xv6: 一个简单的、类 Unix 的教学操作系统

Russ Cox Frans Kaashoek Robert Morris

2020 年 8 月 31 日

目录

前言与致谢

这是一份为操作系统课程准备的草稿文本。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 是基于 Dennis Ritchie 和 Ken Thompson 的 Unix 版本 6（v6）[14]建模的。xv6 大致遵循 v6 的结构和风格，但使用 ANSI C [6]为多核 RISC-V [12]实现。

本文应与 xv6 的源代码一起阅读，这种方法受到 John Lions 的《UNIX 第 6 版注释》[9]的启发。有关 v6 和 xv6 的在线资源，包括使用 xv6 的几个实验作业，请参见 https://pdos.csail.mit.edu/6.S081。

我们已在麻省理工学院的操作系统课程 6.828 和 6.S081 中使用了此文本。我们感谢这些课程的教师、助教和学生，他们都直接或间接地为 xv6 做出了贡献。特别感谢 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错误或提出改进建议的人：Abutalib Aghayev、Sebastian Boehm、Anton Burtsev、Raphael Carvalho、Tej Chajed、Rasit Eskicioglu、Color Fuzzy、Giuseppe、Tao Guo、Naoki Hayama、Robert Hilderman、Wolfgang Keller、Austin Liew、Pavan Maddamsetti、Jacek Masiulaniec、Michael McConville、m3hm00d、miguelgvieira、Mark Morrissey、Harry Pan、Askar Safin、Salman Shah、Adeodato Simó、Ruslan Savchenko、Pawel Szczurko、Warren Toomey、tyfkda、tzerbib、Xi Wang 和 Zou Chang Wei。

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）。

第一章

操作系统接口

操作系统的任务是在多个程序之间共享计算机资源，并提供比单独硬件支持更有用的服务集。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，使它们能够（或看起来）同时运行。最后，操作系统为程序提供受控的交互方式，以便它们可以共享数据或协同工作。

操作系统通过接口向用户程序提供服务。设计一个好的接口是非常困难的。一方面，我们希望接口简单且狭窄，因为这样更容易实现正确。另一方面，我们可能会倾向于为应用程序提供许多复杂的功能。解决这种矛盾的关键在于设计依赖于少数机制的接口，这些机制可以组合起来提供广泛的通用性。

本书使用一个单一的操作系统作为具体示例来说明操作系统概念。该操作系统，xv6，提供了由 Ken Thompson 和 Dennis Ritchie 的 Unix 操作系统[14]引入的基本接口，并模仿了 Unix 的内部设计。Unix 提供了一个狭窄的接口，其机制结合得很好，提供了令人惊讶的通用性。这个接口非常成功，以至于现代操作系统——BSD、Linux、Mac OS X、Solaris，甚至在较小程度上，Microsoft Windows——都有类似 Unix 的接口。

理解 xv6 是理解这些系统及许多其他系统的良好起点。

如图 1.1 所示，xv6 采用了传统的内核形式，即一个为运行中的程序提供服务的特殊程序。每个运行中的程序，称为进程，都拥有包含指令、数据和栈的内存。指令实现了程序的计算。数据是计算所作用的变量。栈则组织程序的过程调用。

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，进程在用户空间和内核空间之间交替执行。

内核利用 CPU 提供的硬件保护机制来确保每个

[1] 本文通常使用术语 CPU（中央处理单元的缩写）来指代执行计算的硬件元素。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。

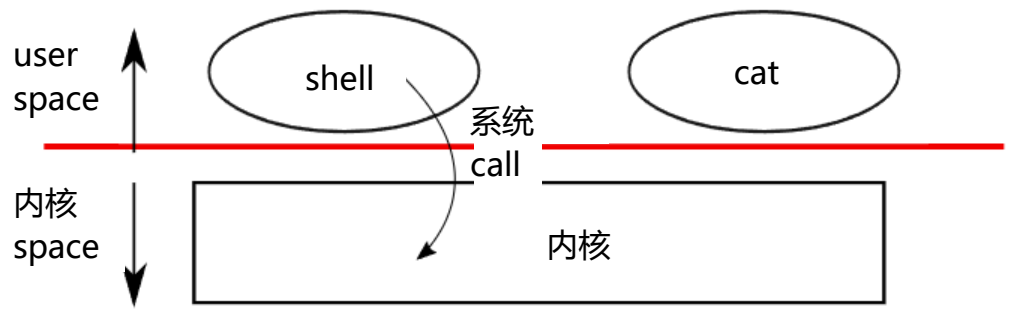

图 1.1：一个内核和两个用户进程。

在用户空间执行的进程只能访问自己的内存。内核执行时具有实现这些保护所需的硬件权限；用户程序执行时没有这些权限。当用户程序调用系统调用时，硬件会提升权限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所见的接口。xv6 内核提供了传统 Unix 内核所提供服务及系统调用的一个子集。

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——并通过代码片段和讨论来说明它们，展示 Unix 的命令行用户界面 shell 如何使用这些服务。shell 对系统调用的使用展示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：shell 并没有什么特别之处。这也意味着 shell 很容易被替换；因此，现代 Unix 系统有多种 shell 可供选择，每种 shell 都有其自己的用户界面和脚本功能。xv6 shell 是 Unix Bourne shell 精髓的简单实现。它的实现可以在 (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状态组成。Xv6 分时共享进程：它在等待执行的进程集合中透明地切换可用的 CPU。当一个进程不执行时，xv6 保存其 CPU 寄存器，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 系统调用来创建一个新进程。Fork 创建一个新进程，称为子进程，其内存内容与调用进程（称为父进程）完全相同。Fork 在父进程和子进程中都会返回。在父进程中，fork 返回子进程的 PID；在子进程中，fork 返回零。例如，考虑以下用 C 编程语言编写的程序片段[6]：

```
int pid = fork();
if(pid > 0){ printf("parent: child=%d\n", pid);
```

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()。无返回值。
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。返回 0，或-1 表示错误。
int getpid()	返回当前进程的 PID。
int sleep(int n)	暂停 n 个时钟滴答。
int exec(char *file, char *argv[])	加载文件并使用参数执行它；仅在出错时返回。char *sbrk(int n) 将进程的内存增加 n 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 表示读/写；返回一个 fd（文件描述符）。
int write(int fd, char *buf, int n)	将 buf 中的 n 个字节写入文件描述符 fd；返回 n。
int read(int fd, char *buf, int n)	将 n 个字节读入 buf；返回读取的字节数；如果到达文件末尾则返回 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，指向与 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 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 stat(char *file, struct stat *st)	将命名文件的信息放入 *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。

图 1.2：Xv6 系统调用。除非另有说明，这些调用在无错误时返回 0，有错误时返回-1。

```
pid = wait((int *) 0); printf("子进程 %d 已完成
\n", pid); } else if(pid == 0){ printf("子进程:
退出\n"); exit(0); } else { printf("fork 错误
\n"); }
```

exit 系统调用导致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已退出（或被杀死）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；如果调用者的任何子进程尚未退出，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，它可以传递一个 0 地址给 wait。

在示例中，输出行

```
parent: child=1234
child: 退出
```

可能会以任意顺序出现，这取决于父进程或子进程谁先到达其 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 已完成
```

虽然子进程最初拥有与父进程相同的内存内容，但父进程和子进程在不同的内存和不同的寄存器中执行：在一个进程中修改变量不会影响另一个进程。例如，当父进程将 wait 的返回值存储到 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 系统调用将调用进程的内存替换为从文件系统中存储的文件加载的新内存映像。文件必须具有特定的格式，指定文件的哪一部分包含指令，哪一部分是数据，从哪条指令开始执行等。xv6 使用 ELF 格式，第 3 章将更详细地讨论该格式。当 exec 成功时，它不会返回到调用程序；相反，从文件加载的指令从 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");
```

此片段将调用程序替换为运行参数列表 `echo hello` 的 `/bin/echo` 程序实例。大多数程序会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 使用上述调用来代表用户运行程序。shell 的主要结构很简单；参见 main (user/sh.c:145)。主循环使用 getcmd 从用户读取一行输入。然后它调用 fork，创建 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 输入了“echo hello”，runcmd 将会以“echo hello”作为参数被调用。runcmd (user/sh.c:58) 运行实际的命令。对于“echo hello”，它会调用 exec (user/sh.c:78)。如果 exec 成功，子进程将执行 echo 的指令而不是 runcmd。在某个时刻，echo 将调用 exit，这将导致父进程从 main (user/sh.c:145) 中的 wait 返回。

你可能会好奇为什么 fork 和 exec 没有合并成一个调用；我们稍后会看到，shell 在实现 I/O 重定向时利用了这种分离。为了避免创建重复进程然后立即替换它（通过 exec）的浪费，操作系统内核通过使用诸如写时复制（参见第 4.6 节）等虚拟内存技术来优化 fork 的实现。

Xv6 隐式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，exec 分配足够的内存来保存可执行文件

文件。一个在运行时需要更多内存（可能是为了 malloc）的进程可以调用 sbrk(n)来将其数据内存增加 n 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，代表一个由内核管理的对象，进程可以从中读取或写入。进程可以通过打开文件、目录或设备，或创建管道，或复制现有描述符来获取文件描述符。为了简单起见，我们通常将文件描述符所指的对象称为“文件”；文件描述符接口抽象了文件、管道和设备之间的差异，使它们看起来都像字节流。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 利用这一惯例来实现 I/O 重定向和管道。shell 确保它始终有三个文件描述符处于打开状态（user/sh.c:151），默认情况下这些文件描述符用于控制台。

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)从文件描述符 fd 中读取最多 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，然后将该偏移量向前推进读取的字节数：后续的 read 将返回第一次 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束。

调用 `write(fd, buf, n)` 从 `buf` 向文件描述符 `fd` 写入 `n` 字节，并返回写入的字节数。只有在发生错误时，写入的字节数才会少于 `n`。与 `read` 类似，`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：每次写入都从上一次写入结束的位置继续。

以下程序片段（构成了程序 cat 的核心）将其标准输入的数据复制到标准输出。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;
  如果(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` 并不知道它是在从文件、控制台还是管道中读取数据。同样，`cat` 也不知道它是在向控制台、文件还是其他任何地方输出数据。使用文件描述符以及文件描述符 0 表示输入、文件描述符 1 表示输出的约定，使得 `cat` 的实现变得简单。

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中编号最小的未使用描述符。

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，因此子进程开始时拥有与父进程完全相同的打开文件。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 fork、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。以下是 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); }
```

在子进程关闭文件描述符 0 后，`open`保证会为新打开的`input.txt`使用该文件描述符：0 将是可用的最小文件描述符。然后，`cat`执行时，文件描述符 0（标准输入）指向`input.txt`。父进程的文件描述符不会因这一系列操作而改变，因为它只修改了子进程的描述符。

xv6 shell 中的 I/O 重定向代码正是以这种方式工作的（user/sh.c:82）。回想一下，在代码的这一点上，shell 已经 fork 了子 shell，而 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，这些标志以位的形式表示，用于控制 `open` 的行为。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，这些指令用于指示 open 函数如何打开文件用于读取、写入或同时进行读取和写入，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将 fork 和 exec 分开调用是有帮助的：在这两者之间，shell 有机会在不干扰主 shell 的 I/O 设置的情况下重定向子进程的 I/O。人们可能会想象一个假设的 forkexec 组合系统调用，但选项

可以想象一个假设的`forkexec`组合系统调用来进行 I/O 重定向，但这样的调用似乎有些笨拙。shell 可以在调用`forkexec`之前修改自己的 I/O 设置（然后再撤销这些修改）；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会自己进行 I/O 重定向。

尽管 fork 复制了文件描述符表，但每个底层文件的偏移量在父进程和子进程之间是共享的。考虑以下示例：

```
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。父进程中的写操作（由于 wait 的作用，只有在子进程完成后才会运行）从子进程写操作停止的地方继续。这种行为有助于从 shell 序列中产生顺序输出。

命令，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，该描述符引用相同的底层 I/O 对象。两个文件描述符共享一个偏移量，就像由 fork 复制的文件描述符一样。这是将 hello world 写入文件的另一种方式：

```
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，即使它们是通过对同一文件的 open 调用产生的。Dup 允许 shell 实现如下命令：

ls existing-file non-existing-file > tmp1 2>&1。2>&1 告诉 shell 将

命令文件描述符 2 是描述符 1 的副本。现有文件的名称和不存在的文件的错误信息都将显示在文件 tmp1 中。xv6 shell 不支持错误文件描述符的 I/O 重定向，但现在你知道如何实现它了。

文件描述符是一种强大的抽象，因为它们隐藏了它们所连接对象的细节：一个进程向文件描述符 1 写入时，可能是在向文件、控制台等设备或管道写入。

1.3 Pipes

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。将数据写入管道的一端使得该数据可以从管道的另一端读取。管道为进程提供了一种通信方式。

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]);
   关闭(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`，它创建了一个新的管道，并在数组`p`中记录读和写的文件描述符。在`fork`之后，父进程和子进程都有指向管道的文件描述符。子进程调用`close`和`dup`，使文件描述符 0 指向管道的读端，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端，向管道写入数据，然后关闭写端。

如果没有数据可用，对管道的读取操作会等待数据被写入，或者所有引用写端的文件描述符被关闭；在后一种情况下，读取操作将返回 0，就像到达数据文件末尾一样。读取操作会阻塞直到不可能有新数据到达，这是为什么子进程在执行前关闭管道的写端如此重要的一个原因。

wc 上面：如果 wc 的文件描述符之一指向管道的写入端，wc 将永远不会看到文件结束符。

xv6 shell 以类似于上述代码的方式实现了管道，例如 `grep fork sh.c | wc -l`（user/sh.c:100）。子进程创建一个管道，将管道的左端与右端连接起来。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`），它本身会分叉出两个新的子进程（一个用于 `b`，一个用于 `c`）。因此，shell 可能会创建一个进程树。这棵树的叶子是命令，内部节点是等待左右子进程完成的进程。

原则上，可以让内部节点运行管道的左端，但这样做会正确实现会变得复杂。考虑仅进行以下修改：更改 sh.c，使其不为 p->left 进行 fork，并在内部进程中运行 runcmd(p->left)。然后，例如，`echo hi | wc`将不会产生输出，因为当`echo hi`在 runcmd 中退出时，内部进程退出并且永远不会调用 fork 来运行管道的右端。

内部进程退出并且从不调用 fork 来运行管道的右端，不正确的行为可以通过不在内部进程的 runcmd 中调用 exit 来修复，但这种修复会使代码复杂化：现在 runcmd 需要知道它是否是内部进程。当不为 runcmd(p->right)进行 fork 时，也会出现复杂情况。例如，仅进行该修改，`sleep 10 | echo hi` 会立即打印“hi”而不是在 10 秒后，因为 echo 会立即运行并退出，不会等待 sleep 完成。由于 sh.c 的目标是尽可能简单，它不会尝试避免创建内部进程。

管道可能看起来并不比临时文件更强大：管道

```
echo hello world | wc
```

可以在没有管道的情况下实现

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

在这种情况下，管道相比临时文件至少具有四个优势。首先，管道会自动清理自己；而使用文件重定向时，shell 必须小心地在完成后删除 `/tmp/xyz`。其次，管道可以传递任意长度的数据流，而文件重定向需要磁盘上有足够的空闲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完成后第二个程序才能开始。第四，如果你正在实现进程间通信，管道的阻塞读写比文件的非阻塞语义更高效。

1.4 文件系统

xv6 文件系统提供了数据文件，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。目录形成一棵树，从称为根的特殊目录开始。像 `/a/b/c` 这样的路径指的是根目录 `/` 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 `/` 开头的路径相对于调用进程的当前目录进行评估，可以使用 `chdir` 系统调用来更改当前目录。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。

有系统调用来创建新文件和目录：mkdir 创建新目录，带有 O_CREATE 标志的 open 创建新数据文件，mknod 创建新设备文件。这个例子展示了所有三种情况：

```
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。与设备文件相关联的是主设备号和次设备号（mknod 的两个参数），它们唯一标识一个内核设备。当进程稍后打开设备文件时，内核会将读写系统调用重定向到内核设备实现，而不是传递给文件系统。

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。每个链接由目录中的一个条目组成；该条目包含一个文件名和对 inode 的引用。inode 保存有关文件的元数据，包括其类型（文件、目录或设备）、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所引用的 inode 中检索信息。
填充一个结构体 stat，定义在 stat.h（kernel/stat.h）中为：

```
#define T_DIR 1 // 目录
#define T_FILE 2 // 文件
#define T_DEVICE 3 // 设备

struct stat { int dev; // 文件系统的磁盘设备 uint ino; //
Inode 编号 short type; // 文件类型 short nlink; // 文件的
链接数 uint64 size; // 文件大小（字节） };
```

link 系统调用创建另一个文件系统名称，指向与现有文件相同的 inode。这段代码创建了一个名为 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 读取或写入是相同的。每个 inode 都由一个唯一的 inode 编号标识。在上述代码序列之后，通过检查 fstat 的结果可以确定 a 和 b 引用相同的基础内容：两者都将返回相同的 inode 编号（ino），并且 nlink 计数将设置为 2。

unlink 系统调用从文件系统中移除一个名称。文件的 inode 和存储其内容的磁盘空间只有在文件的链接计数为零且没有文件描述符引用它时才会被释放。因此，添加

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

到最后一个代码序列时，inode 和文件内容仍然可以作为 b 访问。此外，

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

是一种惯用的方式，用于创建一个没有名称的临时 inode，当进程关闭文件描述符（fd）或退出时，该 inode 将被清理。

Unix 提供了可从 shell 调用的文件实用程序作为用户级程序，例如 mkdir、ln、rm 等命令。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，这个计划似乎显而易见，但在 Unix 设计时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，它内置于 shell 中（user/sh.c:160）。cd 必须改变 shell 本身的当前工作目录。如果 cd 作为常规命令运行，那么 shell 会 fork 一个子进程，子进程会运行 cd，而 cd 会改变子进程的工作目录。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、管道以及方便的 shell 语法组合在一起，是编写通用可重用程序的一大进步。这一理念催生了“软件工具”文化，这种文化在很大程度上推动了 Unix 的强大和流行，而 shell 则是第一种所谓的“脚本语言”。Unix 的系统调用接口至今仍存在于 BSD、Linux 和 Mac OS X 等系统中。

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 并不符合 POSIX 标准：它缺少许多系统调用（包括像 lseek 这样的基本调用），而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。一些人已经扩展了 xv6，增加了更多的系统调用和一个简单的 C 库，以便运行基本的 Unix 程序。然而，现代内核比 xv6 提供了更多的系统调用和更多种类的内核服务。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。现代内核持续且快速地发展，并提供了许多超越 POSIX 的功能。

Unix 通过一组文件名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。这一思想可以扩展到更多种类的资源；一个很好的例子是 Plan 9 [13]，它将“资源即文件”的概念应用于网络、图形等领域。然而，大多数源自 Unix 的操作系统并未遵循这一路线。

文件系统和文件描述符一直是强大的抽象概念。即便如此，操作系统接口还有其他模型。Multics，Unix 的前身，以一种使文件存储看起来像内存的方式进行抽象，产生了非常不同的接口风格。Multics 设计的复杂性直接影响了 Unix 的设计者，他们试图构建更简单的东西。

Xv6 不提供用户概念或保护一个用户免受另一个用户的影响；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多路复用到底层硬件上，隔离进程，并提供受控的进程间通信机制。通过学习 xv6，你应该能够查看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 系统调用在两个进程之间通过一对管道“乒乓”传递一个字节，每个方向一个管道。测量程序的性能，以每秒交换次数为单位。

第 2 章

操作系统组织

操作系统的一个关键要求是同时支持多个活动。例如，使用第 1 章中描述的系统调用接口，一个进程可以通过 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，即使进程数量超过硬件 CPU 的数量，操作系统也必须确保所有进程都有机会执行。操作系统还必须安排进程之间的隔离。也就是说，如果一个进程有错误并发生故障，它不应该影响不依赖于该错误进程的进程。然而，完全隔离过于严格，因为进程之间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，有许多方法可以做到这一点，但本文重点介绍以单一内核为中心的主流设计，许多 Unix 操作系统都采用这种设计。本章还概述了 xv6 中的进程，这是 xv6 中的隔离单元，以及 xv6 启动时创建的第一个进程。

Xv6 运行在多核 RISC-V 微处理器上，其许多低级功能（例如进程实现）是特定于 RISC-V 的。RISC-V 是一种 64 位 CPU，而 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 int 是 32 位的。本书假设读者已经在某种架构上进行过一些机器级编程，并将在需要时介绍 RISC-V 特定的概念。RISC-V 的有用参考是《The RISC-V Reader: An Open Architecture Atlas》[12]。用户级 ISA [2] 和特权架构 [1] 是官方规范。

完整的计算机中的 CPU 被支持硬件包围，其中大部分以 I/O 接口的形式存在。Xv6 是为 qemu 的“-machine virt”选项模拟的支持硬件编写的。这包括 RAM、包含引导代码的 ROM、与用户键盘/屏幕的串行连接以及用于存储的磁盘。

---

[1] 通过“多核”，本文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”也可以更具体地指代具有多个独立处理器芯片的计算机。

2.1 抽象物理资源

当遇到一个操作系统时，人们可能会问的第一个问题是为什么要有它？也就是说，可以将图 1.2 中的系统调用实现为一个库，应用程序与之链接。在这种方案中，每个应用程序甚至可以拥有一个根据其需求定制的库。应用程序可以直接与硬件资源交互，并以最适合应用程序的方式使用这些资源（例如，以实现高或可预测的性能）。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，这些应用程序必须表现良好。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且没有错误，这种协作式时间共享方案可能是可行的。但更常见的情况是应用程序之间不相互信任，并且存在错误，因此通常需要比协作式方案提供的更强的隔离性。

为了实现强隔离，禁止应用程序直接访问敏感的硬件资源，并将这些资源抽象为服务是有帮助的。例如，Unix 应用程序仅通过文件系统的 open、read、write 和 close 系统调用来与存储交互，而不是直接读写磁盘。这为应用程序提供了路径名的便利，并允许操作系统（作为接口的实现者）管理磁盘。即使不考虑隔离，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，因此应用程序无需意识到时间共享。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时也能共享 CPU。

另一个例子是，Unix 进程使用 exec 来构建其内存映像，而不是直接与物理内存交互。这使得操作系统能够决定将进程放置在内存中的哪个位置；如果内存紧张，操作系统甚至可能将进程的部分数据存储在磁盘上。Exec 还为用户提供了文件系统的便利，用于存储可执行程序映像。

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了许多细节（例如，管道或文件中的数据存储位置），而且它们的定义方式也简化了交互。例如，如果管道中的一个应用程序失败，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，又实现了强隔离的可能性。Unix 接口并非抽象资源的唯一方式，但它已被证明是一种非常好的方式。

2.2　用户模式、监督模式和系统调用

强隔离要求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受到影响。

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无法修改（甚至读取）操作系统的数据结构和指令，并且应用程序无法访问其他进程的内存。

CPU 提供了硬件支持以实现强隔离。例如，RISC-V 有三种模式可以执行指令：机器模式（machine mode）、监管模式（supervisor mode）和用户模式（user mode）。在机器模式下执行的指令具有完全权限；CPU 启动时处于机器模式。机器模式主要用于配置计算机。Xv6 在机器模式下执行几行代码，然后切换到监管模式。

在监督模式下，CPU 可以执行特权指令：例如，启用和禁用中断、读取和写入保存页表地址的寄存器等。如果用户模式下的应用程序尝试执行特权指令，CPU 不会执行该指令，而是切换到监督模式，以便监督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 展示了这种组织结构。应用程序只能执行用户模式指令（例如，加法运算等），并被称为在用户空间中运行，而监督模式下的软件还可以执行特权指令，并被称为在内核空间中运行。在内核空间（或监督模式）中运行的软件称为内核。

一个想要调用内核函数的应用程序（例如，xv6 中的 read 系统调用）必须切换到内核模式。CPU 提供了一条特殊指令，用于将 CPU 从用户模式切换到监督模式，并进入由内核指定的入口点。（RISC-V 为此提供了 ecall 指令。）一旦 CPU 切换到监督模式，内核就可以验证系统调用的参数，决定是否允许应用程序执行请求的操作，然后拒绝或执行它。重要的是，内核控制着切换到监督模式的入口点；如果应用程序可以决定内核入口点，恶意应用程序可能会在跳过参数验证的点进入内核。

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模式下运行。一种可能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模式下运行。这种组织方式被称为单片内核。

在这种组织方式中，整个操作系统以完全的硬件权限运行。这种组织方式很方便，因为操作系统设计者不需要决定操作系统的哪一部分不需要完全的硬件权限。此外，操作系统的不同部分更容易协作。例如，操作系统可能有一个缓冲区缓存，可以同时被文件系统和虚拟内存系统共享。

单片式组织的一个缺点是操作系统不同部分之间的接口通常很复杂（正如我们将在本文的其余部分看到的那样），因此操作系统开发人员很容易犯错。在单片式内核中，错误是致命的，因为超级用户模式下的错误通常会导致内核崩溃。如果内核崩溃，

