

xv6: a simple, Unix-like teaching operating system

Russ Cox Frans Kaashoek Robert Morris

August 31, 2020

翻译人员:

王辉宇,

反馈邮箱: huiyu.w@foxmail.com

前言和致谢

这是一篇为操作系统课准备的原稿, 它通过研究名为 xv6 的内核来探索操作系统的主要概念。xv6 是通过对 Dennis Ritchie 和 Ken Thompson 的 Unix Version 6 (v6)[14]为模型来进行组织代码的。xv6 大致沿用了 v6 的结构和风格, 但在多核 RISC-V[12]中用 ANSI C[6]实现。

这篇文章应该和 xv6 的源代码一起阅读, 这种方法受到 John Lions Commentary on UNIX 第六版[9]的启发。参见 <https://pdos.csail.mit.edu/6.S081>, 获取 v6 和 xv6 的在线资源, 包括一些 xv6 的实验。

我们在麻省理工学院的操作系统课 6.828 和 6.S081 中使用过这篇手稿。我们感谢这些课程的教师、助教和学生, 他们都对 xv6 做出了直接或间接的贡献。我们尤其要感谢 Adam Belay、Austin Clements 和 Nickolai Zeldovich。最后, 我们要感谢那些通过电子邮件向我们发送文本中的错误或改进建议的人。Abutalib Aghayev、Sebastian Boehm、Anton Burtsev、Raphael Carvalho、Tej Chajed、Rasit Eskicioglu、Color Fuzzy、Giuseppe、Tao Guo、Naoki Hayama、Robert Hildermand、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、zerbib、Xi Wang、Zou Chang Wei。

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

目录

xv6: a simple, Unix-like teaching operating system.....	1
前言和致谢.....	2
Chapter 1 Operating system interfaces.....	6
1.1 Processes and memory	7
1.2 I/O and File descriptors.....	9
1.3 Pipes	12
1.4 File system	13
1.5 Real world	15
1.6 Exercises	16
Chapter 2 Operating system organization	17
2.1 Abstracting physical resources	17
2.2 User mode, supervisor mode, and system calls	18
2.3 Kernel organization	19
2.4 Code: xv6 organization.....	20
2.5 Process overview	20
2.6 Code: starting xv6 and the first process	21
2.7 Real world	22
2.8 Exercises	22
Chapter 3 Page tables.....	23
3.1 Paging hardware	23
3.2 Kernel address space.....	25
3.3 Code: creating an address space	26
3.4 Physical memory allocation	27
3.5 Code: Physical memory allocator.....	27
3.6 Process address space	28
3.7 Code: sbrk.....	29
3.8 Code: exec	29
3.9 Real world	30
3.10 Exercises.....	31
Chapter 4 Traps and system calls	32

4.1 RISC-V trap machinery.....	32
4.2 Traps from user space	33
4.3 Code: Calling system calls	35
4.5 Traps from kernel space.....	35
4.7 Real world	36
4.8 Exercises	37
Chapter 5 Interrupts and device drivers.....	38
5.1 Code: Console input.....	38
5.2 Code: Console output.....	39
5.3 Concurrency in drivers.....	39
5.4 Timer interrupts	40
5.5 Real world	40
5.6 Exercises	41
Chapter 6 Locking.....	42
6.1 Race conditions.....	42
6.2 Code: Locks	45
6.3 Code: Using locks.....	46
6.4 Deadlock and lock ordering	46
6.5 Locks and interrupt handlers.....	47
6.6 Instruction and memory ordering.....	48
6.7 Sleep locks.....	49
6.8 Real world	49
6.9 Exercises	50
Chapter 7 Scheduling	51
7.1 Multiplexing.....	51
7.2 Code: Context switching	51
7.3 Code: Scheduling	52
7.4 Code: mycpu and myproc.....	53
7.5 Sleep and wakeup.....	54
7.6 Code: Sleep and wakeup	57
7.7 Code: Pipes.....	58
7.8 Code: Wait, exit, and kill.....	58

7.9 Real world	60
7.10 Exercises	61
Chapter 8 File system	63
8.1 Overview.....	63
8.2 Buffer cache layer	64
8.3 Code: Buffer cache.....	65
8.4 Logging layer.....	66
8.5 Log design	66
8.6 Code: logging	67
8.7 Code: Block allocator	68
8.8 Inode layer.....	68
8.9 Code: Inodes.....	69
8.10 Code: Inode content.....	71
8.11 Code: directory layer.....	72
8.12 Code: Path names.....	73
8.13 File descriptor layer	73
8.14 Code: System calls	74
8.15 Real world.....	75
8.16 Exercises	76
Chapter 9 Concurrency revisited	78
9.1 Locking patterns	78
9.2 Lock-like patterns.....	78
9.3 No locks at all	79
9.4 Parallelism	79
9.5 Exercises	80

Chapter 1 Operating system interfaces

操作系统的工作是在多个程序之间共享一台计算机, 并提供一套比硬件单独支持更有用的服务。操作系统管理和抽象低级硬件, 因此, 例如, 文字处理程序 (word processor) 不需要关心使用的何种磁盘硬件。操作系统在多个程序之间共享硬件, 使它们能同时运行 (或看起来是同时运行)。最后, 操作系统为程序提供了可控的交互方式, 使它们能够共享数据或共同工作。

操作系统通过接口为用户程序提供服务。设计一个好的接口很困难的。一方面, 我们希望接口是简单而单一的, 因为这样更容易得到正确的实现。另一方面, 我们可能会想为应用程序提供许多复杂的功能。解决这种矛盾的诀窍是设计出依靠一些机制的接口, 这些机制可以通过组合提高通用性 (如管道)。

本书用一个单一的操作系统作为具体的例子来说明操作系统的概念。该操作系统 xv6 提供了 Ken Thompson 和 Dennis Ritchie 的 Unix 操作系统[14]所介绍的基本接口, 同时也模仿了 Unix 的内部设计。Unix 提供了一个单一的接口, 其机制结合得很好, 提供了惊人的通用性。这种接口非常成功, 以至于现代操作系统 BSD、Linux、Mac OS X、Solaris, 甚至微软 Windows 都有类似 Unix 的接口。理解 xv6 是理解这些系统和许多其它系统的一个良好开端。

如图 1.1 所示, xv6 采用了传统的**内核**形式, 内核是一个特殊程序, 可以为其他运行进程提供服务。每个正在运行的程序, 称为进程, 拥有自己的内存, 其中包含指令、数据和堆栈。指令实现了程序的计算。数据是计算操作对象。栈允许了函数调用。一台计算机通常有许多进程, 但只有一个内核。

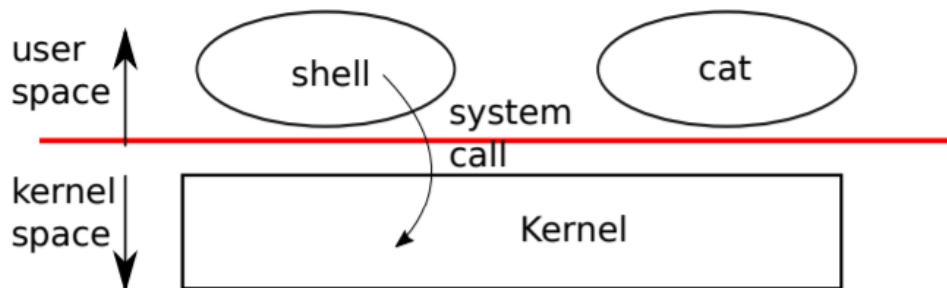


Figure 1.1: A kernel and two user processes.

当一个进程需要调用一个内核服务时, 它就会调用**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, 一个进程在用户空间和内核空间中交替执行。

内核使用 CPU¹ 提供的硬件保护机制来确保在用户空间中执行的每个进程只能访问其自己的内存。内核运行时拥有硬件特权, 可以访问这些受到保护的资源; 用户程序执行时没有这些特权。当用户程序调用系统调用(接口)时, 硬件提高特权级别并开始执行内核中预先安排的函数。

¹ 本文一般用 CPU 中央处理单元的缩写来指代执行计算的硬件元素。其他文本(如 RISC-V 规范)也使用处理器、内核和 hart 等词代替 CPU。

内核提供的系统调用集合是用户程序看到的接口。xv6 内核提供了传统 Unix 内核所提供的服务和系统调用的一个子集。图 1.2 列出了 xv6 的所有系统调用。

系统调用	描述
<code>int fork()</code>	创建一个进程，返回子进程的 PID
<code>int exit(int status)</code>	终止当前进程，status 传递给 <code>wait()</code> 。不会返回
<code>int kill(int pid)</code>	终止给定 PID 的进程，成功返回 0，失败返回 -1
<code>int getpid()</code>	返回当前进程的 PID
<code>int sleep(int n)</code>	睡眠 n 个时钟周期
<code>int exec(char *file, char *argv[])</code>	通过给定参数加载并执行一个文件；只在错误是返回
<code>char *sbrk(int n)</code>	使进程内存增加 n 字节，返回新内存的起始地址
<code>int write(int fd, char *buf, int n)</code>	将 buf 中 n 字节写入到文件描述符中；返回 n
<code>int read(int fd, char *buf, int n)</code>	从文件描述符中读取 n 字节到 buf；返回读取字节数，文件结束为 0
<code>int close(int fd)</code>	释放一个文件描述符
<code>int dup(int fd)</code>	返回一个新文件描述符，其引用与 fd 相同的文件
<code>int pipe(int p[])</code>	创建管道，将读/写文件描述符放置在 p[0]和 p[1]
<code>int chdir(char *dir)</code>	改变当前目录
<code>int mkdir(char *dir)</code>	创建新目录
<code>int mknod(char *file, int, int)</code>	创建新设备文件
<code>int fstat(int fd, struct stat *st)</code>	将打开的文件的信息放置在*st 中
<code>int stat(char *file, struct stat *st)</code>	将命名文件信息放置在*st 中
<code>int link(char *file1, char * file2)</code>	为文件 file1 创建一个新的名称(file2)
<code>int unlink(char *file)</code>	移除一个文件

Figure 1.2 xv6 系统调用。如果没有特别说明，这些调用成功返回 0，失败返回 -1

本章其余部分概述了 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 shell（Unix 的命令行用户接口），以及使用它们。shell 对系统调用的使用说明系统调用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功能:shell 没有什么特别之处。这也意味着外壳很容易更换;因此，现代 Unix 系统有许多 shell 可供选择，每个 shell 都有自己的用户界面和脚本特性。xv6 shell 是 Unix Bourne shell 的一个简单实现。它的实现可以在 (user/sh.c:1)找到。

1.1 Processes and memory

一个 xv6 进程由用户空间内存（指令、数据和堆栈）和内核私有的进程状态组成。Xv6 的进程共享 cpu，它透明地切换当前 cpu 正在执行的进程。当一个进程暂时不使用 cpu 时，xv6 会保存它的 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 PID(进程标识符)。

可以使用 **fork** 系统调用创建一个新的进程。**Fork** 创建的新进程，称为子进程，其内存内容与调用的进程完全相同，原进程被称为父进程。在父进程和子进程中，**fork** 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 0。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);
    pid = wait((int *)0);
    printf("child %d is done\n", pid);
}
else if (pid == 0)
{
    printf("child: exiting\n");
    exit(0);
}
else
{
    printf("fork error\n");
}
```

exit 系统调用退出调用进程，并释放资源，如内存和打开的文件。**exit** 需要一个整数状态参数，通常 0 表示成功，1 表示失败。**wait** 系统调用返回当前进程的一个已退出（或被杀死）的子进程的 PID，并将该子进程的退出状态码复制到一个地址，该地址由 **wait** 参数提供；如果调用者的子进程都没有退出，则 **wait** 等待一个子进程退出。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，可以传递一个 0 地址给 **wait**。

在上面的例子中，输出为：

```
parent: child=3884
child: exiting
child 3884 is done
```

可能会以任何一种顺序输出，这取决于父进程还是子进程先执行它的 **printf** 调用。在子程序退出后，父进程的 **wait** 返回，父进程执行 **printf**。

虽然子进程最初与父进程拥有相同的内存内容，但父进程和子进程是在不同的内存和不同的寄存器中执行的：改变其中一个进程中的变量不会影响另一个进程。例如，当 **wait** 的返回值存储到父进程的 **pid** 变量中时，并不会改变子进程中的变量 **pid**。子进程中的 **pid** 值仍然为零。

exec 系统调用使用新内存映像来替换进程的内存，新内存映像从文件系统中的文件中进行读取。这个文件必须有特定的格式，它指定了文件中哪部分存放指令，哪部分是数据，在哪条指令开始，等等。xv6 使用 ELF 格式，第 3 章将详细讨论。当 **exec** 成功时，它并不返回到调用程序；相反，从文件中加载的指令在 ELF 头声明的入口点开始执行。**exec** 需要两个参数：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");
```

上述代码会执行/bin/echo 程序，并将 argv 数组作为参数。大多数程序都会忽略参数数组的第一个元素，也就是程序名称。

xv6 shell 使用上述调用来在用户空间运行程序。shell 的主结构很简单，参见 `main(user/sh.c:145)`。主循环用 `getcmd` 读取用户的一行输入，然后调用 `fork`，创建 shell 副本。父进程调用 `wait`，而子进程则运行命令。例如，如果用户向 shell 输入了 `echo hello`，那么就会调用 `runcmd`，参数为 `echo hello`。`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，那么子进程将执行 echo 程序的指令，而不是 `runcmd` 的。在某些时候，`echo` 会调用 `exit`，这将使父程序从 `main(user/sh.c:145)` 中的 `wait` 返回。

你可能会奇怪为什么 `fork` 和 `exec` 没有结合在一次调用中，我们后面会看到 shell 在实现 I/O 重定向时利用了这种分离的特性。为了避免创建相同进程并立即替换它（使用 `exec`）所带来的浪费，内核通过使用虚拟内存技术（如 `copy-on-write`）来优化这种用例的 `fork` 实现（见 4.6 节）。

Xv6 隐式分配大部分用户空间内存：`fork` 复制父进程的内存到子进程，`exec` 分配足够的内存来容纳可执行文件。一个进程如果在运行时需要更多的内存（可能是为了 `malloc`），可以调用 `sbrk(n)` 将其数据内存增长 `n` 个字节；`sbrk` 返回新内存的位置。

1.2 I/O and File descriptors

文件描述符是一个小整数，代表一个可由进程读取或写入的内核管理对象。一个进程可以通过打开一个文件、目录、设备，或者通过创建一个管道，或者通过复制一个现有的描述符来获得一个文件描述符。为了简单起见，我们通常将文件描述符所指向的对象称为文件；文件描述符接口将文件、管道和设备之间的差异抽象化，使它们看起来都像字节流。我们把输入和输出称为 I/O。

在内部，xv6 内核为每一个进程单独维护一个以文件描述符为索引的表，因此每个进程都有一个从 0 开始的文件描述符私有空间。按照约定，一个进程从文件描述符 0(标准输入)读取数据，向文件描述符 1(标准输出)写入输出，向文件描述符 2(标准错误)写入错误信息。正如我们将看到的那样，shell 利用这个约定来实现 I/O 重定向和管道。shell 确保自己总是有三个文件描述符打开 (`user/sh.c:151`)，这些文件描述符默认是控制台的文件描述符。

`read/write` 系统调用可以从文件描述符指向的文件读写数据。调用 `read(fd, buf, n)` 从文件描述符 `fd` 中读取不超过 `n` 个字节的数据，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相关的偏移量。读取从当前文件偏移量中读取数据，然后按读取的字节数推进偏移量，随后的读取将返回上次读取之后的数据。当没有更多的字节可读时，读返回零，表示文件的结束。

`write(fd, buf, n)`表示将 `buf` 中的 `n` 个字节写入文件描述符 `fd` 中, 并返回写入的字节数。若写入字节数小于 `n` 则该次写入发生错误。和 `read` 一样, `write` 在当前文件偏移量处写入数据, 然后按写入的字节数将偏移量向前推进: 每次写入都从上一次写入的地方开始。

下面的程序片段(程序 `cat` 的核心代码)将数据从其标准输入复制到其标准输出。如果出现错误, 它会向标准错误写入一条消息。

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

在这个代码片段中, 需要注意的是, `cat` 不知道它是从文件、控制台还是管道中读取的。同样, `cat` 也不知道它是在打印到控制台、文件还是其他什么地方。文件描述符的使用和 0 代表输入, 1 代表输出的约定, 使得 `cat` 可以很容易实现。

`close` 系统调用会释放一个文件描述符, 使它可以被以后的 `open`、`pipe` 或 `dup` 系统调用所重用 (见下文)。新分配的文件描述符总是当前进程中最小的未使用描述符。

文件描述符和 `fork` 相互作用, 使 I/O 重定向易于实现。`Fork` 将父进程的文件描述符表和它的内存一起复制, 这样子进程开始时打开的文件和父进程完全一样。系统调用 `exec` 替换调用进程的内存, 但会保留文件描述符表。这种行为允许 shell 通过 `fork` 实现 I/O 重定向, 在子进程中重新打开所选的文件描述符, 然后调用 `exec` 运行新程序。下面是 shell 运行 `cat < input.txt` 命令的简化版代码。

```
char *argv[2];
argv[0] = "cat";
argv[1] = 0;
if (fork() == 0)
{
    close(0); // 释放标准输入的文件描述符
    open("input.txt", O_RDONLY); // 这时 input.txt 的文件描述符为 0
                                // 即标准输入为 input.txt
    exec("cat", argv); // cat 从 0 读取, 并输出到 1, 见上个代码段
}
```

```
}
```

在子进程关闭文件描述符 0 后，**open** 保证对新打开的 **input.txt** 使用该文件描述符 0。因为此时 0 将是最小的可用文件描述符。然后 **Cat** 执行时，文件描述符 0（标准输入）引用 **input.txt**。这不会改变父进程的文件描述符，它只会修改子进程的描述符。

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 shell 的代码，shell 已经 **fork** 子 shell，**runcmd** 将调用 **exec** 来加载新的程序。

open 的第二个参数由一组用位表示的标志组成，用来控制 **open** 的工作。可能的值在文件控制(fcntl)头(kernel/fcntl.h:1-5)中定义。**O_RDONLY**, **O_WRONLY**, **O_RDWR**, **O_CREATE**, 和 **O_TRUNC**, 它们指定 **open** 打开文件时的功能，读，写，读和写，如果文件不存在创建文件，将文件截断为零。

现在应该清楚为什么 **fork** 和 **exec** 是分开调用的：在这两个调用之间，shell 有机会重定向子进程的 I/O，而不干扰父进程的 I/O 设置。我们可以假设一个由 **fork** 和 **exec** 组成的系统调用 **forkexec**，但是用这种调用来做 I/O 重定向似乎很笨拙。shell 在调用 **forkexec** 之前修改自己的 I/O 设置（然后取消这些修改），或者 **forkexec** 可以将 I/O 重定向的指令作为参数，或者（最糟糕的方案）每个程序（比如 cat）都需要自己做 I/O 重定向。

虽然 **fork** 复制了文件描述符表，但每个底层文件的偏移量都是父子共享的。想一想下面的代码。

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

在这个片段的最后，文件描述符 1 所引用的文件将包含数据 hello world。父文件中的 **write**（由于有了 **wait**，只有在子文件完成后才会运行）会从子文件的 **write** 结束的地方开始。这种行为有助于从 shell 命令的序列中产生有序的输出，比如(**echo hello; echo world**) >output.txt。

dup 系统调用复制一个现有的文件描述符，返回一个新的描述符，它指向同一个底层 I/O 对象。两个文件描述符共享一个偏移量，就像被 **fork** 复制的文件描述符一样。这是将 hello world 写进文件的另一种方法。

```
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的 **fork** 和 **dup** 调用从同一个原始文件描述符衍生出来的，那么这两个文件描述符共享一个偏移量。否则，文件描述符不共享偏移量，即使它们是由同一个文件的打开调用产生的。**Dup** 允许 shell 实现这样的命令：**ls existing-file non-**

`existing-file > tmp1 2>&1`。2>&1 表示 2 是 1 的复制品 (`dup(1)`)，即重定向错误信息到标准输出，已存在文件的名称和不存在文件的错误信息都会显示在文件 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。

文件描述符是一个强大的抽象，因为它们隐藏了它们连接的细节：一个向文件描述符 1 写入的进程可能是在向一个文件、控制台等设备或向一个管道写入。

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，一个用于写。将数据写入管道的一端就可以从管道的另一端读取数据。管道为进程提供了一种通信方式。

下面的示例代码运行程序 `wc`，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); // 释放文件描述符 0
    dup(p[0]); // 复制一个 p[0](管道读端)，此时文件描述符 0 (标准输入) 也
               // 引用管道读端，故改变了标准输入。
    close(p[0]);
    close(p[1]);
    exec("/bin/wc", argv); // wc 从标准输入读取数据，并写入到参数中的每
                          // 一个文件
}
else
{
    close(p[0]);
    write(p[1], "hello world\n", 12);
    close(p[1]);
}
```

程序调用 `pipe`，创建一个新的管道，并将读写文件描述符记录在数组 `p` 中，经过 `fork` 后，父进程和子进程的文件描述符都指向管道。子进程调用 `close` 和 `dup` 使文件描述符 0 引用管道的读端，并关闭 `p` 中的文件描述符，并调用 `exec` 运行 `wc`。当 `wc` 从其标准输入端读取时，它将从管道中读取。父进程关闭管道的读端，向管道写入，然后关闭写端。

如果没有数据可用，管道上的 `read` 会等待数据被写入，或者等待所有指向写端的文件描述符被关闭；在后一种情况下，读将返回 0，就像数据文件的结束一样。事实上，如果没有数据写入，读会无限阻塞，直到新数据不可能到达为止（写端被关闭），这也是子进程在

执行上面的 **wc** 之前关闭管道的写端很重要的一个原因：如果 **wc** 的一个文件描述符仍然引用了管道的写端，那么 **wc** 将永远看不到文件的关闭（被自己阻塞）。

xv6 的 shell 实现了管道，如 **grep fork sh.c | wc -l**，shell 的实现类似于上面的代码（user/sh.c:100）。执行 shell 的子进程创建一个管道来连接管道的左端和右端（去看源码，不看难懂）。然后，它在管道左端（写入端）调用 **fork** 和 **runcmd**，在右端（读取端）调用 **fork** 和 **runcmd**，并等待两者的完成²。管道的右端（读取端）可以是一个命令，也可以是包含管道的多个命令（例如，**a | b | c**），它又会分叉为两个新的子进程（一个是 **b**，一个是 **c**）。因此，shell 可以创建一棵进程树。这棵树的叶子是命令，内部（非叶子）节点是等待左右子进程完成的进程。

原则上，我们可以让内部节点（非叶节点）运行管道的左端，但这样的实现会更加复杂。考虑只做以下修改：修改 **sh.c**，使其不为 **runcmd(p->left)** **fork** 进程，直接递归运行 **runcmd(p->left)**。像这样，**echo hi | wc** 不会产生输出，因为当 **echo hi** 在 **runcmd** 中退出时，内部进程会退出，而不会调用 **fork** 来运行管道的右端。这种不正确的行为可以通过不在 **runcmd** 中为内部进程调用 **exit** 来修正，但是这种修正会使代码变得复杂：**runcmd** 需要知道该进程是否是内部进程（非叶节点）。当不为 **runcmd(p->right)** **fork** 进程时，也会出现复杂的情况。像这样的修改，**sleep 10 | echo hi** 就会立即打印出 **hi**，而不是 10 秒后，因为 **echo** 会立即运行并退出，而不是等待 **sleep** 结束。由于 **sh.c** 的目标是尽可能的简单，所以它并没有试图避免创建内部进程。

管道似乎没有比临时文件拥有更多的功能：

```
echo hello world | wc
```

不使用管道：

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

在这种情况下，管道比临时文件至少有四个优势。首先，管道会自动清理自己；如果是文件重定向，shell 在完成后必须小心翼翼地删除 **/tmp/xyz**。第二，管道可以传递任意长的数据流，而文件重定向则需要磁盘上有足够的空闲空间来存储所有数据。第三，管道可以分阶段的并行执行，而文件方式则需要在第二个程序开始之前完成第一个程序。第四，如果你要实现进程间的通信，管道阻塞读写比文件的非阻塞语义更有效率。

1.4 File system

xv6 文件系统包含了数据文件（拥有字节数组）和目录（拥有对数据文件和其他目录的命名引用）。这些目录形成一棵树，从一个被称为根目录的特殊目录开始。像 **/a/b/c** 这样的路径指的是根目录 **/** 中的 **a** 目录中的 **b** 目录中的名为 **c** 的文件或目录。不以 **/** 开头的路径是相对于调用进程的当前目录进行计算其绝对位置的，可以通过 **chdir** 系统调用来改变进程的当前目录。下面两个 **open** 打开了同一个文件（假设所有涉及的目录都存在）。

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

² 读取端会因为管道无数据且输入端未关闭而阻塞，即只能等待左边的命令执行完才能执行右边的命令，写入端需要将自己的输出写入到管道，或者关闭管道，右边的命令读取管道并将其作为自己的输入（可以没有参数），

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

前两行将进程的当前目录改为/a/b；后面两行既不引用也不改变进程的当前目录。

有一些系统调用来可以创建新的文件和目录：**mkdir** 创建一个新的目录，用 **O_CREATE** 标志创建并打开一个新的数据文件，以及 **mknod** 创建一个新的设备文件。这个例子说明了这两个系统调用的使用。

