），为了获得需要的物理地址，硬件会做下面这样的事：

```
1    // get top 2 bits of 14-bit VA
2    Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3    // now get offset
4    Offset  = VirtualAddress & OFFSET_MASK
5    if (Offset >= Bounds[Segment])
6        RaiseException(PROTECTION_FAULT)
7    else
8        PhysAddr = Base[Segment] + Offset
9        Register = AccessMemory(PhysAddr)
```

在我们的例子中，可以为上面的常量填上值。具体来说，SEG_MASK 为 0x3000，SEG_SHIFT 为 12，OFFSET_MASK 为 0xFFF。

你或许已经注意到，上面使用两位来区分段，但实际只有 3 个段（代码、堆、栈），因此有一个段的地址空间被浪费。因此有些系统中会将堆和栈当作同一个段，因此只需要一位来做标识[LL82]。

硬件还有其他方法来决定特定地址在哪个段。在隐式（implicit）方式中，硬件通过地址产生的方式来确定段。例如，如果地址由程序计数器产生（即它是指令获取），那么地址在代码段。如果基于栈或基址指针，它一定在栈段。其他地址则在堆段。

16.3 栈怎么办

到目前为止，我们一直没有讲地址空间中的一个重要部分：栈。在表 16.1 中，栈被重定位到物理地址 28KB。但有一点关键区别，它反向增长。在物理内存中，它始于 28KB，增长回到 26KB，相应虚拟地址从 16KB 到 14KB。地址转换必须有所不同。

首先，我们需要一点硬件支持。除了基址和界限外，硬件还需要知道段的增长方向（用一位区分，比如 1 代表自小而大增长，0 反之）。在表 16.2 中，我们更新了硬件记录的视图。

表 16.2 段寄存器（支持反向增长）

段	基址	大小	是否反向增长
代码	32KB	2KB	1
堆	34KB	2KB	1
栈	28KB	2KB	0

硬件理解段可以反向增长后，这种虚拟地址的地址转换必须有点不同。下面来看一个栈虚拟地址的例子，将它进行转换，以理解这个过程：

在这个例子中，假设要访问虚拟地址 15KB，它应该映射到物理地址 27KB。该虚拟地址的二进制形式是：11 1100 0000 0000（十六进制 0x3C00）。硬件利用前两位（11）来指定段，但然后我们要处理偏移量 3KB。为了得到正确的反向偏移，我们必须从 3KB 中减去最大的段地址：在这个例子中，段可以是 4KB，因此正确的偏移量是 3KB 减去 4KB，即−1KB。只要用这个反向偏移量（−1KB）加上基址（28KB），就得到了正确的物理地址 27KB。用户可以进行界限检查，确保反向偏移量的绝对值小于段的大小。

16.4 支持共享

随着分段机制的不断改进，系统设计人员很快意识到，通过再多一点的硬件支持，就能实现新的效率提升。具体来说，要节省内存，有时候在地址空间之间共享（share）某些内存段是有用的。尤其是，代码共享很常见，今天的系统仍然在使用。

为了支持共享，需要一些额外的硬件支持，这就是保护位（protection bit）。基本为每个

段增加了几个位，标识程序是否能够读写该段，或执行其中的代码。通过将代码段标记为只读，同样的代码可以被多个进程共享，而不用担心破坏隔离。虽然每个进程都认为自己独占这块内存，但操作系统秘密地共享了内存，进程不能修改这些内存，所以假象得以保持。

表 16.3 展示了一个例子，是硬件（和操作系统）记录的额外信息。可以看到，代码段的权限是可读和可执行，因此物理内存中的一个段可以映射到多个虚拟地址空间。

表 16.3 段寄存器的值（有保护）

段	基址	大小	是否反向增长	保护
代码	32KB	2KB	1	读—执行
堆	34KB	2KB	1	读—写
栈	28KB	2KB	0	读—写

有了保护位，前面描述的硬件算法也必须改变。除了检查虚拟地址是否越界，硬件还需要检查特定访问是否允许。如果用户进程试图写入只读段，或从非执行段执行指令，硬件会触发异常，让操作系统来处理出错进程。

16.5 细粒度与粗粒度的分段

到目前为止，我们的例子大多针对只有很少的几个段的系统（即代码、栈、堆）。我们可以认为这种分段是粗粒度的（coarse-grained），因为它将地址空间分成较大的、粗粒度的块。但是，一些早期系统（如 Multics[CV65, DD68]）更灵活，允许将地址空间划分为大量较小的段，这被称为细粒度（fine-grained）分段。

支持许多段需要进一步的硬件支持，并在内存中保存某种段表（segment table）。这种段表通常支持创建非常多的段，因此系统使用段的方式，可以比之前讨论的方式更灵活。例如，像 Burroughs B5000 这样的早期机器可以支持成千上万的段，有了操作系统和硬件的支持，编译器可以将代码段和数据段划分为许多不同的部分[RK68]。当时的考虑是，通过更细粒度的段，操作系统可以更好地了解哪些段在使用哪些没有，从而可以更高效地利用内存。

16.6 操作系统支持

现在你应该大致了解了分段的基本原理。系统运行时，地址空间中的不同段被重定位到物理内存中。与我们之前介绍的整个地址空间只有一个基址/界限寄存器对的方式相比，大量节省了物理内存。具体来说，栈和堆之间没有使用的区域就不需要再分配物理内存，让我们能将更多地址空间放进物理内存。

然而，分段也带来了一些新的问题。我们先介绍必须关注的操作系统新问题。第一个是老问题：操作系统在上下文切换时应该做什么？你可能已经猜到了：各个段寄存器中的内容必须保存和恢复。显然，每个进程都有自己独立的虚拟地址空间，操作系统必须在进程运行前，确保这些寄存器被正确地赋值。

第二个问题更重要，即管理物理内存的空闲空间。新的地址空间被创建时，操作系统需要在物理内存中为它的段找到空间。之前，我们假设所有的地址空间大小相同，物理内存可以被认为是一些槽块，进程可以放进去。现在，每个进程都有一些段，每个段的大小也可能不同。

一般会遇到的问题是，物理内存很快充满了许多空闲空间的小洞，因而很难分配给新的段，或扩大已有的段。这种问题被称为外部碎片（external fragmentation）[R69]，如图 16.3（左边）所示。

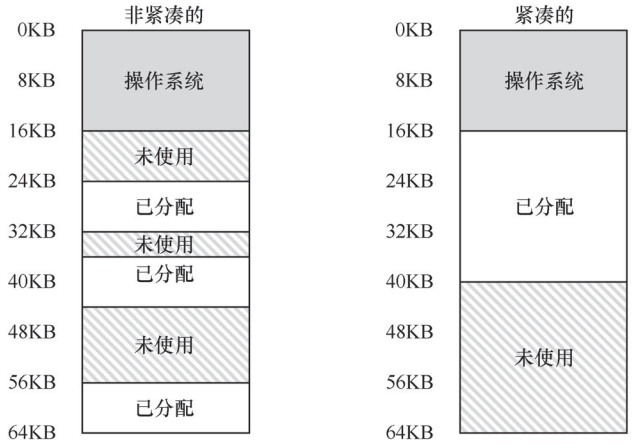


图 16.3 非紧凑和紧凑的内存

在这个例子中，一个进程需要分配一个 20KB 的段。当前有 24KB 空闲，但并不连续（是 3 个不相邻的块）。因此，操作系统无法满足这个 20KB 的请求。

该问题的一种解决方案是紧凑（compact）物理内存，重新安排原有的段。例如，操作系统先终止运行的进程，将它们的数据复制到连续的内存区域中去，改变它们的段寄存器中的值，指向新的物理地址，从而得到了足够大的连续空闲空间。这样做，操作系统能让新的内存分配请求成功。但是，内存紧凑成本很高，因为拷贝段是内存密集型的，一般会占用大量的处理器时间。图 16.3（右边）是紧凑后的物理内存。

一种更简单的做法是利用空闲列表管理算法，试图保留大的内存块用于分配。相关的算法可能有成百上千种，包括传统的最优匹配（best-fit，从空闲链表中找最接近需要分配空间的空闲块返回）、最坏匹配（worst-fit）、首次匹配（first-fit）以及像伙伴算法（buddy algorithm）[K68]这样更复杂的算法。Wilson 等人做过一个很好的调查[W+95]，如果你想对这些算法了解更多，可以从它开始，或者等到第 17 章，我们将介绍一些基本知识。但遗憾的是，无论算法多么精妙，都无法完全消除外部碎片，因此，好的算法只是试图减小它。

提示：如果有一千个解决方案，就没有特别好的

存在如此多不同的算法来尝试减少外部碎片，正说明了解决这个问题没有最好的办法。因此我们满足于找到一个合理的足够好的方案。唯一真正的解决办法就是（我们会在后续章节看到），完全避免这个问题，永远不要分配不同大小的内存块。

16.7 小结

分段解决了一些问题，帮助我们实现了更高效的虚拟内存。不只是动态重定位，通过避免地址空间的逻辑段之间的大量潜在的内存浪费，分段能更好地支持稀疏地址空间。它还很快，因为分段要求的算法很容易，很适合硬件完成，地址转换的开销极小。分段还有一个附加的好处：代码共享。如果代码放在独立的段中，这样的段就可能被多个运行的程序共享。

但我们已经知道，在内存中分配不同大小的段会导致一些问题，我们希望克服。首先，是我们上面讨论的外部碎片。由于段的大小不同，空闲内存被割裂成各种奇怪的大小，因此满足内存分配请求可能会很难。用户可以尝试采用聪明的算法[W+95]，或定期紧凑内存，但问题很根本，难以避免。

第二个问题也许更重要，分段还是不足以支持更一般化的稀疏地址空间。例如，如果有一个很大但是稀疏的堆，都在一个逻辑段中，整个堆仍然必须完整地加载到内存中。换言之，如果使用地址空间的方式不能很好地匹配底层分段的设计目标，分段就不能很好地工作。因此我们需要找到新的解决方案。你准备好了吗？

参考资料

[CV65]“Introduction and Overview of the Multics System”
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
在秋季联合计算机大会上发表的关于 Multics 的 5 篇论文之一。啊，多希望那天我在那个房间里！

[DD68]“Virtual Memory, Processes, and Sharing in Multics”Robert C. Daley and Jack B. Dennis
Communications of the ACM, Volume 11, Issue 5, May 1968
一篇关于如何在 Multics 中进行动态链接的早期文章。文中的内容远远领先于它的时代。随着大型 X-windows 库的要求，动态链接最终在 20 年后回到系统中。有人说，这些大型的 X11 库是 MIT 对 UNIX 的早期版本中取消对动态链接支持的“报复”！

[G62]“Fact Segmentation”
M. N. Greenfield
Proceedings of the SJCC, Volume 21, May 1962
另一篇关于分段的早期论文，发表的时间太早了，以至于没有引用其他人的工作。

[H61]“Program Organization and Record Keeping for Dynamic Storage”
A. W. Holt
Communications of the ACM, Volume 4, Issue 10, October 1961

一篇关于分段及其一些用途的论文，发表时间非常早且难以阅读。

[I09]“Intel 64 and IA-32 Architectures Software Developer’s Manuals”Intel, 2009
尝试阅读这里的分段（第 3a 卷第 3 章），它会让你伤脑筋，至少有一点“头疼”。

[K68]“The Art of Computer Programming: Volume I”Donald Knuth
Addison-Wesley, 1968
Knuth 不仅因其早期关于计算机编程艺术的书而闻名，而且因其排版系统 TeX 而闻名。该系统仍然是当今专业人士使用的强大排版工具，并且排版了这本书。他关于算法的论述很早就引用了许多当今计算系统的算法。

[L83]“Hints for Computer Systems Design”Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
关于如何构建系统的宝贵建议。一下子读完这篇文章很难，每次读几页，就像品一杯美酒，或把它当作一本参考手册。

[LL82]“Virtual Memory Management in the VAX/VMS Operating System”Henry M. Levy and Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982)
一个经典的内存管理系统，在设计上有很多常识。我们将在后面的章节中更详细地研究它。

[RK68]“Dynamic Storage Allocation Systems”
B．Randell and C.J. Kuehner Communications of the ACM
Volume 11(5), pages 297-306, May 1968
对分页和分段两者的差异有一个很好的阐述，其中还有各种机器的历史讨论。

[R69]“A note on storage fragmentation and program segmentation”Brian Randell
Communications of the ACM
Volume 12(7), pages 365-372, July 1969
One of the earliest papers to discuss fragmentation.
最早讨论碎片问题的论文之一。

[W+95]“Dynamic Storage Allocation: A Survey and Critical Review”Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles In International Workshop on Memory Management
Scotland, United Kingdom, September 1995
一份关于内存分配程序的很棒的调查报告。

作业

该程序允许你查看在具有分段的系统中如何执行地址转换。详情请参阅 README 文件。

问题

1．先让我们用一个小地址空间来转换一些地址。这里有一组简单的参数和几个不同的随机种子。你可以转换这些地址吗？

```
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
```

2．现在，让我们看看是否理解了这个构建的小地址空间（使用上面问题的参数）。段 0 中最高的合法虚拟地址是什么？段 1 中最低的合法虚拟地址是什么？在整个地址空间中，最低和最高的非法地址是什么？最后，如何运行带有-A 标志的 segmentation.py 来测试你是否正确？

3．假设我们在一个 128 字节的物理内存中有一个很小的 16 字节地址空间。你会设置什么样的基址和界限，以便让模拟器为指定的地址流生成以下转换结果：有效，有效，违规，违反，有效，有效？假设用以下参数：

```
segmentation.py -a 16 -p 128
  -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
  --b0 ? --l0 ? --b1 ? --l1 ?
```

4．假设我们想要生成一个问题，其中大约 90%的随机生成的虚拟地址是有效的（即不产生段异常）。你应该如何配置模拟器来做到这一点？哪些参数很重要？

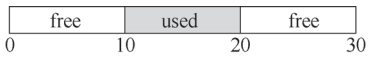
5．你可以运行模拟器，使所有虚拟地址无效吗？怎么做到？

第 17 章　空闲空间管理

本章暂且将对虚拟内存的讨论放在一边，来讨论所有内存管理系统的一个基本方面，无论是 malloc 库（管理进程中堆的页），还是操作系统本身（管理进程的地址空间）。具体来说，我们会讨论空闲空间管理（free-space management）的一些问题。

让问题更明确一点。管理空闲空间当然可以很容易，我们会在讨论分页概念时看到。如果需要管理的空间被划分为固定大小的单元，就很容易。在这种情况下，只需要维护这些大小固定的单元的列表，如果有请求，就返回列表中的第一项。

如果要管理的空闲空间由大小不同的单元构成，管理就变得困难（而且有趣）。这种情况出现在用户级的内存分配库（如 malloc()和 free()），或者操作系统用分段（segmentation）的方式实现虚拟内存。在这两种情况下，出现了外部碎片（external fragmentation）的问题：空闲空间被分割成不同大小的小块，成为碎片，后续的请求可能失败，因为没有一块足够大的连续空闲空间，即使这时总的空闲空间超出了请求的大小。



上面展示了该问题的一个例子。在这个例子中，全部可用空闲空间是 20 字节，但被切成两个 10 字节大小的碎片，导致一个 15 字节的分配请求失败。所以本章需要解决的问题是：

关键问题：如何管理空闲空间

要满足变长的分配请求，应该如何管理空闲空间？什么策略可以让碎片最小化？不同方法的时间和空间开销如何？

17.1　假设

本章的大多数讨论，将聚焦于用户级内存分配库中分配程序的辉煌历史。我们引用了 Wilson 的出色调查 [W+95]，有兴趣的读者可以从原文了解更多细节[①]。

我们假定基本的接口就像 malloc()和 free()提供的那样。具体来说，void * malloc(size t size)需要一个参数 size，它是应用程序请求的字节数。函数返回一个指针（没有具体的类型，在 C 语言的术语中是 void 类型），指向这样大小（或较大一点）的一块空间。对应的函数 void free(void *ptr)函数接受一个指针，释放对应的内存块。请注意该接口的隐含意义，在释放空间时，用户不需告知库这块空间的大小。因此，在只传入一个指针的情况下，库必须能够弄清楚这块内存的大小。我们将在稍后介绍是如何得知的。

① 它有近 80 页长。因此，你必须要真的对它感兴趣！

该库管理的空间由于历史原因被称为堆，在堆上管理空闲空间的数据结构通常称为空闲列表（free list）。该结构包含了管理内存区域中所有空闲块的引用。当然，该数据结构不一定真的是列表，而只是某种可以追踪空闲空间的数据结构。

进一步假设，我们主要关心的是外部碎片（external fragmentation），如上所述。当然，分配程序也可能有内部碎片（internal fragmentation）的问题。如果分配程序给出的内存块超出请求的大小，在这种块中超出请求的空间（因此而未使用）就被认为是内部碎片（因为浪费发生在已分配单元的内部），这是另一种形式的空间浪费。但是，简单起见，同时也因为它更有趣，这里主要讨论外部碎片。

我们还假设，内存一旦被分配给客户，就不可以被重定位到其他位置。例如，一个程序调用 malloc()，并获得一个指向堆中一块空间的指针，这块区域就“属于”这个程序了，库不再能够移动，直到程序调用相应的 free()函数将它归还。因此，不可能进行紧凑（compaction）空闲空间的操作，从而减少碎片[①]。但是，操作系统层在实现分段（segmentation）时，却可以通过紧凑来减少碎片（正如第 16 章讨论的那样）。

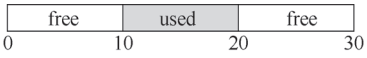
最后我们假设，分配程序所管理的是连续的一块字节区域。在一些情况下，分配程序可以要求这块区域增长。例如，一个用户级的内存分配库在空间快用完时，可以向内核申请增加堆空间（通过 sbrk 这样的系统调用），但是，简单起见，我们假设这块区域在整个生命周期内大小固定。

17.2 底层机制

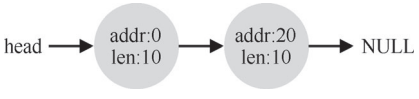
在深入策略细节之前，我们先来介绍大多数分配程序采用的通用机制。首先，探讨空间分割与合并的基本知识。其次，看看如何快速并相对轻松地追踪已分配的空间。最后，讨论如何利用空闲区域的内部空间维护一个简单的列表，来追踪空闲和已分配的空间。

分割与合并

空闲列表包含一组元素，记录了堆中的哪些空间还没有分配。假设有下面的 30 字节的堆：



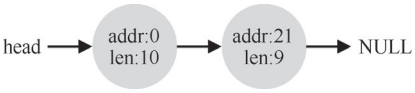
这个堆对应的空闲列表会有两个元素，一个描述第一个 10 字节的空闲区域（字节 0～9），一个描述另一个空闲区域（字节 20～29）：



① 一旦将指向内存块的一个指针交给 C 程序，通常很难确定所有对该区域的引用（指针），这些引用（指针）可能存储在其他变量中，或者甚至在执行的某个时刻存储在寄存器中。在更强类型的、带垃圾收集的语言中，情况可能并非如此，因此可以用紧凑技术来减少碎片。

通过上面的介绍可以看出，任何大于 10 字节的分配请求都会失败（返回 NULL），因为没有足够的连续可用空间。而恰好 10 字节的需求可以由两个空闲块中的任何一个满足。但是，如果申请小于 10 字节空间，会发生什么？

假设我们只申请一个字节的内存。这种情况下，分配程序会执行所谓的分割（splitting）动作：它找到一块可以满足请求的空闲空间，将其分割，第一块返回给用户，第二块留在空闲列表中。在我们的例子中，假设这时遇到申请一个字节的请求，分配程序选择使用第二块空闲空间，对 malloc()的调用会返回 20（1 字节分配区域的地址），空闲列表会变成这样：



从上面可以看出，空闲列表基本没有变化，只是第二个空闲区域的起始位置由 20 变成 21，长度由 10 变为 9 了[①]。因此，如果请求的空间大小小于某块空闲块，分配程序通常会进行分割。

许多分配程序中因此也有一种机制，名为合并（coalescing）。还是看前面的例子（10 字节的空闲空间，10 字节的已分配空间，和另外 10 字节的空闲空间）。

对于这个（小）堆，如果应用程序调用 free(10)，归还堆中间的空间，会发生什么？如果只是简单地将这块空闲空间加入空闲列表，不多想想，可能得到如下的结果：

问题出现了：尽管整个堆现在完全空闲，但它似乎被分割成了 3 个 10 字节的区域。这时，如果用户请求 20 字节的空间，简单遍历空闲列表会找不到这样的空闲块，因此返回失败。

为了避免这个问题，分配程序会在释放一块内存时合并可用空间。想法很简单：在归还一块空闲内存时，仔细查看要归还的内存块的地址以及邻近的空闲空间块。如果新归还的空间与一个原有空闲块相邻（或两个，就像这个例子），就将它们合并为一个较大的空闲块。通过合并，最后空闲列表应该像这样：

实际上，这是堆的空闲列表最初的样子，在所有分配之前。通过合并，分配程序可以更好地确保大块的空闲空间能提供给应用程序。

追踪已分配空间的大小

你可能注意到，free(void *ptr)接口没有块大小的参数。因此它是假定，对于给定的指针，内存分配库可以很快确定要释放空间的大小，从而将它放回空闲列表。

要完成这个任务，大多数分配程序都会在头块（header）中保存一点额外的信息，它在内存中，通常就在返回的内存块之前。我们再看一个例子（见图 17.1）。在这个例子中，我

① 这里的讨论假设没有头块，这是我们现在做出的一个不现实但简化的假设。

们检查一个 20 字节的已分配块，由 ptr 指着，设想用户调用了 malloc()，并将结果保存在 ptr 中：ptr = malloc(20)。

该头块中至少包含所分配空间的大小（这个例子中是 20）。它也可能包含一些额外的指针来加速空间释放，包含一个幻数来提供完整性检查，以及其他信息。我们假定，一个简单的头块包含了分配空间的大小和一个幻数：

```
typedef struct   header_t {
    int size;
    int magic;
} header_t;
```

上面的例子看起来会像图 17.2 的样子。用户调用 free(ptr)时，库会通过简单的指针运算得到头块的位置：

```
void free(void *ptr) {
    header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

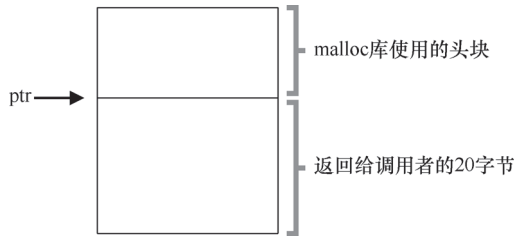


图 17.1 一个已分配的区域加上头块

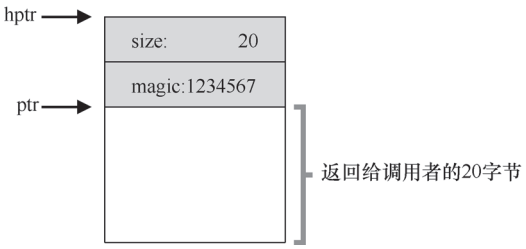


图 17.2 头块的具体内容

获得头块的指针后，库可以很容易地确定幻数是否符合预期的值，作为正常性检查（assert（hptr->magic == 1234567）），并简单计算要释放的空间大小（即头块的大小加区域长度）。请注意前一句话中一个小但重要的细节：实际释放的是头块大小加上分配给用户的空间的大小。因此，如果用户请求 *N* 字节的内存，库不是寻找大小为 *N* 的空闲块，而是寻找 *N* 加上头块大小的空闲块。

嵌入空闲列表

到目前为止，我们这个简单的空闲列表还只是一个概念上的存在，它就是一个列表，描述了堆中的空闲内存块。但如何在空闲内存自己内部建立这样一个列表呢？

在更典型的列表中，如果要分配新节点，你会调用 malloc()来获取该节点所需的空间。遗憾的是，在内存分配库内，你无法这么做！你需要在空闲空间本身中建立空闲空间列表。虽然听起来有点奇怪，但别担心，这是可以做到的。

假设我们需要管理一个 4096 字节的内存块（即堆是 4KB）。为了将它作为一个空闲空间列表来管理，首先要初始化这个列表。开始，列表中只有一个条目，记录了大小为 4096 的空间（减去头块的大小）。下面是该列表中一个节点描述：

```
typedef struct   node_t {
    int    size;
```

```
    struct   node_t *next;
} node_t;
```

现在来看一些代码，它们初始化堆，并将空闲列表的第一个元素放在该空间中。假设堆构建在某块空闲空间上，这块空间通过系统调用 mmap()获得。这不是构建这种堆的唯一选择，但在这个例子中很合适。下面是代码：

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size    = 4096 - sizeof(node_t);
head->next    = NULL;
```

执行这段代码之后，列表的状态是它只有一个条目，记录大小为 4088。

是的，这是一个小堆，但对我们是一个很好的例子。head 指针指向这块区域的起始地址，假设是 16KB（尽管任何虚拟地址都可以）。堆看起来如图 17.3 所示。

现在，假设有一个 100 字节的内存请求。为了满足这个请求，库首先要找到一个足够大小的块。因为只有一个 4088 字节的块，所以选中这个块。然后，这个块被分割（split）为两块：一块足够满足请求（以及头块，如前所述），一块是剩余的空闲块。假设记录头块为 8 个字节（一个整数记录大小，一个整数记录幻数），堆中的空间如图 17.4 所示。

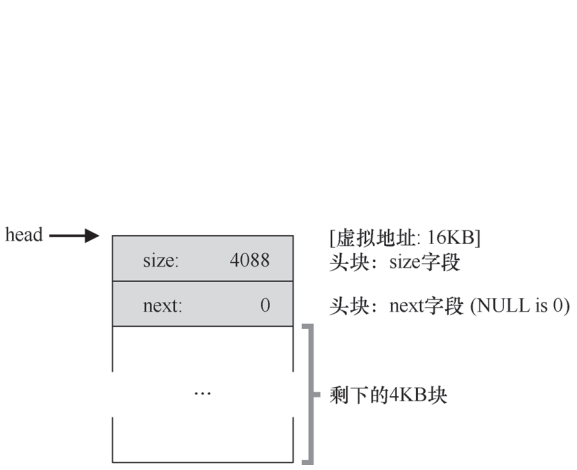


图 17.3 有一个空闲块的堆

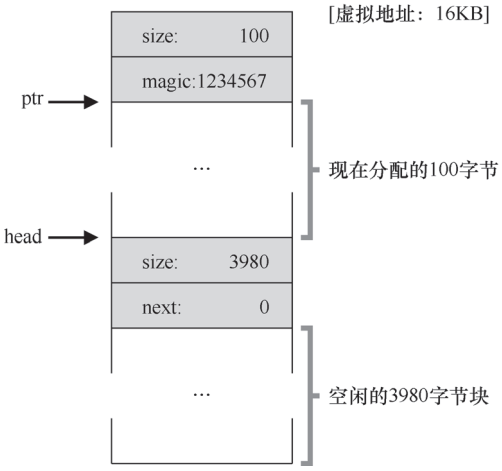


图 17.4 在一次分配之后的堆

至此，对于 100 字节的请求，库从原有的一个空闲块中分配了 108 字节，返回指向它的一个指针（在上图中用 ptr 表示），并在其之前连续的 8 字节中记录头块信息，供未来的 free()函数使用。同时将列表中的空闲节点缩小为 3980 字节（4088−108）。