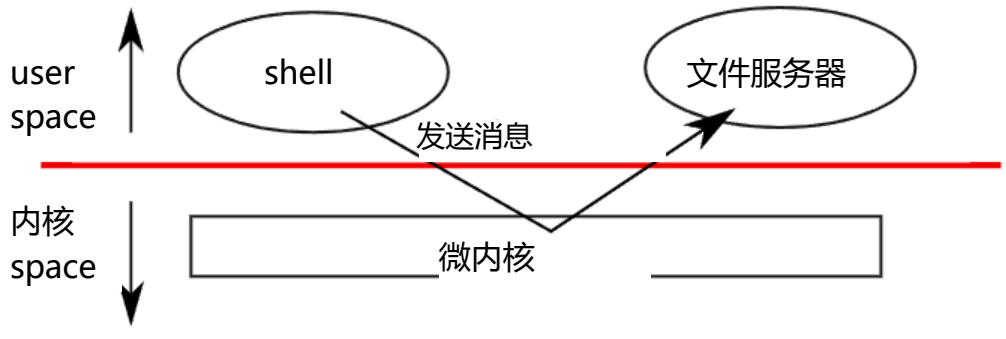

图 2.1：带有文件系统服务器的微内核

电脑停止工作，因此所有应用程序也都会失败。电脑必须重新启动才能再次运行。

为了减少内核中错误的风险，操作系统设计者可以尽量减少在监督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代码。这种内核组织被称为微内核。

图 2.1 展示了这种微内核设计。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 OS 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，用于从一个用户模式进程向另一个用户模式进程发送消息。例如，如果像 shell 这样的应用程序想要读取或写入文件，它会向文件服务器发送一条消息并等待响应。

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。这种组织方式使得内核相对简单，因为操作系统的大部分功能驻留在用户级服务器中。

Xv6 实现为一个单体内核，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，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更小，但从概念上讲，xv6 是单体内核。

2.4 代码：xv6 组织

xv6 内核源代码位于 kernel/子目录中。源代码按照模块化的粗略概念被划分为多个文件；图 2.2 列出了这些文件。模块间的接口定义在
defs.h (kernel/defs.h)

2.5 进程概述

xv6（以及其他类 Unix 操作系统）中的隔离单位是进程。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，因此进程无法颠覆内核的隔离机制。内核必须谨慎地实现进程抽象，因为有缺陷或恶意的应用程序可能会诱使内核或硬件执行不良操作（例如，绕过隔离）。内核用于实现这些机制的

File	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	最初的启动指令。
exec.c	exec() 系统调用。file.c 文件描述符支持。fs.c 文件系统。
kalloc.c	物理页面分配器。
kernelvec.S	处理来自内核的陷阱和定时器中断。log.c 文件系统日志和崩溃恢复。
main.c	控制启动期间其他模块的初始化。
pipe.c	管道。
plic.c	RISC-V 中断控制器。
printf.c	格式化输出到控制台。
proc.c	进程和调度。sleeplock.c 让出 CPU 的锁。
spinlock.c	不释放 CPU 的锁。start.c 早期的机器模式启动代码。
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	用于处理和从陷阱及中断返回的 C 代码。
uart.c	串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。

图 2.2：Xv6 内核源文件。

内核用于实现进程的机制包括用户/超级用户模式标志、地址空间和线程的时间切片。

为了帮助加强隔离，进程抽象为程序提供了它拥有自己私有机器的假象。进程为程序提供了一个看似私有的内存系统或地址空间，其他进程无法读取或写入。进程还为程序提供了一个看似属于自己的 CPU 来执行程序的指令。

Xv6 使用页表（由硬件实现）为每个进程提供自己的地址空间。RISC-V 页表将虚拟地址（RISC-V 指令操作的地址）转换（或“映射”）为物理地址（CPU 芯片发送到主存储器的地址）。

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，地址空间包括从虚拟地址开始的进程用户内存

图 2.3：进程虚拟地址空间的布局

一个地址空间包括从虚拟地址零开始的进程用户内存。首先是指令，接着是全局变量，然后是栈，最后是一个“堆”区域（用于 malloc），进程可以根据需要扩展这个区域。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；而 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:348）。在地址空间的顶部，xv6 保留了一个页面用于 trampoline 和一个页面用于映射进程的 trapframe 以切换到内核，我们将在第 4 章中解释这一点。

xv6 内核为每个进程维护了许多状态信息，这些信息被收集到一个结构体 `struct proc` 中（kernel/proc.h:86）。一个进程最重要的内核状态包括其页表、内核栈和运行状态。我们将使用符号 `p->xxx` 来引用 `proc` 结构体中的元素；
例如，p->pagetable 是指向进程页表的指针。

每个进程都有一个执行线程（简称线程），用于执行进程的指令。线程可以被挂起，稍后再恢复。为了在进程之间透明地切换，内核会挂起当前运行的线程并恢复另一个进程的线程。线程的大部分状态（局部变量、函数调用返回地址）存储在线程的栈上。每个进程有两个栈：用户栈和内核栈（p->kstack）。当进程执行用户指令时，只有其用户栈在使用，而内核栈为空。当进程进入内核（由于系统调用或中断）时，内核代码在进程的内核栈上执行；当进程在内核中时，其用户栈仍然包含保存的数据，但不会被主动使用。进程的线程在主动使用其用户栈和内核栈之间交替切换。内核栈是独立的（并且受到用户代码的保护），因此即使进程破坏了其用户栈，内核仍然可以执行。

进程可以通过执行 RISC-V 的 ecall 指令来进行系统调用。该指令会提升硬件特权级别，并将程序计数器更改为内核定义的入口点。入口点处的代码会切换到内核栈，并执行实现系统调用的内核指令。当系统调用完成时，内核会切换回用户模式。

内核通过调用`sret`指令切换回用户栈并返回到用户空间，该指令降低硬件特权级别，并在系统调用指令之后恢复执行用户指令。进程的线程可以在内核中“阻塞”以等待 I/O，并在 I/O 完成后从离开的地方恢复执行。

p->state 表示进程是否已分配、准备运行、正在运行、等待 I/O 或正在退出。

p->pagetable 保存了进程的页表，格式符合 RISC-V 硬件的预期。xv6 在执行用户空间的进程时，使分页硬件使用该进程的 p->pagetable。进程的页表还充当了分配给存储进程内存的物理页地址的记录。

2.6 代码：启动 xv6 和第一个进程

为了使 xv6 更加具体，我们将概述内核如何启动并运行第一个进程。随后的章节将更详细地描述此概述中出现的机制。

当 RISC-V 计算机启动时，它会初始化自身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:6）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。

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

_entry 处的指令设置了一个栈，以便 xv6 可以运行 C 代码。Xv6 在文件 start.c (kernel/start.c:11) 中声明了一个初始栈的空间，即 stack0。_entry 处的代码将栈指针寄存器 sp 加载为 stack0+4096 的地址，即栈的顶部，因为 RISC-V 上的栈是向下增长的。现在内核有了一个栈，_entry 调用 start 处的 C 代码。
(kernel/start.c:21).

函数 `start` 执行一些仅在机器模式下允许的配置，然后切换到监督模式。为了进入监督模式，RISC-V 提供了指令 `mret`。该指令通常用于从监督模式到机器模式的先前调用中返回。`start` 并不是从这样的调用中返回，而是设置了一些东西，就好像有一个这样的调用一样：它将寄存器 `mstatus` 中的先前特权模式设置为监督模式，通过将 `main` 的地址写入寄存器 `mepc` 来设置返回地址为 `main`，通过将页表寄存器 `satp` 写入 0 来禁用监督模式下的虚拟地址转换，并将所有中断和异常委托给监督模式。

在跳转到监管模式之前，`start` 执行了最后一个任务：它编程时钟芯片以生成定时器中断。完成这些准备工作后，`start` 通过调用 `mret` “返回”到监管模式。这导致程序计数器更改为 `main`（kernel/main.c:11）。

在 main (kernel/main.c:11) 初始化了几个设备和子系统之后，它通过调用 userinit (kernel/proc.c:212) 创建了第一个进程。第一个进程执行一个用 RISC-V 汇编编写的小程序 initcode.S (user/initcode.S:1)，该程序通过调用 exec 系统调用重新进入内核。正如我们在第 1 章中看到的，exec 替换了当前进程的内存和寄存器。

exec 用新程序（在本例中为 /init）替换当前进程的内存和寄存器。一旦内核完成 exec，它就会返回到 /init 进程的用户空间。Init（user/init.c:15）在需要时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。接着，它在控制台上启动一个 shell。系统启动完成。

2.7 现实世界

在现实世界中，人们可以发现既有宏内核也有微内核。许多 Unix 内核是宏内核。例如，Linux 拥有一个宏内核，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）。像 L4、Minix 和 QNX 这样的内核被组织为带有服务器的微内核，并且在嵌入式环境中得到了广泛应用。

大多数操作系统都采用了进程的概念，而且大多数进程看起来与 xv6 的进程相似。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。在进程中支持多个线程涉及相当多的机制，这些机制 xv6 并不具备，包括潜在的接口更改（例如，Linux 的 clone，fork 的一个变体），以控制进程线程共享哪些方面。

2.8 练习

1. 你可以使用 gdb 来观察第一次内核到用户的转换。运行 make qemu-gdb。在另一个窗口中，在同一目录下，运行 gdb。输入 gdb 命令 break *0x3ffffff10e，这会在内核中跳转到用户空间的 sret 指令处设置一个断点。输入 continue gdb 命令。gdb 应该在断点处停止，即将执行 sret。

   输入 `stepi`。gdb 现在应该显示它正在地址 0x0 处执行，这是在用户空间中的 initcode.S 的起始位置。

第 3 章

页表

页表是操作系统为每个进程提供其私有地址空间和内存的机制。页表决定了内存地址的含义以及可以访问的物理内存部分。它们允许 xv6 隔离不同进程的地址空间，并将它们多路复用到单个物理内存上。页表还提供了一层间接性，使 xv6 能够执行一些技巧：在多个地址空间中映射相同的内存（一个跳板页），并使用未映射的页来保护内核和用户栈。本章的其余部分将解释 RISC-V 硬件提供的页表以及 xv6 如何使用它们。

3.1　分页硬件

作为提醒，RISC-V 指令（包括用户和内核）操作的是虚拟地址。机器的 RAM 或物理内存则通过物理地址进行索引。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 运行在 Sv39 RISC-V 上，这意味着只使用 64 位虚拟地址的底部 39 位；顶部 25 位未被使用。在这种 Sv39 配置中，RISC-V 页表逻辑上是一个包含 2^27（134,217,728）个页表项（PTEs）的数组。每个 PTE 包含一个 44 位的物理页号（PPN）和一些标志位。分页硬件通过使用 39 位中的顶部 27 位来索引页表以找到一个 PTE，并生成一个 56 位的物理地址，其顶部 44 位来自 PTE 中的 PPN，底部 12 位则从原始虚拟地址中复制而来。图 3.1 展示了这一过程，页表被逻辑地视为一个简单的 PTE 数组（更详细的描述请参见图 3.2）。页表使操作系统能够以 4096（2^12）字节对齐的块为单位控制虚拟到物理地址的转换。这样的块称为页。

在 Sv39 RISC-V 中，虚拟地址的高 25 位不用于转换；未来，RISC-V 可能会使用这些位来定义更多的转换级别。物理地址也有增长的空间：PTE 格式中有空间可以让物理页号再增长 10 位。

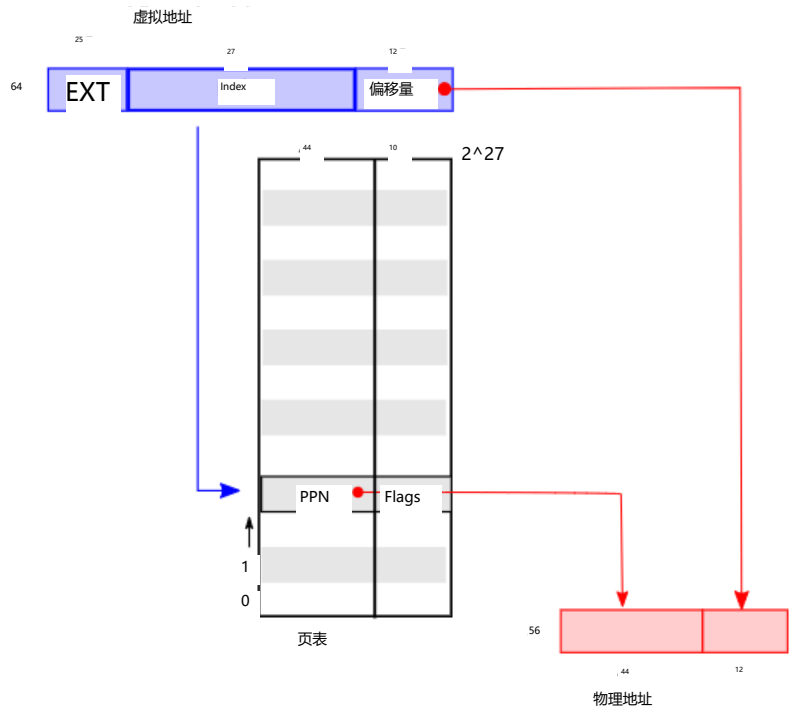

图 3.1：RISC-V 虚拟和物理地址，带有简化的逻辑页表。

如图 3.2 所示，实际的转换过程分为三个步骤。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。每个页表页包含 512 个 PTE，用于树的最后一级。分页硬件使用 27 位中的高 9 位选择根页表页中的 PTE，中间 9 位选择树中下一级页表页中的 PTE，低 9 位选择最终的 PTE。

如果翻译地址所需的三个 PTE 中的任何一个不存在，分页硬件会引发一个页面错误异常，由内核来处理该异常（见第 4 章）。这种三级结构允许页表在常见情况下省略整个页表页，即当大范围的虚拟地址没有映射时。

每个 PTE 包含标志位，这些标志位告诉分页硬件如何使用相关的虚拟地址。PTE_V 指示 PTE 是否存在：如果未设置，引用该页面会导致异常（即不允许）。PTE_R 控制是否允许指令读取该页面。PTE_W 控制是否允许指令写入该页面。PTE_X 控制 CPU 是否可以将页面内容解释为指令并执行它们。PTE_U 控制是否允许用户模式下的指令访问该页面；如果未设置 PTE_U，则 PTE 只能在监督模式下使用。图 3.2 展示了这一切是如何工作的。这些标志位以及所有其他与页面硬件相关的结构都在(kernel/riscv.h)中定义。

要告诉硬件使用页表，内核必须将根页表页的物理地址写入 satp 寄存器。每个 CPU 都有自己的 satp。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，以便不同的 CPU 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。

关于术语的一些说明。物理内存指的是 DRAM 中的存储单元。一个字节的物理内存有一个地址，称为物理地址。指令只使用虚拟地址，分页硬件将虚拟地址转换为物理地址，然后发送到 DRAM 硬件进行读取。

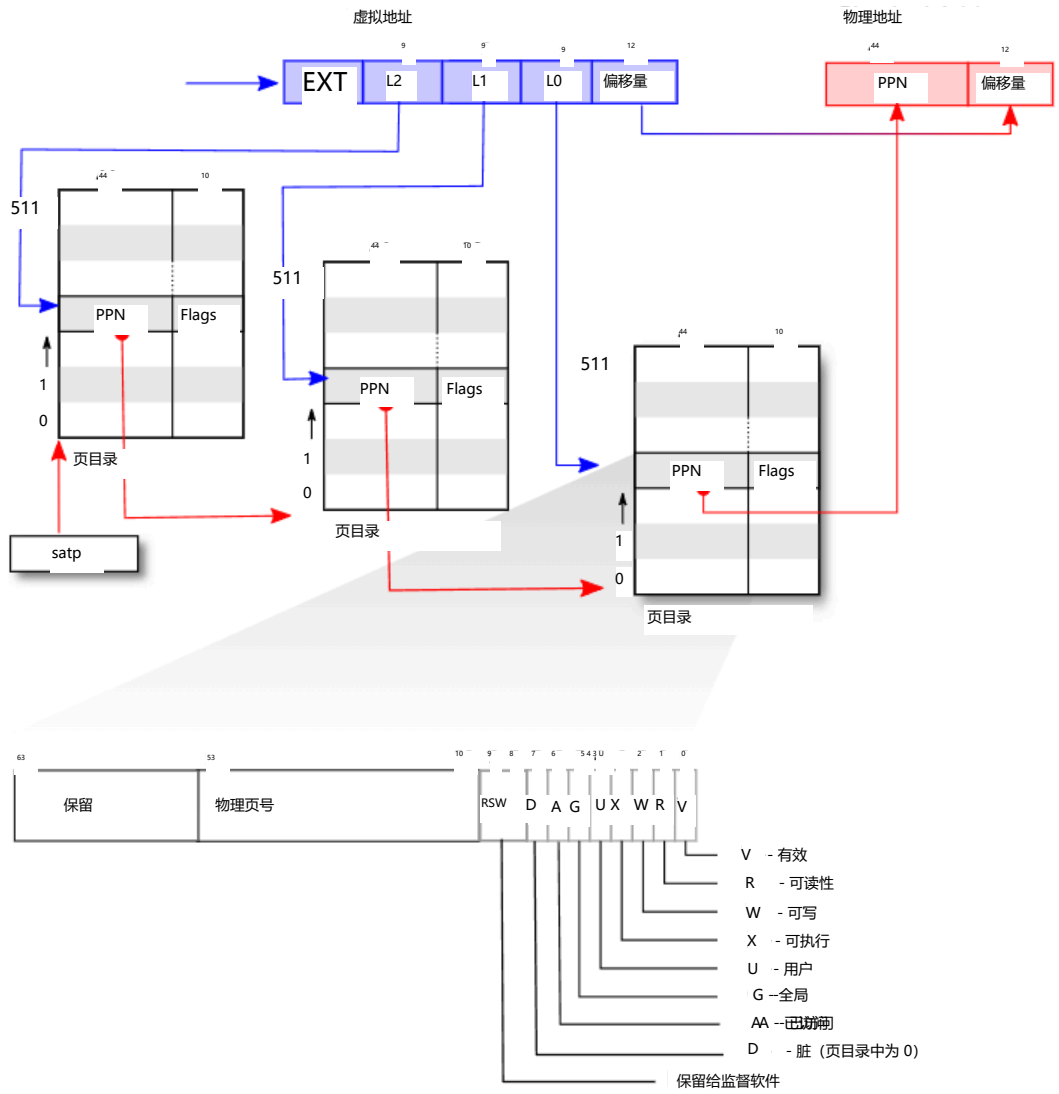

图 3.2：RISC-V 地址转换细节。

然后发送到 DRAM 硬件以读取或写入存储。与物理内存和虚拟地址不同，虚拟内存不是物理对象，而是指内核提供的用于管理物理内存和虚拟地址的抽象和机制的集合。

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，再加上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 显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，其中包括从物理地址 0x80000000 开始并至少延续到 0x86400000 的 RAM（物理内存），xv6 称之为 PHYSTOP。QEMU 模拟还包括诸如磁盘接口等 I/O 设备。QEMU 将设备接口暴露给软件，作为位于物理地址空间 0x80000000 以下的内存映射控制寄存器。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。第 4 章解释了 xv6 如何与设备交互。

内核通过“直接映射”访问 RAM 和内存映射设备寄存器；也就是说，将资源映射到与物理地址相等的虚拟地址。例如，

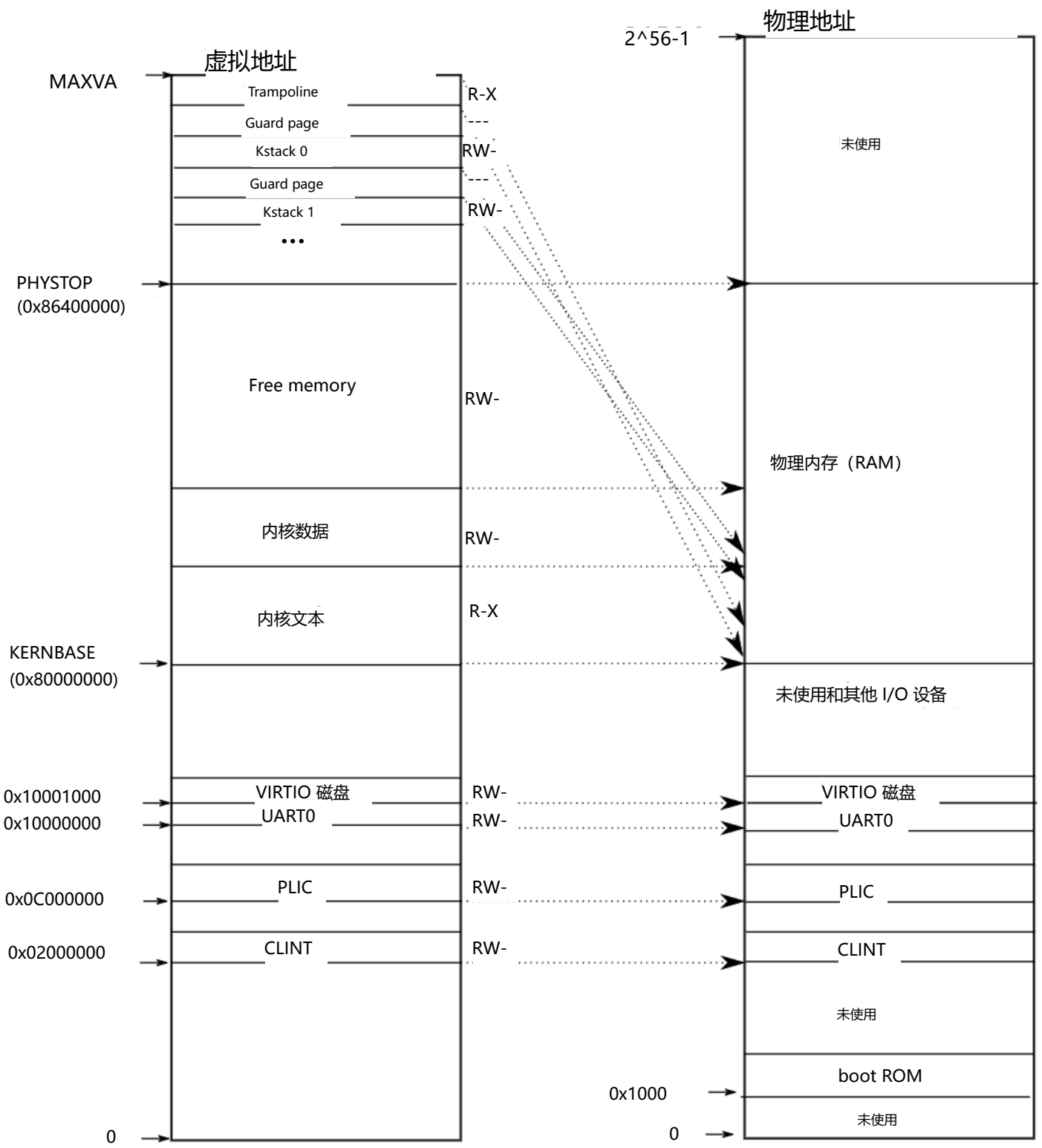

图 3.3：左侧是 xv6 的内核地址空间。RWX 表示 PTE 的读、写和执行权限。右侧是 xv6 期望看到的 RISC-V 物理地址空间。

内核本身位于虚拟地址空间和物理内存中的 KERNBASE=0x80000000。直接映射简化了读取或写入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到子进程时，直接将该地址用作虚拟地址。

有几个内核虚拟地址不是直接映射的：

• 蹦床页面。它被映射到虚拟地址空间的顶部；用户页表具有相同的映射。第 4 章讨论了蹦床页面的作用，但我们在这里看到了页表的一个有趣用例；一个物理页面（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，一次通过直接映射。

• 内核栈页面。每个进程都有自己的内核栈，它被映射到高位，以便在其下方 xv6 可以留下一个未映射的防护页。防护页的 PTE 是无效的（即 PTE_V 未设置），因此如果内核溢出内核栈，可能会导致异常并使内核崩溃。如果没有防护页，溢出的栈会覆盖其他内核内存，导致操作错误。崩溃比错误操作更可取。

虽然内核通过高内存映射使用其堆栈，但它们也可以通过直接映射地址访问内核。另一种设计可能只有直接映射，并在直接映射地址使用堆栈。然而，在这种安排中，提供保护页将涉及取消映射原本指向物理内存的虚拟地址，这将使其难以使用。

内核以权限 PTE_R 和 PTE_X 映射跳板页和内核文本的页面。内核从这些页面读取并执行指令。内核以权限 PTE_R 和 PTE_W 映射其他页面，以便能够读写这些页面中的内存。保护页的映射是无效的。

3.3 代码：创建一个地址空间

大多数用于操作地址空间和页表的 xv6 代码位于 vm.c 中（kernel/vm.c:1）。核心数据结构是 pagetable_t，它实际上是指向 RISC-V 根页表页的指针；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，以及 mappages，它为新的映射安装 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；其他函数两者都适用。copyout 和 copyin 将数据复制到或从作为系统调用参数提供的用户虚拟地址中；它们位于 vm.c 中，因为它们需要显式转换这些地址以找到相应的物理内存。