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

mknod("/console", 1, 1);
```

mknod 创建了一个引用设备的特殊文件。与设备文件相关联的是主要设备号和次要设备号(**mknod** 的两个参数)，它们唯一地标识一个内核设备。当一个进程打开设备文件后，内核会将系统的读写调用转移到内核设备实现上，而不是将它们传递给文件系统。

文件名称与文件是不同的；底层文件（非磁盘上的文件）被称为 inode³，一个 inode 可以有多个名称，称为链接。每个链接由目录中的一个项组成；该项包含一个文件名和对 inode 的引用。inode 保存着一个文件的 *metadata* (元数据)，包括它的类型（文件或目录或设备），它的长度，文件内容在磁盘上的位置，以及文件的链接数量。

fstat 系统调用从文件描述符引用的 inode 中检索信息。它定义在 **stat.h** (kernel/stat.h) 的 **stat** 结构中：

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

link 系统调用创建了一个引用了同一个 inode 的文件（文件名）。下面的片段创建了引用了同一个 inode 两个文件 a 和 b。

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

读写 a 与读写 b 是一样的，每个 inode 都有一个唯一的 inode 号来标识。经过上面的代码序列后，可以通过检查 **fstat** 的结果来确定 a 和 b 指的是同一个底层内容：两者将返回相同的 inode 号 (**ino**)，并且 **nlink** 计数为 2。

³ Inode 是 linux 和类 unix 操作系统用来储存除了文件名和实际数据的数据结构，它是用来连接实际数据和文件名的。

unlink 系统调用会从文件系统中删除一个文件名。只有当文件的链接数为零且没有文件描述符引用它时，文件的 inode 和存放其内容的磁盘空间才会被释放。

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

上面这行代码会删除 a，此时只有 b 会引用 inode。

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

这段代码是创建一个临时文件的一种惯用方式，它创建了一个无名称 inode，故会在进程关闭 **fd** 或者退出时删除文件。

Unix 提供了 shell 可调用的文件操作程序，作为用户级程序，例如 **mkdir**、**ln** 和 **rm**。这种设计允许任何人通过添加新的用户级程序来扩展命令行接口。现在看来，这个设计似乎是显而易见的，但在 Unix 时期设计的其他系统通常将这类命令内置到 shell 中（并将 shell 内置到内核中）。

1.5 Real world

Unix 将标准文件描述符、管道和方便的 shell 语法结合起来进行操作，是编写通用可重用程序的一大进步。这个想法引发了一种软件工具文化，这也是 Unix 强大和流行的主要原因，而 shell 是第一种所谓的脚本语言。Unix 系统调用接口今天仍然存在于 BSD、Linux 和 Mac OS X 等系统中。

Xv6 并不符合 POSIX 标准：它缺少许多系统调用（包括基本的系统调用，如 **lseek**），而且它提供的许多系统调用与标准不同。我们对 xv6 的主要目标是简单明了，同时提供一个简单的类似 UNIX 的系统调用接口。一些人已经添加了一些系统调用和一个简单的 C 库扩展了 xv6，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等。现代内核不断快速发展，并提供了许多超越 POSIX 的功能。

Unix 用一套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。这个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]，它把资源就是文件的概念应用到网络、图形等方面。然而，大多数 Unix 衍生的操作系统都没有遵循这一路线。

文件系统和文件描述符已经是强大的抽象。即便如此，操作系统接口还有其他模式。Multics 是 Unix 的前身，它以一种使文件存储看起来像内存的方式抽象了文件存储，产生了一种截然不同的接口。Multics 设计的复杂性直接影响了 Unix 的设计者，他们试图建立一些更简单的东西。

Xv6 没有用户系统；用 Unix 的术语来说，所有的 xv6 进程都以 root 身份运行。

本书研究的是 xv6 如何实现其类似 Unix 的接口，但其思想和概念不仅仅适用于 Unix。任何操作系统都必须将进程复用到底层硬件上，将进程相互隔离，并提供受控进程间通信的机制。在学习了 xv6 之后，您应该能够研究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 Exercises

1. 使用 UNIX 的系统调用，编写一个程序，可以在两个进程中通过管道交换一个字节的，测试它的一秒中交换次数的性能。

Chapter 2 Operating system organization

操作系统的一个关键要求是同时支持几个活动。例如，使用第 1 章中描述的系统调用接口，一个进程可以用 **fork** 创建新进程。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，即使进程的数量多于硬件 CPU 的数量，操作系统也必须保证所有的进程都有机会执行。操作系统还必须安排进程之间的隔离。也就是说，如果一个进程出现了 bug 并发生了故障，不应该影响不依赖 bug 进程的进程。然而，隔离性太强了也不可取，因为进程间可能需要进行交互，例如管道。因此，一个操作系统必须满足三个要求：多路复用、隔离和交互。

本章概述了如何组织操作系统来实现这三个要求。现实中有很多方法，但本文主要介绍以宏内核⁴为中心的主流设计，很多 Unix 操作系统都采用这种设计。本章还介绍了 xv6 进程的概述，xv6 进程是 xv6 中的隔离单元，以及 xv6 启动时第一个进程的创建。

Xv6 运行在多核⁵RISC-V 微处理器上，它的许多底层功能（例如，它的进程实现）是 RISC-V 所特有的。RISC-V 是一个 64 位的 CPU，xv6 是用 "LP64" C 语言编写的，这意味着 C 编程语言中的 long(L)和指针(P)是 64 位的，但 int 是 32 位的。本书假定读者在某种架构上做过一点机器级的编程，并懂一些 RISC-V 特有的思想。RISC-V 有用的参考资料是 "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。

一台完整的计算机中的 CPU 周围都是硬件，其中大部分是 I/O 接口的形式。Xv6 编写的代码是基于通过 "-machine virt" 选项的 qemu。这包括 RAM、包含启动代码的 ROM、与用户键盘/屏幕的串行连接以及用于存储的磁盘。

2.1 Abstracting physical resources

遇到一个操作系统，人们可能会问的第一个问题是为什么需要它呢？答案是，我们可以把图 1.2 中的系统调用作为一个库来实现，应用程序与之连接。在这个想法中，每个应用程序可以根据自己的需要定制自己的库。应用程序可以直接与硬件资源进行交互，并以最适合应用程序的方式使用这些资源（例如，实现高性能）。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。

这种系统库方式的缺点是，如果有多个应用程序在运行，这些应用程序必须正确执行。例如，每个应用程序必须定期放弃 CPU，以便其他应用程序能够运行。如果所有的应用程序都相互信任并且没有 bug，这样的 *cooperative* 分时方案可能是 OK 的。更典型的情况是，应用程序之间互不信任，并且有 bug，所以人们通常希望比 *cooperative* 方案提供 stronger 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，而将资源抽象为服务是很有帮助的。例如，Unix 应用程序只通过文件系统的 **open**、**read**、**write** 和 **close** 系统调用与文

4 与微内核设计理念相对应的理念，这也是一个源自操作系统级别的概念。对于宏内核来说，整个操作系统就是一个整体，包括了进程管理、内存管理、文件系统等

5 本文所说的 "多核" 是指多个共享内存但并行执行的 CPU，每个 CPU 都有自己的一套寄存器。本文有时使用多处理器一词作为多核的同义词，但多处理器也可以更具体地指具有多个不同处理器芯片的计算机。

件系统进行交互，而不是直接读写磁盘。这为应用程序带来了路径名的便利，而且它允许操作系统（作为接口的实现者）管理磁盘。即使不考虑隔离问题，那些有意交互的程序（或者只是希望互不干扰）很可能会发现文件系统是一个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，必要时保存和恢复寄存器状态，这样应用程序就不必意识到时间共享。这种透明性允许操作系统共享 CPU，即使一些应用程序处于无限循环中。

另一个例子是，Unix 进程使用 **exec** 来建立它们的内存映像，而不是直接与物理内存交互。这使得操作系统可以决定将进程放在内存的什么位置；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** 还允许用户将可执行文件储存在文件系统中。

Unix 进程之间的许多形式的交互都是通过文件描述符进行的。文件描述符不仅可以抽象出许多细节（例如，管道或文件中的数据存储在哪里），而且它们的定义方式也可以简化交互。例如，如果管道中的一个应用程序崩溃了，内核就会为管道中的另一个进程产生一个文件结束信号。

图 1.2 中的系统调用接口经过精心设计，既为程序员提供了便利，又提供了强隔离的可能。Unix 接口并不是抽象资源的唯一方式，但事实证明它是一种非常好的方式。

2.2 User mode, supervisor mode, and system calls

强隔离要求应用程序和操作系统之间有一个分界线。如果应用程序发生错误，我们不希望操作系统崩溃，也不希望其他应用程序崩溃。相反，操作系统应该能够清理崩溃的应用程序并继续运行其他应用程序。为了实现强隔离，操作系统必须安排应用程序不能修改（甚至不能读取）操作系统的数据结构和指令，应用程序不能访问其他进程的内存。

CPU 提供了强隔离的硬件支持。例如，RISC-V 有三种模式，CPU 可以执行指令：**机器模式**、**监督者 (supervisor) 模式**和**用户模式**。在机器模式下执行的指令具有完全的权限，一个 CPU 在机器模式下启动。机器模式主要用于配置计算机。Xv6 会在机器模式下执行几条指令，然后转为监督者模式。

在**监督者 (supervisor)** 模式下，CPU 被允许执行特权指令：例如，启用和禁用中断，读写保存页表地址的寄存器等。如果用户模式下的应用程序试图执行一条特权指令，CPU 不会执行该指令，而是切换到监督者模式，这样监督者模式的代码就可以终止应用程序，因为它做了不该做的事情。第 1 章的图 1.1 说明了这种组织方式。一个应用程序只能执行用户模式的指令（如数字相加等），被称为运行在用户空间，而处于监督者模式的软件也可以执行特权指令，被称为运行在内核空间。运行在内核空间（或监督者模式）的软件称为内核。

一个应用程序如果要调用内核函数（如 xv6 中的读系统调用），必须过渡到内核。CPU 提供了一个特殊的指令，可以将 CPU 从用户模式切换到监督模式，并在内核指定的入口处进入内核。（RISC-V 为此提供了 `ecall` 指令。）一旦 CPU 切换到监督者模式，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，然后拒绝或执行该操作。内核控制监督者模式的入口点是很重要的；如果应用程序可以决定内核的入口点，那么恶意应用程序就能够进入内核，例如，通过跳过参数验证而进入内核。

2.3 Kernel organization

一个关键的设计问题是操作系统的哪一部分应该在监督者模式下运行。一种可能是整个操作系统驻留在内核中，这样所有系统调用的实现都在监督者模式下运行。这种组织方式称为**宏内核**。

在这种组织方式中，整个操作系统以全硬件权限运行。这种组织方式很方便，因为操作系统设计者不必决定操作系统的哪一部分不需要全硬件权限。此外，操作系统的不同部分更容易合作。例如，一个操作系统可能有一个缓冲区，缓存文件系统和虚拟内存系统共享的数据。

宏内核组织方式的一个缺点是操作系统的不同部分之间的接口通常是复杂的(我们将在本文的其余部分看到)，因此操作系统开发者很容易写 bug。在宏内核中，一个错误是致命的，因为监督者模式下的错误往往会导致内核崩溃。如果内核崩溃，计算机就会停止工作，因此所有的应用程序也会崩溃。计算机必须重启。

为了降低内核出错的风险，操作系统设计者可以尽量减少在监督者模式下运行的操作系统代码量，而在用户模式下执行操作系统的大部分代码。这种内核组织方式称为**微内核**。

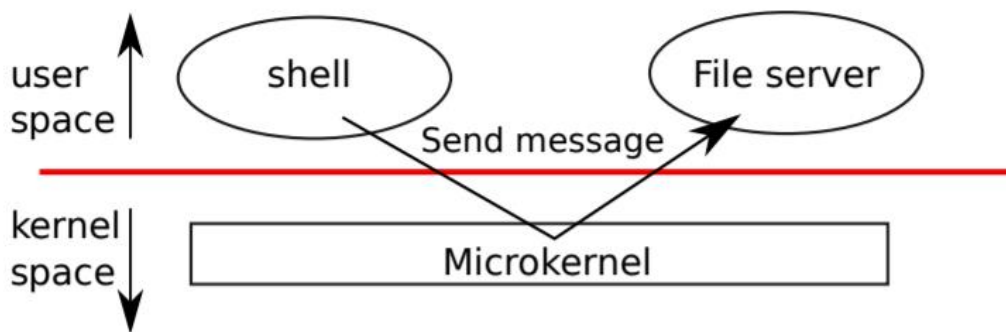


Figure 2.1: A microkernel with a file-system server

图 2.1 说明了这种微内核设计。在图中，文件系统作为一个用户级进程运行。作为进程运行的 OS 服务称为服务器。为了让应用程序与文件服务器进行交互，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进程发送消息。例如，如果一个像 shell 这样的应用程序想要读写文件，它就会向文件服务器发送一个消息，并等待响应。

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

xv6 和大多数 Unix 操作系统一样，是以宏内核的形式实现的。因此，xv6 内核接口与操作系统接口相对应，内核实现了完整的操作系统。由于 xv6 不提供很多服务，所以它的内核比一些微内核要小，但从概念上讲 xv6 是宏内核。

2.4 Code: xv6 organization

xv6 内核源码在 **kernel/**子目录下。按照模块化的概念，源码被分成了多个文件，图 2.2 列出了这些文件。模块间的接口在 **defs.h(kernel/defs.h)**中定义。

File	Description
bio.c	Disk block cache for the file system.
console.c	Connect to the user keyboard and screen.
entry.S	Very first boot instructions.
exec.c	exec() system call.
file.c	File descriptor support.
fs.c	File system.
kalloc.c	Physical page allocator.
kernelvec.S	Handle traps from kernel, and timer interrupts.
log.c	File system logging and crash recovery.
main.c	Control initialization of other modules during boot.
pipe.c	Pipes.
plic.c	RISC-V interrupt controller.
printf.c	Formatted output to the console.
proc.c	Processes and scheduling.
sleeplock.c	Locks that yield the CPU.
spinlock.c	Locks that don't yield the CPU.
start.c	Early machine-mode boot code.
string.c	C string and byte-array library.
swtch.S	Thread switching.
syscall.c	Dispatch system calls to handling function.
sysfile.c	File-related system calls.
sysproc.c	Process-related system calls.
trampoline.S	Assembly code to switch between user and kernel.
trap.c	C code to handle and return from traps and interrupts.
uart.c	Serial-port console device driver.
virtio_disk.c	Disk device driver.
vm.c	Manage page tables and address spaces.

Figure 2.2: Xv6 kernel source files.

2.5 Process overview

xv6 中的隔离单位（和其他 Unix 操作系统一样）是一个进程。进程抽象可以防止一个进程破坏或监视另一个进程的内存、CPU、文件描述符等。它还可以防止进程破坏内核，所以进程不能破坏内核的隔离机制。内核必须小心翼翼地实现进程抽象，因为一个错误或恶意的应用程序可能会欺骗内核或硬件做一些不好的事情（例如，规避隔离）。内核用来实现进程的机制包括：用户/监督模式标志、地址空间和线程的时间分割。

为了帮助实施隔离，进程抽象为程序提供了一种错觉，即它有自己的私有机器。一个进程为程序提供了一个看似私有的内存系统，或者说是地址空间，其他进程不能对其进行读写。进程还为程序提供了“私有”的 CPU，用来执行程序的指令。

Xv6 使用页表（由硬件实现）给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，定义该进程的地址空间。如图 2.3 所示，进程的用户空间内存的地址空间从虚拟地址 0 开始的。指令存放在最前面，其次是全局变量，然后是栈，最后是一个堆区（用于 **malloc**），进程可以根据需要扩展。有一些因素限制了进程地址空间的最大长度：RISC-V 上的指针是 64 位宽；硬件在页表中查找虚拟地址时只使用低的 39 位；xv6 只使用 39 位中的 38 位。因此，最大地址是 $2^{38}-1 = 0x3fffffff$ ，也就是 **MAXVA**（kernel/riscv.h:348）。在地址空间的顶端，xv6 保留了一页，用于 **trampoline** 和映射进程 **trapframe** 的页，以便切换到内核，我们将在第 4 章中解释。

xv6 内核为每个进程维护了许多状态，它将这些状态在 **proc** 结构体中(kernel/proc.h:86)。一个进程最重要的内核状态是它的页表、内核栈和运行状态。我们用 **p->xxx** 来表示 **proc** 结构的元素，例如，**p->pagetable** 是指向进程页表的指针。

每个进程都有一个执行线程（简称线程），执行进程的指令。一个线程可以被暂停，然后再恢复。为了在进程之间透明地切换，内核会暂停当前运行的线程，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）都存储在线程的栈中。每个进程有两个栈：用户栈和内核栈（**p->kstack**）。当进程在执行用户指令时，只有它的用户栈在使用，而它的内核栈是空的。当进程进入内核时（为了系统调用或中断），内核代码在进程的内核栈上执行；当进程在内核中时，它的用户栈仍然包含保存的数据，但不被主动使用。进程的线程在用户栈和内核栈中交替执行。内核栈是独立的（并且受到保护，不受用户代码的影响），所以即使一个进程用户栈被破坏了，内核也可以执行。

一个进程可以通过执行 RISC-V **ecall** 指令进行系统调用。该指令提高硬件权限级别，并将程序计数器改变为内核定义的入口点。入口点的代码会切换到内核栈，并执行实现系统调用的内核指令。当系统调用完成后，内核切换回用户栈，并通过调用 **sret** 指令返回用户空间，降低硬件特权级别，恢复执行系统调用前的用户指令。进程的线程可以在内核中阻塞等待 I/O，当 I/O 完成后，再从离开的地方恢复。

p->state 表示进程是创建、就绪、运行、等待 I/O，还是退出。

p->pagetable 以 RISC-V 硬件需要的格式保存进程的页表，当进程在用户空间执行时，xv6 使分页硬件使用进程的 **p->pagetable**。进程的页表也会记录分配给该进程内存的物理页地址。

2.6 Code: starting xv6 and the first process

为了使 xv6 更加具体，我们将概述内核如何启动和运行第一个进程。后面的章节将更详细地描述这个概述中出现的机制。

当 RISC-V 计算机开机时，它会初始化自己，并运行一个存储在只读存储器中的 **boot loader**。**Boot loader** 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 在禁用分页硬件的情况下启动：虚拟地址直接映射到物理地址。

loader 将 xv6 内核加载到物理地址 0x80000000 的内存中。之所以将内核放在 0x80000000 而不是 0x0，是因为地址范围 0x0-0x80000000 包含 I/O 设备。

_entry 处的指令设置了一个栈，这样 xv6 就可以运行 C 代码。Xv6 在文件 start.c(kernel/start.c:11)中声明了初始栈的空间，即 **stack0**。在 **_entry** 处的代码加载栈指针寄

寄存器 `sp`，地址为 `stack0+4096`，也就是栈的顶部，因为 RISC-V 的栈是向下扩张的。现在内核就拥有了栈，`_entry` 调用 `start(kernel/start.c:21)`，并执行其 C 代码。

函数 `start` 执行一些只有在机器模式下才允许的配置，然后切换到监督者模式。为了进入监督者模式，RISC-V 提供了指令 `mret`。为了进入监督者模式，RISC-V 提供了指令 `mret`。这条指令最常用从上一轮的调用中返回，上一次调用从监督者模式到机器模式。`start` 并不是从这样的调用中返回，而是把事情设置得像有过这样的调用一样：它在寄存器 `mstatus` 中把上一轮的特权模式设置为特权者模式，它把 `main` 的地址写入寄存器 `mepc` 中，把返回地址设置为 `main` 函数的地址，在特权者模式中把 0 写入页表寄存器 `satp` 中，禁用虚拟地址转换，并把所有中断和异常委托给特权者模式。

