xv6: 一个简单的、类 Unix 的教学操作系统

Russ Cox Frans Kaashoek Robert Morris

July 30, 2025

Contents

前言与致谢

这是一份用于操作系统课程的教材草稿。它通过研究一个名为 xv6 的示例内核来解释操作系统的主要概念。Xv6 仿照 Dennis Ritchie 和 Ken Thompson 的 Unix 第 6 版 (v6) [17] 建模。Xv6 松散地遵循了 v6 的结构和风格，但是使用 ANSI C [7] 为多核 RISC-V [15] 实现。

本文本应与 xv6 的源代码一起阅读，这种方法受到了 John Lions 的《UNIX 第 6 版评注》 [11] 的启发；文本中包含了指向 https://github.com/mit-pdos/xv6-riscv 源代码的超链接。有关 v6 和 xv6 的在线资源，包括几个使用 xv6 的实验作业，请参见 https://pdos.csail.mit.edu/6.1810。

我们在 MIT 的操作系统课程 6.828 和 6.1810 中使用了这本教材。我们感谢这些课程的教职员工、助教和学生，他们都直接或间接地为 xv6 做出了贡献。我们尤其要感谢 Adam Belay、Austin Clements 和 Nickolai Zeldovich。最后，我们要感谢那些通过电子邮件向我们报告文本中的错误或提出改进建议的人们：Abutalib Aghayev, Sebastian Boehm, brandb97, Anton Burtsev, Raphael Carvalho, Tej Chajed,Brendan Davidson, Rasit Eskicioglu, Color Fuzzy, Wojciech Gac, Giuseppe, Tao Guo, Haibo Hao, Naoki Hayama, Chris Henderson, Robert Hilderman, Eden Hochbaum, Wolfgang Keller, Paweł Kraszewski, Henry Laih, Jin Li, Austin Liew, lyazj@github.com, Pavan Maddamsetti, Jacek Masiulaniec, Michael McConville, m3hm00d, miguelgvieira, Mark Morrissey, Muhammed Mourad, Harry Pan, Harry Porter, Siyuan Qian, Zhefeng Qiao, Askar Safin, Salman Shah, Huang Sha, Vikram Shenoy, Adeodato Simó, Ruslan Savchenko, Pawel Szczurko, Warren Toomey, tyfkda, tzerbib, Vanush Vaswani, Xi Wang, and Zou Chang Wei, Sam Whitlock, Qiongsi Wu, LucyShawYang, ykf1114@gmail.com, and Meng Zhou

如果您发现错误或有改进建议，请发送电子邮件至 Frans Kaashoek 和 Robert Morris (kaashoek,rtm@csail.mit.edu)。

Chapter 1

操作系统接口

操作系统的任务是在多个程序之间共享一台计算机，并提供一组比硬件本身支持的更有用的服务。操作系统管理和抽象底层硬件，因此，例如，文字处理器无需关心正在使用哪种类型的磁盘硬件。操作系统在多个程序之间共享硬件，以便它们可以同时运行（或看起来是同时运行）。最后，操作系统为程序之间的交互提供了受控的方式，以便它们可以共享数据或协同工作。

操作系统通过一个接口向用户程序提供服务。设计一个好的接口是困难的。一方面，我们希望接口简单而精炼，因为这样更容易确保实现的正确性。另一方面，我们可能想为应用程序提供许多复杂的功能。解决这种矛盾的诀窍是设计依赖于少数机制的接口，这些机制可以组合起来提供很大的通用性。

本书使用一个单一的操作系统作为具体例子来说明操作系统的概念。那个操作系统，xv6，提供了由 Ken Thompson 和 Dennis Ritchie 的 Unix 操作系统 [17] 引入的基本接口，并模仿了 Unix 的内部设计。Unix 提供了一个精炼的接口，其机制结合得很好，提供了惊人的通用性。这个接口非常成功，以至于现代操作系统——BSD、Linux、macOS、Solaris，甚至在较小程度上，Microsoft Windows——都有类似 Unix 的接口。理解 xv6 是理解这些系统以及许多其他系统的一个良好开端。

如图1.1所示，xv6 采用了传统的内核形式，它是一个为正在运行的程序提供服务的特殊程序。每个正在运行的程序，称为进程，都有一块内存，其中包含指令、数据和一个栈。指令实现了程序的计算。数据是计算作用于的变量。栈组织程序的过程序调用。一台给定的计算机通常有许多进程，但只有一个内核。

当一个进程需要调用内核服务时，它会调用一个系统调用，这是操作系统接口中的一个调用。系统调用进入内核；内核执行服务并返回。因此，一个进程在用户空间和内核空间之间交替执行。

如后续章节所详述，内核使用 CPU[1]提供的硬件保护机制来确保每个在用户空间执行的进程只能访问自己的内存。内核以实现这些保护所需的硬件特权执行；用户程序在没有这些特权的情况下执行。当用户程序调用系统调用时，硬件会提升特权级别，并开始执行内核中预先安排好的函数。

内核提供的系统调用集合是用户程序看到的接口。xv6 内核提供了 Unix 内核传统上提

[1]本文档通常使用术语 CPU（中央处理单元的缩写）来指代执行计算的硬件元件。其他文档（例如, RISC-V 规范）也使用处理器、核心和 hart 等词来代替 CPU。

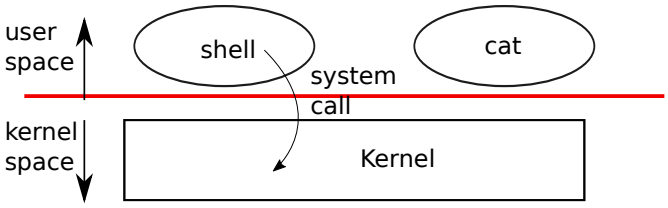

Figure 1.1: 一个内核和两个用户进程。

供的一部分服务和系统调用。图1.2列出了 xv6 的所有系统调用。

本章的其余部分概述了 xv6 的服务——进程、内存、文件描述符、管道和文件系统——并通过代码片段和对 shell（Unix 的命令行用户界面）如何使用它们的讨论来说明它们。shell 对系统调用的使用说明了它们是如何被精心设计的。

shell 是一个普通的程序，它从用户那里读取命令并执行它们。shell 是一个用户程序，而不是内核的一部分，这一事实说明了系统调用接口的强大之处：shell 没有什么特别之处。这也意味着 shell 很容易被替换；因此，现代 Unix 系统有各种各样的 shell 可供选择，每种都有自己的用户界面和脚本功能。xv6 shell 是 Unix Bourne shell 精髓的一个简单实现。其实现可以在(user/sh.c:1)找到。

1.1 进程和内存

一个 xv6 进程由用户空间内存（指令、数据和栈）和内核私有的每个进程的状态组成。Xv6 分时处理进程：它透明地在等待执行的进程集合中切换可用的 CPU。当一个进程不执行时，xv6 会保存该进程的 CPU 寄存器，并在下次运行该进程时恢复它们。内核为每个进程关联一个进程标识符，或PID。

一个进程可以使用fork系统调用创建一个新进程。fork为新进程提供调用进程内存的精确副本：它将调用进程的指令、数据和栈复制到新进程的内存中。fork在原始进程和新进程中都会返回。在原始进程中，fork返回新进程的 PID。在新进程中，fork返回零。原始进程和新进程通常被称为父进程和子进程。

例如，考虑以下用 C 编程语言 [7] 编写的程序片段：

```
int pid = fork();
if(pid > 0){
  printf("parent: child=%d\n", pid);
  pid = wait((int *) 0);
  printf("child %d is done\n", pid);
} else if(pid == 0){
  printf("child: exiting\n");
  exit(0);
} else {
  printf("fork error\n");
}
```

exit系统调用使调用进程停止执行并释放资源，如内存和打开的文件。Exit 接受一个整数状态参数，通常 0 表示成功，1 表示失败。wait系统调用返回当前进程的一个已退出（或

系统调用	描述
int fork()	创建一个进程，返回子进程的 PID。
int exit(int status)	终止当前进程；状态报告给 wait()。无返回。
int wait(int *status)	等待一个子进程退出；退出状态在 *status 中；返回子进程 PID。
int kill(int pid)	终止进程 PID。成功返回 0，错误返回-1。
int getpid()	返回当前进程的 PID。
int sleep(int n)	暂停 n 个时钟周期。
int exec(char *file, char *argv[])	加载一个文件并带参数执行它；仅在出错时返回。
char *sbrk(int n)	将进程的内存增长 n 个零字节。返回新内存的起始地址。
int open(char *file, int flags)	打开一个文件；flags 指示读/写；返回一个 fd（文件描述符）。
int write(int fd, char *buf, int n)	从 buf 向文件描述符 fd 写入 n 个字节；返回 n。
int read(int fd, char *buf, int n)	读入 n 个字节到 buf；返回读取的字节数；文件末尾返回 0。
int close(int fd)	释放打开的文件 fd。
int dup(int fd)	返回一个新的文件描述符，引用与 fd 相同的文件。
int pipe(int p[])	创建一个管道，将读/写文件描述符放入 p[0] 和 p[1]。
int chdir(char *dir)	更改当前目录。
int mkdir(char *dir)	创建一个新目录。
int mknod(char *file, int, int)	创建一个设备文件。
int fstat(int fd, struct stat *st)	将有关打开文件的信息放入 *st。
int link(char *file1, char *file2)	为文件 file1 创建另一个名称 (file2)。
int unlink(char *file)	删除一个文件。

Figure 1.2: Xv6 系统调用。如无特别说明，这些调用成功时返回 0，出错时返回-1。

被杀死）的子进程的 PID，并将子进程的退出状态复制到传递给 wait 的地址；如果调用者的子进程都没有退出，wait会等待其中一个退出。如果调用者没有子进程，wait立即返回-1。如果父进程不关心子进程的退出状态，它可以向wait传递一个 0 地址。

在示例中，输出行

```
parent: child=1234
child: exiting
```

可能会以任何顺序出现（甚至混合在一起），这取决于父进程或子进程哪个先到达其printf调用。子进程退出后，父进程的wait返回，导致父进程打印

```
parent: child 1234 is done
```

尽管子进程最初具有与父进程相同的内存内容，但父进程和子进程使用独立的内存和独立的寄存器执行：在一个进程中更改一个变量不会影响另一个进程。例如，当wait的返回值存储到父进程的pid中时，它不会改变子进程中的变量pid。子进程中的pid值仍将为零。

exec系统调用用从文件系统中加载的新内存映像替换调用进程的内存。该文件必须具有特定的格式，该格式指定文件的哪一部分包含指令，哪一部分是数据，从哪个指令开始执行等。Xv6 使用 ELF 格式，第3章将对此进行更详细的讨论。通常，该文件是编译程序源代码的结果。当exec成功时，它不会返回到调用程序；相反，从文件加载的指令在 ELF 头中声明的入口点开始执行。exec接受两个参数：包含可执行文件的文件名和一个字符串参

数数组。例如：

```
char *argv[3];
argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo", argv);
printf("exec error\n");
```

该片段将调用程序替换为以参数列表echo hello运行的程序/bin/echo的实例。大多数程序忽略参数数组的第一个元素，该元素通常是程序的名称。

xv6 shell 使用上述调用代表用户运行程序。shell 的主要结构很简单；请参见main (user/sh.c:146)。主循环使用getcmd从用户读取一行输入。然后它调用fork，创建一个 shell 进程的副本。父进程调用wait，而子进程运行命令。例如，如果用户向 shell 键入 "echo hello"，runcmd将被调用，参数为 "echo hello"。如果exec成功，则子进程将执行来自echo的指令，而不是runcmd。在某个时候echo会调用exit，这将导致父进程从main (user/sh.c:146)中的wait返回。

您可能想知道为什么fork和exec没有合并成一个调用；我们稍后会看到，shell 在其 I/O 重定向的实现中利用了这种分离。为了避免创建重复进程然后立即替换它（使用exec）的浪费，操作系统内核通过使用虚拟内存技术（如写时复制（见第4.6节））来优化fork在这种用例下的实现。

Xv6 隐式地分配大多数用户空间内存：fork分配子进程复制父进程内存所需的内存，而exec分配足以容纳可执行文件的内存。在运行时需要更多内存的进程（例如，对于malloc）可以调用sbrk(n)来将其数据内存增长 n 个零字节；sbrk返回新内存的位置。

1.2 I/O 和文件描述符

文件描述符是一个小整数，代表一个内核管理的对象，进程可以从中读取或向其写入。进程可以通过打开文件、目录或设备，或者通过创建管道，或者通过复制现有描述符来获取文件描述符。为简单起见，我们通常将文件描述符引用的对象称为 "文件"；文件描述符接口抽象了文件、管道和设备之间的差异，使它们都看起来像字节流。我们将输入和输出称为 I/O。

在内部，xv6 内核使用文件描述符作为每个进程表的索引，因此每个进程都有一个从零开始的私有文件描述符空间。按照惯例，进程从文件描述符 0（标准输入）读取，将输出写入文件描述符 1（标准输出），并将错误消息写入文件描述符 2（标准错误）。正如我们将看到的，shell 利用这个惯例来实现 I/O 重定向和管道。shell 确保它始终有三个打开的文件描述符(user/sh.c:152)，默认情况下是控制台的文件描述符。

read和write系统调用从由文件描述符命名的打开文件中读取字节和写入字节。调用read(fd, buf, n)最多从文件描述符fd读取n个字节，将它们复制到buf中，并返回读取的字节数。每个引用文件的文件描述符都有一个与之关联的偏移量。read从当前文件偏移量读取数据，然后将该偏移量增加读取的字节数：随后的read将返回第一个read返回的字节之后的字节。当没有更多字节可读时，read返回零以指示文件结尾。

调用write(fd, buf, n)将n个字节从buf写入文件描述符fd并返回写入的字节数。只有在发生错误时才会写入少于n个字节。像read一样，write在当前文件偏移量处写入数据，然后将

该偏移量增加写入的字节数：每个write都从前一个停止的地方继续。

以下程序片段（构成程序cat的精髓）将其标准输入的数据复制到其标准输出。如果发生错误，它会向标准错误写入一条消息。

```
char buf[512];
int n;
for(;;){
  n = read(0, buf, sizeof buf);
  if(n == 0)
    break;
  if(n < 0){
    fprintf(2, "read error\n");
    exit(1);
  }
  if(write(1, buf, n) != n){
    fprintf(2, "write error\n");
    exit(1);
  }
}
```

代码片段中需要注意的重要一点是，cat不知道它是在从文件、控制台还是管道读取。同样，cat也不知道它是在向控制台、文件还是其他任何地方打印。文件描述符的使用以及文件描述符 0 是输入、文件描述符 1 是输出的惯例，使得cat的实现很简单。

close系统调用释放一个文件描述符，使其可以被未来的open、pipe或dup系统调用（见下文）重用。新分配的文件描述符始终是当前进程的最低编号的未使用描述符。

文件描述符和fork的交互使得 I/O 重定向易于实现。fork复制父进程的文件描述符表及其内存，因此子进程以与父进程完全相同的打开文件开始。系统调用exec替换调用进程的内存，但保留其文件表。这种行为允许 shell 通过 forking，在子进程中重新打开选定的文件描述符，然后调用exec来运行新程序，从而实现 I/O 重定向。这是一个 shell 为命令cat < input.txt运行的代码的简化版本：

```
char *argv[2];
argv[0] = "cat";
argv[1] = 0;
if(fork() == 0) {
  close(0);
  open("input.txt", O_RDONLY);
  exec("cat", argv);
}
```

子进程关闭文件描述符 0 后，open保证为新打开的input.txt使用该文件描述符：0 将是最小的可用文件描述符。然后cat执行，其文件描述符 0（标准输入）引用input.txt。父进程的文件描述符不受此序列的影响，因为它只修改子进程的描述符。

xv6 shell 中 I/O 重定向的代码正是以这种方式工作的(user/sh.c:83)。回想一下，在代码的这一点上，shell 已经 fork 了子 shell，并且runcmd将调用exec来加载新程序。

open的第二个参数由一组标志组成，以位表示，控制open的功能。可能的值在文件控制 (fcntl) 头文件(kernel/fcntl.h:1-5)中定义：O_RDONLY、O_WRONLY、O_RDWR、

O_CREATE和O_TRUNC，它们指示open 以只读方式打开文件，或以只写方式，或以读写方式，如果文件不存在则创建文件，并将文件截断为零长度。

现在应该清楚为什么fork和exec是分开的调用很有帮助: 在这两者之间, shell 有机会重定向子进程的 I/O, 而不会干扰主 shell 的 I/O 设置。人们可以想象一个假设的组合forkexec系统调用，但使用这种调用进行 I/O 重定向的选项似乎很笨拙。shell 可以在调用forkexec之前修改自己的 I/O 设置（然后撤消这些修改）；或者forkexec可以将 I/O 重定向的指令作为参数；或者（最不吸引人的）每个像cat这样的程序都可以被教导自己进行 I/O 重定向。

尽管fork复制了文件描述符表，但每个底层的文件偏移量在父子进程之间是共享的。考虑这个例子:

```
if(fork() == 0) {
  write(1, "hello ", 6);
  exit(0);
} else {
  wait(0);
  write(1, "world\n", 6);
}
```

在这个片段的末尾，附加到文件描述符 1 的文件将包含数据hello world。父进程中的write（由于wait，仅在子进程完成后运行）从子进程的write停止的地方继续。这种行为有助于从 shell 命令序列中产生顺序输出，例如(echo hello; echo world) >output.txt。

dup系统调用复制一个现有的文件描述符，返回一个新的文件描述符，它引用相同的底层 I/O 对象。两个文件描述符共享一个偏移量，就像fork复制的文件描述符一样。这是另一种将hello world写入文件的方式:

```
fd = dup(1);
write(1, "hello ", 6);
write(fd, "world\n", 6);
```

如果两个文件描述符是通过一系列fork和dup调用从同一个原始文件描述符派生出来的，那么它们共享一个偏移量。否则，文件描述符不共享偏移量，即使它们是由对同一文件的open调用产生的。dup允许 shell 实现像这样的命令: ls existing-file non-existing-file > tmp1 2>&1。2>&1告诉 shell 给命令一个文件描述符 2，它是描述符 1 的副本。现有文件的名称和不存在文件的错误消息都将显示在文件tmp1中。xv6 shell 不支持错误文件描述符的 I/O 重定向，但现在您知道如何实现它了。

文件描述符是一个强大的抽象，因为它们隐藏了它们所连接的细节：一个向文件描述符 1 写入的进程可能是在向文件、像控制台这样的设备或管道写入。

1.3 管道

管道是一个小的内核缓冲区，作为一对文件描述符暴露给进程，一个用于读取，一个用于写入。向管道的一端写入数据，使得这些数据可以从管道的另一端读取。管道为进程间通信提供了一种方式。

以下示例代码运行程序wc，其标准输入连接到管道的读取端。

```
int p[2];
```

```
char *argv[2];
argv[0] = "wc";
argv[1] = 0;
pipe(p);
if(fork() == 0) {
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  exec("/bin/wc", argv);
} else {
  close(p[0]);
  write(p[1], "hello world\n", 12);
  close(p[1]);
}
```

程序调用pipe，它创建一个新管道，并将读写文件描述符记录在数组p中。fork之后，父子进程都有引用该管道的文件描述符。子进程调用close和dup使文件描述符零引用管道的读取端，关闭p中的文件描述符，并调用exec来运行wc。当wc从其标准输入读取时，它从管道中读取。父进程关闭管道的读取端，向管道写入，然后关闭写入端。

如果没有数据可用，对管道的read会等待数据被写入或所有引用写入端的文件描述符被关闭；在后一种情况下，read将返回 0，就像到达数据文件的末尾一样。read会阻塞直到不可能有新数据到达，这是为什么在上面的wc执行之前，子进程关闭管道的写入端很重要的原因之一：如果wc的一个文件描述符引用了管道的写入端，wc将永远看不到文件结束符。

xv6 shell 以类似于上述代码的方式实现诸如grep fork sh.c | wc -l之类的管道(user/sh.c:101)。子进程创建一个管道以连接管道的左端和右端。然后它为管道的左端调用fork和runcmd，为管道的右端调用fork和runcmd，并等待两者完成。管道的右端可能是一个本身包含管道的命令（例如，a | b | c），它本身会派生两个新的子进程（一个用于b，一个用于c）。因此，shell 可能会创建一个进程树。该树的叶子是命令，内部节点是等待左右子节点完成的进程。

管道似乎并不比临时文件更强大：管道

```
echo hello world | wc
```

可以不用管道实现为

```
echo hello world >/tmp/xyz; wc </tmp/xyz
```

在这种情况下，管道至少比临时文件有三个优势。首先，管道会自动清理自己；使用文件重定向，shell 必须小心在完成后删除/tmp/xyz。其次，管道可以传递任意长的数据流，而文件重定向需要在磁盘上有足够的可用空间来存储所有数据。第三，管道允许管道阶段的并行执行，而文件方法要求第一个程序在第二个程序开始之前完成。

1.4 文件系统

xv6 文件系统提供数据文件，其中包含未解释的字节数组，以及目录，其中包含对数据文件和其他目录的命名引用。目录形成一个树，从一个称为根的特殊目录开始。像/a/b/c这

样的路径引用根目录/中名为a的目录中名为b的目录中名为c的文件或目录。不以/开头的路径是相对于调用进程的当前目录来评估的，该目录可以用chdir系统调用来更改。这两个代码片段打开相同的文件（假设所有涉及的目录都存在）：

```
chdir("/a");
chdir("b");
open("c", O_RDONLY);
open("/a/b/c", O_RDONLY);
```

第一个片段将进程的当前目录更改为/a/b；第二个片段既不引用也不更改进程的当前目录。

有创建新文件和目录的系统调用：mkdir创建一个新目录，带有O_CREATE标志的open创建一个新数据文件，mknod创建一个新设备文件。这个例子说明了所有三种情况：

```
mkdir("/dir");
fd = open("/dir/file", O_CREATE|O_WRONLY);
close(fd);
mknod("/console", 1, 1);
```

mknod创建一个引用设备的特殊文件。与设备文件关联的是主设备号和次设备号（mknod的两个参数），它们唯一地标识一个内核设备。当一个进程稍后打开一个设备文件时，内核会将read和write系统调用转移到内核设备实现，而不是将它们传递给文件系统。

一个文件的名称与文件本身是不同的；同一个底层文件，称为 inode，可以有多个名称，称为链接。每个链接由一个目录中的一个条目组成；该条目包含一个文件名和对一个 inode 的引用。一个 inode 持有关于文件的元数据，包括其类型（文件、目录或设备）、其长度、文件内容在磁盘上的位置，以及文件的链接数。

fstat系统调用从文件描述符引用的 inode 中检索信息。它填充一个在stat.h (kernel/stat.h)中定义的struct stat：

```
#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device
struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

link系统调用创建另一个引用与现有文件相同 inode 的文件系统名称。这个片段创建了一个名为a和b的新文件。

```
open("a", O_CREATE|O_WRONLY);
link("a", "b");
```

从a读取或写入与从b读取或写入相同。每个 inode 都由一个唯一的 inode 号标识。在上面的代码序列之后，可以通过检查fstat的结果来确定a和b引用相同的基础内容：两者都将返回相同的 inode 号（ino），并且nlink计数将设置为 2。

unlink系统调用从文件系统中删除一个名称。只有当文件的链接计数为零且没有文件描述符引用它时，文件的 inode 和保存其内容的磁盘空间才会被释放。因此，将