在启动序列的早期，`main` 调用 `kvminit`（kernel/vm.c:22）来创建内核的页表。这个调用发生在 xv6 在 RISC-V 上启用分页之前，因此地址直接引用物理内存。`Kvminit` 首先分配一页物理内存来保存根页表页。然后它调用 `kvmmap` 来安装内核所需的地址转换。这些转换包括内核的指令和数据、物理内存直到 `PHYSTOP`，以及实际上是设备的内存范围。

kvmmap (kernel/vm.c:118) 调用 mappages (kernel/vm.c:149)，它将映射安装到将一系列虚拟地址的页表映射到相应的物理地址范围。它对范围内的每个虚拟地址分别进行映射，以页为单位。对于每个要映射的虚拟地址，`mappages` 调用 `walk` 来找到该地址的 PTE 地址。然后，它初始化 PTE 以保存相关的物理页号、所需的权限（`PTE_W`、`PTE_X` 和/或 `PTE_R`），以及 `PTE_V` 以将 PTE 标记为有效（kernel/vm.c:161）。

walk (kernel/vm.c:72) 模仿 RISC-V 分页硬件查找虚拟地址的 PTE（参见图 3.2）。walk 逐级下降三级页表，每次使用 9 位虚拟地址。它使用每一级的 9 位虚拟地址来查找下一级页表或最终页面的 PTE。

它使用每一级的 9 位虚拟地址来查找下一级页表或最终页面的 PTE（kernel/vm.c:78）。如果 PTE 无效，则所需的页面尚未分配；如果

alloc 参数被设置时，walk 会分配一个新的页表页并将其物理地址放入 PTE 中。它返回树中最低层的 PTE 地址（kernel/vm.c:88）。

上述代码依赖于物理内存直接映射到内核虚拟地址空间。例如，当`walk`逐级下降页表时，它从 PTE 中提取下一级页表的（物理）地址（kernel/vm.c:80），然后使用该地址作为虚拟地址来获取下一级的 PTE（kernel/vm.c:78）。

main 调用 kvminithart（kernel/vm.c:53）来安装内核页表。它将根页表页的物理地址写入寄存器 satp。此后，CPU 将使用内核页表来转换地址。由于内核使用恒等映射，下一条指令的虚拟地址将映射到正确的物理内存地址。

procinit (kernel/proc.c:26)，由 main 调用，为每个进程分配一个内核栈。它将每个栈映射到由 KSTACK 生成的虚拟地址，这为无效的栈保护页留出了空间。kvmmap 将映射的 PTE 添加到内核页表中，调用 kvminithart 将内核页表重新加载到 satp 中，以便硬件知道新的 PTE。

每个 RISC-V CPU 在转换后备缓冲区（TLB）中缓存页表项，当 xv6 更改页表时，它必须告诉 CPU 使相应的缓存 TLB 项无效。如果不这样做，那么稍后 TLB 可能会使用旧的缓存映射，指向一个在此期间已被分配给另一个进程的物理页，结果可能导致一个进程能够篡改其他进程的内存。RISC-V 有一条指令

sfence.vma 刷新当前 CPU 的 TLB。xv6 在重新加载 satp 寄存器后在 kvminithart 中执行 sfence.vma，并在切换到用户页表后返回到用户空间之前在 trampoline 代码中执行（kernel/trampoline.S:79）。

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 使用内核结束到 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线一个链表来跟踪哪些页面是空闲的。分配包括从链表中移除一个页面；释放包括将释放的页面添加到链表中。

3.5 代码：物理内存分配器

分配器位于 kalloc.c（kernel/kalloc.c:1）。分配器的数据结构是一个可用于分配的物理内存页面的空闲列表。每个空闲页面的列表元素是一个

struct run (kernel/kalloc.c:17)。分配器从哪里获取内存来保存这个数据结构？它将每个空闲页面的 run 结构存储在空闲页面本身中，因为那里没有存储其他内容。空闲列表由一个自旋锁保护（kernel/kalloc.c:21-24）。列表和锁是

列表和锁被包装在一个结构体中，以明确锁保护结构体中的字段。现在，忽略锁以及调用 `acquire`和`release`；第 6 章将详细讨论锁。

函数 `main` 调用 `kinit` 来初始化分配器（kernel/kalloc.c:27）。`kinit` 将空闲列表初始化为包含内核结束到 `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。然而，xv6 假设机器有 128 兆字节的 RAM。`kinit` 调用 `freerange` 来通过每页调用 `kfree` 将内存添加到空闲列表中。一个 PTE 只能引用对齐在 4096 字节边界上的物理地址（即 4096 的倍数），因此 `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` 的调用为它提供了一些内存来管理。

分配器有时将地址视为整数以对它们执行算术运算（例如，遍历 freerange 中的所有页面），有时将地址用作指针来读写内存（例如，操作存储在每个页面中的 run 结构）；这种地址的双重使用是分配器代码中充满 C 类型转换的主要原因。另一个原因是释放和分配本质上会改变内存的类型。

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。这将导致在释放内存后使用内存的代码（使用“悬空引用”）读取垃圾数据而不是旧的合法内容；希望这会使此类代码更快地崩溃。然后 `kfree` 将页面添加到空闲列表的前面：它将 `pa` 强制转换为指向 `struct run` 的指针，将空闲列表的旧起始位置记录在 `r->next` 中，并将空闲列表设置为 `r`。`kalloc` 移除并返回空闲列表中的第一个元素。

3.6 进程地址空间

每个进程都有一个独立的页表，当 xv6 在进程之间切换时，它也会更改页表。如图 2.3 所示，进程的用户内存从虚拟地址零开始，可以增长到 MAXVA（kernel/riscv.h:348），允许进程原则上寻址 256 GB 的内存。

当一个进程向 xv6 请求更多的用户内存时，xv6 首先使用 kalloc 分配物理页。然后，它将指向新物理页的 PTE 添加到进程的页表中。Xv6 在这些 PTE 中设置 PTE_W、PTE_X、PTE_R、PTE_U 和 PTE_V 标志。大多数进程不会使用整个用户地址空间；xv6 在未使用的 PTE 中保持 PTE_V 为清除状态。

我们在这里看到了一些使用页表的很好的例子。首先，不同进程的页表将用户地址转换为物理内存的不同页面，因此每个进程都有私有的用户内存。其次，每个进程将其内存视为从零开始的连续虚拟地址，而进程的物理内存可以是非连续的。第三，内核在用户地址空间的顶部映射了一个包含蹦床代码的页面，因此一个物理内存页面出现在所有地址空间中。

图 3.4 更详细地展示了 xv6 中执行进程的用户内存布局。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最顶部。紧接着下面是一些值，这些值允许程序从 main 函数开始执行，就像函数

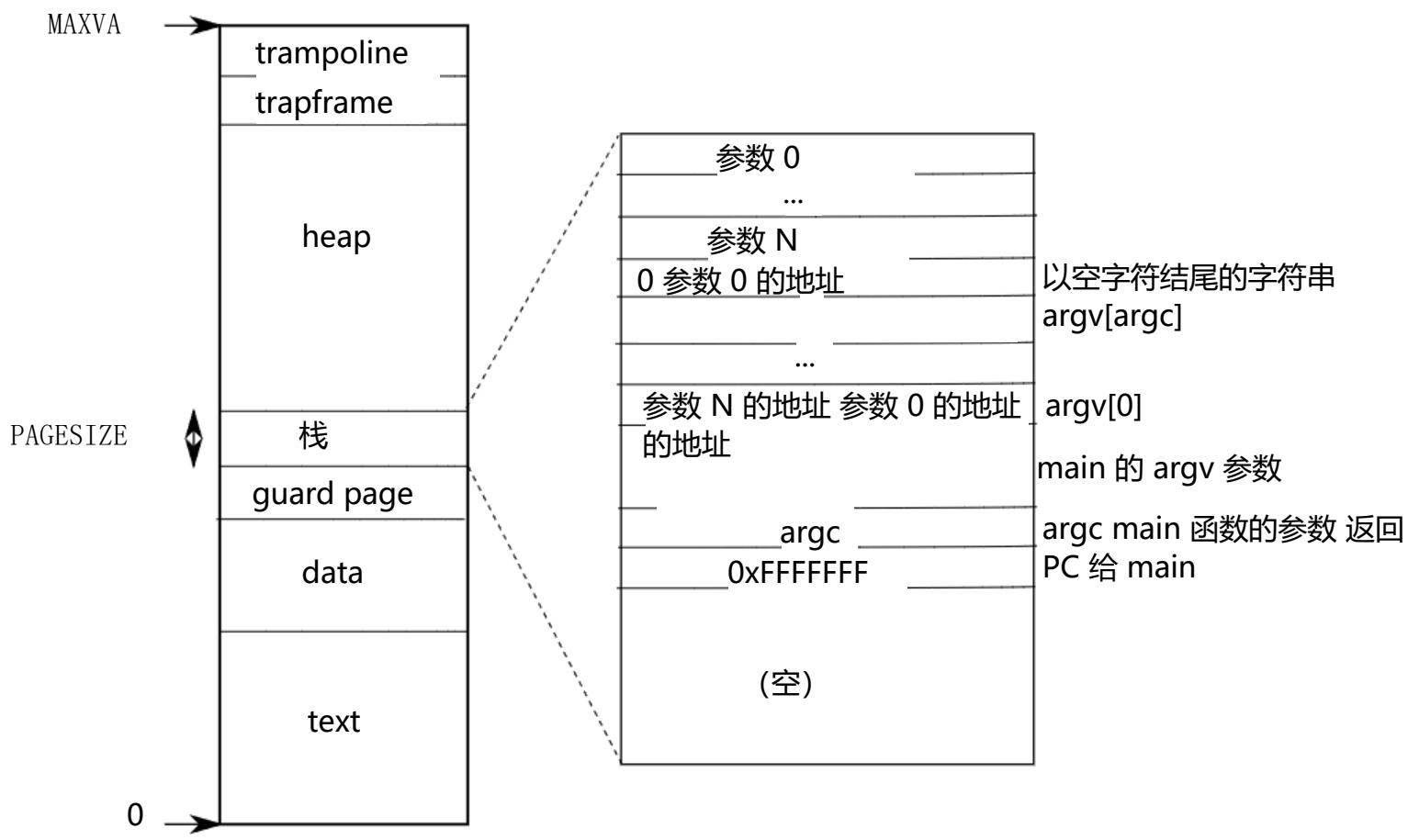

图 3.4：进程的用户地址空间及其初始栈。

紧接其下的是允许程序从 `main` 开始执行的值，就像刚刚调用了函数 `main(argc, argv)` 一样。

为了检测用户栈是否溢出分配的栈内存，xv6 在栈的正下方放置了一个无效的防护页。如果用户栈溢出并且进程尝试使用栈下方的地址，硬件将生成一个页错误异常，因为映射无效。现实世界中的操作系统可能会在栈溢出时自动为用户栈分配更多内存。

3.7 代码：sbrk

Sbrk 是进程用于缩小或增长其内存的系统调用。该系统调用的实现
由函数 `growproc`（`kernel/proc.c:239`）完成。`growproc` 调用 `uvmalloc` 或 `uvmdealloc`，
根据 n 是正数还是负数来决定。uvmalloc (kernel/vm.c:229) 使用 kalloc 分配物理内存，并使用 mappages 将 PTE 添加到用户页表中。uvmdealloc 调用 uvmunmap (kernel/vm.c:174)，后者使用 walk 查找 PTE 并使用 kfree 释放它们所引用的物理内存。
xv6 使用进程的页表不仅是为了告诉硬件如何映射用户虚拟地址，还作为分配给该进程的物理内存页的唯一记录。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

Exec 是创建地址空间用户部分的系统调用。它从文件系统中存储的文件初始化地址空间的用户部分。Exec (kernel/exec.c:13) 使用 namei (kernel/exec.c:26) 打开指定的二进制路径，这将在第 8 章中解释。然后，它读取 ELF 头。Xv6 应用程序以广泛使用的 ELF 格式描述，定义在 (kernel/elf.h) 中。一个 ELF 二进制文件由一个 ELF 头 struct elfhdr (kernel/elf.h:6) 和一系列程序段头 struct proghdr (kernel/elf.h:25) 组成。每个 proghdr 描述了必须加载到内存中的应用程序的一个部分；xv6 程序只有一个程序段头，但其他系统可能为指令和数据分别设置不同的段。

第一步是快速检查文件是否可能包含一个 ELF 二进制文件。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 头具有正确的魔数，exec 就假定该二进制文件格式良好。

Exec 使用 proc_pagetable (kernel/exec.c:38) 分配一个没有用户映射的新页表，使用 uvmalloc (kernel/exec.c:52) 为每个 ELF 段分配内存，并使用 loadseg (kernel/exec.c:10) 将每个段加载到内存中。loadseg 使用 walkaddr 找到分配的物理内存地址，以便写入 ELF 段的每一页，并使用 readi 从文件中读取数据。

/init 的程序段头，这是用 exec 创建的第一个用户程序，看起来像这样：

```
# objdump -p _init user/_init: 文件格式 elf64-littleriscv


程序头：
     LOAD off 0x00000000000000b0 vaddr 0x0000000000000000
    paddr 0x0000000000000000 align 2**3 filesz 0x0000000000000840 memsz 0x0000000000000858
    flags rwx STACK off 0x0000000000000000 vaddr 0x0000000000000000

           paddr 0x0000000000000000 align 2**4 filesz
           0x0000000000000000 memsz 0x0000000000000000
           flags rw-
```

程序段头的`filesz`可能小于`memsz`，表示它们之间的间隙应该用零填充（用于 C 全局变量），而不是从文件中读取。对于`/init`，`filesz`是 2112 字节，`memsz`是 2136 字节，因此`uvmalloc`分配了足够的物理内存来容纳 2136 字节，但只从文件`/init`中读取了 2112 字节。

现在，exec 分配并初始化用户栈。它只分配一个栈页。Exec 将参数字符串逐个复制到栈的顶部，并在 ustack 中记录它们的指针。它在将要传递给 main 的 argv 列表的末尾放置一个空指针。ustack 中的前三个条目是伪造的返回程序计数器、argc 和 argv 指针。

Exec 在栈页下方放置了一个不可访问的页面，这样尝试使用超过一个页面的程序将会出错。这个不可访问的页面还允许 exec 处理过大的参数；在这种情况下，exec 用于将参数复制到栈的 copyout（kernel/vm.c:355）函数会注意到目标页面不可访问，并返回 -1。

在准备新的内存映像期间，如果 exec 检测到错误（如无效的程序段），它会跳转到标签 bad，释放新的映像，并返回-1。Exec 必须等待

Exec 必须等待直到确定系统调用成功后才能释放旧映像：如果旧映像已经消失，系统调用无法返回 -1 给它。exec 中唯一的错误情况发生在创建映像期间。一旦映像完成，exec 就可以提交到新的页表（kernel/exec.c:113）并
释放旧的（kernel/exec.c:117）。
将 ELF 文件中的字节加载到 ELF 文件指定的内存地址中。用户或进程可以在 ELF 文件中放置他们想要的任何地址。因此，exec 是有风险的，因为 ELF 文件中的地址可能无意或有意地引用内核。对于一个不谨慎的内核来说，后果可能从崩溃到恶意破坏内核的隔离机制（即安全漏洞）。xv6 执行了一系列检查来避免这些风险。例如，`if(ph.vaddr + ph.memsz < ph.vaddr)`检查总和是否溢出 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`指向用户选择的地址，而`ph.memsz`足够大，使得总和溢出到 0x1000，这看起来像是一个有效的值。在用户地址空间也包含内核（但在用户模式下不可读/写）的旧版本 xv6 中，用户可以选择一个对应于内核内存的地址，从而将 ELF 二进制文件中的数据复制到内核中。 在 RISC-V 版本的 xv6 中，这种情况不会发生，因为内核有自己的独立页表；loadseg 加载到进程的页表中，而不是内核的页表中。
内核开发者很容易忽略一个关键的检查，而现实世界中的内核长期以来一直存在缺少检查的问题，这些缺失的检查可能被用户程序利用来获取内核权限。xv6 可能没有完全验证提供给内核的用户级数据，恶意用户程序可能利用这一点来绕过 xv6 的隔离机制。

3.9 现实世界

与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页错误异常，比 xv6 更复杂地使用分页，我们将在第 4 章讨论这一点。
Xv6 通过内核使用虚拟地址和物理地址之间的直接映射以及其假设物理 RAM 位于地址 0x8000000（内核期望加载的位置）来简化。这在 QEMU 中有效，但在实际硬件上却是一个糟糕的想法；实际硬件将 RAM 和设备放置在不可预测的物理地址上，因此（例如）在 0x8000000 可能没有 RAM，而 xv6 期望能够在此处存储内核。更严肃的内核设计利用页表将任意的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 支持在物理地址级别进行保护，但 xv6 没有使用该功能。在具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 KB 的内存，给它一个完整的 4 MB 超级页的物理内存是浪费的。在具有大量 RAM 的机器上，较大的页是有意义的，并且可能减少页表操作的开销。
xv6 内核缺乏类似 malloc 的分配器，无法为小对象提供内存，这阻碍了内核使用需要动态分配的复杂数据结构。

内存分配是一个长期的热门话题，基本问题是如何高效利用有限的内存并为未知的未来请求做好准备[7]。如今，人们更关心速度而非空间效率。此外，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 xv6 那样只分配 4096 字节的块；一个真正的内核分配器需要同时处理小分配和大分配。

3.10 练习

1. 解析 RISC-V 的设备树以找到计算机的物理内存量。

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 包含什么内容？

3. 修改 xv6 以使用超级页（super pages）作为内核。

4. 修改 xv6，使得当用户程序解引用空指针时，它会收到一个异常。也就是说，修改 xv6，使得虚拟地址 0 不为用户程序映射。

5. Unix 实现中的 exec 传统上包含对 shell 脚本的特殊处理。如果要执行的文件以文本 #! 开头，则第一行被视为用于解释文件的程序。例如，如果调用 exec 来运行 myprog arg1 并且 myprog 的第一

   如果行的内容是 #!/interp，那么 exec 会运行 /interp，并将命令行参数设置为 /interp myprog arg1。在 xv6 中实现对这一约定的支持。

6. 为内核实现地址空间随机化。

第 4 章

陷阱和系统调用

有三种事件会导致 CPU 暂停普通指令的执行，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 指令以请求内核为其执行某些操作时。另一种情况是异常：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，当设备发出信号表示需要关注时，例如磁盘硬件完成读取或写入请求时。

本书使用“trap”作为这些情况的通用术语。通常，在 trap 发生时正在执行的代码稍后需要恢复，并且不应该意识到发生了任何特殊的事情。也就是说，我们通常希望 trap 是透明的；这对于中断尤其重要，因为被中断的代码通常不会预料到中断的发生。通常的流程是：trap 强制将控制权转移到内核；内核保存寄存器和其它状态，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 trap 返回；原始代码从中断处继续执行。

xv6 内核处理所有陷阱。这对于系统调用来说是自然的。对于中断来说也是有意义的，因为隔离要求用户进程不能直接使用设备，而且只有内核拥有处理设备所需的状态。对于异常来说也是有意义的，因为 xv6 通过杀死违规程序来响应来自用户空间的所有异常。

Xv6 的陷阱处理分为四个阶段：RISC-V CPU 执行的硬件操作、为内核 C 代码做准备的汇编“向量”、决定如何处理陷阱的 C 陷阱处理程序，以及系统调用或设备驱动服务例程。虽然三种陷阱类型的共性表明内核可以使用单一代码路径处理所有陷阱，但事实证明，为三种不同情况分别设置汇编向量和 C 陷阱处理程序更为方便：来自用户空间的陷阱、来自内核空间的陷阱和定时器中断。

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，并且内核可以读取这些寄存器来了解已发生的陷阱。RISC-V 文档包含了完整的信息[1]。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概述：

- stvec: 内核将其陷阱处理程序的地址写入此处；RISC-V 跳转到这里来处理陷阱。

- sepc: 当发生陷阱时，RISC-V 会将程序计数器保存在此处（因为 pc 随后会被 stvec 覆盖）。sret（从陷阱返回）指令将 sepc 复制到 pc。
  内核可以写入 sepc 以控制 sret 的去向。

- scause: RISC-V 在这里放置一个数字，用于描述陷阱的原因。

- sscratch: 内核在这里放置一个值，这个值在陷阱处理程序的最开始非常有用。

- sstatus: sstatus 中的 SIE 位控制设备中断是否启用。如果内核清除 SIE，RISC-V 将推迟设备中断，直到内核设置 SIE。SPP 位指示陷阱是来自用户模式还是监管模式，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相关，它们无法在用户模式下读取或写入。对于在机器模式下处理的陷阱，有一组等效的控制寄存器；xv6 仅在定时器中断的特殊情况下使用它们。
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 正在处理陷阱。
当需要强制陷入时，RISC-V 硬件对所有陷入类型（除了定时器中断）执行以下操作：

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

2. 通过清除 SIE 来禁用中断。

3. 将 pc 复制到 sepc。

4. 将当前模式（用户或监督者）保存在 sstatus 中的 SPP 位。

5. 将 scause 设置为反映陷阱的原因。

6. 将模式设置为 supervisor。

7. 将 stvec 复制到 pc。

8. 从新的 pc 开始执行。注意，CPU 不会切换到内核页表，不会切换到内核中的栈，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 在陷阱期间执行最少工作的一个原因是为软件提供灵活性；例如，某些操作系统在某些情况下不需要页表切换，这可以提高性能。

你可能会想知道 CPU 硬件的陷阱处理序列是否可以进一步简化。例如，假设 CPU 没有切换程序计数器。那么陷阱可能会在仍然运行用户指令的同时切换到监督模式。这些用户指令可能会破坏用户/内核隔离，例如通过修改 satp 寄存器指向一个允许访问所有物理内存的页表。因此，CPU 切换到一个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

如果用户程序进行系统调用（ecall 指令）、执行非法操作或设备中断，可能会在用户空间执行时发生陷阱。从用户空间发生陷阱的高级路径是 uservec（kernel/trampoline.S:16），然后是 usertrap（kernel/trap.c:37）；当重新

转向，usertrapret（kernel/trap.c:90），然后 userret（kernel/trampoline.S:16）。

来自用户代码的陷阱比来自内核的更具挑战性，因为 satp 指向一个不映射内核的用户页表，而且栈指针可能包含无效甚至恶意的值。

因为 RISC-V 硬件在陷入时不切换页表，所以用户页表必须包含对 uservec 的映射，即 stvec 指向的陷入向量指令。uservec 必须切换 satp 以指向内核页表；为了在切换后继续执行指令，uservec 必须在内核页表中与用户页表中相同的地址进行映射。

Xv6 通过包含 uservec 的 trampoline 页面满足这些约束。Xv6 在内核页表和每个用户页表中将 trampoline 页面映射到相同的虚拟地址。这个虚拟地址是 TRAMPOLINE（正如我们在图 2.3 和图 3.3 中看到的）。trampoline 的内容在 trampoline.S 中设置，并且（在执行用户代码时）stvec 被设置为 uservec。

(kernel/trampoline.S:16).