在进入特权者模式之前，`start` 还要执行一项任务：对时钟芯片进行编程以初始化定时器中断。在完成了这些基本管理后，`start` 通过调用 `mret` “返回”到监督者模式。这将导致程序计数器变为 `main` (`kernel/main.c:11`) 的地址。

在 `main(kernel/main.c:11)` 初始化几个设备和子系统后，它通过调用 `userinit(kernel/proc.c:212)` 来创建第一个进程。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`)，它通过调用 `exec` 系统调用重新进入内核。正如我们在第一章中所看到的，`exec` 用一个新的程序（本例中是 `/init`）替换当前进程的内存和寄存器。一旦内核完成 `exec`，它就会在 `/init` 进程中返回到用户空间。`init` (`user/init.c:15`) 在需要时会创建一个新的控制台设备文件，然后以文件描述符 0、1 和 2 的形式打开它。然后它在控制台上启动一个 shell。这样系统就启动了。

2.7 Real world

在现实世界中，既可以找到宏内核，也可以找到微内核。许多 Unix 内核都是宏内核。例如，Linux 的内核，尽管有些操作系统的功能是作为用户级服务器运行的（如 windows 系统）。L4、Minix 和 QNX 等内核是以服务器的形式组织的微内核，并在嵌入式环境中得到了广泛的部署。大多数操作系统都采用了进程概念，大多数进程都与 xv6 的相似。

然而，现代操作系统支持进程可以拥有多个线程，以允许一个进程利用多个 CPU。在一个进程中支持多个线程涉及到不少 xv6 没有的机制，包括潜在的接口变化(如 Linux 的 `clone`，`fork` 的变种)，以控制线程共享进程的哪些方面。

2.8 Exercises

1、你可以使用 `gdb` 来观察 `kernel mode` 到 `user mode` 的第一次转换。运行 `make qemu-gdb`。在同一目录下的另一个窗口中，运行 `gdb`。输入 `gdb` 命令 `break *0x3ffffff10e`，这将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，并即将执行 `sret`。`gdb` 现在应该显示它正在地址 `0x0` 处执行，该地址在 `initcode.S` 的用户空间开始处。

Chapter 3 Page tables

页表是操作系统为每个进程提供自己私有地址空间和内存的机制。页表决定了内存地址的含义，以及物理内存的哪些部分可以被访问。它们允许 xv6 隔离不同进程的地址空间，并将它们映射到物理内存上。页表还提供了一个间接层次，允许 xv6 执行一些技巧：在几个地址空间中映射同一内存(trampoline 页)，以及用一个未映射页来保护内核和用户的栈。本章其余部分将解释 RISC-V 硬件提供的页表以及 xv6 如何使用它们。

3.1 Paging hardware

提醒一下，RISC-V 指令(包括用户和内核)操作的是虚拟地址。机器的 RAM，或者说物理内存，是用物理地址来做索引的，RISC-V 分页硬件⁶将这两种地址联系起来，通过将每个虚拟地址映射到物理地址上。

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，一个 RISC-V 页表在逻辑上是一个 2^{27} (134,217,728) **页表项 (Page Table Entry, PTE)** 的数组。每个 PTE 包含一个 44 位的**物理页号(Physical Page Number, PPN)**和一些标志位。分页硬件通过利用 39 位中的高 27 位索引到页表中找到一个 PTE 来转换一个虚拟地址，并计算出一个 56 位的物理地址，它的前 44 位来自于 PTE 中的 PPN，而它的后 12 位则是从原来的虚拟地址复制过来的。图 3.1 显示了这个过程，在逻辑上可以把页表看成是一个简单的 PTE 数组（更完整的描述见图 3.2）。页表让操作系统控制虚拟地址到物理地址的转换，其粒度为 4096 (2^{12}) 字节的对齐块。这样的分块称为页。

在 Sv39 RISC-V 中，虚拟地址的前 25 位不用于转换地址；将来，RISC-V 可能会使用这些位来定义更多的转换层。物理地址也有增长的空间：在 PTE 格式中，物理页号还有 10 位的增长空间。

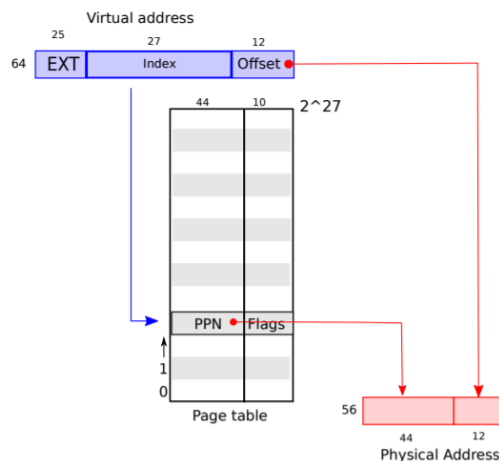


Figure 3.1: RISC-V virtual and physical addresses, with a simplified logical page table.

⁶ 一般指内存管理单元(Memory Management Unit, MMU)

如图 3.2 所示，实际转换分三步进行。一个页表以三层树的形式存储在物理内存中。树的根部是一个 4096 字节的页表页，它包含 512 个 PTE，这些 PTE 包含树的下一级页表页的物理地址。每一页都包含 512 个 PTE，用于指向下一个页表或物理地址。分页硬件用 27 位中的顶 9 位选择根页表页中的 PTE，用中间 9 位选择树中下一级页表页中的 PTE，用底 9 位选择最后的 PTE。

如果转换一个地址所需的三个 PTE 中的任何一个不存在，分页硬件就会引发一个页面错误的异常(*page-fault exception*)，让内核来处理这个异常（见第 4 章）。这种三层结构的一种好处是，当有大范围的虚拟地址没有被映射时，可以省略整个页表页。

每个 PTE 包含标志位，告诉分页硬件如何允许使用相关的虚拟地址。**PTE_V** 表示 PTE 是否存在：如果没有设置，对该页的引用会引起异常（即不允许）。**PTE_R** 控制是否允许指令读取到页。**PTE_W** 控制是否允许指令向写该页。**PTE_X** 控制 CPU 是否可以将页面的内容解释为指令并执行。**PTE_U** 控制是否允许用户模式下的指令访问页面；如果不设置 **PTE_U**，PTE 只能在监督者模式下使用。图 3.2 显示了一切的工作原理。标志位和与页相关的结构体定义在(kernel/riscv.h)。

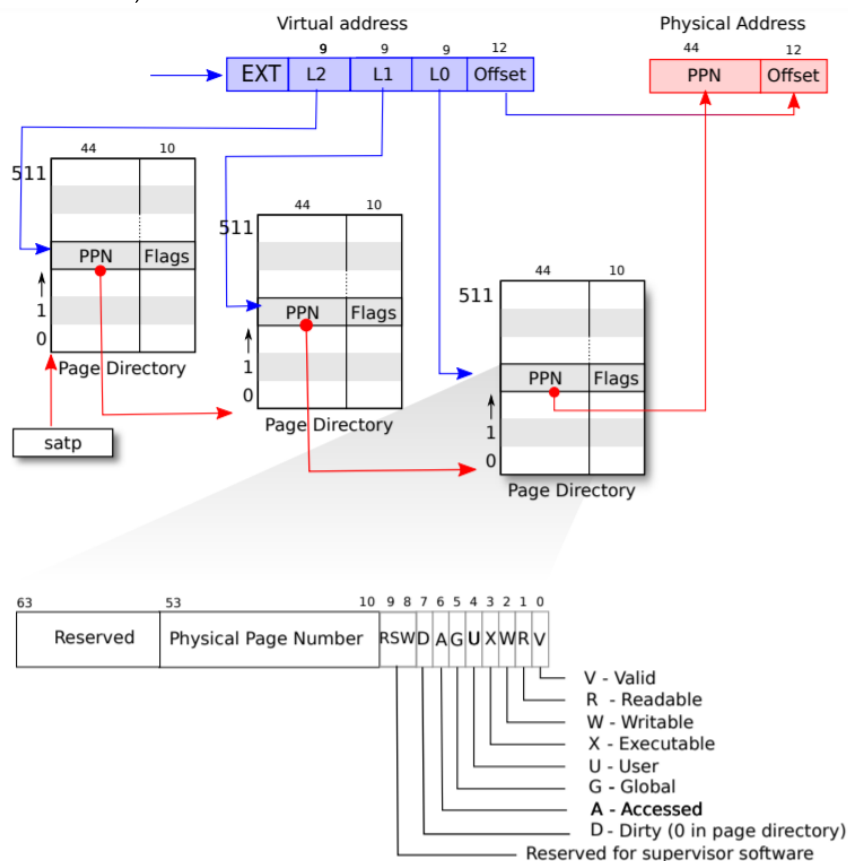


Figure 3.2: RISC-V address translation details.

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** 寄存器中。每个 CPU 都有自己的 **satp** 寄存器。一个 CPU 将使用自己的 **satp** 所指向的页表来翻译后续指令产生的所有地址。每个 CPU 都有自己的 **satp**，这样不同的 CPU 可以运行不同的进程，每个进程都有自己的页表所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。物理存储器的一个字节

有一个地址，称为物理地址。当指令操作虚拟地址时，分页硬件会将其翻译成物理地址，然后发送给 DRAM 硬件，以读取或写入存储。不像物理内存和虚拟地址，虚拟内存不是一个物理对象，而是指内核提供的管理物理内存和虚拟地址的抽象和机制的集合。

3.2 Kernel address space

Xv6 为每个进程维护页表，一个是进程的用户地址空间，外加一个内核地址空间的单页表。内核配置其地址空间的布局，使其能够通过可预测的虚拟地址访问物理内存和各种硬件资源。图 3.3 显示了这个设计是如何将内核虚拟地址映射到物理地址的。文件 (kernel/memlayout.h) 声明了 xv6 内核内存布局的常量。

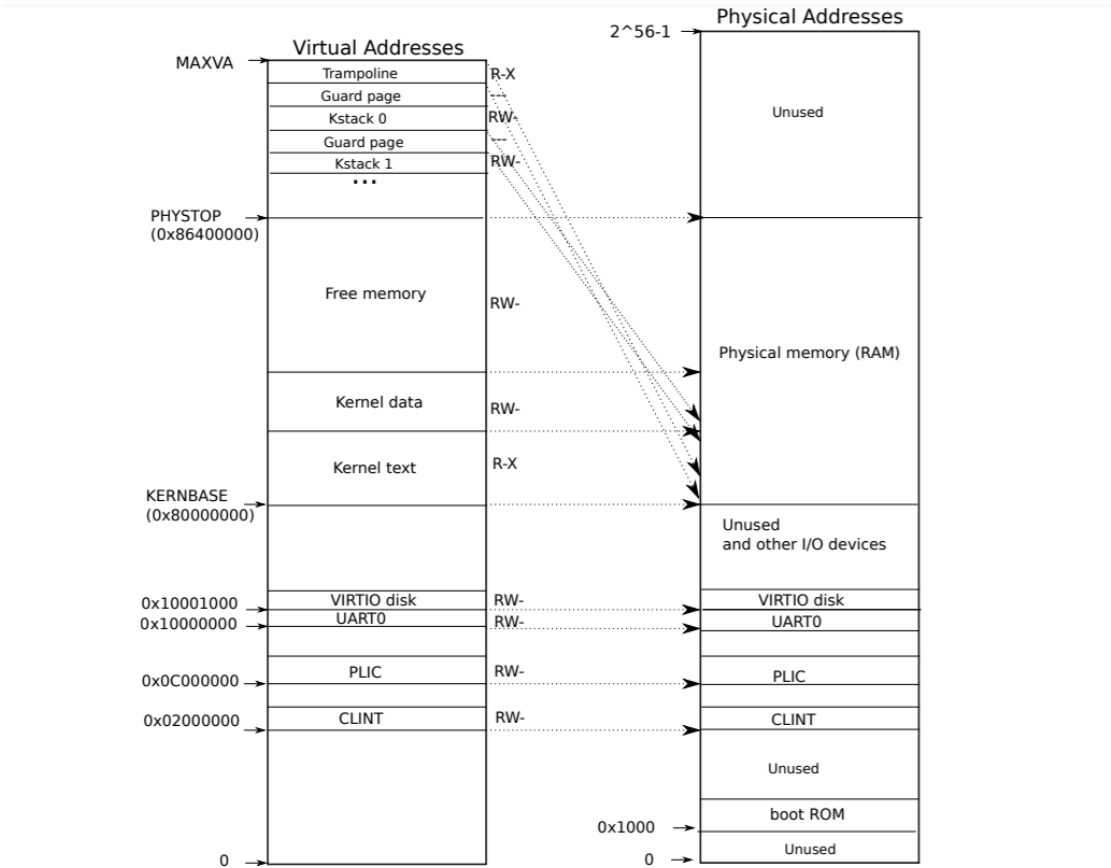


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

QEMU 模拟的计算机包含 RAM（物理内存），从物理地址 0x80000000，至少到 0x86400000，xv6 称之为 PHYSTOP。QEMU 模拟还包括 I/O 设备，如磁盘接口。QEMU 将设备接口作为 *memory-mapped(内存映射)* 控制寄存器暴露给软件，这些寄存器位于物理地址空间的 0x80000000 以下。内核可以通过读取/写入这些特殊的物理地址与设备进行交互；这种读取和写入与设备硬件而不是与 RAM 进行通信。第 4 章解释了 xv6 如何与设备交互。

内核使用“直接映射”RAM 和 *内存映射* 设备寄存器，也就是在虚拟地址上映射硬件资源，这些地址与物理地址相等。例如，内核本身在虚拟地址空间和物理内存中的位置都是

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

有几个内核虚拟地址不是直接映射的：

1. **trampoline** 页。它被映射在虚拟地址空间的顶端；用户页表也有这个映射。第 4 章讨论了 **trampoline** 页的作用，但我们在这里看到了页表的一个有趣的用例；一个物理页（存放 **trampoline** 代码）在内核的虚拟地址空间中被映射了两次：一次是在虚拟地址空间的顶部，一次是直接映射。

2. **内核栈页**。每个进程都有自己的内核栈，内核栈被映射到地址高处，所以在它后面 **xv6** 可以留下一个未映射的守护页。守护页的 PTE 是无效的（设置 **PTE_V**），这样如果内核溢出内核 **stack**，很可能引起异常，内核会报错。如果没有防护页，栈溢出时会覆盖其他内核内存，导致不正确的操作。报错还是比较好的

当内核通过高地址映射使用 **stack** 时，它们也可以通过直接映射的地址被内核访问。另一种的设计是只使用直接映射，并在直接映射的地址上使用 **stack**。在这种安排中，提供保护页将涉及到取消映射虚拟地址，否则这些地址将指向物理内存，这将很难使用。

内核为 **trampoline** 和 **text**(可执行程序代码段)映射的页会有 **PTE_R** 和 **PTE_X** 权限。内核从这些页读取和执行指令。内核映射的其他 **page** 会有 **PTE_R** 和 **PTE_W** 权限，以便内核读写这些页面的内存。守护页的映射是无效的（设置 **PTE_V**）；

3.3 Code: creating an address space

大部分用于操作地址空间和页表的 **xv6** 代码都在 **vm.c(kernel/vm.c:1)** 中。核心数据结构是 **pagetable_t**，它实际上是一个指向 RISC-V 根页表页的指针；**pagetable_t** 可以是内核页表，也可以是进程的页表。核心函数是 **walk** 和 **mappages**，前者通过虚拟地址得到 PTE，后者将虚拟地址映射到物理地址。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；其他函数用于这两种页表。**copyout** 可以将内核数据复制到用户虚拟地址，**copyin** 可以将用户虚拟地址的数据复制到内核空间地址，用户虚拟地址由系统调用的参数指定；它们在 **vm.c** 中，因为它们需要显式转换这些地址，以便找到相应的物理内存。

在启动序列的前面，**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 启用分页之前，所以地址直接指向物理内存。**Kvminit** 首先分配一页物理内存来存放根页表页。然后调用 **kvmmap** 将内核所需要的硬件资源映射到物理地址。这些资源包括内核的指令和数据，**KERNBASE** 到 **PHYSTOP** (**0x86400000**) 的物理内存，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**)，它将一个虚拟地址范围映射到一个物理地址范围。它将范围内地址分割成多页（忽略余数），每次映射一页的顶端地址。对于每个要映射的虚拟地址（页的顶端地址），**mappages** 调用 **walk** 找到该地址的最后一级 PTE 的地址。然后，它配置 PTE，使其持有相关的物理页号、所需的权限(**PTE_W**、**PTE_X** 和/或 **PTE_R**)，以及 **PTE_V** 来标记 PTE 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(见图 3.2)。**walk** 每次降低 3 级页表的 9 位。它使用每一级的 9 位虚拟地址来查找下一级页表或最后一级 (**kernel/vm.c:78**) 的 PTE。如果 PTE 无效，那么所需的物理页还没有被分配；如果 **alloc** 参数被设置 **true**，**walk** 会分配一个新的页表页，并把它的物理地址放在 PTE 中。它返回 PTE

在树的最低层的地址(kernel/vm.c:88)。

main 调用 **kvminithart** (kernel/vm.c:53) 来映射内核页表。它将根页表页的物理地址写入寄存器 **satp** 中。在这之后, CPU 将使用内核页表翻译地址。由于内核使用唯一映射, 所以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), 它由 **main** 调用, 为每个进程分配一个内核栈。它将每个栈映射在 **KSTACK** 生成的虚拟地址上, 这就为栈守护页留下了空间。**Kvmmmap** 栈的虚拟地址映射到申请的物理内存上, 然后调用 **kvminithart** 将内核页表重新加载到 **satp** 中, 这样硬件就知道新的 PTE 了。

每个 RISC-V CPU 都会在 *Translation Look-aside Buffer(TLB)*中缓存页表项, 当 xv6 改变页表时, 必须告诉 CPU 使相应的缓存 TLB 项无效。如果它不这样做, 那么在以后的某个时刻, TLB 可能会使用一个旧的缓存映射, 指向一个物理页, 而这个物理页在此期间已经分配给了另一个进程, 这样的话, 一个进程可能会在其他进程的内存上“乱写乱画”。RISC-V 有一条指令 **sfence.vma**, 可以刷新当前 CPU 的 TLB。xv6 在重新加载 **satp** 寄存器后, 在 **kvminithart** 中执行 **sfence.vma**, 也会在从内核空间返回用户空间前, 切换到用户页表的 trampoline 代码中执行 **sfence.vma**(kernel/trampoline.S:79)。

3.4 Physical memory allocation

内核必须在运行时为页表、用户内存、内核堆栈和管道缓冲区分配和释放物理内存。xv6 使用内核地址结束到 **PHYSTOP** 之间的物理内存进行运行时分配。它每次分配和释放整个 4096 字节的页面。它通过保存空闲页链表, 来记录哪些页是空闲的。分配包括从链表中删除一页; 释放包括将释放的页面添加到空闲页链表中。

3.5 Code: Physical memory allocator

分配器在 **kalloc.c**(kernel/kalloc.c:1)中。分配器的数据结构是一个可供分配的物理内存页的**空闲页链表**, 每个空闲页的链表元素是一个结构体 **run** (kernel/kalloc.c:17)。分配器从哪里获得内存来存放这个结构体呢? 它把每个空闲页的 **run** 结构体存储在空闲页本身, 因为那里没有其他东西存储。空闲链表由一个**自旋锁**保护(kernel/kalloc.c:21-24)。链表和锁被包裹在一个结构体中, 以明确锁保护的是结构体中的字段。现在, 请忽略锁以及 **acquire** 和 **release** 的调用; 第 6 章将详细研究锁。

main 函数调用 **kinit** 来初始化分配器(kernel/kallo.c:27)。**kinit** 初始空闲页链表, 以保存内核地址结束到 **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。但是它没有做, 而是假设机器有 128M 字节的 RAM。**Kinit** 通过调用 **freerange** 来添加内存到空闲页链表, **freerange** 则对每一页都调用 **kfree**。PTE 只能引用按 4096 字节边界对齐的物理地址(4096 的倍数), 因此 **freerange** 使用 **PGROUNDUP** 来确保它只添加对齐的物理地址到空闲链表中。分配器开始时没有内存;这些对 **kfree** 的调用给了它一些内存管理。

分配器有时把地址当作整数来处理, 以便对其进行运算 (如 **freerange** 遍历所有页), 有时把地址当作指针来读写内存 (如操作存储在每页中的 **run** 结构体); 这种对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, 释放和分配本质上改变了内存的类型。

函数 **kfree** (kernel/kalloc.c:47) 将被释放的内存中的每个字节设置为 1。这将使得释放内存后使用内存的代码(使用悬空引用)读取垃圾而不是旧的有效内容；希望这将导致这类代码更快地崩溃。然后 **kfree** 将页面预存入释放列表：它将 **pa** (物理地址) 转为指向结构体 **run** 的指针，在 **r->next** 中记录空闲链表之前的节点，并将释放列表设为 **r**，**kalloc** 移除并返回空闲链表中的第一个元素。

3.6 Process address space

每个进程都有一个单独的页表，当 **xv6** 在进程间切换时，也会改变页表。如图 2.3 所示，一个进程的用户内存从虚拟地址 0 开始，可以增长到 **MAXVA**(kernel/riscv.h:348)，原则上允许一个进程寻址 256GB 的内存。

当一个进程要求 **xv6** 提供更多的用户内存时，**xv6** 首先使用 **kalloc** 来分配物理页，然后将指向新物理页的 PTE 添加到进程的页表中。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。大多数进程不使用整个用户地址空间；**xv6** 使用 **PTE_V** 来清除不使用的 PTE。

我们在这里看到了几个例子，是关于使用页表的。首先，不同的进程页表将用户地址转化为物理内存的不同页，这样每个进程都有私有的用户内存。第二，每个进程都认为自己的内存具有从零开始的连续的虚拟地址，而进程的物理内存可以是不连续的。第三，内核会映射带有 **trampoline** 代码的页，该 **trampoline** 处于用户地址空间顶端，因此，在所有地址空间中都会出现一页物理内存。

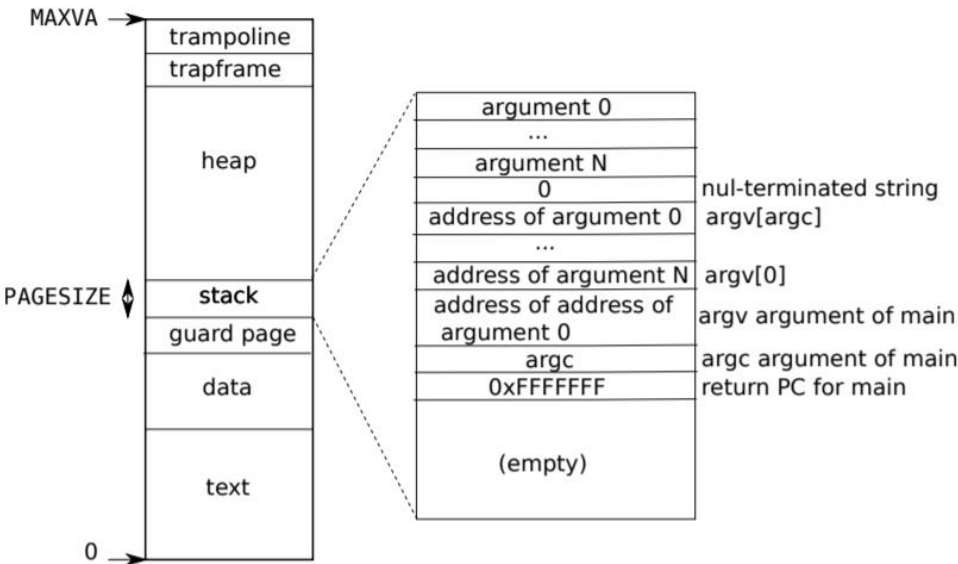


Figure 3.4: A process's user address space, with its initial stack.

图 3.4 更详细地显示了 **xv6** 中执行进程的用户内存布局。栈只有一页，图中显示的是由 **exec** 创建的初始内容。字符串的值，以及指向这些参数的指针数组，位于栈的最顶端。下面是允许程序在 **main** 启动的值，就像函数 **main(argc, argv)** 刚刚被调用一样。

为了检测用户栈溢出分配的栈内存，xv6 会在 stack 的下方放置一个无效的保护页。如果用户栈溢出，而进程试图使用栈下面的地址，硬件会因为该映射无效而产生一个页错误异常。现实世界中的操作系统可能会在用户栈溢出时自动为其分配更多的内存。

3.7 Code: sbrk

Sbrk 是一个进程收缩或增长内存的系统调用。该系统调用由函数 **growproc**(kernel/proc.c:239)实现，**growproc** 调用 **uvmalloc** 或 **uvmdealloc**，取决于 *n* 是正数还是负数。**uvmdealloc** 调用 **uvmunmap** (kernel/vm.c:174)，它使用 **walk** 来查找 PTE，使用 **kfree** 来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，也是将其作为分配给该进程的物理地址的唯一记录。这就是为什么释放用户内存（**uvmunmap** 中）需要检查用户页表的原因。

3.8 Code: exec

Exec 是创建用户地址空间的系统调用。它读取储存在文件系统上的文件用来初始化用户地址空间。**Exec** (kernel/exec.c:13)使用 **namei** (kernel/exec.c:26)打开二进制文件路径，这在第 8 章中有解释。然后，它读取 ELF 头。xv6 应用程序用 ELF 格式来描述可执行文件，它定义在(kernel/elf.h)。一个 ELF 二进制文件包括一个 ELF 头，**elfhdr** 结构体(kernel/elf.h:6)，后面是一个程序节头 (program section header) 序列，程序节头为一个结构体 **proghdr**(kernel/elf.h:25)。每一个 **proghdr** 描述了一个必须加载到内存中的程序节；xv6 程序只有一个程序节头，但其他系统可能有单独的指令节和数据节需要加载到内存。

第一步是快速检查文件是否包含一个 ELF 二进制文件。一个 ELF 二进制文件以四个字节的“魔法数字” 0x7F、‘E’、‘L’、‘F’或 **ELF_MAGIC**(kernel/elf.h:3)开始。如果 ELF 头有正确的“魔法数字”，**exec** 就会认为该二进制文件是正确的类型。

Exec 使用 **proc_pagetable**(kernel/exec.c:38)分配一个没有使用的页表，使用 **uvmalloc** (kernel/exec.c:52)为每一个 ELF 段分配内存，通过 **loadseg** (kernel/exec.c:10)加载每一个段到内存中。**loadseg** 使用 **walkaddr** 找到分配内存的物理地址，在该地址写入 ELF 段的每一页，页的内容通过 **readi** 从文件中读取。

