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……

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value    : %d\n", counter);
    return 0;
}
```

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）