当 `uservec` 开始时，所有 32 个寄存器都包含被中断代码拥有的值。但 `uservec` 需要能够修改一些寄存器以便设置 `satp` 并生成保存寄存器的地址。RISC-V 通过 `sscratch` 寄存器提供了一种帮助。`uservec` 开头的 `csrrw` 指令交换了 `a0` 和 `sscratch` 的内容。现在，用户代码的 `a0` 被保存了；`uservec` 有一个寄存器（`a0`）可以使用；并且 `a0` 包含了内核先前放置在 `sscratch` 中的值。

uservec 的下一个任务是保存用户寄存器。在进入用户空间之前，内核先前已将 sscratch 设置为指向每个进程的 trapframe，该 trapframe（除其他外）有空间保存所有用户寄存器（kernel/proc.h:44）。由于 satp 仍然引用用户页表，

因为 satp 仍然引用用户页表，uservec 需要 trapframe 映射在用户地址空间中。在创建每个进程时，xv6 为进程的 trapframe 分配一个页面，并安排它始终映射在用户虚拟地址 TRAPFRAME 处，该地址位于 TRAMPOLINE 下方。进程的 p->trapframe 也指向 trapframe，尽管是它的物理地址，以便内核可以通过内核页表使用它。

因此，在交换 a0 和 sscratch 之后，a0 持有指向当前进程的 trapframe 的指针。uservec 现在将所有用户寄存器保存在那里，包括用户的 a0，从 sscratch 读取。trapframe 包含指向当前进程的内核栈的指针、当前 CPU 的 hartid、usertrap 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，并调用 usertrap。

usertrap 的任务是确定陷阱的原因，处理它，然后返回（kernel/-trap.c:37) 如上所述，它首先更改 stvec，以便在内核中的陷阱将由 kernelvec 处理。它保存了 sepc（保存的用户程序计数器），再次因为 usertrap 中可能发生进程切换，这可能导致 sepc 被覆盖。如果陷阱是系统调用，syscall 处理它；如果是设备中断，devintr 处理；否则它是一个异常，内核会杀死出错的进程。系统调用路径将保存的用户 pc 增加四，因为在系统调用的情况下，RISC-V 将程序指针指向 ecall 指令。在退出时，usertrap 检查进程是否已被杀死或是否应该让出 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）。该函数设置 RISC-V 控制寄存器，为将来从用户空间的陷阱做准备。这包括将 `stvec` 更改为引用 `uservec`，准备 `uservec` 依赖的 `trapframe` 字段，并将 `sepc` 设置为之前保存的用户程序计数器。最后，`usertrapret` 在用户和内核页表中都映射的 trampoline 页面上调用 `userret`；原因是 `userret` 中的汇编代码将切换页表。

usertrapret 调用 userret 时，将进程的用户页表指针传递给 a0，并将 TRAPFRAME 传递给 a1（kernel/trampoline.S:88）。userret 将 satp 切换到进程的用户页表。回想一下，用户页表映射了 trampoline 页面和 TRAPFRAME，但没有映射内核的其他部分。再次强调，trampoline 页面在用户和内核页表中映射到相同的虚拟地址，这使得 uservec 在更改 satp 后能够继续执行。

userret 将 trapframe 中保存的用户 a0 复制到 sscratch，为稍后与 TRAPFRAME 的交换做准备。从这一点开始，userret 可以使用的唯一数据是寄存器内容和 trapframe 的内容。接下来，userret 从 trapframe 中恢复保存的用户寄存器，最后交换 a0 和 sscratch 以恢复用户 a0 并为下一次陷阱保存 TRAPFRAME，然后使用 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 系统调用的实现的。
用户代码将 exec 的参数放入寄存器 a0 和 a1 中，并将系统调用号放入 a7 中。系统调用号与 syscalls 数组中的条目匹配，这是一个函数表。

一个函数指针表（kernel/syscall.c:108）。ecall 指令陷入内核并执行 uservec、usertrap，然后执行 syscall，正如我们上面所看到的。

syscall (kernel/syscall.c:133) 从保存在 trapframe 中的 a7 中检索系统调用号，并使用它来索引 syscalls。对于第一个系统调用，a7 包含 SYS_exec (kernel/syscall.h:8)，导致调用系统调用实现函数 sys_exec。

当系统调用实现函数返回时，`syscall`将其返回值记录在`p->trapframe->a0`中。这将导致原始用户空间调用`exec()`返回该值，因为在 RISC-V 上的 C 调用约定中，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：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。函数 argint、argaddr 和 argfd 从陷阱帧中检索第 n 个系统调用参数，分别作为整数、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:35）。

一些系统调用将指针作为参数传递，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了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，用户程序可能存在错误或恶意，可能会向内核传递一个无效的指针，或者一个旨在欺骗内核访问内核内存而非用户内存的指针。其次，xv6 内核的页表映射与用户页表映射不同，因此内核无法使用普通指令从用户提供的地址加载或存储数据。

内核实现了安全地在用户提供的地址之间传输数据的功能。
fetchstr 是一个例子（kernel/syscall.c:25）。文件系统调用（如 exec）使用 fetchstr 从用户空间检索字符串文件名参数。fetchstr 调用 copyinstr 来完成繁重的工作。

从用户页表 `pagetable` 中的虚拟地址 `srcva` 复制最多 `max` 字节到 `dst`。它使用 `walkaddr`（调用 `walk`）在软件中遍历页表，以确定 `srcva` 的物理地址 `pa0`。由于内核将所有物理 RAM 地址映射到相同的内核虚拟地址，`copyinstr` 可以直接将字符串字节从 `pa0` 复制到 `dst`。
`walkaddr` (kernel/vm.c:95) 检查用户提供的虚拟地址是否属于进程的用户地址空间，因此程序无法欺骗内核读取其他内存。

copyinstr (kernel/vm.c:406)

一个类似的函数，copyout，将数据从内核复制到用户提供的地址。

4.5 内核空间的陷阱

Xv6 根据用户代码或内核代码的执行情况，对 CPU 的陷阱寄存器进行了不同的配置。当内核在 CPU 上执行时，内核将 stvec 指向 kernelvec 处的汇编代码（kernel/kernelvec.S:10）。由于 xv6 已经处于内核态，kernelvec 可以依赖 satp 被设置为内核页表，并且栈指针指向一个有效的内核栈。kernelvec 保存所有寄存器，以便被中断的代码最终能够不受干扰地恢复执行。

kernelvec 将寄存器保存在被中断的内核线程的栈上，这是有意义的，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，这一点尤为重要——在这种情况下，陷阱实际上会在新线程的栈上返回，将被中断线程的保存寄存器安全地留在其栈上。

kernelvec 在保存寄存器后跳转到 kerneltrap (kernel/trap.c:134)。kerneltrap 为两种类型的陷阱做好准备：设备中断和异常。它调用 devintr (kernel/trap.c:177) 来检查并处理前者。如果陷阱不是设备中断，那么它必须是一个异常，并且在 xv6 内核中发生的异常总是致命的错误；内核会调用 panic 并停止执行。

如果 kerneltrap 是由于定时器中断而被调用，并且一个进程的内核线程正在运行（而不是调度器线程），kerneltrap 会调用 yield 以给其他线程运行的机会。在某个时刻，其中一个线程会 yield，让我们的线程及其 kerneltrap 再次恢复运行。第 7 章解释了 yield 中发生的情况。

当 kerneltrap 的工作完成后，它需要返回到被陷阱中断的代码。由于 yield 可能已经扰乱了 sstatus 中保存的 sepc 和之前模式，kerneltrap 在开始时保存它们。现在它恢复这些控制寄存器并返回到 kernelvec（kernel/kernelvec.S:48）。kernelvec 从堆栈中弹出保存的寄存器并执行 sret，这将 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 CPU 从用户空间进入内核时将其 stvec 设置为 kernelvec；你可以在 usertrap (kernel/trap.c:29) 中看到这一点。在内核执行但 stvec 设置为 uservec 的时间窗口内，禁用设备中断至关重要。幸运的是，RISC-V 在开始处理陷阱时总是禁用中断，而 xv6 在设置 stvec 之前不会再次启用它们。

4.6　页错误异常

Xv6 对异常的处理相当简单：如果异常发生在用户空间，内核会终止出错的进程。如果异常发生在内核中，内核会崩溃。真正的操作系统通常会以更有趣的方式响应。

例如，许多内核使用页面错误来实现写时复制（COW）fork。为了解释写时复制 fork，考虑 xv6 的 fork，如第 3 章所述。fork 通过调用 uvmcopy（kernel/vm.c:309）来分配内存，使子进程拥有与父进程相同的内存内容。

c:309) 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能够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，因为这会导致父进程和子进程通过对共享栈和堆的写入互相干扰对方的执行。

父进程和子进程可以通过写时复制（copy-on-write）的 fork 安全地共享物理内存，这是由页面错误驱动的。当 CPU 无法将虚拟地址转换为物理地址时，CPU 会生成一个页面错误异常。RISC-V 有三种不同类型的页面错误：加载页面错误（当加载指令无法转换其虚拟地址时）、存储页面错误（当存储指令无法转换其虚拟地址时）和指令页面错误（当指令的地址无法转换时）。scause 寄存器中的值指示页面错误的类型，stval 寄存器包含无法转换的地址。

COW fork 的基本计划是让父进程和子进程最初共享所有物理页面，但将它们映射为只读。因此，当子进程或父进程执行存储指令时，RISC-V CPU 会引发页面错误异常。作为对此异常的响应，内核会复制包含错误地址的页面。它将一个副本映射为子进程地址空间中的读/写，另一个副本映射为父进程地址空间中的读/写。更新页表后，内核在引发错误的指令处恢复故障进程。由于内核已更新相关 PTE 以允许写入，故障指令现在将无错误地执行。

这个 COW（写时复制）方案在 fork 操作中表现良好，因为通常子进程在 fork 后会立即调用 exec，用新的地址空间替换其原有的地址空间。在这种常见情况下，子进程只会遇到少量的页错误，内核可以避免进行完整的复制。此外，COW fork 是透明的：应用程序无需任何修改即可从中受益。

页表和页错误的结合为除了 COW fork 之外的其他有趣可能性打开了大门。另一个广泛使用的特性称为惰性分配，它包含两个部分。首先，当应用程序调用`sbrk`时，内核会扩展地址空间，但在页表中将新地址标记为无效。其次，当这些新地址发生页错误时，内核会分配物理内存并将其映射到页表中。由于应用程序通常请求的内存比实际需要的多，惰性分配是一个优势：内核仅在应用程序实际使用时才分配内存。与 COW fork 一样，内核可以透明地实现这一特性，对应用程序无感知。

另一个广泛使用的利用页面错误的功能是从磁盘分页。如果应用程序需要的内存超过了可用的物理 RAM，内核可以驱逐一些页面：将它们写入磁盘等存储设备，并将它们的 PTE 标记为无效。如果应用程序读取或写入被驱逐的页面，CPU 将遇到页面错误。然后，内核可以检查错误地址。如果该地址属于磁盘上的页面，内核会分配一页物理内存，将该页面从磁盘读取到该内存中，更新 PTE 使其有效并引用该内存，然后恢复应用程序。为了为该页面腾出空间，内核可能不得不驱逐另一个页面。此功能无需对应用程序进行任何更改，并且如果应用程序具有引用局部性（即，在任何给定时间仅使用其内存的一部分），则效果良好。

其他结合分页和页面错误异常的功能包括自动扩展栈和内存映射文件。

4.7 现实世界

如果内核内存被映射到每个进程的用户页表中（带有适当的 PTE 权限标志），则可以消除对特殊跳板页的需求。这也将消除从用户空间陷入内核时进行页表切换的需要。这反过来将使内核中的系统调用实现能够利用当前进程的用户内存映射，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想法来提高效率。Xv6 为了避免由于无意中使用用户指针而导致内核中的安全漏洞，并减少确保用户和内核虚拟地址不重叠所需的复杂性，选择不使用这些方法。

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，使内核映射用户程序，`copyin` 和 `copyinstr` 可以使用 `memcpy` 将系统调用参数复制到内核空间，依赖硬件进行页表遍历。

2. 实现惰性内存分配

3. 实现 COW fork

第 5 章

中断和设备驱动程序

驱动程序是操作系统中管理特定设备的代码：它配置设备硬件，指示设备执行操作，处理产生的中断，并与可能正在等待设备 I/O 的进程交互。驱动程序代码可能很棘手，因为驱动程序与其管理的设备并发执行。此外，驱动程序必须理解设备的硬件接口，这可能很复杂且文档不全。

需要操作系统关注的设备通常可以配置为生成中断，这是一种陷阱。内核陷阱处理代码会识别设备何时引发了中断，并调用驱动程序的中断处理程序；在 xv6 中，这种调度发生在

devintr (kernel/trap.c:177).

许多设备驱动程序在两个上下文中执行代码：一个是在进程的内核线程中运行的顶部部分，另一个是在中断时执行的底部部分。顶部部分通过诸如`read`和`write`等系统调用来调用，这些系统调用希望设备执行 I/O 操作。此代码可能会要求硬件开始操作（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底部部分，确定已完成的操作，如果合适则唤醒等待的进程，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（console.c）是驱动程序结构的一个简单示例。控制台驱动程序通过连接到 RISC-V 的 UART 串口硬件接收人类输入的字符。控制台驱动程序一次累积一行输入，处理特殊输入字符，如退格键和 control-u。用户进程（如 shell）使用 read 系统调用从控制台获取输入行。当你在 QEMU 中向 xv6 输入时，你的按键通过 QEMU 模拟的 UART 硬件传递给 xv6。驱动程序与之通信的 UART 硬件是由 QEMU 模拟的 16550 芯片[11]。在真实的计算机上，16550 会管理一个连接到终端或其他计算机的 RS232 串行链路。

当运行 QEMU 时，它连接到你的键盘和显示器。

UART 硬件对软件来说表现为一组内存映射的控制寄存器。

也就是说，RISC-V 硬件将一些物理地址连接到 UART 设备，因此加载和存储操作会与设备硬件交互，而不是与 RAM 交互。UART 的内存映射地址从 0x10000000 开始，即 UART0（kernel/memlayout.h:21）。UART 有少量控制寄存器，每个寄存器的宽度为一个字节。它们相对于 UART0 的偏移量在（kernel/uart.c:22）中定义。例如，LSR 寄存器包含指示是否有输入字符等待软件读取的位。这些字符（如果有的话）可以从 RHR 寄存器中读取。每次读取一个字符时，UART 硬件会将其从等待字符的内部 FIFO 中删除，并在 FIFO 为空时清除 LSR 中的“就绪”位。UART 的发送硬件与接收硬件基本上是独立的；如果软件向 THR 写入一个字节，UART 会发送该字节。

Xv6 的 main 函数调用 consoleinit（kernel/console.c:184）来初始化 UART 硬件。此代码配置 UART 以在 UART 接收到每个输入字节时生成接收中断，并在 UART 完成发送每个输出字节时生成发送完成中断。

(kernel/uart.c:53).

xv6 shell 通过由 init.c (user/init.c:19) 打开的文件描述符从控制台读取数据。对 read 系统调用的调用会通过内核到达 consoleread (kernel/console.c:82)。consoleread 等待输入到达（通过中断）并缓冲在 cons.buf 中，将输入复制到用户空间，并在整行到达后返回到用户进程。如果用户尚未输入完整行，任何读取进程将在 sleep 调用 (kernel/console.c:98) 中等待（第 7 章详细解释了 sleep 的细节）。

当用户输入一个字符时，UART 硬件请求 RISC-V 引发一个中断，这会激活 xv6 的陷阱处理程序。陷阱处理程序调用 devintr（kernel/trap.c:177），它查看 RISC-V 的 scause 寄存器以发现中断来自外部设备。然后它请求一个称为 PLIC [1]的硬件单元告诉它是哪个设备引发了中断（kernel/trap.c:186）。如果是 UART，devintr 调用 uartintr。

uartintr (kernel/uart.c:180) 从 UART 硬件读取任何等待的输入字符，并将它们传递给 consoleintr (kernel/console.c:138)；它不会等待字符，因为未来的输入会引发新的中断。consoleintr 的任务是将输入字符累积到 cons.buf 中，直到一整行到达。consoleintr 特别处理退格键和一些其他字符。当换行符到达时，consoleintr 会唤醒等待的 consoleread（如果有的话）。

一旦被唤醒，`consoleread` 将观察到 `cons.buf` 中有一整行数据，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进行写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` 以启动设备传输（如果尚未启动），然后返回。`uartputc` 等待的唯一情况是缓冲区已满。

每次 UART 完成发送一个字节时，它会生成一个中断。uartintr 调用 uartstart，

它检查设备是否真的已完成发送，并将下一个缓冲的输出字符交给设备。因此，如果进程向控制台写入多个字节，通常第一个字节将由 uartputc 调用 uartstart 发送，而剩余的缓冲字节将在传输完成中断到达时由 uartintr 调用 uartstart 发送。

一个值得注意的通用模式是通过缓冲和中断将设备活动与进程活动解耦。即使没有进程在等待读取输入，控制台驱动程序也可以处理输入；随后的读取操作将看到这些输入。同样，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，当设备速度较慢（如 UART）或需要立即响应（如回显键入的字符）时，这一点尤为重要。这种思想有时被称为 I/O 并发。

5.3 驱动程序中的并发性

你可能已经注意到在`consoleread`和`consoleintr`中调用了`acquire`。这些调用获取了一个锁，用于保护控制台驱动程序的数据结构免受并发访问。这里有三种并发危险：不同 CPU 上的两个进程可能同时调用`consoleread`；硬件可能在一个 CPU 已经在执行`consoleread`时要求该 CPU 传递一个控制台（实际上是 UART）中断；以及硬件可能在`consoleread`执行时在另一个 CPU 上传递控制台中断。第 6 章探讨了锁在这些场景中的作用。

并发性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设备的输入，但输入到达的中断信号可能在不同的进程（或根本没有进程）运行时到达。因此，中断处理程序不允许考虑它们中断的进程或代码。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。中断处理程序通常只做相对较少的工作（例如，只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时钟，并使其能够在计算密集型进程之间切换；usertrap 和 kerneltrap 中的 yield 调用导致这种切换。定时器中断来自连接到每个 RISC-V CPU 的时钟硬件。Xv6 对这个时钟硬件进行编程，使其定期中断每个 CPU。

RISC-V 要求定时器中断必须在机器模式下处理，而不是在监督模式下。RISC-V 机器模式在没有分页的情况下执行，并且使用一组独立的控制寄存器，因此在机器模式下运行普通的 xv6 内核代码是不实际的。因此，xv6 完全独立于上述的陷阱机制来处理定时器中断。

在 start.c 中执行的机器模式代码，在 main 之前，设置接收定时器中断(kernel/start.c:57) . 部分工作是对 CLINT 硬件（核心本地中断器）进行编程，以在一定延迟后生成中断。另一部分是设置一个类似于

类似于 trapframe，用于帮助定时器中断处理程序保存寄存器和 CLINT 寄存器的地址。
最后，`start` 将 `mtvec` 设置为 `timervec` 并启用定时器中断。
定时器中断可以在用户或内核代码执行的任何时刻发生；内核无法在关键操作期间禁用定时器中断。因此，定时器中断处理程序必须以一种保证不会干扰被中断的内核代码的方式完成其工作。基本策略是让处理程序请求 RISC-V 引发一个“软件中断”并立即返回。RISC-V 通过普通的陷阱机制将软件中断传递给内核，并允许内核禁用它们。处理由定时器中断生成的软件中断的代码可以

在 devintr（kernel/trap.c:204）中看到。

机器模式的定时器中断向量是 timervec（kernel/kernelvec.S:93）。它在 start 准备的暂存区中保存了一些寄存器，告诉 CLINT 何时生成下一个定时器中断，请求 RISC-V 引发一个软件中断，恢复寄存器，然后返回。定时器中断处理程序中没有 C 代码。

5.5 现实世界

Xv6 允许在执行内核代码时以及执行用户程序时发生设备和定时器中断。定时器中断会强制从定时器中断处理程序中进行线程切换（调用 yield），即使是在执行内核代码时也是如此。如果内核线程有时花费大量时间进行计算而不返回用户空间，那么公平地在内核线程之间进行 CPU 时间切片的能力是有用的。然而，内核代码需要意识到它可能会被挂起（由于定时器中断）并在稍后在不同的 CPU 上恢复，这是 xv6 中一些复杂性的来源。如果设备和定时器中断仅在执行用户代码时发生，内核可能会变得更简单一些。

支持典型计算机上的所有设备并充分发挥其功能是一项艰巨的工作，因为设备众多，功能丰富，且设备与驱动程序之间的协议可能复杂且文档不完善。在许多操作系统中，驱动程序的代码量超过了核心内核。

UART 驱动程序通过读取 UART 控制寄存器一次检索一个字节的数据；这种模式称为程序控制 I/O，因为软件驱动数据移动。程序控制 I/O 简单，但在高数据速率下速度太慢。需要高速传输大量数据的设备通常使用直接内存访问（DMA）。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，然后通过一次写入控制寄存器来告诉设备处理准备好的数据。

当设备需要在不可预测的时间且不太频繁地引起注意时，中断是有意义的。
但是中断的 CPU 开销很高。因此，高速设备（如网络和磁盘控制器）使用一些技巧来减少中断的需求。一个技巧是为整个批量的传入或传出请求引发单个中断。另一个技巧是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。如果设备执行操作非常快，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。一些驱动程序在轮询和中断之间动态切换。

一些驱动程序根据当前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的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，但对于那些生成或消耗数据非常快的设备来说，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。

5.6 练习

1. 修改 uart.c 以完全不使用中断。你可能还需要修改 console.c。

2. 添加一个以太网卡的驱动程序。

第 6 章

锁定

大多数内核，包括 xv6，都会交错执行多个活动。交错的一个来源是多处理器硬件：具有多个独立执行的 CPU 的计算机，例如 xv6 的 RISC-V。这些多个 CPU 共享物理 RAM，xv6 利用这种共享来维护所有 CPU 读取和写入的数据结构。这种共享带来了一个 CPU 在另一个 CPU 正在更新数据结构时读取它的可能性，甚至多个 CPU 同时更新相同的数据；如果没有仔细设计，这种并行访问可能会导致不正确的结果或损坏的数据结构。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在错误的时间修改了与某些可中断代码相同的数据，可能会损坏数据。并发一词指的是由于多处理器并行性、线程切换或中断而导致多个指令流交错的情况。

内核中充满了并发访问的数据。例如，两个 CPU 可能同时调用 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多种并发，因为它可以通过并行性提高性能，并增加响应性。然而，因此内核设计者花费大量精力来确保在这种并发情况下的正确性。有许多方法可以实现正确的代码，其中一些比其他方法更容易推理。旨在并发下确保正确性的策略，以及支持它们的抽象，被称为并发控制技术。

Xv6 根据情况使用了多种并发控制技术；还有许多其他可能的技术。本章重点介绍一种广泛使用的技术：锁。锁提供了互斥性，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，并且代码在使用数据项时始终持有相关的锁，那么该数据项一次只能被一个 CPU 使用。在这种情况下，我们说锁保护了数据项。尽管锁是一种易于理解的并发控制机制，但锁的缺点是它们可能会降低性能，因为它们将并发操作串行化。

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。

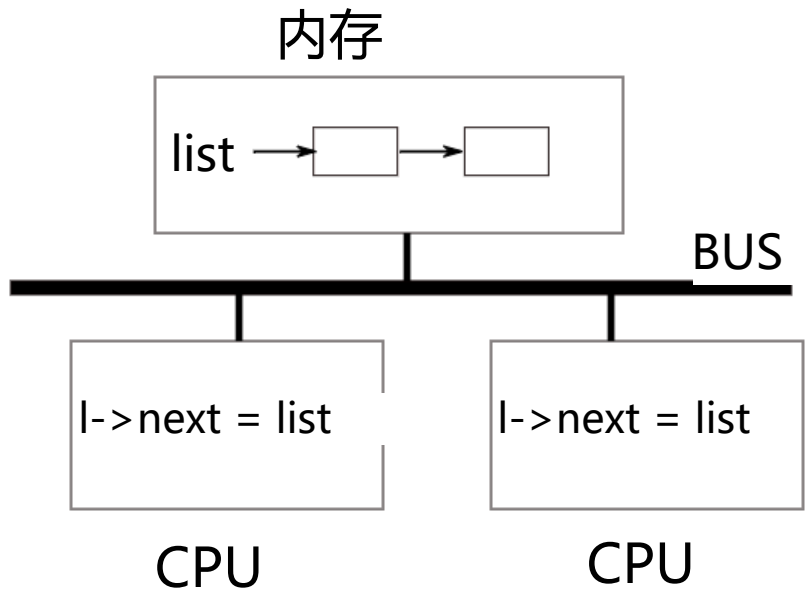

图 6.1：简化的 SMP 架构

6.1 竞争条件

作为我们需要锁的一个例子，考虑两个进程在两个不同的 CPU 上调用 wait。wait 会释放子进程的内存。因此，在每个 CPU 上，内核将调用 kfree 来释放子进程的页面。内核分配器维护一个链表：kalloc()（kernel/kalloc.c:69）从空闲页面列表中弹出一个内存页面，而 kfree()（kernel/kalloc.c:47）将一个页面推入空闲列表。为了获得最佳性能，我们可能希望两个父进程的 kfree 能够并行执行，而不必等待对方，但根据 xv6 的 kfree 实现，这是不正确的。

图 6.1 更详细地展示了这一设置：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链表。（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有一个单一的共享内存。）如果没有并发请求，你可能会像下面这样实现一个列表推送操作：