```
unlink("a");
```

添加到最后一个代码序列会使 inode 和文件内容可以作为b访问。此外，

```
fd = open("/tmp/xyz", O_CREATE|O_RDWR);
unlink("/tmp/xyz");
```

是创建没有名称的临时 inode 的惯用方法，当进程关闭fd或退出时，该 inode 将被清理。

Unix 提供了可从 shell 调用的用户级程序作为文件实用程序，例如mkdir、ln和rm。这种设计允许任何人通过添加新的用户级程序来扩展命令行界面。事后看来，这个计划似乎是显而易见的，但在 Unix 时代设计的其他系统通常将这些命令内置到 shell 中（并将 shell 内置到内核中）。

一个例外是cd，它内置在 shell 中(user/sh.c:161)。cd必须更改 shell 本身的当前工作目录。如果cd作为常规命令运行，那么 shell 将派生一个子进程，子进程将运行cd，并且cd将更改子进程的工作目录。父进程（即 shell）的工作目录不会改变。

1.5 现实世界

Unix 的“标准”文件描述符、管道和方便的 shell 语法的组合，是编写通用可重用程序的一个重大进步。这个想法激发了一种“软件工具”文化，这是 Unix 强大功能和普及的主要原因，而 shell 是第一个所谓的“脚本语言”。Unix 系统调用接口至今仍在 BSD、Linux 和 macOS 等系统中存在。

Unix 系统调用接口已通过可移植操作系统接口（POSIX）标准进行了标准化。Xv6 不符合 POSIX 标准：它缺少许多系统调用（包括像lseek这样的基本调用），并且它提供的许多系统调用与标准不同。我们对 xv6 的主要目标是简单和清晰，同时提供一个简单的类 UNIX 系统调用接口。有几个人通过添加更多的系统调用和一个简单的 C 库来扩展 xv6，以便运行基本的 Unix 程序。然而，现代内核提供的系统调用和内核服务种类比 xv6 多得多。例如，它们支持网络、窗口系统、用户级线程、许多设备的驱动程序等等。现代内核不断快速发展，并提供许多超出 POSIX 的功能。

Unix 用一组文件名称和文件描述符接口统一了对多种类型资源（文件、目录和设备）的访问。这个想法可以扩展到更多种类的资源；一个很好的例子是 Plan 9[16]，它将“资源即文件”的概念应用于网络、图形等。然而，大多数源自 Unix 的操作系统并没有走这条路。

文件系统和文件描述符是强大的抽象。即便如此，还有其他操作系统接口的模型。Multics 是 Unix 的前身，它以一种使其看起来像内存的方式抽象文件存储，产生了一种截然不同的接口风格。Multics 设计的复杂性直接影响了 Unix 的设计者，他们的目标是构建更简单的东西。

Xv6 不提供用户或保护一个用户免受另一个用户侵害的概念；用 Unix 的术语来说，所有 xv6 进程都以 root 身份运行。

本书探讨了 xv6 如何实现其类 Unix 接口，但这些思想和概念不仅适用于 Unix。任何操作系统都必须将进程复用到基础硬件上，将进程相互隔离，并提供受控的进程间通信机

制。学习 xv6 之后，您应该能够看到其他更复杂的操作系统，并在这些系统中看到 xv6 的基本概念。

1.6 练习

1. 编写一个程序，使用 UNIX 系统调用在一对管道上（每个方向一个）在两个进程之间“乒乓”一个字节。测量程序的性能，以每秒交换次数为单位。

Chapter 2

操作系统组织

操作系统的关键要求是同时支持多种活动。例如，使用第 1章中描述的系统调用接口，一个进程可以用 fork 启动新进程。操作系统必须在这些进程之间分时共享计算机的资源。例如，即使进程数量多于硬件 CPU 的数量，操作系统也必须确保所有进程都有机会执行。操作系统还必须为进程之间安排隔离。也就是说，如果一个进程有 bug 并发生故障，它不应该影响不依赖于该有 bug 进程的进程。然而，完全隔离又太强了，因为进程应该能够有意地进行交互；管道就是一个例子。因此，操作系统必须满足三个要求：多路复用、隔离和交互。

本章概述了操作系统如何组织以实现这三个要求。事实证明，有很多方法可以做到这一点，但本文侧重于围绕单体内核的主流设计，许多 Unix 操作系统都使用这种设计。本章还概述了 xv6 进程（xv6 中的隔离单元）以及 xv6 启动时第一个进程的创建。

Xv6 运行在多核[1]RISC-V 微处理器上，其许多底层功能（例如，其进程实现）都特定于 RISC-V。RISC-V 是一个 64 位 CPU，xv6 是用“LP64”C 语言编写的，这意味着 C 编程语言中的 long（L）和指针（P）是 64 位的，但一个 int 是 32 位的。本书假定读者在某种架构上做过一些机器级编程，并将随着 RISC-V 特定思想的出现而介绍它们。用户级 ISA [2] 和特权架构 [3] 文档是完整的规范。您也可以参考《RISC-V 读本：一本开放架构地图集》 [15]。

一台完整计算机中的 CPU 被支持硬件所包围，其中大部分是 I/O 接口的形式。Xv6 是为 qemu 的“-machine virt”选项所模拟的支持硬件编写的。这包括 RAM、一个包含引导代码的 ROM、一个与用户键盘/屏幕的串行连接，以及一个用于存储的磁盘。

2.1 抽象物理资源

当遇到操作系统时，人们可能问的第一个问题是为什么要有它？也就是说，人们可以把图 1.2中的系统调用实现为一个库，应用程序与之链接。在这个方案中，每个应用程序甚至可以有自己根据其需求量身定制的库。应用程序可以直接与硬件资源交互，并以最适合应

[1]本文中的“多核”指的是共享内存但并行执行的多个 CPU，每个 CPU 都有自己的一组寄存器。本文有时使用术语多处理器作为多核的同义词，尽管多处理器也可以更具体地指具有多个不同处理器芯片的计算机。

用程序的方式使用这些资源（例如，为了获得高或可预测的性能）。一些用于嵌入式设备或实时系统的操作系统就是以这种方式组织的。

这种库方法的缺点是，如果有多个应用程序在运行，这些应用程序必须是行为良好的。例如，每个应用程序必须定期放弃 CPU，以便其他应用程序可以运行。如果所有应用程序都相互信任并且没有 bug，那么这种协作式分时方案可能是可以的。但更典型的情况是应用程序互不信任，并且有 bug，所以人们通常希望有比协作式方案提供的更强的隔离。

为了实现强隔离，禁止应用程序直接访问敏感的硬件资源，而是将资源抽象成服务是有帮助的。例如，Unix 应用程序仅通过文件系统的open、read、write和close系统调用与存储交互，而不是直接读写磁盘。这为应用程序提供了路径名的便利，并且允许操作系统（作为接口的实现者）来管理磁盘。即使隔离不是一个问题，有意交互（或只是希望互不干扰）的程序也可能会发现文件系统是比直接使用磁盘更方便的抽象。

同样，Unix 在进程之间透明地切换硬件 CPU，根据需要保存和恢复寄存器状态，这样应用程序就不必意识到分时。这种透明性使得操作系统即使在某些应用程序处于无限循环中时也能共享 CPU。

再举一个例子，Unix 进程使用exec来建立它们的内存映像，而不是直接与物理内存交互。这使得操作系统可以决定将进程放在内存的哪个位置；如果内存紧张，操作系统甚至可能将进程的一部分数据存储在磁盘上。exec还为用户提供了使用文件系统存储可执行程序映像的便利。

Unix 进程之间的许多交互形式都是通过文件描述符进行的。文件描述符不仅抽象掉了许多细节（例如，管道或文件中的数据存储在哪里），而且它们的定义方式也简化了交互。例如，如果一个管道中的一个应用程序失败了，内核会为管道中的下一个进程生成一个文件结束信号。

图 1.2中的系统调用接口经过精心设计，既提供了程序员的便利性，也提供了强隔离的可能性。Unix 接口不是抽象资源的唯一方式，但它已被证明是一种很好的方式。

2.2 用户模式、监管者模式和系统调用

强隔离要求在应用程序和操作系统之间有一个硬边界。如果应用程序出错，我们不希望操作系统失败或其他应用程序失败。相反，操作系统应该能够清理失败的应用程序并继续运行其他应用程序。为了实现强隔离，操作系统必须安排应用程序不能修改（甚至读取）操作系统的数据结构和指令，并且应用程序不能访问其他进程的内存。

CPU 为强隔离提供了硬件支持。例如，RISC-V 有三种 CPU 可以执行指令的模式：机器模式、监管者模式和用户模式。在机器模式下执行的指令拥有完全的特权；CPU 在机器模式下启动。机器模式主要用于在引导期间设置计算机。Xv6 在机器模式下执行几行代码，然后切换到监管者模式。

在监管者模式下，CPU 被允许执行特权指令：例如，启用和禁用中断，读写存放页表地址的寄存器等。如果用户模式下的应用程序试图执行特权指令，那么 CPU 不会执行该指令，而是切换到监管者模式，以便监管者模式的代码可以终止该应用程序，因为它做了不该做的事情。第 1章中的图 1.1说明了这种组织结构。一个应用程序只能执行用户模式指令（例如，加法等），并且据说是在用户空间中运行，而在监管者模式下的软件也可以执行特权指令，并且据说是在内核空间中运行。在内核空间（或监管者模式）中运行的软件被称为内核。

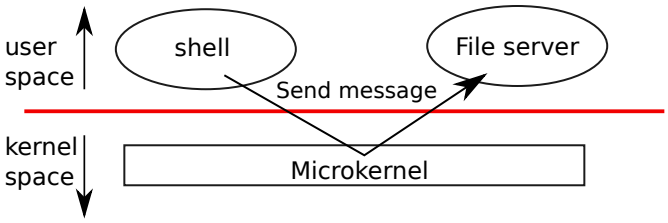

Figure 2.1: 一个带有文件系统服务器的微内核

想要调用内核函数（例如，xv6 中的read系统调用）的应用程序必须转换到内核；应用程序不能直接调用内核函数。CPU 提供一个特殊的指令，将 CPU 从用户模式切换到监管者模式，并在内核指定的入口点进入内核。（RISC-V 为此提供了 ecall 指令。）一旦 CPU 切换到监管者模式，内核就可以验证系统调用的参数（例如，检查传递给系统调用的地址是否是应用程序内存的一部分），决定是否允许应用程序执行请求的操作（例如，检查应用程序是否被允许写入指定的文件），然后拒绝它或执行它。内核控制到监管者模式的转换入口点是很重要的；如果应用程序可以决定内核入口点，一个恶意的应用程序就可以，例如，在跳过参数验证的地方进入内核。

2.3 内核组织

一个关键的设计问题是操作系统的哪一部分应该在监管者模式下运行。一种可能性是整个操作系统都驻留在内核中，因此所有系统调用的实现都在监管者模式下运行。这种组织被称为单体内核。

在这种组织中，整个操作系统由一个以完全硬件特权运行的单一程序组成。这种组织很方便，因为操作系统设计者不必决定操作系统的哪些部分不需要完全的硬件特权。此外，操作系统不同部分之间的协作也更容易。例如，一个操作系统可能有一个缓冲区缓存，可以被文件系统和虚拟内存系统共享。

单体组织的缺点是，操作系统不同部分之间的交互通常很复杂（我们将在本文的其余部分看到），因此操作系统开发人员很容易犯错。在单体内核中，一个错误是致命的，因为在监管者模式下的一个错误通常会导致内核失败。如果内核失败，计算机就会停止工作，因此所有应用程序也会失败。计算机必须重新启动才能再次启动。

为了降低内核中出错的风险，操作系统设计者可以最小化在监管者模式下运行的操作系统代码量，并在用户模式下执行大部分操作系统。这种内核组织被称为微内核。

图 2.1说明了这种微内核设计。在图中，文件系统作为一个用户级进程运行。作为进程运行的操作系统服务被称为服务器。为了允许应用程序与文件服务器交互，内核提供了一种进程间通信机制，用于将消息从一个用户模式进程发送到另一个用户模式进程。例如，如果像 shell 这样的应用程序想要读或写一个文件，它会向文件服务器发送一个消息并等待响应。

在微内核中，内核接口由一些用于启动应用程序、发送消息、访问设备硬件等的低级函数组成。这种组织使得内核相对简单，因为大部分操作系统都驻留在用户级服务器中。

在现实世界中，单体内核和微内核都很流行。许多 Unix 内核是单体的。例如，Linux 有一个单体内核，尽管一些操作系统功能作为用户级服务器运行（例如，窗口系统）。Linux

为操作系统密集型应用程序提供了高性能，部分原因是因为内核的子系统可以紧密集成。

诸如 Minix、L4 和 QNX 等操作系统被组织为带有服务器的微内核，并在嵌入式领域得到了广泛部署。L4 的一个变体 seL4 足够小，以至于已经针对内存安全和其他安全属性进行了验证 [8]。

操作系统开发者之间关于哪种组织更好有很多争论，并且没有确凿的证据表明一种比另一种更好。此外，它很大程度上取决于“更好”意味着什么：更快的性能、更小的代码大小、内核的可靠性、整个操作系统（包括用户级服务）的可靠性等。

还有一些实际考虑因素可能比哪个组织更好的问题更重要。一些操作系统有一个微内核，但为了性能原因，在内核空间运行一些用户级服务。一些操作系统有单体内核，因为它们就是这样开始的，而且几乎没有动力去转向纯粹的微内核组织，因为新功能可能比重写现有操作系统以适应微内核设计更重要。

从本书的角度来看，微内核和单体操作系统共享许多关键思想。它们实现系统调用，它们使用页表，它们处理中断，它们支持进程，它们使用锁进行并发控制，它们实现一个文件系统，等等。本书侧重于这些核心思想。

Xv6 是作为一个单体内核实现的，像大多数 Unix 操作系统一样。因此，xv6 内核接口对应于操作系统接口，并且内核实现了完整的操作系统。由于 xv6 不提供许多服务，其内核比一些微内核要小，但概念上 xv6 是单体的。

2.4 代码: xv6 组织

xv6 内核源代码位于 kernel/ 子目录中。源代码被划分为多个文件，遵循一个粗略的模块化概念；图 2.2 列出了这些文件。模块间的接口在 defs.h (kernel/defs.h) 中定义。

2.5 进程概述

xv6 中的隔离单位（与其他 Unix 操作系统一样）是进程。进程抽象防止一个进程破坏或窥探另一个进程的内存、CPU、文件描述符等。它还防止进程破坏内核本身，这样进程就无法颠覆内核的隔离机制。内核必须小心地实现进程抽象，因为一个有 bug 或恶意的应用程序可能会欺骗内核或硬件去做一些坏事（例如，规避隔离）。内核用来实现进程的机制包括用户/监管者模式标志、地址空间和线程的时间分片。

为了帮助强制隔离，进程抽象为程序提供了一种它拥有自己的私有机器的错觉。进程为程序提供了一个看起来是私有的内存系统，或地址空间，其他进程无法读写。进程还为程序提供了一个看起来是它自己的 CPU 来执行程序的指令。

Xv6 使用页表（由硬件实现）来为每个进程提供自己的地址空间。RISC-V 页表将一个虚拟地址（RISC-V 指令操作的地址）翻译（或“映射”）到一个物理地址（CPU 发送到主内存的地址）。

Xv6 为每个进程维护一个单独的页表，该页表定义了该进程的地址空间。如图 2.3所示，一个地址空间包括从虚拟地址零开始的进程的用户内存。指令排在最前面，然后是全局变量，然后是栈，最后是一个进程可以根据需要扩展的“堆”区（用于 malloc）。有许多因素限制了进程地址空间的最大大小：RISC-V 上的指针是 64 位宽的；硬件在页表中查找虚拟地址时只使用低 39 位；xv6 只使用这 39 位中的 38 位。因此，最大地址是 $2^{38} - 1$

= 0x3fffffffff，即 MAXVA (kernel/riscv.h:382)。在地址空间的顶部，xv6 放置了一个蹦床页（4096 字节）和一个陷阱帧页。Xv6 使用这两个页面来转换到内核并返回；蹦床页包含转换进出内核的代码，陷阱帧是内核保存进程用户寄存器的地方，如第 4章所述。

xv6 内核为每个进程维护许多状态,它将这些状态收集到一个 struct proc (kernel/proc.h:85) 中。一个进程最重要的内核状态是它的页表、它的内核栈和它的运行状态。我们将使用符号 p->xxx 来引用 proc 结构的元素；例如，p->pagetable 是指向进程页表的指针。

每个进程都有一个控制线程（或简称线程），它持有执行进程所需的状态。在任何给定时间，一个线程可能正在一个 CPU 上执行，或者被挂起（不执行，但能够在将来恢复执行）。为了在进程之间切换 CPU，内核会挂起当前在该 CPU 上运行的线程并保存其状态,然后恢复另一个进程先前挂起的线程的状态。线程的大部分状态（局部变量、函数调用返回地址）都存储在线程的栈上。每个进程有两个栈：一个用户栈和一个内核栈 (p->kstack)。当进程执行用户指令时，只有它的用户栈在使用，而它的内核栈是空的。当进程进入内核时（对于系统调用或中断），内核代码在进程的内核栈上执行；当一个进程在内核中时，它的用户栈仍然包含保存的数据，但没有被主动使用。一个进程的线程交替地主动使用其用户栈和内核栈。内核栈是独立的（并且受到用户代码的保护），这样即使一个进程破坏了它的用户栈，内核也可以执行。

进程可以通过执行 RISC-V ecall 指令来进行系统调用。该指令提升硬件特权级别并将程序计数器更改为内核定义的入口点。入口点的代码切换到进程的内核栈并执行实现系统调用的内核指令。当系统调用完成时，内核切换回用户栈并通过调用 sret 指令返回用户空间，该指令降低硬件特权级别并恢复执行紧跟在系统调用指令之后的用户指令。一个进程的线程可以在内核中“阻塞”以等待 I/O，并在 I/O 完成后从它离开的地方恢复。

p->state 指示进程是否已分配、准备运行、当前正在 CPU 上运行、等待 I/O 或正在退出。

p->pagetable 以 RISC-V 硬件期望的格式保存进程的页表。当在用户空间执行该进程时，Xv6 会使分页硬件使用进程的 p->pagetable。进程的页表也作为记录为存储进程内存而分配的物理页地址的记录。

总之，一个进程捆绑了两个设计思想：一个地址空间，给进程一个拥有自己内存的错觉；一个线程，给进程一个拥有自己 CPU 的错觉。在 xv6 中，一个进程由一个地址空间和一个线程组成。在真正的操作系统中，一个进程可能有多个线程以利用多个 CPU。

2.6 代码：启动 xv6，第一个进程和系统调用

为了让 xv6 更具体，我们将概述内核如何启动和运行第一个进程。后续章节将更详细地描述本次概述中出现的机制。

当 RISC-V 计算机上电时，它会自行初始化并运行存储在只读存储器中的引导加载程序。引导加载程序将 xv6 内核加载到内存中。然后，在机器模式下，CPU 从_entry (kernel/entry.S:7)开始执行 xv6。RISC-V 启动时分页硬件是禁用的：虚拟地址直接映射到物理地址。

加载程序将 xv6 内核加载到物理地址 0x80000000 的内存中。它将内核放在 0x80000000 而不是 0x0 的原因是地址范围 0x0:0x80000000 包含 I/O 设备。

函数start执行一些只能在机器模式下进行的配置，然后切换到监管者模式。为了进入监管者模式，RISC-V 提供了指令mret。该指令最常用于从先前从监管者模式到机器模式

的调用中返回。start并非从这样的调用中返回，而是将其设置为好像是这样：它在寄存器mstatus中将先前的特权模式设置为监管者，通过将main的地址写入寄存器mepc来将返回地址设置为main，通过将0写入页表寄存器satp来禁用监管者模式下的虚拟地址转换，并将所有中断和异常委托给监管者模式。

在跳转到监管者模式之前，start执行了另一项任务：它对时钟芯片进行编程以生成定时器中断。完成这些内务处理后，start通过调用mret“返回”到监管者模式。这导致程序计数器更改为main (kernel/main.c:11)，即先前存储在mepc中的地址。

一旦内核完成了exec，它就会在/init进程中返回用户空间。init (user/init.c:15)如果需要，会创建一个新的控制台设备文件，然后将其作为文件描述符 0、1 和 2 打开。然后它在控制台上启动一个 shell。系统启动完成。

2.7 安全模型

您可能想知道操作系统如何处理有 bug 或恶意的代码。因为应对恶意比处理意外的 bug 要困难得多，所以主要关注提供针对恶意的安全性是合理的。以下是操作系统设计中典型安全假设和目标的高级视图。

操作系统必须假设进程的用户级代码会尽其所能破坏内核或其他进程。用户代码可能会尝试解引用其允许的地址空间之外的指针；它可能会尝试执行任何 RISC-V 指令，甚至是那些不打算用于用户代码的指令；它可能会尝试读写任何 RISC-V 控制寄存器；它可能会尝试直接访问设备硬件；它可能会向系统调用传递巧妙的值，以试图欺骗内核崩溃或做一些愚蠢的事情。内核的目标是限制每个用户进程，使其所能做的只是读/写/执行自己的用户内存，使用 32 个通用 RISC-V 寄存器，以及以系统调用旨在允许的方式影响内核和其他进程。内核必须阻止任何其他操作。这通常是内核设计中的一个绝对要求。