```
# objdump -p _init
user/_init:      file format elf64-littleriscv

Program Header:
  LOAD off      0x00000000000000b0 vaddr 0x0000000000000000
                                     paddr 0x0000000000000000 align 2**3
                                     filesz 0x0000000000000084 memsz 0x00000000000000858 flags rwx
  STACK off     0x0000000000000000 vaddr 0x0000000000000000
                                     paddr 0x0000000000000000 align 2**4
                                     filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
```

用 **exec** 创建的第一个用户程序 **/init** 的程序部分 header 是上面这样的。

程序部分头的 **filesz** 可能小于 **memsz**，说明它们之间的空隙应该用 0（用于 C 语言全局变量）来填充，而不是从文件中读取。对于 **/init** 来说，**filesz** 是 2112 字节，**memsz** 是 2136

字节, 因此 `uvmmalloc` 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 2112 字节。

`exec` 在栈页的下方放置了一个不可访问页, 这样程序如果试图使用多个页面, 就会出现故障。这个不可访问的页允许 `exec` 处理过大的参数; 在这种情况下, `exec` 用来复制参数到栈的 `copyout(kernel/vm.c:355)` 函数会注意到目标页不可访问, 并返回 -1。

在准备新的内存映像的过程中, 如果 `exec` 检测到一个错误, 比如一个无效的程序段, 它就会跳转到标签 `bad`, 释放新的映像, 并返回 -1。`exec` 必须延迟释放旧映像, 直到它确定 `exec` 系统调用会成功: 如果旧映像消失了, 系统调用就不能返回 -1。`exec` 中唯一的错误情况发生在创建映像的过程中。一旦镜像完成, `exec` 就可以提交到新的页表(`kernel/exec.c:113`) 并释放旧的页表(`kernel/exec.c:117`)。

Exec 将 ELF 文件中的字节按 ELF 文件指定的地址加载到内存中。用户或进程可以将任何他们想要的地址放入 ELF 文件中。因此, **Exec** 是有风险的, 因为 ELF 文件中的地址可能会意外地或故意地指向内核。对于一个不小心的内核来说, 后果可能从崩溃到恶意颠覆内核的隔离机制(即安全漏洞)。xv6 执行了一些检查来避免这些风险。例如 `if(ph.vaddr + ph.memsz < ph.vaddr)` 检查总和是否溢出一个 64 位整数。危险的是, 用户可以用指向用户选择的地址的 `ph.vaddr` 和足够大的 `ph.memsz` 来构造一个 ELF 二进制, 使总和溢出到 `0x1000`, 这看起来像是一个有效值。在旧版本的 xv6 中, 用户地址空间也包含内核(但在用户模式下不可读/写), 用户可以选择一个对应内核内存的地址, 从而将 ELF 二进制中的数据复制到内核中。在 RISC-V 版本的 xv6 中, 这是不可能的, 因为内核有自己独立的页表; `loadseg` 加载到进程的页表中, 而不是内核的页表中。

内核开发人员很容易忽略一个关键的检查, 现实中的内核有很长一段缺少检查的空档期, 用户程序可以利用缺少这些检查来获得内核特权。xv6 在验证需要提供给内核的用户程序数据的时候, 并没有完全验证其是否是恶意的, 恶意用户程序可能利用这些数据来绕过 xv6 的隔离。

3.9 Real world

像大多数操作系统一样, xv6 使用分页硬件进行内存保护和映射。大多数操作系统对分页的使用要比 xv6 复杂得多, 它将分页和分页错误异常结合起来, 我们将在第 4 章中讨论。

Xv6 的内核使用虚拟地址和物理地址之间的直接映射, 这样会更简单, 并假设在地址 `0x8000000` 处有物理 RAM, 即内核期望加载的地方。这在 QEMU 中是可行的, 但是在真实的硬件上, 它被证明是一个糟糕的想法; 真实的硬件将 RAM 和设备放置在不可预测的物理地址上, 例如在 `0x8000000` 处可能没有 RAM, 而 xv6 期望能够在那里存储内核。更好的内核设计利用页表将任意的硬件物理内存布局变成可预测的内核虚拟地址布局。

RISC-V 支持物理地址级别的保护, 但 xv6 没有使用该功能。

在有大量内存的机器上, 使用 RISC-V 对超级页(4MB 的页)的支持可能是有意义的。当物理内存很小的时候, 小页是有意义的, 可以精细地分配和分页到磁盘。例如, 如果一个程序只使用 8 千字节的内存, 那么给它整整 4 兆字节的超级物理内存页是浪费的。更大的页在有大量内存的机器上是有意义的, 可以减少页表操作的开销。

xv6 内核缺乏一个类 `malloc` 的分配器为小程序提供内存, 这使得内核没有使用需要动

态分配的复杂数据结构，从而简化了设计。

内存分配是一个常年的热门话题，基本问题是有效利用有限的内存和为未来未知的请求做准备[7]。如今人们更关心的是速度而不是空间效率。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（在 xv6 中）只分配 4096 字节的块；一个真正的内核分配器需要处理小块分配以及大块分配。

3.10 Exercises

- 1、分析 RISC-V 的设备树（device tree），找出计算机有多少物理内存。
- 2、编写一个用户程序，通过调用 `sbrk(1)` 使其地址空间增加一个字节。运行该程序，研究调用 `sbrk` 之前和调用 `sbrk` 之后的程序页表。内核分配了多少空间？新内存的 PTE 包含哪些内容？
- 3、修改 xv6 使得内核使用超级页 4M)
- 4、修改 xv6，使用户程序间接引用⁷一个空指针时，会收到一个异常，即修改 xv6，使用户程序的虚拟地址 0 不被映射。
- 5、Unix 实现的 `exec` 传统上包括对 shell 脚本的特殊处理。如果要执行的文件以文本 `#!` 开头，那么第一行就被认为是要运行的程序来解释文件。例如，如果调用 `exec` 运行 `myprog arg1`，而 `myprog` 的第一行是 `#!/interp`，那么 `exec` 执行 `/interp myprog arg1`。在 xv6 中实现对这个约定的支持。

⁷ dereference，就是对地址取值，例如 `*p`。

Chapter 4 Traps and system calls

有三种事件会导致 CPU 搁置普通指令的执行，强制将控制权转移给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** 指令要求内核为其做某事时。另一种情况是**异常**：一条指令（用户或内核）做了一些非法的事情，如除以零或使用无效的虚拟地址。第三种情况是设备**中断**，当一个设备发出需要注意的信号时，例如当磁盘硬件完成一个读写请求时。

本书使用 **trap** 作为这些情况的通用术语。通常，代码在执行时发生 trap，之后都会被恢复，而且不需要意识到发生了什么特殊的事情。也就是说，我们通常希望 trap 是透明的；这一点对于中断来说尤其重要，被中断的代码通常不会意识到会发生 trap。通常的顺序是：trap 迫使控制权转移到内核；内核保存寄存器和其他状态，以便恢复执行；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 中返回；代码从原来的地方恢复。

xv6 内核会处理所有的 trap。这对于系统调用来说是很自然的。这对中断来说也是合理的，因为隔离要求用户进程不能直接使用设备，而且只有内核才有设备处理所需的状态。这对异常处理来说也是合理的，因为 xv6 响应所有来自用户空间的异常，并杀死该违规程序。

Xv6 trap 处理分为四个阶段：RISC-V CPU 采取的硬件行为，为内核 C 代码准备的汇编入口，处理 trap 的 C 处理程序，以及系统调用或设备驱动服务。虽然三种 trap 类型之间的共性表明，内核可以用单一的代码入口处理所有的 trap，但事实证明，为三种不同的情况，即来自用户空间的 trap、来自内核空间的 trap 和定时器中断，设置单独的汇编入口和 C trap 处理程序是很方便的。

4.1 RISC-V trap machinery

每个 RISC-V CPU 都有一组控制寄存器，内核写入这些寄存器来告诉 CPU 如何处理 trap，内核可以通过读取这些寄存器来发现已经发生的 trap。RISC-V 文档包含了完整的叙述[1]。riscv.h (kernel/riscv.h:1) 包含了 xv6 使用的定义。这里是最重要的寄存器的概述。

- **stvec**：内核在这里写下 trap 处理程序的地址；RISC-V 到这里来处理 trap。
- **sepc**：当 trap 发生时，RISC-V 会将程序计数器保存在这里（因为 **PC** 会被 **stvec** 覆盖）。**sret**(从 trap 中返回)指令将 **sepc** 复制到 pc 中。内核可以写 **sepc** 来控制 **sret** 的返回到哪里。
- **scause**：RISC -V 在这里放了一个数字，描述了 trap 的原因。
- **sscratch**：内核在这里放置了一个值，在 trap 处理程序开始的时候很方便。
- **sstatus**： **sstatus** 中的 **SIE** 位控制设备中断是否被启用，如果内核清除 **SIE**，RISC-V 将推迟设备中断，直到内核设置 **SIE**。如果内核清除 **SIE**，RISC-V 将推迟设备中断，直到内核设置 **SIE**。**SPP** 位表示 trap 是来自用户模式还是监督者模式，并控制 **sret** 返回到什么模式。

上述寄存器与在监督者模式下处理的 trap 有关，在用户模式下不能读或写。对于机器模式下处理的 trap，有一组等效的控制寄存器；xv6 只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，而且在任何时候都可能有多 CPU 在处理一个 trap。

当需要执行 trap 时，RISC-V 硬件对所有的 trap 类型（除定时器中断外）进行以下操作：

1. 如果该 trap 是设备中断，且 **sstatus SIE** 位为 0，则不要执行以下任何操作。
2. 通过清除 SIE 来禁用中断。
3. 复制 pc 到 **sepc**
4. 将当前模式（用户或监督者）保存在 sstatus 的 **SPP** 位。
5. 在 **scause** 设置该次 trap 的原因。
6. 将模式转换为监督者。
7. 将 **stvec** 复制到 pc。
8. 执行新的 pc

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 pc 以外的任何寄存器。内核软件必须执行这些任务。CPU 在 trap 期间做很少的工作的一个原因是为了给软件提供灵活性，例如，一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 CPU 的 trap 处理流程是否可以进一步简化。例如，假设 CPU 没有切换程序计数器（pc）。那么 trap 可以切换到监督者模式时，还在运行用户指令。这些用户指令可以打破用户空间/内核空间的隔离，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 必须切换到内核指定的指令地址，即 **stvec**。

4.2 Traps from user space

在用户空间执行时，如果用户程序进行了系统调用（**ecall** 指令），或者做了一些非法的事情，或者设备中断，都可能发生 trap。来自用户空间的 trap 的处理路径是 **uservec**(kernel/trampoline.S:16)，然后是 **usertrap**(kernel/trap.c:37)；返回时是 **usertrapret**(kernel/trap.c:90)，然后是 **userret**(kernel/trampoline.S:16)。

来自用户代码的 trap 比来自内核的 trap 更具挑战性，因为 **satp** 指向的用户页表并不映射内核，而且栈指针可能包含一个无效甚至恶意的值。

因为 RISC-V 硬件在 trap 过程中不切换页表，所以用户页表必须包含 **uservec** 的映射，即 **stvec** 指向的 trap 处理程序地址。**uservec** 必须切换 **satp**，使其指向内核页表；为了在切换后继续执行指令，**uservec** 必须被映射到内核页表与用户页表相同的地址。

Xv6 用一个包含 **uservec** 的 trampoline 页来满足这些条件。Xv6 在内核页表和每个用户页表中的同一个虚拟地址上映射了 trampoline 页。这个虚拟地址就是 **TRAMPOLINE**（如我们在图 2.3 和图 3.3 中看到的）。**trampoline.S** 中包含 trampoline 的内容，（执行用户代码时）**stvec** 设置为 **uservec**（kernel/trampoline.S:16）。

当 **uservec** 启动时，所有 32 个寄存器都包含被中断的代码所拥有的值。但是 **uservec** 需要能够修改一些寄存器，以便设置 **satp** 和生成保存寄存器的地址。RISC-V 通过 **sscratch** 寄存器提供了帮助。**uservec** 开始时的 **csrww** 指令将 **a0** 和 **sscratch** 的内容互换。现在用户代码的 **a0** 被保存了；**uservec** 有一个寄存器(**a0**)可以使用；**a0** 包含了内核之前放在 **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，内核先设置 **sscratch** 指向该进程的 **trapframe**，这个 **trapframe** 可以保存所有用户寄存器（kernel/proc.h:44）。因为 **satp** 仍然是指用户页表，所以 **uservec** 需要将 **trapframe** 映射到用户地址空间中。当创建每个进程时，xv6 为进程的 **trapframe** 分配一页内存，并将它映射在用户虚拟地址 **TRAPFRAME**，也就是 **TRAMPOLINE** 的下面。进程的 **p->trapframe** 也指向 **trapframe**，不过是指向它的物理地址⁸，这样内核可以通过内核页表来使用它

因此，在交换 **a0** 和 **sscratch** 后，**a0** 将指向当前进程的 **trapframe**。**uservec** 将在 **trapframe** 保存全部的寄存器，包括从 **sscratch** 读取的 **a0**。

trapframe 包含指向当前进程的内核栈、当前 CPU 的 **hartid**、**usertrap** 的地址和内核页表的地址的指针，**uservec** 将这些值设置到相应的寄存器中，并将 **satp** 切换到内核页表和刷新 TLB，然后调用 **usertrap**。

usertrap 的作用是确定 trap 的原因，处理它，然后返回(kernel/ trap.c:37)。如上所述，它首先改变 **stvec**，这样在内核中发生的 trap 将由 **kernelvec** 处理。它保存了 **sepc**(用户 PC)，这也是因为 **usertrap** 中可能会有一个进程切换，导致 **sepc** 被覆盖。如果 trap 是系统调用，**syscall** 会处理它；如果是设备中断，**devintr** 会处理；否则就是异常，内核会杀死故障进程。**usertrap** 会把用户 pc 加 4，因为 RISC-V 在执行系统调用时，会留下指向 **ecall** 指令的程序指针⁹。在退出时，**usertrap** 检查进程是否已经被杀死或应该让出 CPU（如果这个 trap 是一个定时器中断）。

回到用户空间的第一步是调用 **usertrapret**(kernel/trap.c:90)。这个函数设置 RISC-V 控制寄存器，为以后用户空间 trap 做准备。这包括改变 **stvec** 来引用 **uservec**，准备 **uservec** 所依赖的 **trapframe** 字段，并将 **sepc** 设置为先前保存的用户程序计数器。最后，**usertrapret** 在用户页表和内核页表中映射的 **trampoline** 页上调用 **userret**，因为 **userret** 中的汇编代码会切换页表。

usertrapret 对 **userret** 的调用传递了参数 **a0**，**a1**，**a0** 指向 **TRAPFRAME**，**a1** 指向用户进程页表(kernel/trampoline.S:88)，**userret** 将 **satp** 切换到进程的用户页表。回想一下，用户页表同时映射了 **trampoline** 页和 **TRAPFRAME**，但没有映射内核的其他内容。同样，事实上，在用户页表和内核页表中，**trampoline** 页被映射在相同的虚拟地址上，这也是允许 **uservec** 在改变 **satp** 后继续执行的原因。**userret** 将 **trapframe** 中保存的用户 **a0** 复制到 **sscratch** 中，为以后与 **TRAPFRAME** 交换做准备。从这时开始，**userret** 能使用的数据只有寄存器内容和 **trapframe** 的内容。接下来 **userret** 从 **trapframe** 中恢复保存的用户寄存器，对 **a0** 和 **sscratch** 做最后的交换，恢复用户 **a0** 并保存 **TRAPFRAME**，为下一次 trap 做准备，并使用 **sret** 返回用户空间。

⁸ 内核中物理地址和虚拟地址时直接映射的，所以可以在启用分页时，可以通过物理地址访问。

⁹ 执行系统调用时，进程的 pc 会指向 **ecall** 指令，这里需要加 4 清除，因为进程栈的地址空间是从高到低。

4.3 Code: Calling system calls

第 2 章以 `initcode.S` 调用 `exec` 系统调用结束 (`user/initcode.S:11`)。让我们来看看用户调用是如何在内核中实现 `exec` 系统调用的。

用户代码将 `exec` 的参数放在寄存器 `a0` 和 `a1` 中，并将系统调用号放在 `a7` 中。系统调用号与函数指针表 `syscalls` 数组(`kernel/syscall.c:108`)中的项匹配。`ecall` 指令进入内核，执行 `uservec`、`usertrap`，然后执行 `syscall`，就像我们上面看到的那样。

`syscall` (`kernel/syscall.c:133`)从 `trapframe` 中的 `a7` 中得到系统调用号，并将其作为索引在 `syscalls` 查找相应函数。对于第一个系统调用 `exec`，`a7` 将为 `SYS_exec(kernel/syscall.h:8)`，这会让 `syscall` 调用 `exec` 的实现函数 `sys_exec`。

当系统调用函数返回时，`syscall` 将其返回值记录在 `p->trapframe->a0` 中。用户空间的 `exec()` 将会返回该值，因为 RISC-V 上的 C 调用通常将返回值放在 `a0` 中。系统调用返回负数表示错误，0 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。

内核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的包装函数，参数首先会存放在寄存器中，这是 C 语言存放参数的惯例位置。内核 `trap` 代码将用户寄存器保存到当前进程的 `trapframe` 中，内核代码可以在那里找到它们。函数 `argint`、`argaddr` 和 `argfd` 从 `trapframe` 中以整数、指针或文件描述符的形式检索第 `n` 个系统调用参数。它们都调用 `argraw` 在 `trapframe` 中检索相应的数据(`kernel/syscall.c:35`)。

一些系统调用传递指针作为参数，而内核必须使用这些指针来读取或写入用户内存。例如，`exec` 系统调用会向内核传递一个指向用户空间中的字符串的指针数组。这些指针带来了两个挑战。首先，用户程序可能是错误的或恶意的，可能会传递给内核一个无效的指针或一个旨在欺骗内核访问内核内存而不是用户内存的指针。第二，`xv6` 内核页表映射与用户页表映射不一样，所以内核不能使用普通指令从用户提供的地址加载或存储。

内核实现了安全地将数据复制到用户提供的地址或从用户提供的地址复制数据的函数。例如 `fetchstr(kernel/syscall.c:25)`。文件系统调用，如 `exec`，使用 `fetchstr` 从用户空间中检索字符串文件名参数，`fetchstr` 调用 `copyinstr` 来做这些困难的工作。

`copyinstr` (`kernel/vm.c:406`) 将用户页表 `pagetable` 中的虚拟地址 `srcva` 复制到 `dst`，需指定最大复制字节数。它使用 `walkaddr` (调用 `walk` 函数) 在软件中模拟分页硬件的操作，以确定 `srcva` 的物理地址 `pa0`。`walkaddr` (`kernel/vm.c:95`)检查用户提供的虚拟地址是否是进程用户地址空间的一部分，所以程序不能欺骗内核读取其他内存。类似的函数 `copyout`，可以将数据从内核复制到用户提供的地址。

4.5 Traps from kernel space

`Xv6` 对异常的响应是相当固定：如果一个异常发生在用户空间，内核就会杀死故障进程。如果一个异常发生在内核中，内核就会 `panic`。真正的操作系统通常会以更有趣的方式进行响应。

举个例子，许多内核使用页面故障来实现 *写时复制 (copy-on-write, cow)* `fork`。要解释写时复制 `fork`，可以想一想 `xv6` 的 `fork`，在第 3 章中介绍过。`fork` 通过调用

`uvmcopy(kernel/vm.c:309)`为子进程分配物理内存，并将父进程的内存复制到子程序中，使子进程拥有与父进程相同的内存内容。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，直接实现这种方法是行不通的，因为父进程和子进程对共享栈和堆的写入会中断彼此的执行。

通过使用写时复制 fork，可以让父进程和子进程安全地共享物理内存，通过页面故障来实现。当 CPU 不能将虚拟地址翻译成物理地址时，CPU 会产生一个页面故障异常(page-fault exception)。RISC-V 有三种不同的页故障：load 页故障（当加载指令不能翻译其虚拟地址时）、store 页故障（当存储指令不能翻译其虚拟地址时）和指令页故障（当指令的地址不能翻译时）。`scause` 寄存器中的值表示页面故障的类型，`stval` 寄存器中包含无法翻译的地址。

COW fork 中的基本设计是父进程和子进程最初共享所有的物理页面，但将它们映射设置为只读。因此，当子进程或父进程执行 store 指令时，RISC-V CPU 会引发一个页面故障异常。作为对这个异常的响应，内核会拷贝一份包含故障地址的页。然后将一个副本的读/写映射在子进程地址空间，另一个副本的读/写映射在父进程地址空间。更新页表后，内核在引起故障的指令处恢复故障处理。因为内核已经更新了相关的 PTE，允许写入，所以现在故障指令将正常执行。

这个 COW 设计对 fork 很有效，因为往往子程序在 fork 后立即调用 exec，用新的地址空间替换其地址空间。在这种常见的情况下，子程序只会遇到一些页面故障，而内核可以避免进行完整的复制。此外，COW fork 是透明的：不需要对应用程序进行修改，应用程序就能受益。

页表和页故障的结合，将会有更多种有趣的可能性的应用。另一个被广泛使用的特性叫做**懒分配 (lazy allocation)**，它有两个部分。首先，当一个应用程序调用 `sbrk` 时，内核会增长地址空间，但在页表中把新的地址标记为无效。第二，当这些新地址中的一个出现页面故障时，内核分配物理内存并将其映射到页表中。由于应用程序经常要求获得比他们需要的更多的内存，所以懒分配是一个胜利：内核只在应用程序实际使用时才分配内存。像 COW fork 一样，内核可以对应用程序透明地实现这个功能。

另一个被广泛使用的利用页面故障的功能是从**磁盘上分页(paging from disk)**。如果应用程序需要的内存超过了可用的物理 RAM，内核可以交换出一些页：将它们写入一个存储设备，比如磁盘，并将其 PTE 标记为无效。如果一个应用程序读取或写入一个被换出到磁盘的页，CPU 将遇到一个页面故障。内核就可以检查故障地址。如果该地址属于磁盘上的页面，内核就会分配一个物理内存的页面，从磁盘上读取页面到该内存，更新 PTE 为有效并引用该内存，然后恢复应用程序。为了给该页腾出空间，内核可能要交换另一个页。这个特性不需要对应用程序进行任何修改，如果应用程序具有引用的位置性（即它们在任何时候都只使用其内存的一个子集），这个特性就能很好地发挥作用。

其他结合分页和分页错误异常的功能包括自动扩展堆栈和内存映射文件。

4.7 Real world

如果将内核内存映射到每个进程的用户页表中（使用适当的 PTE 权限标志），就不需要特殊的 trampoline 页了。这也将消除从用户空间 trap 进入内核时对页表切换的需求。这也可以让内核中的系统调用实现利用当前进程的用户内存被映射的优势，让内核代码直接去间接引用（对地址取值）用户指针。许多操作系统已经使用这些想法来提高效率。Xv6 没有实

现这些想法，以减少由于无意使用用户指针而导致内核出现安全漏洞的机会，并减少一些复杂性，以确保用户和内核虚拟地址不重叠。

4.8 Exercises

- 1、函数 **copyin** 和 **copyinstr** 在软件中 walk 用户页表。设置内核页表，使内核拥有用户程序的内存映射，**copyin** 和 **copyinstr** 可以使用 **memcpy** 将系统调用参数复制到内核空间，依靠硬件来完成页表的 walk。
- 2、实现内存的懒分配。
- 3、实现写时复制 fork。

Chapter 5 Interrupts and device drivers

驱动是操作系统中管理特定设备的代码，他有如下功能：1、配置设备相关的硬件，2、告诉设备需要怎样执行，3、处理设备产生的中断，4、与等待设备 I/O 的进程进行交互。驱动程序的代码写起来可能很棘手，因为驱动程序与它所管理的设备会同时执行。此外，驱动程序编写人员必须了解设备的硬件接口，但硬件接口可能是很复杂的，而且文档不够完善。

需要操作系统关注的设备通常可以被配置为产生中断，这是 trap 的一种类型。内核 trap 处理代码可以知道设备何时引发了中断，并调用驱动的中断处理程序；在 xv6 中，这个处理发生在 `devintr(kernel/trap.c:177)` 中。

许多设备驱动程序在两个 context 中执行代码：上半部分(*top half*)在进程的内核线程中运行，下半部分(*bottom half*)在中断时执行。上半部分是通过系统调用，如希望执行 I/O 的 read 和 write。这段代码可能会要求硬件开始一个操作（比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一个中断。驱动程序的中断处理程序，作为**下半部分**，推算出什么操作已经完成，如果合适的话，唤醒一个等待该操作的进程，并告诉硬件执行下一个操作。

5.1 Code: Console input

控制台驱动(`console.c`)是驱动结构的一个简单说明。控制台驱动通过连接到 RISC-V 上的 UART 串行端口硬件，接受输入的字符。控制台驱动程序每次累计一行输入，处理特殊的输入字符，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，你的按键会通过 QEMU 的模拟 UART 硬件传递给 xv6。

与驱动交互的 UART 硬件是由 QEMU 仿真的 16550 芯片[11]。在真实的计算机上，16550 将管理一个连接到终端或其他计算机的 RS232 串行链接。当运行 QEMU 时，它连接到你的键盘和显示器上。

UART 硬件在软件看来是一组**内存映射**的控制寄存器。也就是说，有一些 RISC-V 硬件的物理内存地址会连接到 UART 设备，因此加载和存储与设备硬件而不是 RAM 交互。UART 的内存映射地址从 0x10000000 开始，即 **UART0**(`kernel/memlayout.h:21`)。这里有一些 UART 控制寄存器，每个寄存器的宽度是一个字节。它们与 UART0 的偏移量定义在(`kernel/uart.c:22`)。例如，**LSR** 寄存器中一些位表示是否有输入字符在等待软件读取。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，UART 硬件就会将其从内部等待字符的 FIFO 中删除，并在 FIFO 为空时清除 **LSR** 中的就绪位。UART 传输硬件在很大程度上是独立于接收硬件的，如果软件向 **THR** 写入一个字节，UART 就会发送该字节。

Xv6 的 **main** 调用 **consoleinit** (`kernel/console.c:184`) 来初始化 UART 硬件。这段代码配置了 UART，当 UART 接收到一个字节的输入时，就产生一个接收中断，当 UART 每次完成发送一个字节的输出时，产生一个**传输完成**(*transmit complete*)中断(`kernel/uart.c:53`)。

xv6 shell 通过 **init.c**(`user/init.c:19`)打开的文件描述符从控制台读取。**consoleread** 等待输入的到来(通过中断)，输入会被缓冲在 **cons.buf**，然后将输入复制到用户空间，再然后(在一整行到达后)返回到用户进程。如果用户还没有输入完整的行，任何 **read** 进程将在 **sleep**

调用中等待(kernel/console.c:98)(第 7 章解释了 sleep 的细节)。

当用户键入一个字符时, UART 硬件向 RISC-V 抛出一个中断, 从而激活 xv6 的 **trap** 处理程序。trap 处理程序调用 **devintr**(kernel/trap.c:177), 它查看 RISC-V 的 **scause** 寄存器, 发现中断来自一个外部设备。然后它向一个叫做 PLIC[1]的硬件单元询问哪个设备中断了(kernel/trap.c:186)。如果是 UART, **devintr** 调用 **uartintr**。

uartintr (kernel/uart.c:180) 从 **UART** 硬件中读取在等待的输入字符, 并将它们交给 **consoleintr** (kernel/console.c:138); 它不会等待输入字符, 因为以后的输入会引发一个新的中断。**consoleintr** 的工作是将中输入字符积累 **cons.buf** 中, 直到有一行字符。**consoleintr** 会特别处理退格键和其他一些字符。当一个新行到达时, **consoleintr** 会唤醒一个等待的 **consoleread** (如果有的话)。

一旦被唤醒, **consoleread** 将会注意到 **cons.buf** 中的完整行, 并将其复制到用户空间, 并返回 (通过系统调用) 到用户空间。

5.2 Code: Console output

向控制台写数据的 **write** 系统调用最终会到达 **uartputc**(kernel/uart.c:87)。设备驱动维护了一个输出缓冲区(**uart_tx_buf**), 这样写入过程就不需要等待 UART 完成发送; 相反, **uartputc** 将每个字符追加到缓冲区, 调用 **uartstart** 来启动设备发送(如果还没有的话), 然后返回。**uartputc** 只有在缓冲区满的时候才会等待。

每次 UART 发送完成一个字节, 它都会产生一个中断。**uartintr** 调用 **uartstart**, **uartintr** 检查设备是否真的发送完毕, 并将下一个缓冲输出字符交给设备, 每当 UART 发送完一个字节, 就会产生一个中断。因此, 如果一个进程向控制台写入多个字节, 通常第一个字节将由 **uartputc** 调用 **uartstart** 发送, 其余的缓冲字节将由 **uartintr** 调用 **uartstart** 发送, 因为发送完成中断到来。

uartintr 调用 **uartstart**, **uartintr** 查看设备是否真的发送完成, 并将下一个缓冲输出字符交给设备, 每当 UART 发送完一个字节, 就会产生一个中断。因此, 如果一个进程向控制台写入多个字节, 通常第一个字节将由 **uartputc** 对 **uartstart** 的调用发送, 其余的缓冲字节将随着发送完成中断的到来由 **uartintr** 的 **uartstart** 调用发送。

有一个通用模式需要注意, 设备活动和进程活动需要解耦, 这将通过缓冲和中断来实现。控制台驱动程序可以处理输入, 即使没有进程等待读取它; 随后的读取将看到输入。同样, 进程可以发送输出字节, 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, 当设备速度很慢 (如 UART) 或需要立即关注 (如打印键入的字节) 时, 这种解耦尤为重要。这个 idea 有时被称为 *I/O 并发*。

5.3 Concurrency in drivers

你可能已经注意到在 **consoleread** 和 **consoleintr** 中会调用 **acquire**。**acquire** 调用会获取一个锁, 保护控制台驱动的数据结构不被并发访问。这里有三个并发风险: 不同 CPU 上的两个进程可能会同时调用 **consoleread**; 硬件可能会在一个 CPU 正在执行 **consoleread** 时, 向该 CPU 抛出一个控制台 (实际上是 UART) 中断; 硬件可能会在 **consoleread** 执行时向另

一个 CPU 抛出一个控制台中断。第 6 章探讨锁如何在这些情况下提供帮助。

需要关注驱动并发安全的另一个原因是，一个进程可能正在等待来自设备的输入，但是此时该进程已经没有在运行（被切换）。因此，中断处理程序不允许知道被中断的进程或代码。例如，一个中断处理程序不能安全地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，只是将输入数据复制到缓冲区），并唤醒上半部分代码来做剩下的工作。

5.4 Timer interrupts

Xv6 使用定时器中断来维护它的时钟，并使它能够切换正在运行的进程；`usertrap` 和 `kerneltrap` 中的 `yield` 调用会导致这种切换。每个 RISC-V CPU 的时钟硬件都会抛出时钟中断。Xv6 对这个时钟硬件进行编程，使其定期中断相应的 CPU。

RISC-V 要求在机器模式下处理定时器中断，而不是监督者模式。RISC-V 机器模式执行时没有分页，并且有一套单独的控制寄存器，因此在机器模式下运行普通的 xv6 内核代码是不实用的。因此，xv6 对定时器中断的处理与上面谈到的 trap 机制完全分离了。

在 `main` 执行之前的 `start.c`，是在机器模式下执行的，设置了接收定时器中断（`kernel/start.c:57`）。一部分工作是对 `CLINT` 硬件（`core-local interruptor`）进行编程，使其每隔一定时间产生一次中断。另一部分是设置一个类似于 `trapframe` 的 `scratch` 区域，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`，启用定时器中断。

定时器中断可能发生在用户或内核代码执行的任何时候；内核没有办法在关键操作中禁用定时器中断。因此，定时器中断处理程序必须以保证不干扰被中断的内核代码的方式进行工作。基本策略是处理程序要求 RISC-V 引发一个软件中断并立即返回。RISC-V 用普通的 trap 机制将软件中断传递给内核，并允许内核禁用它们。处理定时器中断产生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 `scratch` 区域保存一些寄存器，告诉 `CLINT` 何时产生下一个定时器中断，使 RISC-V 产生一个软件中断，恢复寄存器，然后返回。在定时器中断处理程序中没有 C 代码。

5.5 Real world

Xv6 允许在内核和用户程序执行时使用设备和定时器中断。定时器中断可以强制从定时器中断处理程序进行线程切换（调用 `yield`），即使是在内核中执行。如果内核线程有时会花费大量的时间进行计算，而不返回用户空间，那么在内核线程之间公平地对 CPU 进行时间划分的能力是很有用的。然而，内核代码需要注意它可能会被暂停（由于定时器中断），然后在不同的 CPU 上恢复，这是 xv6 中一些复杂的根源。如果设备和定时器中断只发生在执行用户代码时，内核可以变得更简单一些。

在一台典型的计算机上支持所有设备的全貌是一件很辛苦的事情，因为设备很多，设备有很多功能，设备和驱动程序之间的协议可能很复杂，而且文档也不完善。在许多操作系统中，驱动程序所占的代码比核心内核还多。

在一台正常的计算机上支持所有设备是一件很辛苦的事情，因为设备很多，设备有很多功能，设备和驱动程序之间的协议可能很复杂，而且文档也不完善。在许多操作系统中，驱动程序所占的代码比核心内核还多。

UART 驱动器通过读取 UART 控制寄存器，一次检索一个字节的数据；这种模式被称为编程 I/O，因为软件在驱动数据移动。程序化 I/O 简单，但速度太慢，无法在高数据速率下使用。需要高速移动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 中读取传出数据。现代磁盘和网络设备都使用 DMA。DMA 设备的驱动程序会在 RAM 中准备数据，然后使用对控制寄存器的一次写入来告诉设备处理准备好的数据。

UART 驱动器通过读取 UART 控制寄存器，一次读取一个字节的数据；这种模式被称为编程 I/O，因为软件在控制数据移动。程序化 I/O 简单，但速度太慢，无法在高数据速率下使用。需要高速移动大量数据的设备通常使用 *直接内存访问(direct memory access, DMA)*。DMA 设备硬件直接将传入数据写入 RAM，并从 RAM 中读取传出数据。现代磁盘和网络设备都使用 DMA。DMA 设备的驱动程序会在 RAM 中准备数据，然后使用对控制寄存器的一次写入来告诉设备处理准备好的数据。

当设备在不可预知的时间需要关注时，中断是很有用的，而且不会太频繁。但中断对 CPU 的开销很大。因此，高速设备，如网络和磁盘控制器，使用了减少对中断需求的技巧。其中一个技巧是对整批传入或传出的请求提出一个单一的中断。另一个技巧是让驱动程序完全禁用中断，并定期检查设备是否需要关注。这种技术称为 *轮询 (polling)*。如果设备执行操作的速度非常快，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。一些驱动程序会根据当前设备的负载情况，在轮询和中断之间动态切换。

UART 驱动首先将输入的数据复制到内核的缓冲区，然后再复制到用户空间。这在低数据速率下是有意义的，但对于那些快速生成或消耗数据的设备来说，这样的双重拷贝会大大降低性能。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 Exercises

- 1、修改 uart.c，使其完全不使用中断。你可能还需要修改 console.c。
2. 添加一个网卡驱动。

Chapter 6 Locking

大多数内核，包括 xv6，都会交错执行多个任务。多处理器硬件可以交错执行任务：具有多个 CPU 独立执行的计算机，如 xv6 的 RISC-V。这些多个 CPU 共享物理 RAM，xv6 利用共享来维护所有 CPU 读写的数据结构。这种共享带来了一种可能性，即一个 CPU 读取一个数据结构，而另一个 CPU 正在中途更新它，甚至多个 CPU 同时更新同一个数据；如果不仔细设计，这种并行访问很可能产生不正确的结果或破坏数据结构。即使在单处理器上，内核也可能在多个线程之间切换 CPU，导致它们的执行交错。最后，如果中断发生的时间不对，一个设备中断处理程序可能会修改与一些可中断代码相同的数据，从而破坏数据。并发一词指的是由于多处理器并行、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 可以同时调用 `kalloc`，从而并发地从空闲内存链表的头部 `push`。内核设计者喜欢允许大量的并发，因为它可以通过并行来提高性能，提高响应速度。然而，结果是内核设计者花了很多精力说服自己，尽管存在并发，但仍然是正确的。有很多方法可以写出正确的代码，有些方法比其他方法更简单。以并发下的正确性为目标的策略，以及支持这些策略的抽象，被称为并发控制技术。

Xv6 根据不同的情况，使用了很多并发控制技术，还有更多的可能。本章重点介绍一种广泛使用的技术：锁。锁提供了相互排斥的功能，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，并且代码在使用某项时总是持有关联的锁，那么该项每次只能由一个 CPU 使用。在这种情况下，我们说锁保护了数据项。虽然锁是一种简单易懂的并发控制机制，但锁的缺点是会扼杀性能，因为锁将并发操作串行化了。

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

6.1 Race conditions

作为我们为什么需要锁的一个例子，考虑两个进程在两个不同的 CPU 上调用 `wait`，`wait` 释放子进程的内存。因此，在每个 CPU 上，内核都会调用 `kfree` 来释放子进程的内存页。内核分配器维护了一个链表：`kalloc()` (`kernel/kalloc.c:69`) 从空闲页链表中 `pop` 一页内存，`kfree()` (`kernel/kalloc.c:47`) 将一页 `push` 空闲链表中。为了达到最好的性能，我们可能希望两个父进程的 `kfree`s 能够并行执行，而不需要任何一个进程等待另一个进程，但是考虑到 xv6 的 `kfree` 实现，这是不正确的。

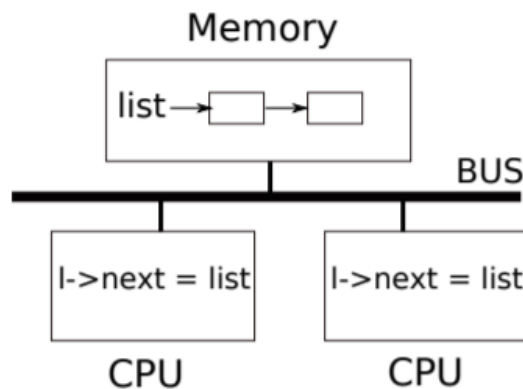


Figure 6.1: Simplified SMP architecture

图 6.1 更详细地说明了这种设置：链表在两个 CPU 共享的内存中，CPU 使用加载和存储指令操作链表。(在现实中，处理器有缓存，但在概念上，多处理器系统的行为就像有一个单一的共享内存一样)。如果没有并发请求，你可能会实现如下的链表 push 操作：

```

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

```

如果单独执行，这个实现是正确的。但是，如果多个副本同时执行，代码就不正确。如果两个 CPU 同时执行 push，那么两个 CPU 可能都会执行图 6.1 所示的第 15 行，然后其中一个才执行第 16 行，这就会产生一个不正确的结果，如图 6.2 所示。这样就会出现两个 list 元素，将 next 设为 list 的前值。当对 list 的两次赋值发生在第 16 行时，第二次赋值将覆盖第一次赋值；第一次赋值中涉及的元素将丢失。

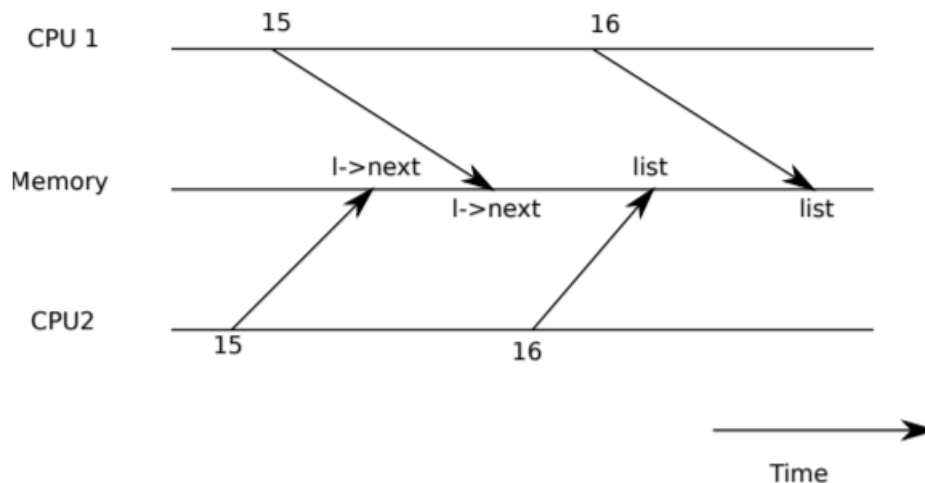


Figure 6.2: Example race

第 16 行的丢失更新是**竞争条件(race condition)**的一个例子。竞争条件是指同时访问一个内存位置，并且至少有一次访问是写的情况。竞争通常是一个错误的标志，要么是丢失

更新（如果访问是写），要么是读取一个不完全更新的数据结构。竞争的结果取决于所涉及的两个 CPU 的确切时间，以及它们的内存操作如何被内存系统排序，这可能会使竞争引起的错误难以重现和调试。例如，在调试 **push** 时加入 **print** 语句可能会改变执行的时机，足以使竞争消失。

避免竞争的通常方法是使用锁。锁确保了相互排斥，因此一次只能有一个 CPU 执行 **push** 的哪一行；这就使得上面的情况不可能发生。上面代码的正确 **lock** 版本只增加了几行代码。