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

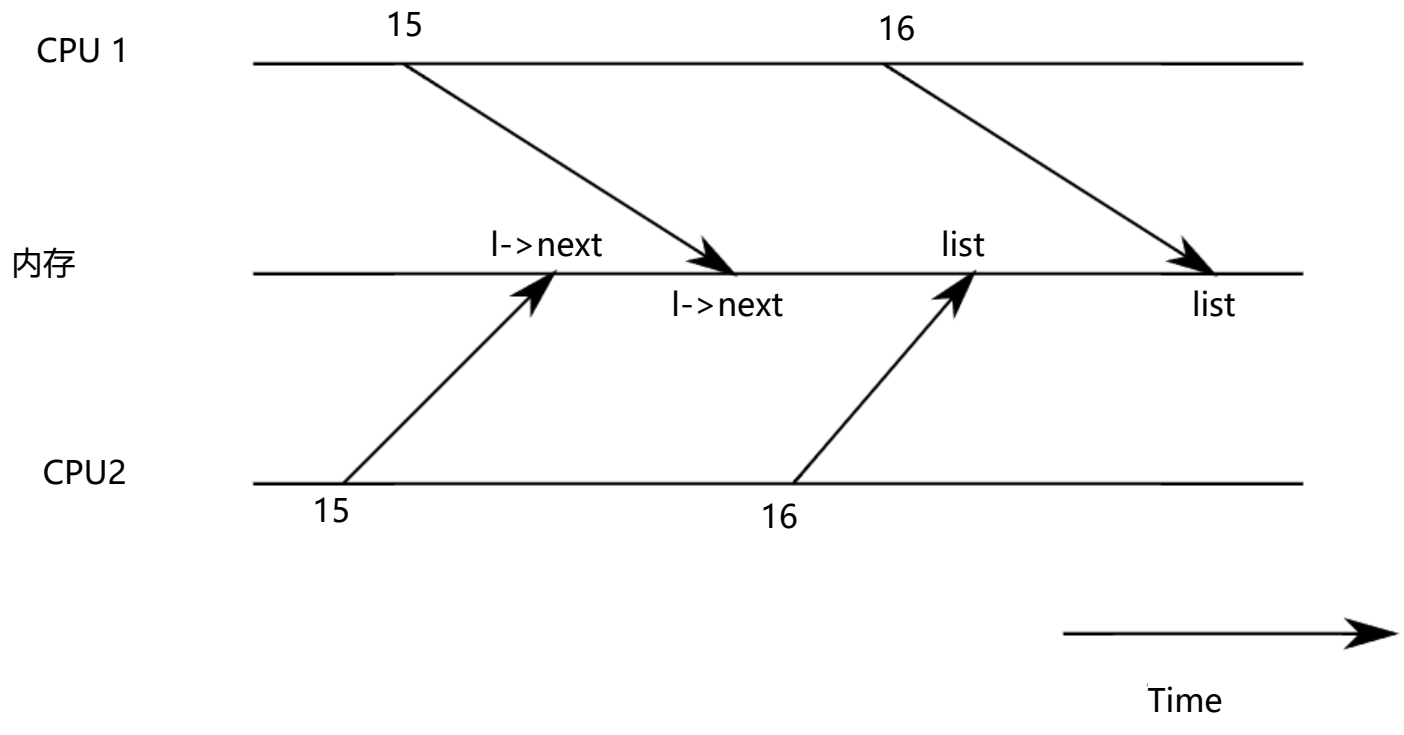

图 6.2：示例竞争

```
17      }
```

如果单独执行，这个实现是正确的。然而，如果有多个副本同时执行，代码就不正确了。如果两个 CPU 同时执行`push`，它们可能会在任何一个执行第 16 行之前都执行第 15 行，如图 6.1 所示，这将导致如图 6.2 所示的错误结果。然后会有两个列表元素的`next`设置为`list`的先前值。当第 16 行的两个赋值发生时，第二个赋值将覆盖第一个；涉及第一个赋值的元素将丢失。

第 16 行的丢失更新是竞态条件的一个例子。竞态条件是指内存位置被并发访问，且至少有一次访问是写操作的情况。竞态通常是错误的标志，可能是丢失更新（如果访问是写操作）或读取未完全更新的数据结构。竞态的结果取决于所涉及的两个 CPU 的确切时间安排以及内存系统如何排序它们的内存操作，这使得由竞态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间安排，足以使竞态消失。

避免竞争条件的常用方法是使用锁。锁确保互斥，因此一次只有一个 CPU 可以执行 push 的敏感代码行；这使得上述场景不可能发生。上述代码的正确锁定版本只需添加几行（以黄色高亮显示）：

```
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      }
```

在`acquire`和`release`之间的指令序列通常称为临界区。
锁通常被称为保护列表。
当我们说锁保护数据时，实际上是指锁保护了适用于数据的一些不变量的集合。不变量是数据结构在操作之间保持的属性。通常，操作的正确行为依赖于操作开始时不变量的真实性。操作可能会暂时违反不变量，但必须在完成之前重新建立它们。例如，在链表的例子中，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一不变量：在第 17 行，`l`指向下一个列表元素，但`list`尚未指向`l`（在第 18 行重新建立）。我们之前检查的竞态条件发生的原因是第二个 CPU 在（暂时）违反不变量时执行了依赖于列表不变量的代码。正确使用锁确保在临界区中一次只有一个 CPU 可以对数据结构进行操作，因此当数据结构的不变量不成立时，不会有 CPU 执行数据结构操作。

你可以将锁视为将并发的临界区序列化，使它们一次只运行一个，从而保持不变量（假设临界区在单独情况下是正确的）。你也可以将受同一锁保护的临界区视为彼此原子性，这样每个临界区只能看到之前临界区的完整更改集，而永远不会看到部分完成的更新。

虽然正确使用锁可以使不正确的代码变得正确，但锁会限制性能。例如，如果两个进程同时调用`kfree`，锁将串行化这两个调用，我们在不同 CPU 上运行它们时不会获得任何好处。我们说，如果多个进程同时想要同一个锁，或者锁遇到争用，那么这些进程就会发生冲突。内核设计中的一个主要挑战是避免锁争用。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以避免锁争用。在列表的例子中，内核可能会为每个 CPU 维护一个空闲列表，只有在 CPU 的列表为空且必须从另一个 CPU 窃取内存时，才会触及另一个 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，将`acquire`提前到`push`中是可行的：可以将`acquire`的调用移到第 13 行之前。这可能会降低性能，因为对`malloc`的调用也会被串行化。下面的“使用锁”部分提供了一些关于在何处插入`acquire`和`release`调用的指导原则。

6.2 代码：Locks

Xv6 有两种类型的锁：自旋锁和睡眠锁。我们将从自旋锁开始。Xv6 将自旋锁表示为一个结构体 spinlock（kernel/spinlock.h:2）。该结构中的重要字段是

结构中的重要字段是`locked`，当锁可用时该字段为零，当锁被持有时为非零。从逻辑上讲，xv6 应该通过执行类似以下的代码来获取锁：

```
21      void
22      acquire(struct spinlock *lk) // 不起作用!
23      {
24        for(;;) {
25          如果(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。我们需要一种方法使第 25 行和第 26 行作为一个原子（即不可分割的）步骤执行。
因为锁被广泛使用，多核处理器通常提供实现第 25 行和第 26 行的原子版本的指令。在 RISC-V 中，这个指令是 amoswap r, a。
amoswap 读取内存地址 a 处的值，将寄存器 r 的内容写入该地址，并将读取的值放入 r。也就是说，它交换了寄存器和内存地址的内容。它使用特殊的硬件原子地执行这一序列，以防止任何其他 CPU 在读取和写入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，这归结为`amoswap`指令；返回值是`lk->locked`的旧（交换后）内容。`acquire`函数将交换操作包装在一个循环中，不断重试（自旋），直到获取锁为止。每次迭代都会将 1 交换到`lk->locked`中，并检查之前的值；如果之前的值是 0，那么我们已经获取了锁，交换操作会将`lk->locked`设置为 1。如果之前的值是 1，那么其他 CPU 持有锁，我们原子地将 1 交换到`lk->locked`中并没有改变它的值。

一旦锁被获取，`acquire`会记录获取锁的 CPU，用于调试。
lk->cpu 字段受锁保护，只有在持有锁时才能更改。

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，`release` 只需要将 `lk->locked` 赋值为零。C 标准允许编译器使用多个存储指令来实现赋值，因此 C 赋值在并发代码中可能不是原子的。相反，`release` 使用 C 库函数 `__sync_lock_release` 来执行原子赋值。这个函数最终也会归结为 RISC-V 的 `amoswap` 指令。

6.3 代码：使用锁

Xv6 在许多地方使用锁来避免竞争条件。如上所述，kalloc (kernel/kalloc.c:69) 和 kfree (kernel/kalloc.c:47) 是一个很好的例子。尝试练习 1 和 2，看看如果这些函数省略锁会发生什么。你可能会发现很难触发错误行为，这表明很难可靠地测试代码是否没有锁定错误和竞争。xv6 可能存在一些竞争条件。

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

上述规则说明了何时需要锁，但并未说明何时不需要锁。为了提高效率，重要的是不要过度加锁，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，而不必担心锁的问题。一个简单的内核可以通过在进入内核时获取一个锁并在退出内核时释放该锁（尽管像管道读取或等待这样的系统调用会带来问题）在多处理器上实现这一点。许多单处理器操作系统已经通过这种方法（有时称为“大内核锁”）转换为在多处理器上运行，但这种方法牺牲了并行性：一次只能有一个 CPU 在内核中执行。如果内核进行大量计算，使用一组更细粒度的锁会更高效，这样内核可以在多个 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一空闲列表。如果不同 CPU 上的多个进程同时尝试分配页面，每个进程都必须在 acquire 中自旋等待其轮次。自旋会降低性能，因为它不是有用的工作。如果锁的争用浪费了大量 CPU 时间，或许可以通过更改分配器设计以拥有多个空闲列表，每个列表都有自己的锁，从而允许真正的并行分配来提高性能。

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，因此操作不同文件的进程通常可以继续执行，而无需等待彼此的锁。如果希望允许进程同时写入同一文件的不同区域，文件锁定方案可以更加细粒度。最终，锁粒度的决策需要由性能测量和复杂性考虑来驱动。

随着后续章节对 xv6 的各个部分进行解释，它们将提到 xv6 使用锁来处理并发性的示例。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非常重要的。如果不是这样，就有可能发生死锁。假设 xv6 中的两个代码路径需要锁 A 和 B，但代码路径 1 以 A 然后 B 的顺序获取锁，

Lock	描述
bcache.lock	保护块缓冲区缓存条目的分配
cons.lock	串行化对控制台硬件的访问，避免输出混杂
ftable.lock	串行化文件表中 struct file 的分配
icache.lock	保护 inode 缓存条目的分配
vdisk_lock	串行化对磁盘硬件和 DMA 描述符队列的访问
kmem.lock	串行化内存的分配
log.lock	串行化事务日志的操作
pipe’s pi->lock	串行化每个管道的操作
pid_lock	串行化 next_pid 的递增
proc’s p->lock	串行化进程状态的更改
tickslock	串行化对 ticks 计数器的操作
inode’s ip->lock	串行化每个 inode 及其内容的操作
buf’s b->lock	串行化每个块缓冲区的操作

图 6.3：xv6 中的锁

但代码路径 1 以 A 然后 B 的顺序获取锁，而另一条路径以 B 然后 A 的顺序获取锁。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，T2 将尝试获取锁 A。这两个获取操作都将无限期地阻塞，因为在两种情况下，另一个线程都持有所需的锁，并且在其获取操作返回之前不会释放它。为了避免这种死锁，所有代码路径必须以相同的顺序获取锁。全局锁获取顺序的需求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。

Xv6 有许多长度为二的锁顺序链，涉及每个进程的锁（每个 struct proc 中的锁），这是由于 sleep 的工作方式（见第 7 章）。例如，consoleintr（kernel/console.c:138）是处理键入字符的中断例程。当换行符到达时，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，而 wakeup 会获取等待进程的锁以唤醒它。因此，全局死锁避免的锁顺序包括一条规则：必须在获取任何进程锁之前获取 cons.lock。文件系统代码包含 xv6 中最长的锁链。例如，创建一个文件需要同时持有目录的锁、新文件的 inode 的锁、磁盘块缓冲区的锁、磁盘驱动器的 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照前一句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。有时锁的顺序与程序的逻辑结构相冲突，例如，代码模块 M1 调用模块 M2，但锁的顺序要求 M2 中的锁在 M1 中的锁之前获取。有时锁的身份事先未知，可能是因为必须持有某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中的连续组件时会出现，以及在等待和退出代码中搜索进程表以查找子进程时也会出现。最后，死锁的危险常常是

死锁的危险通常限制了锁方案的细粒度，因为更多的锁通常意味着更多的死锁机会。避免死锁的需求通常是内核实现中的一个主要因素。

6.5 锁和中断处理程序

一些 xv6 自旋锁保护了由线程和中断处理程序共同使用的数据。例如，clockintr 定时器中断处理程序可能会在大约同一时间递增 ticks（kernel/trap.c:163）。
内核线程在 sys_sleep（kernel/sysproc.c:64）中读取 ticks 的时间。锁 tickslock
序列化这两个访问。
自旋锁和中断的交互带来了潜在的危险。假设 `sys_sleep` 持有 `tickslock`，并且它的 CPU 被一个定时器中断打断。`clockintr` 会尝试获取
tickslock，发现它被持有，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。因此，CPU 将死锁，任何需要这两个锁的代码也会冻结。
为了避免这种情况，如果自旋锁被中断处理程序使用，CPU 在持有该锁时必须禁用中断。Xv6 更为保守：当 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，因此中断的获取可以等待线程释放自旋锁；只是不在同一个 CPU 上。

xv6 在 CPU 不持有自旋锁时重新启用中断；它必须进行一些簿记工作
处理嵌套的临界区。`acquire`调用`push_off`（kernel/spinlock.c:89），而`release`
调用 `pop_off` (kernel/spinlock.c:100) 来跟踪当前 CPU 上的锁的嵌套级别。当该计数达到零时，`pop_off` 会恢复在最外层临界区开始时存在的中断启用状态。`intr_off` 和 `intr_on` 函数分别执行 RISC-V 指令来禁用和启用中断。
重要的是，acquire 在设置 lk->locked 之前必须严格调用 push_off（kernel/spinlock.c:28）。如果两者顺序颠倒，将会有一个短暂的时间窗口，锁被持有且中断被启用，不幸的中断时间会导致系统死锁。同样，release 必须在释放锁之后才调用 pop_off（kernel/spinlock.c:66）。

6.6 指令和内存排序

很自然地，我们会认为程序是按照源代码语句出现的顺序执行的。然而，许多编译器和 CPU 为了提高性能，会以乱序的方式执行代码。如果一条指令需要多个周期才能完成，CPU 可能会提前发出该指令，以便它能与其他指令重叠执行，从而避免 CPU 停顿。例如，CPU 可能会注意到在连续的指令序列中，指令 A 和 B 彼此之间没有依赖关系。CPU 可能会先开始执行指令 B，要么是因为它的输入在 A 的输入之前就准备好了，要么是为了重叠执行 A 和 B。编译器也可能通过为一个语句生成指令，早于源代码中位于它之前的语句的指令，来进行类似的重新排序。
编译器和 CPU 在重新排序时遵循规则，以确保它们不会改变正确编写的串行代码的结果。然而，这些规则确实允许重新排序，从而改变

规则确实允许重新排序，这会改变并发代码的结果，并且很容易导致多处理器上的不正确行为[2, 3]。CPU 的排序规则被称为内存模型。

例如，在这段用于`push`的代码中，如果编译器或 CPU 将第 4 行对应的存储操作移动到第 6 行的释放操作之后，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);
```

如果发生了这样的重排序，将会有一个窗口期，在此期间另一个 CPU 可以获取锁并观察到更新后的列表，但看到一个未初始化的`list->next`。

为了告诉硬件和编译器不要执行这种重新排序，xv6 使用 __sync_synchronize() 在 acquire（kernel/spinlock.c:22）和 release（kernel/spinlock.c:47）中。__sync_synchronize() 是一个内存屏障：它告诉编译器和 CPU 不要在屏障前后重新排序加载或存储操作。xv6 中的获取和释放屏障在几乎所有重要的情况下都强制了顺序，因为 xv6 在访问共享数据时使用了锁。第 9 章讨论了一些例外情况。

6.7 Sleep locks

有时 xv6 需要长时间持有锁。例如，文件系统（第 8 章）在读取和写入磁盘上的文件内容时保持文件锁定，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这个锁，持有自旋锁这么长时间会导致浪费，因为获取锁的进程会在自旋时长时间浪费 CPU。自旋锁的另一个缺点是，进程在持有自旋锁时不能放弃 CPU；我们希望这样做，以便在持有锁的进程等待磁盘时，其他进程可以使用 CPU。在持有自旋锁时放弃 CPU 是非法的，因为如果第二个线程随后尝试获取自旋锁，可能会导致死锁；由于 acquire 不会放弃 CPU，第二个线程的自旋可能会阻止第一个线程运行并释放锁。在持有锁时放弃 CPU 也会违反自旋锁持有期间必须关闭中断的要求。因此，我们需要一种在等待获取锁时放弃 CPU，并在持有锁时允许放弃 CPU（和中断）的锁类型。

Xv6 以睡眠锁的形式提供了这样的锁。`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。在高层次上，睡眠锁有一个由自旋锁保护的 `locked` 字段，`acquiresleep` 调用 `sleep` 原子地让出 CPU 并释放自旋锁。结果是，在 `acquiresleep` 等待时，其他线程可以执行。

因为睡眠锁在保持中断使能的情况下，它们不能在中断处理程序中使用。由于 acquiresleep 可能会让出 CPU，睡眠锁不能在自旋锁的临界区内使用（尽管自旋锁可以在睡眠锁的临界区内使用）。

自旋锁最适合用于短临界区，因为等待它们会浪费 CPU 时间；睡眠锁适用于长时间的操作。

6.8 现实世界

尽管在并发原语和并行性方面进行了多年的研究，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。如果你使用锁进行编程，明智的做法是使用一种工具来尝试识别竞争条件，因为很容易忽略需要锁的不变量。

大多数操作系统支持 POSIX 线程（Pthreads），它允许用户进程在不同的 CPU 上并发运行多个线程。Pthreads 支持用户级锁、屏障等。支持 Pthreads 需要操作系统的支持。例如，如果一个 pthread 在系统调用中阻塞，同一进程的另一个 pthread 应该能够在该 CPU 上运行。再比如，如果一个 pthread 更改了其进程的地址空间（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 更新其硬件页表，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 [8]，但这样做代价高昂，大多数操作系统使用原子指令。

如果许多 CPU 同时尝试获取同一个锁，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，并且可能会使该缓存行的其他副本失效。从另一个 CPU 的缓存中获取缓存行可能比从本地缓存中获取一行要昂贵得多。

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 [5, 10]。例如，可以实现一个类似于本章开头的链表，在链表搜索期间不需要锁，并且只需一条原子指令即可在链表中插入一个项目。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的问题。使用锁编程已经很难了，因此 xv6 避免了无锁编程的额外复杂性。

6.9 练习

1. 注释掉 kalloc 中对 acquire 和 release 的调用（kernel/kalloc.c:69）。这似乎应该会导致调用 kalloc 的内核代码出现问题；你预计会看到什么症状？当你运行 xv6 时，你看到这些症状了吗？运行 usertests 时呢？如果你没有看到问题，为什么？尝试通过在 kalloc 的临界区插入虚拟循环来引发问题。
2. 假设你改为在 kfree 中注释掉锁（在恢复 kalloc 中的锁之后）。现在可能会出现什么问题？kfree 中缺少锁是否比在 kalloc 中缺少锁的危害更小？

kalloc?

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 以增加并行性，使得来自不同 CPU 的 kalloc 调用可以同时进行而无需相互等待。

4. 使用 POSIX 线程编写一个并行程序，这在大多数操作系统中都得到支持。例如，实现一个并行哈希表，并测量 put/get 操作的数量是否随着核心数量的增加而扩展。

5. 在 xv6 中实现 Pthreads 的一个子集。即实现一个用户级线程库，使得用户进程可以拥有多个线程，并安排这些线程可以在不同的 CPU 上并行运行。设计一个能够正确处理线程进行阻塞系统调用和更改其共享地址空间的方案。

第 7 章

调度

任何操作系统都可能运行比计算机拥有的 CPU 更多的进程，因此需要一个计划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将进程多路复用到硬件 CPU 上，为每个进程提供拥有自己虚拟 CPU 的假象。本章解释了 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进行多路复用。首先，xv6 的 sleep 和 wakeup 机制在进程等待设备或管道 I/O 完成、等待子进程退出或在 sleep 系统调用中等待时进行切换。其次，xv6 定期强制切换以处理长时间计算而不睡眠的进程。这种多路复用创建了每个进程拥有自己的 CPU 的假象，就像 xv6 使用内存分配器和硬件页表创建每个进程拥有自己的内存的假象一样。

实现多路复用面临几个挑战。首先，如何从一个进程切换到另一个进程？尽管上下文切换的概念很简单，但其实现是 xv6 中最不透明的代码之一。其次，如何以对用户进程透明的方式强制切换？Xv6 使用定时器中断驱动上下文切换的标准技术。第三，许多 CPU 可能同时在不同进程之间切换，需要一个锁定计划来避免竞争。第四，进程退出时必须释放其内存和其他资源，但它不能自己完成所有这些操作，因为（例如）它不能在使用自己的内核栈时释放它。第五，多核机器的每个核心必须记住它正在执行哪个进程，以便系统调用影响正确的进程的内核状态。最后，sleep 和 wakeup 允许一个进程放弃 CPU 并睡眠等待事件，并允许另一个进程唤醒第一个进程。需要小心避免导致唤醒通知丢失的竞争。Xv6 试图尽可能简单地解决这些问题，但最终的代码仍然很复杂。

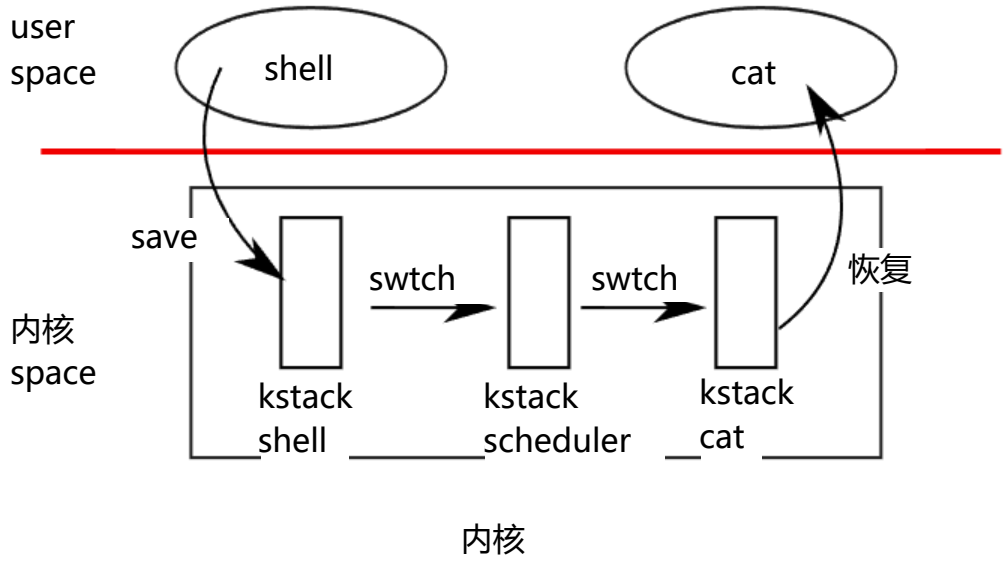

图 7.1：从一个用户进程切换到另一个用户进程。在此示例中，xv6 运行在一个 CPU 上（因此只有一个调度器线程）。

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：用户-内核转换（系统调用或中断）到旧进程的内核线程，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 调度器为每个 CPU 都有一个专用线程（保存的寄存器和堆栈），因为在旧进程的内核堆栈上执行调度器是不安全的：其他核心可能会唤醒该进程并运行它，而在两个不同的核心上使用相同的堆栈将是一场灾难。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，并恢复新线程之前保存的寄存器；栈指针和程序计数器被保存和恢复的事实意味着 CPU 将切换栈并切换它正在执行的代码。

函数 `swtch` 执行内核线程切换的保存和恢复操作。`swtch` 并不直接了解线程；它只是保存和恢复寄存器集，称为上下文。当一个进程需要放弃 CPU 时，该进程的内核线程会调用 `swtch` 来保存自己的上下文并返回到调度器上下文。每个上下文都包含在一个 `struct context` 中。(kernel/proc.h:2)，它本身包含在进程的 struct proc 或 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到 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` 进入调度程序。我们在第 4 章中看到，中断结束时的一种可能是 `usertrap` 调用 `yield`。`yield` 接着调用 `sched`，`sched` 调用 `swtch` 将当前上下文保存在 `p->context` 中，并切换到之前保存的调度程序上下文。

保存在 cpu->scheduler (kernel/proc.c:509)。

Swtch (kernel/swtch.S:3) 仅保存被调用者保存的寄存器；调用者保存的寄存器（如果需要）由调用的 C 代码保存在栈上。Swtch 知道每个寄存器字段在 struct context 中的偏移量。它不保存程序计数器。相反，swtch 保存了 ra 寄存器，该寄存器保存了调用 swtch 的返回地址。现在 swtch 从

现在，`swtch` 从新的上下文中恢复寄存器，这些寄存器保存了之前 `swtch` 保存的寄存器值。当 `swtch` 返回时，它返回到由恢复的 `ra` 寄存器指向的指令，即新线程之前调用 `swtch` 的指令。此外，它在新线程的栈上返回。
在我们的示例中，`sched` 调用了 `swtch` 来切换到 `cpu->scheduler`，即每个 CPU 的调度器上下文。该上下文是由调度器调用 `swtch` 时保存的（kernel/proc.c:475）。当我们正在跟踪的 `swtch` 返回时，它不会返回到 `sched`，而是返回到 `scheduler`，并且它的栈指针指向当前 CPU 的调度器栈。