现在再来看该堆，其中有 3 个已分配区域，每个 100（加上头块是 108）。这个堆如图 17.5 所示。

可以看出，堆的前 324 字节已经分配，因此我们看到该空间中有 3 个头块，以及 3 个 100 字节的用户使用空间。空闲列表还是无趣：只有一个节点（由 head 指向），但在 3 次分割后，现在大小只有 3764 字节。但如果用户程序通过 free()归还一些内存，会发生什么？

在这个例子中，应用程序调用 free(16500)，归还了中间的一块已分配空间（内存块的起

始地址 16384 加上前一块的 108，和这一块的头块的 8 字节，就得到了 16500）。这个值在前图中用 sptr 指向。

库马上弄清楚了这块要释放空间的大小，并将空闲块加回空闲列表。假设我们将它插入到空闲列表的头位置，该空间如图 17.6 所示。

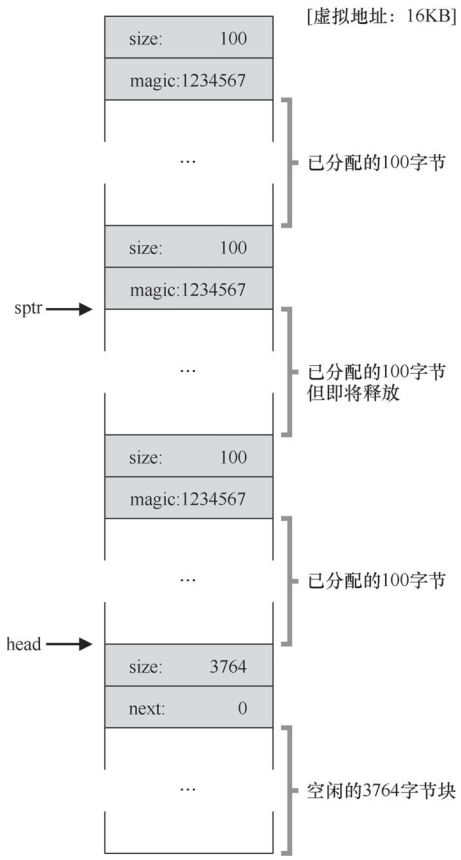


图 17.5 空闲空间和 3 个已分配块

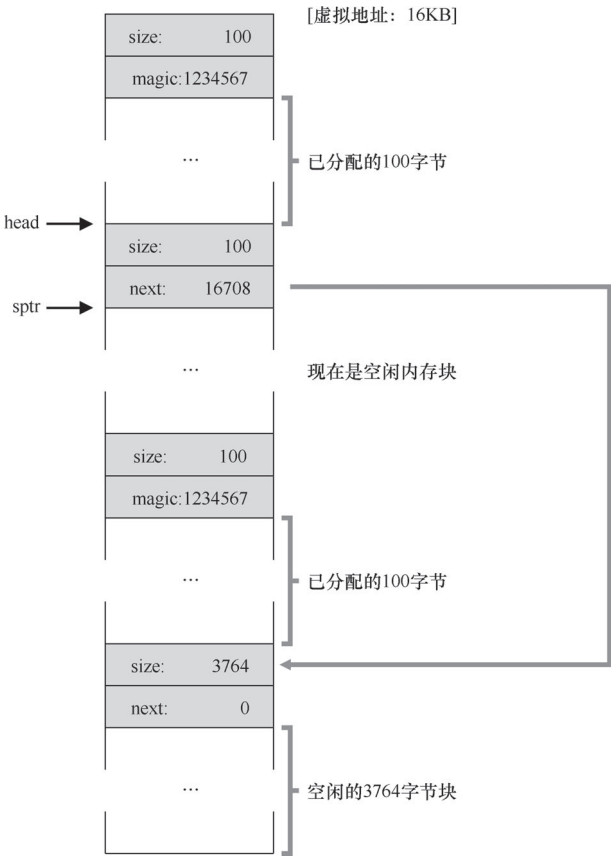


图 17.6 空闲空间和两个已分配的块

现在的空闲列表包括一个小空闲块（100 字节，由列表的头指向）和一个大空闲块（3764 字节）。

我们的列表终于有不止一个元素了！是的，空闲空间被分割成了两段，但很常见。

最后一个例子：现在假设剩余的两块已分配的空间也被释放。没有合并，空闲列表将非常破碎，如图 17.7 所示。

从图中可以看出，我们现在一团糟！为什么？简单，我们忘了合并（coalesce）列表项，虽然整个内存空间是空闲的，但却被分成了小段，因此形成了碎片化的内存空间。解决方案很简单：遍历列表，合并（merge）相邻块。完成之后，堆又成了一个整体。

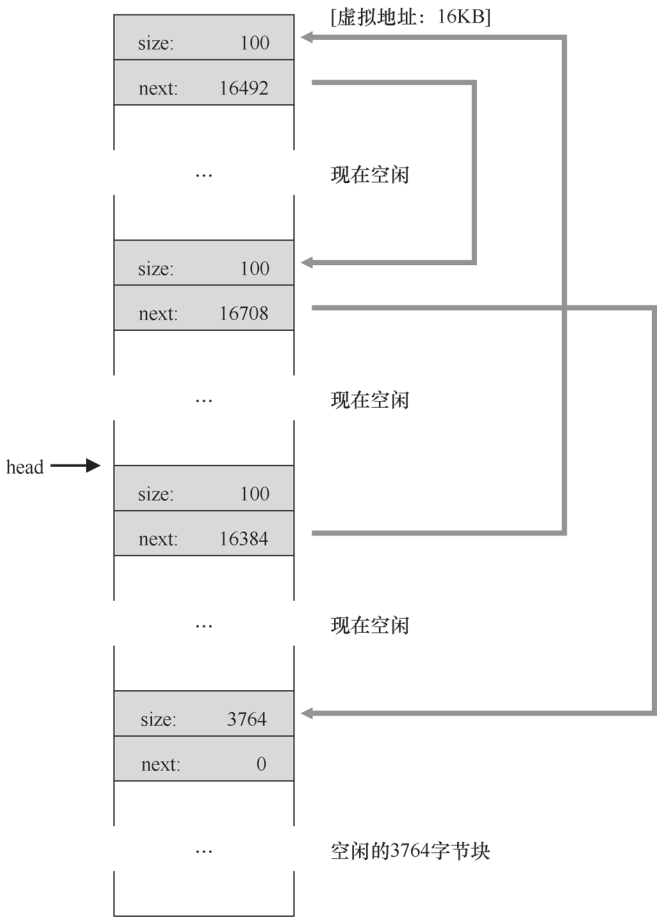


图 17.7 未合并的空闲空间列表

让堆增长

我们应该讨论最后一个很多内存分配库中都有的机制。具体来说，如果堆中的内存空间耗尽，应该怎么办？最简单的方式就是返回失败。在某些情况下这也是唯一的选择，因此返回 NULL 也是一种体面的方式。别太难过！你尽力了，即使失败，你也虽败犹荣。

大多数传统的分配程序会从很小的堆开始，当空间耗尽时，再向操作系统申请更大的空间。通常，这意味着它们进行了某种系统调用（例如，大多数 UNIX 系统中的 sbrk），让堆增长。操作系统在执行 sbrk 系统调用时，会找到空闲的物理内存页，将它们映射到请求进程的地址空间中去，并返回新的堆的末尾地址。这时，就有了更大的堆，请求就可以成功满足。

17.3 基本策略

既然有了这些底层机制，让我们来看看管理空闲空间的一些基本策略。这些方法大多基于简单的策略，你也能想到。在阅读之前试试，你是否能想出所有的选择（也许还有新策略！）。

理想的分配程序可以同时保证快速和碎片最小化。遗憾的是，由于分配及释放的请求序列是任意的（毕竟，它们由用户程序决定），任何特定的策略在某组不匹配的输入下都会变得非常差。所以我们不会描述“最好”的策略，而是介绍一些基本的选择，并讨论它们的优缺点。

最优匹配

最优匹配（best fit）策略非常简单：首先遍历整个空闲列表，找到和请求大小一样或更大的空闲块，然后返回这组候选者中最小的一块。这就是所谓的最优匹配（也可以称为最小匹配）。只需要遍历一次空闲列表，就足以找到正确的块并返回。

最优匹配背后的想法很简单：选择最接近用户请求大小的块，从而尽量避免空间浪费。然而，这有代价。简单的实现在遍历查找正确的空闲块时，要付出较高的性能代价。

最差匹配

最差匹配（worst fit）方法与最优匹配相反，它尝试找最大的空闲块，分割并满足用户需求后，将剩余的块（很大）加入空闲列表。最差匹配尝试在空闲列表中保留较大的块，而不是向最优匹配那样可能剩下很多难以利用的小块。但是，最差匹配同样需要遍历整个空闲列表。更糟糕的是，大多数研究表明它的表现非常差，导致过量的碎片，同时还有很高的开销。

首次匹配

首次匹配（first fit）策略就是找到第一个足够大的块，将请求的空间返回给用户。同样，剩余的空闲空间留给后续请求。

首次匹配有速度优势（不需要遍历所有空闲块），但有时会让空闲列表开头的部分有很多小块。因此，分配程序如何管理空闲列表的顺序就变得很重要。一种方式是基于地址排序（address-based ordering）。通过保持空闲块按内存地址有序，合并操作会很容易，从而减少了内存碎片。

下次匹配

不同于首次匹配每次都从列表的开始查找，下次匹配（next fit）算法多维护一个指针，指向上一次查找结束的位置。其想法是将对空闲空间的查找操作扩散到整个列表中去，避免对列表开头频繁的分割。这种策略的性能与首次匹配很接近，同样避免了遍历查找。

例子

下面是上述策略的一些例子。设想一个空闲列表包含 3 个元素，长度依次为 10、30、20（我们暂时忽略头块和其他细节，只关注策略的操作方式）：

假设有一个 15 字节的内存请求。最优匹配会遍历整个空闲列表，发现 20 字节是最优匹配，因为它是满足请求的最小空闲块。结果空闲列表变为：

本例中发生的情况，在最优匹配中常常发生，现在留下了一个小空闲块。最差匹配类似，但会选择最大的空闲块进行分割，在本例中是 30。结果空闲列表变为：

在这个例子中，首次匹配会和最差匹配一样，也发现满足请求的第一个空闲块。不同的是查找开销，最优匹配和最差匹配都需要遍历整个列表，而首次匹配只找到第一个满足需求的块即可，因此减少了查找开销。

这些例子只是内存分配策略的肤浅分析。真实场景下更详细的分析和更复杂的分配行为（如合并），需要更深入的理解。也许可以作为作业，你说呢？

17.4 其他方式

除了上述基本策略外，人们还提出了许多技术和算法，来改进内存分配。这里我们列出一些来供你考虑（就是让你多一些思考，不只局限于最优匹配）。

分离空闲列表

一直以来有一种很有趣的方式叫作分离空闲列表（segregated list）。基本想法很简单：如果某个应用程序经常申请一种（或几种）大小的内存空间，那就用一个独立的列表，只管理这样大小的对象。其他大小的请求都交给更通用的内存分配程序。

这种方法的好处显而易见。通过拿出一部分内存专门满足某种大小的请求，碎片就不再是问题了。而且，由于没有复杂的列表查找过程，这种特定大小的内存分配和释放都很快。

就像所有好主意一样，这种方式也为系统引入了新的复杂性。例如，应该拿出多少内存来专门为某种大小的请求服务，而将剩余的用来满足一般请求？超级工程师 Jeff Bonwick 为 Solaris 系统内核设计的厚块分配程序（slab allocator），很优雅地处理了这个问题[B94]。

具体来说，在内核启动时，它为可能频繁请求的内核对象创建一些对象缓存（object cache），如锁和文件系统 inode 等。这些的对象缓存每个分离了特定大小的空闲列表，因此能够很快地响应内存请求和释放。如果某个缓存中的空闲空间快耗尽时，它就向通用内存分配程序申请一些内存厚块（slab）（总量是页大小和对象大小的公倍数）。相反，如果给定厚块中对象的引用计数变为 0，通用的内存分配程序可以从专门的分配程序中回收这些空间，这通常发生在虚拟内存系统需要更多的空间的时候。

补充：了不起的工程师真的了不起

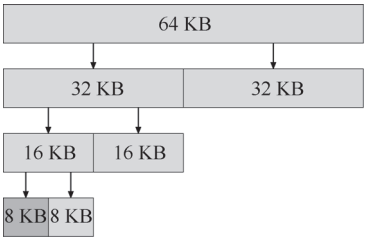
像 Jeff Bonwick 这样的工程师（Jeff Bonwick 不仅写了上面提到的厚块分配程序，还是令人惊叹的文件系统 ZFS 的负责人），是硅谷的灵魂。在每一个伟大的产品或技术后面都有这样一个人（或一小群人），他们的天赋、能力和奉献精神远超众人。Facebook 的 Mark Zuckerberg 曾经说过：“那些在自己的领域中超凡脱俗的人，比那些相当优秀的人强得不是一点点。”这就是为什么，会有人成立自己的公司，然后永远地改变了这个世界（想想 Google、Apple 和 Facebook）。努力工作，你也可能成为这种“以一当百”的人。做不到的话，就和这样的人一起工作，你会明白什么是“听君一席话，胜读十年书”。如果都做不到，那就太难过了。

厚块分配程序比大多数分离空闲列表做得更多，它将列表中的空闲对象保持在预初始化的状态。Bonwick 指出，数据结构的初始化和销毁的开销很大[B94]。通过将空闲对象保持在初始化状态，厚块分配程序避免了频繁的初始化和销毁，从而显著降低了开销。

伙伴系统

因为合并对分配程序很关键，所以人们设计了一些方法，让合并变得简单，一个好例子就是二分伙伴分配程序（binary buddy allocator）[K65]。

在这种系统中，空闲空间首先从概念上被看成大小为 2^N 的大空间。当有一个内存分配请求时，空闲空间被递归地一分为二，直到刚好可以满足请求的大小（再一分为二就无法满足）。这时，请求的块被返回给用户。在下面的例子中，一个 64KB 大小的空闲空间被切分，以便提供 7KB 的块：



在这个例子中，最左边的 8KB 块被分配给用户（如上图中深灰色部分所示）。请注意，这种分配策略只允许分配 2 的整数次幂大小的空闲块，因此会有内部碎片（internal fragment）的麻烦。

伙伴系统的漂亮之处在于块被释放时。如果将这个 8KB 的块归还给空闲列表，分配程序会检查“伙伴”8KB 是否空闲。如果是，就合二为一，变成 16KB 的块。然后会检查这个 16KB 块的伙伴是否空闲，如果是，就合并这两块。这个递归合并过程继续上溯，直到合并整个内存区域，或者某一个块的伙伴还没有被释放。

伙伴系统运转良好的原因，在于很容易确定某个块的伙伴。怎么找？仔细想想上面例子中的各个块的地址。如果你想得够仔细，就会发现每对互为伙伴的块只有一位不同，正是这一位决定了它们在整个伙伴树中的层次。现在你应该已经大致了解了二分伙伴分配程序的工作方式。更多的细节可以参考 Wilson 的调查[W+95]。

其他想法

上面提到的众多方法都有一个重要的问题，缺乏可扩展性（scaling）。具体来说，就是查找列表可能很慢。因此，更先进的分配程序采用更复杂的数据结构来优化这个开销，牺牲简单性来换取性能。例子包括平衡二叉树、伸展树和偏序树[W+95]。

考虑到现代操作系统通常会有多核，同时会运行多线程的程序（本书之后关于并发的章节将会详细介绍），因此人们做了许多工作，提升分配程序在多核系统上的表现。两个很棒的例子参见 Berger 等人的[B+00]和 Evans 的[E06]，看看文章了解更多细节。

这只是人们为了优化内存分配程序，在长时间内提出的几千种想法中的两种。感兴趣的话可以深入阅读。或者阅读 glibc 分配程序的工作原理[S15]，你会更了解现实的情形。

17.5 小结

在本章中，我们讨论了最基本的内存分配程序形式。这样的分配程序存在于所有地方，与你编写的每个 C 程序链接，也和管理其自身数据结构的内存的底层操作系统链接。与许多系统一样，在构建这样一个系统时需要做许多折中。对分配程序提供的确切工作负载了解得越多，就越能调整它以更好地处理这种工作负载。在现代计算机系统中，构建一个适用于各种工作负载、快速、空间高效、可扩展的分配程序仍然是一个持续的挑战。

参考资料

[B+00]“Hoard: A Scalable Memory Allocator for Multithreaded Applications”Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson ASPLOS-IX, November 2000
Berger 和公司的优秀多处理器系统分配程序。它不仅是一篇有趣的论文，也是能用于指导实战的！

[B94]“The Slab Allocator: An Object-Caching Kernel Memory Allocator”Jeff Bonwick
USENIX ’94
一篇关于如何为操作系统内核构建分配程序的好文章，也是如何专门针对特定通用对象大小的一个很好的例子。

[E06]“A Scalable Concurrent malloc(3) Implementation for FreeBSD”Jason Evans
本文详细介绍如何构建一个真正的现代分配程序以用于多处理器。“jemalloc”分配程序今天在 FreeBSD、NetBSD、Mozilla Firefox 和 Facebook 中已广泛使用。

[K65]“A Fast Storage Allocator”Kenneth C. Knowlton
Communications of the ACM, Volume 8, Number 10, October 1965
伙伴分配的常见引用。一个奇怪的事实是：Knuth 不是把这个想法归功于 Knowlton，而是归功于获得诺贝尔奖的经济学家 Harry Markowitz。另一个奇怪的事实是：Knuth 通过秘书收发他的所有电子邮件。他不会

自己发送电子邮件，而是告诉他的秘书要发送什么邮件，然后秘书负责发送电子邮件。最后一个关于 Knuth 的事实：他创建了 TeX，这是用于排版本书的工具。这是一个惊人的软件[①]。

[S15]“Understanding glibc malloc”Sploitfun
深入了解 glibc malloc 是如何工作的。本文详细得令人惊讶，一篇非常好的阅读材料。

[W+95]“Dynamic Storage Allocation: A Survey and Critical Review”Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles International Workshop on Memory Management
Kinross, Scotland, September 1995
对内存分配的许多方面进行了卓越且深入的调查，比这个小小的章节中所含的内容拥有更多的细节！

作业

程序 malloc.py 让你探索本章中描述的简单空闲空间分配程序的行为。有关其基本操作的详细信息，请参见 README 文件。

问题

1．首先运行 flag -n 10 -H 0 -p BEST -s 0 来产生一些随机分配和释放。你能预测 malloc()/free()会返回什么吗？你可以在每次请求后猜测空闲列表的状态吗？随着时间的推移，你对空闲列表有什么发现？

2．使用最差匹配策略搜索空闲列表（-p WORST）时，结果有何不同？什么改变了？

3．如果使用首次匹配（-p FIRST）会如何？使用首次匹配时，什么变快了？

4．对于上述问题，列表在保持有序时，可能会影响某些策略找到空闲位置所需的时间。使用不同的空闲列表排序（-l ADDRSORT，-l SIZESORT +，-l SIZESORT-）查看策略和列表排序如何相互影响。

5．合并空闲列表可能非常重要。增加随机分配的数量（比如说-n 1000）。随着时间的推移，大型分配请求会发生什么？在有和没有合并的情况下运行（即不用和采用-C 标志）。你看到了什么结果差异？每种情况下的空闲列表有多大？在这种情况下，列表的排序是否重要？

6．将已分配百分比-P 改为高于 50，会发生什么？它接近 100 时分配会怎样？接近 0 会怎样？

7．要生成高度碎片化的空闲空间，你可以提出怎样的具体请求？使用-A 标志创建碎片化的空闲列表，查看不同的策略和选项如何改变空闲列表的组织。

① 实际上我们使用 LaTeX，它基于 Lamport 对 TeX 的补充，但二者非常相似。

第 18 章　分页：介绍

有时候人们会说，操作系统有两种方法，来解决大多数空间管理问题。第一种是将空间分割成不同长度的分片，就像虚拟内存管理中的分段。遗憾的是，这个解决方法存在固有的问题。具体来说，将空间切成不同长度的分片以后，空间本身会碎片化（fragmented），随着时间推移，分配内存会变得比较困难。

因此，值得考虑第二种方法：将空间分割成固定长度的分片。在虚拟内存中，我们称这种思想为分页，可以追溯到一个早期的重要系统，Atlas[KE+62, L78]。分页不是将一个进程的地址空间分割成几个不同长度的逻辑段（即代码、堆、段），而是分割成固定大小的单元，每个单元称为一页。相应地，我们把物理内存看成是定长槽块的阵列，叫作页帧（page frame）。每个这样的页帧包含一个虚拟内存页。我们的挑战是：

关键问题：如何通过页来实现虚拟内存

如何通过页来实现虚拟内存，从而避免分段的问题？基本技术是什么？如何让这些技术运行良好，并尽可能减少空间和时间开销？

18.1　一个简单例子

为了让该方法看起来更清晰，我们用一个简单例子来说明。图 18.1 展示了一个只有 64 字节的小地址空间，有 4 个 16 字节的页（虚拟页 0、1、2、3）。真实的地址空间肯定大得多，通常 32 位有 4GB 的地址空间，甚至有 64 位[①]。在本书中，我们常常用小例子，让大家更容易理解。

物理内存，如图 18.2 所示，也由一组固定大小的槽块组成。在这个例子中，有 8 个页帧（由 128 字节物理内存构成，也是极小的）。从图中可以看出，虚拟地址空间的页放在物理内存的不同位置。图中还显示，操作系统自己用了一些物理内存。

可以看到，与我们以前的方法相比，分页有许多优点。可能最大的改进就是灵活性：通过完善的分页方法，操作系统能够高效地提供地址空间的抽象，不管进程如何使用地址空间。例如，我们不会假定堆和栈的增长方向，以及它们如何使用。

另一个优点是分页提供的空闲空间管理的简单性。例如，如果操作系统希望将 64 字节的小地址空间放到 8 页的物理地址空间中，它只要找到 4 个空闲页。也许操作系统保存了一个所有空闲页的空闲列表（free list），只需要从这个列表中拿出 4 个空闲页。在这个例子

① 64 位地址空间很难想象，它大得惊人。类比可能有助于理解：如果说 32 位地址空间有网球场那么大，则 64 位地址空间大约与欧洲的面积大小相当！

里，操作系统将地址空间的虚拟页 0 放在物理页帧 3，虚拟页 1 放在物理页帧 7，虚拟页 2 放在物理页帧 5，虚拟页 3 放在物理页帧 2。页帧 1、4、6 目前是空闲的。

图 18.1 一个简单的 64 字节地址空间

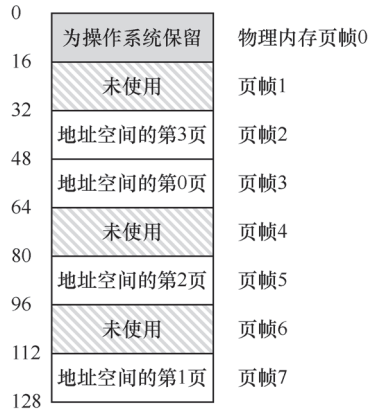


图 18.2 64 字节的地址空间在 128 字节的物理内存中

为了记录地址空间的每个虚拟页放在物理内存中的位置，操作系统通常为每个进程保存一个数据结构，称为页表（page table）。页表的主要作用是为地址空间的每个虚拟页面保存地址转换（address translation），从而让我们知道每个页在物理内存中的位置。对于我们的简单示例（见图 18.2），页表因此具有以下 4 个条目：（虚拟页 0→物理帧 3）、（VP 1→PF 7）、（VP 2→PF 5）和（VP 3→PF 2）。

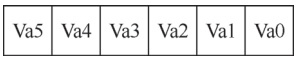
重要的是要记住，这个页表是一个每进程的数据结构（我们讨论的大多数页表结构都是每进程的数据结构，我们将接触的一个例外是倒排页表，inverted page table）。如果在上面的示例中运行另一个进程，操作系统将不得不为它管理不同的页表，因为它的虚拟页显然映射到不同的物理页面（除了共享之外）。

现在，我们了解了足够的信息，可以完成一个地址转换的例子。设想拥有这个小地址空间（64 字节）的进程正在访问内存：

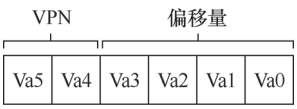
```
movl <virtual address>, %eax
```

具体来说，注意从地址<virtual address>到寄存器 eax 的数据显式加载（因此忽略之前肯定会发生的指令获取）。

为了转换（translate）该过程生成的虚拟地址，我们必须首先将它分成两个组件：虚拟页面号（virtual page number，VPN）和页内的偏移量（offset）。对于这个例子，因为进程的虚拟地址空间是 64 字节，我们的虚拟地址总共需要 6 位（$2^6 = 64$）。因此，虚拟地址可以表示如下：



在该图中，Va5 是虚拟地址的最高位，Va0 是最低位。因为我们知道页的大小（16 字节），所以可以进一步划分虚拟地址，如下所示：



页面大小为 16 字节，位于 64 字节的地址空间。因此我们需要能够选择 4 个页，地址的前 2 位就是做这件事的。因此，我们有一个 2 位的虚拟页号（VPN）。其余的位告诉我们，感兴趣该页的哪个字节，在这个例子中是 4 位，我们称之为偏移量。

当进程生成虚拟地址时，操作系统和硬件必须协作，将它转换为有意义的物理地址。例如，让我们假设上面的加载是虚拟地址 21：

```
movl 21, %eax
```