对内核自身代码的期望则大不相同。内核代码被假定是由善意和谨慎的程序员编写的。内核代码被期望是无 bug 的，当然也不包含任何恶意内容。这个假设影响了我们如何分析内核代码。例如，有许多内部内核函数（例如，自旋锁）如果内核代码使用不当会导致严重问题。在检查任何特定的内核代码片段时，我们会希望说服自己它的行为是正确的。然而，我们假设，总的来说，内核代码是正确编写的，并遵循了关于使用内核自身函数和数据结构的所有规则。在硬件层面，RISC-V CPU、RAM、磁盘等被假定按照文档中的描述运行，没有硬件 bug。

当然，在现实生活中，事情并非如此简单。很难阻止聪明的用户代码通过消耗受内核保护的资源（磁盘空间、CPU 时间、进程表槽位等）来使系统无法使用（或导致其恐慌）。编写无 bug 的内核代码或设计无 bug 的硬件通常是不可能的；如果恶意用户代码的编写者意识到内核或硬件的 bug，他们会利用它们。即使在成熟、广泛使用的内核中，例如 Linux，人们也不断发现新的漏洞 [1]。在内核中设计针对其可能存在 bug 的保障措施是值得的：断言、类型检查、堆栈保护页等。最后，用户和内核代码之间的区别有时会变得模糊：一些特权用户级进程可能提供基本服务，并实际上成为操作系统的一部分，在某些操作系统中，特权用户代码可以将新代码插入内核（就像 Linux 的可加载内核模块一样）。

2.8 现实世界

大多数操作系统都采用了进程概念，而且大多数进程看起来都与 xv6 的相似。然而，现代操作系统支持一个进程内的多个线程，以允许单个进程利用多个 CPU。在一个进程中支持多个线程涉及到 xv6 所没有的大量机制，通常包括接口的改变（例如，Linux 的clone，fork的一个变体），以控制线程共享进程的哪些方面。

2.9 练习

1. 向 xv6 添加一个系统调用，返回可用的空闲内存量。

文件	描述
bio.c	文件系统的磁盘块缓存。
console.c	连接到用户键盘和屏幕。
entry.S	最开始的引导指令。
exec.c	exec() 系统调用。
file.c	文件描述符支持。
fs.c	文件系统。
kalloc.c	物理页分配器。
kernelvec.S	处理来自内核的陷阱。
log.c	文件系统日志和崩溃恢复。
main.c	在引导期间控制其他模块的初始化。
pipe.c	管道。
plic.c	RISC-V 中断控制器。
printf.c	向控制台格式化输出。
proc.c	进程和调度。
sleeplock.c	放弃 CPU 的锁。
spinlock.c	不放弃 CPU 的锁。
start.c	早期的机器模式引导代码。
string.c	C 字符串和字节数组库。
swtch.S	线程切换。
syscall.c	将系统调用分派给处理函数。
sysfile.c	文件相关的系统调用。
sysproc.c	进程相关的系统调用。
trampoline.S	用于在用户和内核之间切换的汇编代码。
trap.c	用于处理陷阱和中断并从中返回的 C 代码。
uart.c	串行端口控制台设备驱动程序。
virtio_disk.c	磁盘设备驱动程序。
vm.c	管理页表和地址空间。

Figure 2.2: Xv6 内核源文件。

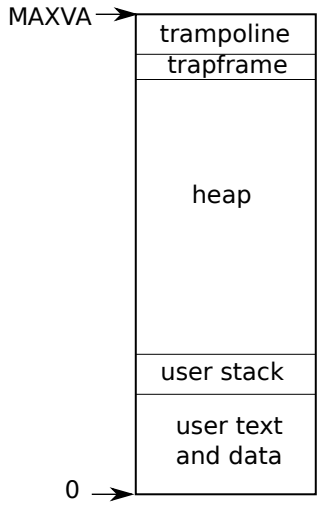

Figure 2.3: 进程虚拟地址空间的布局

Chapter 3

页表

页表是操作系统为每个进程提供其自己的私有地址空间和内存的最流行机制。页表决定了内存地址的含义，以及可以访问物理内存的哪些部分。它们允许 xv6 隔离不同进程的地址空间，并将它们复用到单个物理内存上。页表是一种流行的设计，因为它们提供了一个间接层，允许操作系统执行许多技巧。Xv6 执行了一些技巧：在多个地址空间中映射相同的内存（一个蹦床页），并用一个未映射的页面保护内核和用户栈。本章的其余部分解释了 RISC-V 硬件提供的页表以及 xv6 如何使用它们。

3.1 分页硬件

提醒一下，RISC-V 指令（用户和内核）操作的是虚拟地址。机器的 RAM 或物理内存，是使用物理地址索引的。RISC-V 页表硬件通过将每个虚拟地址映射到物理地址来连接这两种地址。

Xv6 在 Sv39 RISC-V 上运行，这意味着 64 位虚拟地址中只有低 39 位被使用；高 25 位未使用。在这种 Sv39 配置中，RISC-V 页表在逻辑上是一个包含 2^{27} (134,217,728) 个页表条目 (PTE) 的数组。每个 PTE 包含一个 44 位的物理页号 (PPN) 和一些标志。分页硬件通过使用 39 位中的高 27 位来索引页表以查找 PTE，并创建一个 56 位的物理地址，其高 44 位来自 PTE 中的 PPN，低 12 位从原始虚拟地址复制而来，从而转换虚拟地址。图 3.1 展示了这一过程，其中页表的逻辑视图是一个简单的 PTE 数组（更完整的故事请参见图 3.2）。页表使操作系统能够以 4096 (2^{12}) 字节的对齐块（称为页）为粒度控制虚拟到物理地址的转换。

在 Sv39 RISC-V 中，虚拟地址的高 25 位不用于转换。物理地址也有增长空间：PTE 格式中有空间让物理页号再增长 10 位。RISC-V 的设计者基于技术预测选择了这些数字。2^{39} 字节是 512 GB，这对于在 RISC-V 计算机上运行的应用程序来说应该是足够的地址空间。2^{56} 的物理内存空间足以在近期内容纳许多 I/O 设备和 RAM 芯片。如果需要更多，RISC-V 设计者定义了具有 48 位虚拟地址的 Sv48 [3]。

如图 3.2 所示，RISC-V CPU 分三步将虚拟地址转换为物理地址。页表以三级树的形式存储在物理内存中。树的根是一个 4096 字节的页表页，包含 512 个 PTE，这些 PTE 包含树中下一级页表页的物理地址。这些页中的每一个都包含 512 个 PTE，用于树的最后一级。分页硬件使用 27 位中的高 9 位在根页表页中选择一个 PTE，中间 9 位在树的下一级

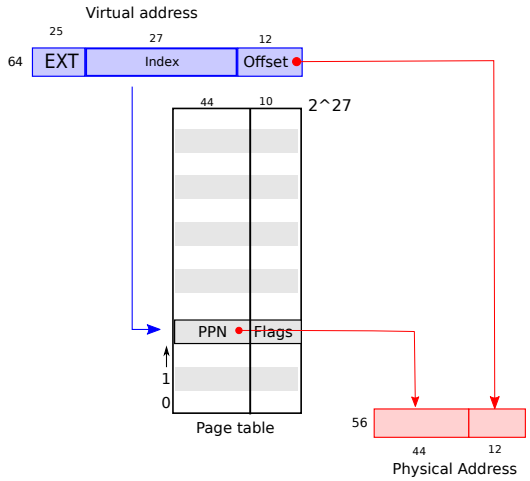

Figure 3.1: RISC-V 虚拟和物理地址，带有一个简化的逻辑页表。

页表页中选择一个 PTE，低 9 位选择最终的 PTE。（在 Sv48 RISC-V 中，页表有四个级别，虚拟地址的第 39 到 47 位索引到顶层。）

如果转换地址所需的三个 PTE 中有任何一个不存在，分页硬件会引发一个缺页异常，由内核来处理该异常（参见第 4章）。

图 3.2 的三级结构与图 3.1 的单级设计相比，提供了一种内存高效的记录 PTE 的方式。在大量虚拟地址没有映射的常见情况下，三级结构可以省略整个页目录。例如，如果一个应用程序只使用从地址零开始的几个页面，那么顶级页目录的条目 1 到 511 都是无效的，内核就不必为这 511 个中间页目录分配页面。此外，内核也不必为这 511 个中间页目录的底层页目录分配页面。因此，在这个例子中，三级设计节省了 511 个中间页目录的页面和 511×512 个底层页目录的页面。

尽管 CPU 在硬件中遍历三级结构作为执行加载或存储指令的一部分，但三级的一个潜在缺点是 CPU 必须从内存加载三个 PTE 来执行加载/存储指令中虚拟地址到物理地址的转换。为了避免从物理内存加载 PTE 的成本，RISC-V CPU 在一个翻译后备缓冲区 (TLB) 中缓存页表条目。

每个 PTE 包含一些标志位，告诉分页硬件关联的虚拟地址允许如何使用。PTE_V 指示 PTE 是否存在：如果未设置，引用该页会导致异常（即不允许）。PTE_R 控制指令是否允许读取该页。PTE_W 控制指令是否允许写入该页。PTE_X 控制 CPU 是否可以将该页的内容解释为指令并执行它们。PTE_U 控制用户模式下的指令是否允许访问该页；如果 PTE_U 未设置，则该 PTE 只能在 supervisor 模式下使用。图 3.2 展示了这一切是如何工作的。标志和所有其他与分页硬件相关的结构都在 (kernel/riscv.h) 中定义。

为了告诉 CPU 使用一个页表，内核必须将根页表页的物理地址写入 satp 寄存器。CPU 将使用其自己的 satp 指向的页表来转换后续指令生成的所有地址。每个 CPU 都有自己的 satp，这样不同的 CPU 就可以运行不同的进程，每个进程都有由其自己的页表描述的私有地址空间。

从内核的角度来看，页表是存储在内存中的数据，内核使用类似于任何树形数据结构的代码来创建和修改页表。

关于本书中使用的一些术语的说明。物理内存指的是 RAM 中的存储单元。物理内存的

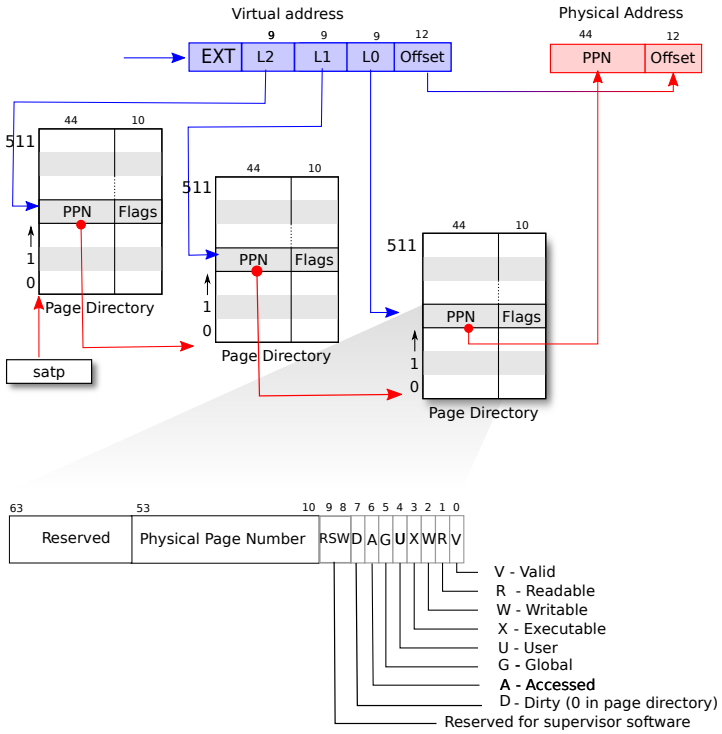

Figure 3.2: RISC-V 地址转换细节。

一个字节有一个地址，称为物理地址。解引用地址的指令（如加载、存储、跳转和函数调用）只使用虚拟地址，分页硬件将其转换为物理地址，然后发送到 RAM 硬件以读取或写入存储。一个地址空间是在给定页表中有效的一组虚拟地址；每个 xv6 进程都有一个独立的的用户地址空间，xv6 内核也有自己的地址空间。用户内存指的是进程的用户地址空间加上页表允许进程访问的物理内存。虚拟内存指的是与管理页表和使用它们实现隔离等目标相关的思想和技术。

3.2 内核地址空间

Xv6 为每个进程维护一个页表，描述每个进程的用户地址空间，外加一个描述内核地址空间的单一页表。内核配置其地址空间的布局，以便在可预测的虚拟地址上访问物理内存和各种硬件资源。图 3.3 展示了这种布局如何将内核虚拟地址映射到物理地址。文件 (kernel/memlayout.h) 声明了 xv6 内核内存布局的常量。

QEMU 模拟了一台计算机，其 RAM（物理内存）从物理地址 0x80000000 开始，至少持续到 0x88000000，xv6 称之为 PHYSTOP。QEMU 模拟还包括 I/O 设备，如磁盘接口。QEMU 将设备接口作为内存映射的控制寄存器暴露给软件，这些寄存器位于物理地址空间中 0x80000000 以下。内核可以通过读/写这些特殊的物理地址与设备交互；这些读写操作是与设备硬件通信，而不是与 RAM 通信。第 4章解释了 xv6 如何与设备交互。

内核使用“直接映射”来访问 RAM 和内存映射的设备寄存器；也就是说，将资源映射到与物理地址相等的虚拟地址。例如，内核本身位于虚拟地址空间和物理内存中的

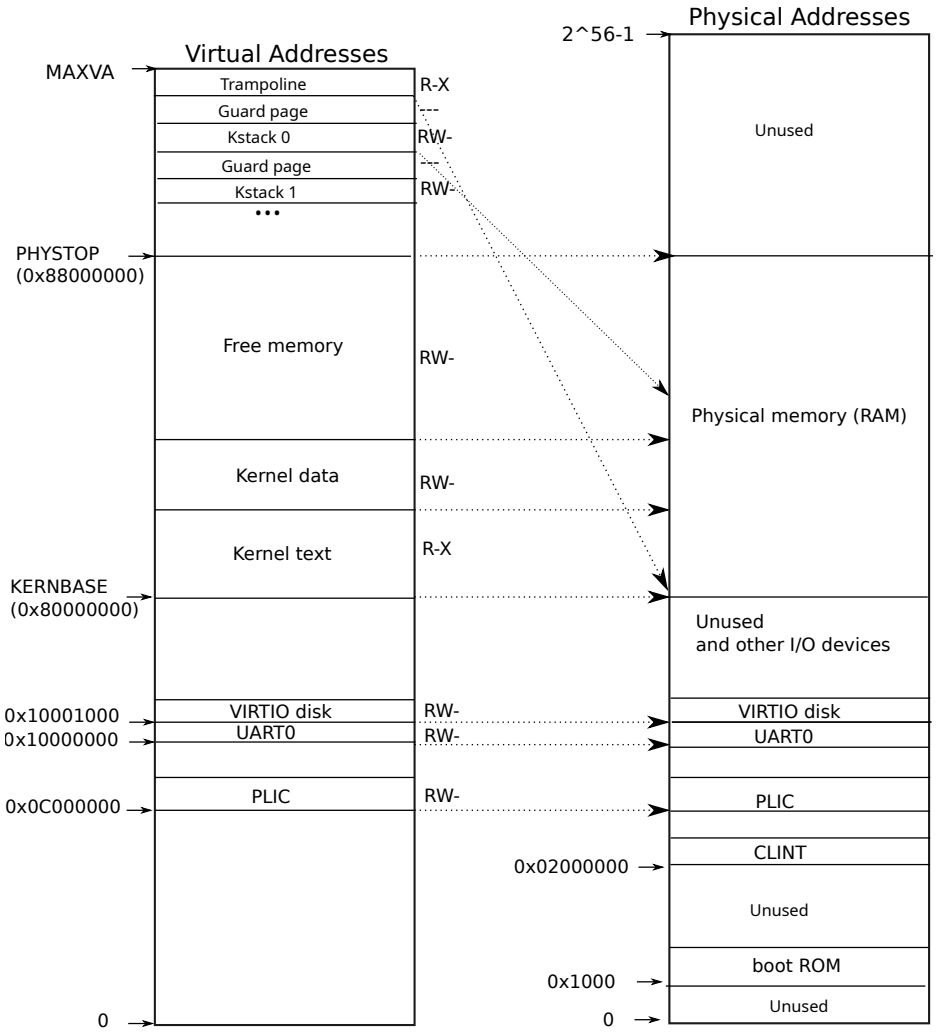

Figure 3.3: 左侧是 xv6 的内核地址空间。RWX 指的是 PTE 的读、写和执行权限。右侧是 xv6 期望看到的 RISC-V 物理地址空间。

KERNBASE=0x80000000。直接映射简化了读写物理内存的内核代码。例如，当 fork 为子进程分配用户内存时，分配器返回该内存的物理地址；fork 在将父进程的用户内存复制给子进程时，直接使用该地址作为虚拟地址。

有几个内核虚拟地址不是直接映射的：

- 蹦床页。它被映射在虚拟地址空间的顶部；用户页表也有这个相同的映射。第 4章讨论了蹦床页的作用，但我们在这里看到了页表的一个有趣用例；一个物理页（包含蹦床代码）在内核的虚拟地址空间中被映射了两次：一次在虚拟地址空间的顶部，一次是直接映射。

- 内核栈页。每个进程都有自己的内核栈，它被映射在高地址，以便 xv6 可以在其下方留下一个未映射的保护页。保护页的 PTE 是无效的（即 PTE_V 未设置），因此

如果内核溢出一个内核栈，它很可能会导致异常，内核会恐慌。没有保护页，溢出的栈会覆盖其他内核内存，导致不正确的操作。恐慌崩溃是更可取的。

虽然内核通过高内存映射使用其栈，但它们也可以通过直接映射的地址被内核访问。另一种设计可能只有直接映射，并在直接映射的地址上使用栈。然而，在这种安排中，提供保护页将涉及取消映射那些否则会引用物理内存的虚拟地址，而这些物理内存之后将难以使用。

内核使用权限 PTE_R 和 PTE_X 映射蹦床和内核文本的页面。内核从这些页面读取和执行指令。内核使用权限 PTE_R 和 PTE_W 映射其他页面，以便它可以读写这些页面中的内存。保护页的映射是无效的。

3.3 代码：创建地址空间

大多数用于操作地址空间和页表的 xv6 代码位于 vm.c (kernel/vm.c:1) 中。核心数据结构是 pagetable_t，它实际上是一个指向 RISC-V 根页表页的指针；一个 pagetable_t 可以是内核页表，也可以是每个进程的页表之一。核心函数是 walk，它查找虚拟地址的 PTE，和 mappages，它为新映射安装 PTE。以 kvm 开头的函数操作内核页表；以 uvm 开头的函数操作用户页表；其他函数则两者都用。copyout 和 copyin 将数据复制到或从来自由系统调用参数提供的用户虚拟地址；它们在 vm.c 中，因为它们需要显式地转换这些地址以找到相应的物理内存。

main 调用 kvminithart (kernel/vm.c:62) 来安装内核页表。它将根页表页的物理地址写入 satp 寄存器。此后，CPU 将使用内核页表转换地址。由于内核使用直接映射，下一条指令的现在的虚拟地址将映射到正确的物理内存地址。

每个 RISC-V CPU 都在一个翻译后备缓冲区 (TLB) 中缓存页表条目，当 xv6 更改页表时，它必须告诉 CPU 使相应的缓存 TLB 条目无效。如果不这样做，那么在稍后的某个时间点，TLB 可能会使用一个旧的缓存映射，指向一个在此期间已分配给另一个进程的物理页，结果，一个进程可能会在另一个进程的内存上乱写。RISC-V 有一个指令 sfence.vma，可以刷新当前 CPU 的 TLB。Xv6 在 kvminithart 中重新加载 satp 寄存器后，以及在切换到用户页表返回用户空间之前的蹦床代码中执行 sfence.vma (kernel/trampoline.S:89)。

在更改 satp 之前，也有必要发出 sfence.vma，以等待所有未完成的加载和存储完成。这种等待确保了对页表的先前更新已经完成，并确保了先前的加载和存储使用旧的页表，而不是新的页表。

为了避免刷新完整的 TLB，RISC-V CPU 可能支持地址空间标识符 (ASID) [3]。然后，内核可以只刷新特定地址空间的 TLB 条目。Xv6 不使用此功能。

3.4 物理内存分配

内核必须在运行时为页表、用户内存、内核栈和管道缓冲区分配和释放物理内存。

Xv6 使用内核末尾和 PHYSTOP 之间的物理内存进行运行时分配。它一次分配和释放整个 4096 字节的页面。它通过在页面本身中穿插一个链表来跟踪哪些页面是空闲的。分配包括从链表中删除一个页面；释放包括将被释放的页面添加到列表中。

3.5 代码：物理内存分配器

函数 main 调用 kinit 来初始化分配器 (kernel/kalloc.c:27)。kinit 初始化空闲列表以容纳内核末尾和 PHYSTOP 之间的每一页。Xv6 应该通过解析硬件提供的配置信息来确定有多少物理内存可用。相反，xv6 假设机器有 128 兆字节的 RAM。kinit 调用 freerange，通过对每页调用 kfree 将内存添加到空闲列表中。一个 PTE 只能引用一个在 4096 字节边界上对齐的物理地址（是 4096 的倍数），所以 freerange 使用 PGROUNDUP 来确保它只释放对齐的物理地址。分配器开始时没有内存；这些对 kfree 的调用给了它一些来管理。

分配器有时将地址视为整数以对其执行算术运算（例如，在 freerange 中遍历所有页面），有时使用地址作为指针来读写内存（例如，操作存储在每个页面中的 run 结构）；这种地址的双重用途是分配器代码充满 C 类型转换的主要原因。

函数 kfree (kernel/kalloc.c:47) 首先将被释放的内存中的每个字节设置为值 1。这将导致在释放后使用内存的代码（使用“悬空引用”）读取垃圾而不是旧的有效内容；希望这会使此类代码更快地崩溃。然后 kfree 将页面前置到空闲列表中：它将 pa 转换为指向 struct run 的指针，将旧的空闲列表的开头记录在 r->next 中，并将空闲列表设置为 r。kalloc 删除并返回空闲列表中的第一个元素。

3.6 进程地址空间

每个进程都有自己的页表，当 xv6 在进程之间切换时，它也会更改页表。图 3.4 比图 2.3 更详细地显示了进程的地址空间。进程的用户内存从虚拟地址零开始，可以增长到 MAXVA (kernel/riscv.h:379)，原则上允许一个进程寻址 256 GB 的内存。

进程的地址空间由包含程序文本的页面（xv6 使用 PTE_R、PTE_X 和 PTE_U 权限映射）、包含程序预初始化数据的页面、一个用于栈的页面和用于堆的页面组成。Xv6 使用 PTE_R、PTE_W 和 PTE_U 权限映射数据、栈和堆。

在用户地址空间内使用权限是加固用户进程的常用技术。如果文本是用 PTE_W 映射的，那么进程可能会意外地修改自己的程序；例如，一个编程错误可能导致程序写入一个空指针，修改地址 0 处的指令，然后继续运行，可能会造成更大的破坏。为了立即检测到此类错误，xv6 映射文本时不带 PTE_W；如果程序意外尝试存储到地址 0，硬件将拒绝执行该存储并引发一个缺页（见第 4.6节）。然后内核杀死该进程并打印一条信息性消息，以便开发人员可以追查问题。

同样，通过不带 PTE_X 映射数据，用户程序不能意外地跳转到程序数据中的地址并从该地址开始执行。

在现实世界中，通过仔细设置权限来加固进程也有助于防御安全攻击。攻击者可能会向程序（例如，Web 服务器）提供精心构造的输入，以触发程序中的一个错误，希望将该错误转化为一个漏洞利用 [14]。仔细设置权限和其他技术，例如随机化用户地址空间的布局，使此类攻击更加困难。

栈是一个单独的页面，并显示了由 exec 创建的初始内容。包含命令行参数的字符串，以及指向它们的指针数组，位于栈的顶端。紧随其后的是允许程序从 main 开始的值，就好像函数 main(argc, argv) 刚刚被调用一样。

为了检测用户栈溢出分配的栈内存，xv6 在栈的正下方放置了一个不可访问的保护页，方法是清除 PTE_U 标志。如果用户栈溢出并且进程试图使用栈下方的地址，硬件将生成

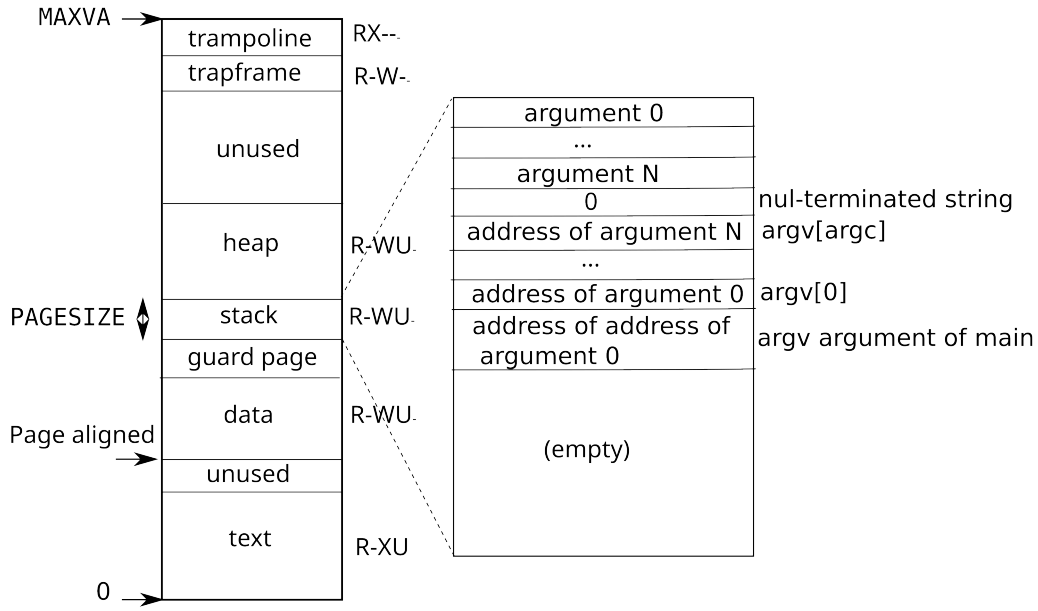

Figure 3.4: 一个进程的用户地址空间，及其初始栈。

一个缺页异常，因为保护页对于在用户模式下运行的程序是不可访问的。一个现实世界的操作系统可能会在用户栈溢出时自动分配更多的内存。

当一个进程向 xv6 请求更多用户内存时，xv6 会增长进程的堆。Xv6 首先使用 kalloc 分配物理页面。然后它向进程的页表添加指向新物理页面的 PTE。Xv6 在这些 PTE 中设置 PTE_W、PTE_R、PTE_U 和 PTE_V 标志。大多数进程不使用整个用户地址空间；xv6 在未使用的 PTE 中将 PTE_V 保持清除。

我们在这里看到了一些页表使用的好例子。首先，不同进程的页表将用户地址转换为不同的物理内存页面，因此每个进程都有私有的用户内存。其次，每个进程都看到其内存具有从零开始的连续虚拟地址，而进程的物理内存可以是不连续的。第三，内核在用户地址空间的顶部映射一个带有蹦床代码的页面（不带 PTE_U），因此一个物理内存页面出现在所有地址空间中，但只能由内核使用。

3.7 **代码：**sbrk

Xv6 不仅使用进程的页表来告诉硬件如何映射用户虚拟地址，而且还将其作为分配给该进程的物理内存页面的唯一记录。这就是为什么释放用户内存（在 uvmunmap 中）需要检查用户页表的原因。

3.8 代码：exec

第一步是快速检查文件是否可能包含 ELF 二进制文件。一个 ELF 二进制文件以四字节的“魔数” 0x7F、‘E’、‘L’、‘F’ 或 ELF_MAGIC (kernel/elf.h:3) 开头。如果 ELF 头有正确的魔数，exec 就假定该二进制文件格式正确。

/init 的程序段头，即用 exec 创建的第一个用户程序，如下所示：