7.3 代码：调度

上一节探讨了`swtch`的低级细节；现在我们将`swtch`视为既定事实，并研究从一个进程的内核线程通过调度器切换到另一个进程的过程。调度器以每个 CPU 一个特殊线程的形式存在，每个线程运行调度器函数。该函数负责选择接下来要运行的进程。一个想要放弃 CPU 的进程必须获取自己的进程锁`p->lock`，释放它持有的任何其他锁，更新自己的状态（`p->state`），然后调用 `sched`。`Yield`（kernel/proc.c:515）遵循这一惯例，`sleep`和`exit`也是如此，我们稍后将详细研究这些函数。`Sched`会再次检查这些条件。
(kernel/proc.c:499-504) 然后是这些条件的一个含义：由于持有锁，中断应该被禁用。最后，sched 调用 swtch 将当前上下文保存在 p->context 中，并切换到 cpu->scheduler 中的调度器上下文。Swtch 在调度器的栈上返回，就像调度器的 swtch 已经返回一样。调度器继续 for 循环，找到一个要运行的进程，切换到它，然后循环重复。
我们刚刚看到 xv6 在调用 swtch 时持有 p->lock：swtch 的调用者必须已经持有锁，并且锁的控制权会传递给切换到的代码。这种锁的约定在锁的使用中并不常见；通常获取锁的线程也负责释放锁，这使得正确性推理更加容易。对于上下文切换来说，打破这种约定是必要的，因为 p->lock 保护了进程状态和上下文字段的不变量，而这些不变量在 swtch 执行期间并不成立。如果在 swtch 期间没有持有 p->lock，可能会出现的一个问题是：在 yield 将进程状态设置为 RUNNABLE 之后，但在 swtch 导致它停止使用自己的内核栈之前，另一个 CPU 可能会决定运行该进程。结果将是两个 CPU 在同一个栈上运行，这显然是不正确的。
内核线程总是在`sched`中放弃其 CPU，并且总是切换到调度器中的同一位置，而调度器（几乎）总是切换到之前调用`sched`的某个内核线程。因此，如果打印出 xv6 切换线程的行号，人们会观察到...
遵循以下简单模式：(kernel/proc.c:475), (kernel/proc.c:509), (kernel/proc.c:475), (kernel/proc.c:509)，等等。这种在两个线程之间进行风格化切换的过程有时被称为协程；在这个例子中，sched 和 scheduler 是彼此的协程。
有一种情况，调度器调用`swtch`不会在`sched`中结束。当一个新进程首次被调度时，它从 `forkret`（kernel/proc.c:527）开始。`forkret`的存在是为了释放`p->lock`；否则，新进程可以从`usertrapret`开始。

调度程序（kernel/proc.c:457）运行一个简单的循环：找到一个要运行的进程，运行它直到它让出，

重复。调度程序循环遍历进程表，寻找一个可运行的进程，即一个具有
p->state == RUNNABLE。一旦找到一个进程，它就会设置每个 CPU 的当前进程变量 c->proc，将进程标记为 RUNNING，然后调用 swtch 开始运行它（kernel/proc.c:470-
475).
思考调度代码结构的一种方式是，它强制执行关于每个进程的一组不变量，并在这些不变量不成立时持有`p->lock`。一个不变量是，如果一个进程是`RUNNING`状态，定时器中断的`yield`必须能够安全地从该进程切换出去；这意味着 CPU 寄存器必须持有该进程的寄存器值（即`swtch`尚未将它们移动到上下文中），并且`c->proc`必须引用该进程。另一个不变量是，如果一个进程是`RUNNABLE`状态，空闲 CPU 的调度程序必须能够安全地运行它；这意味着`p->context`必须持有该进程的寄存器（即它们实际上不在真实的寄存器中），没有 CPU 在该进程的内核栈上执行，并且没有 CPU 的`c->proc`引用该进程。注意，这些属性在持有`p->lock`时通常不成立。
维护上述不变性是为什么 xv6 经常在一个线程中获取 p->lock 并在另一个线程中释放它的原因，例如在 yield 中获取并在 scheduler 中释放。一旦 yield 开始修改一个运行进程的状态以使其变为 RUNNABLE，锁必须保持持有直到不变性恢复：最早的正确释放点是在 scheduler（在其自己的堆栈上运行）清除 c->proc 之后。类似地，一旦 scheduler 开始将一个 RUNNABLE 进程转换为 RUNNING，锁就不能释放，直到内核线程完全运行（例如在 swtch 之后，在 yield 中）。
p->lock 还保护其他内容：exit 和 wait 之间的交互、避免丢失唤醒的机制（参见第 7.5 节），以及避免进程退出与其他进程读取或写入其状态之间的竞争（例如，exit 系统调用查看 p->pid 并设置 p->killed (kernel/proc.c:611)）。为了清晰性和可能的性能提升，值得考虑是否可以将 p->lock 的不同功能拆分。

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，可以使用一个全局变量指向当前的 proc。这在多核机器上不起作用，因为每个核心执行不同的进程。解决这个问题的方法是利用每个核心都有自己的一组寄存器的事实；我们可以使用其中一个寄存器来帮助找到每个核心的信息。
Xv6 为每个 CPU 维护一个 struct cpu（kernel/proc.h:22），它记录了当前在该 CPU 上运行的进程（如果有的话）、CPU 调度器线程的保存寄存器以及管理中断禁用所需的嵌套自旋锁计数。函数 mycpu（kernel/proc.c:60）返回指向当前 CPU 的 struct cpu 的指针。RISC-V 为其 CPU 编号，给每个 CPU 一个 hartid。Xv6 确保在内核中运行时，每个 CPU 的 hartid 存储在该 CPU 的 tp 寄存器中。

这使得 mycpu 可以使用 tp 来索引一个 cpu 结构数组以找到正确的一个。
确保 CPU 的 tp 寄存器始终保存 CPU 的 hartid 有点复杂。mstart 在 CPU 启动序列的早期设置 tp 寄存器，此时仍处于机器模式（kernel/start.c:46）。usertrapret 将 tp 保存在 trampoline 页面中，因为用户进程可能会修改 tp。最后，

uservec 在从用户空间进入内核时恢复保存的 tp（kernel/trampoline.S:70）。编译器保证永远不会使用 tp 寄存器。如果 RISC-V 允许 xv6 直接读取当前的 hartid，那将更加方便，但这仅在机器模式下允许，而不在监督模式下允许。

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 CPU，然后移动到另一个 CPU，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，并且只有在使用完返回的 struct cpu 后才重新启用中断。

函数 myproc (kernel/proc.c:68) 返回当前 CPU 上运行的进程的 struct proc 指针。myproc 会禁用中断，调用 mycpu，从 struct cpu 中获取当前进程指针 (c->proc)，然后重新启用中断。即使启用了中断，myproc 的返回值也是安全的：如果定时器中断将调用进程移动到不同的 CPU，其 struct proc 指针将保持不变。

7.5 Sleep 和 wakeup

调度和锁帮助隐藏一个进程的存在于另一个进程，但到目前为止，我们还没有帮助进程有意交互的抽象。许多机制被发明来解决这个问题。Xv6 使用了一种称为 sleep 和 wakeup 的机制，它允许一个进程在等待事件时睡眠，而另一个进程在事件发生后唤醒它。Sleep 和 wakeup 通常被称为序列协调或条件同步机制。

为了说明这一点，让我们考虑一种称为信号量[4]的同步机制，它协调生产者和消费者。信号量维护一个计数并提供两个操作。“V”操作（针对生产者）增加计数。“P”操作（针对消费者）等待直到计数非零，然后递减计数并返回。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，且编译器没有进行过于激进的优化，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         获取(&s->lock);
109         s->count += 1;
110         release(&s->lock);
111 }
112
113     void
114     P(struct semaphore *s)
115     {
```

```
116         当 s->count == 0 时
117           ;
118         获取(&s->lock);
119         s->count -= 1;
120         release(&s->lock);
121     }
```

上述实现的开销很大。如果生产者很少行动，消费者将花费大部分时间在 while 循环中自旋，希望 count 不为零。消费者的 CPU 可以通过重复轮询 s->count 来找到比忙等待更有成效的工作。避免忙等待需要一种方法让消费者让出 CPU，并在 V 增加 count 后恢复执行。

这是一个朝着正确方向迈出的一步，尽管我们会看到这还不够。让我们想象一对调用，`sleep` 和 `wakeup`，它们的工作方式如下。`Sleep(chan)` 在任意值 `chan` 上睡眠，这个值被称为等待通道。`Sleep` 使调用进程进入睡眠状态，释放 CPU 以供其他工作使用。`Wakeup(chan)` 唤醒所有在 `chan` 上睡眠的进程（如果有的话），使它们的 `sleep` 调用返回。如果没有进程在 `chan` 上等待，`wakeup` 则不做任何操作。我们可以更改信号量的实现以使用 `sleep` 和 `wakeup`（更改部分以黄色高亮显示）：

```
200     void
201     V(struct semaphore *s)
202     {
203         获取(&s->lock);
204         s->count += 1;
205         wakeup(s);
206         release(&s->lock);
207 }
208
209     void
210     P(struct semaphore *s)
211     {
212       当 s->count == 0 时
213           sleep(s);
214       获取(&s->lock);
215       s->count -= 1;
216       release(&s->lock);
217     }
```

现在它放弃了 CPU 而不是自旋，这很好。然而，事实证明，使用这个接口设计 sleep 和 wakeup 而不遭受所谓的“丢失唤醒问题”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，wakeup 发现没有进程在睡眠，因此什么也不做。现在 P 继续在第 213 行执行：它调用 sleep 并进入睡眠状态。这导致了一个问题：P 正在等待一个已经发生的 V 调用。除非我们运气好，生产者再次调用 V，否则即使计数不为零，消费者也会永远等待。

这个问题的根源在于，P 只在 s->count == 0 时睡眠的不变量被 V 在错误时刻运行所破坏。保护这个不变量的错误方法是将 P 中的锁获取（如下面黄色高亮部分）移动，使其对计数的检查和调用 sleep 是原子的：

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

有人可能希望这个版本的 P 能够避免丢失唤醒，因为锁阻止了 V 在 313 和 314 行之间执行。它确实做到了这一点，但也会导致死锁：P 在睡眠时持有锁，因此 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传递给 sleep，以便在调用进程被标记为睡眠并等待睡眠通道后释放锁。该锁将强制并发的 V 等待，直到 P 完成将自己置于睡眠状态，这样唤醒操作将找到睡眠的消费者并唤醒它。一旦消费者再次醒来，sleep 在返回之前会重新获取锁。我们新的正确的 sleep/wakeup 方案如下所示（黄色高亮显示更改）：

```
400     void
401     V(struct semaphore *s)
402     {
403        获取(&s->lock);
404        s->count += 1;
405        wakeup(s);
406        release(&s->lock);
407 }
408
409     void
410     P(struct semaphore *s)
411     {
412        获取(&s->lock);
```

```
413 当(s->count == 0)时
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 c->count 和调用 sleep 之间尝试唤醒它。然而，需要注意的是，我们需要 sleep 原子性地释放 s->lock 并使消费进程进入睡眠状态。

7.6 代码：Sleep 和 wakeup

让我们看一下 `sleep`（kernel/proc.c:548）和 `wakeup`（kernel/proc.c:582）的实现。

基本思想是让 sleep 将当前进程标记为 SLEEPING，然后调用 sched 释放 CPU；wakeup 查找在给定等待通道上睡眠的进程并将其标记为

RUNNABLE。sleep 和 wakeup 的调用者可以使用任何相互方便的数字作为通道。Xv6 经常使用涉及等待的内核数据结构的地址。

获取睡眠p->lock (kernel/proc.c:559)。现在，即将进入睡眠的进程持有 p->lock 和 lk。在调用者（在示例中为 P）中持有 lk 是必要的：它确保没有其他进程（在示例中为运行 V 的进程）可以开始调用 wakeup(chan)。现在 sleep 持有 p->lock，释放 lk 是安全的：其他进程可能会开始调用 wakeup(chan)，但 wakeup 会等待获取 p->lock，因此会等待 sleep 完成将进程放入睡眠状态，从而避免 wakeup 错过 sleep。

有一个小问题：如果 lk 与 p->lock 是同一个锁，那么 sleep 在尝试获取 p->lock 时会与自身发生死锁。但如果调用 sleep 的进程已经持有 p->lock，它不需要做更多的事情来避免错过并发的 wakeup。这种情况

当 wait (kernel/proc.c:582) 调用 sleep 并传入 p->lock 时，问题就出现了。

既然 `sleep` 持有 `p->lock` 且没有其他锁，它可以通过记录睡眠通道、将进程状态更改为 `SLEEPING` 并调用 `sched`（kernel/proc.c:564-567）来使进程进入睡眠状态。稍后将清楚为什么在进程被标记为 `SLEEPING` 之前（由调度程序）不释放 `p->lock` 是至关重要的。

在某些时候，一个进程将获取条件锁，设置睡眠者等待的条件，并调用 `wakeup(chan)`。重要的是，`wakeup` 是在持有条件锁的情况下调用的。`wakeup` 会遍历进程表（`kernel/proc.c:582`）。它会获取它检查的每个进程的 `p->lock`，这既是因为它可能会操作该进程的状态，也是因为 `p->lock` 确保 `sleep` 和 `wakeup` 不会错过彼此。当 `wakeup` 找到一个状态为 `SLEEPING` 且 `chan` 匹配的进程时，它会将该进程的状态更改为 `RUNNABLE`。下次调度器运行时，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睡眠进程不会错过唤醒？睡眠进程要么持有条件锁，要么持有自己的 p->lock，或者两者都持有，从 a

---

[1] 严格来说，只要 wakeup 在 acquire 之后执行就足够了（也就是说，可以在 release 之后调用 wakeup）。

在检查条件之前，它会被标记为 SLEEPING。调用 wakeup 的进程在 wakeup 的循环中持有这两个锁。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 严格在被标记为 SLEEPING 之后检查睡眠线程。然后 wakeup 会看到睡眠的进程并唤醒它（除非有其他东西先唤醒它）。

有时会有多个进程在同一个通道上休眠；例如，多个进程从管道中读取数据。一次`wakeup`调用会唤醒所有这些进程。其中一个进程会首先运行并获取`sleep`调用时使用的锁，然后（在管道的情况下）读取管道中等待的数据。其他进程会发现，尽管被唤醒，但没有数据可读。从它们的角度来看，这次唤醒是“虚假的”，它们必须再次休眠。因此，`sleep`总是在检查条件的循环中被调用。

如果两个使用 sleep/wakeup 的操作意外选择了相同的通道，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 的很大魅力在于它既轻量级（无需创建特殊的数据结构作为睡眠通道），又提供了一层间接性（调用者无需知道它们正在与哪个特定进程交互）。

7.7 代码：Pipes

一个更复杂的例子是 xv6 中管道的实现，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：写入管道一端的数据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，其中包含一个锁和一个数据缓冲区。
字段 `nread` 和 `nwrite` 分别统计从缓冲区读取和写入的总字节数。缓冲区是循环的：在 `buf[PIPESIZE-1]` 之后写入的下一个字节是 `buf[0]`。计数不会循环。这种约定让实现能够区分满缓冲区（`nwrite == nread+PIPESIZE`）和空缓冲区（`nwrite == nread`），但这也意味着索引缓冲区时必须使用 `buf[nread % PIPESIZE]` 而不是简单的 `buf[nread]`（对于 `nwrite` 也是如此）。
nwrite
假设在两个不同的 CPU 上同时调用 piperead 和 pipewrite。Pipewrite（kernel/pipe.c:77）首先获取管道的锁，该锁保护计数、数据及其相关的不变量。Piperead（kernel/pipe.c:103）随后也尝试获取锁，但无法成功。它在 acquire（kernel/spinlock.c:22）中自旋等待锁。在 piperead 等待的同时，pipewrite 循环遍历要写入的字节（addr[0..n-1]），依次将它们添加到管道中（kernel/pipe.c:95）。在这个循环中，可能会发生缓冲区填满的情况（kernel/pipe.c:85）。在这种情况下，pipewrite 调用 wakeup 来提醒任何睡眠的读者缓冲区中有数据等待，然后在&pi->nwrite 上睡眠，等待读者从缓冲区中取出一些字节。Sleep 在将 pipewrite 的进程置于睡眠状态时释放 pi->lock。

现在 pi->lock 可用，piperead 设法获取它并进入其临界区
发现 pi->nread != pi->nwrite (kernel/pipe.c:110) (pipewrite 进入睡眠状态

原因在于 pi->nwrite == pi->nread+PIPESIZE (kernel/pipe.c:85)，因此它继续执行到 for 循环循环，将数据从管道中复制出来（kernel/pipe.c:117），并将 nread 增加复制的字节数。现在有这么多字节可用于写入，因此 piperead 在返回之前调用 wakeup（kernel/pipe.c:124）来唤醒任何睡眠的写者。Wakeup 找到一个在&pi->nwrite 上睡眠的进程，即运行 pipewrite 但在缓冲区填满时停止的进程。它将该进程标记为 RUNNABLE。

管道代码为读取者和写入者使用了独立的睡眠通道（pi->nread 和 pi->nwrite）；这在极不可能的情况下，即有许多读取者和写入者在等待同一个管道时，可能会使系统更高效。管道代码在检查睡眠条件的循环内部睡眠；如果有多个读取者或写入者，除了第一个被唤醒的进程外，其他进程都会看到条件仍然为假并再次睡眠。

7.8 代码：Wait、exit 和 kill

睡眠和唤醒可以用于多种等待情况。一个有趣的例子，在第 1 章中介绍过，是子进程的退出与其父进程的等待之间的交互。在子进程死亡时，父进程可能已经在等待中睡眠，或者正在做其他事情；在后一种情况下，后续的等待调用必须观察到子进程的死亡，可能是在调用退出很久之后。xv6 记录子进程的死亡直到等待观察到它的方式是，让退出将调用者置于 ZOMBIE 状态，直到父进程的等待注意到它，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，父进程将子进程交给 init 进程，init 进程会不断调用等待；因此每个子进程都有一个父进程来清理它。主要的实现挑战是父进程和子进程之间的等待和退出，以及退出和退出之间的竞争和死锁的可能性。

Wait 使用调用进程的 p->lock 作为条件锁以避免丢失唤醒，并在开始时获取该锁（kernel/proc.c:398）。然后它扫描进程表。如果它找到一个处于 ZOMBIE 状态的子进程，它会释放该子进程的资源及其 proc 结构，将子进程的退出状态复制到提供给 wait 的地址（如果不为 0），并返回子进程的进程 ID。如果 wait 找到子进程但没有一个退出，它会调用 sleep 等待其中一个退出（kernel/proc.c:445），然后再次扫描。在这里，sleep 中释放的条件锁是等待进程的 p->lock，即上述的特殊情况。注意，wait 经常持有两个锁；它在尝试获取任何子进程的锁之前先获取自己的锁；因此，xv6 的所有部分都必须遵守相同的锁顺序（父进程，然后是子进程）以避免死锁。

Wait 查看每个进程的 np->parent 以找到其子进程。它在没有持有 np->lock 的情况下使用 np->parent，这违反了共享变量必须由锁保护的常规规则。有可能 np 是当前进程的祖先，在这种情况下获取 np->lock 可能会导致死锁，因为这会违反上述顺序。在这种情况下，不加锁地检查 np->parent 似乎是安全的；一个进程的 parent 字段只能由其父进程更改，因此如果 np->parent==p 为真，除非当前进程更改它，否则该值不会改变。

退出（kernel/proc.c:333）记录退出状态，释放一些资源，将任何子进程交给

将任何子进程交给 init 进程，唤醒父进程以防它在等待中，将调用者标记为僵尸，并永久地让出 CPU。最后的步骤有点复杂。退出的进程在将其状态设置为 ZOMBIE 并唤醒父进程时，必须持有父进程的锁，因为父进程的锁是防止在 wait 中丢失唤醒的条件锁。子进程还必须持有自己的 p->lock，否则父进程可能会看到它处于 ZOMBIE 状态并在它仍在运行时释放它。锁的获取顺序对于避免死锁很重要：由于 wait 在获取子进程的锁之前先获取父进程的锁，exit 必须使用相同的顺序。

Exit 调用了一个专门的唤醒函数 wakeup1，它只唤醒父进程，并且只有在父进程在 wait 中睡眠时才唤醒（kernel/proc.c:598）。看起来子进程在将其状态设置为 ZOMBIE 之前唤醒父进程可能是不正确的，但这是安全的：尽管 wakeup1 可能会导致父进程运行，但 wait 中的循环在子进程的 p->lock 被调度程序释放之前无法检查子进程，因此 wait 在 exit 将其状态设置为 ZOMBIE 之后才能查看退出的进程（ker-

nel/proc.c:386)。

虽然 exit 允许进程终止自身，但 kill（kernel/proc.c:611）允许一个进程请求另一个进程终止。让 kill 直接销毁目标进程会过于复杂，因为目标进程可能正在另一个 CPU 上执行，可能正处于对内核数据结构进行敏感更新序列的中间。因此，kill 只做很少的事情：它只是设置目标的 p->killed 并且，如果它正在睡眠，则唤醒它。最终，受害者将进入或离开内核，此时如果 p->killed 被设置，usertrap 中的代码将调用 exit。如果受害者正在用户空间运行，它将很快通过系统调用或因为计时器（或其他设备）中断而进入内核。

如果受害者进程处于睡眠状态，`kill`调用`wakeup`将导致受害者从睡眠中返回。这可能是危险的，因为等待的条件可能不成立。然而，xv6 中的`sleep`调用总是被包裹在一个`while`循环中，该循环在`sleep`返回后重新测试条件。一些`sleep`调用还会在循环中测试`p->killed`，如果设置了该标志，则放弃当前活动。这只有在放弃是正确的情况下才会进行。例如，如果设置了`killed`标志，管道读写代码会返回；最终代码将返回到`trap`，再次检查标志并退出。

一些 xv6 的睡眠循环没有检查`p->killed`，因为代码正处于一个应该是原子的多步骤系统调用中。virtio 驱动程序（`kernel/virtio_disk.c:242`）就是一个例子：它没有检查`p->killed`，因为磁盘操作可能是一组写入操作之一，这些操作都是文件系统保持正确状态所必需的。一个在等待磁盘 I/O 时被终止的进程，只有在完成当前系统调用并且`usertrap`看到终止标志后才会退出。

7.9 现实世界

xv6 调度器实现了一种简单的调度策略，即依次运行每个进程。这种策略称为轮转调度。实际的操作系统实现了更复杂的策略，例如允许进程具有优先级。其思想是，调度器会优先选择可运行的高优先级进程，而不是可运行的低优先级进程。这些策略可能会迅速变得复杂，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，复杂的策略可能会导致不可预测的行为。

复杂的策略可能导致意外的交互，例如优先级反转和车队效应。当低优先级和高优先级进程共享一个锁时，可能会发生优先级反转，当低优先级进程获取锁时，可能会阻止高优先级进程的进展。当许多高优先级进程等待获取共享锁的低优先级进程时，可能会形成一个长时间等待的车队；一旦车队形成，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。

睡眠和唤醒是一种简单而有效的同步方法，但还有许多其他方法。所有方法中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。原始的 Unix 内核的睡眠只是简单地禁用中断，这在 Unix 运行在单 CPU 系统上时已经足够。因为 xv6 运行在多处理器上，所以它给睡眠添加了一个显式的锁。FreeBSD 的 msleep 采用了相同的方法。Plan 9 的睡眠使用了一个回调函数，该函数在持有调度锁的情况下运行，就在进入睡眠之前；该函数作为对睡眠条件的最后一刻检查，以避免丢失唤醒。Linux 内核的睡眠使用了一个显式的进程队列，称为等待队列，而不是等待通道；队列有自己的内部锁。

在 wakeup 中扫描整个进程列表以查找具有匹配 chan 的进程是低效的。更好的解决方案是将 sleep 和 wakeup 中的 chan 替换为一个数据结构，该数据结构保存了在该结构上睡眠的进程列表，例如 Linux 的等待队列。Plan 9 的 sleep 和 wakeup 将该结构称为 rendezvous point 或 Rendez。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征：睡眠条件由某种锁保护，在睡眠期间原子性地释放。

wakeup 的实现会唤醒所有等待在特定通道上的进程，可能会有许多进程在等待该特定通道。操作系统将调度所有这些进程，它们将竞相检查睡眠条件。以这种方式行为的进程有时被称为惊群（thundering herd），最好避免这种情况。大多数条件变量有两个用于唤醒的原语：signal，它唤醒一个进程，以及 broadcast，它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等。将显式计数作为抽象的一部分可以避免“丢失唤醒”问题：存在一个显式的唤醒次数计数。计数还避免了虚假唤醒和惊群问题。

终止进程并清理它们给 xv6 带来了很多复杂性。在大多数操作系统中，这甚至更加复杂，因为例如，受害进程可能在内核深处休眠，展开其堆栈需要非常谨慎的编程。许多操作系统使用显式的异常处理机制（如 longjmp）来展开堆栈。此外，还有其他事件可能导致休眠的进程被唤醒，即使它等待的事件尚未发生。例如，当 Unix 进程休眠时，另一个进程可能会向它发送信号。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，并将错误代码设置为 EINTR。应用程序可以检查这些值并决定如何处理。Xv6 不支持信号，因此不会出现这种复杂性。

Xv6 对 kill 的支持并不完全令人满意：存在一些睡眠循环，可能应该检查 p->killed。一个相关的问题是，即使对于检查 p->killed 的睡眠循环，也存在 sleep 和 kill 之间的竞争；后者可能在受害者的循环检查 p->killed 之后但在调用 sleep 之前设置 p->killed 并尝试唤醒受害者。如果发生此问题，受害者将不会注意到 p->killed，直到它等待的条件发生。这可能是在相当长的时间之后（例如，当 virtio 驱动程序返回受害者正在等待的磁盘块时）或永远不会（例如，如果受害者正在等待来自控制台的输入，但用户没有输入任何内容）。

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

7.10 练习

1. Sleep 必须检查 lk != &p->lock 以避免死锁（kernel/proc.c:558-561）。假设通过替换来消除特殊情况

```
如果(lk != &p->lock){
   获取(&p->lock);
   release(lk);
}
```

with

```
release(lk);
获取(&p->lock);
```

这样做会破坏 sleep。如何破坏?