将“21”变成二进制形式，是“010101”，因此我们可以检查这个虚拟地址，看看它是如何分解成虚拟页号（VPN）和偏移量的：

因此，虚拟地址“21”在虚拟页“01”（或 1）的第 5 个（“0101”）字节处。通过虚拟页号，我们现在可以检索页表，找到虚拟页 1 所在的物理页面。在上面的页表中，物理帧号（PFN）（有时也称为物理页号，physical page number 或 PPN）是 7（二进制 111）。因此，我们可以通过用 PFN 替换 VPN 来转换此虚拟地址，然后将载入发送给物理内存（见图 18.3）。

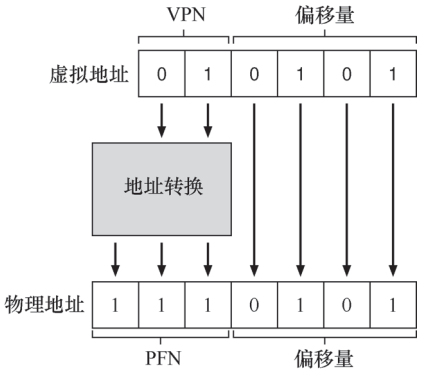


图 18.3 地址转换过程

请注意，偏移量保持不变（即未翻译），因为偏移量只是告诉我们页面中的哪个字节是我们想要的。我们的最终物理地址是 1110101（十进制 117），正是我们希望加载指令（见图 18.2）获取数据的地方。

有了这个基本概念，我们现在可以询问（希望也可以回答）关于分页的一些基本问题。例如，这些页表在哪里存储？页表的典型内容是什么，表有多大？分页是否会使系统变（得很）慢？这些问题和其他迷人的问题（至少部分）在下文中回答。请继续阅读！

18.2 页表存在哪里

页表可以变得非常大，比我们之前讨论过的小段表或基址/界限对要大得多。例如，想象一个典型的 32 位地址空间，带有 4KB 的页。这个虚拟地址分成 20 位的 VPN 和 12 位的偏移量（回想一下，1KB 的页面大小需要 10 位，只需增加两位即可达到 4KB）。

一个 20 位的 VPN 意味着，操作系统必须为每个进程管理 2^{20} 个地址转换（大约一百万）。假设每个页表格条目（PTE）需要 4 个字节，来保存物理地址转换和任何其他有用的东西，每个页表就需要巨大的 4MB 内存！这非常大。现在想象一下有 100 个进程在运行：这意味着操作系统会需要 400MB 内存，只是为了所有这些地址转换！即使是现在，机器拥有千兆字节的内存，将它的一大块仅用于地址转换，这似乎有点疯狂，不是吗？我们甚至不敢想 64 位地址空间的页表有多大。那太可怕了，也许把你吓坏了。

由于页表如此之大，我们没有在 MMU 中利用任何特殊的片上硬件，来存储当前正在运行的进程的页表，而是将每个进程的页表存储在内存中。现在让我们假设页表存在于操作系统管理的物理内存中，稍后我们会看到，很多操作系统内存本身都可以虚拟化，因此页表可以存储在操作系统的虚拟内存中（甚至可以交换到磁盘上），但是现在这太令人困惑了，所以我们会忽略它。图 18.4 展示了操作系统内存中的页表，看到其中的一小组地址转换了吗？

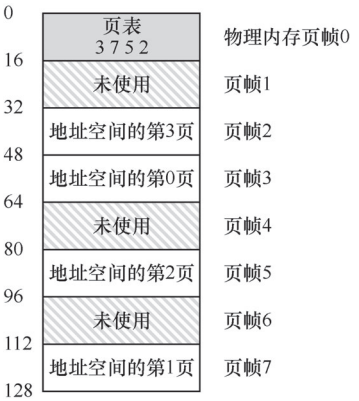


图 18.4 例子：内核物理内存中的页表

18.3 列表中究竟有什么

让我们来谈谈页表的组织。页表就是一种数据结构，用于将虚拟地址（或者实际上，是虚拟页号）映射到物理地址（物理帧号）。因此，任何数据结构都可以采用。最简单的形式称为线性页表（linear page table），就是一个数组。操作系统通过虚拟页号（VPN）检索该数组，并在该索引处查找页表项（PTE），以便找到期望的物理帧号（PFN）。现在，我们将假设采用这个简单的线性结构。在后面的章节中，我们将利用更高级的数据结构来帮助解决一些分页问题。

至于每个 PTE 的内容，我们在其中有许多不同的位，值得有所了解。有效位（valid bit）通常用于指示特定地址转换是否有效。例如，当一个程序开始运行时，它的代码和堆在其地址空间的一端，栈在另一端。所有未使用的中间空间都将被标记为无效（invalid），如果进程尝试访问这种内存，就会陷入操作系统，可能会导致该进程终止。因此，有效位对于支持稀疏地址空间至关重要。通过简单地将地址空间中所有未使用的页面标记为无效，我们不再需要为这些页面分配物理帧，从而节省大量内存。

我们还可能有保护位（protection bit），表明页是否可以读取、写入或执行。同样，以这些位不允许的方式访问页，会陷入操作系统。

还有其他一些重要的部分，但现在我们不会过多讨论。存在位（present bit）表示该页是在物理存储器还是在磁盘上（即它已被换出，swapped out）。当我们研究如何将部分地址空间交换（swap）到磁盘，从而支持大于物理内存的地址空间时，我们将进一步理解这一机制。交换允许操作系统将很少使用的页面移到磁盘，从而释放物理内存。脏位（dirty bit）也很常见，表明页面被带入内存后是否被修改过。

参考位（reference bit，也被称为访问位，accessed bit）有时用于追踪页是否被访问，也用于确定哪些页很受欢迎，因此应该保留在内存中。这些知识在页面替换（page replacement）时非常重要，我们将在随后的章节中详细研究这一主题。

图 18.5 显示了来自 x86 架构的示例页表项[I09]。它包含一个存在位（P），确定是否允许写入该页面的读/写位（R/W） 确定用户模式进程是否可以访问该页面的用户/超级用户位（U/S），有几位（PWT、PCD、PAT 和 G）确定硬件缓存如何为这些页面工作，一个访问位（A）和一个脏位（D），最后是页帧号（PFN）本身。

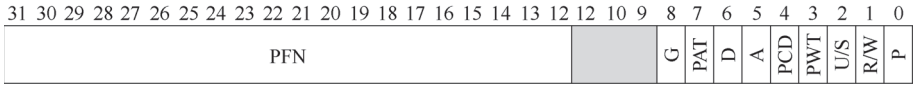


图 18.5 一个 x86 页表项（PTE）

阅读英特尔架构手册[I09]，以获取有关 x86 分页支持的更多详细信息。然而，要事先警告，阅读这些手册时，尽管非常有用（对于在操作系统中编写代码以使用这些页表的用户而言，这些手册当然是必需的），但起初可能很具挑战性。需要一点耐心和强烈的愿望。

18.4 分页：也很慢

内存中的页表，我们已经知道它们可能太大了。事实证明，它们也会让速度变慢。以简单的指令为例：

```
movl 21, %eax
```

同样，我们只看对地址 21 的显式引用，而不关心指令获取。在这个例子中，我们假定硬件为我们执行地址转换。要获取所需数据，系统必须首先将虚拟地址（21）转换为正确的物理地址（117）。因此，在从地址 117 获取数据之前，系统必须首先从进程的页表中提取适当的页表项，执行转换，然后从物理内存中加载数据。

为此，硬件必须知道当前正在运行的进程的页表的位置。现在让我们假设一个页表基址寄存器（page-table base register）包含页表的起始位置的物理地址。为了找到想要的 PTE 的位置，硬件将执行以下功能：

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

在我们的例子中，VPN MASK 将被设置为 0x30（十六进制 30，或二进制 110000），它从完整的虚拟地址中挑选出 VPN 位；SHIFT 设置为 4（偏移量的位数），这样我们就可以将 VPN 位向右移动以形成正确的整数虚拟页码。例如，使用虚拟地址 21（010101），掩码将此值转换为 010000，移位将它变成 01，或虚拟页 1，正是我们期望的值。然后，我们使用该值作为页表基址寄存器指向的 PTE 数组的索引。

一旦知道了这个物理地址，硬件就可以从内存中获取 PTE，提取 PFN，并将它与来自虚拟地址的偏移量连接起来，形成所需的物理地址。具体来说，你可以想象 PFN 被 SHIFT 左移，然后与偏移量进行逻辑或运算，以形成最终地址，如图 18.6 所示。

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
1    // Extract the VPN from the virtual address
2    VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4    // Form the address of the page-table entry (PTE)
5    PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7    // Fetch the PTE
```

```
8    PTE = AccessMemory(PTEAddr)
9
10   // Check if process can access the page
11   if (PTE.Valid == False)
12       RaiseException(SEGMENTATION_FAULT)
13   else if (CanAccess(PTE.ProtectBits) == False)
14       RaiseException(PROTECTION_FAULT)
15   else
16       // Access is OK: form physical address and fetch it
17       offset   = VirtualAddress & OFFSET_MASK
18       PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19       Register = AccessMemory(PhysAddr)
```

图 18.6 利用分页访问内存

最后，硬件可以从内存中获取所需的数据并将其放入寄存器 eax。程序现在已成功从内存中加载了一个值！

总之，我们现在描述了在每个内存引用上发生的情况的初始协议。基本方法如图 18.6 所示。对于每个内存引用（无论是取指令还是显式加载或存储），分页都需要我们执行一个额外的内存引用，以便首先从页表中获取地址转换。工作量很大！额外的内存引用开销很大，在这种情况下，可能会使进程减慢两倍或更多。

现在你应该可以看到，有两个必须解决的实际问题。如果不仔细设计硬件和软件，页表会导致系统运行速度过慢，并占用太多内存。虽然看起来是内存虚拟化需求的一个很好的解决方案，但这两个关键问题必须先克服。

18.5 内存追踪

在结束之前，我们现在通过一个简单的内存访问示例，来演示使用分页时产生的所有内存访问。我们感兴趣的代码片段（用 C 写的，名为 array.c）是这样的：

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

我们编译 array.c 并使用以下命令运行它：

补充：数据结构——页表

现代操作系统的内存管理子系统中最重要的数据结构之一就是页表（page table）。通常，页表存储虚拟—物理地址转换（virtual-to-physical address translation），从而让系统知道地址空间的每个页实际驻留在物理内存中的哪个位置。由于每个地址空间都需要这种转换，因此一般来说，系统中每个进程都有一个页表。页表的确切结构要么由硬件（旧系统）确定，要么由 OS（现代系统）更灵活地管理。

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

当然，为了真正理解这个代码片段（它只是初始化一个数组）进程怎样的内存访问，我们必须知道（或假设）一些东西。首先，我们必须反汇编结果二进制文件（在 Linux 上使用 objdump 或在 Mac 上使用 otool），查看使用什么汇编指令来初始化循环中的数组。以下是生成的汇编代码：

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne  0x1024
```

如果懂一点 x86，代码实际上很容易理解[①]。第一条指令将零值（显示为$0x0）移动到数组位置的虚拟内存地址，这个地址是通过取%edi 的内容并将其加上%eax 乘以 4 来计算的。因此，%edi 保存数组的基址，而%eax 保存数组索引（i）。我们乘以 4，因为数组是一个整型数组，每个元素的大小为 4 个字节。

第二条指令增加保存在%eax 中的数组索引，第三条指令将该寄存器的内容与十六进制值 0x03e8 或十进制数 1000 进行比较。如果比较结果显示两个值不相等（这就是 jne 指令测试），第四条指令跳回到循环的顶部。

为了理解这个指令序列（在虚拟层和物理层）所访问的内存，我们必须假设虚拟内存中代码片段和数组的位置，以及页表的内容和位置。

对于这个例子，我们假设一个大小为 64KB 的虚拟地址空间（不切实际地小）。我们还假定页面大小为 1KB。

我们现在需要知道页表的内容，以及它在物理内存中的位置。假设有一个线性（基于数组）的页表，它位于物理地址 1KB（1024）。

至于其内容，我们只需要关心为这个例子映射的几个虚拟页面。首先，存在代码所在的虚拟页面。由于页大小为 1KB，虚拟地址 1024 驻留在虚拟地址空间的第二页（VPN = 1，因为 VPN = 0 是第一页）。假设这个虚拟页映射到物理帧 4（VPN 1→PFN 4）。

接下来是数组本身。它的大小是 4000 字节（1000 整数），我们假设它驻留在虚拟地址 40000 到 44000（不包括最后一个字节）。它的虚拟页的十进制范围是 VPN = 39……VPN = 42。因此，我们需要这些页的映射。针对这个例子，让我们假设以下虚拟到物理的映射：

(VPN 39 → PFN 7), (VPN 40 → PFN 8), (VPN 41 → PFN 9), (VPN 42 → PFN 10)

我们现在准备好跟踪程序的内存引用了。当它运行时，每个获取指将产生两个内存引用：一个访问页表以查找指令所在的物理框架，另一个访问指令本身将其提取到 CPU 进行处理。另外，在 mov 指令的形式中，有一个显式的内存引用，这会首先增加另一个页表访问（将数组虚拟地址转换为正确的物理地址），然后时数组访问本身。

图 18.7 展示了前 5 次循环迭代的整个过程。最下面的图显示了 *y* 轴上的指令内存引用（黑色虚拟地址和右边的实际物理地址）。中间的图以深灰色展示了数组访问（同样，虚拟在左侧，物理在右侧）；最后，最上面的图展示了浅灰色的页表内存访问（只有物理的，因为本例中的页表位于物理内存中）。整个追踪的 *x* 轴显示循环的前 5 个迭代中内存访问。每个循环有 10 次内存访问，其中包括 4 次取指令，一次显式更新内存，以及 5 次页表访问，

① 我们在这里隐瞒了一点事实，假设每条指令的大小都是 4 字节，实际上，x86 指令是可变大小的。

为这 4 次获取和一次显式更新进行地址转换。

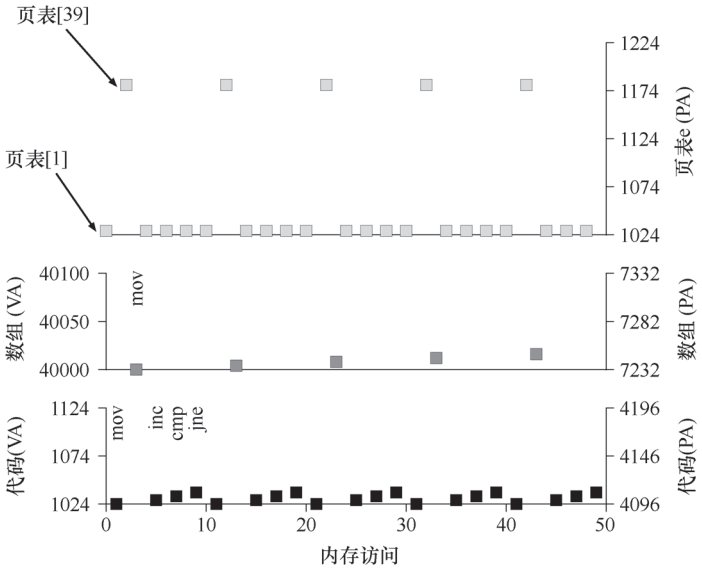


图 18.7　虚拟（和物理）内存追踪

看看你是否可以理解在这个可视化中出现的模式。特别是，随着循环继续，超过前 5 次迭代，会发生什么变化？哪些新的内存位置将被访问？你能弄明白吗？

这只是最简单的例子（只有几行 C 代码），但你可能已经能够感觉到理解实际应用程序的实际内存行为的复杂性。别担心：它肯定会变得更糟，因为我们即将引入的机制只会使这个已经很复杂的机器更复杂。

18.6　小结

我们已经引入了分页（paging）的概念，作为虚拟内存挑战的解决方案。与以前的方法（如分段）相比，分页有许多优点。首先，它不会导致外部碎片，因为分页（按设计）将内存划分为固定大小的单元。其次，它非常灵活，支持稀疏虚拟地址空间。

然而，实现分页支持而不小心考虑，会导致较慢的机器（有许多额外的内存访问来访问页表）和内存浪费（内存被页表塞满而不是有用的应用程序数据）。因此，我们不得不努力想出一个分页系统，它不仅可以工作，而且工作得很好。幸运的是，接下来的两章将告诉我们如何去做。

参考资料

[KE+62]“One-level Storage System”

T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner IRE Trans. EC-11, 2 (1962), pp. 223-235

(Reprinted in Bell and Newell,“Computer Structures: Readings and Examples”McGraw-Hill, New York, 1971).
Atlas 开创了将内存划分为固定大小页面的想法，在许多方面，都是我们在现代计算机系统中看到的内存管理思想的早期形式。

[I09]“Intel 64 and IA-32 Architectures Software Developer’s Manuals”Intel, 2009
Available.
具体来说，要注意《卷 3A：系统编程指南第 1 部分》和《卷 3B：系统编程指南第 2 部分》。

[L78]“The Manchester Mark I and atlas: a historical perspective”
S. H. Lavington
Communications of the ACM archive Volume 21, Issue 1 (January 1978), pp. 4-12 Special issue on computer architecture
本文是一些重要计算机系统发展历史的回顾。我们在美国有时会忘记，这些新想法中的许多来自其他国家。

作业

在这个作业中，你将使用一个简单的程序（名为 paging-linear-translate.py），来看看你是否理解了简单的虚拟—物理地址转换如何与线性页表一起工作。详情请参阅 README 文件。

问题

1．在做地址转换之前，让我们用模拟器来研究线性页表在给定不同参数的情况下如何改变大小。在不同参数变化时，计算线性页表的大小。一些建议输入如下，通过使用-v 标志，你可以看到填充了多少个页表项。

首先，要理解线性页表大小如何随着地址空间的增长而变化：

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
```

然后，理解线性页面大小如何随页大小的增长而变化：

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
```

在运行这些命令之前，请试着想想预期的趋势。页表大小如何随地址空间的增长而改变？随着页大小的增长呢？为什么一般来说，我们不应该使用很大的页呢？

2．现在让我们做一些地址转换。从一些小例子开始，使用-u 标志更改分配给地址空间的页数。例如：

```
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
```

如果增加每个地址空间中的页的百分比，会发生什么？

3．现在让我们尝试一些不同的随机种子，以及一些不同的（有时相当疯狂的）地址空间参数：

```
paging-linear-translate.py -P 8  -a 32   -p 1024 -v -s 1
paging-linear-translate.py -P 8k -a 32k  -p 1m   -v -s 2
paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
```

哪些参数组合是不现实的？为什么？

4．利用该程序尝试其他一些问题。你能找到让程序无法工作的限制吗？例如，如果地址空间大小大于物理内存，会发生什么情况？

第 19 章　分页：快速地址转换（TLB）

使用分页作为核心机制来实现虚拟内存，可能会带来较高的性能开销。因为要使用分页，就要将内存地址空间切分成大量固定大小的单元（页），并且需要记录这些单元的地址映射信息。因为这些映射信息一般存储在物理内存中，所以在转换虚拟地址时，分页逻辑上需要一次额外的内存访问。每次指令获取、显式加载或保存，都要额外读一次内存以得到转换信息，这慢得无法接受。因此我们面临如下问题：

关键问题：如何加速地址转换

如何才能加速虚拟地址转换，尽量避免额外的内存访问？需要什么样的硬件支持？操作系统该如何支持？

想让某些东西更快，操作系统通常需要一些帮助。帮助常常来自操作系统的老朋友：硬件。我们要增加所谓的（由于历史原因[CP78]）地址转换旁路缓冲存储器（translation-lookaside buffer，TLB[CG68,C95]），它就是频繁发生的虚拟到物理地址转换的硬件缓存（cache）。因此，更好的名称应该是地址转换缓存（address-translation cache）。对每次内存访问，硬件先检查 TLB，看看其中是否有期望的转换映射，如果有，就完成转换（很快），不用访问页表（其中有全部的转换映射）。TLB 带来了巨大的性能提升，实际上，因此它使得虚拟内存成为可能[C95]。

19.1　TLB 的基本算法

图 19.1 展示了一个大体框架，说明硬件如何处理虚拟地址转换，假定使用简单的线性页表（linear page table，即页表是一个数组）和硬件管理的 TLB（hardware-managed TLB，即硬件承担许多页表访问的责任，下面会有更多解释）。

```
1    VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2    (Success, TlbEntry) = TLB_Lookup(VPN)
3    if (Success == True)    // TLB Hit
4        if (CanAccess(TlbEntry.ProtectBits) == True)
5            Offset   = VirtualAddress & OFFSET_MASK
6            PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7            AccessMemory(PhysAddr)
8        else
9            RaiseException(PROTECTION_FAULT)
10   else    // TLB Miss
11       PTEAddr = PTBR + (VPN * sizeof(PTE))
12       PTE = AccessMemory(PTEAddr)
```

```
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else if (CanAccess(PTE.ProtectBits) == False)
16          RaiseException(PROTECTION_FAULT)
17      else
18          TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19          RetryInstruction()
```

图 19.1 TLB 控制流算法

硬件算法的大体流程如下：首先从虚拟地址中提取页号（VPN）（见图 19.1 第 1 行），然后检查 TLB 是否有该 VPN 的转换映射（第 2 行）。如果有，我们有了 TLB 命中（TLB hit），这意味着 TLB 有该页的转换映射。成功！接下来我们就可以从相关的 TLB 项中取出页帧号（PFN），与原来虚拟地址中的偏移量组合形成期望的物理地址（PA），并访问内存（第 5～7 行），假定保护检查没有失败（第 4 行）。

如果 CPU 没有在 TLB 中找到转换映射（TLB 未命中），我们有一些工作要做。在本例中，硬件访问页表来寻找转换映射（第 11～12 行），并用该转换映射更新 TLB（第 18 行），假设该虚拟地址有效，而且我们有相关的访问权限（第 13、15 行）。上述系列操作开销较大，主要是因为访问页表需要额外的内存引用（第 12 行）。最后，当 TLB 更新成功后，系统会重新尝试该指令，这时 TLB 中有了这个转换映射，内存引用得到很快处理。

TLB 和其他缓存相似，前提是在一般情况下，转换映射会在缓存中（即命中）。如果是这样，只增加了很少的开销，因为 TLB 处理器核心附近，设计的访问速度很快。如果 TLB 未命中，就会带来很大的分页开销。必须访问页表来查找转换映射，导致一次额外的内存引用（或者更多，如果页表更复杂）。如果这经常发生，程序的运行就会显著变慢。相对于大多数 CPU 指令，内存访问开销很大，TLB 未命中导致更多内存访问。因此，我们希望尽可能避免 TLB 未命中。

19.2 示例：访问数组

为了弄清楚 TLB 的操作，我们来看一个简单虚拟地址追踪，看看 TLB 如何提高它的性能。在本例中，假设有一个由 10 个 4 字节整型数组成的数组，起始虚地址是 100。进一步假定，有一个 8 位的小虚地址空间，页大小为 16B。我们可以把虚地址划分为 4 位的 VPN（有 16 个虚拟内存页）和 4 位的偏移量（每个页中有 16 个字节）。

图 19.2 展示了该数组的布局，在系统的 16 个 16 字节的页上。如你所见，数组的第一项（a[0]）开始于（VPN=06，offset=04），只有 3 个 4 字节整型数存放在该页。数组在下一页（VPN=07）继续，其中有接下来 4 项（a[3] … a[6]）。10 个元素的数组的最后 3 项（a[7] … a[9]）位于地址空间的下一页（VPN=08）。

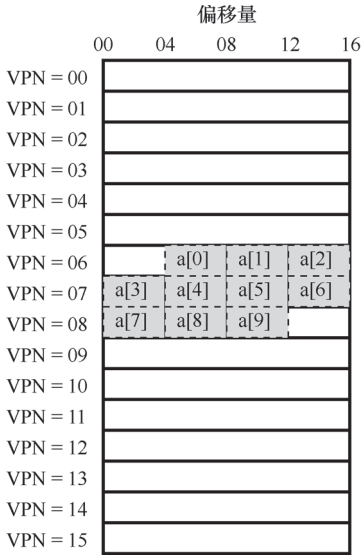


图 19.2 示例：小地址空间中的一个数组

现在考虑一个简单的循环，访问数组中的每个元素，类似下面的 C 程序：

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

简单起见，我们假装循环产生的内存访问只是针对数组（忽略变量 *i* 和 *sum*，以及指令本身）。当访问第一个数组元素（a[0]）时，CPU 会看到载入虚存地址 100。硬件从中提取 VPN（VPN=06），然后用它来检查 TLB，寻找有效的转换映射。假设这里是程序第一次访问该数组，结果是 TLB 未命中。

接下来访问 a[1]，这里有好消息：TLB 命中！因为数组的第二个元素在第一个元素之后，它们在同一页。因为我们之前访问数组的第一个元素时，已经访问了这一页，所以 TLB 中缓存了该页的转换映射。因此成功命中。访问 a[2]同样成功（再次命中），因为它和 a[0]、a[1]位于同一页。

遗憾的是，当程序访问 a[3]时，会导致 TLB 未命中。但同样，接下来几项（a[4] … a[6]）都会命中 TLB，因为它们位于内存中的同一页。

最后，访问 a[7]会导致最后一次 TLB 未命中。系统会再次查找页表，弄清楚这个虚拟页在物理内存中的位置，并相应地更新 TLB。最后两次访问（a[8]、a[9]）受益于这次 TLB 更新，当硬件在 TLB 中查找它们的转换映射时，两次都命中。

我们来总结一下这 10 次数组访问操作中 TLB 的行为表现：未命中、命中、命中、未命中、命中、命中、命中、未命中、命中、命中。命中的次数除以总的访问次数，得到 TLB 命中率（hit rate）为 70%。尽管这不是很高（实际上，我们希望命中率接近 100%），但也不是零，是零我们就要奇怪了。即使这是程序首次访问该数组，但得益于空间局部性（spatial locality），TLB 还是提高了性能。数组的元素被紧密存放在几页中（即它们在空间中紧密相邻），因此只有对页中第一个元素的访问才会导致 TLB 未命中。

也要注意页大小对本例结果的影响。如果页大小变大一倍（32 字节，而不是 16），数组访问遇到的未命中更少。典型页的大小一般为 4KB，这种情况下，密集的、基于数组的访问会实现极好的 TLB 性能，每页的访问只会遇到一次未命中。

关于 TLB 性能还有最后一点：如果在这次循环后不久，该程序再次访问该数组，我们会看到更好的结果，假设 TLB 足够大，能缓存所需的转换映射：命中、命中、命中、命中、命中、命中、命中、命中、命中、命中。在这种情况下，由于时间局部性（temporal locality），即在短时间内对内存项再次引用，所以 TLB 的命中率会很高。类似其他缓存，TLB 的成功依赖于空间和时间局部性。如果某个程序表现出这样的局部性（许多程序是这样），TLB 的命中率可能很高。