```
# objdump -p user/_init

user/_init:     file format elf64-little

Program Header:
0x70000003 off    0x0000000000006bb0 vaddr 0x0000000000000000
                                     paddr 0x0000000000000000 align 2**0
         filesz 0x000000000000004a memsz 0x0000000000000000 flags r--
    LOAD off    0x0000000000001000 vaddr 0x0000000000000000
                                     paddr 0x0000000000000000 align 2**12
         filesz 0x0000000000001000 memsz 0x0000000000001000 flags r-x
    LOAD off    0x0000000000002000 vaddr 0x0000000000001000
                                     paddr 0x0000000000001000 align 2**12
         filesz 0x0000000000000010 memsz 0x0000000000000030 flags rw-
   STACK off    0x0000000000000000 vaddr 0x0000000000000000
                                     paddr 0x0000000000000000 align 2**4
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
```

我们看到文本段应该加载到内存中的虚拟地址 0x0（没有写权限），其内容来自文件中的偏移量 0x1000。我们还看到数据应该加载到地址 0x1000，这是一个页边界，并且没有执行权限。

程序段头的 filesz 可能小于 memsz，这表明它们之间的差距应该用零填充（对于 C 全局变量），而不是从文件中读取。对于 /init，数据 filesz 是 0x10 字节，memsz 是 0x30 字节，因此 uvmalloc 分配了足够的物理内存来容纳 0x30 字节，但只从文件 /init 中读取 0x10 字节。

现在 exec 分配并初始化用户栈。它只分配一个栈页。exec 一次一个地将参数字符串复制到栈顶，并将指向它们的指针记录在 ustack 中。它在将传递给 main 的 argv 列表的末尾放置一个空指针。值 argc 和 argv 通过系统调用返回路径传递给 main：argc 通过系统调用返回值传递，该返回值进入 a0，argv 通过进程的陷阱帧的 a1 条目传递。

exec 在栈页的正下方放置一个不可访问的页面，这样试图使用多于一个页面的程序就会出错。这个不可访问的页面也允许 exec 处理过大的参数；在这种情况下，exec 用来将参数复制到栈的 copyout (kernel/vm.c:359) 函数会注意到目标页面不可访问，并返回 -1。

exec 将字节从 ELF 文件加载到由 ELF 文件指定的地址的内存中。用户或进程可以将任何他们想要的地址放入 ELF 文件中。因此 exec 是有风险的，因为 ELF 文件中的地址可能会意外地或故意地引用内核。对于一个不警惕的内核来说，后果可能从崩溃到恶意颠覆内核的隔离机制（即安全漏洞利用）。Xv6 执行了许多检查来避免这些风险。例如 if (ph.vaddr + ph.memsz < ph.vaddr) 检查和是否溢出一个 64 位整数。危险在于用户可以构造一个 ELF 二进制文件，其中 ph.vaddr 指向一个用户选择的地址，而 ph.memsz 足够大，以至于和溢出到 0x1000，这看起来像一个有效值。在旧版本的 xv6 中，用户地址空间也包

含内核（但在用户模式下不可读/写），用户可以选择一个对应于内核内存的地址，从而将数据从 ELF 二进制文件复制到内核中。在 RISC-V 版本的 xv6 中，这不会发生，因为内核有自己独立的页表；loadseg 加载到进程的页表中，而不是内核的页表中。

内核开发人员很容易忽略一个关键的检查，而现实世界的内核有很长的历史，都存在着被用户程序利用以获取内核权限的缺失检查。xv6 很可能没有完全验证提供给内核的用户级数据，恶意用户程序可能会利用这一点来规避 xv6 的隔离。

3.9 现实世界

像大多数操作系统一样，xv6 使用分页硬件进行内存保护和映射。大多数操作系统通过结合分页和缺页异常，对分页的使用比 xv6 复杂得多，我们将在第 4章中讨论。

Xv6 因为内核使用虚拟地址和物理地址之间的直接映射，以及它假设在地址 0x80000000 处有物理 RAM（内核期望被加载到那里）而得以简化。这在 QEMU 中可行，但在真实硬件上却是个坏主意；真实硬件将 RAM 和设备放置在不可预测的物理地址，因此（例如）在 0x80000000 可能没有 RAM，而 xv6 期望能够存储内核。更严肃的内核设计利用页表将任意硬件物理内存布局转换为可预测的内核虚拟地址布局。

RISC-V 支持物理地址级别的保护，但 xv6 没有使用该功能。

在拥有大量内存的机器上，使用 RISC-V 对“超级页”的支持可能是有意义的。当物理内存较小时，小页面是有意义的，以便以细粒度进行分配和页面换出到磁盘。例如，如果一个程序只使用 8 千字节的内存，给它一个完整的 4 兆字节的超级页物理内存是浪费的。在拥有大量 RAM 的机器上，较大的页面更有意义，并且可以减少页表操作的开销。

xv6 内核缺乏一个类似 malloc 的分配器，可以为小对象提供内存，这使得内核无法使用需要动态分配的复杂数据结构。一个更复杂的内核可能会分配许多不同大小的小块，而不是（像在 xv6 中那样）只分配 4096 字节的块；一个真正的内核分配器需要处理小分配和大的分配。

内存分配是一个常年热门的话题，基本问题是有效利用有限的内存和为未知的未来请求做准备 [9]。如今，人们更关心速度而不是空间效率。

3.10 练习

1. 解析 RISC-V 的设备树以找出计算机拥有的物理内存量。

2. 编写一个用户程序，通过调用 sbrk(1) 将其地址空间增加一个字节。运行该程序，并在调用 sbrk 之前和之后调查程序的页表。内核分配了多少空间？新内存的 PTE 包含什么?

3. 修改 xv6 以便为内核使用超级页。

4. Unix 的 exec 实现传统上包括对 shell 脚本的特殊处理。如果待执行文件的开头是文本 #!，那么第一行就被视为运行以解释该文件的程序。例如，如果调用 exec 来运行 myprog arg1，并且 myprog 的第一行是 #!/interp，那么 exec 就会用命令行 /interp myprog arg1 来运行 /interp。在 xv6 中实现对此约定的支持。

5. 为内核实现地址空间布局随机化。

Chapter 4

陷阱和系统调用

有三种事件会导致 CPU 搁置正常的指令执行，并强制将控制权转移到处理该事件的特殊代码。一种情况是系统调用，当用户程序执行 ecall 指令请求内核为其执行某项操作时。另一种情况是异常：一条指令（用户或内核）执行了非法操作，例如除以零或使用无效的虚拟地址。第三种情况是设备中断，当设备发出需要注意的信号时，例如当磁盘硬件完成读或写请求时。

本书使用陷阱作为这些情况的通用术语。通常，在陷阱发生时正在执行的任何代码稍后都需要恢复，并且不应该意识到发生了任何特殊情况。也就是说，我们通常希望陷阱是透明的；这对于设备中断尤其重要，因为被中断的代码通常不期望发生中断。通常的顺序是，陷阱强制将控制权转移到内核；内核保存寄存器和其他状态，以便可以恢复执行；内核执行适当的处理程序代码（例如，系统调用实现或设备驱动程序）；内核恢复保存的状态并从陷阱中返回；原始代码从中断处继续执行。

Xv6 在内核中处理所有陷阱；陷阱不会传递给用户代码。在内核中处理陷阱对于系统调用来说是很自然的。对于中断来说，这是有道理的，因为隔离要求只允许内核使用设备，并且因为内核是多个进程之间共享设备的便捷机制。对于异常来说，这也是有道理的，因为 xv6 对所有来自用户空间的异常的响应都是终止有问题的程序。

Xv6 的陷阱处理分为四个阶段：RISC-V CPU 采取的硬件操作、为内核 C 代码准备的一些汇编指令、一个决定如何处理陷阱的 C 函数，以及系统调用或设备驱动程序服务例程。虽然三种陷阱类型之间的共性表明内核可以用单个代码路径处理所有陷阱，但事实证明，为两种不同情况分别设置代码会更方便：来自用户空间的陷阱和来自内核空间的陷阱。处理陷阱的内核代码（汇编或 C）通常被称为处理程序；第一个处理程序的指令通常用汇编（而不是 C）编写，有时被称为向量。

4.1 RISC-V 陷阱机制

每个 RISC-V CPU 都有一组控制寄存器，内核通过写入这些寄存器来告诉 CPU 如何处理陷阱，内核也可以读取这些寄存器来了解已发生的陷阱。RISC-V 文档包含了完整的故事 [3]。riscv.h (kernel/riscv.h:1) 包含了 xv6 使用的定义。以下是最重要寄存器的纲要：

- stvec: 内核在此处写入其陷阱处理程序的地址；RISC-V 跳转到 stvec 中的地址来处理陷阱。

- sepc: 发生陷阱时，RISC-V 在此处保存程序计数器（因为 pc 随后会被 stvec 中的值覆盖）。sret（从陷阱返回）指令将 sepc 复制到 pc。内核可以写入 sepc 来控制 sret 的去向。

- scause: RISC-V 在此处放置一个数字，描述陷阱的原因。

- sscratch: 陷阱处理程序代码使用 sscratch 来帮助其在保存用户寄存器之前避免覆盖它们。

- sstatus: sstatus 中的 SIE 位控制是否启用设备中断。如果内核清除 SIE，RISC-V 将延迟设备中断，直到内核设置 SIE。SPP 位指示陷阱是来自用户模式还是监督者模式，并控制 sret 返回到哪种模式。

上述寄存器与在监督者模式下处理的陷阱有关，不能在用户模式下读取或写入。

多核芯片上的每个 CPU 都有自己的一组这些寄存器，并且在任何给定时间可能有多个 CPU 在处理陷阱。

当需要强制陷阱时，RISC-V 硬件对所有陷阱类型执行以下操作：

1. 如果陷阱是设备中断，并且 sstatus 的 SIE 位被清除，则不执行以下任何操作。

2. 通过清除 sstatus 中的 SIE 位来禁用中断。

3. 将 pc 复制到 sepc。

4. 在 sstatus 的 SPP 位中保存当前模式（用户或监督者）。

5. 设置 scause 以反映陷阱的原因。

6. 将模式设置为监督者。

7. 将 stvec 复制到 pc。

8. 在新的 pc 处开始执行。

请注意，CPU 不会切换到内核页表，不会切换到内核中的堆栈，也不会保存除 pc 之外的任何寄存器。内核软件必须执行这些任务。CPU 在陷阱期间只做最少的工作的一个原因是为软件提供灵活性；例如，一些操作系统在某些情况下省略页表切换以提高陷阱性能。

值得思考的是，上面列出的步骤中是否有任何一个可以省略，也许是为了追求更快的陷阱。虽然在某些情况下更简单的序列可以工作，但通常省略许多步骤是危险的。例如，假设 CPU 没有切换程序计数器。那么来自用户空间的陷阱可以在仍在运行用户指令的情况下切换到监督者模式。这些用户指令可能会破坏用户/内核隔离，例如通过修改 satp 寄存器以指向一个允许访问所有物理内存的页表。因此，CPU 切换到内核指定的指令地址，即 stvec，是很重要的。

4.2 来自用户空间的陷阱

Xv6 根据陷阱是在内核中执行还是在用户代码中执行来区别处理。以下是来自用户代码的陷阱的故事；第 4.5节描述了来自内核代码的陷阱。

xv6 陷阱处理设计的一个主要限制是 RISC-V 硬件在强制陷阱时不切换页表。这意味着 stvec 中的陷阱处理程序地址必须在用户页表中具有有效映射，因为在陷阱处理代码开始执行时，该页表是有效的。此外，xv6 的陷阱处理代码需要切换到内核页表；为了能够在切换后继续执行，内核页表也必须具有对 stvec 指向的处理程序的映射。

Xv6 使用蹦床页来满足这些要求。蹦床页包含 uservec，即 stvec 指向的 xv6 陷阱处理代码。蹦床页在每个进程的页表中映射在地址 TRAMPOLINE，该地址位于虚拟地址空间的顶部，以便它位于程序自己使用的内存之上。蹦床页也映射在内核页表中的地址 TRAMPOLINE。参见图 2.3和图 3.3。因为蹦床页映射在用户页表中，所以陷阱可以在监督者模式下在那里开始执行。因为蹦床页在内核地址空间中映射在相同的地址，所以陷阱处理程序可以在切换到内核页表后继续执行。

uservec 陷阱处理程序的代码在 trampoline.S (kernel/trampoline.S:22) 中。当 uservec 启动时，所有 32 个寄存器都包含被中断的用户代码所拥有的值。这 32 个值需要保存在内存中的某个地方，以便稍后内核可以在返回用户空间之前恢复它们。存储到内存需要使用一个寄存器来保存地址，但此时没有可用的通用寄存器！幸运的是，RISC-V 以 sscratch 寄存器的形式提供了帮助。uservec 开头的 csrw 指令将 a0 保存在 sscratch 中。现在 uservec 有一个寄存器（a0）可以使用。

uservec 的下一个任务是保存 32 个用户寄存器。内核为每个进程分配一页内存用于一个 trapframe 结构，该结构（除其他外）有空间保存 32 个用户寄存器 (kernel/proc.h:43)。因为 satp 仍然引用用户页表，所以 uservec 需要将陷阱帧映射在用户地址空间中。Xv6 将每个进程的陷阱帧映射在该进程的用户页表中的虚拟地址 TRAPFRAME；TRAPFRAME 就在 TRAMPOLINE 下面。进程的 p->trapframe 也指向陷阱帧，尽管是在其物理地址上，以便内核可以通过内核页表使用它。

因此，uservec 将地址 TRAPFRAME 加载到 a0 中，并在那里保存所有用户寄存器，包括从 sscratch 读回的用户 a0。

trapframe 包含当前进程的内核堆栈地址、当前 CPU 的 hartid、usertrap 函数的地址以及内核页表的地址。uservec 检索这些值，将 satp 切换到内核页表，然后跳转到 usertrap。

usertrap 的工作是确定陷阱的原因，处理它，然后返回 (kernel/trap.c:37)。它首先更改 stvec，以便在内核中的陷阱将由 kernelvec 而不是 uservec 处理。它保存 sepc 寄存器（保存的用户程序计数器），因为 usertrap 可能会调用 yield 切换到另一个进程的内核线程，并且该进程可能会返回到用户空间，在此过程中它将修改 sepc。如果陷阱是系统调用，usertrap 调用 syscall 来处理它；如果是设备中断，则调用 devintr；否则是异常，内核将终止出错的进程。系统调用路径将保存的用户程序计数器加四，因为 RISC-V 在系统调用的情况下，会将程序指针指向 ecall 指令，但用户代码需要在后续指令处恢复执行。在退出时，usertrap 检查进程是否已被终止或应该让出 CPU（如果此陷阱是定时器中断）。

返回用户空间的第一步是调用 usertrapret (kernel/trap.c:90)。此函数设置 RISC-V 控制寄存器，为将来来自用户空间的陷阱做准备：将 stvec 设置为 uservec 并准备 uservec 依赖的陷阱帧字段。usertrapret 将 sepc 设置为先前保存的用户程序计数器。最后，usertrapret 在映射在用户和内核页表中的蹦床页上调用 userret；原因是在 userret 中的汇编代码将切

换页表。

usertrapret 对 userret 的调用在 a0 中传递一个指向进程用户页表的指针 (kernel/trampoline.S:101)。userret 将 satp 切换到进程的用户页表。回想一下，用户页表映射了蹦床页和 TRAPFRAME，但没有映射内核的其他任何内容。在用户和内核页表中以相同虚拟地址映射的蹦床页允许 userret 在更改 satp 后继续执行。从这一点开始，userret 唯一可以使用的数据是寄存器内容和陷阱帧的内容。userret 将 TRAPFRAME 地址加载到 a0 中，通过 a0 从陷阱帧中恢复保存的用户寄存器，恢复保存的用户 a0，并执行 sret 返回到用户空间。

4.3 代码：调用系统调用

第 2章以 initcode.S 调用 exec 系统调用 (user/initcode.S:11) 结束。让我们看看用户调用是如何到达内核中 exec 系统调用实现的。

initcode.S 将 exec 的参数放在寄存器 a0 和 a1 中，并将系统调用号放在 a7 中。系统调用号与 syscalls 数组中的条目匹配，这是一个函数指针表 (kernel/syscall.c:107)。ecall 指令陷入内核并导致 uservec、usertrap，然后是 syscall 执行，如我们上面所见。

当 sys_exec 返回时，syscall 将其返回值记录在 p->trapframe->a0 中。这将导致原始用户空间对 exec() 的调用返回该值，因为 RISC-V 上的 C 调用约定将返回值放在 a0 中。系统调用通常返回负数表示错误，返回零或正数表示成功。如果系统调用号无效，syscall 会打印错误并返回 -1。

4.4 代码：系统调用参数

内核中的系统调用实现需要找到用户代码传递的参数。因为用户代码调用系统调用包装函数，所以参数最初位于 RISC-V C 调用约定放置它们的位置：在寄存器中。内核陷阱代码将用户寄存器保存到当前进程的陷阱帧中，内核代码可以在那里找到它们。内核函数 argint、argaddr 和 argfd 从陷阱帧中检索第 n 个系统调用参数，分别为整数、指针或文件描述符。它们都调用 argraw 来检索适当的已保存用户寄存器 (kernel/syscall.c:34)。

一些系统调用传递指针作为参数，内核必须使用这些指针来读取或写入用户内存。例如，exec 系统调用向内核传递一个指向用户空间中字符串参数的指针数组。这些指针带来了两个挑战。首先，用户程序可能有错误或恶意，可能会向内核传递无效指针或旨在欺骗内核访问内核内存而不是用户内存的指针。其次，xv6 内核页表映射与用户页表映射不同，因此内核不能使用普通指令从用户提供的地址加载或存储。

内核实现了安全地将数据传入和传出用户提供地址的函数。fetchstr 是一个例子 (kernel/syscall.c:25)。诸如 exec 之类的文件系统调用使用 fetchstr 从用户空间检索字符串文件名参数。fetchstr 调用 copyinstr 来完成艰苦的工作。

4.5 来自内核空间的陷阱

Xv6 处理来自内核代码的陷阱的方式与处理来自用户代码的陷阱不同。当进入内核时，usertrap 将 stvec 指向位于 kernelvec (kernel/kernelvec.S:12) 的汇编代码。由于 kernelvec 仅在 xv6 已在内核中时执行，因此 kernelvec 可以依赖于 satp 已设置为内核页表，并且堆栈

指针指向有效的内核堆栈。kernelvec 将所有 32 个寄存器压入堆栈，稍后将从中恢复它们，以便被中断的内核代码可以不受干扰地继续执行。

kernelvec 将寄存器保存在被中断的内核线程的堆栈上，这是有道理的，因为寄存器值属于该线程。如果陷阱导致切换到另一个线程，这一点尤其重要——在这种情况下，陷阱实际上将从新线程的堆栈返回，而被中断线程的已保存寄存器安全地留在其堆栈上。

如果 kerneltrap 是由于定时器中断而被调用的，并且一个进程的内核线程正在运行（而不是调度程序线程），kerneltrap 会调用 yield 以让其他线程有机会运行。在某个时候，其中一个线程会让步，让我们的线程和它的 kerneltrap 再次恢复。第 7章解释了 yield 中发生的事情。

当 kerneltrap 的工作完成时，它需要返回到被陷阱中断的任何代码。因为 yield 可能已经扰乱了 sepc 和 sstatus 中的先前模式，所以 kerneltrap 在启动时保存了它们。它现在恢复这些控制寄存器并返回到 kernelvec (kernel/kernelvec.S:38)。kernelvec 从堆栈中弹出已保存的寄存器并执行 sret，它将 sepc 复制到 pc 并恢复被中断的内核代码。

值得思考的是，如果 kerneltrap 由于定时器中断而调用了 yield，陷阱返回是如何发生的。

Xv6 在一个 CPU 从用户空间进入内核时，将该 CPU 的 stvec 设置为 kernelvec；你可以在 usertrap (kernel/trap.c:29) 中看到这一点。有一段时间窗口，内核已经开始执行但 stvec 仍设置为 uservec，至关重要的是在该窗口期间不能发生设备中断。幸运的是，RISC-V 在开始处理陷阱时总是禁用中断，而 usertrap 在设置 stvec 之后才再次启用它们。

4.6 页错误异常

Xv6 对异常的响应相当无聊：如果异常发生在用户空间，内核会杀死出错的进程。如果异常发生在内核中，内核会 panic。真正的操作系统通常会以更有趣的方式做出响应。

举个例子，许多内核使用页错误来实现写时复制（COW）fork。为了解释写时复制 fork，请考虑第 3章中描述的 xv6 的 fork。fork 导致子进程的初始内存内容与 fork 时父进程的内存内容相同。Xv6 使用 uvmcopy (kernel/vm.c:313) 来实现 fork，它为子进程分配物理内存并将父进程的内存复制到其中。如果子进程和父进程可以共享父进程的物理内存，效率会更高。然而，直接实现这一点是行不通的，因为它会导致父进程和子进程通过对共享堆栈和堆的写入来相互干扰彼此的执行。

父进程和子进程可以通过适当使用页表权限和页错误来安全地共享物理内存。当使用没有映射的虚拟地址，或者映射的 PTE_V 标志被清除，或者映射的权限位（PTE_R、PTE_W、PTE_X、PTE_U）禁止正在尝试的操作时，CPU 会引发页错误异常。RISC-V 区分三种类型的页错误：加载页错误（由加载指令引起）、存储页错误（由存储指令引起）和指令页错误（由获取要执行的指令引起）。scause 寄存器指示页错误的类型，stval 寄存器包含无法转换的地址。

COW fork 的基本计划是父进程和子进程最初共享所有物理页，但每个进程都将它们映射为只读（PTE_W 标志被清除）。父进程和子进程可以从共享的物理内存中读取。如果任何一方写入给定页面，RISC-V CPU 会引发页错误异常。内核的陷阱处理程序通过分配一个新的物理内存页并将出错地址映射到的物理页复制到其中来响应。内核更改出错进程页表中的相关 PTE，以指向副本并允许写入和读取，然后在导致错误的指令处恢复出错进程。因为 PTE 现在允许写入，所以重新执行的指令将不会出现错误地执行。写时复制需

要簿记来帮助决定何时可以释放物理页，因为每个页可以被可变数量的页表引用，具体取决于 fork、页错误、exec 和退出的历史记录。这种簿记允许一个重要的优化：如果一个进程发生存储页错误并且物理页仅从该进程的页表引用，则不需要复制。

写时复制使 fork 更快，因为 fork 不需要复制内存。一些内存稍后在写入时必须被复制，但通常情况下，大部分内存永远不需要被复制。一个常见的例子是 fork 后跟 exec：fork 之后可能会写入几页，但随后子进程的 exec 会释放从父进程继承的大部分内存。写时复制 fork 消除了复制这部分内存的需要。此外，COW fork 是透明的：应用程序无需修改即可受益。

页表和页错误的结合开启了除 COW fork 之外的各种有趣的可能性。另一个广泛使用的功能称为惰性分配，它有两个部分。首先，当应用程序通过调用 sbrk 请求更多内存时，内核会记录大小的增加，但不会分配物理内存，也不会为新的虚拟地址范围创建 PTE。其次，在这些新地址之一上发生页错误时，内核会分配一个物理内存页并将其映射到页表中。与 COW fork 一样，内核可以对应用程序透明地实现惰性分配。

由于应用程序通常会请求比它们需要的更多的内存，因此惰性分配是一个胜利：对于应用程序从不使用的页面，内核根本不需要做任何工作。此外，如果应用程序请求大幅增加地址空间，那么没有惰性分配的 sbrk 是昂贵的：如果一个应用程序请求一千兆字节的内存，内核必须分配并清零 262,144 个 4096 字节的页面。惰性分配允许将此成本分摊到一段时间内。另一方面，惰性分配会带来页错误的额外开销，这涉及用户/内核转换。操作系统可以通过为每个页错误分配一批连续的页面而不是一个页面，以及通过专门化此类页错误的内核进入/退出代码来降低此成本。

利用页错误的另一个广泛使用的功能是按需分页。在 exec 中，xv6 在启动应用程序之前将应用程序的所有文本和数据加载到内存中。由于应用程序可能很大并且从磁盘读取需要时间，因此这种启动成本对用户来说是显而易见的。为了减少启动时间，现代内核最初不会将可执行文件加载到内存中，而只是创建用户页表，并将所有 PTE 标记为无效。内核启动程序运行；每次程序第一次使用页面时，都会发生页错误，作为响应，内核从磁盘读取页面的内容并将其映射到用户地址空间中。与 COW fork 和惰性分配一样，内核可以对应用程序透明地实现此功能。

计算机上运行的程序可能需要比计算机拥有的 RAM 更多的内存。为了优雅地应对，操作系统可以实现分页到磁盘。其思想是仅在 RAM 中存储一小部分用户页面，其余部分存储在磁盘上的分页区域中。内核将对应于存储在分页区域中（因此不在 RAM 中）的内存的 PTE 标记为无效。如果应用程序尝试使用已“换出”到磁盘的页面之一，应用程序将发生页错误，并且该页面必须被“换入”：内核陷阱处理程序将分配一页物理 RAM，从磁盘将页面读入 RAM，并修改相关的 PTE 以指向 RAM。

如果需要换入一个页面但没有空闲的物理 RAM 会发生什么？在这种情况下，内核必须首先通过将其换出或“驱逐”到磁盘上的分页区域来释放一个物理页面，并将引用该物理页面的 PTE 标记为无效。驱逐是昂贵的，因此如果分页不频繁，其性能最佳：如果应用程序仅使用其内存页面的一个子集，并且这些子集的并集适合 RAM。此属性通常被称为具有良好的引用局部性。与许多虚拟内存技术一样，内核通常以对应用程序透明的方式实现分页到磁盘。

计算机通常在几乎没有或没有“空闲”物理内存的情况下运行，无论硬件提供多少 RAM。例如，云提供商在单台机器上多路复用许多客户，以经济高效地使用其硬件。另一个例子是，用户在少量物理内存的智能手机上运行许多应用程序。在这种情况下，分配一

个页面可能需要首先驱逐一个现有页面。因此，当空闲物理内存稀缺时，分配是昂贵的。

当空闲内存稀缺且程序仅积极使用其分配内存的一小部分时，惰性分配和按需分页尤其有利。这些技术还可以避免在分配或加载但从未使用或在使用前被驱逐的页面上浪费的工作。

结合分页和页错误异常的其他功能包括自动扩展堆栈和内存映射文件，这些文件是程序使用 mmap 系统调用映射到其地址空间的文件，以便程序可以使用加载和存储指令来读写它们。

4.7 现实世界

蹦床和陷阱帧可能看起来过于复杂。一个驱动力是 RISC-V 有意在强制陷阱时尽可能少地做事情，以允许非常快速的陷阱处理的可能性，这被证明是重要的。结果是，内核陷阱处理程序的前几条指令实际上必须在用户环境中执行：用户页表和用户寄存器内容。并且陷阱处理程序最初不知道诸如正在运行的进程的身份或内核页表的地址之类的有用事实。一个解决方案是可能的，因为 RISC-V 提供了受保护的地方，内核可以在进入用户空间之前隐藏信息：sscratch 寄存器和指向内核内存但受缺少 PTE_U 保护的用户页表条目。Xv6 的蹦床和陷阱帧利用了这些 RISC-V 功能。

如果内核内存被映射到每个进程的用户页表中（PTE_U 清除），则可以消除对特殊蹦床页的需求。这也将消除在从用户空间陷入内核时进行页表切换的需要。这反过来又允许内核中的系统调用实现利用当前进程的用户内存被映射的优势，从而允许内核代码直接解引用用户指针。许多操作系统已经使用这些思想来提高效率。Xv6 避免了它们，以减少由于无意中使用用户指针而导致内核中安全漏洞的机会，并减少为确保用户和内核虚拟地址不重叠所需的一些复杂性。

生产操作系统实现了写时复制 fork、惰性分配、按需分页、分页到磁盘、内存映射文件等。此外，生产操作系统会尝试在物理内存的所有区域中存储有用的东西，通常在进程不使用的内存中缓存文件内容。

生产操作系统还向应用程序提供系统调用来管理其地址空间，并通过 mmap、munmap 和 sigaction 系统调用实现自己的页错误处理，以及提供将内存固定到 RAM 的调用（参见 mlock）和建议内核应用程序计划如何使用其内存的调用（参见 madvise）。

4.8 练习

1. 函数 copyin 和 copyinstr 在软件中遍历用户页表。设置内核页表，以便内核映射了用户程序，并且 copyin 和 copyinstr 可以使用 memcpy 将系统调用参数复制到内核空间，依赖硬件来完成页表遍历。
2. 实现惰性内存分配。
3. 实现 COW fork。
4. 有没有办法消除每个用户地址空间中的特殊 TRAPFRAME 页面映射? 例如，是否可以修改 uservec 以简单地将 32 个用户寄存器推送到内核堆栈，或将它们存储在 proc 结构中?

5. xv6 是否可以修改以消除特殊的 TRAMPOLINE 页面映射?

6. 实现 mmap。

Chapter 5

中断和设备驱动程序

驱动程序(driver) 是操作系统中管理特定设备的代码：它配置设备硬件，告诉设备执行操作，处理由此产生的中断，并与可能正在等待设备 I/O 的进程进行交互。驱动程序代码可能很棘手，因为驱动程序与其管理的设备同时执行。此外，驱动程序必须了解设备的硬件接口，而这些接口可能很复杂且文档记录不完善。

需要操作系统关注的设备通常可以配置为产生中断，中断是陷阱的一种类型。内核陷阱处理代码识别设备何时引发中断，并调用驱动程序的中断处理程序；在 xv6 中，此分派发生在 devintr (kernel/trap.c:185) 中。

许多设备驱动程序在两个上下文中执行代码：一个在进程的内核线程中运行的上半部分(top half)，以及一个在中断时执行的下半部分(bottom half)。上半部分通过诸如 read 和 write 等希望设备执行 I/O 的系统调用来调用。此代码可能会要求硬件启动一个操作（例如，要求磁盘读取一个块）；然后代码等待操作完成。最终设备完成操作并引发中断。驱动程序的中断处理程序作为下半部分，确定已完成的操作，在适当时唤醒等待的进程，并告诉硬件开始处理任何等待的下一个操作。

5.1 代码：控制台输入

控制台驱动程序 (kernel/console.c) 是驱动程序结构的简单示例。控制台驱动程序通过连接到 RISC-V 的 UART 串行端口硬件接受由人键入的字符。控制台驱动程序一次累积一行输入，处理诸如退格和 control-u 之类的特殊输入字符。用户进程（例如 shell）使用 read 系统调用从控制台获取输入行。当您在 QEMU 中向 xv6 输入时，您的击键通过 QEMU 的模拟 UART 硬件传递给 xv6。

驱动程序与之通信的 UART 硬件是 QEMU 模拟的 16550 芯片 [13]。在真实的计算机上，16550 将管理连接到终端或其他计算机的 RS232 串行链路。在运行 QEMU 时，它连接到您的键盘和显示器。

一旦被唤醒，consoleread 将在 cons.buf 中观察到完整的一行，将其复制到用户空间，并通过系统调用机制返回到用户空间。

5.2 代码：控制台输出

对连接到控制台的文件描述符的 write 系统调用最终会到达 uartputc (kernel/uart.c:87)。设备驱动程序维护一个输出缓冲区（uart_tx_buf），以便写入进程不必等待 UART 完成发送；相反，uartputc 将每个字符附加到缓冲区，调用 uartstart 来启动设备传输（如果尚未启动），然后返回。uartputc 等待的唯一情况是缓冲区已满。

每当 UART 完成发送一个字节时，它就会产生一个中断。uartintr 调用 uartstart，后者检查设备是否真的完成了发送，并将下一个缓冲的输出字符交给设备。因此，如果一个进程向控制台写入多个字节，通常第一个字节将由 uartputc 对 uartstart 的调用发送，而剩余的缓冲字节将在发送完成中断到达时由来自 uartintr 的 uartstart 调用发送。

需要注意的一个通用模式是通过缓冲和中断将设备活动与进程活动解耦。控制台驱动程序即使在没有进程等待读取输入时也可以处理输入；后续的读取将看到该输入。类似地，进程可以发送输出而不必等待设备。这种解耦可以通过允许进程与设备 I/O 并发执行来提高性能，并且在设备缓慢（如 UART）或需要立即关注（如回显键入的字符）时尤其重要。这个想法有时被称为 I/O 并发性(I/O concurrency)。

5.3 驱动程序中的并发

您可能已经注意到在 consoleread 和 consoleintr 中有对 acquire 的调用。这些调用获取一个锁，以保护控制台驱动程序的数据结构免受并发访问。这里有三个并发危险：两个不同 CPU 上的进程可能同时调用 consoleread；硬件可能会请求一个 CPU 在该 CPU 已经在 consoleread 内部执行时传递一个控制台（实际上是 UART）中断；以及硬件可能会在一个不同的 CPU 上传递一个控制台中断，而 consoleread 正在执行。第 6章解释了如何使用锁来确保这些危险不会导致不正确的结果。

并发在驱动程序中需要小心的另一种方式是，一个进程可能正在等待来自设备的输入，但表示输入到达的中断可能在另一个进程（或根本没有进程）正在运行时到达。因此，中断处理程序不允许考虑它们已中断的进程或代码。例如，中断处理程序不能安全地使用当前进程的页表调用 copyout。中断处理程序通常做相对较少的工作（例如，只是将输入数据复制到缓冲区），并唤醒上半部分代码来完成其余的工作。

5.4 定时器中断

Xv6 使用定时器中断来维护其当前时间的概念，并在计算密集型进程之间进行切换。定时器中断来自连接到每个 RISC-V CPU 的时钟硬件。Xv6 对每个 CPU 的时钟硬件进行编程，以使其定期中断 CPU。

start.c (kernel/start.c:53) 中的代码设置了一些控制位，允许在 supervisor 模式下访问定时器控制寄存器，然后请求第一个定时器中断。time 控制寄存器包含一个硬件以稳定速率递增的计数；这作为当前时间的概念。stimecmp 寄存器包含 CPU 将引发定时器中断的时间；将 stimecmp 设置为 time 的当前值加上 x 将在未来 x 个时间单位后安排一个中断。对于 qemu 的 RISC-V 仿真，1000000 个时间单位大约是十分之一秒。

定时器中断像其他设备中断一样, 通过 usertrap 或 kerneltrap 以及 devintr 到达。定时器中断到达时，scause 的低位设置为 5；trap.c 中的 devintr 检测到这种情况并调用 clockintr (kernel/trap.c:164)。后一个函数递增 ticks，允许内核跟踪时间的流逝。递增只在一个 CPU 上发生，以避免在有多个 CPU 时时间过得更快。clockintr 唤醒任何在 sleep 系统调用中等待的进程，并通过写入 stimecmp 来安排下一个定时器中断。

devintr 对定时器中断返回 2，以向 kerneltrap 或 usertrap 指示它们应该调用 yield，以便 CPU 可以在可运行的进程之间进行多路复用。

内核代码可能被定时器中断中断, 该中断通过 yield 强制进行上下文切换, 这是 usertrap 中的早期代码在启用中断之前小心保存诸如 sepc 之类的状态的部分原因。这些上下文切换也意味着内核代码的编写必须意识到它可能会在没有警告的情况下从一个 CPU 移动到另一个 CPU。

5.5 现实世界

与许多操作系统一样，Xv6 允许在内核中执行时发生中断甚至上下文切换（通过 yield）。这样做的原因是在运行时间较长的复杂系统调用期间保持快速的响应时间。然而，如上所述，允许在内核中中断是一些复杂性的来源；因此，一些操作系统只允许在执行用户代码时中断。

要完全支持典型计算机上的所有设备是一项繁重的工作，因为设备众多，设备功能繁多，而且设备和驱动程序之间的协议可能复杂且文档记录不佳。在许多操作系统中，驱动程序占用的代码比核心内核还多。

UART 驱动程序通过读取 UART 控制寄存器一次一个字节地检索数据；这种模式称为编程 I/O(programmed I/O)，因为软件正在驱动数据移动。编程 I/O 很简单，但速度太慢，无法用于高数据速率。需要高速移动大量数据的设备通常使用直接内存访问 (DMA)(direct memory access)。DMA 设备硬件直接将传入数据写入 RAM，并从 RAM 读取传出数据。现代磁盘和网络设备使用 DMA。DMA 设备的驱动程序会在 RAM 中准备数据，然后使用对控制寄存器的单次写入来告诉设备处理准备好的数据。

当设备在不可预测的时间需要关注, 并且频率不高时, 中断是有意义的。但中断的 CPU 开销很高。因此，高速设备，如网络和磁盘控制器，使用技巧来减少对中断的需求。一种技巧是为一批传入或传出的请求引发单个中断。另一种技巧是让驱动程序完全禁用中断，并定期检查设备以查看是否需要关注。这种技术称为轮询(polling)。如果设备以高速率执行操作，轮询是有意义的，但如果设备大部分时间处于空闲状态，则会浪费 CPU 时间。一些驱动程序根据当前的设备负载动态地在轮询和中断之间切换。

UART 驱动程序首先将传入数据复制到内核中的缓冲区，然后再复制到用户空间。这在低数据速率下是有意义的，但对于非常快速地生成或消耗数据的设备，这种双重复制会显著降低性能。一些操作系统能够直接在用户空间缓冲区和设备硬件之间移动数据，通常使用 DMA。

如第 1章所述，控制台对应用程序显示为常规文件，应用程序使用 read 和 write 系统调用来读取输入和写入输出。应用程序可能希望控制无法通过标准文件系统调用表达的设备方面（例如，在控制台驱动程序中启用/禁用行缓冲）。Unix 操作系统为此类情况支持 ioctl 系统调用。

计算机的某些用法要求系统必须在有限的时间内做出响应。例如，在安全关键系统中，

错过最后期限可能导致灾难。Xv6 不适用于硬实时设置。用于硬实时的操作系统往往是与应用程序链接的库，以便能够进行分析以确定最坏情况的响应时间。Xv6 也不适用于软实时应用程序，在这种应用程序中，偶尔错过最后期限是可以接受的，因为 xv6 的调度程序过于简单，并且其内核代码路径中中断被长时间禁用。

5.6 练习

1. 修改 uart.c 以完全不使用中断。您可能还需要修改 console.c。
2. 为以太网卡添加一个驱动程序。

Chapter 6

锁

大多数内核，包括 xv6 在内，都会交错执行多个活动。交错的一个来源是多处理器硬件：具有多个独立执行 CPU 的计算机，例如 xv6 的 RISC-V。这些多个 CPU 共享物理内存，xv6 利用这种共享来维护所有 CPU 读写的数据结构。这种共享带来了一种可能性，即一个 CPU 正在读取一个数据结构，而另一个 CPU 正在中途更新它，甚至多个 CPU 同时更新同一个数据；如果没有仔细的设计，这种并行访问很可能会产生不正确的结果或损坏数据结构。即使在单处理器上，内核也可能在多个线程之间切换 CPU，导致它们的执行交错。最后，如果设备中断处理程序修改了与某些可中断代码相同的数据，那么在错误的时间发生中断可能会损坏数据。术语并发指的是多个指令流由于多处理器并行、线程切换或中断而交错执行的情况。

内核中充满了并发访问的数据。例如，两个 CPU 可以同时调用 kalloc，从而并发地从空闲列表的头部弹出。内核设计者喜欢允许大量的并发，因为这可以通过并行来提高性能和响应能力。然而，因此，内核设计者必须确信尽管存在这种并发，代码仍然是正确的。有很多方法可以获得正确的代码，其中一些比其他的更容易推理。旨在在并发下保证正确性的策略以及支持它们的抽象称为并发控制技术。

Xv6 根据情况使用多种并发控制技术；还有更多可能的技术。本章重点介绍一种广泛使用的技术：锁。锁提供互斥，确保一次只有一个 CPU 可以持有锁。如果程序员为每个共享数据项关联一个锁，并且代码在使用一个项时总是持有相关的锁，那么这个项将一次只被一个 CPU 使用。在这种情况下，我们说锁保护了数据项。虽然锁是一种易于理解的并发控制机制，但锁的缺点是它们会限制性能，因为它们会序列化并发操作。

本章的其余部分将解释为什么 xv6 需要锁，xv6 如何实现它们，以及如何使用它们。

6.1 竞争

图 6.1 更详细地说明了这种情况：空闲页的链表位于由两个 CPU 共享的内存中，它们使用加载和存储指令来操作该列表。（实际上，处理器有缓存，但从概念上讲，多处理器系统的行为就像只有一个共享内存一样。）如果没有并发请求，你可能会像下面这样实现一个列表 push 操作：