2. 大多数进程清理工作可以由 exit 或 wait 完成。事实证明，exit 必须是关闭打开文件的那个。为什么？答案涉及管道。

3. 在 xv6 中实现信号量而不使用 sleep 和 wakeup（但可以使用自旋锁）。用信号量替换 xv6 中的 sleep 和 wakeup 的使用。判断结果。

4. 修复上述提到的`kill`和`sleep`之间的竞争条件，使得在受害者的`sleep`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系统调用。

5. 设计一个方案，使得每个睡眠循环都检查 `p->killed`，这样，例如，一个在 virtio 驱动中的进程如果被另一个进程杀死，可以快速从 while 循环中返回。

6. 修改 xv6，使其在从一个进程的内核线程切换到另一个进程时仅使用一次上下文切换，而不是通过调度器线程进行切换。让出 CPU 的线程需要自己选择下一个线程并调用 swtch。面临的挑战包括防止多个核心意外执行同一个线程；正确处理锁；以及避免死锁。

7. 修改 xv6 的调度器，使其在没有可运行进程时使用 RISC-V 的 WFI（等待中断）指令。尽量确保在任何时候有可运行进程等待运行时，没有核心在 WFI 中暂停。

8. 锁 p->lock 保护了许多不变量，当查看由 p->lock 保护的特定 xv6 代码片段时，可能难以确定正在强制执行哪个不变量。设计一个更清晰的方案，将 p->lock 拆分为多个锁。

第 8 章

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，以便在重启后数据仍然可用。

xv6 文件系统提供了类似 Unix 的文件、目录和路径名（见第 1 章），并将其数据存储在 virtio 磁盘上以实现持久性（见第 4 章）。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目录和文件的树结构，记录保存每个文件内容的块的身份，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），文件系统在重启后仍必须正常工作。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，一个块既被文件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性。
- 访问磁盘比访问内存慢几个数量级，因此文件系统必须在内存中维护一个常用块的缓存。

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 硬盘驱动器上读写块。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特定块中的数据。日志层允许更高层将多个块的更新包装在一个事务中，并确保在面临

图 8.1：xv6 文件系统的层次结构。

并确保在崩溃时块能够原子性地更新（即全部更新或全部不更新）。inode 层提供单个文件，每个文件表示为一个具有唯一 i-number 的 inode 和一些包含文件数据的块。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。

文件系统必须有一个计划，用于在磁盘上存储 inode 和内容块的位置。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 称为超级块；它包含有关文件系统的元数据（文件系统的大小以块为单位、数据块的数量、inode 的数量以及日志中的块数）。从块 2 开始的是日志。日志之后是 inode，每个块中有多个 inode。接下来是位图块，用于跟踪哪些数据块正在使用。其余的块是数据块；每个数据块要么在位图块中标记为空闲，要么包含文件或目录的内容。超级块由一个名为 mkfs 的单独程序填充，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。注意那些在较低层次精心选择的抽象如何简化了较高层次设计的情况。

8.2　Buffer cache layer

缓冲区缓存有两个任务：(1) 同步对磁盘块的访问，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，以便不需要从慢速磁盘重新读取。代码位于 bio.c。

缓冲区缓存导出的主要接口包括 bread 和 bwrite；前者获取一个包含块副本的 buf，可以在内存中读取或修改，后者将修改后的缓冲区写入磁盘上的适当块。内核线程在使用完缓冲区后必须通过调用 brelse 来释放它。缓冲区缓存使用每个缓冲区的睡眠锁来确保

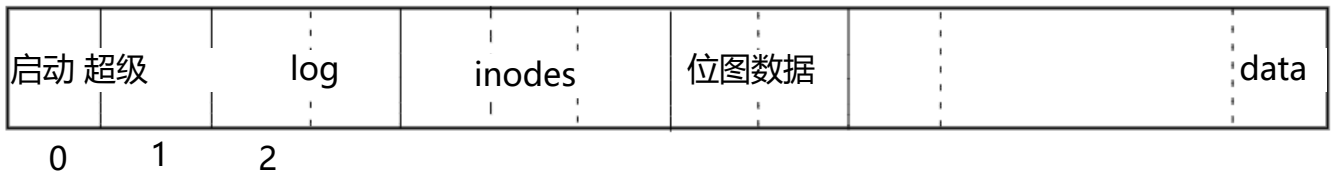

图 8.2：xv6 文件系统的结构。

缓冲区缓存使用每个缓冲区的睡眠锁来确保每次只有一个线程使用每个缓冲区（以及每个磁盘块）；bread 返回一个锁定的缓冲区，而 brelse 释放锁。

让我们回到缓冲区缓存。缓冲区缓存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一个不在缓存中的块，缓冲区缓存必须回收当前持有其他块的缓冲区。缓冲区缓存会为新的块回收最近最少使用的缓冲区。假设是最近最少使用的缓冲区最不可能很快再次被使用。

8.3 代码：Buffer cache

缓冲区缓存是一个双向链表。函数 binit 由 main（kernel/-main.c:27），使用静态数组 buf 中的 NBUF 缓冲区初始化列表（kernel/bio.c:43-52）。所有其他对缓冲区缓存的访问都通过 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，将数据从磁盘写入数据）。

Bread (kernel/bio.c:93) 调用 bget 来获取给定扇区 (kernel/bio.c:97) 的缓冲区。如果缓冲区需要从磁盘读取，bread 在返回缓冲区之前调用 virtio_disk_rw 来完成读取操作。

Bget (kernel/bio.c:59) 扫描缓冲区列表，寻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 会获取该缓冲区的睡眠锁。然后，bget 返回已锁定的缓冲区。

如果没有为给定扇区缓存的缓冲区，bget 必须创建一个，可能会重用持有不同扇区的缓冲区。它再次扫描缓冲区列表，寻找一个未使用的缓冲区（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，并获取其睡眠锁。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之前的内容。

重要的是，每个磁盘扇区最多只能有一个缓存缓冲区，以确保读者能看到写入操作，并且因为文件系统使用缓冲区上的锁进行同步。Bget 通过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）期间持续持有 bache.lock 来确保这一不变性。这使得检查块是否存在以及（如果不存在）指定一个缓冲区来保存块的操作成为原子操作。

bget 在 bcache.lock 临界区之外获取缓冲区的睡眠锁是安全的，因为非零的 b->refcnt 防止了缓冲区被重新用于不同的目的。

由于非零的 b->refcnt 阻止了缓冲区被重新用于不同的磁盘块。sleep-lock 保护了对块缓冲内容的读写，而 bcache.lock 保护有关哪些块被缓存的信息。
如果所有缓冲区都忙，那么有太多进程同时在执行文件系统调用；bget 会引发恐慌。更优雅的响应可能是休眠直到某个缓冲区变为空闲，尽管这样可能会导致死锁。
一旦 bread 读取了磁盘（如果需要）并将缓冲区返回给调用者，调用者就拥有缓冲区的独占使用权，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 将更改后的数据写入磁盘。Bwrite

(kernel/bio.c:107)调用 virtio_disk_rw 与磁盘硬件通信。
当调用者使用完一个缓冲区时，必须调用`brelse`来释放它。（`brelse`这个名字是`b-release`的缩写，虽然晦涩但值得学习：它起源于 Unix，并在 BSD、Linux 和 Solaris 中也使用。）`Brelse`（kernel/bio.c:117）释放睡眠锁并将缓冲区移动到链表的前面（kernel/bio.c:128-133）。移动缓冲区会导致链表按缓冲区最近使用（即释放）的顺序排列：链表中的第一个缓冲区是最近使用的，最后一个是最久未使用的。`bget`中的两个循环利用了这一点：在最坏情况下，扫描现有缓冲区必须处理整个链表，但如果引用局部性良好，首先检查最近使用的缓冲区（从`bcache.head`开始并跟随`next`指针）将减少扫描时间。选择要重用的缓冲区的扫描通过向后扫描（跟随`prev`指针）选择最久未使用的缓冲区。

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。这个问题的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的崩溃可能会使磁盘上的文件系统处于不一致的状态。例如，假设在文件截断（将文件长度设置为零并释放其内容块）期间发生崩溃。根据磁盘写入的顺序，崩溃可能会留下一个引用被标记为空闲的内容块的 inode，或者可能会留下一个已分配但未被引用的内容块。
后者相对无害，但一个引用已释放块的 inode 在重启后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，现在我们就有了两个不同的文件无意中指向了同一个块。如果 xv6 支持多用户，这种情况可能会成为一个安全问题，因为旧文件的所有者将能够读取和写入由不同用户拥有的新文件中的块。

Xv6 通过一种简单的日志形式解决了文件系统操作期间的崩溃问题。
xv6 系统调用不会直接写入磁盘文件系统的数据结构。相反，它将所有希望进行的磁盘写入操作的描述记录在磁盘的日志中。一旦系统调用记录了所有的写入操作，它会在磁盘上写入一个特殊的提交记录，表示日志中包含一个完整的操作。此时，系统调用将这些写入操作复制到磁盘文件系统的数据结构中。在这些写入操作完成后，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按如下方式从崩溃中恢复。如果日志标记为包含完整操作，那么

然后，恢复代码将写操作复制到磁盘文件系统中它们应属的位置。如果日志未标记为包含完整操作，恢复代码将忽略该日志。恢复代码通过擦除日志来完成操作。

为什么 xv6 的日志解决了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操作，如果操作已经开始将它们写入磁盘数据结构，可能会重复这些写操作。无论哪种情况，日志都使得操作相对于崩溃是原子的：恢复后，要么操作的所有写操作都出现在磁盘上，要么一个都不出现。

8.5　日志设计

日志位于超级块中指定的已知固定位置。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扇区号，以及日志块的数量。磁盘上头部块中的计数要么为零，表示日志中没有事务，要么为非零，表示日志包含一个完整的已提交事务，并指示了日志块的数量。Xv6 在事务提交时写入头部块，但在此之前不会写入，并在将日志块复制到文件系统后将计数设置为零。因此，事务中途崩溃将导致日志头部块中的计数为零；提交后崩溃将导致计数为非零。

每个系统调用的代码指示了必须相对于崩溃保持原子性的写序列的开始和结束。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操作累积到一个事务中。因此，单个提交可能涉及多个完整系统调用的写操作。为了避免将系统调用拆分到多个事务中，日志系统仅在没有任何文件系统系统调用进行时提交。

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供更多的并发写入，可能允许磁盘在单次旋转期间写入所有数据。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 在磁盘上分配了固定大小的空间来保存日志。一个事务中系统调用写入的块总数必须适应这个空间。这带来了两个后果。首先，不允许任何单个系统调用写入比日志空间更多的不同块。对于大多数系统调用来说这不是问题，但有两个系统调用可能会写入许多块：write 和 unlink。写入一个大文件可能会写入许多数据块和许多位图块以及一个 inode 块；删除一个大文件可能会写入许多位图块和一个 inode。Xv6 的 write 系统调用将大写入分解为多个适合日志的小写入，而 unlink 不会造成问题，因为实际上 xv6 文件系统只使用一个位图块。日志空间有限的另一个后果是，日志系统不能允许一个系统调用

日志空间有限的另一个后果是，日志系统不能允许系统调用开始，除非它确定系统调用的写入将适合日志中剩余的空间。

8.6 代码：logging

系统调用中日志的典型用法如下所示：

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

begin_op (kernel/log.c:126) 等待直到日志系统当前没有提交，并且有足够的未保留日志空间来容纳此调用的写入。log.outstanding 计数已保留日志空间的系统调用数量；总保留空间为 log.outstanding 乘以 MAXOPBLOCKS。增加 log.outstanding 既保留了空间，又防止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

可能会写入多达 MAXOPBLOCKS 个不同的块。
log_write (kernel/log.c:214) 作为 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预留一个槽位，并将缓冲区固定在块缓存中，以防止块缓存将其驱逐。该块必须保留在缓存中，直到提交为止：在此之前，缓存中的副本是修改的唯一记录；在提交之前，它不能写入磁盘上的位置；同一事务中的其他读取必须看到这些修改。log_write 会注意到在一个事务中多次写入一个块，并为该块分配日志中的相同槽位。这种优化通常称为吸收。例如，包含多个文件 inode 的磁盘块在一个事务中被多次写入是很常见的。通过将多次磁盘写入吸收为一次，文件系统可以节省日志空间，并且可以实现更好的性能，因为只需将磁盘块的一个副本写入磁盘。
end_op (kernel/log.c:146) 首先减少未完成的系统调用计数。如果计数现在为零，则通过调用 commit() 提交当前事务。此过程有四个阶段。write_log() (kernel/log.c:178) 将事务中修改的每个块从缓冲区缓存复制到磁盘上的日志槽中。write_head() (kernel/log.c:102) 将头块写入磁盘：这是提交点，写入后的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正确位置。最后，end_op 写入计数为零的日志头；这必须在下一个事务开始写入日志块之前发生，以便崩溃不会导致恢复使用一个事务的头和后续事务的日志块。

recover_from_log (kernel/log.c:116) 从 initlog (kernel/log.c:55) 调用，而 initlog 又被调用在启动期间，从 `fsinit(kernel/fs.c:42)` 运行，直到第一个用户进程运行之前（`kernel/proc.c:539`）。它读取日志头，如果头信息表明日志包含已提交的事务，则模拟 `end_op` 的操作。

日志的一个使用示例出现在 `filewrite`（`kernel/file.c:135`）中。事务看起来像这样：

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

这段代码被包裹在一个循环中，该循环将大型写入操作分解为每次仅涉及几个扇区的单独事务，以避免日志溢出。调用`writei`时，会作为此事务的一部分写入多个块：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，每个块对应一位。零位表示相应的块是空闲的；一位表示它正在使用中。程序 mkfs 设置了引导扇区、超级块、日志块、inode 块和位图块对应的位。

块分配器提供了两个函数：`balloc` 用于分配一个新的磁盘块，`bfree` 用于释放一个块。`balloc` 函数中的循环（位于 `kernel/fs.c:71`）从块 0 开始，一直到文件系统中的块数 `sb.size`，检查每个块。它寻找位图位为零的块，表示该块是空闲的。如果 `balloc` 找到这样的块，它会更新位图并返回该块。为了提高效率，循环被分成两部分。外部循环读取每个位图块的位。内部循环检查单个位图块中的所有 `BPB` 位。如果两个进程同时尝试分配一个块，可能会发生的竞争被缓冲缓存所防止，因为缓冲缓存一次只允许一个进程使用任何一个位图块。

Bfree (kernel/fs.c:90) 找到正确的位图块并清除相应的位。同样，bread 和 brelse 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许多代码一样，balloc 和 bfree 必须在事务内部调用。

8.8 Inode 层

术语 inode 可以有两种相关的含义。它可能指的是包含文件大小和数据块编号列表的磁盘上的数据结构。或者，“inode” 可能指的是内存中的 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 被紧密地打包在一个称为 inode 块的连续磁盘区域中。每个 inode 的大小相同，因此给定一个数字 n，很容易找到磁盘上的第 n 个 inode。实际上，这个数字 n，称为 inode 号或 i-number，是 inode 在实现中的标识方式。

磁盘上的 inode 由结构体 `dinode`（kernel/fs.h:32）定义。`type` 字段用于区分文件、目录和特殊文件（设备）。`type` 为零表示磁盘上的 inode 是空闲的。

类型为零表示磁盘上的 inode 是空闲的。nlink 字段统计引用此 inode 的目录条目数量，以便识别何时应释放磁盘上的 inode 及其数据块。size 字段记录文件中内容的字节数。addrs 数组记录存储文件内容的磁盘块的块号。

内核将活动 inode 集合保存在内存中；`struct inode`（kernel/file.h:17）是磁盘上`struct dinode`的内存副本。只有当有 C 指针引用该 inode 时，内核才会将其存储在内存中。`ref`字段统计引用内存中 inode 的 C 指针数量，如果引用计数降至零，内核将从内存中丢弃该 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、当前工作目录以及诸如`exec`之类的临时内核代码。

xv6 的 inode 代码中有四种锁或类似锁的机制。icache.lock 保护了一个不变性：一个 inode 在缓存中最多出现一次，并且缓存 inode 的 ref 字段记录了指向该缓存 inode 的内存指针数量。每个内存中的 inode 都有一个包含睡眠锁的 lock 字段，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。如果一个 inode 的 ref 大于零，系统会将该 inode 保留在缓存中，并且不会将该缓存条目用于其他 inode。最后，每个 inode 包含一个 nlink 字段（在磁盘上，如果缓存则在内存中复制），用于记录指向文件的目录条目数量；如果 inode 的链接计数大于零，xv6 不会释放该 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，指针所引用的内存也不会被重新用于不同的 inode。iget()提供了对 inode 的非独占访问，因此可以有多个指针指向同一个 inode。文件系统代码的许多部分依赖于 iget()的这种行为，既是为了保持对 inode 的长期引用（如打开的文件和当前目录），也是为了在操作多个 inode 的代码中（如路径名查找）避免竞争条件并防止死锁。

iget 返回的 struct inode 可能不包含任何有用的内容。为了确保它持有磁盘上的 inode 副本，代码必须调用 ilock。这会锁定 inode（这样其他进程就不能再锁定它），并从磁盘读取 inode（如果尚未读取）。iunlock 释放 inode 上的锁。将 inode 指针的获取与锁定分开有助于在某些情况下避免死锁，例如在目录查找期间。多个进程可以持有指向由 iget 返回的 inode 的 C 指针，但一次只能有一个进程锁定该 inode。

inode 缓存仅缓存内核代码或数据结构持有 C 指针的 inode。
它的主要工作实际上是同步多个进程的访问；缓存是次要的。如果一个 inode 被频繁使用，即使 inode 缓存没有保留它，缓冲区缓存也可能会将其保留在内存中。inode 缓存是写透的，这意味着修改缓存中 inode 的代码必须立即使用 iupdate 将其写入磁盘。

8.9 代码：Inodes

要分配一个新的 inode（例如，在创建文件时），xv6 会调用 ialloc（kernel/fs.c:196）。Ialloc 与 balloc 类似：它遍历磁盘上的 inode 结构，每次一个块，寻找标记为空闲的 inode。当它找到一个空闲的 inode 时，它会通过将新类型写入磁盘来声明它，然后通过尾调用 iget （kernel/fs.c:210）从 inode 缓存中返回一个条目。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 可用并尝试声明它。

Iget (kernel/fs.c:243) 在 inode 缓存中查找具有所需设备和 inode 编号的活动条目（ip->ref > 0）。如果找到，它会返回对该 inode 的新引用（kernel/fs.c:252-256）。在 iget 扫描时，它会记录第一个空槽的位置（kernel/fs.c:257-

258)，如果它需要分配一个缓存条目，它会使用它。

代码在读取或写入其元数据或内容之前，必须使用`ilock`锁定 inode。`ilock` （kernel/fs.c:289）为此目的使用了一个睡眠锁。一旦`ilock`获得了对 inode 的独占访问权，它会在需要时从磁盘（更可能是缓冲区缓存）读取 inode。函数`iunlock` (kernel/fs.c:317)释放睡眠锁，这可能会导致任何正在睡眠的进程被唤醒。

Iput (kernel/fs.c:333) 通过减少引用计数 (kernel/fs.c:356) 来释放一个指向 inode 的 C 指针。如果这是最后一个引用，inode 在 inode 缓存中的槽现在可以释放，并可以重新用于不同的 inode。

如果 `iput` 发现没有 C 指针引用一个 inode，并且该 inode 没有链接（即不在任何目录中），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:338）。

在`iput`中释放 inode 的情况下，锁定协议值得仔细研究。一个危险是，并发线程可能正在`ilock`中等待使用这个 inode（例如，读取文件或列出目录），并且不会准备好发现 inode 不再被分配。这种情况不会发生，因为如果系统调用没有指向它的链接并且`ip->ref`为 1，那么系统调用就无法获得指向缓存 inode 的指针。这个引用是由调用`iput`的线程拥有的。确实，`iput`在其`icache.lock`临界区外检查引用计数是否为 1，但在那时已知链接计数为零，因此没有线程会尝试获取新的引用。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`写入磁盘后才会发生，此时 inode 的类型为零。这种竞争是无害的；分配线程会礼貌地等待获取 inode 的睡眠锁，然后才会读取或写入 inode，而此时`iput`已经完成了操作。

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，因为系统调用可能是最后一个持有文件引用的调用。即使是像 read() 这样看似只读的调用，最终也可能调用 iput()。这反过来意味着，即使只读的系统调用，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，因为某些进程可能仍然持有内存中 inode 的引用：某个进程可能仍在读取和写入文件，因为它成功打开了该文件。但是，如果在最后一个进程关闭文件描述符之前发生崩溃

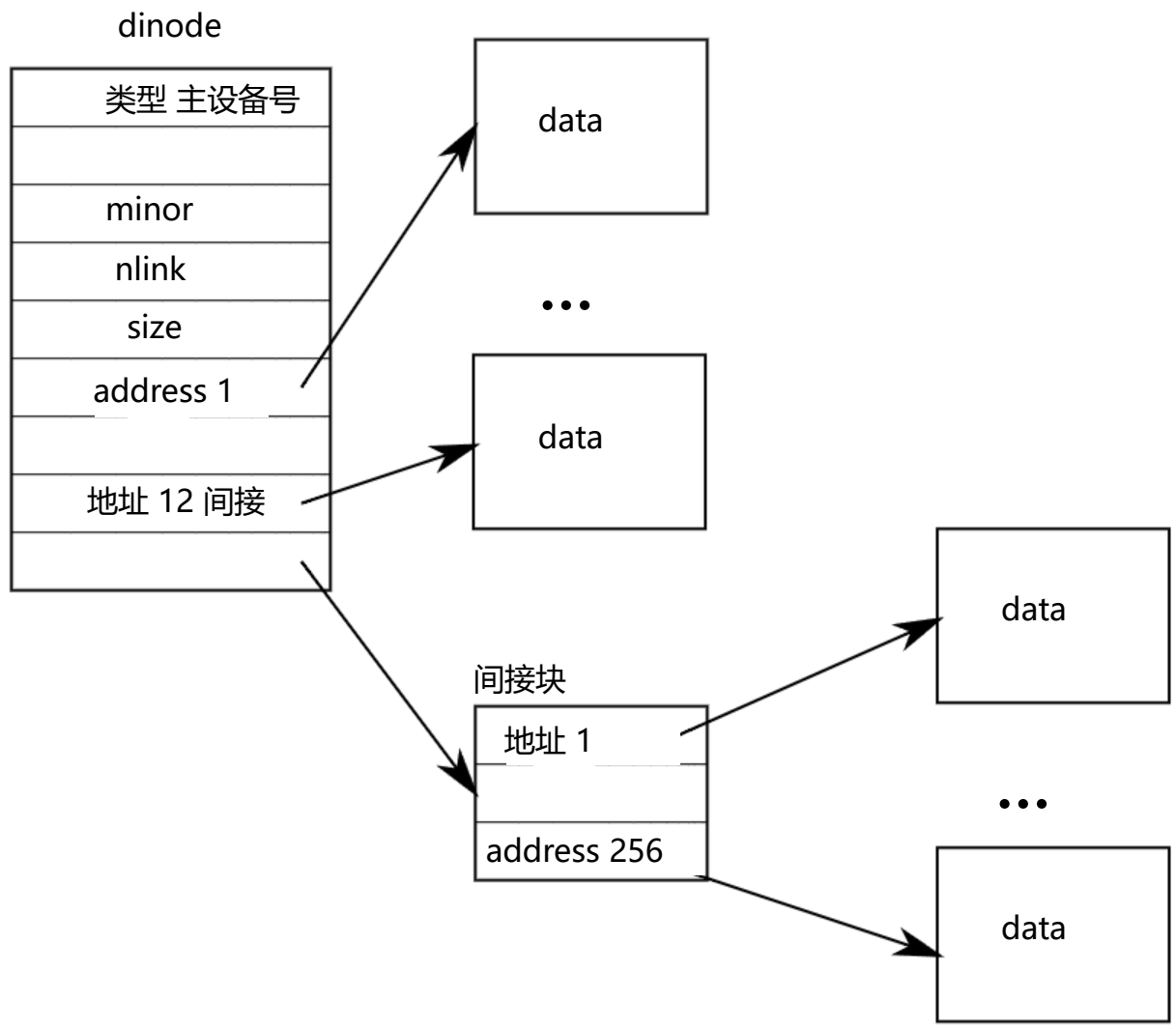

图 8.3：磁盘上文件的表示。

如果崩溃发生在最后一个进程关闭文件的文件描述符之前，那么文件将在磁盘上被标记为已分配，但没有目录项指向它。文件系统以两种方式之一处理这种情况。简单的解决方案是在恢复时，即重启后，文件系统扫描整个文件系统，查找那些标记为已分配但没有目录项指向它们的文件。如果存在这样的文件，那么可以释放这些文件。