提示：尽可能利用缓存

缓存是计算机系统中最基本的性能改进技术之一，一次又一次地用于让“常见的情况更快”[HP06]。硬件缓存背后的思想是利用指令和数据引用的局部性（locality）。通常有两种局部性：时间局部性（temporal locality）和空间局部性（spatial locality）。时间局部性是指，最近访问过的指令或数据项可能很快会再次访问。想想循环中的循环变量或指令，它们被多次反复访问。空间局部性是指，当程序访问

内存地址 x 时，可能很快会访问邻近 x 的内存。想想遍历某种数组，访问一个接一个的元素。当然，这些性质取决于程序的特点，并不是绝对的定律，而更像是一种经验法则。

硬件缓存，无论是指令、数据还是地址转换（如 TLB），都利用了局部性，在小而快的芯片内存储器中保存一份内存副本。处理器可以先检查缓存中是否存在就近的副本，而不是必须访问（缓慢的）内存来满足请求。如果存在，处理器就可以很快地访问它（例如在几个 CPU 时钟内），避免花很多时间来访问内存（好多纳秒）。

你可能会疑惑：既然像 TLB 这样的缓存这么好，为什么不做更大的缓存，装下所有的数据？可惜的是，这里我们遇到了更基本的定律，就像物理定律那样。如果想要快速地缓存，它就必须小，因为光速和其他物理限制会起作用。大的缓存注定慢，因此无法实现目的。所以，我们只能用小而快的缓存。剩下的问题就是如何利用好缓存来提升性能。

19.3 谁来处理 TLB 未命中

有一个问题我们必须回答：谁来处理 TLB 未命中？可能有两个答案：硬件或软件（操作系统）。以前的硬件有复杂的指令集（有时称为复杂指令集计算机，Complex-Instruction Set Computer，CISC），造硬件的人不太相信那些搞操作系统的人。因此，硬件全权处理 TLB 未命中。为了做到这一点，硬件必须知道页表在内存中的确切位置（通过页表基址寄存器，page-table base register，在图 19.1 的第 11 行使用），以及页表的确切格式。发生未命中时，硬件会“遍历”页表，找到正确的页表项，取出想要的转换映射，用它更新 TLB，并重试该指令。这种“旧”体系结构有硬件管理的 TLB，一个例子是 x86 架构，它采用固定的多级页表（multi-level page table，详见第 20 章），当前页表由 CR3 寄存器指出[I09]。

更现代的体系结构（例如，MIPS R10k[H93]、Sun 公司的 SPARC v9[WG00]，都是精简指令集计算机，Reduced-Instruction Set Computer，RISC），有所谓的软件管理 TLB（software-managed TLB）。发生 TLB 未命中时，硬件系统会抛出一个异常（见图 19.3 第 11 行），这会暂停当前的指令流，将特权级提升至内核模式，跳转至陷阱处理程序（trap handler）。接下来你可能已经猜到了，这个陷阱处理程序是操作系统的一段代码，用于处理 TLB 未命中。这段代码在运行时，会查找页表中的转换映射，然后用特别的“特权”指令更新 TLB，并从陷阱返回。此时，硬件会重试该指令（导致 TLB 命中）。

```
1    VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2    (Success, TlbEntry) = TLB_Lookup(VPN)
3    if (Success == True)    // TLB Hit
4        if (CanAccess(TlbEntry.ProtectBits) == True)
5            Offset    = VirtualAddress & OFFSET_MASK
6            PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7            Register = AccessMemory(PhysAddr)
8        else
9            RaiseException(PROTECTION_FAULT)
10   else                    // TLB Miss
11       RaiseException(TLB_MISS)
```

图 19.3 TLB 控制流算法（操作系统处理）

接下来讨论几个重要的细节。首先，这里的从陷阱返回指令稍稍不同于之前提到的服务于系统调用的从陷阱返回。在后一种情况下，从陷阱返回应该继续执行陷入操作系统之后那条指令，就像从函数调用返回后，会继续执行此次调用之后的语句。在前一种情况下，在从 TLB 未命中的陷阱返回后，硬件必须从导致陷阱的指令继续执行。这次重试因此导致该指令再次执行，但这次会命中 TLB。因此，根据陷阱或异常的原因，系统在陷入内核时必须保存不同的程序计数器，以便将来能够正确地继续执行。

第二，在运行 TLB 未命中处理代码时，操作系统需要格外小心避免引起 TLB 未命中的无限递归。有很多解决方案，例如，可以把 TLB 未命中陷阱处理程序直接放到物理内存中 [它们没有映射过（unmapped），不用经过地址转换]。或者在 TLB 中保留一些项，记录永久有效的地址转换，并将其中一些永久地址转换槽块留给处理代码本身，这些被监听的（wired）地址转换总是会命中 TLB。

软件管理的方法，主要优势是灵活性：操作系统可以用任意数据结构来实现页表，不需要改变硬件。另一个优势是简单性。从 TLB 控制流中可以看出（见图 19.3 的第 11 行，对比图 19.1 的第 11～19 行），硬件不需要对未命中做太多工作，它抛出异常，操作系统的未命中处理程序会负责剩下的工作。

补充：RISC 与 CISC

在 20 世纪 80 年代，计算机体系结构领域曾发生过一场激烈的讨论。一方是 CISC 阵营，即复杂指令集计算（Complex Instruction Set Computing），另一方是 RISC，即精简指令集计算（Reduced Instruction Set Computing）[PS81]。RISC 阵营以 Berkeley 的 David Patterson 和 Stanford 的 John Hennessy 为代表（他们写了一些非常著名的书[HP06]），尽管后来 John Cocke 凭借他在 RISC 上的早期工作 [CM00]获得了图灵奖。

CISC 指令集倾向于拥有许多指令，每条指令比较强大。例如，你可能看到一个字符串拷贝，它接受两个指针和一个长度，将一些字节从源拷贝到目标。CISC 背后的思想是，指令应该是高级原语，这让汇编语言本身更易于使用，代码更紧凑。

RISC 指令集恰恰相反。RISC 背后的关键观点是，指令集实际上是编译器的最终目标，所有编译器实际上需要少量简单的原语，可以用于生成高性能的代码。因此，RISC 倡导者们主张，尽可能从硬件中拿掉不必要的东西（尤其是微代码），让剩下的东西简单、统一、快速。

早期的 RISC 芯片产生了巨大的影响，因为它们明显更快[BC91]。人们写了很多论文，一些相关的公司相继成立（例如 MIPS 和 Sun 公司）。但随着时间的推移，像 Intel 这样的 CISC 芯片制造商采纳了许多 RISC 芯片的优点，例如添加了早期流水线阶段，将复杂的指令转换为一些微指令，于是它们可以像 RISC 的方式运行。这些创新，加上每个芯片中晶体管数量的增长，让 CISC 保持了竞争力。争论最后平息了，现在两种类型的处理器都可以跑得很快。

19.4 TLB 的内容

我们来详细看一下硬件 TLB 中的内容。典型的 TLB 有 32 项、64 项或 128 项，并且是全相联的（fully associative）。基本上，这就意味着一条地址映射可能存在 TLB 中的任意位置，硬件会并行地查找 TLB，找到期望的转换映射。一条 TLB 项内容可能像下面这样：

VPN | PFN | 其他位

注意，VPN 和 PFN 同时存在于 TLB 中，因为一条地址映射可能出现在任意位置（用硬件的术语，TLB 被称为全相联的（fully-associative）缓存）。硬件并行地查找这些项，看看是否有匹配。

补充：TLB 的有效位!=页表的有效位

常见的错误是混淆 TLB 的有效位和页表的有效位。在页表中，如果一个页表项（PTE）被标记为无效，就意味着该页并没有被进程申请使用，正常运行的程序不应该访问该地址。当程序试图访问这样的页时，就会陷入操作系统，操作系统会杀掉该进程。

TLB 的有效位不同，只是指出 TLB 项是不是有效的地址映射。例如，系统启动时，所有的 TLB 项通常被初始化为无效状态，因为还没有地址转换映射被缓存在这里。一旦启用虚拟内存，当程序开始运行，访问自己的虚拟地址，TLB 就会慢慢地被填满，因此有效的项很快会充满 TLB。

TLB 有效位在系统上下文切换时起到了很重要的作用，后面我们会进一步讨论。通过将所有 TLB 项设置为无效，系统可以确保将要运行的进程不会错误地使用前一个进程的虚拟到物理地址转换映射。

更有趣的是“其他位”。例如，TLB 通常有一个有效（valid）位，用来标识该项是不是有效地转换映射。通常还有一些保护（protection）位，用来标识该页是否有访问权限。例如，代码页被标识为可读和可执行，而堆的页被标识为可读和可写。还有其他一些位，包括地址空间标识符（address-space identifier）、脏位（dirty bit）等。下面会介绍更多信息。

19.5 上下文切换时对 TLB 的处理

有了 TLB，在进程间切换时（因此有地址空间切换），会面临一些新问题。具体来说，TLB 中包含的虚拟到物理的地址映射只对当前进程有效，对其他进程是没有意义的。所以在发生进程切换时，硬件或操作系统（或二者）必须注意确保即将运行的进程不要误读了之前进程的地址映射。

为了更好地理解这种情况，我们来看一个例子。当一个进程（P1）正在运行时，假设 TLB 缓存了对它有效的地址映射，即来自 P1 的页表。对这个例子，假设 P1 的 10 号虚拟页映射到了 100 号物理帧。

在这个例子中，假设还有一个进程（P2），操作系统不久后决定进行一次上下文切换，运行 P2。这里假定 P2 的 10 号虚拟页映射到 170 号物理帧。如果这两个进程的地址映射都在 TLB 中，TLB 的内容如表 19.1 所示。

表 19.1 TLB 的内容

VPN	PFN	有效位	保护位
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

在上面的 TLB 中，很明显有一个问题：VPN 10 被转换成了 PFN 100（P1）和 PFN 170

（P2），但硬件分不清哪个项属于哪个进程。所以我们还需要做一些工作，让 TLB 正确而高效地支持跨多进程的虚拟化。因此，关键问题是：

关键问题：进程切换时如何管理 TLB 的内容

如果发生进程间上下文切换，上一个进程在 TLB 中的地址映射对于即将运行的进程是无意义的。硬件或操作系统应该做些什么来解决这个问题呢？

这个问题有一些可能的解决方案。一种方法是在上下文切换时，简单地清空（flush）TLB，这样在新进程运行前 TLB 就变成了空的。如果是软件管理 TLB 的系统，可以在发生上下文切换时，通过一条显式（特权）指令来完成。如果是硬件管理 TLB，则可以在页表基址寄存器内容发生变化时清空 TLB（注意，在上下文切换时，操作系统必须改变页表基址寄存器（PTBR）的值）。不论哪种情况，清空操作都是把全部有效位（valid）置为 0，本质上清空了 TLB。

上下文切换的时候清空 TLB，这是一个可行的解决方案，进程不会再读到错误的地址映射。但是，有一定开销：每次进程运行，当它访问数据和代码页时，都会触发 TLB 未命中。如果操作系统频繁地切换进程，这种开销会很高。

为了减少这种开销，一些系统增加了硬件支持，实现跨上下文切换的 TLB 共享。比如有的系统在 TLB 中添加了一个地址空间标识符（Address Space Identifier，ASID）。可以把 ASID 看作是进程标识符（Process Identifier，PID），但通常比 PID 位数少（PID 一般 32 位，ASID 一般是 8 位）。

如果仍以上面的 TLB 为例，加上 ASID，很清楚不同进程可以共享 TLB 了：只要 ASID 字段来区分原来无法区分的地址映射。表 19.2 展示了添加 ASID 字段后的 TLB。

表 19.2 添加 ASID 字段后的 TLB

VPN	PFN	有效位	保护位	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

因此，有了地址空间标识符，TLB 可以同时缓存不同进程的地址空间映射，没有任何冲突。当然，硬件也需要知道当前是哪个进程正在运行，以便进行地址转换，因此操作系统在上下文切换时，必须将某个特权寄存器设置为当前进程的 ASID。

补充一下，你可能想到了另一种情况，TLB 中某两项非常相似。在表 19.3 中，属于两个不同进程的两项，将两个不同的 VPN 指向了相同的物理页。

表 19.3 包含相似两项的 TLB

VPN	PFN	有效位	保护位	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

如果两个进程共享同一物理页（例如代码段的页），就可能出现这种情况。在上面的例子中，进程 P1 和进程 P2 共享 101 号物理页，但是 P1 将自己的 10 号虚拟页映射到该物理页，而 P2 将自己的 50 号虚拟页映射到该物理页。共享代码页（以二进制或共享库的方式）是有用的，因为它减少了物理页的使用，从而减少了内存开销。

19.6 TLB 替换策略

TLB 和其他缓存一样，还有一个问题要考虑，即缓存替换（cache replacement）。具体来说，向 TLB 中插入新项时，会替换（replace）一个旧项，这样问题就来了：应该替换那一个？

关键问题：如何设计 TLB 替换策略

在向 TLB 添加新项时，应该替换哪个旧项？目标当然是减小 TLB 未命中率（或提高命中率），从而改进性能。

在讨论页换出到磁盘的问题时，我们将详细研究这样的策略。这里我们先简单指出几个典型的策略。一种常见的策略是替换最近最少使用（least-recently-used，LRU）的项。LRU 尝试利用内存引用流中的局部性，假定最近没有用过的项，可能是好的换出候选项。另一种典型策略就是随机（random）策略，即随机选择一项换出去。这种策略很简单，并且可以避免一种极端情况。例如，一个程序循环访问 n+1 个页，但 TLB 大小只能存放 n 个页。这时之前看似“合理”的 LRU 策略就会表现得不可理喻，因为每次访问内存都会触发 TLB 未命中，而随机策略在这种情况下就好很多。

19.7 实际系统的 TLB 表项

最后，我们简单看一下真实的 TLB。这个例子来自 MIPS R4000[H93]，它是一种现代的系统，采用软件管理 TLB。图 19.4 展示了稍微简化的 MIPS TLB 项。

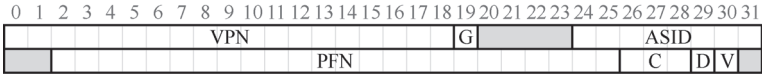


图 19.4 MIPS 的 TLB 项

MIPS R4000 支持 32 位的地址空间，页大小为 4KB。所以在典型的虚拟地址中，预期会看到 20 位的 VPN 和 12 位的偏移量。但是，你可以在 TLB 中看到，只有 19 位的 VPN。事实上，用户地址只占地址空间的一半（剩下的留给内核），所以只需要 19 位的 VPN。VPN 转换成最大 24 位的物理帧号（PFN），因此可以支持最多有 64GB 物理内存（2^{24} 个 4KB 内存页）的系统。

MIPS TLB 还有一些有趣的标识位。比如全局位（Global，G），用来指示这个页是不是所有进程全局共享的。因此，如果全局位置为 1，就会忽略 ASID。我们也看到了 8 位的 ASID，

操作系统用它来区分进程空间（像上面介绍的一样）。这里有一个问题：如果正在运行的进程数超过 256（2^8）个怎么办？最后，我们看到 3 个一致性位（Coherence，C），决定硬件如何缓存该页（其中一位超出了本书的范围）；脏位（dirty），表示该页是否被写入新数据（后面会介绍用法）；有效位（valid），告诉硬件该项的地址映射是否有效。还有没在图 19.4 中展示的页掩码（page mask）字段，用来支持不同的页大小。后面会介绍，为什么更大的页可能有用。最后，64 位中有一些未使用（图 19.4 中灰色部分）。

MIPS 的 TLB 通常有 32 项或 64 项，大多数提供给用户进程使用，也有一小部分留给操作系统使用。操作系统可以设置一个被监听的寄存器，告诉硬件需要为自己预留多少 TLB 槽。这些保留的转换映射，被操作系统用于关键时候它要使用的代码和数据，在这些时候，TLB 未命中可能会导致问题（例如，在 TLB 未命中处理程序中）。

由于 MIPS 的 TLB 是软件管理的，所以系统需要提供一些更新 TLB 的指令。MIPS 提供了 4 个这样的指令：TLBP，用来查找指定的转换映射是否在 TLB 中；TLBR，用来将 TLB 中的内容读取到指定寄存器中；TLBWI，用来替换指定的 TLB 项；TLBWR，用来随机替换一个 TLB 项。操作系统可以用这些指令管理 TLB 的内容。当然这些指令是特权指令，这很关键。如果用户程序可以任意修改 TLB 的内容，你可以想象一下会发生什么可怕的事情。

提示：RAM 不总是 RAM（Culler 定律）

随机存取存储器（Random-Access Memory，RAM）暗示你访问 RAM 的任意部分都一样快。虽然一般这样想 RAM 没错，但因为 TLB 这样的硬件/操作系统功能，访问某些内存页的开销较大，尤其是没有被 TLB 缓存的页。因此，最好记住这个实现的窍门：RAM 不总是 RAM。有时候随机访问地址空间，尤其是 TLB 没有缓存的页，可能导致严重的性能损失。因为我的一位导师 David Culler 过去常常指出 TLB 是许多性能问题的源头，所以我们以他来命名这个定律：Culler 定律（Culler's Law）。

19.8 小结

我们了解了硬件如何让地址转换更快的方法。通过增加一个小的、芯片内的 TLB 作为地址转换的缓存，大多数内存引用就不用访问内存中的页表了。因此，在大多数情况下，程序的性能就像内存没有虚拟化一样，这是操作系统的杰出成就，当然对现代操作系统中的分页非常必要。

但是，TLB 也不能满足所有的程序需求。具体来说，如果一个程序短时间内访问的页数超过了 TLB 中的页数，就会产生大量的 TLB 未命中，运行速度就会变慢。这种现象被称为超出 TLB 覆盖范围（TLB coverage），这对某些程序可能是相当严重的问题。解决这个问题的一种方案是支持更大的页，把关键数据结构放在程序地址空间的某些区域，这些区域被映射到更大的页，使 TLB 的有效覆盖率增加。对更大页的支持通常被数据库管理系统（Database Management System，DBMS）这样的程序利用，它们的数据结构比较大，而且是随机访问。

另一个 TLB 问题值得一提：访问 TLB 很容易成为 CPU 流水线的瓶颈，尤其是有所谓

的物理地址索引缓存（physically-indexed cache）。有了这种缓存，地址转换必须发生在访问该缓存之前，这会让操作变慢。为了解决这个潜在的问题，人们研究了各种巧妙的方法，用虚拟地址直接访问缓存，从而在缓存命中时避免昂贵的地址转换步骤。像这种虚拟地址索引缓存（virtually-indexed cache）解决了一些性能问题，但也为硬件设计带来了新问题。更多细节请参考 Wiggins 的调查[W03]。

参考资料

[BC91]“Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization”
D. Bhandarkar and Douglas W. Clark
Communications of the ACM, September 1991
关于 RISC 和 CISC 的一篇很好的、公平的比较性的文章。本质上，在类似的硬件上，RISC 的性能是 CISC 的 3 倍。

[CM00]“The evolution of RISC technology at IBM”John Cocke and V. Markstein
IBM Journal of Research and Development, 44:1/2
IBM 801 的概念和工作总结，许多人认为它是第一款真正的 RISC 微处理器。

[C95]“The Core of the Black Canyon Computer Corporation”John Couleur
IEEE Annals of History of Computing, 17:4, 1995
在这个引人入胜的计算历史讲义中，Couleur 谈到了他在 1964 年为通用电气公司工作时如何发明了 TLB，以及与麻省理工学院的 MAC 项目人员之间偶然而幸运的合作。

[CG68]“Shared-access Data Processing System”John F. Couleur and Edward L. Glaser
Patent 3412382, November 1968
包含用关联存储器存储地址转换的想法的专利。据 Couleur 说，这个想法产生于 1964 年。

[CP78]“The architecture of the IBM System/370”
R.P. Case and A. Padegs
Communications of the ACM. 21:1, 73-96, January 1978
也许是第一篇使用术语“地址转换旁路缓冲存储器（translation lookaside buffer）”的文章。 这个名字来源于缓存的历史名称，即旁路缓冲存储器（lookaside buffer），在曼彻斯特大学开发 Atlas 系统的人这样叫它。地址转换缓存因此成为地址转换旁路缓冲存储器。尽管术语“旁路缓冲存储器”不再流行，但 TLB 似乎仍在持续使用，其原因不明。

[H93]“MIPS R4000 Microprocessor User’s Manual”. Joe Heinrich, Prentice-Hall, June 1993

[HP06]“Computer Architecture: A Quantitative Approach” John Hennessy and David Patterson
Morgan-Kaufmann, 2006
一本关于计算机架构的好书。我们对经典的第 1 版特别有感情。

[I09]“Intel 64 and IA-32 Architectures Software Developer’s Manuals”Intel, 2009
Available.
尤其要注意《卷 3A：系统编程指南第 1 部分》和《卷 3B：系统编程指南第 2 部分》。

[PS81]“RISC-I: A Reduced Instruction Set VLSI Computer”
D.A．Patterson and C.H. Sequin ISCA ’81, Minneapolis, May 1981
这篇文章介绍了 RISC 这个术语，开启了为性能而简化计算机芯片的研究狂潮。

[SB92]“CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking”
Rafael H. Saavedra-Barrera
EECS Department, University of California, Berkeley Technical Report No. UCB/CSD-92-684, February 1992
一篇卓越的论文，探讨将应用的执行时间分解为组成部分，知道每个部分的成本，从而预测应用的执行时间。也许这项工作最有趣的部分是衡量缓存层次结构细节的工具（在第 5 章中介绍）。一定要看看其中的精彩图表。

[W03]“A Survey on the Interaction Between Caching, Translation and Protection”Adam Wiggins
University of New South Wales TR UNSW-CSE-TR-0321, August, 2003
关于 TLB 如何与 CPU 管道的其他部分（即硬件缓存）进行交互的一次很好的调查。

[WG00]“The SPARC Architecture Manual: Version 9”David L. Weaver and Tom Germond, September 2000
SPARC International, San Jose, California

作业（测量）

本次作业要测算一下 TLB 的容量和访问 TLB 的开销。这个想法参考了 Saavedra-Barrera 的工作[SB92]，他用设计了一个简单而漂亮的用户级程序，来测算缓存层级结构的方方面面。更多细节请阅读他的论文。

基本原理就是访问一个跨多个内存页的大尺寸数据结构（例如数组），然后统计访问时间。例如，假设一个机器的 TLB 大小为 4（这很小，但对这个讨论有用）。如果写一个程序访问 4 个或更少的页，每次访问都会命中 TLB，因此相对较快。但是，如果在一个循环里反复访问 5 个或者更多的页，每次访问的开销就会突然跃升，因为发生 TLB 未命中。