```
1     struct element {
2       int data;
```

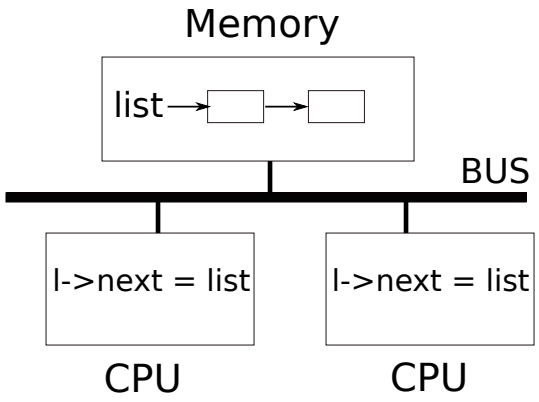

Figure 6.1: 简化的 SMP 架构

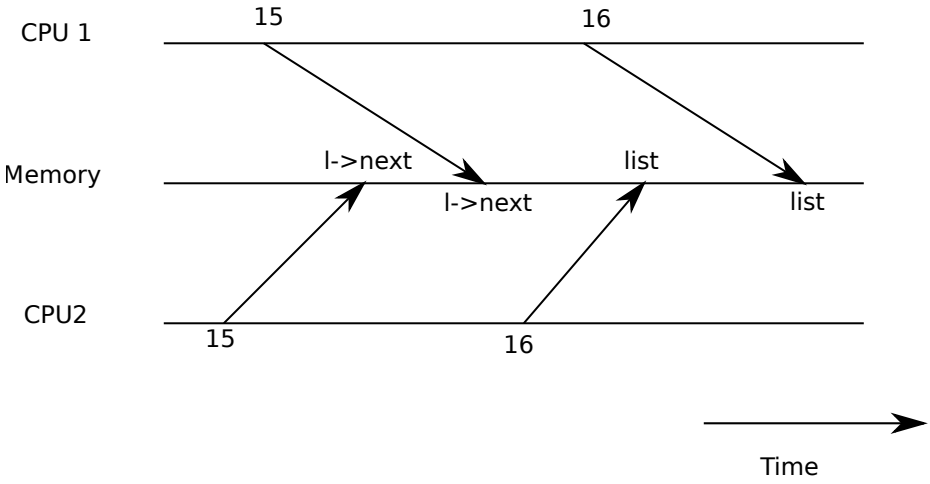

Figure 6.2: 竞争示例

```
3         struct element *next;
4       };
5
6       struct element *list = 0;
7
8       void
9       push(int data)
10      {
11        struct element *l;
12
13        l = malloc(sizeof *l);
14        l->data = data;
15        l->next = list;
16        list = l;
17      }
```

如果单独执行，这个实现是正确的。但是，如果多个副本并发执行，代码就不正确了。如果两个 CPU 同时执行 push，它们都可能在执行第 16 行之前执行第 15 行，如图 6.1 所示，这将导致图 6.2 所示的错误结果。这样就会有两个列表元素的 next 被设置为 list 的旧值。当两个对 list 的赋值发生在第 16 行时，第二个赋值会覆盖第一个；第一个赋值所涉及的元素将丢失。

第 16 行的更新丢失是竞争的一个例子。竞争是指一个内存位置被并发访问，并且至少有一个访问是写操作的情况。竞争通常是错误的标志，要么是更新丢失（如果访问是写操作），要么是读取了未完全更新的数据结构。竞争的结果取决于编译器生成的机器代码、两个相关 CPU 的时序以及内存系统如何对它们的内存操作进行排序，这使得由竞争引起的错误难以重现和调试。例如，在调试 push 时添加打印语句可能会改变执行的时序，从而使竞争消失。

避免竞争的常用方法是使用锁。锁确保互斥，以便一次只有一个 CPU 可以执行 push 的敏感代码行；这使得上述情况不可能发生。正确加锁的上述代码版本只添加了几行（以黄色突出显示）：

```
 6     struct element *list = 0;
 7     struct lock listlock;
 8
 9     void
10     push(int data)
11     {
12       struct element *l;
13       l = malloc(sizeof *l);
14       l->data = data;
15
16       acquire(&listlock);
17       l->next = list;
18       list = l;
19       release(&listlock);
20     }
```

acquire 和 release 之间的指令序列通常被称为临界区。通常说锁在保护 list。

当我们说一个锁保护数据时，我们真正的意思是锁保护了适用于该数据的一些不变量集合。不变量是数据结构在操作中保持的属性。通常，一个操作的正确行为取决于操作开始时不变量为真。操作可能会暂时违反不变量，但必须在完成前重新建立它们。例如，在链表的情况下，不变量是 list 指向列表中的第一个元素，并且每个元素的 next 字段指向下一个元素。push 的实现暂时违反了这个不变量：在第 17 行中，l 指向下一个列表元素，但 list 还没有指向 l （在第 18 行重新建立）。我们上面研究的竞争发生是因为第二个 CPU 在列表不变量被（暂时）违反时执行了依赖于这些不变量的代码。正确使用锁可以确保一次只有一个 CPU 可以在临界区内操作数据结构，这样就不会有 CPU 在数据结构的不变量不成立时执行数据结构操作。

你可以把锁看作是序列化并发的临界区，使它们一次一个地运行，从而保持不变量（假设临界区在隔离状态下是正确的）。你也可以认为由同一个锁保护的临界区彼此之间是原子的，这样每个临界区只能看到来自早期临界区的完整变更集，而永远不会看到部分完成的更新。

虽然锁对于正确性很有用，但它们天生会限制性能。例如，如果两个进程同时调用 kfree，锁将序列化这两个临界区，因此在不同的 CPU 上运行它们没有任何好处。我们说，如果多个进程同时想要同一个锁，它们就会发生冲突，或者说锁经历了争用。内核设计中的一个主要挑战是避免锁争用以追求并行性。Xv6 在这方面做得很少，但复杂的内核会专门组织数据结构和算法来避免锁争用。在列表的例子中，一个内核可能会为每个 CPU 维护一个单独的空闲列表，并且只有在当前 CPU 的列表为空并且必须从另一个 CPU 窃取内存时才接触另一个 CPU 的空闲列表。其他用例可能需要更复杂的设计。

锁的放置对性能也很重要。例如，在 push 中将 acquire 移到更早的位置，在第 13 行之前是正确的。但这可能会降低性能，因为这样一来，对 malloc 的调用就会被序列化。下面的“使用锁”一节提供了一些关于在何处插入 acquire 和 release 调用的指导方针。

6.2 代码：锁

Xv6 有两种类型的锁：自旋锁和睡眠锁。我们从自旋锁开始。Xv6 将自旋锁表示为 struct spinlock (kernel/spinlock.h:2)。结构中重要的字段是 locked，这是一个字，当锁可用时为零，当锁被持有时为非零。逻辑上，xv6 应该通过执行类似以下代码来获取锁：

```
21    void
22    acquire(struct spinlock *lk) // 无法工作！
23    {
24      for(;;) {
25        if(lk->locked == 0) {
26          lk->locked = 1;
27          break;
28        }
29      }
30    }
```

不幸的是，这个实现在多处理器上不能保证互斥。可能会发生两个 CPU 同时到达第 25 行，看到 lk->locked 为零，然后都通过执行第 26 行来获取锁。此时，两个不同的 CPU 持有锁，这违反了互斥属性。我们需要的是一种使第 25 行和第 26 行作为一个原子（即不可分割的）步骤执行的方法。

由于锁被广泛使用，多核处理器通常提供实现第 25 和 26 行原子版本的指令。在 RISC-V 上，这个指令是 amoswap r, a。amoswap 读取内存地址 a 的值，将寄存器 r 的内容写入该地址，并将其读取的值放入 r 中。也就是说，它交换了寄存器和内存地址的内容。它原子地执行这个序列，使用特殊的硬件来防止任何其他 CPU 在读和写之间使用该内存地址。

Xv6 的 acquire (kernel/spinlock.c:22) 使用了可移植的 C 库调用 __sync_lock_test_and_set，它最终归结为 amoswap 指令；返回值是 lk->locked 的旧（交换过的）内容。acquire 函数将交换包装在一个循环中，重试（自旋）直到它获得锁。每次迭代都将一个 1 交换到 lk->locked 中并检查之前的值；如果之前的值是零，那么我们就获得了锁，并且交换将把 lk->locked 设置为 1。如果之前的值是 1，那么某个其他的 CPU 持有锁，并且我们原子地将 1 交换到 lk->locked 中并不会改变它的值。

一旦获取了锁，acquire 会为了调试而记录获取锁的 CPU。lk->cpu 字段受锁保护，并且只能在持有锁时更改。

函数 release (kernel/spinlock.c:47) 与 acquire 相反：它清除 lk->cpu 字段，然后释放锁。从概念上讲，释放只需要将零赋给 lk->locked。C 标准允许编译器用多个存储指令来实现一个赋值，所以一个 C 赋值对于并发代码来说可能是非原子的。相反，release 使用 C 库函数 __sync_lock_release 来执行原子赋值。这个函数也归结为一个 RISC-V amoswap 指令。

6.3 代码：使用锁

使用锁的一个难点是决定使用多少个锁，以及每个锁应该保护哪些数据和不变量。有几个基本原则。首先，任何时候一个变量可以被一个 CPU 写入，而同时另一个 CPU 可以读取或写入它，就应该使用一个锁来防止这两个操作重叠。其次，记住锁保护的是不变量：如果一个不变量涉及多个内存位置，通常所有这些位置都需要由一个锁来保护，以确保不变量得以维持。

上面的规则说明了什么时候需要锁，但没有说明什么时候不需要锁，为了效率，不过度加锁是很重要的，因为锁会降低并行性。如果并行性不重要，那么可以安排只有一个线程，而不用担心锁。一个简单的内核可以在多处理器上通过一个单一的锁来实现这一点，这个锁必须在进入内核时获取，在退出内核时释放（尽管阻塞系统调用如管道读取或 wait 会带来问题）。许多单处理器操作系统已经使用这种方法被转换成在多处理器上运行，有时被称为“大内核锁”，但这种方法牺牲了并行性：一次只有一个 CPU 可以在内核中执行。如果内核进行任何繁重的计算，使用更大的一组更细粒度的锁会更有效率，这样内核就可以在多个 CPU 上同时执行。

作为粗粒度锁定的一个例子，xv6 的 kalloc.c 分配器有一个由单个锁保护的单个空闲列表。如果不同 CPU 上的多个进程试图同时分配页面，每个进程都必须通过在 acquire 中自旋来等待轮到自己。自旋会浪费 CPU 时间，因为它不是有用的工作。如果对锁的争用浪费了相当一部分 CPU 时间，那么也许可以通过改变分配器设计来提高性能，使其具有多个空闲列表，每个列表都有自己的锁，以允许真正的并行分配。

作为细粒度锁定的一个例子，xv6 为每个文件都有一个单独的锁，这样操作不同文件的进程通常可以继续进行而无需等待对方的锁。如果想要允许进程同时写入同一文件的不同区域，文件锁定方案可以变得更加细粒度。最终，锁的粒度决策需要由性能测量和复杂性考虑来驱动。

随着后续章节解释 xv6 的每个部分，它们将提到 xv6 使用锁来处理并发的例子。作为预览，图 6.3 列出了 xv6 中的所有锁。

6.4 死锁和锁顺序

如果内核中的一个代码路径必须同时持有多个锁，那么所有代码路径以相同的顺序获取这些锁是很重要的。如果它们不这样做，就有死锁的风险。假设 xv6 中的两个代码路径需要锁 A 和 B，但代码路径 1 按 A 然后 B 的顺序获取锁，而另一个路径按 B 然后 A 的顺序获取它们。假设线程 T1 执行代码路径 1 并获取锁 A，线程 T2 执行代码路径 2 并获取锁 B。接下来 T1 将尝试获取锁 B，而 T2 将尝试获取锁 A。两个获取都将无限期地阻塞，因为在这两种情况下，另一个线程都持有需要的锁，并且在它的获取返回之前不会释放它。为了避免这种死锁，所有代码路径都必须以相同的顺序获取锁。需要一个全局锁获取顺序

锁	描述
bcache.lock	保护块缓冲缓存条目的分配
cons.lock	序列化对控制台硬件的访问，避免输出混杂
ftable.lock	序列化文件表中 struct file 的分配
itable.lock	保护内存中 inode 条目的分配
vdisk_lock	序列化对磁盘硬件和 DMA 描述符队列的访问
kmem.lock	序列化内存分配
log.lock	序列化对事务日志的操作
pipe's pi->lock	序列化对每个管道的操作
pid_lock	序列化 next_pid 的增量
proc's p->lock	序列化对进程状态的更改
wait_lock	帮助 wait 避免丢失唤醒
tickslock	序列化对滴答计数器的操作
inode's ip->lock	序列化对每个 inode 及其内容的操作
buf's b->lock	序列化对每个块缓冲区的操作

Figure 6.3: xv6 中的锁

意味着锁实际上是每个函数规范的一部分：调用者必须以导致锁按约定顺序获取的方式调用函数。

由于 sleep 的工作方式（参见第 7 章），Xv6 有许多涉及每个进程锁（每个 struct proc 中的锁）的长度为 2 的锁顺序链。例如，consoleintr (kernel/console.c:136) 是处理键入字符的中断例程。当一个新行到达时，任何等待控制台输入的进程都应该被唤醒。为此，consoleintr 在调用 wakeup 时持有 cons.lock，wakeup 会获取等待进程的锁以唤醒它。因此，全局死锁避免锁顺序包括 cons.lock 必须在任何进程锁之前获取的规则。文件系统代码包含 xv6 最长的锁链。例如，创建一个文件需要同时持有一个目录的锁、一个新文件 inode 的锁、一个磁盘块缓冲区的锁、磁盘驱动程序的 vdisk_lock 以及调用进程的 p->lock。为了避免死锁，文件系统代码总是按前面句子中提到的顺序获取锁。

遵守全局死锁避免顺序可能出人意料地困难。有时锁顺序与逻辑程序结构冲突，例如，也许代码模块 M1 调用模块 M2，但锁顺序要求在 M1 中的锁之前获取 M2 中的锁。有时锁的身份事先不知道，也许是因为必须持有一个锁才能发现下一个要获取的锁的身份。这种情况在文件系统中查找路径名的连续组件时出现，在 wait 和 exit 的代码中搜索进程表以查找子进程时也出现。最后，死锁的危险通常是对锁方案可以做得多细粒度的一个约束，因为更多的锁通常意味着更多的死锁机会。避免死锁的需要通常是内核实现中的一个主要因素。

6.5 可重入锁

似乎可以通过使用可重入锁来避免一些死锁和锁排序挑战，可重入锁也称为递归锁。其思想是，如果锁由一个进程持有，并且该进程再次尝试获取该锁，那么内核可以允许这样做（因为该进程已经拥有该锁），而不是像 xv6 内核那样调用 panic。

然而，事实证明，可重入锁使得对并发的推理更加困难：可重入锁打破了锁使临界区相对于其他临界区是原子的直觉。考虑以下函数 f 和 g，以及一个假设的函数 h：

```
struct spinlock lock;
int data = 0; // 受 lock 保护

f() {
  acquire(&lock);
  if(data == 0){
    call_once();
    h();
    data = 1;
  }
  release(&lock);
}

g() {
  acquire(&lock);
  if(data == 0){
    call_once();
    data = 1;
  }
  release(&lock);
}

h() {
    ...
}
```

看这段代码，直觉是 call_once 只会被调用一次：要么由 f 调用，要么由 g 调用，但不会被两者都调用。

但是如果允许可重入锁，并且 h 恰好调用了 g，call_once 将被调用两次。

如果不允许可重入锁，那么 h 调用 g 会导致死锁，这也不是很好。但是，假设调用 call_once 两次是一个严重的错误，那么死锁是更可取的。内核开发人员会观察到死锁（内核 panic）并可以修复代码以避免它，而调用 call_once 两次可能会悄无声息地导致一个难以追踪的错误。

因此，xv6 使用更容易理解的非可重入锁。然而，只要程序员牢记锁定规则，任何一种方法都可以工作。如果 xv6 要使用可重入锁，就需要修改 acquire 来注意锁当前正被调用线程持有。还需要在 struct spinlock 中添加一个嵌套获取的计数，风格类似于接下来讨论的 push_off。

6.6 锁和中断处理程序

自旋锁和中断的交互带来了一个潜在的危险。假设 sys_sleep 持有 tickslock，并且它的 CPU 被一个时钟中断打断。clockintr 会尝试获取 tickslock，看到它被持有，然后等待它被释放。

在这种情况下，tickslock 将永远不会被释放：只有 sys_sleep 可以释放它，但 sys_sleep 在 clockintr 返回之前不会继续运行。所以 CPU 将会死锁，任何需要这两个锁的代码也会冻结。

为了避免这种情况，如果一个自旋锁被中断处理程序使用，一个 CPU 绝不能在启用中断的情况下持有该锁。Xv6 更为保守：当一个 CPU 获取任何锁时，xv6 总是禁用该 CPU 上的中断。中断仍然可能在其他 CPU 上发生，所以一个中断的 acquire 可以等待一个线程释放一个自旋锁；只是不能在同一个 CPU 上。

6.7 指令和内存排序

很自然地会认为程序是按照源代码语句出现的顺序执行的。对于单线程代码来说，这是一个合理的心理模型，但是当多个线程通过共享内存交互时，这是不正确的 [2, 4]。一个原因是编译器发出的加载和存储指令的顺序与源代码所暗示的顺序不同，并且可能完全省略它们（例如通过在寄存器中缓存数据）。另一个原因是 CPU 可能会为了提高性能而乱序执行指令。例如，CPU 可能会注意到在一个串行指令序列中，A 和 B 互不依赖。CPU 可能会先启动指令 B，要么是因为它的输入比 A 的输入先准备好，要么是为了重叠 A 和 B 的执行。

作为一个可能出错的例子，在 push 的这段代码中，如果编译器或 CPU 将对应于第 4 行的存储移动到第 6 行的 release 之后，那将是一场灾难：