```
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     acquire(&listlock);
16     l->next = list;
17     list = l;
18     release(&listlock);
19 }
```

acquire 和 **release** 之间的指令序列通常被称为临界区。这里的锁保护 **list**。

当我们说锁保护数据时，我们真正的意思是锁保护了一些适用于数据的 **不变式 (invariant)** 集合。invariant 是数据结构的属性，这些属性在不同的操作中都得到了维护。通常情况下，一个操作的正确行为取决于操作开始时的 invariant 是否为真。操作可能会暂时违反 invariant，但必须在结束前重新建立 invariant。例如，在链表的情况下，invariant 是 **list** 指向列表中的第一个元素，并且每个元素的下一个字段指向下一个元素。**push** 的实现暂时违反了 invariant：在第 17 行，**l** 指向下一个 **list** 元素，但 **list** 还没有指向 **l**（在第 18 行重新建立）。我们上面所研究的竞争条件之所以发生，是因为第二个 CPU 执行了依赖于链表 invariant 的代码，而它们被（暂时）违反了。正确地使用锁可以保证一次只能有一个 CPU 对关键部分的数据结构进行操作，所以当数据结构的 invariant 不成立时，没有 CPU 会执行数据结构操作。

你可以把锁看成是把并发的临界区 **串行化(serializing)** 的一种工具，使它们同时只运行一个，从而保护 invariant（假设临界区是独立的）。你也可以认为由同一个锁保护的临界区，相互之间是原子的，这样每个临界区都只能看到来自之前临界区的完整变化，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**，锁会将两个调用串行化，我们在不同的 CPU 上运行它们不会获得任何好处。我们说，如果多个进程同时想要同一个锁，就会发生冲突，或者说锁经历了争夺。内核

设计的一个主要挑战是避免锁的争用。Xv6 在这方面做得很少，但是复杂的内核会专门组织数据结构和算法来避免锁争用。例如链表，一个内核可以为每个 CPU 维护一个空闲页链表，只有当自己的链表为空时，并且它必须从另一个 CPU 获取内存时，才会接触另一个 CPU 的空闲链表。其他用例可能需要更复杂的设计。

锁的位置对性能也很重要。例如，在 `push` 中把 `acquire` 移到较早的位置是正确的：把 `acquire` 的调用移到第 13 行之前是可以的。这可能会降低性能，因为这样对 `malloc` 的调用也会被锁住。下面的 "Using locks" 一节提供了一些关于在哪里插入 `acquire` 和 `release` 调用的指南。

6.2 Code: Locks

Xv6 有两种类型的锁：自旋锁和睡眠锁。我们先说说自旋锁。Xv6 将自旋锁表示为一个结构体 `spinlock` (`kernel/spinlock.h:2`)。该结构中重要的字段是 `locked`，当锁可获得时，`locked` 为零，当锁被持有时，`locked` 为非零。从逻辑上讲，xv6 获取锁的代码类似于：

```
21 void
22 acquire(struct spinlock *lk) // does not work!
23 {
24     for (;;) {
25         if (lk->locked == 0) {
26             lk->locked = 1;
27             break;
28         }
29     }
30 }
```

不幸的是，这种实现并不能保证多处理器上的相互排斥。可能会出现这样的情况：两个 CPU 同时到达 `if` 语句，看到 `lk->locked` 为零，然后都通过设置 `lk->locked = 1` 来抢夺锁。此时，两个不同的 CPU 持有锁，这就违反了互斥属性。我们需要的是让第 25 行和第 26 行作为一个原子（即不可分割）步骤来执行。

由于锁被广泛使用，多核处理器通常提供了一些原子版的指令。在 RISC-V 上，这条指令是 `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，并将其读取的值放入 `r` 中，也就是说，它将寄存器的内容和内存地址进行了交换。它原子地执行这个序列，使用特殊的硬件来防止任何其他 CPU 使用读和写之间的内存地址。

Xv6 的 `acquire` (`kernel/spinlock.c:22`) 使用了可移植的 C 库调用 `__sync_lock_test_and_set`，它本质上为 `amoswap` 指令；返回值是 `lk->locked` 的旧（交换）内容。`acquire` 函数循环交换，重试（旋转）直到获取了锁。每一次迭代都会将 1 交换到 `lk->locked` 中，并检查之前的值；如果之前的值为 0，那么我们已经获得了锁，并且交换将 `lk->locked` 设置为 1。如果之前的值是 1，那么其他 CPU 持有该锁，而我们原子地将 1 换成 `lk->locked` 并没有改变它的值。

一旦锁被获取，`acquire` 就会记录获取该锁的 CPU，这方便调试。`lk->cpu` 字段受到锁的保护，只有在持有锁的时候才能改变。

函数 `release` (`kernel/spinlock.c:47`) 与 `acquire` 相反：它清除 `lk->cpu` 字段，然后释放锁。从概念上讲，释放只需要给 `lk->locked` 赋值为 0。C 标准允许编译器用多条存储指令来实现赋值，所以 C 赋值对于并发代码来说可能是非原子性的。相反，`release` 使用 C 库函数 `__sync_lock_release` 执行原子赋值。这个函数也是使用了 RISC-V 的 `amoswap` 指令。

6.3 Code: Using locks

xv6 在很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`) 就是一个很好的例子。试试练习 1 和 2，看看如果这些函数省略了锁会发生什么。你可能会发现很难触发不正确的行为，这说明很难可靠地测试代码是否没有锁定错误和竞争。xv6 也会有一些竞争。

使用锁的一个难点是决定使用多少个锁，以及每个锁应该保护哪些数据和 invariant。有几个基本原则。首先，任何时候，当一个 CPU 在另一个 CPU 读写数据的同时，写入变量，都应该使用锁来防止这两个操作重叠。其次，记住锁保护的是 invariant：如果一个 invariant 涉及到多个内存位置，通常需要用锁保护所有的位置，以确保 invariant 得到维护。

上面的规则说了什么时候需要锁，但没说了什么时候不需要锁，为了效率，不要太多锁，因为锁会降低并行性。如果并行性不重要，那么可以只安排一个线程，而不用担心锁的问题。一个简单的内核可以在多处理器上像这样做，通过一个单一的锁，这个锁必须在进入内核时获得，并在退出内核时释放（尽管系统调用，如管道读取或等待会带来一个问题）。许多单处理器操作系统已经被改造成使用这种方法在多处理器上运行，有时被称为“大内核锁”，但这种方法牺牲了并行性：内核中一次只能执行一个 CPU。如果内核做任何繁重的计算，那么使用一组更大的更精细的锁，这样内核可以同时多个 CPU 上执行，效率会更高。

作为粗粒度锁的一个例子，xv6 的 `kalloc.c` 分配器有一个单一的空闲页链表，由一个锁保护。如果不同 CPU 上的多个进程试图同时分配内存页，每个进程都必须通过在 `acquire` 中自旋来等待获取锁。自旋会降低性能，因为这是无意义的。如果对锁的争夺浪费了很大一部分 CPU 时间，也许可以通过改变分配器的设计来提高性能，使其拥有多个空闲页链表，每个链表都有自己的锁，从而实现真正的并行分配。

作为细粒度锁的一个例子，xv6 对每个文件都有一个单独的锁，这样操作不同文件的进程往往可以不等待对方的锁就可以进行。如果想让进程同时写入同一文件的不同区域，文件锁方案可以做得更细。最后，锁粒度的决定需要考虑性能以及复杂性。

在后面的章节解释 xv6 的每个部分时，会提到 xv6 使用锁来处理并发性的例子。作为预览，图 6.3 列出了 xv6 中所有的锁。

6.4 Deadlock and lock ordering

如果一个穿过内核的代码路径必须同时持有多个锁，那么所有的代码路径以相同的顺序获取这些锁是很重要的。如果他们不这样做，就会有死锁的风险。假设线程 T1 执行代码 `path1` 并获取锁 A，线程 T2 执行代码 `path2` 并获取锁 B，接下来 T1 会尝试获取锁 B，T2 会尝试获取锁 A，这两次获取都会无限期地阻塞，因为在这两种情况下，另一个线程都持有所需的锁，并且不会释放它，直到它的获取返回。为了避免这样的死锁，所有的代码路径必须以相同的顺序获取锁。对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：调用者调

用函数的方式必须使锁按照约定的顺序被获取。

由于 **sleep** 的工作方式（见第 7 章），Xv6 有许多长度为 2 的锁序链，涉及到进程锁（**struct proc** 中的锁）。例如，**consoleintr(kernel/console.c:138)** 是处理类型化字符的中断 routine。当一个新数据到达时，任何正在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**，以获取进程锁来唤醒它。因此，全局避免死锁的锁顺序包括了 **cons.lock** 必须在任何进程锁之前获取的规则。文件系统代码包含 xv6 最长的锁链。例如，创建一个文件需要同时持有目录的锁、新文件的 inode 的锁、磁盘块缓冲区的锁、磁盘驱动器的 **vdisk_lock** 和调用进程的 **p->lock**。为了避免死锁，文件系统代码总是按照上一句提到的顺序获取锁。

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids intermixed output
ftable.lock	Serializes allocation of a struct file in file table
icache.lock	Protects allocation of inode cache entries
vdisk_lock	Serializes access to disk hardware and queue of DMA descriptors
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

遵守全局避免死锁的顺序可能会非常困难。有时锁的顺序与逻辑程序结构相冲突，例如，也许代码模块 M1 调用模块 M2，但锁的顺序要求 M2 中的锁在 M1 中的锁之前被获取。有时锁的身份并不是事先知道的，也许是因为必须持有一个锁才能发现接下来要获取的锁的身份。这种情况出现在文件系统中，因为它在路径名中查找连续的组件，也出现在 **wait** 和 **exit** 的代码中，因为它们搜索进程表寻找子进程。最后，死锁的危险往往制约着人们对锁方案的细化程度，因为更多的锁往往意味着更多的死锁机会。避免死锁是内核实现的重要需求。

6.5 Locks and interrupt handlers

一些 xv6 自旋锁保护的数据会被线程和中断处理程序两者使用。例如，**clockintr** 定时器中断处理程序可能会在内核线程读取 **sys_sleep(kernel/sysproc.c:64)** 中的 **ticks** 的同时，递增 **ticks** (**kernel/trap.c:163**)。锁 **tickslock** 将保护两次临界区。

自旋锁和中断的相互作用带来了一个潜在的危险。假设 **sys_sleep** 持有 **tickslock**，而它的 CPU 被一个定时器中断。**clockintr** 会尝试获取 **tickslock**，看到它被持有，并等待它被释放。在这种情况下，**tickslock** 永远不会被释放：只有 **sys_sleep** 可以释放它，但 **sys_sleep** 不会继续运行，直到 **clockintr** 返回。所以 CPU 会死锁，任何需要其他锁的代码也会冻结。

为了避免这种情况，如果一个中断处理程序使用了自旋锁，CPU 决不能在启用中断的情况下持有该锁。Xv6 比较保守：当一个 CPU 获取任何锁时，xv6 总是禁用该 CPU 上的中断。中断仍然可能发生在其他 CPU 上，所以一个中断程序获取锁会等待一个线程释放自旋锁；它们不在同一个 CPU 上。

xv6 在 CPU 没有持有自旋锁时重新启用中断；它必须做一点记录来应对嵌套的临界区。**acquire** 调用 **push_off(kernel/spinlock.c:89)** 和 **release** 调用 **pop_off(kernel/spinlock.c:100)** 来跟踪当前 CPU 上锁的嵌套级别。当该计数达到零时，**pop_off** 会恢复最外层临界区开始时的中断启用状态。**intr_off** 和 **intr_on** 函数分别执行 RISC-V 指令来禁用和启用中断。

在设置 **lk->locked** 之前，严格调用 **push_off** 是很重要的(kernel/spinlock.c:28)。如果两者反过来，那么在启用中断的情况下，锁会有一个窗口（未锁到的位置），在未禁止中断时持有锁，不幸的是，一个定时的中断会使系统死锁。同样，释放锁后才调用 **pop_off** 也很重要（kernel/spinlock.c:66）。

6.6 Instruction and memory ordering

人们很自然地认为程序是按照源代码语句出现的顺序来执行的。然而，许多编译器和 CPU 为了获得更高的性能，会不按顺序执行代码。如果一条指令需要很多周期才能完成，CPU 可能会提前发出该指令，以便与其他指令重叠，避免 CPU 停顿。例如，CPU 可能会注意到在一个串行序列中，指令 A 和 B 互不依赖。CPU 可能先启动指令 B，这是因为它的输入在 A 的输入之前已经准备好了，或者是为了使 A 和 B 的执行重叠。编译器可以执行类似的重新排序，在一条语句的指令之前发出另一条语句的指令，由于它们原来的顺序。

编译器和 CPU 在 re-order 时遵循相应规则，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 re-order，从而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 ordering 规则称为内存模型。

例如，在这段 push 的代码中，如果编译器或 CPU 将第 4 行对应的存储移到第 6 行释放后的某个点，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);
```

如果发生这样的 re-order，就会有一个窗口，在这个窗口中，另一个 CPU 可以获取锁并观察更新的链表，但看到的是一个未初始化的 **list->next**。

为了告诉硬件和编译器不要执行这样的 re-ordering，xv6 在获取(kernel/spinlock.c:22)和释放(kernel/spinlock.c:47)中都使用了 **__sync_synchronize()**。**__sync_synchronize()** 是一个**内存屏障(memory barrier)**：它告诉编译器和 CPU 不要在屏障上 re-order 加载或存储。xv6 中的屏障几乎在所有重要的情况下都会 **acquire** 和 **release** 强制顺序，因为 xv6 在访问共享数据的周围使用锁。第 9 章讨论了一些例外情况。

6.7 Sleep locks

有时 xv6 需要长时间保持一个锁。例如，文件系统（第 8 章）在磁盘上读写文件内容时，会保持一个文件的锁定，这些磁盘操作可能需要几十毫秒。如果另一个进程想获取一个自旋锁，那么保持那么长的时间会导致浪费，因为获取进程在自旋的同时会浪费 CPU 很长时间。自旋锁的另一个缺点是，一个进程在保留自旋锁的同时不能让出 CPU；我们希望做到这一点，这样其他进程可以在拥有锁的进程等待磁盘的时候使用 CPU。在保留自旋锁的同时让出是非法的，因为如果第二个线程试图获取自旋锁，可能会导致死锁；因为获取自旋锁不会让出 CPU，第二个线程的自旋可能会阻止第一个线程运行并释放锁。在持有锁的同时让出也会违反在持有自旋锁时中断必须关闭的要求。因此，我们希望有一种锁，在等待获取的过程中产生 CPU，并在锁被持有时允许让出 CPU（和中断）。

Xv6 以睡眠锁(*sleep-locks*)的形式提供了这样的锁。`acquiresleep(kernel/sleeplock.c:22)`在等待的过程中让出 CPU，使用的技术将在第 7 章解释。在高层次上，**sleep-lock** 有一个由 **spinlock** 保护的锁定字段，而 `acquiresleep` 对 `sleep` 的调用会原子性地让出 CPU 并释放 **spinlock**。其结果是，在 `acquiresleep` 等待时，其他线程可以执行。

Xv6 以睡眠锁的形式提供了这样的锁。`acquiresleep(kernel/sleeplock.c:22)`在等待的过程中让出 CPU，使用的技术将在第 7 章解释。在高层次上，睡眠锁有一个由 **spinlock** 保护的 **locked** 字段，而 `acquiresleep` 对 `sleep` 的调用会原子性地让出 CPU 并释放 **spinlock**。其结果是，在 `acquiresleep` 等待时，其他线程可以执行。

因为睡眠锁会使中断处于启用状态，所以不能在中断处理程序中使用睡眠锁。因为 `acquiresleep` 可能会让出 CPU，所以睡眠锁不能在 **spinlock** 临界区内使用（虽然 **spinlocks** 可以在睡眠锁临界区内使用）。

自旋锁最适合短的临界区，因为等待它们会浪费 CPU 时间；睡眠锁对长时间的操作很有效。

6.8 Real world

尽管对并发基元和并行进行了多年的研究，但使用锁进行编程仍然具有挑战性。通常最好的方法是将锁隐藏在更高级别的构造中，比如同步队列，但 xv6 没有这样做。如果您使用锁编程，明智的做法是使用一个可以识别竞争条件的工具，因为很容易错过一个需要锁的 invariant。

大多数操作系统都支持 POSIX 线程 (Pthreads)，它允许一个用户进程在不同的 CPU 上有多个线程并发运行。Pthreads 对用户级锁、屏障(barriers)等都有支持。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 应该可以在该 CPU 上运行。另一个例子，如果一个 pthread 改变了进程的地址空间（例如，映射或取消映射内存），内核必须安排同一进程中其他线程，在其他 CPU 更新它们的硬件页表以反映地址空间的变化。

在没有原子指令的情况下实现锁是可能的[8]，但成本很高，而且大多数操作系统都使用原子指令。

如果多个 CPU 试图在同一时间获取同一个锁，那么锁的代价会很高。如果一个 CPU 在它的本地缓存中缓存了一个锁，而另一个 CPU 必须获取该锁，那么更新持有该锁的缓存行的原子指令必须将该行从一个 CPU 的缓存中移到另一个 CPU 的缓存中，也许还会使该缓存行的任何其他副本无效。从另一个 CPU 的缓存中获取缓存行的成本可能比从本地缓存中获取行的成本高几个数量级。

为了避免与锁有关的消耗，许多操作系统都采用无锁的数据结构和算法[5, 10]。例如，可以实现像本章开头的链表，在链表搜索过程中不需要锁，插入一个项目时，只需要一条原子指令就可以在链表中。不过，无锁编程比有锁编程更复杂，例如，必须担心指令和内存的 re-order 问题。有锁编程已经很难了，所以 xv6 没有使用无锁编程来额外复杂性。

6.9 Exercises

1. 删去 **kalloc**(kernel/kalloc.c:69)中对 **acquire** 和 **release** 的调用。这似乎会给调用 **kalloc** 的内核代码带来问题；你觉得会发生什么？当你运行 xv6 时，和你想的一样吗？运行 **usertests** 的时候呢？如果你没有看到问题，为什么没有呢？看看你是否可以通过在 **kalloc** 的关键部分插入 dummy loops¹⁰来引发问题。
2. 假设你在 **kfree** 中注释了 lock（在恢复 **kalloc** 的 lock 之后）。现在可能出了什么问题？**kfree** 中缺少锁是否比 **kalloc** 中的危害小？
3. 如果两个 CPU 同时调用 **kalloc**，其中一个就要等待另一个，这对性能不利。修改 **kalloc.c**，使其具有更多的并行性，这样不同 CPU 对 **kalloc** 的同时调用就可以进行，而不需要等待对方。
4. 使用大多数操作系统都支持的 POSIX 线程编写一个并行程序。例如，实现一个并行哈希表，并测量 put/get 的数量是否随着核心数的增加而增加。
5. 在 xv6 中实现 Pthreads 的一个子集。即实现用户级线程库，使一个用户进程可以有 1 个以上的线程，并安排这些线程可以在不同的 CPU 上并行运行。提出一个设计，正确处理线程进行阻塞系统调用和改变其共享地址空间的问题。

¹⁰ 无限循环

Chapter 7 Scheduling

任何操作系统运行的进程数量都可能超过计算机的 CPU 数量，因此需要制定一个方案，在各进程之间分时共享 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复用到硬件 CPU 上，给每个进程提供它有自己的虚拟 CPU 的假象。本章解释 xv6 如何实现这种复用。

7.1 Multiplexing

xv6 通过在两种情况下将 CPU 从一个进程切换到另一个进程来实现复用。首先，xv6 的 **sleep** 和 **wakeup** 机制会进行切换，这发生在进程等待设备或管道 I/O，或等待子进程退出，或在 **sleep** 系统调用中等待。其次，xv6 周期性地强制切换，以应对长时间的计算进程。这种复用造成了每个进程都有自己的 CPU 的假象，就像 xv6 使用内存分配器和硬件页表造成每个进程都有自己的内存的假象一样。

实现复用会有一些挑战。首先，如何从一个进程切换到另一个进程？虽然上下文切换的想法很简单，但实现起来却很难。第二，如何对用户进程透明的强制切换？xv6 采用一般的方式，用定时器中断来驱动上下文切换。第三，许多 CPU 可能会在进程间并发切换，需要设计一个锁来避免竞争。第四，当进程退出时，必须释放进程的内存和其他资源，但它自己不能做到这一切，因为它不能释放自己的内核栈，同时又在使用内核栈。第五，多核机器的每个内核必须记住它正在执行的进程，这样系统调用就会修改相应进程的内核状态。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。需要注意一些竞争可能会使唤醒丢失。Xv6 试图尽可能简单地解决这些问题，但尽管如此，写出来代码还是很棘手。

7.2 Code: Context switching

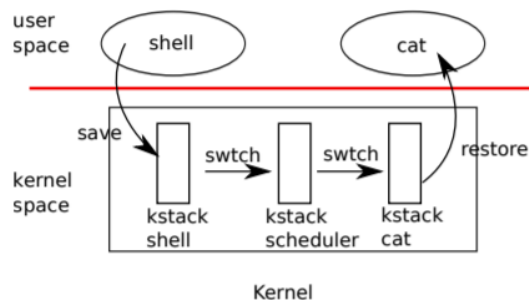


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

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，context 切换到当前 CPU 的调度器线程，context 切换到新进程的内核线程，以及 trap 返回到用户级进程。xv6 调度器在每个 CPU 上有一个专门的线程(保存的寄存器和栈)，因为调度器在旧进程的内核栈上执行是不安全的：其他核心

可能会唤醒进程并运行它,而且在两个不同的核心上使用相同的栈将是一场灾难。在本节中,我们将研究在内核线程和调度线程之间切换的机制。

从一个线程切换到另一个线程,需要保存旧线程的 CPU 寄存器,并恢复新线程之前保存的寄存器;栈指针和 pc 被保存和恢复,意味着 CPU 将切换栈和正在执行的代码。

函数 **swtch** 执行内核线程切换的保存和恢复。**swtch** 并不直接知道线程,它只是保存和恢复寄存器组,称为**上下文(context)**。当一个进程要放弃 CPU 的时候,进程的**内核线程**会调用 **swtch** 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结构体 **context(kernel/proc.h:2)**中,它本身包含在进程的结构体 **proc** 或 CPU 的结构体 **cpu** 中。**Swtch** 有两个参数: **struct context *old** 和 **struct context *new**。它将当前的寄存器保存在 old 中,从 new 中加载寄存器,然后返回。

让我们跟随一个进程通过 **swtch** 进入 **scheduler**。我们在第 4 章看到,在中断结束时,有一种情况是 **usertrap** 调用 **yield**。**yield** 又调用 **sched**,**sched** 调用 **swtch** 将当前上下文保存在 **p->context** 中,并切换到之前保存在 **cpu->scheduler** 中的调度器上下文 (**kernel/proc.c:509**)。

Swtch(kernel/swtch.S:3)只保存 callee-saved 寄存器,caller-saved 寄存器由调用的 C 代码保存在堆栈上(如果需要)。**Swtch** 知道 **struct context** 中每个寄存器字段的偏移量。它不保存 pc。相反,**swtch** 保存了 ra 寄存器¹¹,它保存了 **swtch** 应该返回的地址。现在,**swtch** 从新的上下文中恢复寄存器,新的上下文中保存着前一次 **swtch** 所保存的寄存器值。当 **swtch** 返回时,它返回到被恢复的 ra 寄存器所指向的指令,也就是新线程之前调用 **swtch** 的指令。此外,它还会在新线程的栈上返回。

在我们的例子中,**sched** 调用 **swtch** 切换到 **cpu->scheduler**,即 CPU 的调度上下文。这个上下文已经被调度器对 **swtch** 的调用所保存(**kernel/proc.c:475**)。当我们跟踪的 **swtch** 返回时,它不是返回到 **sched** 而是返回到 **scheduler**,它的栈指针指向当前 CPU 的调度器栈。

7.3 Code: Scheduling

上一节研究了 **swtch** 的低级细节,现在我们把 **swtch** 作为一个给定的条件,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。调度器以 CPU 特殊线程的形式存在,线程运行 **scheduler** 函数。这个函数负责选择下一步运行哪个进程。一个想要放弃 CPU 的进程,必须获取自己的进程锁 **p->lock**,释放它所持有的其他锁,更新自己的状态(**p->state**),然后调用 **sched**。**Yield** (**kernel/proc.c:515**)做到了这些,就像我们稍后要研究的 **sleep** 和 **exit** 一样。**Sched** 对这些条件进行仔细检查(**kernel/proc.c:499-504**),这些条件表明:因为锁被持有,所以中断应该被禁止。最后,**sched** 调用 **swtch** 保存 **p->context** 中的当前上下文,并在 **cpu->scheduler** 中切换到调度器上下文。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** 的过程中持有 **p->lock**: **swtch** 的调用者必须已经持有锁,而锁的控制权则传递给切换到的代码。这种约定对于锁来说是不寻常的;通常获得锁的线程也要负责释放锁。对于上下文切换来说,有必要打破这个约定,因为 **p->lock** 保护了进

¹¹ ra 寄存器: 函数 return 地址

程的状态和 `context` 字段上的**不变式(invariant)**，而这些不变式在 `swtch` 中执行时为 `false`。如果 `p->lock` 在 `swtch` 过程中不被持有，可能会出现问题的一个情况：在 `yield` 将其状态设置为 `RUNNABLE` 之后，但在 `swtch` 切换到新的栈之前，其他 CPU 可能会运行这个进程。结果就是两个 CPU 运行在同一个栈上，这是不正确的。

一个内核线程在 `sched` 中放弃它的 CPU，并且总是切换到调度器的同一个位置，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，如果把 xv6 切换线程的行号打印出来，就会观察到下面的结果：(kernel/proc.c:475)，(kernel/proc.c:509)，(kernel/proc.c:475)，(kernel/proc.c:509)，等等。在两个线程之间发生这种风格化切换的程序有时被称为**协程(coroutine)**；在这个例子中，`sched` 和调度器是彼此的 *coroutines*。

当一个新的进程第一次被调度时，它开始于 `forkret` (kernel/proc.c:527)。`Forkret` 的存在是为了释放 `p->lock`，否则，新进程需要从 `usertrapret` 开始执行。

`scheduler(kernel/proc.c:457)` 运行了一个简单的循环：找到一个可以运行进程，运行它，直到它让出 CPU，一直重复。调度器在进程表上循环寻找一个可运行的进程，即 `p->state == RUNNABLE` 的进程。一旦找到这样的进程，它就会设置 CPU 当前进程 `c->proc`，将该进程标记为 `RUNNING`，然后调用 `swtch` 开始运行它(kernel/proc.c:470- 475)。