循环遍历数组一次的基本代码应该像这样：

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump) {
    a[i] += 1;
}
```

在这个循环中，数组 a 中每页的一个整数被更新，直到 NUMPAGES 指定的页数。通过对这个循环反复执行计时（比如，在外层循环中执行几亿次这个循环，或者运行几秒钟所需的次数），就可以计算出平均每次访问所用的时间。随着 NUMPAGES 的增加，寻找开销

的跃升，可以大致确定第一级 TLB 的大小，确定是否存在第二级 TLB（如果存在，确定它的大小），总体上很好地理解 TLB 命中和未命中对于性能的影响。

图 19.5 是一张示意图。

从图 19.5 中可以看出，如果只访问少数页（8 或更少），平均访问时间大约是 5ns。如果访问 16 页或更多，每次访问时间突然跃升到 20ns。最后一次开销跃升发生在 1024 页时，这时每次访问大约要 70ns。通过这些数据，我们可以总结出这是一个二级的 TLB，第一级较小（大约能存放 8～16 项），第二级较大，但较慢（大约能存放 512 项）。第一级 TLB 的命中和完全未命中的总体差距非常大，大约有 14 倍。TLB 的性能很重要！

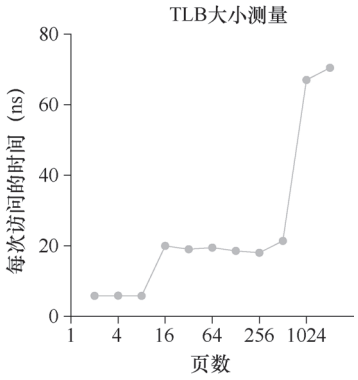


图 19.5　发现 TLB 大小和未命中开销

问题

1．为了计时，可能需要一个计时器，例如 gettimeofday()提供的。这种计时器的精度如何？操作要花多少时间，才能让你对它精确计时？（这有助于确定需要循环多少次，反复访问内存页，才能对它成功计时。）

2．写一个程序，命名为 tlb.c，大体测算一下每个页的平均访问时间。程序的输入参数有：页的数目和尝试的次数。

3．用你喜欢的脚本语言（csh、Python 等）写一段脚本来运行这个程序，当访问页面从 1 增长到几千，也许每次迭代都乘 2。在不同的机器上运行这段脚本，同时收集相应数据。需要试多少次才能获得可信的测量结果？

4．接下来，将结果绘图，类似于上图。可以用 ploticus 这样的好工具画图。可视化使数据更容易理解，你认为是什么原因？

5．要注意编译器优化带来的影响。编译器做各种聪明的事情，包括优化掉循环，如果循环中增加的变量后续没有使用。如何确保编译器不优化掉你写的 TLB 大小测算程序的主循环？

6．还有一个需要注意的地方，今天的计算机系统大多有多个 CPU，每个 CPU 当然有自己的 TLB 结构。为了得到准确的测量数据，我们需要只在一个 CPU 上运行程序，避免调度器把进程从一个 CPU 调度到另一个去运行。如何做到？（提示：在 Google 上搜索“pinning a thread”相关的信息）如果没有这样做，代码从一个 CPU 移到了另一个，会发生什么情况？

7．另一个可能发生的问题与初始化有关。如果在访问数组 a 之前没有初始化，第一次访问将非常耗时，由于初始访问开销，比如要求置 0。这会影响你的代码及其计时吗？如何抵消这些潜在的开销？

第 20 章　分页：较小的表

我们现在来解决分页引入的第二个问题：页表太大，因此消耗的内存太多。让我们从线性页表开始。你可能会记得[①]，线性页表变得相当大。假设一个 32 位地址空间（232 字节），4KB（212 字节）的页和一个 4 字节的页表项。一个地址空间中大约有一百万个虚拟页面（232/212）。乘以页表项的大小，你会发现页表大小为 4MB。回想一下：通常系统中的每个进程都有一个页表！有一百个活动进程（在现代系统中并不罕见），就要为页表分配数百兆的内存！因此，要寻找一些技术来减轻这种沉重的负担。有很多方法，所以我们开始吧。但先看我们的关键问题：

关键问题：如何让页表更小？

简单的基于数组的页表（通常称为线性页表）太大，在典型系统上占用太多内存。如何让页表更小？关键的思路是什么？由于这些新的数据结构，会出现什么效率影响？

20.1　简单的解决方案：更大的页

可以用一种简单的方法减小页表大小：使用更大的页。再以 32 位地址空间为例，但这次假设用 16KB 的页。因此，会有 18 位的 VPN 加上 14 位的偏移量。假设每个页表项（4 字节）的大小相同，现在线性页表中有 218 个项，因此每个页表的总大小为 1MB，页表缩到四分之一。

补充：多种页大小

另外请注意，许多体系结构（例如 MIPS、SPARC、x86-64）现在都支持多种页大小。通常使用一个小的（4KB 或 8KB）页大小。但是，如果一个“聪明的”应用程序请求它，则可以为地址空间的特定部分使用一个大型页（例如，大小为 4MB），从而让这些应用程序可以将常用的（大型的）数据结构放入这样的空间，同时只占用一个 TLB 项。这种类型的大页在数据库管理系统和其他高端商业应用程序中很常见。然而，多种页面大小的主要原因并不是为了节省页表空间。这是为了减少 TLB 的压力，让程序能够访问更多的地址空间而不会遭受太多的 TLB 未命中之苦。然而，正如研究人员已经说明[N+02]一样，采用多种页大小，使操作系统虚拟内存管理程序显得更复杂，因此，有时只需向应用程序暴露一个新接口，让它们直接请求大内存页，这样最容易。

① 或者实际上，你可能记不起来了。分页这件事正在失控，不是吗？虽然这样说，但在进入解决方案之前，一定要确保你理解了正在解决的问题。事实上，如果你理解了问题，通常可以自己推导出解决方案。在这里，问题应该很清楚：简单的线性（基于数组的）页表太大了。

然而，这种方法的主要问题在于，大内存页会导致每页内的浪费，这被称为内部碎片（internal fragmentation）问题（因为浪费在分配单元内部）。因此，结果是应用程序会分配页，但只用每页的一小部分，而内存很快就会充满这些过大的页。因此，大多数系统在常见的情况下使用相对较小的页大小：4KB（如 x86）或 8KB（如 SPARCv9）。问题不会如此简单地解决。

20.2 混合方法：分页和分段

在生活中，每当有两种合理但不同的方法时，你应该总是研究两者的结合，看看能否两全其美。我们称这种组合为杂合（hybrid）。例如，为什么只吃巧克力或简单的花生酱，而不是将两者结合起来，就像可爱的花生酱巧克力杯[M28]？

多年前，Multics 的创造者（特别是 Jack Dennis）在构建 Multics 虚拟内存系统时，偶然发现了这样的想法[M07]。具体来说，Dennis 想到将分页和分段相结合，以减少页表的内存开销。更仔细地看看典型的线性页表，就可以理解为什么这可能有用。假设我们有一个地址空间，其中堆和栈的使用部分很小。例如，我们使用一个 16KB 的小地址空间和 1KB 的页（见图 20.1）。该地址空间的页表如表 20.1 所示。

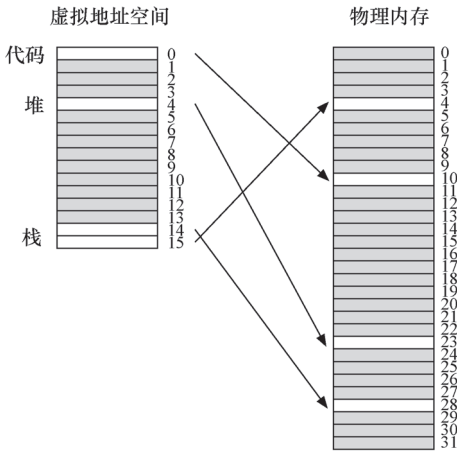


图 20.1 1KB 的页和 16KB 的地址空间

这个例子假定单个代码页（VPN 0）映射到物理页 10，单个堆页（VPN 4）映射到物理页 23，以及地址空间另一端的两个栈页（VPN 14 和 15）被分别映射到物理页 28 和 4。从图 20.1 中可以看到，大部分页表都没有使用，充满了无效的（invalid）项。真是太浪费了！这是一个微小的 16KB 地址空间。想象一下 32 位地址空间的页表和所有潜在的浪费空间！真的，不要想象这样的事情，太可怕了。

表 20.1 16KB 地址空间的页表

PFN	valid	prot	present	dirty
10	1	r-x	1	0
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
23	1	rw-	1	1
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—

续表

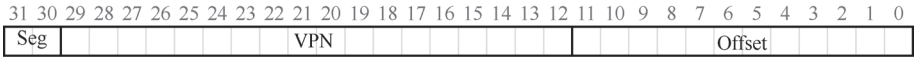
PFN	valid	prot	present	dirty
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
28	1	rw-	1	1
4	1	rw-	1	1

因此，我们的杂合方法不是为进程的整个地址空间提供单个页表，而是为每个逻辑分段提供一个。在这个例子中，我们可能有 3 个页表，地址空间的代码、堆和栈部分各有一个。

现在，回忆在分段中，有一个基址（base）寄存器，告诉我们每个段在物理内存中的位置，还有一个界限（bound）或限制（limit）寄存器，告诉我们该段的大小。在杂合方案中，我们仍然在 MMU 中拥有这些结构。在这里，我们使用基址不是指向段本身，而是保存该段的页表的物理地址。界限寄存器用于指示页表的结尾（即它有多少有效页）。

我们通过一个简单的例子来澄清。假设 32 位虚拟地址空间包含 4KB 页面，并且地址空间分为 4 个段。在这个例子中，我们只使用 3 个段：一个用于代码，另一个用于堆，还有一个用于栈。

要确定地址引用哪个段，我们会用地址空间的前两位。假设 00 是未使用的段，01 是代码段，10 是堆段，11 是栈段。因此，虚拟地址如下所示：



在硬件中，假设有 3 个基本/界限对，代码、堆和栈各一个。当进程正在运行时，每个段的基址寄存器都包含该段的线性页表的物理地址。因此，系统中的每个进程现在都有 3 个与其关联的页表。在上下文切换时，必须更改这些寄存器，以反映新运行进程的页表的位置。

在 TLB 未命中时（假设硬件管理的 TLB，即硬件负责处理 TLB 未命中），硬件使用分段位（SN）来确定要用哪个基址和界限对。然后硬件将其中的物理地址与 VPN 结合起来，形成页表项（PTE）的地址：

```
SN            = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN           = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

这段代码应该看起来很熟悉，它与我们之前在线性页表中看到的几乎完全相同。当然，唯一的区别是使用 3 个段基址寄存器中的一个，而不是单个页表基址寄存器。

杂合方案的关键区别在于，每个分段都有界限寄存器，每个界限寄存器保存了段中最大有效页的值。例如，如果代码段使用它的前 3 个页（0、1 和 2），则代码段页表将只有 3 个项分配给它，并且界限寄存器将被设置为 3。内存访问超出段的末尾将产生一个异常，并可能导致进程终止。以这种方式，与线性页表相比，杂合方法实现了显著的内存节省。栈和堆之间未分配的页不再占用页表中的空间（仅将其标记为无效）。

提示：使用杂合

当你有两个看似相反的好主意时，你应该总是看到你是否可以将它们组合成一个能够实现两全其美的杂合体（hybrid）。例如，杂交玉米物种已知比任何天然存在的物种更强壮。当然，并非所有的杂合都是好主意，请参阅 Zeedonk（或 Zonkey），它是斑马和驴的杂交。如果你不相信这样的生物存在，就查一下，你会大吃一惊。

但是，你可能会注意到，这种方法并非没有问题。首先，它仍然要求使用分段。正如我们讨论的那样，分段并不像我们需要的那样灵活，因为它假定地址空间有一定的使用模式。例如，如果有一个大而稀疏的堆，仍然可能导致大量的页表浪费。其次，这种杂合导致外部碎片再次出现。尽管大部分内存是以页面大小单位管理的，但页表现在可以是任意大小（是 PTE 的倍数）。因此，在内存中为它们寻找自由空间更为复杂。出于这些原因，人们继续寻找更好的方式来实现更小的页表。

20.3 多级页表

另一种方法并不依赖于分段，但也试图解决相同的问题：如何去掉页表中的所有无效区域，而不是将它们全部保留在内存中？我们将这种方法称为多级页表（multi-level page table），因为它将线性页表变成了类似树的东西。这种方法非常有效，许多现代系统都用它（例如 x86 [BOH10]）。我们现在详细描述这种方法。

多级页表的基本思想很简单。首先，将页表分成页大小的单元。然后，如果整页的页表项（PTE）无效，就完全不分配该页的页表。为了追踪页表的页是否有效（以及如果有效，它在内存中的位置），使用了名为页目录（page directory）的新结构。页目录因此可以告诉你页表的页在哪里，或者页表的整个页不包含有效页。

图 20.2 展示了一个例子。图的左边是经典的线性页表。即使地址空间的大部分中间区域无效，我们仍然需要为这些区域分配页表空间（即页表的中间两页）。右侧是一个多级页表。页目录仅将页表的两页标记为有效（第一个和最后一个）；因此，页表的这两页就驻留在内存中。因此，你可以形象地看到多级页表的工作方式：它只是让线性页表的一部分消失（释放这些帧用于其他用途），并用页目录来记录页表的哪些页被分配。

在一个简单的两级页表中，页目录为每页页表包含了一项。它由多个页目录项（Page Directory Entries，PDE）组成。PDE（至少）拥有有效位（valid bit）和页帧号（page frame number，PFN），类似于 PTE。但是，正如上面所暗示的，这个有效位的含义稍有不同：如果 PDE 项是有效的，则意味着该项指向的页表（通过 PFN）中至少有一页是有效的，即在该 PDE 所指向的页中，至少一个 PTE，其有效位被设置为 1。如果 PDE 项无效（即等于零），则 PDE 的其余部分没有定义。

与我们至今为止看到的方法相比，多级页表有一些明显的优势。首先，也许最明显的是，多级页表分配的页表空间，与你正在使用的地址空间内存量成比例。因此它通常很紧凑，并且支持稀疏的地址空间。

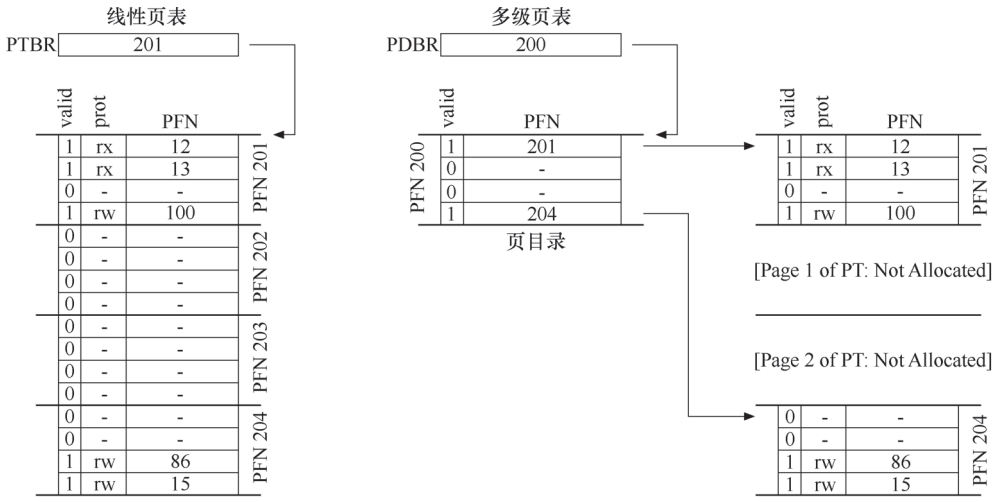


图 20.2 线性（左）和多级（右）页表

其次，如果仔细构建，页表的每个部分都可以整齐地放入一页中，从而更容易管理内存。操作系统可以在需要分配或增长页表时简单地获取下一个空闲页。将它与一个简单的（非分页）线性页表相比[①]，后者仅是按 VPN 索引的 PTE 数组。用这样的结构，整个线性页表必须连续驻留在物理内存中。对于一个大的页表（比如 4MB），找到如此大量的、未使用的连续空闲物理内存，可能是一个相当大的挑战。有了多级结构，我们增加了一个间接层（level of indirection），使用了页目录，它指向页表的各个部分。这种间接方式，让我们能够将页表页放在物理内存的任何地方。

提示：理解时空折中

在构建数据结构时，应始终考虑时间和空间的折中（time-space trade-off）。通常，如果你希望更快地访问特定的数据结构，就必须为该结构付出空间的代价。

应该指出，多级页表是有成本的。在 TLB 未命中时，需要从内存加载两次，才能从页表中获取正确的地址转换信息（一次用于页目录，另一次用于 PTE 本身），而用线性页表只需要一次加载。因此，多级表是一个时间—空间折中（time-space trade-off）的小例子。我们想要更小的表（并得到了），但不是没代价。尽管在常见情况下（TLB 命中），性能显然是相同的，但 TLB 未命中时，则会因较小的表而导致较高的成本。

另一个明显的缺点是复杂性。无论是硬件还是操作系统来处理页表查找（在 TLB 未命中时），这样做无疑都比简单的线性页表查找更复杂。通常我们愿意增加复杂性以提高性能或降低管理费用。在多级表的情况下，为了节省宝贵的内存，我们使页表查找更加复杂。

详细的多级示例

为了更好地理解多级页表背后的想法，我们来看一个例子。设想一个大小为 16KB 的小地址空间，其中包含 64 个字节的页。因此，我们有一个 14 位的虚拟地址空间，VPN 有 8

① 我们在这里做了一些假设，所有的页表全部驻留在物理内存中（即它们没有交换到磁盘）。 我们很快就会放松这个假设。

位，偏移量有 6 位。即使只有一小部分地址空间正在使用，线性页表也会有 2^8（256）个项。图 20.3 展示了这种地址空间的一个例子。

提示：对复杂性表示怀疑

系统设计者应该谨慎对待让系统增加复杂性。好的系统构建者所做的就是：实现最小复杂性的系统，来完成手上的任务。例如，如果磁盘空间非常大，则不应该设计一个尽可能少使用字节的文件系统。同样，如果处理器速度很快，建议在操作系统中编写一个干净、易于理解的模块，而不是 CPU 优化的、手写汇编的代码。注意过早优化的代码或其他形式的不必要的复杂性。这些方法会让系统难以理解、维护和调试。正如 Antoine de Saint-Exupery 的名言：“完美非无可增，乃不可减。”他没有写的是：“谈论完美易，真正实现难。

在这个例子中，虚拟页 0 和 1 用于代码，虚拟页 4 和 5 用于堆，虚拟页 254 和 255 用于栈。地址空间的其余页未被使用。

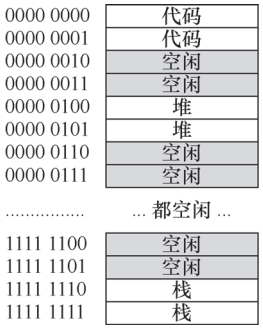
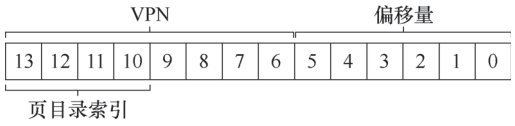


图 20.3 16KB 的地址空间和 64 字节的页

要为这个地址空间构建一个两级页表，我们从完整的线性页表开始，将它分解成页大小的单元。回想一下我们的完整页表（在这个例子中）有 256 个项；假设每个 PTE 的大小是 4 个字节。因此，我们的页大小为 1KB（256×4 字节）。鉴于我们有 64 字节的页，1KB 页表可以分为 16 个 64 字节的页，每个页可以容纳 16 个 PTE。

我们现在需要了解：如何获取 VPN，并用它来首先索引到页目录中，然后再索引到页表的页中。请记住，每个都是一组项。因此，我们需要弄清楚，如何为每个 VPN 构建索引。

我们首先索引到页目录。这个例子中的页表很小：256 个项，分布在 16 个页上。页目录需要为页表的每页提供一个项。因此，它有 16 个项。结果，我们需要 4 位 VPN 来索引目录。我们使用 VPN 的前 4 位，如下所示：



一旦从 VPN 中提取了页目录索引（简称 PDIndex），我们就可以通过简单的计算来找到页目录项（PDE）的地址：PDEAddr = PageDirBase +（PDIndex×sizeof（PDE））。这就得到了页目录，现在我们来看它，在地址转换上取得进一步进展。

如果页目录项标记为无效，则我们知道访问无效，从而引发异常。但是，如果 PDE 有效，我们还有更多工作要做。具体来说，我们现在必须从页目录项指向的页表的页中获取页表项（PTE）。要找到这个 PTE，我们必须使用 VPN 的剩余位索引到页表的部分：

这个页表索引（Page-Table Index，PTIndex）可以用来索引页表本身，给出 PTE 的地址：

```
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
```

请注意，从页目录项获得的页帧号（PFN）必须左移到位，然后再与页表索引组合，才能形成 PTE 的地址。

为了确定这一切是否合理，我们现在代入一个包含一些实际值的多级页表，并转换一个虚拟地址。让我们从这个例子的页目录开始（见表 20.2 的左侧）。

在该表中，可以看到每个页目录项（PDE）都描述了有关地址空间页表的一些内容。在这个例子中，地址空间里有两个有效区域（在开始和结束处），以及一些无效的映射。

在物理页 100（页表的第 0 页的物理帧号）中，我们有 1 页，包含 16 个页表项，记录了地址空间中的前 16 个 VPN。请参见表 20.2（中间部分）了解这部分页表的内容。

表 20.2 页目录和页表

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

页表的这一页包含前 16 个 VPN 的映射。在我们的例子中，VPN 0 和 1 是有效的（代码段），4 和 5（堆）也是。因此，该表有每个页的映射信息。其余项标记为无效。

页表的另一个有效页在 PFN 101 中。该页包含地址空间的最后 16 个 VPN 的映射。具体见表 20.2（右侧）。

在这个例子中，VPN 254 和 255（栈）包含有效的映射。希望从这个例子中可以看出，多级索引结构可以节省多少空间。在这个例子中，我们不是为一个线性页表分配完整的 16 页，而是分配 3 页：一个用于页目录，两个用于页表的具有有效映射的块。大型（32 位或 64 位）地址空间的节省显然要大得多。

最后，让我们用这些信息来进行地址转换。这里是一个地址，指向 VPN 254 的第 0 个

字节：0x3F80，或二进制的 11 1111 1000 0000。

回想一下，我们将使用 VPN 的前 4 位来索引页目录。因此，1111 会从上面的页目录中选择最后一个（第 15 个，如果你从第 0 个开始）。这就指向了位于地址 101 的页表的有效页。然后，我们使用 VPN 的下 4 位（1110）来索引页表的那一页并找到所需的 PTE。1110 是页面中的倒数第二（第 14 个）条，并告诉我们虚拟地址空间的页 254 映射到物理页 55。通过连接 PFN = 55（或十六进制 0x37）和 offset = 000000，可以形成我们想要的物理地址，并向内存系统发出请求：PhysAddr =（PTE.PFN << SHIFT）+ offset = 00 1101 1100 0000 = 0x0DC0。

你现在应该知道如何构建两级页表，利用指向页表页的页目录。但遗憾的是，我们的工作还没有完成。我们现在要讨论，有时两个页级别是不够的！

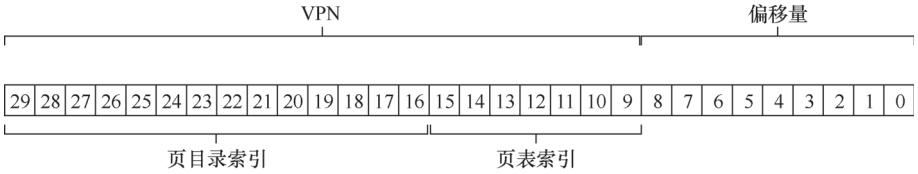
超过两级

在至今为止的例子中，我们假定多级页表只有两个级别：一个页目录和几页页表。在某些情况下，更深的树是可能的（并且确实需要）。

让我们举一个简单的例子，用它来说明为什么更深层次的多级页表可能有用。在这个例子中，假设我们有一个 30 位的虚拟地址空间和一个小的（512 字节）页。因此我们的虚拟地址有一个 21 位的虚拟页号和一个 9 位偏移量。

请记住我们构建多级页表的目标：使页表的每一部分都能放入一个页。到目前为止，我们只考虑了页表本身。但是，如果页目录太大，该怎么办？

要确定多级表中需要多少级别才能使页表的所有部分都能放入一页，首先要确定多少页表项可以放入一页。鉴于页大小为 512 字节，并且假设 PTE 大小为 4 字节，你应该看到，可以在单个页上放入 128 个 PTE。当我们索引页表时，我们可以得出结论，我们需要 VPN 的最低有效位 7 位（$\log_2 128$）作为索引：



在上面你还可能注意到，多少位留给了（大）页目录：14。如果我们的页目录有 2^{14} 个项，那么它不是一个页，而是 128 个，因此我们让多级页表的每一个部分放入一页目标失败了。

为了解决这个问题，我们为树再加一层，将页目录本身拆成多个页，然后在其上添加另一个页目录，指向页目录的页。我们可以按如下方式分割虚拟地址：

现在，当索引上层页目录时，我们使用虚拟地址的最高几位（图中的 PD 索引 0）。该索引用于从顶级页目录中获取页目录项。如果有效，则通过组合来自顶级 PDE 的物理帧号和 VPN 的下一部分（PD 索引 1）来查阅页目录的第二级。最后，如果有效，则可以通过使

用与第二级 PDE 的地址组合的页表索引来形成 PTE 地址。这会有很多工作。所有这些只是为了在多级页表中查找某些东西。

地址转换过程：记住 TLB

为了总结使用两级页表的地址转换的整个过程，我们再次以算法形式展示控制流（见图 20.4）。该图显示了每个内存引用在硬件中发生的情况（假设硬件管理的 TLB）。

从图中可以看到，在任何复杂的多级页表访问发生之前，硬件首先检查 TLB。在命中时，物理地址直接形成，而不像之前一样访问页表。只有在 TLB 未命中时，硬件才需要执行完整的多级查找。在这条路径上，可以看到传统的两级页表的成本：两次额外的内存访问来查找有效的转换映射。

```
1    VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2    (Success, TlbEntry) = TLB_Lookup(VPN)
3    if (Success == True)    // TLB Hit
4        if (CanAccess(TlbEntry.ProtectBits) == True)
5            Offset   = VirtualAddress & OFFSET_MASK
6            PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7            Register = AccessMemory(PhysAddr)
8        else
9            RaiseException(PROTECTION_FAULT)
10   else                  // TLB Miss
11       // first, get page directory entry
12       PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13       PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14       PDE     = AccessMemory(PDEAddr)
15       if (PDE.Valid == False)
16           RaiseException(SEGMENTATION_FAULT)
17       else
18           // PDE is valid: now fetch PTE from page table
19           PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20           PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21           PTE     = AccessMemory(PTEAddr)
22           if (PTE.Valid == False)
23               RaiseException(SEGMENTATION_FAULT)
24           else if (CanAccess(PTE.ProtectBits) == False)
25               RaiseException(PROTECTION_FAULT)
26           else
27               TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28               RetryInstruction()
```

图 20.4 多级页表控制流

20.4 反向页表

在反向页表（inverted page table）中，可以看到页表世界中更极端的空间节省。在这里，我们保留了一个页表，其中的项代表系统的每个物理页，而不是有许多页表（系统的每个进

程一个）。页表项告诉我们哪个进程正在使用此页，以及该进程的哪个虚拟页映射到此物理页。

现在，要找到正确的项，就是要搜索这个数据结构。线性扫描是昂贵的，因此通常在此基础结构上建立散列表，以加速查找。PowerPC 就是这种架构[JM98]的一个例子。

更一般地说，反向页表说明了我们从一开始就说过的内容：页表只是数据结构。你可以对数据结构做很多疯狂的事情，让它们更小或更大，使它们变得更慢或更快。多层和反向页表只是人们可以做的很多事情的两个例子。

20.5 将页表交换到磁盘

最后，我们讨论放松最后一个假设。到目前为止，我们一直假设页表位于内核拥有的物理内存中。即使我们有很多技巧来减小页表的大小，但是它仍然有可能是太大而无法一次装入内存。因此，一些系统将这样的页表放入内核虚拟内存（kernel virtual memory），从而允许系统在内存压力较大时，将这些页表中的一部分交换（swap）到磁盘。我们将在下一章（即 VAX/VMS 的案例研究）中进一步讨论这个问题，在我们更详细地了解了如何将页移入和移出内存之后。

20.6 小结

我们现在已经看到了如何构建真正的页表。不一定只是线性数组，而是更复杂的数据结构。这样的页表体现了时间和空间上的折中（表格越大，TLB 未命中可以处理得更快，反之亦然），因此结构的正确选择强烈依赖于给定环境的约束。

在一个内存受限的系统中（像很多旧系统一样），小结构是有意义的。在具有较多内存，并且工作负载主动使用大量内存页的系统中，用更大的页表来加速 TLB 未命中处理，可能是正确的选择。有了软件管理的 TLB，数据结构的整个世界开放给了喜悦的操作系统创新者（提示：就是你）。你能想出什么样的新结构？它们解决了什么问题？当你入睡时想想这些问题，做一个只有操作系统开发人员才能做的大梦。

参考资料

[BOH10]“Computer Systems: A Programmer’s Perspective”Randal E. Bryant and David R. O’Hallaron
Addison-Wesley, 2010
我们还没有找到很好的多级页表首选参考。然而，Bryant 和 O’Hallaron 编写的这本了不起的教科书深入探讨了 x86 的细节，至少这是一个使用这种结构的早期系统。这也是一本很棒的书。

[JM98]“Virtual Memory: Issues of Implementation”Bruce Jacob and Trevor Mudge
IEEE Computer, June 1998