```
1        l = malloc(sizeof *l);
2        l->data = data;
3        acquire(&listlock);
4        l->next = list;
5        list = l;
6        release(&listlock);
```

如果发生了这种重排序，将会有一个时间窗口，在此期间另一个 CPU 可以获取锁并观察到更新后的 list，但会看到一个未初始化的 list->next。

好消息是，编译器和 CPU 通过遵循一组称为内存模型的规则来帮助并发程序员，并通过提供一些原语来帮助程序员控制重排序。

6.8 睡眠锁

有时 xv6 需要长时间持有一个锁。例如，文件系统（第 8 章）在磁盘上读写文件内容时会保持文件锁定，而这些磁盘操作可能需要几十毫秒。如果另一个进程想要获取它，长时间持有自旋锁会导致浪费，因为获取进程在自旋时会浪费很长时间的 CPU。自旋锁的另一个缺点是进程在保留自旋锁的同时不能让出 CPU；我们希望这样做，以便其他进程可以在持有锁的进程等待磁盘时使用 CPU。在持有自旋锁时让出是非法的，因为如果第二个线程然后尝试获取自旋锁，可能会导致死锁；因为 acquire 不会让出 CPU，第二个线程的自旋可能会阻止第一个线程运行和释放锁。在持有锁时让出也会违反在持有自旋锁时必须关闭中断的要求。因此，我们想要一种在等待获取时让出 CPU，并在持有锁时允许让出（和中断）的锁。

Xv6 以睡眠锁的形式提供了这种锁。acquiresleep (kernel/sleeplock.c:22) 在等待时让出 CPU，使用的技术将在第 7 章中解释。在较高的层次上，睡眠锁有一个由自旋锁保护的 locked 字段，acquiresleep 对 sleep 的调用会原子地让出 CPU 并释放自旋锁。结果是其他线程可以在 acquiresleep 等待时执行。

因为睡眠锁保持中断启用，所以它们不能在中断处理程序中使用。因为 acquiresleep 可能会让出 CPU，所以睡眠锁不能在自旋锁临界区内使用（尽管自旋锁可以在睡眠锁临界区内使用）。

自旋锁最适合短的临界区，因为等待它们会浪费 CPU 时间；睡眠锁适用于冗长的操作。

6.9 现实世界

尽管对并发原语和并行性进行了多年的研究，但使用锁进行编程仍然具有挑战性。通常最好将锁隐藏在更高级别的构造中，如同步队列，尽管 xv6 没有这样做。如果你使用锁编程，明智的做法是使用一个试图识别竞争的工具，因为很容易忽略一个需要锁的不变量。

大多数操作系统支持 POSIX 线程（Pthreads），它允许一个用户进程有多个线程在不同的 CPU 上并发运行。Pthreads 支持用户级锁、屏障等。Pthreads 还允许程序员可选地指定一个锁应该是可重入的。

在用户级别支持 Pthreads 需要操作系统的支持。例如，如果一个 pthread 在一个系统调用中阻塞，同一进程的另一个 pthread 应该能够在该 CPU 上运行。作为另一个例子，如果一个 pthread 改变了它的进程的地址空间（例如，映射或取消映射内存），内核必须安排运行同一进程线程的其他 CPU 更新它们的硬件页表以反映地址空间的变化。

可以在没有原子指令的情况下实现锁 [10]，但这很昂贵，而且大多数操作系统都使用原子指令。

如果许多 CPU 试图同时获取同一个锁，锁的开销可能会很大。如果一个 CPU 在其本地缓存中缓存了一个锁，而另一个 CPU 必须获取该锁，那么更新持有该锁的缓存行的原子指令必须将该行从一个 CPU 的缓存移动到另一个 CPU 的缓存，并可能使该缓存行的任何其他副本失效。从另一个 CPU 的缓存中获取一个缓存行可能比从本地缓存中获取一个行昂贵几个数量级。

为了避免与锁相关的开销，许多操作系统使用无锁数据结构和算法 [6, 12]。例如，可以实现一个像本章开头那样的链表，在列表搜索期间不需要锁，并且用一个原子指令在列表中插入一个项。然而，无锁编程比使用锁编程更复杂；例如，必须担心指令和内存重排序。使用锁编程已经很困难了，所以 xv6 避免了无锁编程的额外复杂性。

6.10 练习

1. 在 kalloc (kernel/kalloc.c:69) 中注释掉对 acquire 和 release 的调用。这似乎应该会给调用 kalloc 的内核代码带来问题；你期望看到什么症状？当你运行 xv6 时，你看到了这些症状吗？运行 usertests 时呢？如果你没有看到问题，为什么？看看你是否可以通过在 kalloc 的临界区中插入虚拟循环来引发问题。

2. 假设你转而在 kfree 中注释掉了锁（在恢复 kalloc 中的锁之后）。现在可能会出什么问题？kfree 中缺少锁是否比 kalloc 中危害小？

3. 如果两个 CPU 同时调用 kalloc，其中一个将不得不等待另一个，这对性能不利。修改 kalloc.c 以具有更多的并行性，以便来自不同 CPU 的对 kalloc 的同时调用可以继续进行而无需等待对方。

4. 使用 POSIX 线程编写一个并行程序，大多数操作系统都支持它。例如，实现一个并行哈希表并测量 puts/gets 的数量是否随着 CPU 数量的增加而扩展。

5. 在 xv6 中实现 Pthreads 的一个子集。也就是说，实现一个用户级线程库，以便一个用户进程可以有多个线程，并安排这些线程可以在不同的 CPU 上并行运行。提出一个设计，正确处理一个线程进行阻塞系统调用和改变其共享地址空间的情况。

Chapter 7

调度

任何操作系统都可能需要运行比计算机 CPU 数量更多的进程，因此需要一个计划来在进程之间分时共享 CPU。理想情况下，这种共享对用户进程是透明的。一种常见的方法是，通过在硬件 CPU 上多路复用（multiplexing）进程，为每个进程提供其拥有虚拟 CPU 的假象。本章将解释 xv6 如何实现这种多路复用。

7.1 多路复用

Xv6 通过在两种情况下将每个 CPU 从一个进程切换到另一个进程来实现多路复用。首先，当一个进程进行阻塞（必须等待一个事件）的系统调用时，xv6 的sleep和wakeup机制会进行切换，这通常发生在read、wait或sleep中。其次，xv6 会周期性地强制切换，以处理那些长时间计算而不阻塞的进程。前者是自愿切换；后者被称为非自愿切换。这种多路复用创造了每个进程都拥有自己 CPU 的假象。

实现多路复用带来了一些挑战。首先，如何从一个进程切换到另一个进程？基本思想是保存和恢复 CPU 寄存器，但 C 语言无法直接表达这一点，使其变得棘手。其次，如何以对用户进程透明的方式强制切换？Xv6 使用了标准技术，即硬件定时器中断驱动上下文切换。第三，所有的 CPU 都在同一组进程之间切换，因此需要一个锁方案来避免竞争。第四，当一个进程退出时，必须释放其内存和其他资源，但它无法自己完成所有这些工作，因为（例如）它不能在使用自己的内核栈时释放它。第五，多核机器的每个 CPU 都必须记住它正在执行哪个进程，以便系统调用影响正确进程的内核状态。最后，sleep和wakeup允许一个进程放弃 CPU 并等待被另一个进程或中断唤醒。需要小心避免导致唤醒通知丢失的竞争。

7.2 代码：上下文切换

图 7.1概述了从一个用户进程切换到另一个用户进程所涉及的步骤：从用户空间到旧进程内核线程的陷阱（系统调用或中断），上下文切换到当前 CPU 的调度器线程，上下文切换到新进程的内核线程，以及陷阱返回到用户级进程。Xv6 有独立的线程（保存的寄存器和栈）来执行调度器，因为让调度器在任何进程的内核栈上执行都是不安全的：其他 CPU 可能会唤醒该进程并运行它，在两个不同的 CPU 上使用同一个栈将是一场灾难。每个 CPU

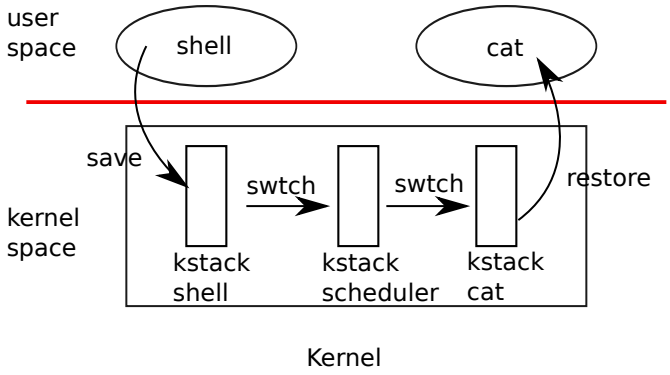

Figure 7.1: 从一个用户进程切换到另一个用户进程。在此示例中，xv6 使用一个 CPU（因此只有一个调度器线程）运行。

都有一个独立的调度器线程，以应对多个 CPU 正在运行希望放弃 CPU 的进程的情况。在本节中，我们将研究内核线程和调度器线程之间切换的机制。

从一个线程切换到另一个线程涉及保存旧线程的 CPU 寄存器，并恢复新线程先前保存的寄存器；栈指针和程序计数器被保存和恢复的事实意味着 CPU 将切换栈和正在执行的代码。

函数swtch为内核线程切换保存和恢复寄存器。swtch不直接了解线程；它只是保存和恢复一组 RISC-V 寄存器，称为上下文（contexts）。当一个进程需要放弃 CPU 时，该进程的内核线程调用swtch来保存自己的上下文并恢复调度器的上下文。每个上下文都包含在一个struct context(kernel/proc.h:2)中，它本身包含在一个进程的struct proc或一个 CPU 的struct cpu中。swtch接受两个参数：struct context*old和struct context*new。它将当前寄存器保存在old中，从new加载寄存器，然后返回。

让我们跟随一个进程通过swtch进入调度器。我们在第 4章看到,中断结束时的一种可能性是usertrap调用yield。yield转而调用sched，后者调用swtch将当前上下文保存在p->context中，并切换到先前保存在cpu->context(kernel/proc.c:506)中的调度器上下文。

swtch(kernel/swtch.S:3)只保存被调用者保存的寄存器；C 编译器在调用者中生成代码以在栈上保存调用者保存的寄存器。swtch知道struct context中每个寄存器字段的偏移量。它不保存程序计数器。相反，swtch保存ra寄存器，该寄存器保存了调用swtch的返回地址。现在swtch从新上下文中恢复寄存器，新上下文保存了先前swtch保存的寄存器值。当swtch返回时，它返回到恢复的ra寄存器所指向的指令，即新线程先前调用swtch的指令。此外，它在新线程的栈上返回，因为恢复的sp指向那里。

在我们的例子中，sched调用swtch切换到cpu->context，即每个 CPU 的调度器上下文。该上下文是在过去某个时刻scheduler调用swtch(kernel/proc.c:466)切换到当前放弃 CPU 的进程时保存的。当我们一直在跟踪的swtch返回时，它不会返回到sched，而是返回到scheduler，栈指针位于当前 CPU 的调度器栈中。

7.3 代码：调度

我们刚刚看到 xv6 在调用swtch期间持有p->lock：swtch的调用者必须已经持有该锁，并且锁的控制权传递给被切换到的代码。这种安排不寻常：更常见的是获取锁的线程也释放它。Xv6 的上下文切换必须打破这个惯例，因为p->lock保护进程的state和context字段的不变性，这些不变性在swtch中执行时是不成立的。例如，如果在swtch期间没有持有p->lock，另一个 CPU 可能会在yield将其状态设置为RUNNABLE之后，但在swtch使其停止使用自己的内核栈之前，决定运行该进程。结果将是两个 CPU 在同一个栈上运行，这会引起混乱。一旦yield开始修改一个正在运行的进程的状态以使其变为RUNNABLE，p->lock必须保持持有直到不变性恢复：最早的正确释放点是在scheduler（在其自己的栈上运行）清除c->proc之后。类似地，一旦scheduler开始将一个RUNNABLE的进程转换为RUNNING，锁就不能被释放，直到进程的内核线程完全运行（在swtch之后，例如在yield中）。

有一种情况是调度器的swtch调用不会在sched中结束。allocproc将新进程的上下文ra寄存器设置为forkret (kernel/proc.c:524)，以便其第一次swtch“返回”到该函数的开头。forkret的存在是为了释放p->lock；否则，由于新进程需要像从fork返回一样返回到用户空间，它可以在usertrapret处开始。

7.4 代码: mycpu 和 myproc

Xv6 经常需要一个指向当前进程的proc结构的指针。在单处理器上，可以有一个全局变量指向当前的proc。这在多核机器上行不通，因为每个 CPU 执行不同的进程。解决这个问题的方法是利用每个 CPU 都有自己的一套寄存器这一事实。

cpuid和mycpu的返回值是脆弱的：如果定时器中断并导致线程让出并在稍后在不同的 CPU 上恢复执行，先前返回的值将不再正确。为了避免这个问题，xv6 要求调用者禁用中断，并且只有在他们完成使用返回的struct cpu后才启用它们。

函数myproc(kernel/proc.c:83)返回当前 CPU 上运行的进程的struct proc指针。myproc禁用中断，调用mycpu，从struct cpu中获取当前进程指针（c->proc），然后启用中断。myproc的返回值即使在启用中断的情况下也是安全的：如果定时器中断将调用进程移动到不同的 CPU，它的struct proc指针将保持不变。

7.5 休眠与唤醒

调度和锁有助于隐藏一个线程对另一个线程的操作，但我们还需要有助于线程有意交互的抽象。例如，xv6 中管道的读取者可能需要等待一个写入进程产生数据；父进程对wait的调用可能需要等待一个子进程退出；以及一个读取磁盘的进程需要等待磁盘硬件完成读取。xv6 内核在这些情况（以及许多其他情况）下使用一种称为休眠和唤醒的机制。休眠允许一个内核线程等待一个特定事件；另一个线程可以调用唤醒来指示等待指定事件的线程应该恢复。休眠和唤醒通常被称为序列协调（sequence coordination）或条件同步（conditional synchronization）机制。

休眠和唤醒提供了一个相对底层的同步接口。为了激发它们在 xv6 中的工作方式，我们将用它们来构建一个更高级别的同步机制，称为信号量（semaphore） [5]，用于协调生

产者和消费者（xv6 不使用信号量）。信号量维护一个计数并提供两个操作。“V”操作（对于生产者）增加计数。“P”操作（对于消费者）等到计数非零，然后递减它并返回。如果只有一个生产者线程和一个消费者线程，并且它们在不同的 CPU 上执行，并且编译器没有进行过于激进的优化，那么这个实现将是正确的：

```
100   struct semaphore {
101     struct spinlock lock;
102     int count;
103   };
104
105   void
106   V(struct semaphore *s)
107   {
108       acquire(&s->lock);
109       s->count += 1;
110       release(&s->lock);
111   }
112
113   void
114   P(struct semaphore *s)
115   {
116       while(s->count == 0)
117         ;
118       acquire(&s->lock);
119       s->count -= 1;
120       release(&s->lock);
121   }
```

上面的实现是昂贵的。如果生产者很少活动，消费者将花费大部分时间在while循环中旋转，希望计数非零。消费者的 CPU 可以通过重复轮询（polling）s->count来找到比忙等待（busy waiting）更有效率的工作。避免忙等待需要一种让消费者让出 CPU 并仅在V增加计数后才恢复的方法。

这是一个朝着这个方向迈出的一步，虽然正如我们将看到的，这还不够。让我们想象一对调用，sleep和wakeup，它们的工作方式如下。sleep(chan)等待由chan的值指定的事件，称为等待通道（wait channel）。sleep将调用进程置于休眠状态，释放 CPU 用于其他工作。wakeup(chan)唤醒所有正在调用具有相同chan的sleep的进程（如果有的话），导致它们的sleep调用返回。如果没有进程在chan上等待，wakeup什么也不做。我们可以改变信号量实现以使用sleep和wakeup（更改以黄色突出显示）：

```
200   void
201   V(struct semaphore *s)
202   {
203       acquire(&s->lock);
204       s->count += 1;
205       wakeup(s);
206       release(&s->lock);
```

```
207     }
208
209     void
210     P(struct semaphore *s)
211     {
212       while(s->count == 0)
213         sleep(s);
214       acquire(&s->lock);
215       s->count -= 1;
216       release(&s->lock);
217     }
```

P现在放弃 CPU 而不是自旋,这很好。然而,事实证明,使用这种接口设计sleep和wakeup而不遭受所谓的丢失唤醒（lost wake-up）问题并不简单。假设P在第 212行发现s->count == 0。当P在第 212行和第 213行之间时，V在另一个 CPU 上运行：它将s->count更改为非零并调用wakeup，它发现没有进程在休眠，因此什么也不做。现在P在第 213行继续执行：它调用sleep并进入休眠状态。这导致一个问题：P正在休眠等待一个已经发生过的V调用。除非我们幸运地生产者再次调用V，否则即使计数非零，消费者也将永远等待。

这个问题的根源在于，P仅在s->count == 0时休眠的不变性被V在恰好错误的时刻运行所破坏。一个不正确的保护不变性的方法是在P中移动锁的获取（下面以黄色突出显示），以便其对计数的检查和对sleep的调用是原子的：

```
300     void
301     V(struct semaphore *s)
302     {
303       acquire(&s->lock);
304       s->count += 1;
305       wakeup(s);
306       release(&s->lock);
307     }
308
309     void
310     P(struct semaphore *s)
311     {
312       acquire(&s->lock);
313       while(s->count == 0)
314         sleep(s);
315       s->count -= 1;
316       release(&s->lock);
317     }
```

人们可能希望这个版本的P会避免丢失唤醒，因为锁阻止了V在第 313行和第 314行之间执行。它确实做到了这一点，但它也导致了死锁：P在休眠时持有锁，所以V将永远阻塞等待锁。

我们将通过改变sleep的接口来修复前面的方案：调用者必须将条件锁（condition lock）传递给sleep，以便它可以在调用进程被标记为休眠并等待在休眠通道上之后释放锁。该锁

将强制一个并发的V等到P完成将自己置于休眠状态，以便wakeup将找到休眠的消费者并唤醒它。一旦消费者再次被唤醒，sleep会在返回之前重新获取锁。我们新的正确的休眠/唤醒方案可按如下方式使用（更改以黄色突出显示）：

```
400   void
401   V(struct semaphore *s)
402   {
403     acquire(&s->lock);
404     s->count += 1;
405     wakeup(s);
406     release(&s->lock);
407   }
408
409   void
410   P(struct semaphore *s)
411   {
412     acquire(&s->lock);
413     while(s->count == 0)
414       sleep(s, &s->lock);
415     s->count -= 1;
416     release(&s->lock);
417   }
```

P持有s->lock的事实阻止了V在P检查s->count和其调用sleep之间尝试唤醒它。然而，sleep必须释放s->lock并将消费进程置于休眠状态，从wakeup的角度来看，这必须是原子的，以避免丢失唤醒。

7.6 代码: 休眠与唤醒

在某个时刻，一个进程将获取条件锁，设置休眠者正在等待的条件，并调用wakeup(chan)。重要的是wakeup在持有条件锁的同时被调用[1]。wakeup遍历进程表(kernel/proc.c:579)。它获取它检查的每个进程的p->lock。当wakeup发现一个处于SLEEPING状态且具有匹配chan的进程时，它将该进程的状态更改为RUNNABLE。下一次scheduler运行时，它将看到该进程已准备好运行。

为什么sleep和wakeup的锁定规则能确保一个即将进入休眠的进程不会错过一个并发的唤醒? 即将进入休眠的进程从 * 检查条件之前 * 到 * 被标记为 SLEEPING 之后 *，都持有条件锁或它自己的p->lock，或两者兼有。调用 wakeup 的进程在 wakeup 的循环中持有 * 两个 * 锁。因此，唤醒者要么在消费线程检查条件之前使条件为真；要么唤醒者的wakeup严格地在休眠线程被标记为 SLEEPING 之后检查它。然后wakeup将看到休眠的进程并唤醒它（除非有其他东西先唤醒它）。

有时多个进程在同一个通道上休眠；例如，多个进程从一个管道读取。一个对wakeup的调用将唤醒所有这些进程。其中一个将首先运行并获取sleep被调用时所持有的锁，并（在管道的情况下）读取任何等待的数据。其他进程会发现，尽管被唤醒，但没有数据可读。从

[1]严格来说，如果wakeup只是跟在acquire之后就足够了（也就是说，可以在release之后调用wakeup）。

它们的角度来看，唤醒是“虚假的”，它们必须再次休眠。因此，sleep总是在一个检查条件的循环内调用。

如果两个 sleep/wakeup 的使用意外地选择了同一个通道，也不会造成伤害：它们会看到虚假的唤醒，但如上所述的循环将容忍这个问题。sleep/wakeup 的魅力很大程度上在于它既是轻量级的（不需要创建特殊的数据结构来充当休眠通道），又提供了一层间接性（调用者不需要知道它们正在与哪个特定的进程交互）。

7.7 代码: 管道

一个更复杂的例子是 xv6 的管道实现，它使用sleep和wakeup来同步生产者和消费者。我们在第 1章中看到了管道的接口：写入管道一端的字节被复制到内核缓冲区，然后可以从管道的另一端读取。未来的章节将探讨围绕管道的文件描述符支持，但现在让我们看看pipewrite和piperead的实现。

每个管道由一个struct pipe表示,其中包含一个lock和一个data缓冲区。字段nread和nwrite计算从缓冲区读取和写入的总字节数。缓冲区是环形的：在buf[PIPESIZE-1]之后写入的下一个字节是buf[0]。计数器不会回绕。这个约定让实现能够区分一个满的缓冲区（nwrite == nread+PIPESIZE）和一个空的缓冲区（nwrite == nread），但这意味着索引缓冲区必须使用buf[nread % PIPESIZE]而不是仅仅buf[nread]（对于nwrite也是如此）。

管道代码为读者和写者使用不同的休眠通道（pi->nread和pi->nwrite）；在有大量读者和写者等待同一个管道的不太可能的情况下，这可能会使系统更有效率。管道代码在一个检查休眠条件的循环内休眠；如果有多个读者或写者，除了第一个被唤醒的进程外，所有进程都会看到条件仍然为假并再次休眠。

7.8 代码: Wait, exit, 和 kill

sleep和wakeup可以用于多种等待。一个有趣的例子，在第 1章中介绍过，是子进程的exit和其父进程的wait之间的交互。在子进程死亡时，父进程可能已经在 wait 中休眠，或者可能在做其他事情；在后一种情况下，后续对 wait 的调用必须观察到子进程的死亡，可能是在它调用 exit 很久之后。xv6 记录子进程死亡直到 wait 观察到它的方式是让 exit 将调用者置于ZOMBIE状态，它一直保持在该状态，直到父进程的 wait 注意到它，将子进程的状态更改为 UNUSED，复制子进程的退出状态，并将子进程的进程 ID 返回给父进程。如果父进程在子进程之前退出，父进程会将子进程交给 init 进程，该进程会永久调用 wait；因此每个子进程都有一个父进程来为其清理。一个挑战是避免同时发生的父子进程的wait和exit 以及同时发生的exit和exit之间的竞争和死锁。

exit (kernel/proc.c:347) 记录退出状态, 释放一些资源, 调用reparent将其子进程交给 init 进程，唤醒父进程以防它在wait中，将调用者标记为僵尸，并永久让出 CPU。exit 在此序列中同时持有wait_lock和p->lock。exit 持有wait_lock是因为它是wakeup(p->parent)的条件锁，防止在wait中的父进程丢失唤醒。exit 也必须持有p->lock，以防止在wait中的父进程在子进程最终调用swtch之前看到子进程处于ZOMBIE状态。exit 以与wait相同的顺序获取这些锁以避免死锁。

exit 在将其状态设置为ZOMBIE之前唤醒父进程可能看起来不正确，但这是安全的：虽

然wakeup可能会导致父进程运行，但wait中的循环在子进程的p->lock被 scheduler 释放之前无法检查子进程，所以wait在exit将其状态设置为ZOMBIE(kernel/proc.c:379)很久之后才能看到退出的进程。

如果受害者进程在sleep中， kill 对wakeup的调用将导致受害者从sleep返回。这可能有潜在的危险，因为正在等待的条件可能不为真。然而，xv6 对sleep的调用总是被包装在一个while循环中，该循环在sleep返回后重新测试条件。一些对sleep的调用也在循环中测试p->killed，并在设置时放弃当前活动。这只有在放弃是正确的情况下才会这样做。例如，管道读写代码(kernel/pipe.c:84)在 killed 标志被设置时返回；最终代码将返回到 trap，trap 将再次检查p->killed并退出。

一些 xv6 的sleep循环不检查p->killed，因为代码正处于一个应该原子化的多步系统调用的中间。virtio 驱动程序(kernel/virtio_disk.c:285)是一个例子：它不检查p->killed，因为一个磁盘操作可能是一组写操作中的一个，而所有这些写操作都是为了让文件系统处于一个正确的状态所必需的。一个在等待磁盘 I/O 时被杀死的进程在完成当前系统调用并且usertrap看到 killed 标志之前不会退出。

7.9 进程锁定

与每个进程关联的锁（p->lock）是 xv6 中最复杂的锁。一种简单的思考p->lock的方式是，在读取或写入以下任何struct proc字段时必须持有它：p->state、p->chan、p->killed、p->xstate和p->pid。这些字段可以被其他进程或其他 CPU 上的调度器线程使用，所以它们必须由一个锁来保护是很自然的。

然而，p->lock的大多数用途是保护 xv6 进程数据结构和算法的更高级别的方面。以下是p->lock所做的全部事情：

- 与p->state一起，它防止了为新进程分配proc[]槽位时的竞争。
- 它在进程创建或销毁时将其隐藏起来。
- 它防止父进程的wait收集一个已经将其状态设置为ZOMBIE但尚未让出 CPU 的进程。
- 它防止另一个 CPU 的调度器在让出进程将其状态设置为RUNNABLE之后但在完成swtch之前决定运行它。
- 它确保只有一个 CPU 的调度器决定运行一个RUNNABLE的进程。
- 它防止定时器中断导致进程在swtch中让出。
- 与条件锁一起，它有助于防止wakeup忽略一个正在调用sleep但尚未完成让出 CPU 的进程。
- 它防止 kill 的受害者进程在 kill 检查p->pid和设置p->killed之间退出并可能被重新分配。
- 它使 kill 对p->state的检查和写入成为原子操作。

p->parent字段由全局锁wait_lock而不是p->lock保护。只有一个进程的父进程会修改p->parent，尽管该字段由进程本身和其他正在寻找其子进程的进程读取。wait_lock的目的是当wait休眠等待任何子进程退出时充当条件锁。一个正在退出的子进程持有wait_lock或p->lock直到它将其状态设置为ZOMBIE，唤醒其父进程，并让出 CPU。wait_lock还序列化了父进程和子进程并发的exit，以便init进程（继承了子进程）保证从其wait中被唤醒。wait_lock是一个全局锁而不是每个父进程的锁，因为，在一个进程获取它之前，它无法知道它的父进程是谁。

7.10 现实世界

xv6 调度器实现了一个简单的调度策略，即轮流运行每个进程。这个策略被称为轮询调度（round robin）。真正的操作系统实现了更复杂的策略，例如，允许进程有优先级。其思想是，一个可运行的高优先级进程将被调度器优先于一个可运行的低优先级进程。这些策略可能很快变得复杂，因为通常存在相互竞争的目标：例如，操作系统可能还希望保证公平性和高吞吐量。此外，复杂的策略可能导致意想不到的交互，例如优先级反转（priority inversion）和护航（convoys）。当一个低优先级和一个高优先级的进程都使用一个特定的锁时，可能会发生优先级反转，当低优先级进程获取该锁时，可能会阻止高优先级进程取得进展。当许多高优先级进程等待一个获取了共享锁的低优先级进程时，可能会形成一个长的等待进程护航；一旦护航形成，它可能会持续很长时间。为了避免这些问题，在复杂的调度器中需要额外的机制。

sleep和wakeup是一个简单而有效的同步方法，但还有许多其他方法。所有这些方法的第一个挑战是避免我们在本章开头看到的“丢失唤醒”问题。最初的 Unix 内核的sleep只是禁用中断，这在 Unix 在单 CPU 系统上运行时就足够了。因为 xv6 在多处理器上运行，它向sleep添加了一个显式锁。FreeBSD 的msleep采取了同样的方法。Plan 9 的sleep使用一个回调函数，该函数在进入休眠前持有调度锁运行；该函数作为对休眠条件的最后检查，以避免丢失唤醒。Linux 内核的sleep使用一个显式的进程队列，称为等待队列，而不是一个等待通道；该队列有自己的内部锁。

在wakeup中扫描整个进程集是低效的。一个更好的解决方案是在sleep和wakeup中用一个保存了在该结构上休眠的进程列表的数据结构替换chan，例如 Linux 的等待队列。Plan 9 的sleep和wakeup称该结构为会合点。许多线程库将相同的结构称为条件变量；在这种情况下，操作sleep和wakeup被称为wait和signal。所有这些机制都有相同的特点：休眠条件由某种在休眠期间原子地释放的锁保护。

wakeup的实现唤醒了所有在特定通道上等待的进程，并且可能有很多进程在等待该特定通道。操作系统将调度所有这些进程，它们将竞争检查休眠条件。以这种方式行为的进程有时被称为惊群效应（thundering herd），最好避免。大多数条件变量有两个用于wakeup的原语：signal，唤醒一个进程，和broadcast，唤醒所有等待的进程。

信号量通常用于同步。计数通常对应于管道缓冲区中可用的字节数或进程拥有的僵尸子进程数。使用显式计数作为抽象的一部分可以避免“丢失唤醒”问题：有一个关于已发生唤醒次数的显式计数。该计数还避免了虚假唤醒和惊群效应问题。

终止进程并清理它们在 xv6 中引入了许多复杂性。在大多数操作系统中，它甚至更复杂，因为，例如，受害者进程可能在内核深处休眠，展开其栈需要小心，因为调用栈上的每个函数可能需要进行一些清理工作。一些语言通过提供异常机制来提供帮助，但 C 语言

没有。此外，还有其他事件可能导致休眠的进程被唤醒，即使它正在等待的事件尚未发生。例如，当一个 Unix 进程正在休眠时，另一个进程可能会向它发送一个signal。在这种情况下，进程将从被中断的系统调用返回，返回值为-1，错误代码设置为 EINTR。应用程序可以检查这些值并决定做什么。Xv6 不支持信号，因此不会出现这种复杂性。

Xv6 对 kill 的支持不完全令人满意：有一些休眠循环可能应该检查p->killed。一个相关的问题是，即使对于检查p->killed的sleep循环，sleep和 kill 之间也存在竞争；后者可能会在受害者的休眠循环检查p->killed之后但在其调用sleep之前设置p->killed并试图唤醒受害者。如果发生此问题，受害者在它等待的条件发生之前不会注意到p->killed。这可能会晚很多，甚至永远不会（例如，如果受害者正在等待来自控制台的输入，但用户不输入任何内容）。

一个真正的操作系统会用一个显式的空闲列表在常数时间内找到空闲的proc结构，而不是像allocproc那样进行线性时间搜索；xv6 为了简单起见使用了线性扫描。

7.11 练习

1. 在 xv6 中实现信号量，不使用sleep和wakeup（但可以使用自旋锁）。选择 xv6 中几个使用 sleep 和 wakeup 的地方，并用信号量替换它们。评价结果。

2. 修复上面提到的 kill 和sleep之间的竞争，以便在受害者的休眠循环检查p->killed之后但在调用sleep之前发生的 kill 会导致受害者放弃当前的系统调用。

3. 设计一个计划，以便每个休眠循环都检查p->killed，这样，例如，在 virtio 驱动程序中的进程如果被另一个进程杀死，可以从 while 循环中快速返回。

4. 修改 xv6，在从一个进程的内核线程切换到另一个进程时只使用一次上下文切换，而不是通过调度器线程切换。让出的线程需要自己选择下一个线程并调用 swtch。挑战将是防止多个 CPU 意外执行同一个线程；正确处理锁定；以及避免死锁。

5. 修改 xv6 的scheduler，在没有可运行进程时使用 RISC-V 的WFI（等待中断）指令。尝试确保，在任何时候有可运行的进程等待运行时，没有 CPU 在 WFI 中暂停。

Chapter 8

文件系统

文件系统的目的是组织和存储数据。文件系统通常支持用户和应用程序之间的数据共享，以及 持久性，以便数据在重启后仍然可用。

xv6 文件系统提供类 Unix 的文件、目录和路径名（参见第 1 章），并将其数据存储在 virtio 磁盘上以实现持久性。该文件系统解决了几个挑战：

- 文件系统需要磁盘上的数据结构来表示命名的目录和文件树，记录每个文件内容所在的块的标识，并记录磁盘的哪些区域是空闲的。
- 文件系统必须支持 崩溃恢复。也就是说，如果发生崩溃（例如，电源故障），文件系统必须在重启后仍能正常工作。风险在于崩溃可能会中断一系列更新，并留下不一致的磁盘数据结构（例如，一个块既在文件中使用又被标记为空闲）。
- 不同的进程可能同时操作文件系统，因此文件系统代码必须协调以维护不变量。
- 访问磁盘比访问内存慢几个数量级，因此文件系统必须维护一个内存中的常用块缓存。

本章的其余部分将解释 xv6 如何应对这些挑战。

8.1 概述

xv6 文件系统的实现分为七层，如图 8.1 所示。磁盘层在 virtio 硬盘上读取和写入块。缓冲区缓存层缓存磁盘块并同步对它们的访问，确保一次只有一个内核进程可以修改存储在任何特定块中的数据。日志层允许更高层将对多个块的更新包装在 事务中，并确保在发生崩溃时，这些块以原子方式更新（即，要么全部更新，要么都不更新）。inode 层提供单个文件，每个文件表示为一个 inode，具有唯一的 i-number 和一些保存文件数据的块。目录层将每个目录实现为一种特殊的 inode，其内容是一系列目录条目，每个条目包含文件名和 i-number。路径名层提供像 /usr/rtm/xv6/fs.c 这样的分层路径名，并通过递归查找来解析它们。文件描述符层使用文件系统接口抽象了许多 Unix 资源（例如，管道、设备、文件等），简化了应用程序员的生活。

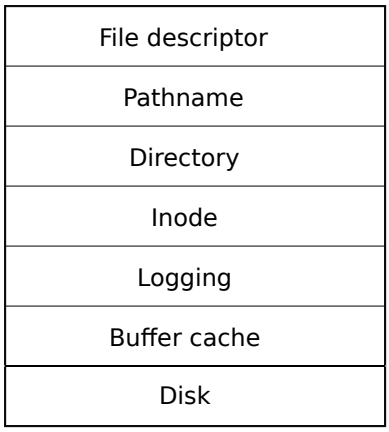

Figure 8.1: xv6 文件系统的层次结构。

传统上，磁盘硬件将磁盘上的数据呈现为一系列编号的 512 字节 块 （也称为 扇区）：扇区 0 是前 512 字节，扇区 1 是下一个，依此类推。操作系统为其文件系统使用的块大小可能与磁盘使用的扇区大小不同，但通常块大小是扇区大小的倍数。Xv6 将其已读入内存的块的副本保存在 struct buf (kernel/buf.h:1) 类型的对象中。此结构中存储的数据有时与磁盘不同步：它可能尚未从磁盘读入（磁盘正在处理但尚未返回扇区的内容），或者它可能已被软件更新但尚未写入磁盘。

文件系统必须有一个计划，用于在磁盘上存储 inode 和内容块。为此，xv6 将磁盘划分为几个部分，如图 8.2 所示。文件系统不使用块 0（它包含引导扇区）。块 1 被称为 超级块；它包含有关文件系统的元数据（文件系统大小（以块为单位）、数据块数、inode 数以及日志中的块数）。从 2 开始的块保存日志。日志之后是 inode，每个块有多个 inode。之后是位图块，跟踪哪些数据块正在使用。其余块是数据块；每个块要么在位图块中标记为空闲，要么保存文件或目录的内容。超级块由一个名为 mkfs 的独立程序填充，该程序构建初始文件系统。

本章的其余部分将讨论每一层，从缓冲区缓存开始。请注意那些在较低层精心选择的抽象简化了较高层设计的情况。

8.2 缓冲区缓存层

缓冲区缓存有两个工作：（1）同步对磁盘块的访问，以确保内存中只有一个块的副本，并且一次只有一个内核线程使用该副本；（2）缓存常用块，这样就不需要从慢速磁盘重新读取它们。代码在 bio.c 中。

缓冲区缓存导出的主要接口包括 bread 和 bwrite；前者获取一个 buf，其中包含一个可以在内存中读取或修改的块的副本，而后者将修改后的缓冲区写入磁盘上的相应块。内核线程在完成缓冲区操作后必须通过调用 brelse 来释放它。缓冲区缓存使用每个缓冲区的休眠锁来确保一次只有一个线程使用每个缓冲区（从而使用每个磁盘块）；bread 返回一个锁定的缓冲区，而 brelse 释放该锁。

让我们回到缓冲区缓存。缓冲区缓存具有固定数量的缓冲区来保存磁盘块，这意味着如果文件系统请求的块不在缓存中，缓冲区缓存必须回收当前保存其他块的缓冲区。缓冲

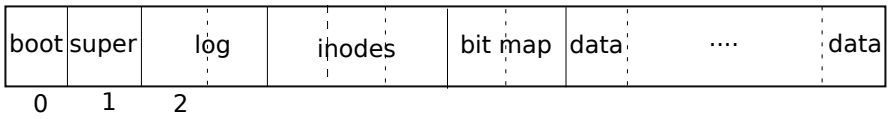

Figure 8.2: xv6 文件系统的结构。

区缓存回收最近最少使用的缓冲区用于新块。假设是最近最少使用的缓冲区是最近最可能再次使用的。

8.3 代码：缓冲区缓存

缓冲区缓存是一个双向链表的缓冲区。函数 binit，由 main (kernel/main.c:27) 调用，用静态数组 buf (kernel/bio.c:43-52) 中的 NBUF 个缓冲区初始化列表。对缓冲区缓存的所有其他访问都通过 bcache.head 引用链表，而不是 buf 数组。

一个缓冲区有两个与之关联的状态字段。字段 valid 表示缓冲区包含该块的副本。字段 disk 表示缓冲区内容已交给磁盘，磁盘可能会更改缓冲区（例如，将数据从磁盘写入 data）。

bread (kernel/bio.c:93) 调用 bget 来获取给定扇区的缓冲区 (kernel/bio.c:97)。如果需要从磁盘读取缓冲区，bread 调用 virtio_disk_rw 来执行此操作，然后返回缓冲区。

bget (kernel/bio.c:59) 扫描缓冲区列表,查找具有给定设备和扇区号的缓冲区 (kernel/bio.c:65-73)。如果存在这样的缓冲区，bget 获取该缓冲区的休眠锁。bget 然后返回锁定的缓冲区。

如果没有给定扇区的缓存缓冲区，bget 必须创建一个，可能会重用一个保存不同扇区的缓冲区。它第二次扫描缓冲区列表，寻找一个未被使用的缓冲区（b->refcnt = 0）；任何这样的缓冲区都可以使用。bget 编辑缓冲区元数据以记录新的设备和扇区号并获取其休眠锁。请注意，赋值 b->valid = 0 确保 bread 将从磁盘读取块数据，而不是错误地使用缓冲区的先前内容。

每个磁盘扇区最多只有一个缓存缓冲区，这一点很重要，以确保读者看到写入，并且因为文件系统使用缓冲区上的锁进行同步。bget 通过持有 bache.lock bcache.lock 从第一个循环检查块是否被缓存到第二个循环声明块现在被缓存（通过设置 dev、blockno 和 refcnt）来确保此不变量。这使得检查块是否存在和（如果不存在）指定一个缓冲区来保存该块是原子的。

bget 在 bcache.lock 临界区之外获取缓冲区的休眠锁是安全的，因为非零的 b->refcnt 阻止了缓冲区被重用于不同的磁盘块。休眠锁保护对块的缓冲内容的读取和写入，而 bcache.lock 保护有关哪些块被缓存的信息。

如果所有缓冲区都繁忙，则说明有太多进程同时执行文件系统调用；bget 会 panic。更优雅的响应可能是休眠直到有缓冲区变为空闲，尽管那样可能会出现死锁。

一旦 bread 读取了磁盘（如果需要）并返回缓冲区给其调用者，调用者就独占使用该缓冲区，可以读取或写入数据字节。如果调用者确实修改了缓冲区，它必须在释放缓冲区之前调用 bwrite 将更改的数据写入磁盘。bwrite (kernel/bio.c:107) 调用 virtio_disk_rw 与磁盘硬件通信。

当调用者完成一个缓冲区时，它必须调用 brelse 来释放它。（名称 brelse，是 b-release 的缩写，虽然晦涩但值得学习：它起源于 Unix，也在 BSD、Linux 和 Solaris 中使用。）brelse (kernel/bio.c:117) 释放休眠锁并将缓冲区移动到链表的前端 (kernel/bio.c:128-133)。移

动缓冲区会导致列表按缓冲区最近使用的顺序排序（意味着释放）：列表中的第一个缓冲区是最近使用的，最后一个是最近最少使用的。bget 中的两个循环利用了这一点：扫描现有缓冲区最坏情况下需要处理整个列表，但首先检查最近使用的缓冲区（从 bcache.head 开始并跟随 next 指针）将在有良好引用局部性时减少扫描时间。选择重用缓冲区的扫描通过向后扫描（跟随 prev 指针）来选择最近最少使用的缓冲区。

8.4 日志层

文件系统设计中最有趣的问题之一是崩溃恢复。这个问题之所以出现，是因为许多文件系统操作涉及对磁盘的多次写入，而在部分写入后发生崩溃可能会使磁盘上的文件系统处于不一致状态。例如，假设在文件截断期间发生崩溃（将文件长度设置为零并释放其内容块）。根据磁盘写入的顺序，崩溃可能会留下一个引用已释放内容块的 inode，或者可能会留下一个已分配但未被引用的内容块。

后者相对良性，但引用已释放块的 inode 在重启后很可能会导致严重问题。重启后，内核可能会将该块分配给另一个文件，现在我们就有两个不同的文件无意中指向同一个块。如果 xv6 支持多用户，这种情况可能是一个安全问题，因为旧文件的所有者将能够读写属于不同用户的新文件中的块。

Xv6 通过一种简单的日志记录形式解决了文件系统操作期间崩溃的问题。xv6 系统调用不直接写入磁盘上的文件系统数据结构。相反，它将它希望进行的所有磁盘写入的描述放在磁盘上的一个 日志 中。一旦系统调用记录了其所有写入，它就会向磁盘写入一个特殊的 提交 记录，指示日志包含一个完整的操作。此时，系统调用将写入复制到磁盘上的文件系统数据结构。在这些写入完成后，系统调用会擦除磁盘上的日志。

如果系统崩溃并重启，文件系统代码在运行任何进程之前会如下恢复：如果日志被标记为包含一个完整的操作，那么恢复代码会将写入复制到它们在磁盘文件系统中的所属位置。如果日志未被标记为包含一个完整的操作，恢复代码会忽略该日志。恢复代码最后会擦除日志。

为什么 xv6 的日志能解决文件系统操作期间崩溃的问题？如果崩溃发生在操作提交之前，那么磁盘上的日志将不会被标记为完成，恢复代码将忽略它，磁盘的状态将如同操作从未开始一样。如果崩溃发生在操作提交之后，那么恢复将重放操作的所有写入，如果操作已开始将它们写入磁盘上的数据结构，则可能会重复它们。在任何一种情况下，日志都使操作相对于崩溃是原子的：恢复后，操作的所有写入要么全部出现在磁盘上，要么全都不出现。

8.5 日志设计

日志位于一个已知的固定位置，在超级块中指定。它由一个头部块和一系列更新的块副本（“日志块”）组成。头部块包含一个扇区号数组，每个日志块对应一个，以及日志块的数量。磁盘上头部块中的计数要么是零，表示日志中没有事务，要么是非零，表示日志包含一个完整的已提交事务，其中包含指定数量的日志块。Xv6 在事务提交时写入头部块，但在此之前不写入，并在将日志块复制到文件系统后将计数设置为零。因此，在事务中途崩溃将导致日志头部块中的计数为零；提交后崩溃将导致非零计数。

每个系统调用的代码都指明了必须相对于崩溃是原子的写入序列的开始和结束。为了允许不同进程并发执行文件系统操作，日志系统可以将多个系统调用的写入累积到一个事务中。因此，单个提交可能涉及多个完整系统调用的写入。为了避免跨事务拆分系统调用，日志系统仅在没有文件系统系统调用正在进行时才提交。

将多个事务一起提交的想法被称为 组提交。组提交减少了磁盘操作的数量，因为它将一次提交的固定成本分摊到多个操作上。组提交还同时向磁盘系统提交了更多的并发写入，也许允许磁盘在一次磁盘旋转期间写入所有这些写入。Xv6 的 virtio 驱动程序不支持这种批处理，但 xv6 的文件系统设计允许这样做。

Xv6 在磁盘上专门分配了固定数量的空间来保存日志。一个事务中系统调用写入的块总数必须能容纳在该空间中。这有两个后果。任何单个系统调用都不允许写入比日志空间中更多的不同块。这对于大多数系统调用来说不是问题，但其中有两个可能会写入许多块：write 和 unlink。一个大的文件写入可能会写入许多数据块和许多位图块以及一个 inode 块；取消链接一个大文件可能会写入许多位图块和一个 inode。Xv6 的写入系统调用将大的写入分解为多个较小的写入，这些写入适合日志，而 unlink 不会引起问题，因为实际上 xv6 文件系统只使用一个位图块。有限日志空间的另一个后果是日志系统不能允许系统调用开始，除非它确定该系统调用的写入将适合日志中剩余的空间。

8.6 代码：日志记录

在系统调用中典型使用日志的方式如下：

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

begin_op (kernel/log.c:127) 等待直到日志系统当前没有在提交，并且有足够的未保留日志空间来容纳此调用的写入。log.outstanding 计算已保留日志空间的系统调用数量；总保留空间是 log.outstanding 乘以 MAXOPBLOCKS。增加 log.outstanding 既保留了空间，又阻止了在此系统调用期间发生提交。代码保守地假设每个系统调用最多可能写入 MAXOPBLOCKS 个不同的块。

log_write (kernel/log.c:215) 充当 bwrite 的代理。它在内存中记录块的扇区号，在磁盘上的日志中为其保留一个槽位，并将缓冲区固定在块缓存中以防止块缓存将其逐出。块必须在提交之前一直留在缓存中：在此之前，缓存的副本是修改的唯一记录；在提交之后才能将其写入其在磁盘上的位置；并且同一事务中的其他读取必须看到修改。log_write 注意到一个块在单个事务中被多次写入时，会为该块在日志中分配相同的槽位。这种优化通常被称为 吸收。例如，包含多个文件的 inode 的磁盘块在一个事务中被多次写入是很常见的。通过将多个磁盘写入吸收为一个，文件系统可以节省日志空间并可以实现更好的性能，因为只需要将磁盘块的一个副本写入磁盘。

end_op (kernel/log.c:147) 首先减少未完成的系统调用计数。如果计数现在为零，它通过调用 commit() 来提交当前事务。此过程有四个阶段。write_log() (kernel/log.c:179) 将事务中

修改的每个块从缓冲区缓存复制到其在磁盘上日志中的槽位。write_head() (kernel/log.c:103) 将头部块写入磁盘：这是提交点，写入后发生崩溃将导致恢复从日志中重放事务的写入。install_trans (kernel/log.c:69) 从日志中读取每个块并将其写入文件系统中的适当位置。最后 end_op 用零计数写入日志头部；这必须在下一个事务开始写入日志块之前发生，这样崩溃就不会导致恢复使用一个事务的头部和后续事务的日志块。

recover_from_log (kernel/log.c:117) 从 initlog (kernel/log.c:55) 调用,而后者又从 fsinit (kernel/fs.c:42) 在引导期间第一个用户进程运行之前调用 (kernel/proc.c:535)。它读取日志头部，如果头部指示日志包含已提交的事务，则模仿 end_op 的操作。

filewrite (kernel/file.c:135) 中出现了一个使用日志的例子。事务看起来像这样：