你可以这样理解调度代码结构，它执行一组关于进程的不变式，并且每当这些不变式为 `False` 时，就持有 `p->lock`。一个不变式是，如果一个进程正在运行，定时器中断的 `yield` 必须能够安全地切换进程；这意味着 CPU 寄存器必须持有进程的寄存器值（即 `swtch` 没有将它们移到上下文中），并且 `c->proc` 必须指向该进程。另一个不变式是，如果一个进程是 `RUNNABLE` 的，那么对于一个空闲的 CPU 调度器来说，运行它必须是安全的；这意味着

- (1) `p->context` 必须拥有进程的寄存器（i.e., 它们实际上并不在真实的寄存器中），
- (2) 没有 CPU 在进程的内核栈上执行，
- (3) 也没有 CPU 的 `c->proc` 指向进程。

请注意，当 `p->lock` 被持有时，这些属性往往不为真。

维护上述不变式的原因：xv6 经常在一个线程中获取 `p->lock`，然后在另一个线程中释放，例如在 `yield` 中获取，在 `schedululer` 中释放。一旦 `yield` 开始修改一个正在运行的进程的状态，使其成为 `RUNNABLE`，锁必须一直保持，直到不变式被恢复：最早正确的释放点是在调度器（运行在自己的堆栈上）清除 `c->proc` 之后。同样，一旦调度器开始将一个 `RUNNABLE` 进程转换为 `RUNNING`，锁就不能被释放，直到内核线程完成运行（在 `swtch` 之后，例如在 `yield` 中）。

`p->lock` 也保护其他的东西：`exit` 和 `wait` 之间的相互作用，避免丢失唤醒的机制（见第 7.5 节），以及避免进程退出和其他进程读写其状态之间的竞争（例如，`exit` 系统调用查看 `p->pid` 并设置 `p->killed` (kernel/proc.c:611)。值得思考的是，是否可以将 `p->lock` 的不同功能拆分开来，这样既清晰，也可能提高性能。

7.4 Code: mycpu and myproc

Xv6 经常需要一个指向当前进程 `proc` 的指针。在单核处理器上，可以用一个全局变量指向当前的 `proc`。这在多核机器上是行不通的，因为每个核都执行不同的进程。解决这个问题的方法是利用每个核都有自己的一组寄存器的事实；我们可以使用其中的一个寄存器来帮助查找每个核的信息。

Xv6 为每个 CPU 维护了一个 `cpu` 结构体(kernel/proc.h:22)，它记录了当前在该 CPU 上

运行的进程(如果有的话), 为 CPU 的调度线程保存的寄存器, 以及管理中断禁用所需的嵌套自旋锁的计数。函数 `mycpu(kernel/proc.c:60)` 返回一个指向当前 CPU 结构体 `cpu` 的指针。RISC-V 对其 CPU 进行编号, 给每个 CPU 一个 `hartid`。Xv6 确保每个 CPU 的 `hartid` 在内核中被存储在该 CPU 的 `tp` 寄存器中。这使得 `mycpu` 可以使用 `tp` 对 `cpu` 结构体的数组进行索引, 从而找到正确的 `cpu`。

确保一个 CPU 的 `tp` 始终保持 CPU 的 `hartid` 是有一点复杂的。`mstart` 在 CPU 启动的早期设置 `tp` 寄存器, 此时 CPU 处于机器模式(`kernel/start.c:46`)。`Usertrapret` 将 `tp` 寄存器保存在 `trampoline` 页中, 因为用户进程可能会修改 `tp` 寄存器。最后, 当从用户空间进入内核时, `uservec` 会恢复保存的 `tp`(`kernel/trampoline.S:70`)。编译器保证永远不使用 `tp` 寄存器。如果 RISC-V 允许 `xv6` 直接读取当前的 `hartid` 会更方便, 但机器模式只有机器模式下能够读取, 不允许在监督者模式下使用。

`cpu` 和 `mycpu` 的返回值很容易错: 如果定时器中断, 导致线程让出 CPU, 然后转移到不同的 CPU 上, 之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, 只有在使用完返回的 `cpu` 结构后才启用中断。

`myproc(kernel/proc.c:68)` 函数返回当前 CPU 上运行的进程的 `proc` 指针。`myproc` 禁用中断, 调用 `mycpu`, 从 `cpu` 中获取当前进程指针(`c->proc`), 然后启用中断。即使启用了中断, `myproc` 的返回值也可以安全使用: 如果定时器中断将调用进程移到了不同的 CPU 上, 它的 `proc` 结构指针将保持不变。

7.5 Sleep and wakeup

调度和锁有助于让一个进程对另一个进程的不可见, 但到目前为止, 我们还没有任何抽象来帮助进程进行交互。人们发明了许多机制来解决这个问题。Xv6 使用了一种叫做睡眠和唤醒的机制, 它允许一个进程睡眠并等待事件, 而另一个进程在事件发生后将其唤醒。睡眠和唤醒通常被称为 *序列协调(sequence coordination)* 或 *条件同步(conditional synchronization)* 机制。

为了说明这一点, 让我们考虑一个叫做 *信号量(semaphore)*[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, 然后将其递减并返回。如果只有一个生产者线程和一个消费者线程, 而且它们在不同的 CPU 上执行, 编译器也没有太过激进的优化, 那么这个实现是正确的。

```
struct semaphore
{
    struct spinlock lock;
    int count;
};

void V(struct semaphore *s)
{
    acquire(&s->lock);
    s->count += 1;
    release(&s->lock);
}
```

```

void P(struct semaphore *s)
{
    while (s->count == 0)
        ;
    acquire(&s->lock);
    s->count -= 1;
    release(&s->lock);
}

```

上面的实现是代价很大。如果生产者很少生产，消费者将把大部分时间花在 while 循环中，希望得到一个非零的计数。消费者的 CPU 可以通过反复 **轮询(polling)** `s->count` 来找到比 **忙碌等待(busy waiting)** 更有效的工作。避免 **忙碌等待** 需要一种方法，让消费者让出 CPU，只有在 V 增加计数后才恢复。

这里是朝着这个方向迈出的一步，虽然他不能完全解决这个问题。让我们想象一对调用，`sleep` 和 `wakeup`，其工作原理如下。`Sleep(chan)` 睡眠 `chan` 上，`chan` 可以为任意值，称为 **等待通道(wait chan)**。`Sleep` 使调用进程进入睡眠状态，释放 CPU 进行其他工作。`Wakeup(chan)` 唤醒所有在 `chan` 上 `sleep` 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，则 `wakeup` 不做任何事情。我们更改信号量实现，以使用 `sleep` 和 `wakeup`。

```

void V(struct semaphore *s)
{
    acquire(&s->lock);
    s->count += 1;
    wakeup(s);
    release(&s->lock);
}

void P(struct semaphore *s)
{
    while (s->count == 0)
        sleep(s);
    acquire(&s->lock);
    s->count -= 1;
    release(&s->lock);
}

```

P 现在放弃 CPU 而不是自旋，这是一个不错的改进。然而，事实证明，像这样设计 `sleep` 和 `wakeup` 并不明确，因为存在可能丢失唤醒的问题。假设执行 P 的 `s->count == 0` 这一行时。当 P 在 `sleep` 之前，V 在另一个 CPU 上运行：它将 `s->count` 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 继续执行：它调用 `sleep` 并进入睡眠状态。这就造成了一个问题：P 正在 `sleep`，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，否则消费者将永远等待，即使计数是非零。

这个问题的根源在于，P 只有在 `s->count == 0` 时才会 `sleep` 的不变式被违反，当 V 运

行在错误的时刻时。保护这个不变式的一个不正确的方法是移动 P 中的锁的获取，这样它对计数的检查和对 sleep 的调用是原子的：

```
void V(struct semaphore *s)
{
    acquire(&s->lock);
    s->count += 1;
    wakeup(s);
    release(&s->lock);
}

void P(struct semaphore *s)
{
    acquire(&s->lock);
    while (s->count == 0)
        sleep(s);
    s->count -= 1;
    release(&s->lock);
}
```

人们可能希望这个版本的 P 能够避免丢失的唤醒，因为锁会阻止 V 在 `s->count == 0` 和 sleep 之间执行。它做到了这一点，但它也会死锁。P 在 **sleep** 时保持着锁，所以 V 将永远阻塞在等待锁的过程中。

我们将通过改变 **sleep** 的接口来修正前面的方案：调用者必须将**条件锁(condition lock)**传递给 **sleep**，这样在调用进程被标记为 **SLEEPING** 并在 chan 上等待后，它就可以释放锁。锁将强制并发的 V 等待到 P 完成使自己进入 **SLEEPING** 状态，这样 **wakeup** 就会发现 **SLEEPING** 的消费者并将其唤醒。一旦消费者再次被唤醒，**sleep** 就会重新获得锁，然后再返回。我们新的正确的睡眠/唤醒方案可以使用如下(**sleep** 函数改变了)。

```
void V(struct semaphore *s)
{
    acquire(&s->lock);
    s->count += 1;
    wakeup(s);
    release(&s->lock);
}

void P(struct semaphore *s)
{
    acquire(&s->lock);
    while (s->count == 0)
        sleep(s, &s->lock);
    s->count -= 1;
    release(&s->lock);
}
```

P 持有 `s->lock` 会阻止了 V 在 P 检查 `c->count` 和调用 **sleep** 之间试图唤醒它。但是，

请注意，我们需要 `sleep` 来原子地释放 `s->lock` 并使消费者进程进入 *SLEEPING* 状态。

7.6 Code: Sleep and wakeup

让我们看看 `sleep` (`kernel/proc.c:548`) 和 `wakeup` (`kernel/proc.c:582`) 的实现。其基本思想是让 `sleep` 将当前进程标记为 *SLEEPING*，然后调用 `sched` 让出 CPU；`wakeup` 则寻找给定的 *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方便的数字作为 `chan`。Xv6 经常使用参与等待的内核数据结构的地址。

`Sleep` 首先获取 `p->lock` (`kernel/proc.c:559`)。现在进入睡眠状态的进程同时持有 `p->lock` 和 `lk`。在调用者(在本例中为 `P`)中，持有 `lk` 是必要的：它保证了没有其他进程(在本例中，运行 `V` 的进程)可以调用 `wakeup(chan)`。现在 `sleep` 持有 `p->lock`，释放 `lk` 是安全的：其他进程可能会调用 `wakeup(chan)`，但 `wakeup` 会等待获得 `p->lock`，因此会等到 `sleep` 将进程状态设置为 *SLEEPING*，使 `wakeup` 不会错过 `sleep` 的进程。

有一个复杂情况：如果 `lk` 和 `p->lock` 是同一个锁，如果 `sleep` 仍试图获取 `p->lock`，就会和自己死锁。但是如果调用 `sleep` 的进程已经持有 `p->lock`，那么它就不需要再做任何事情来避免错过一个并发的 `wakeup`。这样的情况发生在，`wait` (`kernel/proc.c:582`)调用 `sleep` 并持有 `p->lock` 时。

现在 `sleep` 持有 `p->lock`，而没有其他的锁，它可以通过记录它睡眠的 `chan`，将进程状态设置 *SLEEPING*，并调用 `sched`(`kernel/proc.c:564-567`)来使进程进入睡眠状态。稍后我们就会明白为什么在进程被标记为 *SLEEPING* 之前，`p->lock` 不会被释放（由调度器）。

在某些时候，一个进程将获取条件锁，设置睡眠等待的条件，并调用 `wakeup(chan)`。重要的是，`wakeup` 是在持有条件锁¹²的情况下被调用的。`Wakeup` 循环浏览进程表 (`kernel/proc.c:582`)。它获取每个被检查的进程的 `p->lock`，因为它可能会修改该进程的状态，也因为 `p->sleep` 确保 `sleep` 和 `wakeup` 不会相互错过。当 `wakeup` 发现一个进程处于状态为 *SLEEPING* 并有一个匹配的 `chan` 时，它就会将该进程的状态改为 *RUNNABLE*。下一次调度器运行时，就会看到这个进程已经准备好运行了。

为什么 `sleep` 和 `wakeup` 的锁规则能保证睡眠的进程不会错过 `wakeup`？`sleep` 进程从检查条件之前到标记为 *SLEEPING* 之后的这段时间里，持有条件锁或它自己的 `p->lock` 或两者都持有。调用 `wakeup` 的进程在 `wakeup` 的循环中持有这两个锁。因此，唤醒者要么在消费者检查条件之前使条件为真；要么唤醒者的 `wakeup` 在消费者被标记为 *SLEEPING* 之后检查它。无论怎样，`wakeup` 就会看到这个睡眠的进程，并将其唤醒（除非有其他事情先将其唤醒）。

有时会出现多个进程在同一个 `chan` 上睡眠的情况；例如，有多个进程从管道中读取数据。调用一次 `wakeup` 就会把它们全部唤醒。其中一个进程将首先运行，并获得 `sleep` 参数传递的锁，（就管道而言）读取数据都会在管道中等待。其他进程会发现，尽管被唤醒了，但没有数据可读。从他们的角度来看，唤醒是“虚假的”，他们必须再次睡眠。出于这个原因，`sleep` 总是在一个检查条件的循环中被调用。

如果两次使用 `sleep/wakeup` 不小心选择了同一个通道，也不会有害：它们会看到虚假的唤醒，上面提到的循环允许发生这种情况。`sleep/wakeup` 的魅力很大程度上在于它既是

¹² 严格地说，只是在 `acquire` 之后 `wakeup` 就足够了(也就是说，可以在 `release` 之后调用 `wakeup`)。

轻量级的（不需要创建特殊的数据结构来充当睡眠通道），又提供了一层间接性（调用者不需要知道他们正在与哪个具体的进程交互）。

7.7 Code: Pipes

一个使用 **sleep** 和 **wakeup** 来同步生产者和消费者的更复杂的例子是 xv6 的管道实现。我们在第 1 章看到了管道的接口：写入管道一端的字节被复制到内核缓冲区，然后可以从管道的另一端读取。未来的章节将研究管道如何支持文件描述符，但我们现在来看一下 **pipewrite** 和 **piperead** 的实现吧。

每个管道由一个结构体 **pipe** 表示，它包含一个锁和一个数据缓冲区。**nread** 和 **nwrite** 两个字段统计从缓冲区读取和写入的字节总数。缓冲区呈环形：**buf[PIPESIZE-1]**之后写入的下一个字节是 **buf[0]**。计数不呈环形。这个约定使得实现可以区分满缓冲区(**nwrite == nread+PIPESIZE**)和空缓冲区(**nwrite == nread**)，但这意味着对缓冲区的索引必须使用 **buf[nread % PIPESIZE]**，而不是使用 **buf[nread]**(**nwrite** 也是如此)。

假设对 **piperead** 和 **pipewrite** 的调用同时发生在两个不同的 CPU 上。**Pipewrite** (kernel/pipe.c:77)首先获取管道的锁，它保护了计数、数据和相关的不变式。然后，**Piperead** (kernel/pipe.c:103)也试图获取这个锁，但是不会获取成功。它在 **acquire**(kernel/spinlock.c:22)中循环，等待锁的到来。当 **piperead** 等待时，**pipewrite** 会循环写，依次将每个字节添加到管道中(kernel/pipe.c:95)。在这个循环中，可能会发生缓冲区被填满的情况(kernel/pipe.c:85)。在这种情况下，**pipewrite** 调用 **wakeup** 来提醒所有睡眠中的 reader 有数据在缓冲区中等待，然后在 **&pi->nwrite** 上 **sleep**，等待 reader 从缓冲区中取出一些字节。**Sleep** 函数内会释放 **pi->lock**，然后 **pipewrite** 进程睡眠。

现在 **pi->lock** 可用了，**piperead** 设法获取它并进入它的临界区：它发现 **pi->nread != pi->nwrite** (kernel/pipe.c:110) (**pipewrite** 进入睡眠状态是由于 **pi->nwrite == pi->nread+PIPESIZE** (kernel/pipe.c:85))，所以它进入 for 循环，将数据从管道中复制出来(kernel/pipe.c:117)，并按复制的字节数增加 **nread**。现在又可写了，所以 **piperead** 在返回之前调用 **wakeup** (kernel/pipe.c:124) 来唤醒在睡眠的 writer。**Wakeup** 找到一个在 **&pi->nwrite** 上睡眠的进程，这个进程正在运行 **pipewrite**，但在缓冲区填满时停止了。它将该进程标记为 **RUNNABLE**。

管道代码对 reader 和 writer 使用单独的睡眠 **chan** (**pi->nread** 和 **pi->nwrite**)；这可能会使系统在有多个 reader 和 writer 等待同一个管道的情况下更有效率。管道代码在循环内 **sleep**，检查 **sleep** 条件；如果有多个 reader 和 writer，除了第一个被唤醒的进程外，其他进程都会看到条件仍然是假的，然后再次睡眠。

7.8 Code: Wait, exit, and kill

sleep 和 **wakeup** 可以用于许多种需要等待的情况。在第 1 章中介绍的一个有趣的例子是，一个子进程的 **exit** 和其父进程的 **wait** 之间的交互。在子进程退出的时候，父进程可能已经在 **wait** 中睡眠了，也可能在做别的事情；在后一种情况下，后续的 **wait** 调用必须观察子进程的退出，也许是在它调用 **exit** 之后很久。xv6 在 **wait** 观察到子进程退出之前，记录子进程退出的方式是让 **exit** 将调用进程设置为 **ZOMBIE** 状态，在那里停留，直到父进程的

wait 注意到它，将子进程的状态改为 **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。如果父进程比子进程先退出，父进程就把子进程交给 **init** 进程，而 **init** 进程则循环的调用 **wait**；这样每个子进程都有一个“父进程”来清理。主要的实现挑战是父进程和子进程的 **wait** 和 **exit**，以及 **exit** 和 **exit** 之间可能出现竞争和死锁的情况。

Wait 使用调用进程的 **p->lock** 作为条件锁，以避免唤醒丢失，它在开始时获取该锁 (kernel/proc.c:398)。然后它扫描进程表。如果它发现一个处于 **ZOMBIE** 状态的子进程，它释放这个子进程的资源 and 它的 **proc** 结构，将子进程的退出状态复制到提供给 **wait** 的地址 (如果它不是 0)，并返回子进程的 ID。如果 **wait** 找到了子进程但没有一个退出，它调用 **sleep** 等待其中一个子进程退出(kernel/proc.c:445)，然后再次扫描。这里，在 **sleep** 中释放的条件锁是等待进程的 **p->lock**，也就是上面提到的特殊情况。请注意，**wait** 经常持有两个锁；它在试图获取任何子锁之前，会先获取自己的锁；因此 xv6 的所有锁都必须遵守相同的锁顺序 (父进程的锁，然后是子进程的锁)，以避免死锁。

Wait 会查看每个进程的 **np->parent** 来寻找它的子进程。它使用 **np->parent** 而不持有 **np->lock**，这违反了共享变量必须受锁保护的通常规则。但是 **np** 有可能是当前进程的祖先，在这种情况下，获取 **np->lock** 可能会导致死锁，因为这违反了上面提到的顺序。在这种情况下，在没有锁的情况下检查 **np->parent** 似乎是安全的；一个进程的父进程字段只有“父亲”改变，所以如果 **np->parent==p** 为真，除非当前进程改变它，否则该值就不会改变。

Exit (kernel/proc.c:333)记录退出状态，释放一些资源，将所有子进程交给 **init** 进程，在父进程处于等待状态时唤醒它，将调用进程标记为 **zombie**，并永久放弃 CPU。最后的序列有点棘手。退出的进程必须持有父进程的锁，同时将自己状态设置为 **ZOMBIE** 并唤醒父进程，因为父进程的锁是条件锁，可以防止在等待中丢失 **wakeup**。子进程也必须持有自己的 **p->lock**，否则父进程可能会看到它的状态为 **ZOMBIE**，并在它还在运行时释放它。锁的获取顺序对避免死锁很重要：因为 **wait** 在子锁之前获取父锁，所以 **exit** 必须使用相同的顺序。

Exit 调用了一个专门的唤醒函数 **wakeup1**，它只唤醒父函数，而且只有父进程在 **wait** 中睡眠的情况下才会去唤醒它(kernel/proc.c:598)。在将自己的状态设置为 **ZOMBIE** 之前，唤醒父进程可能看起来并不正确，但这是安全的：尽管 **wakeup1** 可能会导致父进程运行，但 **wait** 中的循环不能检查子进程，直到子进程的 **p->lock** 被调度器释放为止，所以 **wait** 不能查看退出的进程，直到 **exit** 将其状态设置为 **ZOMBIE** 之后(kernel/proc.c:386)。

exit 允许一个进程自行终止，而 **kill** (kernel/proc.c:611) 则允许一个进程请求另一个进程终止。如果让 **kill** 直接摧毁进程，那就太复杂了，因为相应进程可能在另一个 CPU 上执行，也许正处于更新内核数据结构的敏感序列中。因此，**kill** 的作用很小：它只是设置进程的 **p->killed**，如果它在 **sleep**，则 **wakeup** 它。最终，进程会进入或离开内核，这时如果 **p->killed** 被设置，**usertrap** 中的代码会调用 **exit**。如果进程在用户空间运行，它将很快通过进行系统调用或因为定时器（或其他设备）中断而进入内核。

如果进程处于睡眠状态，**kill** 调用 **wakeup** 会使进程从睡眠中返回。这是潜在的危险，因为正在等待的条件可能不为真。然而，xv6 对 **sleep** 的调用总是被包裹在一个 **while** 循环中，在 **sleep** 返回后重新检测条件。一些对 **sleep** 的调用也会在循环中检测 **p->killed**，如果设置了 **p->killed**，则离开当前活动。只有当这种离开是正确的时候才会这样做。例如，管道读写代码如果设置了 **killed** 标志就会返回；最终代码会返回到 **trap**，**trap** 会再次检查标

志并退出。

一些 xv6 `sleep` 循环没有检查 `p->killed`，因为代码处于多步骤系统调用的中间，而这个调用应该是原子的。`virtio` 驱动(`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系列写操作中的一个，而这些写操作都是为了让文件系统处于一个正确的状态而需要的。一个在等待磁盘 I/O 时被杀死的进程不会退出，直到它完成当前的系统调用和 `usertrap` 看到 `killed` 的标志。

7.9 Real world

xv6 调度器实现了一个简单的调度策略，它依次运行每个进程。这种策略被称为**轮询调度(round robin)**。真正的操作系统实现了更复杂的策略，例如，允许进程有优先级。这个策略是，一个可运行的高优先级进程将被调度器优先于一个可运行的低优先级进程。这些策略可能会很快变得复杂，因为经常有相互竞争的目标：例如，操作者可能还想保证公平性和高吞吐量。此外，复杂的策略可能会导致不尽人意的交互，如**优先级倒置(priority inversion)**和**护航现象(convoys)**。当低优先级和高优先级进程共享一个锁时，就会发生优先级倒置，当低优先级进程获得锁时，就会阻止高优先级进程的进展。当许多高优先级进程都在等待一个获得共享锁的低优先级进程时，就会形成一个长长的等待进程的车队；一旦护航现象形成，就会持续很长时间。为了避免这类问题，在复杂的调度器中需要额外的机制。

`sleep` 和 `wakeup` 是一种简单有效的同步方法，但还有很多其他的方法。在所有这些方法中，第一个挑战是避免我们在本章开头看到的丢失唤醒问题。最初的 Unix 内核的 `sleep` 只是禁用了中断，这已经足够了，因为 Unix 运行在单 CPU 系统上。因为 xv6 运行在多处理器上，所以它增加了一个显式的 `sleep` 锁。FreeBSD 的 `msleep` 采用了同样的方法。Plan 9 的 `sleep` 使用了一个回调函数，它在进入睡眠前保持调度锁的情况下运行；这个函数的作用是在最后一刻检查 `sleep` 情况，以避免 `wakeup` 丢失。Linux 内核的 `sleep` 使用一个显式的进程队列，称为等待队列，而不是等待通道；队列有自己的内部锁。

在 `wakeup` 过程中扫描整个进程链表，寻找相匹配的 `chan` 的进程，效率很低。一个更好的解决方案是用一个数据结构代替 `sleep` 和 `wakeup` 中的 `chan`，该结构上存放着 `sleep` 的进程列表，比如 Linux 的等待队列。Plan 9 的 `sleep` 和 `wakeup` 将该结构称为 rendezvous point 或 Rendez。许多线程库将同一个结构称为条件变量；在这种情况下，`sleep` 和 `wakeup` 的操作被称为 `wait` 和 `signal`。所有这些机制都有相同的机制：睡眠条件被睡眠过程中原子地释放的锁保护。

`wakeup` 唤醒了所有在某个特定 `chan` 上等待的进程，可能很多进程都在等待这个特定 `chan`。操作系统会调度所有这些进程，它们会争相检查睡眠条件。以这种方式行事的进程有时被称为**惊群效应(thundering herd)**，最好避免这种情况。大多数条件变量都有两个唤醒的基元：信号 (signal)，唤醒一个进程；广播 (broadcast)，唤醒所有等待的进程。

信号量通常用于同步。`count` 通常对应于类似于管道缓冲区中可用的字节数或一个进程拥有的僵尸子进程的数量。使用显式计数作为抽象的一部分，可以避免丢失 `wakeup` 的问题：有一个显式的计数，说明已经发生的唤醒次数。该计数还避免了“虚假的”唤醒和**惊群效应**问题。

终止进程和清理进程在 xv6 中引入了很多复杂性。在大多数操作系统中，它甚至更加复杂，因为，例如，被杀进程可能 `trap` 在内核中睡眠，而解除它的堆栈需要很多仔细的编程。

许多操作系统使用显式的异常处理机制来解除堆栈，比如 `longjmp`¹³。此外，还有其他一些事件可以导致一个睡眠进程被唤醒，即使它正在等待的事件还没有发生。例如，当一个 Unix 进程处于睡眠状态时，另一个进程可能会向它发送一个 `signal`。在这种情况下，该进程将从中断的系统调用中返回，返回值为 -1，错误代码设置为 `EINTR`。应用程序可以检查这些值并决定做什么。Xv6 不支持信号，也就不会出现这种复杂性。

Xv6 对 `kill` 的支持并不完全令人满意：有些 `sleep` 循环可能应该检查 `p->killed`。一个相关的问题是，即使是检查 `p->killed` 的 `sleep` 循环，在 `sleep` 和 `kill` 之间也会有一个竞争；`kill` 可能会设置 `p->killed`，并试图唤醒被杀进程，唤醒时刻发生在循环检查 `p->killed` 之后，但在它调用 `sleep` 之前，就会发生。如果这个问题发生了，被杀进程不会注意到 `p->killed`，直到它所等待的条件发生。这可能会晚很多（例如，当 `virtio` 驱动返回一个被杀进程正在等待的磁盘块时），也可能永远不会发生（例如，如果被杀进程正在等待来自控制台的输入，但用户没有键入任何输入）。

真正的操作系统会在常数时间内用显式的空闲列表来寻找空闲的进程，而不是在 `allocproc` 中进行线性时间的搜索；xv6 为了简单起见，使用了线性扫描的方式。

7.10 Exercises

1、Sleep 必须检查 `lk != & p->lock` 以避免死锁(kernel/proc.c:558-561)。假设一种特殊情况通过将

```
if(lk != &p->lock){
    acquire(&p->lock);
    release(lk);
}
```

替换为：

```
release(lk);
acquire(&p->lock);
```

这样会破坏 `sleep` 吗，怎样破坏。

2、大部分进程退出时，资源清理可以通过 `exit` 或 `wait` 来完成。事实证明，关闭打开的文件一定要在 `exit` 中进行。为什么？答案和管道有关。

3、在 xv6 中实现信号量而不使用 `sleep` 和 `wakeup`(但可以使用 `spin locks`)。在 xv6 中用信号量替换 `sleep` 和 `wakeup` 的使用。判断结果。

4、修正上面提到的 `kill` 和 `sleep` 之间的竞争，使得发生在被杀进程睡眠循环检查 `p->killed` 之后，在它调用 `sleep` 之前的 `kill` 会使得被杀进程放弃当前系统调用。

5、设计一个方案，让每一个睡眠循环都检查 `p->killed`，这样，在 `virtio` 驱动中的进程如果被其他进程杀死，就可以从 `while` 循环中快速返回。

6、修改 xv6，当从一个进程的内核线程切换到另一个进程时，只使用一次上下文切换，而不是先切换到调度线程，再切换到另一个进程。使用一次上下文切换，产生的线程需要自己选择下一个线程，并调用 `swtch`。面临的挑战将是如何防止多个内核意外执行同一个线程；如何正确地进行锁定；以及如何避免死锁。

¹³ 一种异常处理方式，可以去 google 下。

7、修改 xv6 的调度器，当没有进程可运行时，使用 RISC-V **WFI**（等待中断）指令。尽量保证只要有可运行的进程等待运行，就不会有核心通过 WFI 进行暂停。

8、锁 `p->lock` 保护了很多不变式，当看到某段被 **p->lock** 保护的 xv6 代码时，可能很难弄清楚保护了那个不变式。通过将 **p->lock** 拆分成几个锁，设计一个更简洁的方案。

Chapter 8 File system

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，以及支持持久性，以便数据在重启后仍然可用。

xv6 文件系统提供了类 Unix 的文件、目录和路径名（见第 1 章），并将其数据存储在 virtio 磁盘上以实现持久化（见第 4 章）。该文件系统解决了几个挑战：

- 文件系统需要磁盘上的数据结构来表示命名目录和文件的树，记录保存每个文件内容的块的身份，并记录磁盘上哪些区域是空闲的。
- 文件系统必须支持崩溃恢复。也就是说，如果发生崩溃（如电源故障），文件系统必须在重新启动后仍能正常工作。风险在于，崩溃可能会中断更新序列，并在磁盘上留下不一致的数据结构（例如，一个块既在文件中使用，又被标记为空闲）。
- 不同的进程可能并发在文件系统上运行，所以文件系统代码必须协调维护每一个临界区。
- 访问磁盘的速度比访问内存的速度要慢几个数量级，所以文件系统必须维护缓存，用于缓存常用块。

本章剩下的部分将解释 xv6 如何解决这些问题。

8.1 Overview

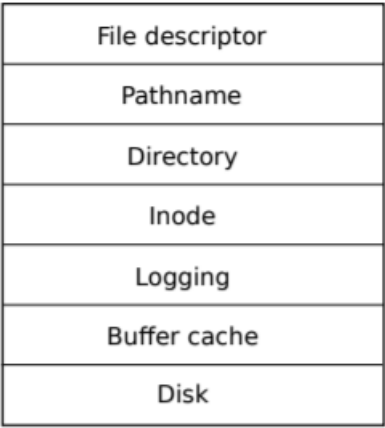


Figure 8.1: Layers of the xv6 file system.

xv6 文件系统的实现分为七层，如图 8.1 所示。磁盘层在 virtio 磁盘上读写块。Buffer 缓存层缓存磁盘块，并同步访问它们，确保一个块只能同时被内核中的一个进程访问。日志层允许上层通过事务更新多个磁盘块，并确保在崩溃时，磁盘块是原子更新的（即全部更新或不更新）。inode 层将一个文件都表示为一个 **inode**，每个文件包含一个唯一的 i-number 和一些存放文件数据的块。目录层将实现了一种特殊的 **inode**，被称为目录，其包含一个目录项序列，每个目录项由文件名称和 i-number 组成。路径名层提供

了层次化的路径名，如`/usr/rtn/xv6/fs.c`，可以用递归查找解析他们。文件描述符层用文件系统接口抽象了许多 Unix 资源（如管道、设备、文件等），使程序员的生产力得到大大的提高。

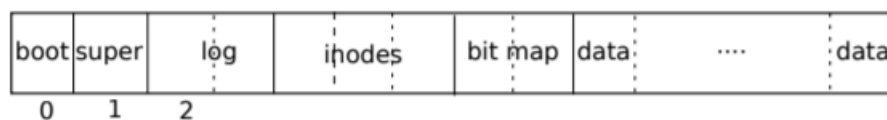


Figure 8.2: Structure of the xv6 file system.