对许多不同系统及其虚拟内存方法的优秀调查。其中有关于 x86、PowerPC、MIPS 和其他体系结构的大量细节内容。

[LL82]“Virtual Memory Management in the VAX/VMS Operating System”Hank Levy and P. Lipman
IEEE Computer, Vol. 15, No. 3, March 1982
一篇关于经典操作系统 VMS 中真实虚拟内存管理程序的精彩论文。它非常棒，实际上，从现在开始的几章，我们将利用它来复习目前为止我们学过的有关虚拟内存的所有内容。

[M28]“Reese’s Peanut Butter Cups”Mars Candy Corporation.
显然，这些精美的“甜点”是由 Harry Burnett Reese 在 1928 年发明的，他以前曾是奶牛场的农夫和 Milton S. Hershey 的运输工长。至少，维基百科上是这么说的。

[N+02]“Practical, Transparent Operating System Support for Superpages”Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox
OSDI ’02, Boston, Massachusetts, October 2002
一篇精彩的论文，展示了将大页或超大页并入现代操作系统中的所有细节。这篇文章阅读起来没有你想象的那么容易。

[M07]“Multics: History”
这个神奇的网站提供了 Multics 系统的大量历史记录，当然是 OS 历史上最有影响力的系统之一。引文如下：“麻省理工学院的 Jack Dennis 为 Multics 的开始提供了有影响力的架构理念，特别是将分页和分段相结合的想法。”

作业

这个有趣的小作业会测试你是否了解多级页表的工作原理。是的，前面句子中使用的“有趣”一词有一些争议。该程序叫作“可能不太怪：paging-multilevel-translate.py”。详情请参阅 README 文件。

问题

1．对于线性页表，你需要一个寄存器来定位页表，假设硬件在 TLB 未命中时进行查找。你需要多少个寄存器才能找到两级页表？三级页表呢？

2．使用模拟器对随机种子 0、1 和 2 执行翻译，并使用-c 标志检查你的答案。需要多少内存引用来执行每次查找？

3．根据你对缓存内存的工作原理的理解，你认为对页表的内存引用如何在缓存中工作？它们是否会导致大量的缓存命中（并导致快速访问）或者很多未命中（并导致访问缓慢）？

第 21 章　超越物理内存：机制

到目前为止，我们一直假定地址空间非常小，能放入物理内存。事实上，我们假设每个正在运行的进程的地址空间都能放入内存。我们将放松这些大的假设，并假设我们需要支持许多同时运行的巨大地址空间。

为了达到这个目的，需要在内存层级（memory hierarchy）上再加一层。到目前为止，我们一直假设所有页都常驻在物理内存中。但是，为了支持更大的地址空间，操作系统需要把当前没有在用的那部分地址空间找个地方存储起来。一般来说，这个地方有一个特点，那就是比内存有更大的容量。因此，一般来说也更慢（如果它足够快，我们就可以像使用内存一样使用，对吗？）。在现代系统中，硬盘（hard disk drive）通常能够满足这个需求。因此，在我们的存储层级结构中，大而慢的硬盘位于底层，内存之上。那么我们的关键问题是：

关键问题：如何超越物理内存

操作系统如何利用大而慢的设备，透明地提供巨大虚拟地址空间的假象？

你可能会问一个问题：为什么我们要为进程支持巨大的地址空间？答案还是方便和易用性。有了巨大的地址空间，你不必担心程序的数据结构是否有足够空间存储，只需自然地编写程序，根据需要分配内存。这是操作系统提供的一个强大的假象，使你的生活简单很多。别客气！一个反面例子是，一些早期系统使用“内存覆盖（memory overlays）”，它需要程序员根据需要手动移入或移出内存中的代码或数据[D97]。设想这样的场景：在调用函数或访问某些数据之前，你需要先安排将代码或数据移入内存。

补充：存储技术

稍后将深入介绍 I/O 设备如何运行。所以少安毋躁！当然，这个较慢的设备可以是硬盘，也可以是一些更新的设备，比如基于闪存的 SSD。我们也会讨论这些内容。但是现在，只要假设有一个大而较慢的设备，可以利用它来构建巨大虚拟内存的假象，甚至比物理内存本身更大。

不仅是一个进程，增加交换空间让操作系统为多个并发运行的进程都提供巨大地址空间的假象。多道程序（能够“同时”运行多个程序，更好地利用机器资源）的出现，强烈要求能够换出一些页，因为早期的机器显然不能将所有进程需要的所有页同时放在内存中。因此，多道程序和易用性都需要操作系统支持比物理内存更大的地址空间。这是所有现代虚拟内存系统都会做的事情，也是现在我们要进一步学习的内容。

21.1　交换空间

我们要做的第一件事情就是，在硬盘上开辟一部分空间用于物理页的移入和移出。在

操作系统中，一般这样的空间称为交换空间（swap space），因为我们将内存中的页交换到其中，并在需要的时候又交换回去。因此，我们会假设操作系统能够以页大小为单元读取或者写入交换空间。为了达到这个目的，操作系统需要记住给定页的硬盘地址（disk address）。

交换空间的大小是非常重要的，它决定了系统在某一时刻能够使用的最大内存页数。简单起见，现在假设它非常大。

在小例子中（见图 21.1），你可以看到一个 4 页的物理内存和一个 8 页的交换空间。在这个例子中，3 个进程（进程 0、进程 1 和进程 2）主动共享物理内存。但 3 个中的每一个，都只有一部分有效页在内存中，剩下的在硬盘的交换空间中。第 4 个进程（进程 3）的所有页都被交换到硬盘上，因此很清楚它目前没有运行。有一块交换空间是空闲的。即使通过这个小例子，你应该也能看出，使用交换空间如何让系统假装内存比实际物理内存更大。

我们需要注意，交换空间不是唯一的硬盘交换目的地。例如，假设运行一个二进制程序（如 ls，或者你自己编译的 main 程序）。这个二进制程序的代码页最开始是在硬盘上，但程序运行的时候，它们被加载到内存中（要么在程序开始运行时全部加载，要么在现代操作系统中，按需要一页一页加载）。但是，如果系统需要在物理内存中腾出空间以满足其他需求，则可以安全地重新使用这些代码页的内存空间，因为稍后它又可以重新从硬盘上的二进制文件加载。

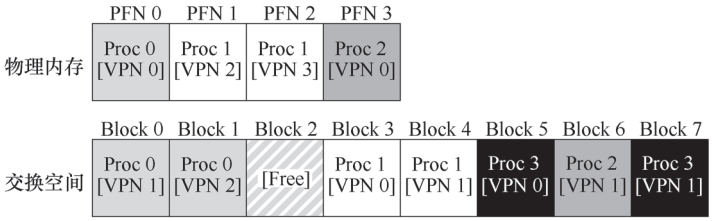


图 21.1 物理内存和交换空间

21.2 存在位

现在我们在硬盘上有一些空间，需要在系统中增加一些更高级的机制，来支持从硬盘交换页。简单起见，假设有一个硬件管理 TLB 的系统。

先回想一下内存引用发生了什么。正在运行的进程生成虚拟内存引用（用于获取指令或访问数据），在这种情况下，硬件将其转换为物理地址，再从内存中获取所需数据。

硬件首先从虚拟地址获得 VPN，检查 TLB 是否匹配（TLB 命中），如果命中，则获得最终的物理地址并从内存中取回。这希望是常见情形，因为它很快（不需要额外的内存访问）。

如果在 TLB 中找不到 VPN（即 TLB 未命中），则硬件在内存中查找页表（使用页表基址寄存器），并使用 VPN 查找该页的页表项（PTE）作为索引。如果页有效且存在于物理内存中，则硬件从 PTE 中获得 PFN，将其插入 TLB，并重试该指令，这次产生 TLB 命中。到现在为止还挺好。

但是，如果希望允许页交换到硬盘，必须添加更多的机制。具体来说，当硬件在 PTE 中查找时，可能发现页不在物理内存中。硬件（或操作系统，在软件管理 TLB 时）判断是

否在内存中的方法，是通过页表项中的一条新信息，即存在位（present bit）。如果存在位设置为 1，则表示该页存在于物理内存中，并且所有内容都如上所述进行。如果存在位设置为零，则页不在内存中，而在硬盘上。访问不在物理内存中的页，这种行为通常被称为页错误（page fault）。

补充：交换术语及其他

对于不同的机器和操作系统，虚拟内存系统的术语可能会有点令人困惑和不同。例如，页错误（page fault）一般是指对页表引用时产生某种错误：这可能包括在这里讨论的错误类型，即页不存在的错误，但有时指的是内存非法访问。事实上，我们将这种完全合法的访问（页被映射到进程的虚拟地址空间，但此时不在物理内存中）称为“错误”是很奇怪的。实际上，它应该被称为“页未命中（page miss）”。但是通常，当人们说一个程序“页错误”时，意味着它正在访问的虚拟地址空间的一部分，被操作系统交换到了硬盘上。

我们怀疑这种行为之所以被称为“错误”，是因为操作系统中的处理机制。当一些不寻常的事情发生的时候，即硬件不知道如何处理的时候，硬件只是简单地把控制权交给操作系统，希望操作系统能够解决。在这种情况下，进程想要访问的页不在内存中。硬件唯一能做的就是触发异常，操作系统从开始接管。由于这与进程执行非法操作处理流程一样，所以我们把这个活动称为“错误”，这也许并不奇怪。

在页错误时，操作系统被唤起来处理页错误。一段称为“页错误处理程序（page-fault handler）”的代码会执行，来处理页错误，接下来就会讲。

21.3 页错误

回想一下，在 TLB 未命中的情况下，我们有两种类型的系统：硬件管理的 TLB（硬件在页表中找到需要的转换映射）和软件管理的 TLB（操作系统执行查找过程）。不论在哪种系统中，如果页不存在，都由操作系统负责处理页错误。操作系统的页错误处理程序（page-fault handler）确定要做什么。几乎所有的系统都在软件中处理页错误。即使是硬件管理的 TLB，硬件也信任操作系统来管理这个重要的任务。

如果一个页不存在，它已被交换到硬盘，在处理页错误的时候，操作系统需要将该页交换到内存中。那么，问题来了：操作系统如何知道所需的页在哪儿？在许多系统中，页表是存储这些信息最自然的地方。因此，操作系统可以用 PTE 中的某些位来存储硬盘地址，这些位通常用来存储像页的 PFN 这样的数据。当操作系统接收到页错误时，它会在 PTE 中查找地址，并将请求发送到硬盘，将页读取到内存中。

补充：为什么硬件不能处理页错误

我们从 TLB 的经验中得知，硬件设计者不愿意信任操作系统做所有事情。那么为什么他们相信操作系统来处理页错误呢？有几个主要原因。首先，页错误导致的硬盘操作很慢。即使操作系统需要很长时间来处理故障，执行大量的指令，但相比于硬盘操作，这些额外开销是很小的。其次，为了能够处理页故障，硬件必须了解交换空间，如何向硬盘发起 I/O 操作，以及很多它当前所不知道的细节。因此，由于性能和简单的原因，操作系统来处理页错误，即使硬件人员也很开心。

当硬盘 I/O 完成时，操作系统会更新页表，将此页标记为存在，更新页表项（PTE）的 PFN 字段以记录新获取页的内存位置，并重试指令。下一次重新访问 TLB 还是未命中，然而这次因为页在内存中，因此会将页表中的地址更新到 TLB 中（也可以在处理页错误时更新 TLB 以避免此步骤）。最后的重试操作会在 TLB 中找到转换映射，从已转换的内存物理地址，获取所需的数据或指令。

请注意，当 I/O 在运行时，进程将处于阻塞（blocked）状态。因此，当页错误正常处理时，操作系统可以自由地运行其他可执行的进程。因为 I/O 操作是昂贵的，一个进程进行 I/O（页错误）时会执行另一个进程，这种交叠（overlap）是多道程序系统充分利用硬件的一种方式。

21.4 内存满了怎么办

在上面描述的过程中，你可能会注意到，我们假设有足够的空闲内存来从存储交换空间换入（page in）的页。当然，情况可能并非如此。内存可能已满（或接近满了）。因此，操作系统可能希望先交换出（page out）一个或多个页，以便为操作系统即将交换入的新页留出空间。选择哪些页被交换出或被替换（replace）的过程，被称为页交换策略（page-replacement policy）。

事实表明，人们在创建好页交换策略上投入了许多思考，因为换出不合适的页会导致程序性能上的巨大损失，也会导致程序以类似硬盘的速度运行而不是以类似内存的速度。在现有的技术条件下，这意味着程序可能会运行慢 10000～100000 倍。因此，这样的策略是我们应该详细研究的。实际上，这也正是我们下一章要做的。现在，我们只要知道有这样的策略存在，建立在之前描述的机制之上。

21.5 页错误处理流程

有了这些知识，我们现在就可以粗略地描绘内存访问的完整流程。换言之，如果有人问你：“当程序从内存中读取数据会发生什么？”，你应该对所有不同的可能性有了很好的概念。有关详细信息，请参见图 21.2 和图 21.3 中的控制流。图 21.2 展示了硬件在地址转换过程中所做的工作，图 21.3 展示了操作系统在页错误时所做的工作。

```
1    VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2    (Success, TlbEntry) = TLB_Lookup(VPN)
3    if (Success == True)    // TLB Hit
4        if (CanAccess(TlbEntry.ProtectBits) == True)
5            Offset     = VirtualAddress & OFFSET_MASK
6            PhysAddr   = (TlbEntry.PFN << SHIFT) | Offset
7            Register   = AccessMemory(PhysAddr)
8        else
9            RaiseException(PROTECTION_FAULT)
```

```
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE  =  AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else
16          if (CanAccess(PTE.ProtectBits) == False)
17              RaiseException(PROTECTION_FAULT)
18          else if (PTE.Present == True)
19              // assuming hardware-managed TLB
20              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21              RetryInstruction()
22          else if (PTE.Present == False)
23              RaiseException(PAGE_FAULT)
```

图 21.2 页错误控制流算法（硬件）

从图 21.2 的硬件控制流图中，可以注意到当 TLB 未命中发生的时候有 3 种重要情景。第一种情况，该页存在（present）且有效（valid）（第 18～21 行）。在这种情况下，TLB 未命中处理程序可以简单地从 PTE 中获取 PFN，然后重试指令（这次 TLB 会命中），并因此继续前面描述的流程。第二种情况（第 22～23 行），页错误处理程序需要运行。虽然这是进程可以访问的合法页（毕竟是有效的），但它并不在物理内存中。第三种情况，访问的是一个无效页，可能由于程序中的错误（第 13～14 行）。在这种情况下，PTE 中的其他位都不重要了。硬件捕获这个非法访问，操作系统陷阱处理程序运行，可能会杀死非法进程。

从图 21.3 的软件控制流中，可以看到为了处理页错误，操作系统大致做了什么。首先，操作系统必须为将要换入的页找到一个物理帧，如果没有这样的物理帧，我们将不得不等待交换算法运行，并从内存中踢出一些页，释放帧供这里使用。在获得物理帧后，处理程序发出 I/O 请求从交换空间读取页。最后，当这个慢操作完成时，操作系统更新页表并重试指令。重试将导致 TLB 未命中，然后再一次重试时，TLB 命中，此时硬件将能够访问所需的值。

```
1   PFN = FindFreePhysicalPage()
2   if (PFN == -1)              // no free page found
3       PFN = EvictPage()       // run replacement algorithm
4   DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5   PTE.present = True          // update page table with present
6   PTE.PFN     = PFN           // bit and translation (PFN)
7   RetryInstruction()          // retry instruction
```

图 21.3 页错误控制流算法（软件）

21.6 交换何时真正发生

到目前为止，我们一直描述的是操作系统会等到内存已经完全满了以后才会执行交换流程，然后才替换（踢出）一个页为其他页腾出空间。正如你想象的那样，这有点不切实

际的，因为操作系统可以更主动地预留一小部分空闲内存。

为了保证有少量的空闲内存，大多数操作系统会设置高水位线（High Watermark，HW）和低水位线（Low Watermark，LW），来帮助决定何时从内存中清除页。原理是这样：当操作系统发现有少于 LW 个页可用时，后台负责释放内存的线程会开始运行，直到有 HW 个可用的物理页。这个后台线程有时称为交换守护进程（swap daemon）或页守护进程（page daemon）[①]，它然后会很开心地进入休眠状态，因为它毕竟为操作系统释放了一些内存。

通过同时执行多个交换过程，我们可以进行一些性能优化。例如，许多系统会把多个要写入的页聚集（cluster）或分组（group），同时写入到交换区间，从而提高硬盘的效率[LL82]。我们稍后在讨论硬盘时将会看到，这种合并操作减少了硬盘的寻道和旋转开销，从而显著提高了性能。

为了配合后台的分页线程，图 21.3 中的控制流需要稍作修改。交换算法需要先简单检查是否有空闲页，而不是直接执行替换。如果没有空闲页，会通知后台分页线程按需要释放页。当线程释放一定数目的页时，它会重新唤醒原来的线程，然后就可以把需要的页交换进内存，继续它的工作。

提示：把一些工作放在后台

当你有一些工作要做的时候，把这些工作放在后台（background）运行是一个好注意，可以提高效率，并允许将这些操作合并执行。操作系统通常在后台执行很多工作。例如，在将数据写入硬盘之前，许多系统在内存中缓冲要写入的数据。这样做有很多好处：提高硬盘效率，因为硬盘现在可以一次写入多次要写入的数据，因此能够更好地调度这些写入。优化了写入延迟，因为数据写入到内存就可以返回。可能减少某些操作，因为写入操作可能不需要写入硬盘（例如，如果文件马上又被删除），也能更好地利用系统空闲时间（idle time），因为系统可以在空闲时完成后台工作，从而更好地利用硬件资源[G+95]。

21.7 小结

在这个简短的一章中，我们介绍了访问超出物理内存大小时的一些概念。要做到这一点，在页表结构中需要添加额外信息，比如增加一个存在位（present bit，或者其他类似机制），告诉我们页是不是在内存中。如果不存在，则操作系统页错误处理程序（page-fault handler）会运行以处理页错误（page fault），从而将需要的页从硬盘读取到内存，可能还需要先换出内存中的一些页，为即将换入的页腾出空间。

回想一下，很重要的是（并且令人惊讶的是），这些行为对进程都是透明的。对进程而言，它只是访问自己私有的、连续的虚拟内存。在后台，物理页被放置在物理内存中的任意（非连续）位置，有时它们甚至不在内存中，需要从硬盘取回。虽然我们希望在一般情况下内存访问速度很快，但在某些情况下，它需要多个硬盘操作的时间。像执行单条指令这样简单的事情，在最坏的情况下，可能需要很多毫秒才能完成。

① “守护进程（daemon）”这个词通常发音为“demon”，它是一个古老的术语，用于后台线程或过程，它可以做一些有用的事情。事实表明，该术语的来源是 Multics [CS94]。

参考资料

[CS94]“Take Our Word For It”
F. Corbato and R. Steinberg
Richard Steinberg 写道：“有人问我守护进程（daemon）这个词什么时候开始用于计算。根据我的研究，最好的结果是，这个词在 1963 年被你的团队在使用 IBM 7094 的 Project MAC 中首次使用。”Corbato 教授回答说：“我们使用守护进程这个词的灵感来源于物理学和热力学的麦克斯韦尔守护进程（Maxwell’s daemon，我的背景是物理学）。麦克斯韦尔守护进程是一个虚构的代理，帮助分拣不同速度的分子，并在后台不知疲倦地工作。我们别出心裁地开始使用守护进程来描述后台进程，这些进程不知疲倦地执行系统任务。”

[D97]“Before Memory Was Virtual”Peter Denning
From In the Beginning: Recollections of Software Pioneers, Wiley, November 1997
优秀的历史性作品，作者是虚拟内存和工作团队的先驱者之一。

[G+95]“Idleness is not sloth”
Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, John Wilkes USENIX ATC ’95, New Orleans, Louisiana
有趣且易于阅读的讨论，关于如何在系统中更好地利用空闲时间，有很多很好的例子。

[LL82]“Virtual Memory Management in the VAX/VMS Operating System”Hank Levy and P. Lipman
IEEE Computer, Vol. 15, No. 3, March 1982
这不是第一个使用这种聚集机制的地方，却是对这种机制如何工作的清晰而简单的解释。

第 22 章　超越物理内存：策略

在虚拟内存管理程序中，如果拥有大量空闲内存，操作就会变得很容易。页错误发生了，你在空闲页列表中找到空闲页，将它分配给不在内存中的页。嘿，操作系统，恭喜！ 你又成功了。

遗憾的是，当内存不够时事情会变得更有趣。在这种情况下，由于内存压力（memory pressure）迫使操作系统换出（paging out）一些页，为常用的页腾出空间。确定要踢出（evict）哪个页（或哪些页）封装在操作系统的替换策略（replacement policy）中。历史上，这是早期的虚拟内存系统要做的最重要的决定之一，因为旧系统的物理内存非常小。至少，有一些策略是非常值得了解的。因此我们的问题如下所示。

关键问题：如何决定踢出哪个页

操作系统如何决定从内存中踢出哪一页（或哪几页）？这个决定由系统的替换策略做出，替换策略通常会遵循一些通用的原则（下面将会讨论），但也会包括一些调整，以避免特殊情况下的行为。

22.1　缓存管理

在深入研究策略之前，先详细描述一下我们要解决的问题。由于内存只包含系统中所有页的子集，因此可以将其视为系统中虚拟内存页的缓存（cache）。因此，在为这个缓存选择替换策略时，我们的目标是让缓存未命中（cache miss）最少，即使得从磁盘获取页的次数最少。或者，可以将目标看成让缓存命中（cache hit）最多，即在内存中找到待访问页的次数最多。

知道了缓存命中和未命中的次数，就可以计算程序的平均内存访问时间（Average Memory Access Time，AMAT，计算机架构师衡量硬件缓存的指标 [HP06]）。具体来说，给定这些值，可以按照如下公式计算 *AMAT*：

$$AMAT = (P_{\text{Hit}} \cdot T_{\text{M}}) + (P_{\text{Miss}} \cdot T_{\text{D}})$$

其中 T_{M} 表示访问内存的成本，T_{D} 表示访问磁盘的成本，P_{Hit} 表示在缓存中找到数据的概率（命中），P_{Miss} 表示在缓存中找不到数据的概率（未命中）。P_{Hit} 和 P_{Miss} 从 0.0 变化到 1.0，并且 $P_{\text{Miss}} + P_{\text{Hit}} = 1.0$。

例如，假设有一个机器有小型地址空间：4KB，每页 256 字节。因此，虚拟地址由两部分组成：一个 4 位 VPN（最高有效位）和一个 8 位偏移量（最低有效位）。因此，本例中的一个进程可以访问总共 2^4=16 个虚拟页。在这个例子中，该进程将产生以下内存引用（即虚拟地址）0x000，0x100，0x200，0x300，0x400，0x500，0x600，0x700，0x800，0x900。

这些虚拟地址指向地址空间中前 10 页的每一页的第一个字节（页号是每个虚拟地址的第一个十六进制数字）。

让我们进一步假设，除了虚拟页 3 之外，所有页都已经在内存中。因此，我们的内存引用序列将遇到以下行为：命中，命中，命中，未命中，命中，命中，命中，命中，命中。我们可以计算命中率（hit rate，在内存中找到引用的百分比）：90%（P_{Hit} = 0.9），因为 10 个引用中有 9 个在内存中。未命中率（miss rate）显然是 10%（P_{Miss} = 0.1）。

要计算 AMAT，需要知道访问内存的成本和访问磁盘的成本。假设访问内存（TM）的成本约为 100ns，并且访问磁盘（TD）的成本大约为 10ms，则我们有以下 AMAT：0.9×100ns + 0.1×10ms，即 90ns + 1ms 或 1.0009ms，或约 1ms。如果我们的命中率是 99.9%（P_{Miss} = 0.001），结果是完全不同的：AMAT 是 10.1μs，大约快 100 倍。当命中率接近 100%时，AMAT 接近 100ns。

遗憾的是，正如你在这个例子中看到的，在现代系统中，磁盘访问的成本非常高，即使很小概率的未命中也会拉低正在运行的程序的总体 AMAT。显然，我们必须尽可能地避免缓存未命中，避免程序以磁盘的速度运行。要做到这一点，有一种方法就是仔细开发一个聪明的策略，像我们现在所做的一样。

22.2 最优替换策略

为了更好地理解一个特定的替换策略是如何工作的，将它与最好的替换策略进行比较是很好的方法。事实证明，这样一个最优（optimal）策略是 Belady 多年前开发的[B66]（原来这个策略叫作 MIN）。最优替换策略能达到总体未命中数量最少。Belady 展示了一个简单的方法（但遗憾的是，很难实现！），即替换内存中在最远将来才会被访问到的页，可以达到缓存未命中率最低。

提示：与最优策略对比非常有用

虽然最优策略非常不切实际，但作为仿真或其他研究的比较者还是非常有用的。比如，单说你喜欢的新算法有 80%的命中率是没有意义的，但加上最优算法只有 82%的命中率（因此你的新方法非常接近最优），就会使得结果很有意义，并给出了它的上下文。因此，在你进行的任何研究中，知道最优策略可以方便进行对比，知道你的策略有多大的改进空间，也用于决定当策略已经非常接近最优策略时，停止做无谓的优化[AD03]。

希望最优策略背后的想法你能理解。这样想：如果你不得不踢出一些页，为什么不踢出在最远将来才会访问的页呢？这样做基本上是说，缓存中所有其他页都比这个页重要。道理很简单：在引用最远将来会访问的页之前，你肯定会引用其他页。

我们追踪一个简单的例子，来理解最优策略的决定。假设一个程序按照以下顺序访问虚拟页：0，1，2，0，1，3，0，3，1，2，1。表 22.1 展示了最优的策略，这里假设缓存可以存 3 个页。

在表 22.1 中，可以看到以下操作。不要惊讶，前 3 个访问是未命中，因为缓存开始是

空的。这种未命中有时也称作冷启动未命中（cold-start miss，或强制未命中，compulsory miss）。然后我们再次引用页 0 和 1，它们都在缓存中。最后，我们又有一个缓存未命中（页 3），但这时缓存已满，必须进行替换！这引出了一个问题：我们应该替换哪个页？使用最优策略，我们检查当前缓存中每个页（0、1 和 2）未来访问情况，可以看到页 0 马上被访问，页 1 稍后被访问，页 2 在最远的将来被访问。因此，最优策略的选择很简单：踢出页面 2，结果是缓存中的页面是 0、1 和 3。接下来的 3 个引用是命中的，然后又访问到被我们之前踢出的页 2，那么又有一个未命中。这里，最优策略再次检查缓存页（0、1 和 3）中每个页面的未来被访问情况，并且看到只要不踢出页 1（即将被访问）就可以。这个例子显示了页 3 被踢出，虽然踢出 0 也是可以的。最后，我们命中页 1，追踪完成。