```
begin_op();
ilock(f->ip);
r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

此代码被包装在一个循环中，该循环将大的写入分解为一次只有几个扇区的单个事务，以避免溢出日志。对 writei 的调用作为此事务的一部分写入许多块：文件的 inode、一个或多个位图块以及一些数据块。

8.7 代码：块分配器

文件和目录内容存储在磁盘块中，这些块必须从空闲池中分配。Xv6 的块分配器在磁盘上维护一个空闲位图，每个块一位。零位表示相应块是空闲的；一位表示它正在使用中。程序 mkfs 设置与引导扇区、超级块、日志块、inode 块和位图块相对应的位。

块分配器提供两个函数：balloc 分配一个新的磁盘块，而 bfree 释放一个块。balloc balloc 中的循环在 (kernel/fs.c:72) 处考虑每个块，从块 0 到 sb.size，即文件系统中的块数。它寻找一个位图位为零的块，表示它是空闲的。如果 balloc 找到这样的块，它会更新位图并返回该块。为提高效率，循环分为两部分。外层循环读取每个位图块。内层循环检查单个位图块中的所有 Bits-Per-Block（BPB）位。如果两个进程试图同时分配一个块，可能会发生竞争，这种情况由缓冲区缓存只允许一个进程一次使用任何一个位图块的事实来防止。

bfree (kernel/fs.c:92) 找到正确的位图块并清除正确的位。同样，由 bread 和 brelse 隐含的独占使用避免了显式锁定的需要。

与本章其余部分描述的大部分代码一样，balloc 和 bfree 必须在事务内部调用。

8.8 Inode 层

术语 inode 可以有两个相关的含义。它可以指包含文件大小和数据块编号列表的磁盘上数据结构。或者“inode”可以指内存中的 inode，它包含磁盘上 inode 的副本以及内核中所需的额外信息。

磁盘上的 inode 被打包在一个称为 inode 块的连续磁盘区域中。每个 inode 大小相同，因此给定一个数字 n，很容易在磁盘上找到第 n 个 inode。实际上，这个数字 n，称为 inode 号或 i-number，是实现中标识 inode 的方式。

磁盘上的 inode 由一个 struct dinode (kernel/fs.h:32) 定义。type 字段区分文件、目录和特殊文件（设备）。类型为零表示磁盘上的 inode 是空闲的。nlink 字段计算引用此 inode 的目录条目数，以便识别何时应释放磁盘上的 inode 及其数据块。size 字段记录文件内容的字节数。addrs 数组记录保存文件内容的磁盘块的块号。

内核将活动 inode 集保存在内存中一个名为 itable 的表中；struct inode (kernel/file.h:17) 是磁盘上 struct dinode 的内存副本。内核仅在有 C 指针引用该 inode 时才将其存储在内存中。ref 字段计算引用内存中 inode 的 C 指针数，如果引用计数降至零，内核会从内存中丢弃该 inode。iget 和 iput 函数获取和释放指向 inode 的指针，修改引用计数。指向 inode 的指针可以来自文件描述符、当前工作目录和临时内核代码，例如 exec。

xv6 的 inode 代码中有四种锁或类似锁的机制。itable.lock 保护了 inode 在 inode 表中最多出现一次的不变量，以及内存中 inode 的 ref 字段计算指向该 inode 的内存指针数的不变量。每个内存中的 inode 都有一个 lock 字段，其中包含一个休眠锁，确保对 inode 字段（如文件长度）以及 inode 的文件或目录内容块的独占访问。inode 的 ref，如果大于零，会导致系统在表中维护该 inode，并且不重用该表条目用于不同的 inode。最后，每个 inode 包含一个 nlink 字段（在磁盘上，如果在内存中则复制在内存中），计算引用文件的目录条目数；如果其链接计数大于零，xv6 不会释放 inode。

由 iget() 返回的 struct inode 指针保证在相应的 iput() 调用之前是有效的；inode 不会被删除，并且指针引用的内存不会被重用于不同的 inode。iget() 提供对 inode 的非独占访问，因此可以有许多指向同一 inode 的指针。文件系统代码的许多部分都依赖于 iget() 的这种行为，既用于持有对 inode 的长期引用（如打开的文件和当前目录），又用于在避免操作多个 inode 的代码中死锁的同时防止竞争（如路径名查找）。

iget 返回的 struct inode 可能没有任何有用的内容。为了确保它持有磁盘上 inode 的副本，代码必须调用 ilock。这将锁定 inode（以便没有其他进程可以 ilock 它）并从磁盘读取 inode，如果尚未读取的话。iunlock 释放 inode 上的锁。将 inode 指针的获取与锁定分开在某些情况下有助于避免死锁，例如在目录查找期间。多个进程可以持有由 iget 返回的指向 inode 的 C 指针，但一次只有一个进程可以锁定该 inode。

inode 表只存储那些内核代码或数据结构持有 C 指针的 inode。它的主要工作是同步多个进程的访问。inode 表也恰好缓存了频繁使用的 inode，但缓存是次要的；如果一个 inode 被频繁使用，缓冲区缓存很可能会将其保存在内存中。修改内存中 inode 的代码会使用 iupdate 将其写入磁盘。

8.9 **代码：** Inode

要分配一个新的 inode（例如，在创建文件时），xv6 调用 ialloc (kernel/fs.c:199)。ialloc 类似于 balloc：它一次一个块地遍历磁盘上的 inode 结构，寻找一个被标记为空闲的 inode。当它找到一个时，它通过向磁盘写入新的 type 来声明它，然后通过对 iget (kernel/fs.c:213) 的尾调用返回一个 inode 表条目。ialloc 的正确操作依赖于这样一个事实：一次只有一个进程可以持有对 bp 的引用：ialloc 可以确保没有其他进程同时看到该 inode 可用并试图声明它。

iget (kernel/fs.c:247) 在 inode 表中查找具有所需设备和 inode 号的活动条目（ip->ref > 0）。如果找到一个，它会返回对该 inode 的新引用 (kernel/fs.c:256-260)。当 iget 扫描时，它记录第一个空槽的位置 (kernel/fs.c:261-262)，如果需要分配表条目，它会使用该位置。

代码必须在使用 ilock 锁定 inode 之后才能读取或写入其元数据或内容。ilock (kernel/fs.c:293) 为此使用休眠锁。一旦 ilock 对 inode 具有独占访问权，它会根据需要从磁盘（更可能是缓冲区缓存）读取 inode。函数 iunlock (kernel/fs.c:321) 释放休眠锁，这可能会唤醒任何正在休眠的进程。

iput (kernel/fs.c:337) 通过递减引用计数 (kernel/fs.c:360) 来释放指向 inode 的 C 指针。如果这是最后一个引用，则 inode 在 inode 表中的槽位现在是空闲的，可以被重用于不同的 inode。

如果 iput 看到一个 inode 没有 C 指针引用，并且该 inode 没有链接到它（即没有出现在任何目录中），那么该 inode 及其数据块必须被释放。iput 调用 itrunc 将文件截断为零字节，释放数据块；将 inode 类型设置为 0（未分配）；并将 inode 写入磁盘 (kernel/fs.c:342)。

在 iput 释放 inode 的情况下，其锁定协议值得仔细研究。一个危险是，一个并发线程可能正在 ilock 中等待使用此 inode（例如，读取文件或列出目录），并且没有准备好发现该 inode 不再被分配。这不会发生，因为如果一个内存中的 inode 没有链接并且 ip->ref 为 1，则系统调用无法获取指向它的指针。这一个引用是调用 iput 的线程所拥有的引用。另一个主要危险是，并发调用 ialloc 可能会选择 iput 正在释放的同一个 inode。这只有在 iupdate 写入磁盘，使 inode 类型为零之后才会发生。这种竞争是良性的；分配线程会礼貌地等待获取 inode 的休眠锁，然后再读取或写入该 inode，此时 iput 已经完成了对它的操作。

iput() 可以写入磁盘。这意味着任何使用文件系统的系统调用都可能写入磁盘，因为该系统调用可能是最后一个持有文件引用的系统调用。即使像 read() 这样看起来是只读的调用，也可能最终调用 iput()。这反过来意味着，即使是只读的系统调用，如果它们使用文件系统，也必须包装在事务中。

iput() 和崩溃之间存在一个具有挑战性的交互。当文件的链接计数降至零时，iput() 不会立即截断文件，因为某个进程可能仍然持有对该 inode 的内存引用：一个进程可能仍在读写该文件，因为它成功地打开了它。但是，如果在最后一个进程关闭该文件的文件描述符之前发生崩溃，那么该文件将在磁盘上被标记为已分配，但没有目录条目指向它。

文件系统通过以下两种方式之一来处理这种情况。简单的解决方案是，在恢复时，重启后，文件系统扫描整个文件系统，查找那些被标记为已分配但没有目录条目指向它们的文件。如果存在任何这样的文件，那么它可以释放这些文件。

第二种解决方案不需要扫描文件系统。在这种解决方案中，文件系统在磁盘上记录（例如，在超级块中）链接计数降至零但引用计数不为零的文件的 inode 号。如果文件系统在引用计数达到 0 时删除该文件，那么它会通过从列表中删除该 inode 来更新磁盘上的列表。在恢复时，文件系统会释放列表中的任何文件。

Xv6 两种解决方案都没有实现，这意味着 inode 可能会在磁盘上被标记为已分配，即使它们已不再使用。这意味着随着时间的推移，xv6 可能会耗尽磁盘空间。

8.10 代码：Inode 内容

磁盘上的 inode 结构，struct dinode，包含一个大小和一个块号数组（见图 8.3）。inode 数据位于 dinode 的 addrs 数组中列出的块中。前 NDIRECT 个数据块列在数组的前 NDIRECT 个条目中；这些块被称为 直接块。接下来的 NINDIRECT 个数据块不是列在 inode 中，而是列在一个称为 间接块 的数据块中。addrs 数组中的最后一个条目给出了间接块的地址。因此，文件的前 12 kB（NDIRECT x BSIZE）字节可以从 inode 中列出的块加载，而接下

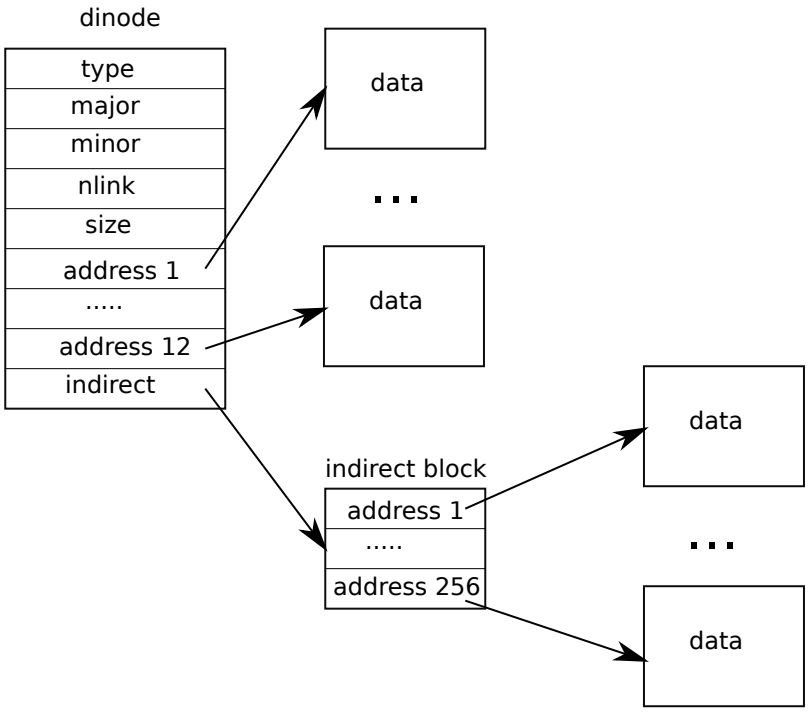

Figure 8.3: 磁盘上文件的表示。

来的 256 kB（NINDIRECT x BSIZE）字节只有在查询间接块后才能加载。这是一个很好的磁盘表示，但对客户端来说很复杂。函数 bmap 管理这种表示，以便更高级别的例程，如 readi 和 writei，我们稍后会看到，不需要管理这种复杂性。bmap 返回 inode ip 的第 bn 个数据块的磁盘块号。如果 ip 还没有这样的块，bmap 会分配一个。

函数 bmap (kernel/fs.c:383) 首先处理简单情况：前 NDIRECT 个块列在 inode 本身中 (kernel/fs.c:388-396)。接下来的 NINDIRECT 个块列在 ip->addrs[NDIRECT] 的间接块中。bmap 读取间接块 (kernel/fs.c:407) 然后从块内的正确位置读取一个块号 (kernel/fs.c:408)。如果块号超过 NDIRECT+NINDIRECT，bmap 会 panic；writei 包含了防止这种情况发生的检查 (kernel/fs.c:513)。

bmap 根据需要分配块。ip->addrs[] 或间接条目为零表示没有分配块。当 bmap 遇到零时，它会用新块的编号替换它们，这些块是按需分配的 (kernel/fs.c:389-390) (kernel/fs.c:401-402)。

itrunc 释放文件的块，将 inode 的大小重置为零。itrunc (kernel/fs.c:426) 首先释放直接块 (kernel/fs.c:432-437)，然后是间接块中列出的块 (kernel/fs.c:442-445)，最后是间接块本身 (kernel/fs.c:447-448)。

bmap 使得 readi 和 writei 很容易访问 inode 的数据。readi (kernel/fs.c:472) 首先确保偏移量和计数不会超出文件末尾。从文件末尾之后开始的读取会返回错误 (kernel/fs.c:477-478)，而从文件末尾开始或跨越文件末尾的读取会返回比请求的少的字节 (kernel/fs.c:479-480)。主循环处理文件的每个块，将数据从缓冲区复制到 dst (kernel/fs.c:482-494)。writei (kernel/fs.c:506) 与 readi 相同，有三个例外：从文件末尾开始或跨越文件末尾的写入会增长文件，直到最大文

件大小 (kernel/fs.c:513-514)；循环将数据复制到缓冲区而不是从中复制出来 (kernel/fs.c:522)；并且如果写入扩展了文件，writei 必须更新其大小 (kernel/fs.c:530-531)。