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（例如，在超级块中）记录一个文件的 inode 号，该文件的链接计数降为零但引用计数不为零。如果文件系统在引用计数达到 0 时删除文件，则通过从列表中删除该 inode 来更新磁盘上的列表。在恢复时，文件系统释放列表中的任何文件。

Xv6 没有实现这两种解决方案，这意味着即使 inode 不再使用，磁盘上仍可能标记为已分配。这意味着随着时间的推移，xv6 可能会面临磁盘空间耗尽的风险。

8.10 代码：Inode 内容

磁盘上的 inode 结构体，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 数组中所列的块中。数据的前 NDIRECT 个块列在数组的前 NDIRECT 个条目中；这些块被称为

数据的前 NDIRECT 块在数组的前 NDIRECT 个条目中列出；这些块称为直接块。接下来的 NINDIRECT 块数据不在 inode 中列出，而是在一个称为间接块的数据块中列出。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 中列出的块中加载，而接下来的 256 kB（NINDIRECT x BSIZE）字节只能在查阅间接块后才能加载。这是一个良好的磁盘表示，但对客户端来说较为复杂。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）可以使用。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:378）首先处理简单的情况：前 `NDIRECT` 个块直接列在 inode 本身中（kernel/fs.c:383-387）。接下来的 `NINDIRECT` 个块列在 `ip->addrs[NDIRECT]` 处的间接块中。`Bmap` 读取间接块（kernel/fs.c:394），然后从块中的正确位置读取块号（kernel/fs.c:395）。如果块号超过 `NDIRECT+NINDIRECT`，`bmap` 会 panic；`writei` 包含防止这种情况发生的检查（kernel/fs.c:490）。

Bmap 根据需要分配块。ip->addrs[] 或间接条目中的零表示没有分配块。当 bmap 遇到零时，它会用新块的编号替换它们。

按需分配（kernel/fs.c:384-385）（kernel/fs.c:392-393）。

itrunc 释放文件的块，将 inode 的大小重置为零。Itrunc (kernel/fs.c:410) 首先释放直接块 (kernel/fs.c:416-421)，然后释放间接块中列出的块 (kernel/fs.c:426-429)，最后释放间接块本身 (kernel/fs.c:431-432)。

Bmap 使得 readi 和 writei 能够轻松访问 inode 的数据。Readi（kernel/fs.c:456）首先确保偏移量和计数不超过文件的末尾。从文件末尾之后开始的读取会返回错误（kernel/fs.c:461-462），而从文件末尾或跨越文件末尾开始的读取则返回比请求更少的字节（kernel/fs.c:463-464）。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:466-474）。writei (kernel/fs.c:483) 与 readi 相同，但有三个例外：从文件末尾开始或跨越文件末尾的写入会增长文件，直到达到最大文件大小 (kernel/fs.c:490-491)；循环将数据复制到缓冲区而不是从缓冲区复制出来 (kernel/fs.c:36)；如果写入扩展了文件，writei 必须更新其

大小（kernel/fs.c:504-511）。

`readi` 和 `writei` 都首先检查 `ip->type == T_DEV`。这种情况处理的是那些数据不在文件系统中的特殊设备；我们将在文件描述符层返回到这种情况。

函数 `stati`（位于 `kernel/fs.c:442`）将 inode 的元数据复制到 `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

目录在内部实现上与文件非常相似。它的 inode 类型为 T_DIR，其数据是一系列目录条目。每个条目是一个 struct dirent（kernel/fs.h:56），其中包含一个名称和一个 inode 编号。名称最多为 DIRSIZ（14）个字符；如果较短，则以 NUL（0）字节终止。inode 编号为零的目录条目是空闲的。

函数 `dirlookup`（`kernel/fs.c:527`）在目录中搜索具有给定名称的条目。

如果找到匹配的条目，它将返回一个指向相应 inode 的指针（未锁定），并将*poff 设置为目录中该条目的字节偏移量，以便调用者可以编辑它。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。调用者已经锁定了 dp，因此如果查找的是当前目录的别名“.”，在返回之前尝试锁定 inode 将会尝试重新锁定 dp 并导致死锁。（还存在更复杂的死锁场景，涉及多个进程和父目录的别名“..”；“.”并不是唯一的问题。）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:554) 将给定名称和 inode 编号的新目录项写入目录 dp。如果名称已存在，dirlink 返回错误 (kernel/fs.c:560-
564) . 主循环读取目录条目，寻找一个未分配的条目。当它找到一个时，它会提前停止循环（kernel/fs.c:538-539），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。无论哪种情况，dirlink 都会通过在偏移量 off 处写入来向目录添加一个新条目（kernel/fs.c:574-577）。

8.12 代码：路径名

路径名查找涉及一系列对 `dirlookup` 的调用，每个路径组件调用一次。`Namei` (kernel/fs.c:661) 评估路径并返回相应的 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:626) 首先决定路径评估的起点。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:630-633)。然后它使用 skipelem 依次考虑路径的每个元素 (kernel/fs.c:635)。循环的每次迭代必须在当前 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:636-640)。（锁定 ip 是必要的，不是因为 ip->type 会在脚下改变——它不会——而是因为在 ilock 运行之前，ip->type 不能保证已经从磁盘加载。）如果调用是 nameiparent 并且这是最后一个路径元素，循环会提前停止，根据 nameiparent 的定义；最后一个路径元素已经被复制到 name 中，所以 namex 只需要返回未锁定的 ip (kernel/fs.c:641-645)。最后，循环使用 dirlookup 查找路径元素，并通过设置 ip = next 为下一次迭代做准备 (kernel/fs.c:646-651)。当循环耗尽路径元素时，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，以读取路径名中遍历的目录的 inode 和目录块（如果它们不在缓冲区缓存中）。Xv6 经过精心设计，使得如果一个内核线程调用 `namex` 时因磁盘 I/O 而阻塞，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。
这种并发性引入了一些挑战。例如，当一个内核线程正在查找路径名时，另一个内核线程可能正在通过取消链接目录来更改目录树。一个潜在的风险是，查找操作可能正在搜索已被另一个内核线程删除的目录，并且其块已被重新用于另一个目录或文件。

Xv6 避免了这种竞争。例如，当在 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 获取的 inode。Iget 增加了 inode 的引用计数。只有在从 dirlookup 接收到 inode 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 还不会删除该 inode，因为 inode 的引用计数仍然大于零。

另一个风险是死锁。例如，在查找“.”时，next 指向与 ip 相同的 inode。在释放 ip 上的锁之前锁定 next 会导致死锁。为了避免这种死锁，namex 在获取 next 的锁之前会解锁目录。这里我们再次看到 iget 和 ilock 之间的分离为何如此重要。

8.13 文件描述符层

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

Xv6 为每个进程提供了自己的打开文件表，或称为文件描述符，正如我们在第 1 章中所见。每个打开的文件都由一个 `struct file`（kernel/file.h:1）表示，它是对 inode 或管道的封装，并包含一个 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，不同的实例将具有不同的 I/O 偏移量。另一方面，一个打开的文件（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` 与子进程共享文件，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。文件可以以只读、只写或读写方式打开。`readable` 和 `writable` 字段用于跟踪这一点。

系统中所有打开的文件都保存在一个全局文件表 ftable 中。文件表具有分配文件（filealloc）、创建重复引用（filedup）、释放引用（fileclose）以及读写数据（fileread 和 filewrite）的功能。

前三个函数遵循了现在熟悉的形式。`filealloc`（`kernel/file.c:30`）扫描文件表以寻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`）减少引用计数。当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 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，然后推进它 (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 函数需要

调用者处理锁定（kernel/file.c:94-96）（kernel/file.c:121-124）（kernel/file.c:163-166）。in-ode 锁定有一个方便的副作用，即读写偏移量是原子更新的，因此同时写入同一文件的多个进程不会覆盖彼此的数据，尽管

尽管它们的写入可能会交错在一起。

8.14 代码：系统调用

通过下层提供的函数，大多数系统调用的实现变得简单（参见(kernel/sysfile.c)）。有几个调用值得更仔细地研究。

函数 `sys_link` 和 `sys_unlink` 编辑目录，创建或删除对 inode 的引用。它们是使用事务的另一个很好的例子。`Sys_link`（kernel/sysfile.c:120）首先获取其参数，两个字符串 `old` 和 `new`（kernel/sysfile.c:125）。假设
old 存在且不是一个目录 (kernel/sysfile.c:129-132)，sys_link 增加其 ip->nlink
然后，sys_link 调用 nameiparent 来找到 new 的父目录和最终路径元素（kernel/sysfile.c:145），并创建一个指向 old 的 inode 的新目录项（kernel/sysfile.c:148）。新的父目录必须存在，并且与现有的 inode 位于同一设备上：inode 编号仅在单个磁盘上具有唯一意义。如果发生此类错误，sys_link 必须返回并递减 ip->nlink。

事务简化了实现，因为它需要更新多个磁盘块，但我们不必担心它们的执行顺序。它们要么全部成功，要么全部失败。例如，在没有事务的情况下，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，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 为现有的 inode 创建一个新名称。函数 create (kernel/sysfile.c:242) 为一个新的 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 创建一个新的目录，mkdev 创建一个新的设备文件。与 sys_link 类似，create 首先调用 nameiparent 获取父目录的 inode。然后调用 dirlookup 检查名称是否已经存在 (kernel/sysfile.c:252)。如果名称已经存在，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 被用于 open (type == T_FILE) 并且存在的名称本身是一个普通文件，那么 open 将其视为成功，因此 create 也这样做 (kernel/sysfile.c:256)。否则，这是一个错误 (kernel/sysfile.c:257-258)。如果名称不存在，create 现在使用 ialloc 分配一个新的 inode (kernel/sysfile.c:261)。如果新的 inode 是一个目录，create 使用 . 和
.. 条目。最后，既然数据已经正确初始化，create 可以将其链接到父目录中（kernel/sysfile.c:274）。与 sys_link 一样，create 同时持有两个 inode 锁：ip 和 dp。不存在死锁的可能性，因为 inode ip 是刚分配的：系统中没有其他进程会持有 ip 的锁然后尝试锁定 dp。

使用 create，可以轻松实现 sys_open、sys_mkdir 和 sys_mknod。Sys_open (kernel/sysfile.c:287) 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 传递了 O_CREATE 标志，它会调用 create (kernel/sysfile.c:301)。否则，它
调用 namei (kernel/sysfile.c:307)。Create 返回一个锁定的 inode，但 namei 不会，因此 sys_open 必须锁定 inode 本身。这提供了一个方便的地方来检查目录是否仅以只读方式打开，而不是写入。假设 inode 以某种方式获取，sys_open 会分配一个文件和一个文件描述符（kernel/sysfile.c:325），然后填充文件（kernel/sysfile.c:337-

c:337342）。请注意，其他进程无法访问部分初始化的文件，因为它仅存在于当前进程的表中。第 7 章在我们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式，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，它将在此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

现实世界中的操作系统中的缓冲区缓存比 xv6 的要复杂得多，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 一样，使用了一个简单的最近最少使用（LRU）淘汰策略；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和使用堆进行 LRU 淘汰。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。提交操作不能与文件系统调用并发进行。系统会记录整个块，即使块中只有几个字节被更改。它执行同步的日志写入，一次一个块，每个块可能需要整个磁盘旋转时间。实际的日志系统解决了所有这些问题。

日志记录并不是提供崩溃恢复的唯一方法。早期的文件系统在重启时使用清理程序（例如，UNIX 的 fsck 程序）来检查每个文件和目录以及块和 inode 的空闲列表，寻找并解决不一致性。对于大型文件系统，清理可能需要数小时，并且在某些情况下，无法以导致原始系统调用原子性的方式解决不一致性。从日志中恢复要快得多，并且在面对崩溃时使系统调用具有原子性。

Xv6 使用与早期 UNIX 相同的磁盘上的 inode 和目录基本布局；这种方案多年来一直非常持久。BSD 的 UFS/FFS 和 Linux 的 ext2/ext3 基本上使用了相同的数据结构。文件系统布局中效率最低的部分是目录，每次查找时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。Microsoft Windows 的 NTFS、Mac OS X 的 HFS 和 Solaris 的 ZFS 等，仅举几例，将目录实现为磁盘上的平衡树块。这很复杂，但保证了对数时间的目录查找。

Xv6 对磁盘故障的处理很天真：如果磁盘操作失败，xv6 会恐慌。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，因此恐慌是可以接受的。另一方面，使用普通磁盘的操作系统应该预期故障并更优雅地处理它们，以便一个文件中的块丢失不会影响文件系统其余部分的使用。

Xv6 要求文件系统必须适应一个磁盘设备且大小不变。随着大型数据库和多媒体文件推动存储需求不断增长，操作系统正在

操作系统正在开发方法来消除“每个文件系统一个磁盘”的瓶颈。基本方法是将多个磁盘组合成一个逻辑磁盘。硬件解决方案如 RAID 仍然是最受欢迎的，但当前的趋势是尽可能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，例如通过动态添加或删除磁盘来扩展或缩小逻辑设备。当然，一个可以动态扩展或缩小的存储层需要一个能够做到相同的文件系统：xv6 使用的固定大小的 inode 块数组在这种环境中效果不佳。将磁盘管理与文件系统分离可能是最干净的设计，但两者之间的复杂接口导致一些系统，如 Sun 的 ZFS，将它们结合在一起。

Xv6 的文件系统缺少现代文件系统的许多其他功能；例如，它不支持快照和增量备份。

现代 Unix 系统允许使用与磁盘存储相同的系统调用来访问多种资源：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）。这些系统通常为每个打开的文件提供一个函数指针表，每个操作对应一个指针，并通过调用该指针来执行该 inode 对该调用的实现，而不是像 xv6 那样在 fileread 和 filewrite 中使用 if 语句。网络文件系统和用户级文件系统提供了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

3. 为什么 filealloc 在文件用尽时不会 panic？为什么这种情况更常见，因此值得处理？

4. 假设在 sys_link 调用 iunlock(ip)和 dirlink 之间，与 ip 对应的文件被另一个进程取消链接。链接会被正确创建吗？为什么？

5. create 进行了四个函数调用（一个调用 ialloc，三个调用 dirlink），这些调用必须成功。如果其中任何一个失败，create 会调用 panic。为什么这是可以接受的？为什么这四个调用中的任何一个都不能失败？

6. sys_chdir 在调用 iput(cp->cwd) 之前调用了 iunlock(ip)，这可能会尝试锁定 cp->cwd 然而，将 iunlock(ip)推迟到 iput 之后不会导致死锁。为什么不会呢？

7. 实现 lseek 系统调用。支持 lseek 还需要你修改 filewrite，以便在 lseek 设置超出 f->ip->size 时用零填充文件中的空洞。

8. 在 open 中添加 O_TRUNC 和 O_APPEND，以便在 shell 中可以使用>和>>操作符。

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. 修改文件和虚拟内存系统以支持内存映射文件。

第 9 章

并发性再探

同时获得良好的并行性能、并发情况下的正确性以及易于理解的代码是内核设计中的一大挑战。直接使用锁是实现正确性的最佳途径，但并不总是可行的。本章重点介绍了 xv6 被迫以复杂方式使用锁的例子，以及 xv6 使用类似锁技术但不使用锁的例子。

9.1　锁定模式

缓存的项通常是一个锁定的挑战。例如，文件系统的块缓存（kernel/bio.c:26）存储了最多 NBUF 个磁盘块的副本。确保一个给定的磁盘块在缓存中最多只有一个副本是至关重要的；否则，不同的进程可能会对应该是同一个块的不同副本做出冲突的更改。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，帮助确保一次只有一个进程使用给定的磁盘块。然而，这个锁是不够的：如果一个块根本不在缓存中，而两个进程同时想要使用它呢？没有 struct buf（因为块还没有被缓存），因此没有东西可以锁定。Xv6 通过将额外的锁（bcache.lock）与缓存块的身份集合关联来处理这种情况。需要检查块是否被缓存的代码（例如，bget（kernel/bio.c:59）），或者更改缓存块集合的代码，必须持有 bcache.lock；在代码找到它需要的块和 struct buf 之后，它可以释放 bcache.lock 并只锁定特定的块。这是一个常见的模式：一个锁用于项集合，加上每个项一个锁。

通常，获取锁的函数也会释放它。但更准确的说法是，锁是在必须表现为原子操作的序列开始时获取的，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如此。锁的功能是强制其他使用者等待，而不是将数据固定到特定的代理上。一个例子是`yield`（kernel/proc.c:515）中的`acquire`，它在调度器线程中释放，而不是在获取锁的进程中。另一个例子是`ilock`（kernel/fs.c:289）中的`acquiresleep`；这段代码在读取磁盘时经常睡眠；它可能会在不同的 CPU 上唤醒，这意味着锁可能在不同的 CPU 上获取和释放。

释放一个由嵌入在对象中的锁保护的对象是一项微妙的任务，因为拥有锁并不足以保证释放操作是正确的。问题出现在当其他线程正在等待获取锁以使用该对象时；释放对象会隐式地释放嵌入的锁，这将导致等待的线程出现故障。一个解决方案是跟踪对象存在的引用数量，以便只有在最后一个引用消失时才释放对象。参见 pipeclose（kernel/pipe.c:59）作为示例；pi->readopen 和 pi->writeopen 跟踪管道是否有文件描述符引用它。

9.2 Lock-like 模式

在许多地方，xv6 使用引用计数或标志作为一种软锁，以指示对象已分配且不应被释放或重用。进程的 p->state 以这种方式工作，文件、inode 和 buf 结构中的引用计数也是如此。虽然在每种情况下锁都保护了标志或引用计数，但正是后者防止了对象被过早释放。

文件系统使用结构体 inode 的引用计数作为一种可以由多个进程持有的共享锁，以避免如果代码使用普通锁可能发生的死锁。例如，namex 函数（kernel/fs.c:626）中的循环依次锁定由每个路径名组件命名的目录。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，它可能会与自己发生死锁。它也可能与涉及目录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，但不锁定。

一些数据项在不同的时间由不同的机制保护，有时可能通过 xv6 代码的结构隐式地保护其免受并发访问，而不是通过显式锁。例如，当一个物理页面空闲时，它由 kmem.lock（kernel/kalloc.c:24）保护。如果该页面随后被分配为管道（kernel/pipe.c:23），则由不同的锁（嵌入的 pi->lock）保护。如果该页面被重新分配为新进程的用户内存，则根本不通过锁保护。相反，分配器不会将该页面分配给任何其他进程（直到它被释放）这一事实保护其免受并发访问。新进程内存的所有权是复杂的：首先父进程在 fork 中分配并操作它，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：数据对象在其生命周期的不同阶段可能以不同的方式受到并发保护，并且保护可能采取隐式结构的形式，而不是显式锁。

最后一个类似锁的例子是在调用 mycpu()时需要禁用中断（kernel/proc.c:68）。禁用中断会使调用代码相对于可能强制上下文切换的定时器中断是原子的，从而防止进程被移动到不同的 CPU 上。

9.3 完全没有锁

xv6 中有一些地方在没有锁的情况下共享可变数据。其中一个是在自旋锁的实现中，尽管可以将 RISC-V 原子指令视为依赖于锁。

尽管可以将 RISC-V 的原子指令视为依赖于硬件实现的锁。另一个例子是 main.c 中的 started 变量（kernel/main.c:7），用于防止其他 CPU 在 CPU 零完成 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。第三个例子是 proc.c 中 p->parent 的一些使用（kernel/proc.c:398）（kernel/proc.c:306），在这些地方，适当的锁定可能会导致死锁，但似乎很清楚没有其他进程会同时修改 p->parent。第四个例子是 p->killed，它在持有 p->锁的情况下设置（kernel/proc.c:611），但在没有持有锁的情况下检查（kernel/trap.c:56）。

Xv6 包含了一些情况，其中一个 CPU 或线程写入一些数据，而另一个 CPU 或线程读取这些数据，但没有专门的锁来保护这些数据。例如，在 fork 中，父进程写入子进程的用户内存页，而子进程（可能是另一个线程，甚至在不同的 CPU 上）读取这些页；没有显式的锁来保护这些页。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（见第 6 章），因为在没有内存屏障的情况下，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，而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

锁定主要是为了在正确性的前提下抑制并行性。由于性能也很重要，内核设计者通常需要考虑如何以既能实现正确性又能获得良好并行性的方式使用锁。虽然 xv6 并未系统性地设计为高性能，但仍然值得考虑哪些 xv6 操作可以并行执行，以及哪些操作可能在锁上发生冲突。

xv6 中的管道是一个相当好的并行性示例。每个管道都有自己的锁，因此不同的进程可以在不同的 CPU 上并行读写不同的管道。然而，对于给定的管道，写者和读者必须等待对方释放锁；他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）必须阻塞，但这并不是由于锁定机制造成的。

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，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，因此它们不必互相等待。然而，一旦进入调度程序，两个 CPU 在搜索可运行进程表时可能会在锁上发生冲突。也就是说，xv6 在上下文切换期间可能会从多个 CPU 中获得性能提升，但可能不如它本可以达到的那样多。

另一个例子是不同 CPU 上的不同进程并发调用 fork。这些调用可能不得不相互等待 pid_lock 和 kmem.lock，以及搜索进程表以查找 UNUSED 进程所需的每个进程锁。另一方面，两个 forking 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，锁定方案在某些情况下牺牲了并行性能。在每种情况下，都可以通过更复杂的设计获得更多的并行性。是否值得这样做取决于细节：相关操作被调用的频率有多高，操作持续的时间有多长。

代码在持有竞争锁的情况下花费的时间，有多少 CPU 可能同时运行冲突操作，代码的其他部分是否是更严格的瓶颈。很难猜测给定的锁定方案是否会导致性能问题，或者新设计是否显著更好，因此通常需要在现实工作负载上进行测量。

9.5 练习

1. 修改 xv6 的管道实现，以允许对同一管道的读取和写入在不同核心上并行进行。
2. 修改 xv6 的 scheduler()以减少当不同核心同时寻找可运行进程时的锁争用。
3. 消除 xv6 的 fork()中的一些序列化。

第 10 章

总结

本文通过逐行研究一个操作系统 xv6，介绍了操作系统中的主要思想。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；其他代码行则展示了如何实现特定的操作系统思想，并且可以很容易地用不同的方式完成（例如，更好的调度算法、更好的磁盘数据结构来表示文件、更好的日志记录以支持并发事务等）。所有这些思想都在一个非常成功的系统调用接口——Unix 接口的背景下进行了说明，但这些思想也适用于其他操作系统的设计。

参考文献

[1] RISC-V 指令集手册：特权架构。https://riscv.org/specifications/privileged-isa/, 2019.

[2] RISC-V 指令集手册：用户级 ISA。https://riscv.org/specifications/isa-spec-pdf/, 2019.

[3] Hans-J Boehm. 线程不能作为库实现。ACM PLDI 会议，2005。

[4] Edsger Dijkstra. 协作顺序进程。https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html, 1965.

[5] Maurice Herlihy 和 Nir Shavit. 《多处理器编程的艺术，修订版》. 2012.

[6] Brian W. Kernighan. The C Programming Language. Prentice Hall Professional Technical Reference, 第 2 版, 1988.

[7] Donald Knuth. 基础算法。计算机程序设计艺术。（第二版），第 1 卷。1997 年。

[8] L Lamport. Dijkstra 并发编程问题的新解决方案。ACM 通讯，1974 年。

[9] John Lions. UNIX 6th Edition 注释. Peer to Peer Communications, 2000.

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

[11] Martin Michael 和 Daniel Durich。NS16550A：UART 设计与应用考虑操作。http://bitsavers.trailing-edge.com/components/national/_appNotes/AN-0491.pdf, 1987.

[12] David Patterson 和 Andrew Waterman. The RISC-V Reader: an open architecture Atlas. 草莓峡谷，2017 年。

[13] Dave Presotto, Rob Pike, Ken Thompson 和 Howard Trickey。Plan 9，一个分布式系统。在 1991 年春季 EurOpen 会议论文集，第 43-50 页，1991 年。

[14] Dennis M. Ritchie 和 Ken Thompson. UNIX 分时系统. Commun. ACM, 17(7):365–375, 1974 年 7 月.

Index