文件系统必须安排好磁盘存储 inode 和内容块的位置。为此，xv6 将磁盘分为几个部分，如图 8.2 所示。文件系统不使用块 0（它存放 boot sector）。第 1 块称为 *superblock*，它包含了文件系统的元数据（以块为单位的文件系统大小、数据块的数量、inode 的数量和日志中的块数）。从块 2 开始存放着日志。日志之后是 inodes，每个块会包含多个 inode。在这些块之后是 *位图块(bitmap)*，记录哪些数据块在使用。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，要么持有文件或目录的内容。超级块由一个单独的程序 `mkfs` 写入，它建立了一个初始文件系统。

本章的其余部分将讨论每一层，从 buffer 缓存开始。低层使用选择合适抽象，可以方便更高层的设计。

8.2 Buffer cache layer

buffer 缓存有两项工作。(1)同步访问磁盘块，以确保磁盘块在内存中只有一个 buffer 缓存，并且一次只有一个内核线程能使用该 buffer 缓存；(2)缓存使用较多的块，这样它们就不需要从慢速磁盘中重新读取。代码见 `bio.c`。

buffer 缓存的主要接口包括 `bread` 和 `bwrite`，`bread` 返回一个在内存中可以读取和修改的块副本 `buf`，`bwrite` 将修改后的 buffer 写到磁盘上相应的块。内核线程在使用完一个 buffer 后，必须通过调用 `brelse` 释放它。buffer 缓存为每个 buffer 的都设有 `sleep-lock`，以确保每次只有一个线程使用 buffer（从而使用相应的磁盘块）；`bread` 返回的 buffer 会被锁定，而 `brelse` 释放锁。

我们再来看看 buffer 缓存。buffer 缓存有固定数量的 buffer 来存放磁盘块，这意味着如果文件系统需要一个尚未被缓存的块，buffer 缓存必须回收一个当前存放其他块的 buffer。buffer 缓存为新块寻找最近使用最少的 buffer（lru 机制）。因为最近使用最少的 buffer 是最不可能被再次使用的 buffer。

8.3 Code: Buffer cache

buffer 缓存是一个由 buffer 组成的双端链表。由函数 **binit** 用静态数组 **buf** 初始化这个链表，**binit** 在启动时由 **main(kernel/main.c:27)**调用。访问 buffer 缓存是通过链表，而不是 **buf** 数组。

buffer 有两个与之相关的状态字段。字段 **valid** 表示是否包含该块的副本（是否从磁盘读取了数据）。字段 **disk** 表示缓冲区的内容已经被修改需要被重新写入磁盘。

bget (kernel/bio.c:59)扫描 buffer 链表，寻找给定设备号和扇区号来查找缓冲区 (kernel/bio.c:65-73)。如果存在，**bget** 就会获取该 buffer 的 sleep-lock。然后 **bget** 返回被锁定的 buffer。

如果给定的扇区没有缓存的 buffer，**bget** 必须生成一个，可能会使用一个存放不同扇区的 buffer，它再次扫描 buffer 链表，寻找没有被使用的 buffer(**b->refcnt = 0**)；任何这样的 buffer 都可以使用。任何这样的 buffer 都可以使用。**bget** 修改 buffer 元数据，记录新的设备号和扇区号，并获得其 sleep-lock。请注意，**b->valid = 0** 可以确保 **bread** 从磁盘读取块数据，而不是错误地使用 buffer 之前的内容。

请注意，每个磁盘扇区最多只能有一个 buffer，以确保写操作对读取者可见，也因为文件系统需要使用 buffer 上的锁来进行同步。**Bget** 通过从第一次循环检查块是否被缓存，第二次循环来生成一个相应的 buffer（通过设置 **dev**、**blockno** 和 **refcnt**），在进行这两步操作时，需要一直持有 **bache.lock**。持有 **bache.lock** 会保证上面两个循环在整体上是原子的。

bget 在 **bcache.lock** 保护的临界区之外获取 buffer 的 sleep-lock 是安全的，因为非零的 **b->refcnt** 可以防止缓冲区被重新用于不同的磁盘块。sleep-lock 保护的是块的缓冲内容的读写，而 **bcache.lock** 保护被缓存块的信息。

如果所有 buffer 都在使用，那么太多的进程同时在执行文件相关的系统调用，**bget** 就会 **panic**。一个更好的处理方式可能是睡眠，直到有 buffer 空闲，尽管这时有可能出现死锁。

一旦 **bread** 读取了磁盘内容（如果需要的话）并将缓冲区返回给它的调用者，调用者就独占该 buffer，可以读取或写入数据。如果调用者修改了 buffer，它必须在释放 buffer 之前调用 **bwrite** 将修改后的数据写入磁盘。**bwrite** (kernel/bio.c:107)调用 **virtio_disk_rw** 与磁盘硬件交互。

当调用者处理完一个 buffer 后，必须调用 **brelease** 来释放它。（**brelease** 这个名字是 **b-release** 的缩写，虽然很神秘，但值得学习，它起源于 Unix，在 BSD、Linux 和 Solaris 中也有使用。）**brelease** (kernel/bio.c:117)释放 sleep-lock，并将该 buffer 移动到链表的头部(kernel/bio.c:128-133)。移动 buffer 会使链表按照 buffer 最近使用的时间（最近释放）排序，链表中的第一个 buffer 是最近使用的，最后一个是最早使用的。**bget** 中的两个循环利用了这一点，在最坏的情况下，获取已缓存 buffer 的扫描必须处理整个链表，由于数据局部性，先检查最近使用的缓冲区（从 **bcache.head** 开始，通过 **next** 指针）将减少扫描时间。扫描选取可使用 buffer 的方法是通过从后向前扫描（通过 **prev** 指针）选取最近使用最少的缓冲区。

8.4 Logging layer

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到对磁盘的多次写入，如果只执行了部分写操作，然后发生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（将文件的长度设置为零并释放其内容块）时发生崩溃。根据磁盘写入的顺序，可能会留下一个引用空闲内容块的 inode，也可能会留下一个已分配但没有被引用的内容块。

后面的这种情况相对来说好一点，但是如果一个 inode 指向被释放的块，很可能在重启后造成严重的问题。重启后，内核可能会将该块分配给另一个文件，现在我们有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，这种情况可能是一个安全问题，因为旧文件的所有者能够读写新文件，即使该文件被另一个用户所拥有。

Xv6 通过简单的日志系统来解决文件系统操作过程中崩溃带来的问题。xv6 的系统调用不直接写磁盘上的文件系统数据结构。相反，它将写入的数据记录在磁盘上的日志中。一旦系统调用记录了全部的写入数据，它就会在磁盘上写一个特殊的提交记录，表明该日志包含了一个完整的操作。这时，系统调用就会将日志中的写入数据写到磁盘上相应的位置。在执行完成后，系统调用将磁盘上的日志清除。

如果系统崩溃并重启，文件系统会在启动过程中恢复自己。如果日志被标记为包含一个完整的操作，那么恢复代码就会将写入的内容复制到它们在磁盘文件系统上的相应位置。如果日志未被标记为包含完整的操作，则恢复代码将忽略并清除该日志。

为什么 xv6 的日志系统可以解决文件系统操作过程中的崩溃问题？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，磁盘的状态就像操作根本没有开始一样。如果崩溃发生在操作提交之后，那么恢复代码会重新执行写操作，可能会重复执行之前的写操作。不管是哪种情况，日志都会使写与崩溃为原子的，即恢复后，所有操作的写入内容，要么都在磁盘上，要么都不在。

8.5 Log design

日志贮存在一个固定位置，由 *superblock* 指定。它由一个 header 块组成，后面是一连串的更新块副本（日志块）。header 块包含一个扇区号数组，其中的每个扇区号都对应一个日志块¹⁴，header 还包含日志块的数量。磁盘上 header 块中的数量要么为零，表示日志中没有事务，要么为非零，表示日志中包含一个完整的提交事务，并有指定数量的日志块。Xv6 在事务提交时会修改 header 块，将日志块复制到文件系统后，会将数量设为零。因此，一个事务中途的崩溃将导致日志 header 块中的计数为零；提交后的崩溃的计数为非零。

为了应对崩溃，每个系统调用都包含一个原子写序列。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，一次提交可能涉及多个完整系统调用的写入。为了避免一个系统调用被分裂到不同的事务中，只有在没有文件系统相关的系统调用正在进行时，日志系统才会提交。

将几个事务一起提交的方法被称为组提交（group commit）。组提交可以减少磁盘操作

¹⁴ 扇区号表明该日志块，应该写入的位置

的次数，因为它将提交的固定成本分摊在了多个操作上。组提交可以让文件系统同时执行更多的并发写，也可以让磁盘在一次磁盘轮转中把它们全部写入。Xv6 的 virtio 驱动不支持这种批处理，但 xv6 的文件系统实现了这种方式。

Xv6 在磁盘上划出固定的空间来存放日志。在一个事务中，系统调用所写的块总数必须适应这个空间的大小。这将导致两个后果：

1、系统调用写入的日志大小必须小于日志空间的大小。这对大多数系统调用来说都不是问题，但有两个系统调用可能会写很多块，**write** 和 **unlink**。大文件的 **write** 可能会写很多数据块和 bitmap 块，以及一个 inode 块；取消链接一个大文件可能会写很多 bitmap 块和一个 inode。Xv6 的 **write** 系统调用将大的写操作分解成多个小的写操作，以适应在日志空间的大小，而 **unlink** 不会引起问题，因为 xv6 文件系统只使用一个位图块。

2、日志空间有限的另一个后果是，日志系统只会在确定了系统调用的写操作可以适应剩余日志空间之后，才会开始执行该系统调用。

8.6 Code: logging

系统调用中一般用法如下：

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

begin_op(kernel/log.c:126)会一直等到日志系统没有 committing，并且有足够的日志空间来容纳这次调用的写。**log.outstanding** 统计当前系统调用的数量，可以通过 **log.outstanding** 乘以 **MAXOPBLOCKS** 来计算已使用的日志空间。自增 **log.outstanding** 既能预留空间，又能防止该系统调用期间进行提交。该代码假设每次系统调用最多写入 **MAXOPBLOCKS** 个块。

log_write (kernel/log.c:214) 是 **bwrite** 的代理。它将扇区号记录在内存中，在磁盘上的日志中使用一个槽，并自增 **buffer.refcnt** 防止该 **buffer** 被重用。在提交之前，块必须留在缓存中，即该缓存的副本是修改的唯一记录；在提交之后才能将其写入磁盘上的位置；该次修改必须对其他读可见。注意，当一个块在一个事务中被多次写入时，他们在日志中的槽是相同的。这种优化通常被称为 *absorption*(吸收)。例如，在一个事务中，包含多个文件的多个 inode 的磁盘块被写多次，这是常见的情况。通过将几次磁盘写吸收为一次，文件系统可以节省日志空间，并且可以获得更好的性能，因为只有一份磁盘块的副本必须写入磁盘。

end_op (kernel/log.c:146) 首先递减 **log.outstanding**。如果计数为零，则通过调用 **commit()** 来提交当前事务。

Commit 分为四个阶段：

1、**write_log**()(kernel/log.c:178)将事务中修改的每个块从 **buffer** 缓存中复制到磁盘上的日志槽中。

2、**write_head()**(kernel/log.c:102)将 header 块写到磁盘上，就表明已提交，为提交点，写完日志后的崩溃，会导致在重启后重新执行日志。

3、**install_trans**(kernel/log.c:69)从日志中读取每个块，并将其写到文件系统中对应的位置。

4、最后修改日志块计数为 0，并写入日志空间的 header 部分。这必须在下一个事务开始之前修改，这样崩溃就不会导致重启后的恢复使用这次的 header 和下次的日志块。

recover_from_log (kernel/log.c:116) 是在 **initlog** (kernel/log.c:55) 中调用的，而 **initlog** 是在第一个用户进程运行 (kernel/proc.c:539) 之前，由 **fsinit**(kernel/fs.c:42) 调用的。它读取日志头，如果日志头显示日志中包含一个已提交的事务，则会像 **end_op** 那样执行日志。