函数 stati (kernel/fs.c:458) 将 inode 元数据复制到 stat 结构中，该结构通过 stat 系统调用暴露给用户程序。

8.11 代码：目录层

目录在内部的实现方式与文件非常相似。它的 inode 类型为 T_DIR，其数据是一系列目录条目。每个条目都是一个 struct dirent (kernel/fs.h:56)，其中包含一个名称和一个 inode 号。名称最多为 DIRSIZ （14）个字符；如果更短，则以 NULL (0) 字节结尾。inode 号为零的目录条目是空闲的。

函数 dirlookup (kernel/fs.c:552) 在目录中搜索具有给定名称的条目。如果找到一个，它返回相应 inode 的指针，未锁定，并设置 *poff 为条目在目录中的字节偏移量，以备调用者希望编辑它。如果 dirlookup 找到具有正确名称的条目，它会更新 *poff 并返回一个通过 iget 获取的未锁定 inode。dirlookup 是 iget 返回未锁定 inode 的原因。调用者已锁定 dp，因此如果查找的是 .，即当前目录的别名，在返回之前尝试锁定 inode 会试图重新锁定 dp 并导致死锁。（存在涉及多个进程和 ..，即父目录的别名的更复杂的死锁场景；. 不是唯一的问题。）调用者可以解锁 dp 然后锁定 ip，确保一次只持有一个锁。

函数 dirlink (kernel/fs.c:580) 将一个具有给定名称和 inode 号的新目录条目写入目录 dp。如果名称已存在，dirlink 返回一个错误 (kernel/fs.c:586-590)。主循环读取目录条目，寻找一个未分配的条目。当找到一个时，它会提前停止循环 (kernel/fs.c:592-597)，并将 off 设置为可用条目的偏移量。否则，循环结束时 off 被设置为 dp->size。无论哪种方式，dirlink 然后通过在偏移量 off 处写入来向目录添加一个新条目 (kernel/fs.c:602-603)。

8.12 代码：路径名

路径名查找涉及一系列对 dirlookup 的调用，每个路径组件一次。namei (kernel/fs.c:687) 评估 path 并返回相应的 inode。函数 nameiparent 是一个变体：它在最后一个元素之前停止，返回父目录的 inode 并将最后一个元素复制到 name 中。两者都调用通用函数 namex 来完成实际工作。

namex (kernel/fs.c:652) 首先决定路径评估从哪里开始。如果路径以斜杠开头，评估从根目录开始；否则，从当前目录开始 (kernel/fs.c:656-659)。然后它使用 skipelem 依次考虑路径的每个元素 (kernel/fs.c:661)。循环的每次迭代都必须在当前 inode ip 中查找 name。迭代开始时锁定 ip 并检查它是否是一个目录。如果不是，查找失败 (kernel/fs.c:662-666)。（锁定 ip 是必要的，不是因为 ip->type 会发生变化——它不会——而是因为直到 ilock 运行，ip->type 不保证已从磁盘加载。）如果调用是 nameiparent 并且这是最后一个路径元素，则循环提前停止，根据 nameiparent 的定义；最后一个路径元素已经复制到 name 中，所以 namex 只需要返回未锁定的 ip (kernel/fs.c:667-671)。最后，循环使用 dirlookup 查找路径元素并设置 ip = next 为下一次迭代做准备 (kernel/fs.c:672-677)。当循环用完路径元素时，它返回 ip。

过程 namex 可能需要很长时间才能完成：它可能涉及多次磁盘操作来读取路径中遍历的目录的 inode 和目录块（如果它们不在缓冲区缓存中）。Xv6 经过精心设计，以便如果一

个内核线程的 namex 调用被磁盘 I/O 阻塞，另一个查找不同路径名的内核线程可以并发进行。namex 分别锁定路径中的每个目录，以便在不同目录中的查找可以并行进行。

这种并发性带来了一些挑战。例如，当一个内核线程正在查找一个路径名时，另一个内核线程可能正在通过取消链接一个目录来更改目录树。一个潜在的风险是，查找可能正在搜索一个已被另一个内核线程删除的目录，并且其块已被重用于另一个目录或文件。

Xv6 避免了这种竞争。例如，在 namex 中执行 dirlookup 时，查找线程持有目录的锁，并且 dirlookup 返回一个使用 iget 获得的 inode。iget 增加了 inode 的引用计数。只有在从 dirlookup 接收到 inode 之后，namex 才会释放目录上的锁。现在另一个线程可能会从目录中取消链接该 inode，但 xv6 不会删除该 inode，因为该 inode 的引用计数仍然大于零。

另一个风险是死锁。例如，当查找 "." 时，next 指向与 ip 相同的 inode。在释放 ip 上的锁之前锁定 next 将导致死锁。为了避免这种死锁，namex 在获取 next 上的锁之前解锁目录。这里我们再次看到为什么 iget 和 ilock 之间的分离很重要。

8.13 文件描述符层

Unix 接口的一个很酷的方面是，Unix 中的大多数资源都表示为文件，包括设备（如控制台）、管道，当然还有真正的文件。文件描述符层就是实现这种统一性的层。

正如我们在第 1 章中看到的，Xv6 为每个进程提供了自己的打开文件表，或文件描述符。每个打开的文件都由一个 struct file (kernel/file.h:1) 表示，它是一个 inode 或管道的包装器，外加一个 I/O 偏移量。每次调用 open 都会创建一个新的打开文件（一个新的 struct file）：如果多个进程独立打开同一个文件，不同的实例将有不同的 I/O 偏移量。另一方面，一个单独的打开文件（同一个 struct file）可以多次出现在一个进程的文件表中，也可以出现在多个进程的文件表中。如果一个进程使用 open 打开文件，然后使用 dup 创建别名，或使用 fork 与子进程共享，就会发生这种情况。引用计数跟踪对特定打开文件的引用次数。一个文件可以以只读、只写或读写方式打开。readable 和 writable 字段跟踪这一点。

系统中所有打开的文件都保存在一个全局文件表中，即 ftable。文件表有分配文件的函数（ filealloc ）、创建重复引用的函数（filedup）、释放引用的函数（ fileclose ），以及读写数据的函数（ fileread 和 filewrite ）。

前三个遵循了现在熟悉的形式。 filealloc (kernel/file.c:30) 扫描文件表，查找一个未被引用的文件（f->ref == 0）并返回一个新的引用； filedup (kernel/file.c:48) 增加引用计数；而 fileclose (kernel/file.c:60) 减少它。当一个文件的引用计数达到零时， fileclose 根据类型释放底层的管道或 inode。

函数 filestat 、 fileread 和 filewrite 实现了对文件的 stat、read 和 write 操作。 filestat (kernel/file.c:88) 只允许在 inode 上调用，并调用 stati。 fileread 和 filewrite 检查操作是否被打开模式允许，然后将调用传递给管道或 inode 的实现。如果文件代表一个 inode， fileread 和 filewrite 使用 I/O 偏移量作为操作的偏移量，然后将其推进 (kernel/file.c:122-123) (kernel/file.c:153-154)。管道没有偏移量的概念。回想一下，inode 函数要求调用者处理锁定 (kernel/file.c:94-96) (kernel/file.c:121-124) (kernel/file.c:163-166)。inode 锁定的一个方便的副作用是读写偏移量是原子更新的，因此多个进程同时写入同一个文件不会覆盖彼此的数据，尽管它们的写入最终可能会交错。

8.14 代码：系统调用

有了底层提供的函数，大多数系统调用的实现都是微不足道的（见 (kernel/sysfile.c)）。有几个调用值得仔细研究。

函数 sys_link 和 sys_unlink 编辑目录，创建或删除对 inode 的引用。它们是使用事务强大功能的另一个好例子。sys_link (kernel/sysfile.c:124) 首先获取其参数，两个字符串 old 和 new (kernel/sysfile.c:129)。假设 old 存在且不是目录 (kernel/sysfile.c:133-136)，sys_link 增加其 ip->nlink 计数。然后 sys_link 调用 nameiparent 来查找 new 的父目录和最后一个路径元素 (kernel/sysfile.c:149) 并创建一个指向 old 的 inode 的新目录条目 (kernel/sysfile.c:152)。新的父目录必须存在并且与现有 inode 在同一设备上：inode 号仅在单个磁盘上有唯一含义。如果发生此类错误，sys_link 必须返回并减少 ip->nlink。

事务简化了实现，因为它需要更新多个磁盘块，但我们不必担心我们执行它们的顺序。它们要么全部成功，要么全部失败。例如，没有事务，在创建链接之前更新 ip->nlink，会使文件系统暂时处于不安全状态，而中间的崩溃可能会导致混乱。有了事务，我们就不必担心这个问题了。

sys_link 为现有 inode 创建一个新名称。函数 create (kernel/sysfile.c:246) 为新 inode 创建一个新名称。它是三个文件创建系统调用的泛化：带有 O_CREATE 标志的 open 创建一个新的普通文件，mkdir 创建一个新目录，而 mkdev 创建一个新的设备文件。像 sys_link 一样，create 首先调用 nameiparent 来获取父目录的 inode。然后它调用 dirlookup 来检查名称是否已存在 (kernel/sysfile.c:256)。如果名称确实存在，create 的行为取决于它被用于哪个系统调用：open 的语义与 mkdir 和 mkdev 不同。如果 create 代表 open 被使用（type == T_FILE）并且存在的名称本身是一个常规文件，那么 open 将其视为成功，所以 create 也这样做 (kernel/sysfile.c:260)。否则，这是一个错误 (kernel/sysfile.c:261-262)。如果名称尚不存在，create 现在用 ialloc 分配一个新的 inode (kernel/sysfile.c:265)。如果新的 inode 是一个目录，create 用 . 和 .. 条目初始化它。最后，既然数据已正确初始化，create 可以将其链接到父目录中 (kernel/sysfile.c:278)。create，像 sys_link 一样，同时持有两个 inode 锁：ip 和 dp。不存在死锁的可能性，因为 inode ip 是新分配的：系统中的没有其他进程会持有 ip 的锁，然后尝试锁定 dp。

使用 create，很容易实现 sys_open、sys_mkdir 和 sys_mknod。sys_open (kernel/sysfile.c:305) 是最复杂的，因为创建一个新文件只是它能做的一小部分。如果 open 被传递 O_CREATE 标志，它会调用 create (kernel/sysfile.c:320)。否则，它会调用 namei (kernel/sysfile.c:326)。create 返回一个锁定的 inode，但 namei 不返回，所以 sys_open 必须自己锁定 inode。这提供了一个方便的地方来检查目录是否只为读取而打开，而不是写入。假设 inode 以某种方式获得，sys_open 分配一个文件和一个文件描述符 (kernel/sysfile.c:344) 然后填充该文件 (kernel/sysfile.c:356-361)。请注意，没有其他进程可以访问部分初始化的文件，因为它只在当前进程的表中。

第 7 章在我们甚至还没有文件系统之前就研究了管道的实现。函数 sys_pipe 通过提供一种创建管道对的方法，将该实现与文件系统连接起来。它的参数是一个指向两个整数空间的指针，它将在那里记录两个新的文件描述符。然后它分配管道并安装文件描述符。

8.15 现实世界

现实世界操作系统中的缓冲区缓存明显比 xv6 的复杂，但它服务于相同的两个目的：缓存和同步对磁盘的访问。xv6 的缓冲区缓存，像 V6 的一样，使用简单的最近最少使用 (LRU) 驱逐策略；还有许多更复杂的策略可以实现，每种策略对某些工作负载有利，而对其他工作负载则不那么有利。一个更高效的 LRU 缓存将消除链表，而是使用哈希表进行查找，使用堆进行 LRU 驱逐。现代缓冲区缓存通常与虚拟内存系统集成，以支持内存映射文件。

Xv6 的日志系统效率低下。提交不能与文件系统系统调用并发进行。系统记录整个块，即使块中只有几个字节被更改。它执行同步日志写入，一次一个块，每次写入都可能需要整个磁盘旋转时间。真正的日志系统解决了所有这些问题。

日志记录不是提供崩溃恢复的唯一方法。早期的文件系统在重启期间使用清道夫（例如，UNIX fsck 程序）来检查每个文件和目录以及块和 inode 空闲列表，查找并解决不一致性。对于大型文件系统，清道夫可能需要数小时，并且在某些情况下，不可能以使原始系统调用具有原子性的方式解决不一致性。从日志恢复要快得多，并使系统调用在崩溃面前具有原子性。

Xv6 使用与早期 UNIX 相同的基本磁盘布局的 inode 和目录；这种方案多年来一直非常持久。BSD 的 UFS/FFS 和 Linux 的 ext2/ext3 使用基本相同的数据结构。文件系统布局中效率最低的部分是目录，每次查找都需要对所有磁盘块进行线性扫描。当目录只有几个磁盘块时，这是合理的，但对于包含许多文件的目录来说，这是昂贵的。微软 Windows 的 NTFS、macOS 的 HFS 和 Solaris 的 ZFS，仅举几例，将目录实现为磁盘上块的平衡树。这很复杂，但保证了对数时间的目录查找。

Xv6 对磁盘故障很天真：如果磁盘操作失败，xv6 会 panic。这是否合理取决于硬件：如果操作系统位于使用冗余来掩盖磁盘故障的特殊硬件之上，也许操作系统很少看到故障，以至于 panic 是可以接受的。另一方面，使用普通磁盘的操作系统应该预料到故障并更优雅地处理它们，以便一个文件中块的丢失不会影响文件系统其余部分的使用。

Xv6 要求文件系统适合一个磁盘设备并且大小不变。随着大型数据库和多媒体文件对存储需求的不断提高，操作系统正在开发消除“每个文件系统一个磁盘”瓶颈的方法。基本方法是将许多磁盘组合成一个逻辑磁盘。像 RAID 这样的硬件解决方案仍然是最流行的，但目前的趋势是尽可能多地在软件中实现这种逻辑。这些软件实现通常允许丰富的功能，如通过动态添加或删除磁盘来增长或缩小逻辑设备。当然，一个可以动态增长或缩小的存储层需要一个可以做同样事情的文件系统：xv6 使用的固定大小的 inode 块数组在这样的环境中效果不佳。将磁盘管理与文件系统分开可能是最清晰的设计，但两者之间复杂的接口导致一些系统，如 Sun 的 ZFS，将它们结合起来。

Xv6 的文件系统缺少现代文件系统的许多其他功能；例如，它缺少对快照和增量备份的支持。

现代 Unix 系统允许使用与磁盘存储相同的系统调用访问多种资源：命名管道、网络连接、远程访问的网络文件系统，以及监控和控制接口，如 /proc。而不是 xv6 在 fileread 和 filewrite 中的 if 语句，这些系统通常为每个打开的文件提供一个函数指针表，每个操作一个，并调用函数指针来调用该 inode 对该调用的实现。网络文件系统和用户级文件系统提供将这些调用转换为网络 RPC 并在返回前等待响应的函数。

8.16 练习

1. 为什么在 balloc 中 panic？xv6 能恢复吗?
2. 为什么在 ialloc 中 panic？xv6 能恢复吗?
3. 为什么 filealloc 在文件用完时不会 panic？为什么这种情况更常见，因此值得处理?
4. 假设与 ip 对应的文件在 sys_link 调用 iunlock(ip) 和 dirlink 之间被另一个进程取消链接。链接会正确创建吗？为什么或为什么不?
5. create 进行了四个函数调用（一个到 ialloc 和三个到 dirlink），它要求这些调用必须成功。如果任何一个不成功，create 会调用 panic。为什么这是可以接受的？为什么这四个调用都不会失败?
6. sys_chdir 在 iput(cp->cwd) 之前调用 iunlock(ip)，而 iput(cp->cwd) 可能会尝试锁定 cp->cwd，然而推迟 iunlock(ip) 到 iput 之后不会导致死锁。为什么不呢?
7. 实现 lseek 系统调用。支持 lseek 还需要你修改 filewrite 以在 lseek 将 off 设置到 f->ip->size 之外时用零填充文件中的空洞。
8. 向 open 添加 O_TRUNC 和 O_APPEND，以便 > 和 >> 运算符在 shell 中工作。
9. 修改文件系统以支持符号链接。
10. 修改文件系统以支持命名管道。
11. 修改文件和虚拟机系统以支持内存映射文件。

Chapter 9

再谈并发

在内核设计中，同时获得良好的并行性能、并发下的正确性以及易于理解的代码是一个巨大的挑战。直接使用锁是通往正确性的最佳途径，但这并非总是可行。本章重点介绍 xv6 被迫以复杂方式使用锁的例子，以及 xv6 使用类似锁的技术但并非锁的例子。

9.1 锁模式

缓存项的加锁通常是一个挑战。例如，文件系统的块缓存 (kernel/bio.c:26) 存储了多达 NBUF 个磁盘块的副本。至关重要的是，一个给定的磁盘块在缓存中最多只能有一个副本；否则，不同的进程可能会对本应是同一个块的不同副本进行冲突的更改。每个缓存的块都存储在一个 struct buf (kernel/buf.h:1) 中。一个 struct buf 有一个锁字段，这有助于确保在同一时间只有一个进程使用一个给定的磁盘块。然而，这个锁是不够的：如果一个块根本不存在于缓存中，而两个进程想同时使用它呢？由于块尚未被缓存，所以没有 struct buf，因此也就没有东西可以锁定。Xv6 通过将一个额外的锁 (bcache.lock) 与缓存块的身份集合相关联来处理这种情况。需要检查一个块是否被缓存的代码 (例如，bget (kernel/bio.c:59))，或者改变缓存块集合的代码，都必须持有 bcache.lock；在该代码找到它需要的块和 struct buf 之后，它可以释放 bcache.lock，只锁定特定的块。这是一个常见的模式：一个锁用于项目集合，每个项目再加一个锁。

通常，获取锁的函数也会释放它。但更精确的看法是，锁是在一个必须以原子方式出现的操作序列开始时获取的，并在该序列结束时释放。如果序列在不同的函数、不同的线程或不同的 CPU 上开始和结束，那么锁的获取和释放也必须这样做。锁的功能是强制其他使用者等待，而不是将一段数据固定在某个特定的代理上。一个例子是 yield (kernel/proc.c:512) 中的 acquire，它在调度器线程中被释放，而不是在获取它的进程中。另一个例子是 ilock (kernel/fs.c:293) 中的 acquiresleep；这段代码在读取磁盘时经常会休眠；它可能会在另一个 CPU 上被唤醒，这意味着锁可能会在不同的 CPU 上被获取和释放。

释放一个受嵌入在对象中的锁保护的对象是一件微妙的事情，因为拥有锁并不足以保证释放是正确的。问题出现在当其他线程在 acquire 中等待使用该对象时；释放该对象会隐式地释放嵌入的锁，这将导致等待的线程发生故障。一个解决方案是跟踪该对象存在多少引用，以便仅在最后一个引用消失时才释放它。参见 pipeclose (kernel/pipe.c:59) 的例子；pi->readopen 和 pi->writeopen 跟踪管道是否有文件描述符引用它。

通常我们看到锁围绕着对一组相关项目的读写序列；锁确保其他线程只看到完整的更新序列（只要它们也加锁）。那么，当更新只是对单个共享变量的简单写入时，情况又如何呢？例如，setkilled 和 killed (kernel/proc.c:619) 在它们对 p->killed 的简单使用周围加锁。如果没有锁，一个线程可能在另一个线程读取 p->killed 的同时写入它。这是一个竞争，C 语言规范说竞争会产生未定义行为，这意味着程序可能会崩溃或产生不正确的结果[1]。锁可以防止竞争并避免未定义行为。

竞争可能破坏程序的一个原因是，如果没有锁或等效的构造，编译器可能会生成与原始 C 代码非常不同的读写内存的机器代码。例如，调用 killed 的线程的机器代码可能会将 p->killed 复制到一个寄存器中，并且只读取那个缓存的值；这意味着该线程可能永远看不到对 p->killed 的任何写入。锁可以防止这种缓存。

9.2 类锁模式

在许多地方，xv6 使用引用计数或标志以类似锁的方式来指示一个对象已被分配，不应被释放或重用。进程的 p->state 就是这样起作用的，file、inode 和 buf 结构中的引用计数也是如此。虽然在每种情况下都有一个锁来保护标志或引用计数，但正是后者防止了对象被过早释放。

文件系统使用 struct inode 引用计数作为一种可以被多个进程持有的共享锁，以避免如果代码使用普通锁会发生的死锁。例如，namex (kernel/fs.c:652) 中的循环会依次锁定每个路径名组件所命名的目录。然而，namex 必须在循环结束时释放每个锁，因为如果它持有多个锁，当路径名包含一个点（例如，a/./b）时，它可能会与自身发生死锁。它也可能与涉及该目录和.. 的并发查找发生死锁。正如第 8章所解释的，解决方案是让循环将目录 inode 带到下一次迭代，其引用计数增加，但未被锁定。

一些数据项在不同时间受不同机制的保护，并且有时可能通过 xv6 代码的结构隐式地免受并发访问，而不是通过显式锁。例如，当一个物理页空闲时，它受 kmem.lock (kernel/kalloc.c:24) 保护。如果该页随后被分配为一个管道 (kernel/pipe.c:23)，它将受到一个不同的锁（嵌入的 pi->lock）的保护。如果该页被重新分配给一个新进程的用户内存，它根本不受锁的保护。相反，分配器不会将该页分配给任何其他进程（直到它被释放）这一事实保护了它免受并发访问。一个新进程内存的所有权是复杂的：首先父进程在 fork 中分配和操作它，然后子进程使用它，并且（在子进程退出后）父进程再次拥有该内存并将其传递给 kfree。这里有两个教训：一个数据对象在其生命周期的不同点可能以不同的方式受到并发保护，并且这种保护可能采取隐式结构的形式，而不是显式锁。

最后一个类锁的例子是在调用 mycpu() (kernel/proc.c:83) 前后需要禁用中断。禁用中断使得调用代码相对于可能强制进行上下文切换的定时器中断是原子的，从而将进程移动到不同的 CPU。

9.3 完全没有锁

有几个地方 xv6 在完全没有锁的情况下共享可变数据。一个是在自旋锁的实现中，尽管可以认为 RISC-V 原子指令依赖于硬件中实现的锁。另一个是 main.c (kernel/main.c:7) 中的

[1] "Threads and data races" in https://en.cppreference.com/w/c/language/memory_model

started 变量，用于防止其他 CPU 在 CPU 0 完成 xv6 初始化之前运行；volatile 确保编译器实际生成加载和存储指令。

Xv6 包含这样的情况：一个 CPU 或线程写入一些数据，而另一个 CPU 或线程读取这些数据，但没有专门用于保护这些数据的特定锁。例如，在 fork 中，父进程写入子进程的用户内存页，而子进程（一个不同的线程，可能在不同的 CPU 上）读取这些页；没有锁明确保护这些页。这严格来说不是一个锁问题，因为子进程在父进程完成写入之前不会开始执行。这是一个潜在的内存排序问题（参见第 6章），因为没有内存屏障，就没有理由期望一个 CPU 能看到另一个 CPU 的写入。然而，由于父进程释放锁，并且子进程在启动时获取锁，acquire 和 release 中的内存屏障确保了子进程的 CPU 能看到父进程的写入。

9.4 并行性

加锁主要是为了正确性而抑制并行性。因为性能也很重要，内核设计者通常必须考虑如何使用锁，既能实现正确性又能允许并行性。虽然 xv6 并非系统地为高性能而设计，但考虑哪些 xv6 操作可以并行执行，以及哪些可能在锁上发生冲突，仍然是值得的。

xv6 中的管道是一个相当好的并行性例子。每个管道都有自己的锁，因此不同的进程可以在不同的 CPU 上并行地读写不同的管道。然而，对于一个给定的管道，写入者和读取者必须等待对方释放锁；它们不能同时读/写同一个管道。同样，从空管道读取（或向满管道写入）也必须阻塞，但这并非由于锁方案。

上下文切换是一个更复杂的例子。两个内核线程，每个都在自己的 CPU 上执行，可以同时调用 yield、sched 和 swtch，并且这些调用将并行执行。每个线程都持有一个锁，但它们是不同的锁，所以它们不必等待对方。然而，一旦进入 scheduler，两个 CPU 在搜索进程表中寻找一个 RUNNABLE 的进程时，可能会在锁上发生冲突。也就是说，xv6 很可能在上下文切换期间从多个 CPU 中获得性能提升，但可能没有它所能达到的那么多。

另一个例子是来自不同 CPU 上不同进程的并发 fork 调用。这些调用可能需要相互等待 pid_lock 和 kmem.lock，以及为了在进程表中搜索一个 UNUSED 进程所需的每个进程的锁。另一方面，两个分叉的进程可以完全并行地复制用户内存页和格式化页表页。

在上述每个例子中，锁方案在某些情况下牺牲了并行性能。在每种情况下，都有可能使用更精巧的设计来获得更多的并行性。这是否值得取决于细节：相关操作被调用的频率，代码持有有争议的锁的时间，可能同时运行冲突操作的 CPU 数量，以及代码的其他部分是否是更严格的瓶颈。很难猜测一个给定的锁方案是否会导致性能问题，或者一个新的设计是否明显更好，因此通常需要对现实的工作负载进行测量。

9.5 练习

1. 修改 xv6 的管道实现，允许对同一个管道的读和写在不同的 CPU 上并行进行。
2. 修改 xv6 的 scheduler()，以减少不同 CPU 同时寻找可运行进程时的锁竞争。
3. 消除 xv6 的 fork() 中的一些串行化。

Chapter 10

总结

本文通过逐行研究一个操作系统 xv6，介绍了操作系统的主要思想。一些代码行体现了主要思想的精髓（例如，上下文切换、用户/内核边界、锁等），每一行都很重要；其他代码行则说明了如何实现一个特定的操作系统思想，并且可以很容易地以不同的方式完成（例如，更好的调度算法、更好的磁盘数据结构来表示文件、更好的日志记录以允许并发事务等）。所有的思想都是在一个特定的、非常成功的系统调用接口（Unix 接口）的背景下进行阐述的，但这些思想也适用于其他操作系统的设计。

Bibliography

[1] Linux common vulnerabilities and exposures (CVEs). https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux.

[2] The RISC-V instruction set manual Volume I: unprivileged specification ISA. https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view?usp=drive_link, 2024.

[3] The RISC-V instruction set manual Volume II: privileged specification. https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link, 2024.

[4] Hans-J Boehm. Threads cannot be implemented as a library. ACM PLDI Conference, 2005.

[5] Edsger Dijkstra. Cooperating sequential processes. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html, 1965.

[6] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming, Revised Reprint. 2012.

[7] Brian W. Kernighan. The C Programming Language. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, page 207–220, 2009.

[9] Donald Knuth. Fundamental Algorithms. The Art of Computer Programming. (Second ed.), volume 1. 1997.

[10] L Lamport. A new solution of dijkstra's concurrent programming problem. Communications of the ACM, 1974.

[11] John Lions. Commentary on UNIX 6th Edition. Peer to Peer Communications, 2000.

[12] Paul E. Mckenney, Silas Boyd-wickizer, and Jonathan Walpole. RCU usage in the linux kernel: One decade later, 2013.

[13] Martin Michael and Daniel Durich. The NS16550A: UART design and application considerations. http://bitsavers.trailing-edge.com/components/national/_appNotes/AN-0491.pdf, 1987.

[14] Aleph One. Smashing the stack for fun and profit. http://phrack.org/issues/49/14.html#article.

[15] David Patterson and Andrew Waterman. The RISC-V Reader: an open architecture Atlas. Strawberry Canyon, 2017.

[16] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey. Plan 9, a distributed system. In In Proceedings of the Spring 1991 EurOpen Conference, pages 43–50, 1991.

[17] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. Commun. ACM, 17(7):365–375, July 1974.

Index