

xv6: a simple, Unix-like teaching operating system

Russ Cox Frans Kaashoek Robert Morris

August 31, 2020

Chapter 1 Operating system interfaces

操作系统的任务是在多个程序之间共享一台计算机，并提供一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。

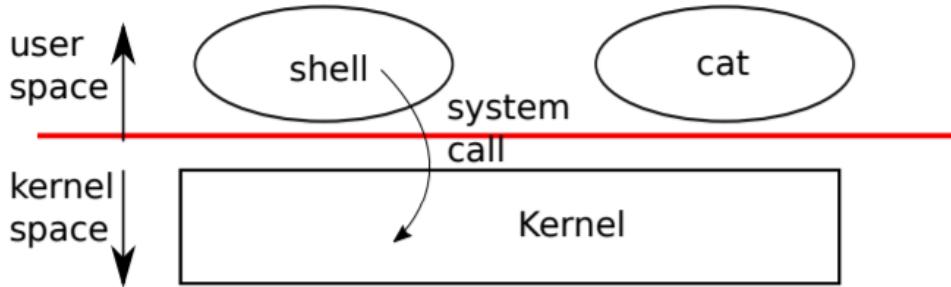


Figure 1.1: A kernel and two user processes.

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；只在错误时返回
<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了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，这取决于是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）。

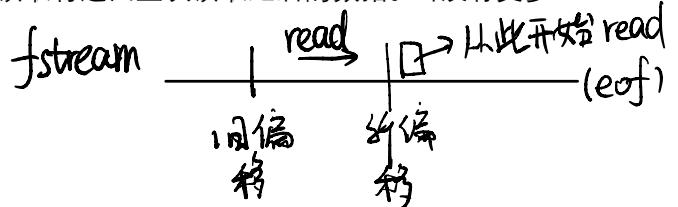
Xv6 隐式分配大部分用户空间内存：fork 复制父进程的内存到子进程，exec 分配足够的内存来容纳可执行文件。一个进程如果在运行时需要更多的内存（可能是为了 malloc），可以调用 sbrk(n) 将其数据内存增长 n 个字节；sbrk 返回新内存的位置。

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。~~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。



file描述符
0
1
2

`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);
    }
}
```

→ 从标准输入读入缓冲 (n 个字节)
↓
 $n \leq \text{sizeof}(buf)$
→ 向标准输出写出 (n 为读入的字节数)

△ 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。

标准
控制台

`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
```

这是大写的 0，不是零

```
}
```

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。

① fork()