一个使用了日志的例子是 **filewrite** (kernel/file.c:135)。这个事务看起来像这样：

```
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，它将大的写分解成每次只有几个扇区的单独事务，以避免溢出日志空间。调用 **writei** 写入许多块作为这个事务的一部分：文件的 inode，一个或多个 bitmap 块，以及一些数据块。

8.7 Code: Block allocator

文件和目录存储在磁盘块中，必须从空闲池中分配，xv6 的块分配器在磁盘上维护一个 bitmap，每个块对应一个位。0 表示对应的块是空闲的，1 表示正在使用中。程序 **mkfs** 设置引导扇区、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** 申请一个新的磁盘块，**bfree** 释放一个块。**balloc** (kernel/fs.c:71)会有一个循环遍历每一个块，从块 0 开始，直到 **sb.size**，即文件系统中的块数。它寻找一个位为 0 的空闲块。如果 **balloc** 找到了这样一个块，它就会更新 bitmap 并返回该块。为了提高效率，这个循环被分成两部分。外循环读取 bitmap 的一个块，内循环检查块中的所有 BPB 位。如果两个进程同时试图分配一个块，可能会发生竞争，但 buffer 缓存只允许块同时被一个进程访问，这就避免了这种情况的发生。

Bfree (kernel/fs.c:90) 找到相应的 bitmap 块并清除相应的位。**bread** 和 **brelse** 暗含的独占性避免了显式锁定。

与本章其余部分描述的大部分代码一样，**balloc** 和 **bfree** 必须在事务中被调用。

8.8 Inode layer

术语 inode 有两种相关的含义。1、它可能指的是磁盘上的数据结构，其中包含了文件的大小和数据块号的列表；2、inode 可能指的是内存中的 inode，它包含了磁盘上 inode 的

副本以及内核中需要的其他信息。

磁盘上的 inode 被放置磁盘的一个连续区域。每一个 inode 的大小都是一样的，所以，给定一个数字 *n*，很容易找到磁盘上的第 *n* 个 inode。事实上，这个数字 *n*，被称为 inode 号或 *i-number*，在实现中就是通过这个识别 inode 的。

结构体 **dinode**(kernel/fs.h:32)定义了磁盘上的 inode。**type** 字段区分了文件、目录和特殊文件（设备）。**type** 为 0 表示该 inode 是空闲的。**nlink** 字段统计引用这个 inode 的目录项的数量，当引用数为 0 时就释放磁盘上的 inode 及其数据块。**size** 字段记录了文件中内容的字节数。**addrs** 数组记录了持有文件内容的磁盘块的块号。

内核将在使用的 inode 保存在内存中；结构体 **inode** (kernel/file.h:17)是磁盘 **dinode** 的拷贝。内核只在有指针指向 inode 才会储存。**ref** 字段为指向 inode 的指针的数量，如果引用数量减少到零，内核就会从内存中丢弃这个 inode。**iget** 和 **iput** 函数引用和释放 inode，并修改引用计数。指向 inode 的指针可以来自文件描述符，当前工作目录，以及短暂的内核代码，如 **exec**。

在 xv6 的 inode 代码中，有四种锁或类似锁的机制。**icache.lock** 保证了一个 inode 在缓存只有一个副本，以及缓存 inode 的 **ref** 字段计数正确。每个内存中的 inode 都有一个包含 **sleep-lock** 的锁字段，它保证了可以独占访问 inode 的其他字段（如文件长度）以及 inode 的文件或目录内容块的。一个 inode 的 **ref** 如果大于 0，则会使系统将该 inode 保留在缓存中，而不会重用该 inode。最后，每个 inode 都包含一个 **nlink** 字段(在磁盘上，缓存时会复制到内存中)，该字段统计链接该 inode 的目录项的数量；如果一个 inode 的链接数大于零，xv6 不会释放它。

在 xv6 的 inode 代码中，有四种锁或类似锁的机制。**icache.lock** 保证了一个 inode 在缓存只有一个副本，以及缓存 inode 的 **ref** 字段计数正确。每个内存中的 inode 都有一个包含 **sleep-lock** 的锁字段，它保证了可以独占访问 inode 的其他字段（如文件长度）以及 inode 的文件或目录内容块的。一个 inode 的 **ref** 如果大于 0，则会使系统将该 inode 保留在缓存中，而不会重用该缓存 buffer。最后，每个 inode 都包含一个 **nlink** 字段(在磁盘上，如果是缓存，则复制在内存中)，该字段统计引用文件的目录项的数量；只有当 inode 的链接数为零时，xv6 才会释放它。

iget()返回的 **inode** 指针在调用 **iput()**之前都是有效的；inode 不会被删除，指针所引用的内存也不会被另一个 inode 重新使用。**iget()**提供了对 inode 的非独占性访问，因此可以有许多指针指向同一个 inode。文件系统代码中的许多部分都依赖于 **iget()**的这种行，既是为了保持对 inode 的长期引用(如打开的文件和当前目录)，也是为了防止竞争，同时避免在操作多个 inode 的代码中出现死锁(如路径名查找)。

inode 缓存只缓存被指针指向的 inode。它的主要工作其实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，如果不被 inode 缓存保存，buffer 缓存可能会把它保存在内存中。inode 缓存是 *write-through* 的，这意味着缓存的 inode 被修改，就必须立即用 **iupdate** 把它写入磁盘。

8.9 Code: Inodes

要创建一个新的 inode(例如，当创建一个文件时)，xv6 会调用 **ialloc**(kernel/fs.c:196)。

ialloc 类似于 **balloc**：它遍历磁盘上的 inode，寻找一个被标记为空闲的 inode。当它找到后，它会修改该 inode 的 **type** 字段来使用它，最后调用 **iget** (kernel/fs.c:210) 来从 inode 缓存中返回一个条目。由于一次只能有一个进程持有对 **bp** 的引用，所以 **ialloc** 可以正确执行。**ialloc** 可以确保其他进程不会同时看到 inode 是可用的并使用它。

iget (kernel/fs.c:243) 在 inode 缓存中寻找一个带有所需设备号和 inode 号码的 active 条目 (**ip->ref** > 0)。如果它找到了，它就返回一个新的对该 inode 的引用 (kernel/fs.c:252-256)。当 **iget** 扫描时，它会记录第一个空槽的位置 (kernel/fs.c:257-258)，当它需要分配一个缓存条目时，它会使用这个空槽。

在读写 inode 的元数据或内容之前，代码必须使用 **ilock** 锁定它。**ilock**(kernel/fs.c:289) 使用 sleep-lock 内部有一个睡眠锁来锁定。一旦 **ilock** 锁定了 inode，它会根据自己的需要从磁盘 (更有可能是 buffer 缓存) 读取 inode。函数 **iunlock** (kernel/fs.c:317) 释放睡眠锁，这会唤醒正在等待该睡眠锁的进程。

iput (kernel/fs.c:333) 通过递减引用次数 (kernel/fs.c:356) 释放指向 inode 的指针。如果递减后的引用数为 0，inode 缓存中的 就会释放掉该 inode 在 inode 缓存中的槽位，该槽位就可以被其他 inode 使用。

如果 **iput** 发现没有指针指向该 inode，并且没有任何目录项链接该 inode (不在任何目录中出现)，那么该 inode 和它的数据块必须被释放。**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)。

iput 在释放 inode 的锁定协议是值得我们仔细研究。一个危险是，一个并发线程可能会在 **ilock** 中等待使用这个 inode (例如，读取一个文件或列出一个目录)，但它没有意识到该 inode 可能被释放掉了。这种情况是不会发生，因为该 inode 的没有被目录项链接且 **ip->ref** 为 1，那么系统调用是没有这个指针的 (如果有，**ip->ref** 应该为 2)。这一个引用是调用 **iput** 的线程所拥有的。的确，**iput** 会在其 **icache.lock** 锁定的临界区之外检查引用数是否为 1，但此时已知链接数为 0，所以没有线程会尝试获取新的引用。另一个主要的危险是，并发调用 **ialloc** 可能会使 **iput** 返回一个正在被释放的 inode。这种情况发生在 **iupdate** 写磁盘时 **ip->type=0**。这种竞争是正常的，分配 inode 的线程会等待获取 inode 的睡眠锁，然后再读取或写入 inode，但此时 **iput** 就结束了。

iput() 会写磁盘。这意味着任何使用文件系统的系统调用都会写磁盘，因为系统调用可能是最后一个对文件有引用的调用。甚至像 **read()** 这样看似只读的调用，最终也可能会调用 **iput()**。这又意味着，即使是只读的系统调用，如果使用了文件系统，也必须用事务来包装。

崩溃发生在 **iput()** 中是相当棘手的。当文件的链接数降到零时，**iput()** 不会立即截断一个文件，因为一些进程可能仍然在内存中持有对 inode 的引用：一个进程可能仍然在对文件进行读写，因为它成功地打开了 inode。但是，如果崩溃发生在该文件的最后一个文件描述符释放时，那么该文件将被标记为已在磁盘上分配，但没有目录项指向它。

文件系统处理这种情况的方法有两种。简单的解决方法是，是在重启后的恢复时，文件系统会扫描整个文件系统，寻找那些被标记为已分配的文件，但没有指向它们的目录项。如果有这样的文件存在，那么就可以释放这些文件。

第二种解决方案不需要扫描文件系统。在这个解决方案中，文件系统在磁盘上 (例如，在 **superblock** 中) 记录链接数为 0 但引用数不为 0 的文件的 inode 的 **inumber**。如果文件系统在其引用计数达到 0 时删除该文件。当文件的引用数为 0 时，文件系统会删除该文件，

同时它更新磁盘上的列表，从列表中删除该 inode。恢复时，文件系统会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 inode 可能会在磁盘上被标记分配，即使它们不再使用。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 Code: Inode content

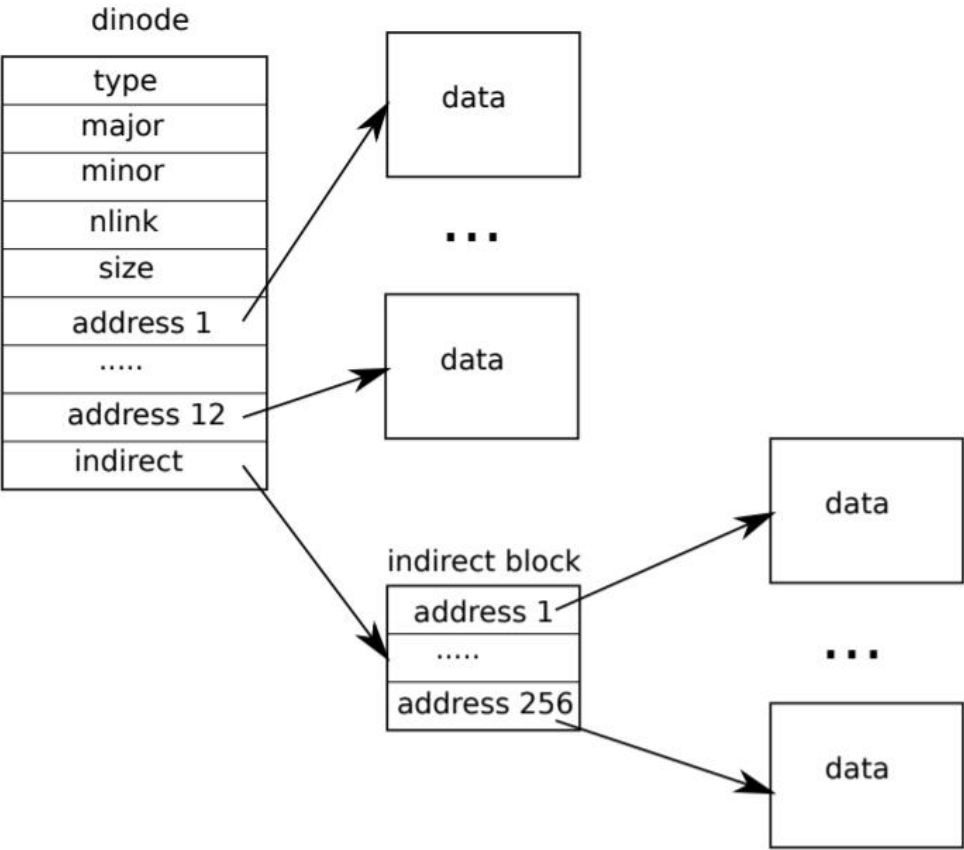


Figure 8.3: The representation of a file on disk.

磁盘上的 **inode**，即 **dinode** 结构体，包含一个 `size` 和一个块号数组（见图 8.3）。**inode** 数据可以在 **dinode** 的 `addrs` 数组中找到。开始的 **NDIRECT** 个数据块放置在数组中的前 **NDIRECT** 个条目中，这些块被称为**直接块**。接下来的 **NINDIRECT** 个数据块并没有放置在 **inode** 中，而是被存放在叫做**间接块**的数据块中。`addrs` 数组中的最后一个条目给出了放置间接块的地址。因此，一个文件的前 12 kB ($\text{NDIRECT} \times \text{BSIZE}$) 字节可以从 **inode** 中列出的块中加载，而接下来的 256 kB ($\text{NINDIRECT} \times \text{BSIZE}$) 字节只能在查阅间接块后才能取出。对于磁盘这是一种不错的表示方式，但对客户机就有点复杂了。函数 **bmap** 包装了这种表示方式的是高层次的函数，如 **readi** 和 **writei** 可以更好的使用。**Bmap** 返回 inode **ip** 的第 **bn** 个数据块的磁盘块号。如果 **ip** 没有第 **bn** 个的数据块，**bmap** 就会分配一个。

函数 **bmap**(kernel/fs.c:378)从简单的情况开始：最前面的 **NDIRECT** 个块储存在 inode(kernel/fs.c:383-387)中，接下来的 **NINDIRECT** 个块放置在 `ip->addrs[NDIRECT]`指向的**间接块**中。**Bmap** 读取间接块(kernel/fs.c:394)，然后从块内的正确的位置读取一个块号

(kernel/fs.c:395)。如果块号超过了 **NDIRECT+NINDIRECT**, **bmap** 就会 **panic**; **writei** 会检查并防止这种情况(kernel/fs.c:490)。

Bmap 根据需要分配块。**ip->addrs[bn]**或间接条目为 0 时表示没有块。当 **bmap** 遇到 0 时, 它会用新的块号来代替 0(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**中指向的块(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** 可以很容易地获取一个 inode 的数据。**Readi** (kernel/fs.c:456)首先要确定偏移量和计数没有超过文件末端。从文件超出末尾开始的读会返回一个错误(kernel/fs.c:461-462), 而从文件末尾开始或读取过程中超出末尾的读会不会返回错误, 只是返回的字节数会少于请求的字节数(kernel/fs.c:463-464)。

主循环会把文件中的每一个块的数据复制到 **dst** 中(kernel/fs.c:466-474)。**writei** (kernel/fs.c:483)与 **readi** 相同, 但有三个不同: (1)、从文件末尾开始或越过文件末尾的写入会使文件增长, 但不会超过文件的最大长度(kernel/fs.c:490-491); (2)、循环将数据复制到缓冲区而不是 **out**(kernel/fs.c:36); (3)、如果写使文件增长了, **writi** 必须更新它的大小(kernel/fs.c:504-511)。

readi 和 **writei** 开始都会检查 **ip->type == T_DEV**。这种情况处理的是数据不在文件系统中的特殊设备; 我们将在文件描述符层中再讨论这种情况。

函数 **stati** (kernel/fs.c:442) 将 inode 元数据复制到 **stat** 结构体中, 通过 **stat** 系统调用暴露给用户程序。

8.11 Code: directory layer

目录的实现机制和文件很类似。它的 **inode** 类型是 **T_DIR**, 它的数据是一个目录项的序列。每个条目是一个结构体 **dirent**(kernel/fs.h:56), 它包含一个名称和一个 inode 号。名称最多包含 **DIRSIZ**(14)个字符, 较短的名称以 **NULL**(0)结束。inode 号为 0 的目录项是空闲的。

函数 **dirlookup** (kernel/fs.c:527)在一个目录中搜索一个带有给定名称的条目。如果找到了, 它返回一个指向相应 inode 的指针, 解锁该 inode, 并将***poff** 设置为目录中条目的字节偏移量, 以防调用者想要编辑它。如果 **dirlookup** 找到一个名字正确的条目, 则更新***poff**, 并返回一个通过 **iget** 获得的未被锁定的 inode。**Dirlookup** 是 **iget** 返回未锁定的 inode 的原因。调用者已经锁定了 **dp**, 所以如果查找的是 **“.”**, 当前目录的别名, 在返回之前试图锁定 inode, 就会试图重新锁定 **dp** 而死锁。(还有更复杂的死锁情况, 涉及到多个进程和**“..”**, 父目录的别名; **“.”**不是唯一的问题。)调用者可以先解锁 **dp**, 然后再锁定 **ip**, 保证一次只持有一个锁。

函数 **dirlink** (kernel/fs.c:554)会在当前目录 **dp** 中创建一个新的目录项, 通过给定的名称和 inode 号。如果名称已经存在, **dirlink** 将返回一个错误(kernel/fs.c:560-564)。主循环读取目录项, 寻找一个未使用的条目。当它找到一个时, 它会提前跳出循环 (kernel/fs.c:538-539), 并将 **off** 设置为该可用条目的偏移量。否则, 循环结束时, 将 **off** 设置为 **dp->size**。不管是哪种方式, **dirlink** 都会在偏移量 **off** 的位置添加一个新的条目到目录中(kernel/fs.c:574-577)。

8.12 Code: Path names

查找路径名会对每一个节点调用一次 **dirlookup**。Namei (kernel/fs.c:661) 解析路径并返回相应的 inode。函数 **nameparent** 是 **namei** 的一个变种：它返回相应 inode 的父目录 inode，并将最后一个元素复制到 **name** 中。这两个函数都通过调用 **namex** 来实现。

Namex (kernel/fs.c:626) 首先确定路径解析从哪里开始。如果路径以斜线开头，则从根目录开始解析；否则，从当前目录开始解析(kernel/fs.c:630-633)。然后它使用 **skipelem** 来遍历路径中的每个元素(kernel/fs.c:635)。循环的每次迭代都必须在当前 inode **ip** 中查找 **name**。迭代的开始是锁定 **ip** 并检查它是否是一个目录。如果不是，查找就会失败(kernel/fs.c:636-640)。(锁定 **ip** 是必要的，不是因为 **ip->type** 可能会改变，而是因为 **ilock** 运行之前，不能保证 **ip->type** 已经从磁盘载入)。如果调用的是 **nameparent**，而且这是最后一个路径元素，按照之前 **nameparent** 的定义，循环应该提前停止，最后一个路径元素已经被复制到 **name** 中，所以 **namex** 只需要返回解锁的 **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)。当循环遍历完路径元素时，它返回 **ip**。

namex 可能需要很长的时间来完成：它可能会涉及几个磁盘操作，通过遍历路径名得到的目录的 inode 和目录块（如果它们不在 buffer 缓存中）。Xv6 经过精心设计，如果一个内核线程对 **namex** 的调用在阻塞在磁盘 I/O 上，另一个内核线程查找不同的路径名可以同时进行。**Namex** 分别锁定路径中的每个目录，这样不同目录的查找就可以并行进行。

这种并发性带来了一些挑战。例如，当一个内核线程在查找一个路径名时，另一个内核线程可能正在取消链接一个目录，这会改变目录数。一个潜在的风险是，可能一个查找线程正在搜索的目录可能已经被另一个内核线程删除了，而它的块已经被另一个目录或文件重用了。

Xv6 避免了这种竞争。例如，在 **namex** 中执行 **dirlookup** 时，查找线程会持有目录的锁，**dirlookup** 返回一个使用 **iget** 获得的 inode。**iget** 会增加 inode 的引用次数。只有从 **dirlookup** 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 inode，但 xv6 还不会删除 inode，因为 inode 的引用数仍然大于零。

另一个风险是死锁。例如，当查找"."时，**next** 指向的 inode 与 **ip** 相同。在释放对 **ip** 的锁之前锁定 **next** 会导致死锁。为了避免这种死锁，**namex** 在获得对 **next** 的锁之前就会解锁目录。这里我们再次看到为什么 **iget** 和 **ilock** 之间的分离是很重要的。

8.13 File descriptor layer

Unix 接口很酷的一点是：Unix 中的大部分资源都是以文件的形式来表示的。包括控制台、管道等设备，当然还有真实的文件。文件描述符层就是实现这种统一性的一层。

Xv6 给每个进程提供了自己的打开文件表，或者说文件描述符表，就像我们在第一章中看到的那样。每个打开的文件由一个结构体 **file**(kernel/file.h:1)表示，它包装 inode 或管道，也包含一个 I/O 偏移量。每次调用 **open** 都会创建一个新的打开文件(一个新的结构体 **file**)，如果多个进程独立打开同一个文件，那么不同的 **file** 实例会有不同的 I/O 偏移量。另一方面，一个打开的文件（同一个结构文件）可以在一个进程的文件表中出现多次，也可以在多个进

程的文件表中出现。如果一个进程使用 **open** 打开文件，然后使用 **dup** 创建别名，或者使用 **fork** 与子进程共享文件，就会出现这种情况。引用计数可以跟踪特定打开文件的引用数量。一个文件的打开方式可以为读，写，或者读写。通过 **readable** 和 **writable** 来指明。

系统中所有打开的文件都保存在一个全局文件表中，即 **fTable**。文件表的功能有：分配文件(**filealloc**)、创建重复引用(**fileup**)、释放引用(**fileclose**)、读写数据(**fileread** 和 **filewrite**)。

前三个函数应该比较熟悉了,就不过多的讨论。**Filealloc** (kernel/file.c:30) 扫描文件表，寻找一个未引用的文件 (**f->ref == 0**)，并返回一个新的引用；**fileup** (kernel/file.c:48) 增加引用计数；**fileclose** (kernel/file.c:60) 减少引用计数。当一个文件的引用数达到 0 时，**fileclose** 会根据类型释放底层的管道或 inode。

函数 **filestat**、**fileread** 和 **filewrite** 实现了对文件的统计、读和写操作。**Filestat**(kernel/file.c:88)只允许对 inodes 进行操作，并调用 **stati**。**Fileread** 和 **filewrite** 首先检查打开模式是否允许该操作，然后再调用管道或 inode 的相关实现。如果文件代表一个 inode，**fileread** 和 **filewrite** 使用 I/O 偏移量作为本次操作的偏移量，然后前移偏移量 (kernel/file.c:122- 123) (kernel/file.c:153-154)。Pipes 没有偏移量的概念。回想一下 inode 的函数需要调用者处理锁的相关操作 (kernel/file.c:94-96) (kernel/file.c:121-124) (kernel/file.c:163-166)。inode 加锁附带了一个不错的作用，那就是读写偏移量是原子式更新的，这样多个进程写一个文件时，自己写的数据就不会被其他进程所覆盖，尽管他们的写入可能最终会交错进行。

8.14 Code: System calls

有了更低层提供的函数，大多数系统调用的实现都是比较简单的(见(kernel/sysfile.c))。有几个调用值得仔细研究一下。

函数 **sys_link** 和 **sys_unlink** 可以编辑目录，创建或删除对 inodes 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) 首先获取它的参数，两个字符串 **old** 和 **new** (kernel/sysfile.c:125)。假设 **old** 存在并且不是一个目录 (kernel/sysfile.c:129-132)，**sys_link** 会递增它的 **ip->nlink** 计数。然后 **sys_link** 调用 **nameiparent** 找到 **new** (kernel/sysfile.c:145) 的父目录和最终路径元素，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，并且和现有的 inode 在同一个设备上，inode 号只在同一个磁盘上有意义。如果出现这样的错误，**sys_link** 必须返回并减少 **ip->nlink**。

事务简化了这个函数的实现，因为它需要更新多个磁盘块，但我们不必担心做这些事情的顺序。它们要么全部成功，要么都不成功。例如，如果没有事务，在创建链接之前更新 **ip->nlink**，会使文件系统暂时处于不安全的状态，中间的崩溃可能会造成破坏。有了事务，我们就不用担心这个问题了。

Sys_link 为一个现有的 inode 创建一个新的名字。而函数 **create** (kernel/sysfile.c:242) 为一个新的 inode 创建一个新的名字。它是三个文件创建相关的系统调用的综合：使用 **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，以及 **mkdev** 创建一个新的设备文件。和 **sys_link** 一样，**create** 也是通过调用 **nameiparent** 来获取父目录的 inode。然后调用 **dirlookup** 来检查名称是否已经存在 (kernel/sysfile.c:252)。如果名称存在，**create** 的行为取决于它被用于哪个系统调用：**open**，**mkdir**，**mkdev** 的语义不同。如果 **create** 是由 **open** 使用的 (**type == T_FILE**)，如果能根据该名字存在且为普通文件，那么打

开就会成功，所以 **create** 也会成功 (kernel/sysfile.c:256)。否则，就会报错 (kernel/sysfile.c:257-258)。如果这个名字不存在，**create** 就会用 **ialloc** 分配一个新的 inode (kernel/sysfile.c:261)。如果新的 inode 是一个目录，**create** 会在该目录添加条目“.”和“..”。最后，现在数据已经被正确地初始化了，**create** 可以把它链接到父目录中 (kernel/sysfile.c:274)。和 **sys_link** 一样，**create** 同时拥有两个 inode 锁：**ip** 和 **dp**。没有死锁的可能性，因为 inode **ip** 是新分配的：系统中没有其他进程会持有 **ip** 的锁并尝试锁住 **dp**。

使用 **create**，很容易实现 **sys_open**、**sys_mkdir** 和 **sys_mknod**。**sys_open** (kernel/sysfile.c:287) 是其中最复杂的，因为创建一个新文件只是它做的一小部分。如果 **open** 的打开模式包含为 **O_CREATE**，它就会调用 **create** (kernel/sysfile.c:301)。否则，它会调用 **namei** (kernel/sysfile.c:307)。**create** 会返回一个锁定的 inode，但 **namei** 不会，所以 **sys_open** 必须锁定 inode。这比较有一个方便的地方是目录只会以读打开，而不是写。假设 inode 是通过某种方式获得的，**sys_open** 会分配一个文件和一个文件描述符 (kernel/sysfile.c:325)，然后设置 file 相关的字段 (kernel/sysfile.c:337- 94 342)。注意，没有其他进程可以访问这个部分初始化的文件，因为它只在当前进程的表中。

第 7 章在我们还没有文件系统之前就研究了管道的实现。函数 **sys_pipe** 通过提供创建管道对的方法将管道实现与文件系统连接起来。它的参数是一个指针，该指针指向一个长度为 2 的 int 类型数组，它将在这里记录两个新的文件描述符。然后它分配管道并装入文件描述符。

8.15 Real world

实际操作系统中的 buffer 缓存要比 xv6 的复杂得多，但它有同样的两个目的：缓存和同步访问磁盘。xv6 的 buffer 缓存和 V6 是一样的，使用简单的最近最少使用 (LRU) 抛弃策略；可以实现许多更复杂的策略，每种策略都对某些情况有好处，而对其它情况没有好处。更高效的 LRU 缓存不使用链表，而使用哈希表进行查找，使用堆进行 LRU 抛弃。现代的 buffer 缓存通常与虚拟内存系统集成在一起，以支持内存映射的文件。

Xv6 的日志系统效率低下。提交不能与文件系统系统调用同时发生。系统会记录整个块，即使一个块中只有几个字节被改变。它执行同步的日志写入，一次写一个块，每一个块都需要整个磁盘旋转时间。真正的日志系统可以解决所有这些问题。

日志不是提供崩溃恢复的唯一方法。早期的文件系统在重启期间使用 scavenger（例如 UNIX fsck 程序）来检查每个文件和目录以及块和 inode 空闲列表，寻找并解决不一致的地方。对于大型文件系统来说，清扫可能需要几个小时的时间，而且在某些情况下，这种方式要想获得的数据一致性，其系统调用必须是一致性的。从日志中恢复要快得多，而且在面对崩溃时，会导致系统调用是原子的。

Xv6 使用了与早期 UNIX 相同的 inodes 和目录的基本磁盘布局；这个方案多年来任还在使用。BSD 的 UFS/FFS 和 Linux 的 ext2/ext3 使用基本相同的数据结构。文件系统布局中最低效的部分是目录，在每次查找过程中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，但对于持有许多文件的目录来说是昂贵的。微软 Windows 的 NTFS，Mac OS X 的 HFS，以及 Solaris 的 ZFS，将一个目录在磁盘上实现了平衡树，用于查找块。这很复杂，但可以保证目录查找的时间复杂度为 $O(\log_n)$ 。

Xv6 对磁盘故障的处理很简单：如果磁盘操作失败，xv6 就会 **panic**。这是否合理取决

于硬件：如果一个操作系统位于特殊的硬件之上，这种硬件会使用冗余来掩盖故障，也许操作系统看到故障的频率很低，以至于直接 **panic** 是可以的。另一方面，使用普通磁盘的操作系统应该使用更加优雅的方式来处理异常，这样一个文件中一个块的丢失就不会影响文件系统其他部分的使用。

Xv6 要求文件系统固定在磁盘设备上，而且大小不能改变。随着大型数据库和多媒体文件对存储要求越来越高，操作系统正在开发消除每个文件系统一个磁盘瓶颈的方法。基本的方法是将许多磁盘组合成一个逻辑磁盘。硬件解决方案（如 RAID）仍然是最流行的，但目前的趋势是尽可能地在软件中实现这种逻辑。这些软件实现通常允许丰富的功能，如通过快速添加或删除磁盘来增长或缩小逻辑设备。当然，一个能够快速增长或收缩的存储层需要一个能够做到同样的文件系统：xv6 使用的固定大小的 inode 块阵列在这样的环境中不能很好地工作。将磁盘管理与文件系统分离可能是最简洁的设计，但也有文件系统将两者通过复杂的接口将他们耦合在一起，如 Sun 公司的 ZFS，将两者结合起来。

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

现代 Unix 系统允许用与磁盘存储相同的系统调用来访问许多种类的资源：命名管道、网络连接、远程访问的网络文件系统以及监视和控制接口，如 `/proc`。这些系统没有 xv6 在 `fileread` 和 `filewrite` 中的 `if` 语句，而是通常给每个打开的文件一个函数指针表，每个代表一个操作，调用函数指针来调用该 inode 的实现调用。网络文件系统和用户级文件系统提供了将这些调用变成网络 RPC 的函数，并在返回前等待响应。

8.16 Exercises

- 1、为什么 **balloc** 需要 **panic**？xv6 能恢复吗？
- 2、为什么 **ialloc** 需要 **panic**？xv6 能恢复吗？
- 3、为什么 **filealloc** 用完文件后不 **panic**？为什么这种情况比较常见，而值得处理？
- 4、假设在执行 **sys_link** 时，另一个进程在 **iunlock(ip)** 和 **dirlink** 之间解除链接该 **ip** 对应的文件。链接会被正确创建吗？为什么或为什么不呢？
- 5、**create** 调用了四次函数（一个 **ialloc**，三次 **dirlink**），它需要这些函数成功返回。如果任何一个不成功，**create** 调用就会 **panic**。为什么这可以接受？为什么这四个调用不能有一个失败呢？
- 6、**sys_chdir** 在 **iput(cp->cwd)** 之前调用 **iunlock(ip)**，这可能会尝试锁定 **cp->cwd**，但将 **iunlock(ip)** 推迟到 **iput** 之后不会造成死锁。为什么不会呢？
- 7、实现 **lseek** 系统调用。**lseek** 还需要你修改 **filewrite**，如果 **lseek** 设置超过 **f->ip->size**，则在文件中用零来填补空缺。
- 8、增加 **O_TRUNC** 和 **O_APPEND** 打开，使 **>** 和 **>>** 操作符在 shell 中工作。
- 9、修改文件系统使其支持符号链接。
- 10、修改文件系统使其支持命名管道。

11、修改文件系统和 VM 使其支持内存映射文件。

Chapter 9 Concurrency revisited

要想同时获得良好的性能, 并发时的正确性和易于理解的代码是内核设计的一大挑战。直接使用锁是得到正确性的最佳途径, 但不总是这样。本章重点介绍了 xv6 不得不使用使用锁的例子, 以及使用类似锁但不是锁的例子。

9.1 Locking patterns

缓存项通常是锁的一个挑战。例如, 文件系统的块缓存(kernel/bio.c:26)存储了 **NBUF** 个磁盘块的副本。一个给定的磁盘块在缓存中最多只有一个副本, 这一点非常重要; 否则, 不同的进程可能会对同一磁盘块的不同副本进行修改时会发生冲突。每一个缓存的磁盘块都被存储在一个 **buf** 结构中(kernel/buf.h:1)。**buf** 结构有一个锁字段, 它有助于确保每次只有一个进程使用一个给定的磁盘块。然而, 这个锁是不够的: 如果一个块根本不存在于缓存中, 而两个进程想同时使用它怎么办? 没有 **buf** (因为该块还没有被缓存), 因此没有什么需要锁定的。Xv6 对每一个块的唯一标识符关联一个额外的锁来处理这种情况。判断块是否被缓存的代码 (e.g. `bget(kernel/bio.c:59)`), 或改变缓存块的集合的代码, 必须持有 **bcache.lock**。当代码找到它所需要的块和 **buf** 结构后, 他就可以释放 **bcache.lock**, 然后锁定特定的块, 这是一种通用模式: 一组项一个锁, 外加每个项一个锁。

通常情况下, 获取锁的同一个函数会释放它。但更准确的想法是, 当一个序列需要保证原子性时, 会在该序列开始时获取锁, 序列结束时释放。如果序列的开始和结束在不同的函数中, 或者不同的线程中, 或者在不同的 CPU 上, 那么锁的获取和释放也必须是一样的。锁的功能是强制其他的使用等待, 而不是将一段数据钉在特定的代理上。一个例子是 `yield` 中的 **acquire**(kernel/proc.c:515), 它是在调度线程中释放的, 而不是在获取锁的进程中释放的。另一个例子是 **ilock**(kernel/fs.c:289)中的 **acquiresleep**; 这段代码经常在读取磁盘时睡眠; 它可能不同的 CPU 上醒来, 这意味着锁可能不同的 CPU 上获取和释放。

释放一个被锁保护的對象时, 若该锁时嵌入在对象里的, 释放这个对象是一件很棘手的事情, 因为拥有锁并不足以保证释放对象的正确性。当有其他线程在 **acquire** 中等待使用对象时, 问题就会出现; 释放这个对象就意味着释放嵌入的锁, 释放这个锁会导致等待线程出错。一种方式是追踪该对象有多少个引用, 为了将只有在最后一个引用消失时才会释放对象。**pipeclose** (kernel/pipe.c:59)就是这样的一个例子。**pi->readopen** 和 **pi->writeopen** 跟踪管道是否有文件描述符引用它。

9.2 Lock-like patterns

在许多地方, xv6 使用引用计数或标志作为一种软锁, 以表明一个对象已被分配, 不应该被释放或重用。进程的 **p->state** 以这种起作用, 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, 锁都会保护标志或引用计数, 但正是标志或引用计数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共享锁, 可以由多个进程持有, 以避免代码使用普通锁时出现的死锁。例如, **namex**(kernel/fs.c:626)中的循环依次锁定每个路径名

命名的目录。然而，**namex** 必须在循环末尾释放每一个锁，因为如果它持有多个锁，那么如果路径名中包含一个点(例如，“a/”，“./b”)，它可能会与自己发生死锁。它也可能因为涉及目录和“..”,“.”的并发查找而死锁。正如第 8 章所解释的那样，解决方案是让循环将目录 inode 带入下一次迭代，并增加其引用计数，但不锁定。

有些数据项在不同的时候会受到不同机制的保护，有时可能会被 xv6 代码的结构隐式保护，而不是通过显式锁来防止并发访问。例如，当一个物理页是空闲的时候，它被 **kmem.lock** (**kernel/kalloc.c:24**) 保护。如果页面被分配作为管道(**kernel/pipe.c:23**)，它将被一个不同的锁(嵌入的 **pi->lock**)保护。如果该页被重新分配给一个新进程的用户内存，它就不会受到锁的保护。相反，分配器不会将该页交给任何其他进程（直到它被释放）的事实保护了它不被并发访问。一个新进程的内存的所有权是很复杂的：首先父进程在 **fork** 中分配和操作它，然后子进程使用它，（在子进程退出后）父进程再次拥有内存，并将其传递给 **kfree**。这里有两个需要注意的地方：第一，一个数据对象在其生命周期中的不同点可以用不同的方式来保护它不被并发访问；第二，保护的形式可能是隐式结构而不是显式锁。

最后一个类似于锁的例子是在调用 **mycpu()**(**kernel/proc.c:68**)时需要禁用中断。禁用中断会导致调用代码对定时器中断是原子性的，而定时器中断可能会强制上下文切换，从而将进程移到不同的 CPU 上。

9.3 No locks at all

xv6 有几个地方是在完全没有锁的情况下共享可变数据的。一个是在 **spinlocks** 的实现中，尽管你可以把 RISC-V 原子指令看作是依靠硬件实现的锁。另一个是 **main.c** (**kernel/main.c:7**)中的 **started** 变量，用来防止其他 CPU 运行，直到 CPU 0 完成 xv6 的初始化；**volatile** 确保编译器真正生成加载和存储指令。

Xv6 包含这样的情况：一个 CPU 或线程写一些数据，另一个 CPU 或线程读数据，但没有专门的锁来保护这些数据。例如，在 **fork** 中，父线程写入子线程的用户内存页，子线程（不同的线程，可能在不同的 CPU 上）读取这些页；没有锁显式地保护这些页。严格来说，这不是锁的问题，因为子线程在父线程写完后才开始执行。这是一个潜在的内存排序问题（见第 6 章），因为没有内存屏障，就没有理由期望一个 CPU 看到另一个 CPU 的写入。然而，由于父线程 CPU 释放锁，而子线程 CPU 在启动时获取锁，所以在 **acquire** 和 **release** 中的内存屏障保证了子线程 CPU 能看到父线程 CPU 的写入。

9.4 Parallelism

锁主要是为了正确性而抑制并行性。因为性能也很重要，所以内核设计者经常要考虑如何使用锁，来保证正确性和良好的并行性。虽然 xv6 并不是为高性能而设计的，但仍然值得考虑哪些 xv6 操作可以并行执行，哪些操作可能在锁上发生冲突。

xv6 中的管道是一个并行性相当好的例子。每个管道都有自己的锁，因此不同的进程可以在不同的 CPU 上并行读写不同的管道。然而，对于一个给定的管道，writer 和 reader 必须等待对方释放锁，他们不能同时读/写同一个管道。还有一种情况是，从一个空管道读（或向一个满管道写）必须阻塞，但这不是因为锁的方案的问题。

上下文切换是一个比较复杂的例子。两个内核线程，每个线程在自己的 CPU 上执行，可以同时调用 **yield**、**sched** 和 **swtch**，这些调用将并行执行。这两个线程各自持有一个锁，但它们是不同的锁，所以它们不必等待对方。但是一旦进入调度器，两个 CPU 在遍历进程表的时候，可能会在一个 **RUNABLE** 的进程上发生锁冲突。也就是说，xv6 在上下文切换的过程中，很可能会从多个 CPU 中获得性能上的好处，但可能没有那么多。

另一个例子是在不同的 CPU 上从不同的进程并发调用 **fork**。这些调用可能需要互相等待 **pid_lock** 和 **kmem.lock**，以及在进程表中搜索一个 **UNUSED** 进程所需的进程锁。另一方面，两个正在 **fork** 的进程可以完全并行地复制用户内存页和格式化页表页。

上述每个例子中的锁方案在某些情况下都牺牲了并行性能。在每一种情况下，都有可能获得更多的并行性通过更复杂的设计。这是否值得取决于实现细节：相关操作被调用的频率、代码在争用锁的情况下所花费的时间、有多少 CPU 可能同时运行冲突的操作、代码的其他部分才是真正的限制性瓶颈。很难猜测一个给定的锁方案是否会导致性能问题，或者一个新的设计是否有明显的改进，所以往往需要在现实的工作负载上进行测量。

9.5 Exercises

- 1、修改 xv6 管道的实现，允许对同一管道的读和写在不同内核上并行进行。
- 2、修改 xv6 **scheduler()**，以减少不同内核同时寻找可运行进程时的锁争用。
- 3、消除 **fork** 中一些串行执行的代码。