表 22.1 追踪最优策略

访问	命中/未命中	踢出	导致缓存状态
0	未命中		0
1	未命中		0、1
2	未命中		0、1、2
0	命中		0、1、2
1	命中		0、1、2
3	未命中	2	0、1、3
0	命中		0、1、3
3	命中		0、1、3
1	命中		0、1、3
2	未命中	3	0、1、2
1	命中		0、1、2

补充：缓存未命中的类型

在计算机体系结构世界中，架构师有时会将未命中分为 3 类：强制性、容量和冲突未命中，有时称为 3C [H87]。发生强制性（compulsory miss）未命中（或冷启动未命中，cold-start miss [EF78]）是因为缓存开始是空的，而这是对项目的第一次引用。与此不同，由于缓存的空间不足而不得不踢出一个项目以将新项目引入缓存，就发生了容量未命中（capacity miss）。第三种类型的未命中（冲突未命中，conflict miss）出现在硬件中，因为硬件缓存中对项的放置位置有限制，这是由于所谓的集合关联性（set-associativity）。它不会出现在操作系统页面缓存中，因为这样的缓存总是完全关联的（fully-associative），即对页面可以放置的内存位置没有限制。详情请见 H&P [HP06]。

我们同时计算缓存命中率：有 6 次命中和 5 次未命中，那么缓存命中率 $\frac{Hits}{Hits + Misses}$ 是 $\frac{6}{6+5}$，或 54.5%。也可以计算命中率中除去强制未命中（即忽略页的第一次未命中），那么命中率为 81.8%。

遗憾的是，正如我们之前在开发调度策略时所看到的那样，未来的访问是无法知道的，

你无法为通用操作系统实现最优策略[①]。因此，在开发一个真正的、可实现的策略时，我们将聚焦于寻找其他决定把哪个页面踢出的方法。因此，最优策略只能作为比较，知道我们的策略有多接近“完美”。

22.3 简单策略：FIFO

许多早期的系统避免了尝试达到最优的复杂性，采用了非常简单的替换策略。例如，一些系统使用 FIFO（先入先出）替换策略。页在进入系统时，简单地放入一个队列。当发生替换时，队列尾部的页（“先入”页）被踢出。FIFO 有一个很大的优势：实现相当简单。

让我们来看看 FIFO 策略如何执行这过程（见表 22.2）。我们再次开始追踪 3 个页面 0、1 和 2。首先是强制性未命中，然后命中页 0 和 1。接下来，引用页 3，缓存未命中。使用 FIFO 策略决定替换哪个页面是很容易的：选择第一个进入的页，这里是页 0（表中的缓存状态列是按照先进先出顺序，最左侧是第一个进来的页），遗憾的是，我们的下一个访问还是页 0，导致另一次未命中和替换（替换页 1）。然后我们命中页 3，但是未命中页 1 和 2，最后命中页 3。

表 22.2 追踪 FIFO 策略

访问	命中/未命中	踢出	导致缓存状态	
0	未命中		先入→	0
1	未命中		先入→	0、1
2	未命中		先入→	0、1、2
0	命中		先入→	0、1、2
1	命中		先入→	0、1、2
3	未命中	0	先入→	1、2、3
0	未命中	1	先入→	2、3、0
3	命中		先入→	2、3、0
1	未命中	2	先入→	3、0、1
2	未命中	3	先入→	0、1、2
1	命中		先入→	0、1、2

对比 FIFO 和最优策略，FIFO 明显不如最优策略，FIFO 命中率只有 36.4%（不包括强制性未命中为 57.1%）。先进先出（FIFO）根本无法确定页的重要性：即使页 0 已被多次访问，FIFO 仍然会将其踢出，因为它是第一个进入内存的。

补充：Belady 的异常

Belady（最优策略发明者）及其同事发现了一个有意思的引用序列[BNS69]。内存引用顺序是：1，2，3，4，1，2，5，1，2，3，4，5。他们正在研究的替换策略是 FIFO。有趣的问题：当缓存大小从 3 变成 4 时，缓存命中率如何变化？

① 如果你可以，请告诉我们，我们可以一起发财，或者，像 “发现” 冷聚变的科学家一样，被众人所讽刺和嘲笑[FP89]。

一般来说，当缓存变大时，缓存命中率是会提高的（变好）。但在这个例子，采用 FIFO，命中率反而下降了！你可以自己计算一下缓存命中和未命中次数。这种奇怪的现象被称为 Belady 的异常（Belady's Anomaly）。

其他一些策略，比如 LRU，不会遇到这个问题。可以猜猜为什么？事实证明，LRU 具有所谓的栈特性（stack property）[M+70]。对于具有这个性质的算法，大小为 $N+1$ 的缓存自然包括大小为 N 的缓存的内容。因此，当增加缓存大小时，缓存命中率至少保证不变，有可能提高。先进先出（FIFO）和随机（Random）等显然没有栈特性，因此容易出现异常行为。

22.4 另一简单策略：随机

另一个类似的替换策略是随机，在内存满的时候它随机选择一个页进行替换。随机具有类似于 FIFO 的属性。实现起来很简单，但是它在挑选替换哪个页时不够智能。让我们来看看随机策略在我们著名的例子上的引用流程（见表 22.3）。

表 22.3 追踪随机策略

访问	命中/未命中	踢出	导致缓存状态
0	未命中		0
1	未命中		0、1
2	未命中		0、1、2
0	命中		0、1、2
1	命中		0、1、2
3	未命中	0	1、2、3
0	未命中	1	2、3、0
3	命中		2、3、0
1	未命中	3	2、0、1
2	命中		2、0、1
1	命中		2、0、1

当然，随机的表现完全取决于多幸运（或不幸）。在上面的例子中，随机比 FIFO 好一点，比最优的差一点。事实上，我们可以运行数千次的随机实验，求得一个平均的结果。图 22.1 显示了 10000 次试验后随机策略的平均命中率，每次试验都有不同的随机种子。正如你所看到的，有些时候（仅仅 40%的概率），随机和最优策略一样好，在上述例子中，命中内存

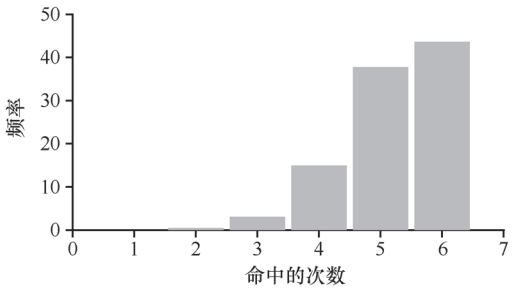


图 22.1 随机策略在 10000 次尝试下的表现

的次数是 6 次。有时候情况会更糟糕，只有 2 次或更少。随机策略取决于当时的运气。

22.5 利用历史数据：LRU

遗憾的是，任何像 FIFO 或随机这样简单的策略都可能会有一个共同的问题：它可能会踢出一个重要的页，而这个页马上要被引用。先进先出（FIFO）将先进入的页踢出。如果这恰好是一个包含重要代码或数据结构的页，它还是会被踢出，尽管它很快会被重新载入。因此，FIFO、Random 和类似的策略不太可能达到最优，需要更智能的策略。

正如在调度策略所做的那样，为了提高后续的命中率，我们再次通过历史的访问情况作为参考。例如，如果某个程序在过去访问过某个页，则很有可能在不久的将来会再次访问该页。

页替换策略可以使用的一个历史信息是频率（frequency）。如果一个页被访问了很多次，也许它不应该被替换，因为它显然更有价值。页更常用的属性是访问的近期性（recency），越近被访问过的页，也许再次访问的可能性也就越大。

这一系列的策略是基于人们所说的局部性原则（principle of locality）[D70]，基本上只是对程序及其行为的观察。这个原理简单地说就是程序倾向于频繁地访问某些代码（例如循环）和数据结构（例如循环访问的数组）。因此，我们应该尝试用历史数据来确定哪些页面更重要，并在需要踢出页时将这些页保存在内存中。

因此，一系列简单的基于历史的算法诞生了。“最不经常使用”（Least-Frequently-Used，LFU）策略会替换最不经常使用的页。同样，“最少最近使用”（Least-Recently-Used，LRU）策略替换最近最少使用的页面。这些算法很容易记住：一旦知道这个名字，就能确切知道它是什么，这种名字就非常好。

补充：局部性类型

程序倾向于表现出两种类型的局部。第一种是空间局部性（spatial locality），它指出如果页 P 被访问，可能围绕它的页（比如 P–1 或 P + 1）也会被访问。第二种是时间局部性（temporal locality），它指出近期访问过的页面很可能在不久的将来再次访问。假设存在这些类型的局部性，对硬件系统的缓存层次结构起着重要作用，硬件系统部署了许多级别的指令、数据和地址转换缓存，以便在存在此类局部性时，能帮助程序快速运行。

当然，通常所说的局部性原则（principle of locality）并不是硬性规定，所有的程序都必须遵守。事实上，一些程序以相当随机的方式访问内存（或磁盘），并且在其访问序列中不显示太多或完全没有局部性。因此，尽管在设计任何类型的缓存（硬件或软件）时，局部性都是一件好事，但它并不能保证成功。相反，它是一种经常证明在计算机系统设计中有用的启发式方法。

为了更好地理解 LRU，我们来看看 LRU 如何在示例引用序列上执行。表 22.4 展示了结果。从表中，可以看到 LRU 如何利用历史记录，比无状态策略（如随机或 FIFO）做得更好。在这个例子中，当第一次需要替换页时，LRU 会踢出页 2，因为 0 和 1 的访问时间更近。然后它替换页 0，因为 1 和 3 最近被访问过。在这两种情况下，基于历史的 LRU 的决

定证明是更准确的，并且下一个引用也是命中。因此，在我们的简单例子中，LRU 的表现几乎快要赶上最优策略了[①]。

表 22.4 追踪 LUR 策略

访问	命中/未命中	踢出	导致缓存状态	
0	未命中		LUR→	0
1	未命中		LUR→	0、1
2	未命中		LUR→	0、1、2
0	命中		LUR→	1、2、0
1	命中		LUR→	2、0、1
3	未命中	2	LUR→	0、1、3
0	命中		LUR→	1、3、0
3	命中		LUR→	1、0、3
1	命中		LUR→	0、3、1
2	未命中	0	LUR→	3、1、2
1	命中		LUR→	3、2、1

我们也应该注意到，与这些算法完全相反的算法也是存在：最经常使用策略（Most-Frequently-Used，MFU）和最近使用策略（Most-Recently-Used，MRU）。在大多数情况下（不是全部!），这些策略效果都不好，因为它们忽视了大多数程序都具有的局部性特点。

22.6 工作负载示例

让我们再看几个例子，以便更好地理解这些策略。在这里，我们将查看更复杂的工作负载（workload），而不是追踪小例子。但是，这些工作负载也被大大简化了。更好的研究应该包含应用程序追踪。

第一个工作负载没有局部性，这意味着每个引用都是访问一个随机页。在这个简单的例子中，工作负载每次访问独立的 100 个页，随机选择下一个要引用的页。总体来说，访问了 10000 个页。在实验中，我们将缓存大小从非常小（1 页）变化到足以容纳所有页（100 页），以便了解每个策略在缓存大小范围内的表现。

图 22.2 展示了最优、LRU、随机和 FIFO 策略的实验结果。图 22.2 中的 y 轴显示了每个策略的命中率。如上所述，x 轴表示缓存大小的变化。

我们可以从图 22.2 中得出一些结论。首先，当工作负载不存在局部性时，使用的策略区别不大。LRU、FIFO 和随机都执行相同的操作，命中率完全由缓存的大小决定。其次，当缓存足够大到可以容纳所有的数据时，使用哪种策略也无关紧要，所有的策略（甚至是随机的）都有 100%的命中率。最后，你可以看到，最优策略的表现明显好于实际的策略。如果有可能的话，偷窥未来，就能做到更好的替换。

① 好吧，我们夸大了结果。但有时候为了证明一个观点，夸大是有必要的。

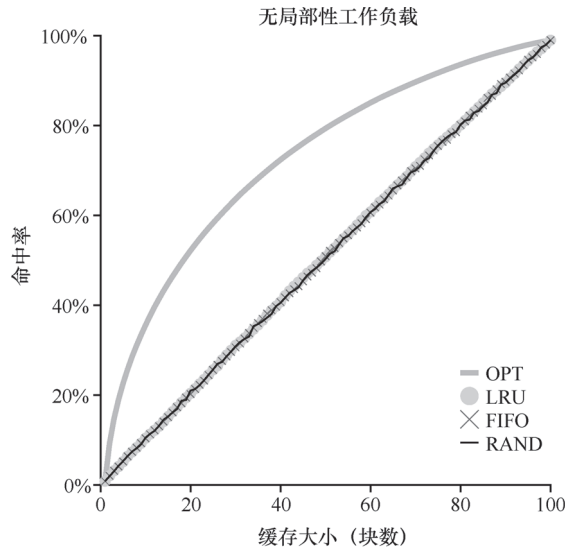


图 22.2 无局部性工作负载

我们下一个工作负载就是所谓的“80—20”负载场景，它表现出局部性：80%的引用是访问 20%的页（“热门”页）。剩下的 20%是对剩余的 80%的页（“冷门”页）访问。在我们的负载场景，总共有 100 个不同的页。因此，“热门”页是大部分时间访问的页，其余时间访问的是“冷门”页。图 22.3 展示了不同策略在这个工作负载下的表现。

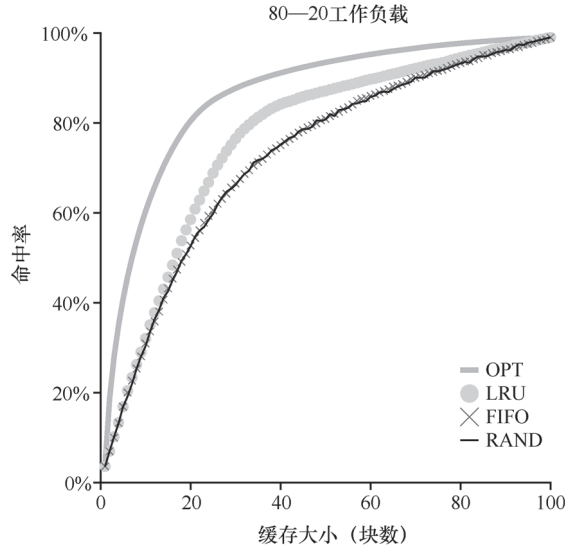


图 22.3 80—20 工作负载

从图 22.3 中可以看出，尽管随机和 FIFO 都很好地运行，但 LRU 更好，因为它更可能保持热门页。由于这些页面过去经常被提及，它们很可能在不久的将来再次被提及。优化再次表现得更好，表明 LRU 的历史信息并不完美。

你现在可能会想：LRU 对随机和 FIFO 的命中率提高真的非常重要么？如往常一样，答案是“视情况而定”。如果每次未命中代价非常大（并不罕见），那么即使小幅提高命中率

（降低未命中率）也会对性能产生巨大的影响。如果未命中的代价不那么大，那么 LRU 带来的好处就不会那么重要。

让我们看看最后一个工作负载。我们称之为“循环顺序”工作负载，其中依次引用 50 个页，从 0 开始，然后是 1，…，49，然后循环，重复访问，总共有 10000 次访问 50 个单独页。图 22.4 展示了这个工作负载下各个策略的行为。

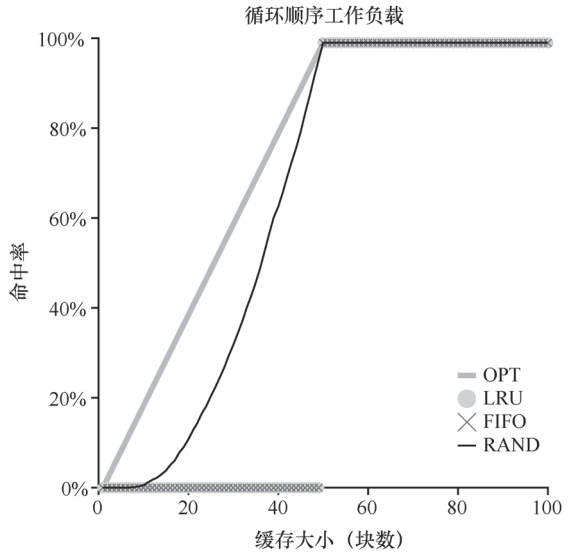


图 22.4 循环工作负载

这种工作负载在许多应用程序（包括重要的商业应用，如数据库[CD85]）中非常常见，展示了 LRU 或者 FIFO 的最差情况。这些算法，在循环顺序的工作负载下，踢出较旧的页。遗憾的是，由于工作负载的循环性质，这些较旧的页将比因为策略决定保存在缓存中的页更早被访问。事实上，即使缓存的大小是 49 页，50 个页面的循环连续工作负载也会导致 0%的命中率。有趣的是，随机策略明显更好，虽然距离最优策略还有距离，但至少达到了非零的命中率。可以看出随机策略有一些不错的属性，比如不会出现特殊情况下奇怪的结果。

22.7 实现基于历史信息的算法

正如你所看到的，像 LRU 这样的算法通常优于简单的策略（如 FIFO 或随机），它们可能会踢出重要的页。遗憾的是，基于历史信息的策略带来了一个新的挑战：应该如何实现呢？

以 LRU 为例。为了实现它，我们需要做很多工作。具体地说，在每次页访问（即每次内存访问，不管是取指令还是加载指令还是存储指令）时，我们都必须更新一些数据，从而将该页移动到列表的前面（即 MRU 侧）。与 FIFO 相比，FIFO 的页列表仅在页被踢出（通过移除最先进入的页）或者当新页添加到列表（已到列表尾部）时才被访问。为了记录哪些页是最少和最近被使用，系统必须对每次内存引用做一些记录工作。显然，如果不十分小心，这样的记录反而会极大地影响性能。

有一种方法有助于加快速度，就是增加一点硬件支持。例如，硬件可以在每个页访问时更新内存中的时间字段（时间字段可以在每个进程的页表中，或者在内存的某个单独的数组中，每个物理页有一个）。因此，当页被访问时，时间字段将被硬件设置为当前时间。然后，在需要替换页时，操作系统可以简单地扫描系统中所有页的时间字段以找到最近最少使用的页。

遗憾的是，随着系统中页数量的增长，扫描所有页的时间字段只是为了找到最精确最少使用的页，这个代价太昂贵。想象一下一台拥有 4GB 内存的机器，内存切成 4KB 的页。这台机器有一百万页，即使以现代 CPU 速度找到 LRU 页也将需要很长时间。这就引出了一个问题：我们是否真的需要找到绝对最旧的页来替换？找到差不多最旧的页可以吗?

关键问题：如何实现 LRU 替换策略

由于实现完美的 LRU 代价非常昂贵，我们能否实现一个近似的 LRU 算法，并且依然能够获得预期的效果?

22.8 近似 LRU

事实证明，答案是肯定的：从计算开销的角度来看，近似 LRU 更为可行，实际上这也是许多现代系统的做法。这个想法需要硬件增加一个使用位（use bit，有时称为引用位，reference bit），这种做法在第一个支持分页的系统 Atlas one-level store[KE + 62]中实现。系统的每个页有一个使用位，然后这些使用位存储在某个地方（例如，它们可能在每个进程的页表中，或者只在某个数组中）。每当页被引用（即读或写）时，硬件将使用位设置为 1。但是，硬件不会清除该位（即将其设置为 0），这由操作系统负责。

操作系统如何利用使用位来实现近似 LRU？可以有很多方法，有一个简单的方法称作时钟算法（clock algorithm）[C69]。想象一下，系统中的所有页都放在一个循环列表中。时钟指针（clock hand）开始时指向某个特定的页（哪个页不重要）。当必须进行页替换时，操作系统检查当前指向的页 P 的使用位是 1 还是 0。如果是 1，则意味着页面 P 最近被使用，因此不适合被替换。然后，P 的使用位设置为 0，时钟指针递增到下一页（$P + 1$）。该算法一直持续到找到一个使用位为 0 的页，使用位为 0 意味着这个页最近没有被使用过（在最坏的情况下，所有的页都已经被使用了，那么就将所有页的使用位都设置为 0）。

请注意，这种方法不是通过使用位来实现近似 LRU 的唯一方法。实际上，任何周期性地清除使用位，然后通过区分使用位是 1 和 0 来判定该替换哪个页的方法都是可以的。Corbato 的时钟算法只是一个早期成熟的算法，并且具有不重复扫描内存来寻找未使用页的特点，也就是它在最差情况下，只会遍历一次所有内存。

图 22.5 展示了时钟算法的一个变种的行为。该变种在需要进行页替换时随机扫描各页，如果遇到一个页的引用位为 1，就清除该位（即将它设置为 0）。直到找到一个使用位为 0 的页，将这个页进行替换。如你所见，虽然时钟算法不如完美的 LRU 做得好，但它比不考虑历史访问的方法要好。

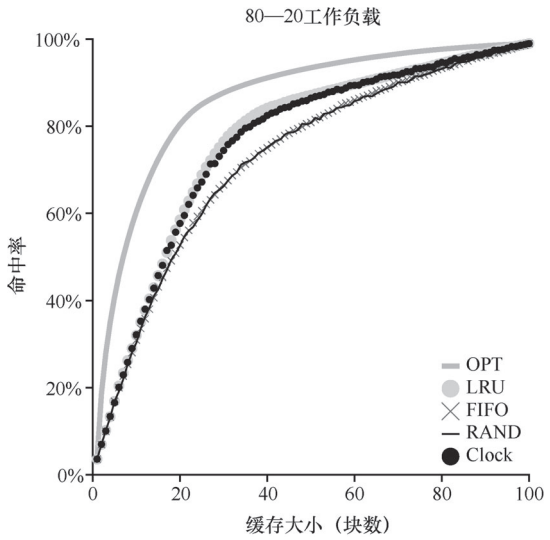


图 22.5 80—20 工作负载和时钟算法

22.9 考虑脏页

时钟算法的一个小修改（最初也由 Corbato [C69]提出），是对内存中的页是否被修改的额外考虑。这样做的原因是：如果页已被修改（modified）并因此变脏（dirty），则踢出它就必须将它写回磁盘，这很昂贵。如果它没有被修改（因此是干净的，clean），踢出就没成本。物理帧可以简单地重用于其他目的而无须额外的 I/O。因此，一些虚拟机系统更倾向于踢出干净页，而不是脏页。

为了支持这种行为，硬件应该包括一个修改位（modified bit，又名脏位，dirty bit）。每次写入页时都会设置此位，因此可以将其合并到页面替换算法中。例如，时钟算法可以被改变，以扫描既未使用又干净的页先踢出。无法找到这种页时，再查找脏的未使用页面，等等。

22.10 其他虚拟内存策略

页面替换不是虚拟内存子系统采用的唯一策略（尽管它可能是最重要的）。例如，操作系统还必须决定何时将页载入内存。该策略有时称为页选择（page selection）策略（因为 Denning 这样命名[D70]），它向操作系统提供了一些不同的选项。

对于大多数页而言，操作系统只是使用按需分页（demand paging），这意味着操作系统在页被访问时将页载入内存中，“按需”即可。当然，操作系统可能会猜测一个页面即将被使用，从而提前载入。这种行为被称为预取（prefetching），只有在有合理的成功机会时才应该这样做。例如，一些系统将假设如果代码页 P 被载入内存，那么代码页 $P+1$ 很可能很快被访问，因此也应该被载入内存。

另一个策略决定了操作系统如何将页面写入磁盘。当然，它们可以简单地一次写出一

个。然而，许多系统会在内存中收集一些待完成写入，并以一种（更高效）的写入方式将它们写入硬盘。这种行为通常称为聚集（clustering）写入，或者就是分组写入（grouping），这样做有效是因为硬盘驱动器的性质，执行单次大的写操作，比许多小的写操作更有效。

22.11 抖动

在结束之前，我们解决了最后一个问题：当内存就是被超额请求时，操作系统应该做什么，这组正在运行的进程的内存需求是否超出了可用物理内存？在这种情况下，系统将不断地进行换页，这种情况有时被称为抖动（thrashing）[D70]。

一些早期的操作系统有一组相当复杂的机制，以便在抖动发生时检测并应对。例如，给定一组进程，系统可以决定不运行部分进程，希望减少的进程工作集（它们活跃使用的页面）能放入内存，从而能够取得进展。这种方法通常被称为准入控制（admission control），它表明，少做工作有时比尝试一下子做好所有事情更好，这是我们在现实生活中以及在现代计算机系统中经常遇到的情况（令人遗憾）。

目前的一些系统采用更严格的方法处理内存过载。例如，当内存超额请求时，某些版本的 Linux 会运行“内存不足的杀手程序（out-of-memory killer）”。这个守护进程选择一个内存密集型进程并杀死它，从而以不怎么委婉的方式减少内存。虽然成功地减轻了内存压力，但这种方法可能会遇到问题，例如，如果它杀死 *X* 服务器，就会导致所有需要显示的应用程序不可用。

22.12 小结

我们已经看到了许多页替换（和其他）策略的介绍，这些策略是所有现代操作系统中虚拟内存子系统的一部分。现代系统增加了对时钟等简单 LRU 近似值的一些调整。例如，扫描抗性（scan resistance）是许多现代算法的重要组成部分，如 ARC [MM03]。扫描抗性算法通常是类似 LRU 的，但也试图避免 LRU 的最坏情况行为，我们曾在循环顺序工作负载中看到这种情况。因此，页替换算法的发展仍在继续。

然而，在许多情况下，由于内存访问和磁盘访问时间之间的差异增加，这些算法的重要性降低了。由于分页到硬盘非常昂贵，因此频繁分页的成本太高。所以，过度分页的最佳解决方案往往很简单：购买更多的内存。

参考资料

[AD03]“Run-Time Adaptation in River”Remzi H. Arpaci-Dusseau
ACM TOCS, 21:1, February 2003
本书作者之一关于 River 系统的研究工作的总结。当然，在其中，他发现与理想情况做比较是系统设计人员的一项重要技术。

[B66]“A Study of Replacement Algorithms for Virtual-Storage Computer”Laszlo A. Belady
IBM Systems Journal 5(2): 78-101, 1966
这篇文章介绍了计算策略最优行为的简单方法（MIN 算法）。

[BNS69]“An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine”
L. A. Belady and R. A. Nelson and G. S. Shedler
Communications of the ACM, 12:6, June 1969
介绍称为“Belady 的异常”的内存引用的小序列的文章。我们想知道，Nelson 和 Shedler 如何看待这个名字呢？