② & I/O 重定向

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。→现在是 shell 做

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-**

↓
不懂

`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

它以文件描述符对的形式提供给进程

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]); // 只留标准输入口
    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]);
}
```

手写注释：
把 0 分配给了 p[0] 所指文件
已知新描述符是 0，就不必再用 0 接收了
只留标准输入口
p[0] > file

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

分析了为什么每个程序进程只能操作有一个写或读端

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）。)

指实现 pipe 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。)

ADN 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 [inode](#)³，一个 [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 (文件唯一标识)
    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。

索引建立

TCB学习
文件名和
索引建立

`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 关于文件系统的操作都被实现为用户级程序 → 这里的用户级程序指什么调用呢？
~~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）。~~

有一个例外，那就是 `cd`，它是在 shell 中实现的。`cd` 必须改变 shell 自身的当前工作目录。如果 `cd` 作为一个普通命令执行，那么 shell 就会 fork 一个子进程，而子进程会运行 `cd`，`cd` 只会改变子进程的当前工作目录。父进程的工作目录保持原样。

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 `Iseek`），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。

对比：
用户级（已被
shell 调用）
和内置于 shell

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** 方案提供更强的隔离性。

为了实现强隔离，禁止应用程序直接访问敏感的硬件资源，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。
管理内存

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。

supervisor mode 是 RISC-V 中对 kernel mode 的叫法 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。

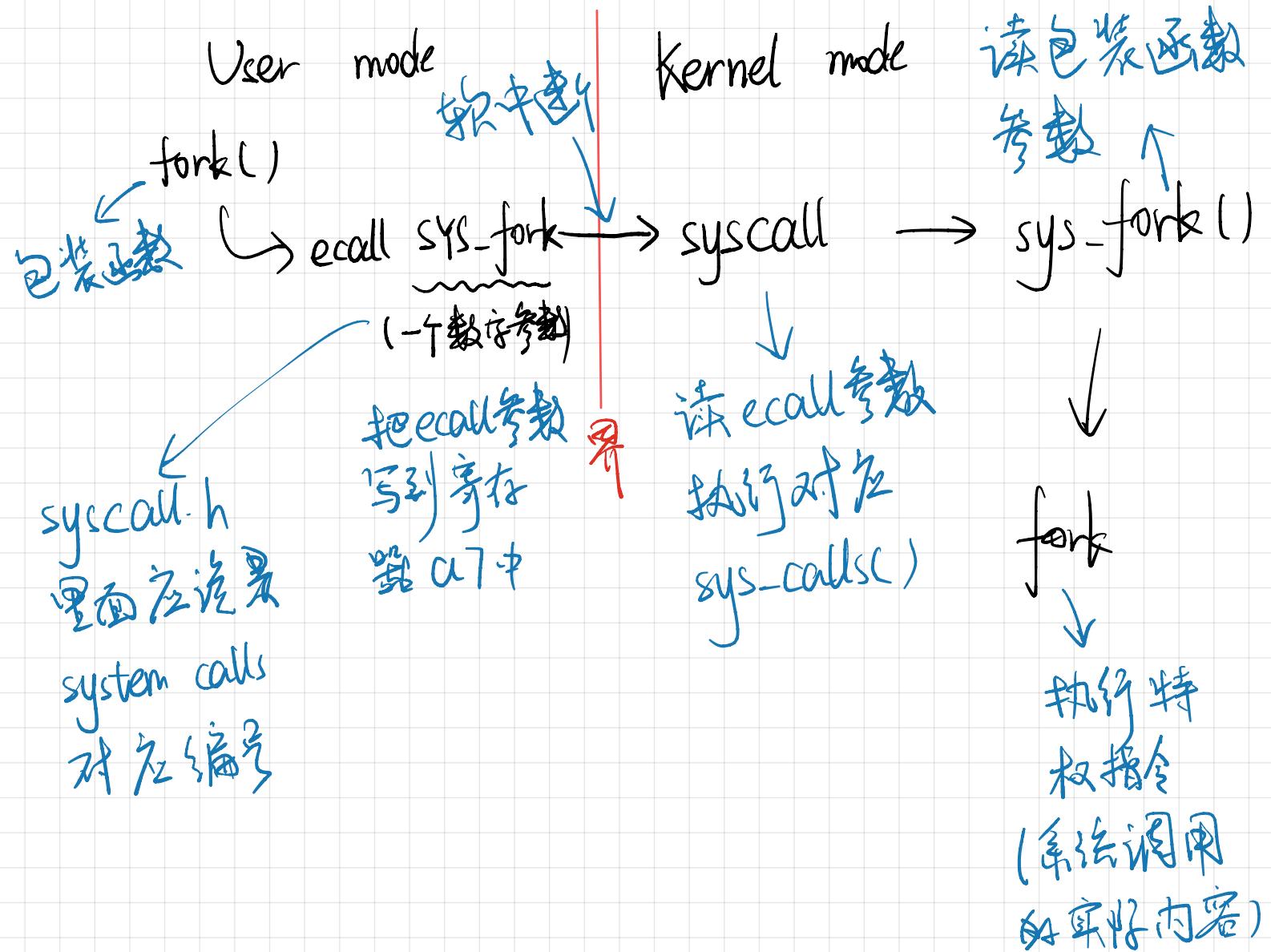
区分运行在
用户/内核
空间的界限
是执行什么
指令（特权
or 普通）

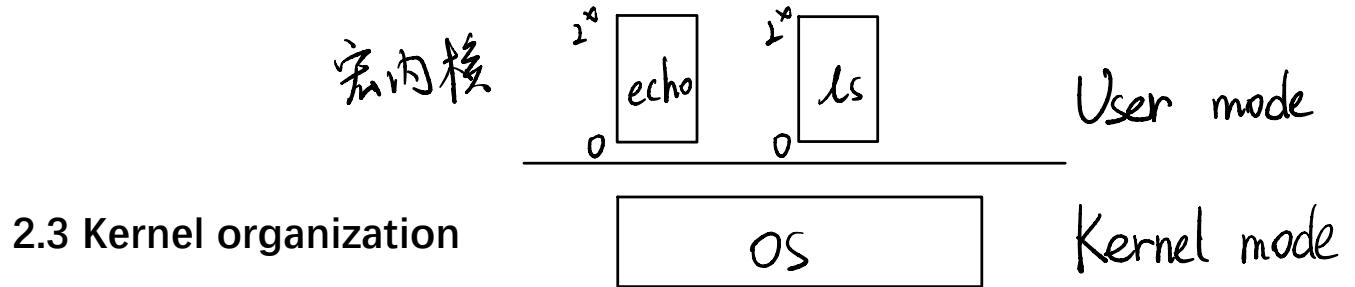