[CD85]“An Evaluation of Buffer Management Strategies for Relational Database Systems”Hong-Tai Chou and David J. DeWitt
VLDB ’85, Stockholm, Sweden, August 1985
关于不同缓冲策略的著名数据库文章，你应该使用多种常见数据库访问模式。如果你知道有关工作负载的某些信息，那么就可以制订策略，让它比操作系统中常见的通用目标策略更好。

[C69]“A Paging Experiment with the Multics System”
F.J. Corbato
Included in a Festschrift published in honor of Prof. P.M. Morse MIT Press, Cambridge, MA, 1969
对时钟算法的最初引用（很难找到!），但不是第一次使用位。感谢麻省理工学院的 H. Balakrishnan 为我们找出这篇论文。

[D70]“Virtual Memory”Peter J. Denning
Computing Surveys, Vol. 2, No. 3, September 1970
Denning 对虚拟存储系统的早期著名调查。

[EF78]“Cold-start vs. Warm-start Miss Ratios”Malcolm C. Easton and Ronald Fagin Communications of the ACM, 21:10, October 1978
关于冷启动与热启动未命中的很好的讨论。

[FP89]“Electrochemically Induced Nuclear Fusion of Deuterium”Martin Fleischmann and Stanley Pons
Journal of Electroanalytical Chemistry, Volume 26, Number 2, Part 1, April, 1989
这篇著名的论文有可能为世界带来革命性的变化，它提供了一种简单的方法，可以从水罐中产生几乎无限的电力，而电力罐中只含有少许金属。但 Pons 和 Fleischmann 发表的（并广为宣传的）实验结果无法重现，因此这两位有梦想的科学家都丧失了名誉（当然也受到了嘲笑）。唯一真正为这个结果感到高兴的人是 Marvin Hawkins，尽管他参与了这项工作，但他的名字却从本文中被删除了。他因而避免将他的名字与 20 世纪最大的科学失误之一联系起来。

[HP06]“Computer Architecture: A Quantitative Approach”John Hennessy and David Patterson
Morgan-Kaufmann, 2006
一本关于计算机体系结构的了不起而奇妙的书，必读！

[H87]“Aspects of Cache Memory and Instruction Buffer Performance”Mark D. Hill

Ph.D. Dissertation, U.C. Berkeley, 1987
Mark Hill 在其论文工作中介绍了 3C，后来因其被包含在 H&P [HP06]中而广泛流行。其中的引述：“我发现根据未命中的原因直观地将未命中划分为 3 个部分是有用的（第 49 页）。”

[KE+62]“One-level Storage System”
T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner IRE Trans. EC-11:2, 1962
虽然 Atlas 有一个使用位，但它只有很少量的页，因此在大型存储器中使用位的扫描并不是作者解决的问题。

[M+70]“Evaluation Techniques for Storage Hierarchies”
R．L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger IBM Systems Journal, Volume 9:2, 1970
一篇主要关于如何高效地模拟缓存层次结构的论文。本文无疑是这方面的经典之作，还有对各种替代算法的一些特性的极佳讨论。你能弄清楚为什么栈属性可能对同时模拟很多不同大小的缓存有用吗？

[MM03]“ARC: A Self-Tuning, Low Overhead Replacement Cache”Nimrod Megiddo and Dharmendra S. Modha FAST 2003, February 2003, San Jose, California
关于替换算法的优秀现代论文，其中包括现在某些系统中使用的新策略 ARC。在 2014 年 FAST ’14 大会上，获得了存储系统社区的“时间考验”奖。

作业

这个模拟器 paging-policy.py 允许你使用不同的页替换策略。详情请参阅 README 文件。

问题

1．使用以下参数生成随机地址：-s 0 -n 10，-s 1 -n 10 和-s 2 -n 10。将策略从 FIFO 更改为 LRU，并将其更改为 OPT。计算所述地址追踪中的每个访问是否命中或未命中。

2．对于大小为 5 的高速缓存，为以下每个策略生成最差情况的地址引用序列：FIFO、LRU 和 MRU（最差情况下的引用序列导致尽可能多的未命中）。对于最差情况下的引用序列，需要的缓存增大多少，才能大幅提高性能，并接近 OPT？

3．生成一个随机追踪序列（使用 Python 或 Perl）。你预计不同的策略在这样的追踪序列上的表现如何？

4．现在生成一些局部性追踪序列。如何能够产生这样的追踪序列？LRU 表现如何？RAND 比 LRU 好多少？CLOCK 表现如何？CLOCK 使用不同数量的时钟位，表现如何？

5．使用像 valgrind 这样的程序来测试真实应用程序并生成虚拟页面引用序列。例如，运行 valgrind --tool = lackey --trace-mem = yes ls 将为程序 ls 所做的每个指令和数据引用，输出近乎完整的引用追踪。为了使上述仿真器有用，你必须首先将每个虚拟内存引用转换为虚拟页码参考（通过屏蔽偏移量并向右移位来完成）。为了满足大部分请求，你的应用程序追踪需要多大的缓存？随着缓存大小的增加绘制其工作集的图形。

第 23 章　VAX/VMS 虚拟内存系统

在我们结束对虚拟内存的研究之前，让我们仔细研究一下 VAX/VMS 操作系统[LL82]的虚拟内存管理器，它特别干净漂亮。本章将讨论该系统，说明如何在一个完整的内存管理器中，将先前章节中提出的一些概念结合在一起。

23.1　背景

数字设备公司（DEC）在 20 世纪 70 年代末推出了 VAX-11 小型机体系结构。在微型计算机时代，DEC 是计算机行业的一个大玩家。遗憾的是，一系列糟糕的决定和个人计算机的出现慢慢（但不可避免地）导致该公司走向倒闭[C03]。该架构有许多实现，包括 VAX-11/780 和功能较弱的 VAX-11/750。

该系统的操作系统被称为 VAX/VMS（或者简单的 VMS），其主要架构师之一是 Dave Cutler，他后来领导开发了微软 Windows NT [C93]。VMS 面临通用性的问题，即它将运行在各种机器上，包括非常便宜的 VAXen（是的，这是正确的复数形式），以及同一架构系列中极高端和强大的机器。因此，操作系统必须具有一些机制和策略，适用于这一系列广泛的系统（并且运行良好）。

关键问题：如何避免通用性“魔咒”

操作系统常常有所谓的“通用性魔咒”问题，它们的任务是为广泛的应用程序和系统提供一般支持。其根本结果是操作系统不太可能很好地支持任何一个安装。VAX-11 体系结构有许多不同的实现。那么，如何构建操作系统以便在各种系统上有效运行？

附带说一句，VMS 是软件创新的很好例子，用于隐藏架构的一些固有缺陷。尽管操作系统通常依靠硬件来构建高效的抽象和假象，但有时硬件设计人员并没有把所有事情都做好。在 VAX 硬件中，我们会看到一些例子，也会看到尽管存在这些硬件缺陷，VMS 操作系统如何构建一个有效的工作系统。

23.2　内存管理硬件

VAX-11 为每个进程提供了一个 32 位的虚拟地址空间，分为 512 字节的页。因此，虚拟地址由 23 位 VPN 和 9 位偏移组成。此外，VPN 的高两位用于区分页所在的段。因此，如前所述，该系统是分页和分段的混合体。

地址空间的下半部分称为“进程空间”，对于每个进程都是唯一的。在进程空间的前半部分（称为 P0）中，有用户程序和一个向下增长的堆。在进程空间的后半部分（P1），有向上增长的栈。地址空间的上半部分称为系统空间（S），尽管只有一半被使用。受保护的操作系统代码和数据驻留在此处，操作系统以这种方式跨进程共享。

VMS 设计人员的一个主要关注点是 VAX 硬件中的页大小非常小（512 字节）。由于历史原因选择的这种尺寸，存在一个根本性问题，即简单的线性页表过大。因此，VMS 设计人员的首要目标之一是确保 VMS 不会用页表占满内存。

系统通过两种方式，减少了页表对内存的压力。首先，通过将用户地址空间分成两部分，VAX-11 为每个进程的每个区域（P0 和 P1）提供了一个页表。因此，栈和堆之间未使用的地址空间部分不需要页表空间。基址和界限寄存器的使用与你期望的一样。一个基址寄存器保存该段的页表的地址，界限寄存器保存其大小（即页表项的数量）。

其次，通过在内核虚拟内存中放置用户页表（对于 P0 和 P1，因此每个进程两个），操作系统进一步降低了内存压力。因此，在分配或增长页表时，内核在段 S 中分配自己的虚拟内存空间。如果内存受到严重压力，内核可以将这些页表的页面交换到磁盘，从而使物理内存可以用于其他用途。

将页表放入内核虚拟内存意味着地址转换更加复杂。例如，要转换 P0 或 P1 中的虚拟地址，硬件必须首先尝试在其页表中查找该页的页表项（该进程的 P0 或 P1 页表）。但是，在这样做时，硬件可能首先需要查阅系统页表（它存在于物理内存中）。随着地址转换完成，硬件可以知道页表页的地址，然后最终知道所需内存访问的地址。幸运的是，VAX 的硬件管理的 TLB 让所有这些工作更快，TLB 通常（很有可能）会绕过这种费力的查找。

23.3 一个真实的地址空间

研究 VMS 有一个很好的方面，我们可以看到如何构建一个真正的地址空间（见图 23.1）。到目前为止，我们一直假设了一个简单的地址空间，只有用户代码、用户数据和用户堆，但正如我们上面所看到的，真正的地址空间显然更复杂。

补充：为什么空指针访问会导致段错误

你现在应该很好地理解一个空指针引用会发生什么。通过这样做，进程生成了一个虚拟地址 0：

```
int *p = NULL; // set p = 0
*p = 10;       // try to store value 10 to virtual address 0
```

硬件试图在 TLB 中查找 VPN（这里也是 0），遇到 TLB 未命中。查询页表，并且发现 VPN 0 的条目被标记为无效。因此，我们遇到无效的访问，将控制权交给操作系统，这可能会终止进程（在 UNIX 系统上，会向进程发出一个信号，让它们对这样的错误做出反应。但是如果信号未被捕获，则会终止进程）。

例如，代码段永远不会从第 0 页开始。相反，该页被标记为不可访问，以便为检测空指针（null-pointer）访问提供一些支持。因此，设计地址空间时需要考虑的一个问题是对调试的支持，这正是无法访问的零页所提供的。

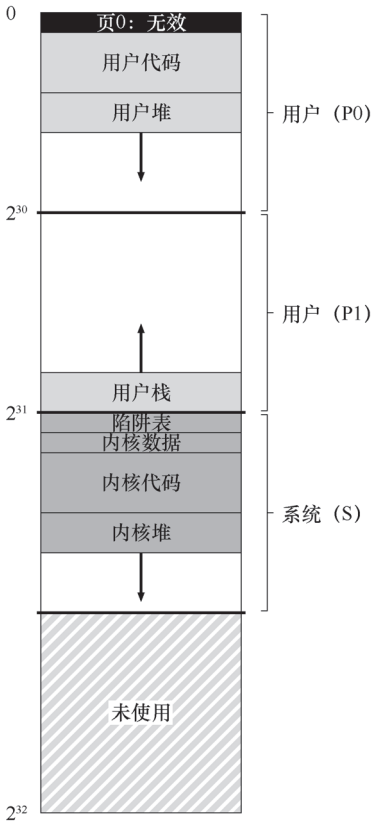


图 23.1 VAX / VMS 地址空间

也许更重要的是，内核虚拟地址空间（即其数据结构和代码）是每个用户地址空间的一部分。在上下文切换时，操作系统改变 P0 和 P1 寄存器以指向即将运行的进程的适当页表。但是，它不会更改 S 基址和界限寄存器，并因此将“相同的”内核结构映射到每个用户的地址空间。

内核映射到每个地址空间，这有一些原因。这种结构使得内核的运转更轻松。例如，如果操作系统收到用户程序（例如，在 write()系统调用中）递交的指针，很容易将数据从该指针处复制到它自己的结构。操作系统自然是写好和编译好的，无须担心它访问的数据来自哪里。相反，如果内核完全位于物理内存中，那么将页表的交换页切换到磁盘是非常困难的。如果内核被赋予了自己的地址空间，那么在用户应用程序和内核之间移动数据将再次变得复杂和痛苦。通过这种构造（现在广泛使用），内核几乎就像应用程序库一样，尽管是受保护的。

关于这个地址空间的最后一点与保护有关。显然，操作系统不希望用户应用程序读取或写入操作系统数据或代码。因此，硬件必须支持页面的不同保护级别才能启用该功能。VAX 通过在页表中的保护位中指定 CPU 访问特定页面所需的特权级别来实现此目的。因此，系统数据和代码被设置为比用户数据和代码更高的保护级别。试图从用户代码访问这些信息，将会在操作系统中产生一个陷阱，并且（你猜对了）可能会终止违规进程。

23.4 页替换

VAX 中的页表项（PTE）包含以下位：一个有效位，一个保护字段（4 位），一个修改（或脏位）位，为 OS 使用保留的字段（5 位），最后是一个物理帧号码（PFN）将页面的位置存储在物理内存中。敏锐的读者可能会注意到：没有引用位（no reference bit）！因此，VMS 替换算法必须在没有硬件支持的情况下，确定哪些页是活跃的。

开发人员也担心会有“自私贪婪的内存”（memory hog）——一些程序占用大量内存，使其他程序难以运行。到目前为止，我们所看到的大部分策略都容易受到这种内存的影响。例如，LRU 是一种全局策略，不会在进程之间公平分享内存。

分段的 FIFO

为了解决这两个问题，开发人员提出了分段的 FIFO（segmented FIFO）替换策略[RL81]。想法很简单：每个进程都有一个可以保存在内存中的最大页数，称为驻留集大小（Resident Set Size，RSS）。每个页都保存在 FIFO 列表中。当一个进程超过其 RSS 时，“先入”的页被驱逐。FIFO 显然不需要硬件的任何支持，因此很容易实现。

正如我们前面看到的，纯粹的 FIFO 并不是特别好。为了提高 FIFO 的性能，VMS 引入了两个二次机会列表（second-chance list），页在从内存中被踢出之前被放在其中。具体来说，是全局的干净页空闲列表和脏页列表。当进程 P 超过其 RSS 时，将从其每个进程的 FIFO 中移除一个页。如果干净（未修改），则将其放在干净页列表的末尾。如果脏（已修改），则将其放在脏页列表的末尾。

如果另一个进程 Q 需要一个空闲页，它会从全局干净列表中取出第一个空闲页。但是，如果原来的进程 P 在回收之前在该页上出现页错误，则 P 会从空闲（或脏）列表中回收，从而避免昂贵的磁盘访问。这些全局二次机会列表越大，分段的 FIFO 算法越接近 LRU [RL81]。

页聚集

VMS 采用的另一个优化也有助于克服 VMS 中的小页面问题。具体来说，对于这样的小页面，交换过程中的硬盘 I/O 可能效率非常低，因为硬盘在大型传输中效果更好。为了让交换 I/O 更有效，VMS 增加了一些优化，但最重要的是聚集（clustering）。通过聚集，VMS 将大批量的页从全局脏列表中分组到一起，并将它们一举写入磁盘（从而使它们变干净）。聚集用于大多数现代系统，因为可以在交换空间的任意位置放置页，所以操作系统对页分组，执行更少和更大的写入，从而提高性能。

补充：模拟引用位

事实证明，你不需要硬件引用位，就可以了解系统中哪些页在用。事实上，在 20 世纪 80 年代早期，Babaoglu 和 Joy 表明，VAX 上的保护位可以用来模拟引用位[BJ81]。其基本思路是：如果你想了解哪些页在系统中被活跃使用，请将页表中的所有页标记为不可访问（但请注意关于哪些页可以被进程真正访问

的信息，也许在页表项的“保留的操作系统字段”部分）。当一个进程访问一页时，它会在操作系统中产生一个陷阱。操作系统将检查页是否真的可以访问，如果是，则将该页恢复为正常保护（例如，只读或读写）。在替换时，操作系统可以检查哪些页仍然标记为不可用，从而了解哪些页最近没有被使用过。

这种引用位“模拟”的关键是减少开销，同时仍能很好地了解页的使用。标记页不可访问时，操作系统不应太激进，否则开销会过高。同时，操作系统也不能太被动，否则所有页面都会被引用，操作系统又无法知道踢出哪一页。

23.5 其他漂亮的虚拟内存技巧

VMS 有另外两个现在成为标准的技巧：按需置零和写入时复制。我们现在描述这些惰性（lazy）优化。

VMS（以及大多数现代系统）中的一种懒惰形式是页的按需置零（demand zeroing）。为了更好地理解这一点，我们来考虑一下在你的地址空间中添加一个页的例子。在一个初级实现中，操作系统响应一个请求，在物理内存中找到页，将该页添加到你的堆中，并将其置零（安全起见，这是必需的。否则，你可以看到其他进程使用该页时的内容。），然后将其映射到你的地址空间（设置页表以根据需要引用该物理页）。但是初级实现可能是昂贵的，特别是如果页没有被进程使用。

利用按需置零，当页添加到你的地址空间时，操作系统的工作很少。它会在页表中放入一个标记页不可访问的条目。如果进程读取或写入页，则会向操作系统发送陷阱。在处理陷阱时，操作系统注意到（通常通过页表项中“保留的操作系统字段”部分标记的一些位），这实际上是一个按需置零页。此时，操作系统会完成寻找物理页的必要工作，将它置零，并映射到进程的地址空间。如果该进程从不访问该页，则所有这些工作都可以避免，从而体现按需置零的好处。

提示：惰性

惰性可以使得工作推迟，但出于多种原因，这在操作系统中是有益的。首先，推迟工作可能会减少当前操作的延迟，从而提高响应能力。例如，操作系统通常会报告立即写入文件成功，只是稍后在后台将其写入硬盘。其次，更重要的是，惰性有时会完全避免完成这项工作。例如，延迟写入直到文件被删除，根本不需要写入。

VMS 有另一个很酷的优化（几乎每个现代操作系统都是这样），写时复制（copy-on-write，COW）。这个想法至少可以回溯到 TENEX 操作系统[BB+72]，它很简单：如果操作系统需要将一个页面从一个地址空间复制到另一个地址空间，不是实际复制它，而是将其映射到目标地址空间，并在两个地址空间中将其标记为只读。如果两个地址空间都只读取页面，则不会采取进一步的操作，因此操作系统已经实现了快速复制而不实际移动任何数据。

但是，如果其中一个地址空间确实尝试写入页面，就会陷入操作系统。操作系统会注意到该页面是一个 COW 页面，因此（惰性地）分配一个新页，填充数据，并将这个新页映

射到错误处理的地址空间。该进程然后继续，现在有了该页的私人副本。

COW 有用有一些原因。当然，任何类型的共享库都可以通过写时复制，映射到许多进程的地址空间中，从而节省宝贵的内存空间。在 UNIX 系统中，由于 fork()和 exec()的语义，COW 更加关键。你可能还记得，fork()会创建调用者地址空间的精确副本。对于大的地址空间，这样的复制过程很慢，并且是数据密集的。更糟糕的是，大部分地址空间会被随后的 exec()调用立即覆盖，它用即将执行的程序覆盖调用进程的地址空间。通过改为执行写时复制的 fork()，操作系统避免了大量不必要的复制，从而保留了正确的语义，同时提高了性能。

23.6 小结

现在我们已经从头到尾地复习整个虚拟存储系统。希望大多数细节都很容易明白，因为你应该已经对大部分基本机制和策略有了很好的理解。Levy 和 Lipman [LL82]出色的（简短的）论文中有更详细的介绍。建议你阅读它，这是了解这些章节背后的资源来源的好方法。

在可能的情况下，你还应该通过阅读 Linux 和其他现代系统来了解更多关于最新技术的信息。有很多原始资料，包括一些不错的书籍[BC05]。有一件事会让你感到惊讶：在诸如 VAX/VMS 这样的较早论文中看到的经典理念，仍然影响着现代操作系统的构建方式。

参考资料

[BB+72]“TENEX, A Paged Time Sharing System for the PDP-10”
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson Communications of the ACM, Volume 15, March 1972
早期的分时操作系统，有许多好的想法来自于此。写时复制只是其中之一，在这里可以找到现代系统许多其他方面的灵感，包括进程管理、虚拟内存和文件系统。

[BJ81]“Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits”Ozalp Babaoglu and William N. Joy
SOSP ’81, Pacific Grove, California, December 1981
关于如何利用机器内现有的保护机制来模拟引用位的巧妙构思。这个想法来自 Berkeley 的小组，他们正在致力于开发他们自己的 UNIX 版本，也就是所谓的伯克利系统发行版，或称为 BSD。该团队在 UNIX 的发展中有很大的影响力，包括虚拟内存、文件系统和网络方面。

[BC05]“Understanding the Linux Kernel (Third Edition)”Daniel P. Bovet and Marco Cesati
O’Reilly Media, November 2005
关于 Linux 的众多图书之一。它们很快就会过时，但许多基础知识仍然存在，值得一读。

[C03]“The Innovator’s Dilemma”Clayton M. Christenson
Harper Paperbacks, January 2003
一本关于硬盘驱动器行业的精彩图书，还涉及新的创新如何颠覆现有的技术。对于商科和计算机科学家来说，这是一本好书，提供了关于大型成功的公司如何完全失败的洞见。

[C93]“Inside Windows NT”Helen Custer and David Solomon Microsoft Press, 1993
这本关于 Windows NT 的书从头到尾地解释了系统，内容过于详细，你可能不喜欢。但认真地说，它是一本相当不错的书。

[LL82]“Virtual Memory Management in the VAX/VMS Operating System”Henry M. Levy, Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982)
这是本章的大部分原始材料来源，它简洁易读。尤其重要的是，如果你想读研究生，你需要做的就是阅读论文、工作，阅读更多的论文、做更多的工作，最后写一篇论文，然后继续工作。但它很有趣！

[RL81]“Segmented FIFO Page Replacement”Rollins Turner and Henry Levy
SIGMETRICS ’81, Las Vegas, Nevada, September 1981
一篇简短的文章，显示了对于一些工作负载，分段的 FIFO 可以接近 LRU 的性能。

第 24 章　内存虚拟化总结对话

学生：（大口吸气）哇，这部分内容很多。

教授：是的，那么……

学生：那么，我应该如何记住这一切？你懂的，为了考试？

教授：天啊，我希望这不是你试图记住它的原因。

学生：那我为什么要记住呢？

教授：算了吧，我以为你领会更好。你试图在这里学习一些东西，这样当你走进这个世界时，就会明白系统是如何工作的。

学生：嗯……你能举个例子吗？

教授：当然！当我还在研究生院时，有一次我和朋友正在测量内存存取的时间，有时候这些数字比我们预期的要高。我们认为所有数据都很好地融入了二级硬件缓存中，你知道，因此应该非常快速地访问。

学生：（点头）

教授：我们无法弄清楚发生了什么事。那么你在这种情况下做什么？很容易，问一位教授！于是我们去问一位教授，他看过我们制作的图表，简单地说“TLB”。啊哈！当然，TLB 未命中！我们为什么没有想到这个？有一个好的虚拟内存模型可以帮助诊断各种有趣的性能问题。

学生：我想我明白了。我要尝试建立这些关于事情如何工作的心智模型，以便我在那里独立工作，当系统不像预期的那样行事时，不会感到惊讶。我甚至应该能够预测系统将如何工作，只要想想它就行。

教授：确实如此。那么你学到了什么？关于虚拟内存如何工作的，你的心智模型有哪些？

学生：我认为我现在对进程引用内存时会发生什么有了很好的概念，正如您多次说过的那样，每次获取指令以及显式加载和存储时都会发生。

教授：听起来不错，说下去。

学生：那么，我会永远记住的一件事是，我们在用户程序中看到的地址，例如用 C 语言编写的……

教授：还有什么其他的语言？

学生：（继续）……是的，我知道你喜欢 C，我也是！无论如何，正如我所说的，我现在真的知道，我们在程序中可以观察到的所有地址都是虚拟地址。作为一名程序员，我只是看到了数据和代码在内存中的假象。我曾经认为能够打印指针的地址是很酷的，但现在我发现它令人沮丧——它只是一个虚拟地址！我看不到数据所在的真实物理地址。

教授：你看不到，操作系统肯定会向你隐藏的。还有什么？

学生：嗯，我认为 TLB 是一个非常关键的部分，为系统提供了一个地址转换的小硬件缓存。页表通常相当大，因此放在大而慢的内存中。没有 TLB，程序运行速度肯定会慢得

多。TLB 似乎真的让虚拟内存成为可能。我无法想象构建一个没有 TLB 的系统！我想到了一个超出 TLB 覆盖范围的程序：所有那些 TLB 未命中，简直不敢看。

教授：是的，蒙住孩子们的眼睛！除了 TLB，你还学到了什么？

学生：我现在也明白，页表是需要了解的数据结构之一。它只是一个数据结构，这意味着几乎可以使用任何结构。我们从简单的结构（如数组，即线性页表）开始，一直到多级表（它们看起来像树），甚至像内核虚拟内存中的可分页页表一样疯狂。全是为了在内存中节省一点空间！

教授：的确如此。

学生：还有一件更重要的事情：我了解到，地址转换结构需要足够灵活，以支持程序员想要处理的地址空间。在这个意义上，像多级表这样的结构是完美的。它们只在用户需要一部分地址空间时才创建表空间，因此几乎没有浪费。早期的尝试，比如简单的基址和界限寄存器，只是不够灵活。这些结构需要与用户期望和想要的虚拟内存系统相匹配。

教授：这是一个很好的观点。我们所学到的关于交换到磁盘的所有内容的情况如何？

学生：好的，学习肯定很有趣，而且很好地知道页替换的工作原理。一些基本的策略是很明显的（比如 LRU），但是建立一个真正的虚拟内存系统似乎更有趣，就像我们在 VMS 案例研究中看到的一样。但不知何故，我发现这些机制更有趣，而策略则不太有趣。

教授：哦，那是为什么？

学生：正如你所说的那样，最终解决策略问题的好办法很简单：购买更多的内存。但是你需要理解的机制才能知道事情是如何运作的。说到……

教授：什么？

学生：好吧，我的机器现在运行速度有点慢……而且内存肯定不会太贵……

教授：噢，很好，很好！这里有些钱，去买一些 DRAM，小事情。

学生：谢谢教授！我再也不会交换到硬盘了——或者，如果发生交换，至少我会知道实际发生了什么！