ECALL (number) 一个应用程序如果要调用内核函数（如 xv6 中的读系统调用），必须过渡到内核。（CPU 提供了一个特殊的指令，可以将 CPU 从用户模式切换到监督模式，并在内核指定的入口处进入内核。）(RISC-V 为此提供了 `ecall` 指令。)一旦 CPU 切换到监督者模式，内核就可以验证系统调用的参数，决定是否允许应用程序执行请求的操作，然后拒绝或执行该操作。内核控制监督者模式的入口点是很重要的；如果应用程序可以决定内核的入口点，那么恶意应用程序就能够进入内核，例如，通过跳过参数验证而进入内核。

ecall 触发软中断 → 查找中断向量表 (储存中断处理程序地址) ↓
→ 执行中断处理程序 (其在内核中，完成向 kernel mode 的切换)
(中断处理程序内容就是拟执行的特权指令)

User/Kernel mode 转换

从 User mode 到 Kernel mode
通过 trapframe
 $\rightarrow a_0/a_1/\dots$





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。）

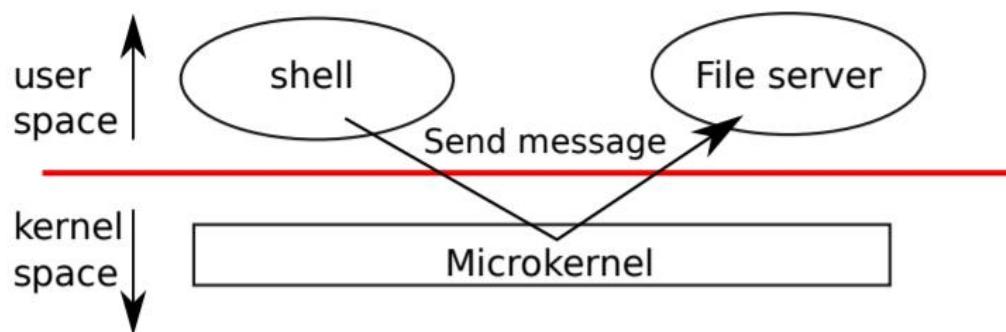


Figure 2.1: A microkernel with a file-system server

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。

文件服务器运行
在用户空间，
那么它如何
使用内核资
源？

这个实验和
“微服务”相
似吗？

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

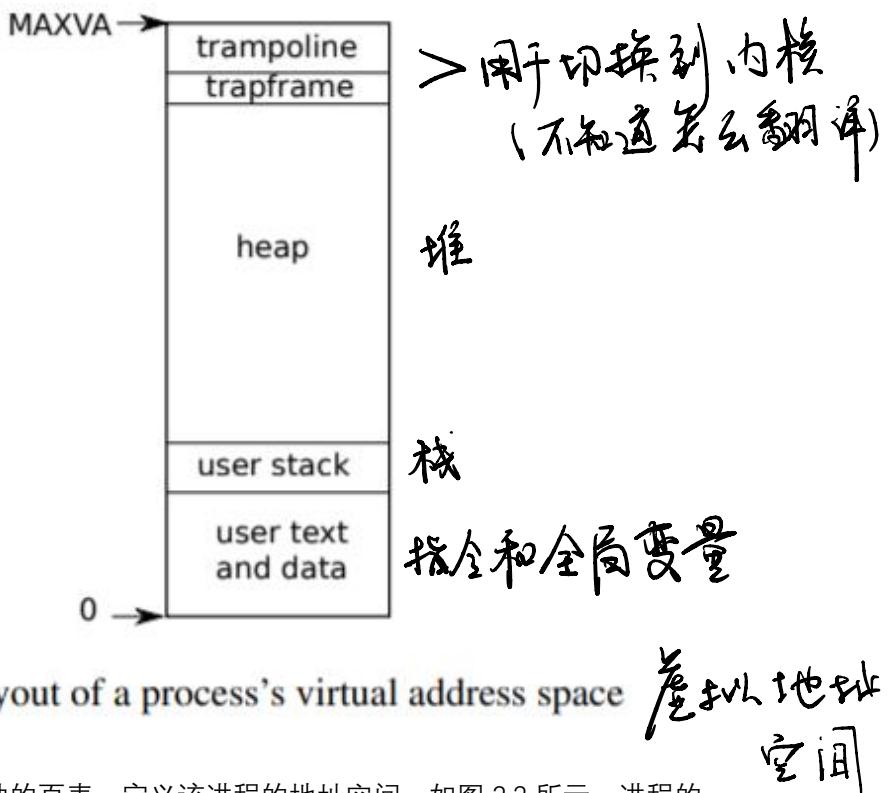


Figure 2.3: Layout of a process's virtual address space

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。

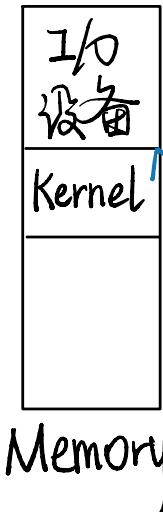
（一个进程可以通过执行 RISC-V **ecall** 指令进行系统调用。）该指令提高硬件权限级别，并将程序计数器改变为内核定义的入口点。入口点的代码会切换到内核栈，并执行实现系统调用的内核指令。当系统调用完成后，内核切换回用户栈，并通过调用 **sret** 指令返回用户空间，降低硬件特权级别，恢复执行系统调用前的用户指令。进程的线程可以在内核中阻塞等

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`。~~ 这条指令最常用来从上一次的调用中返回，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` 中把上一次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。

xv6 内核此时以可执行文件 Kernel (二进制) 形式存在

窗口系统
windowing system

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 *0x3fffffff10e`，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。

for what

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

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

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。

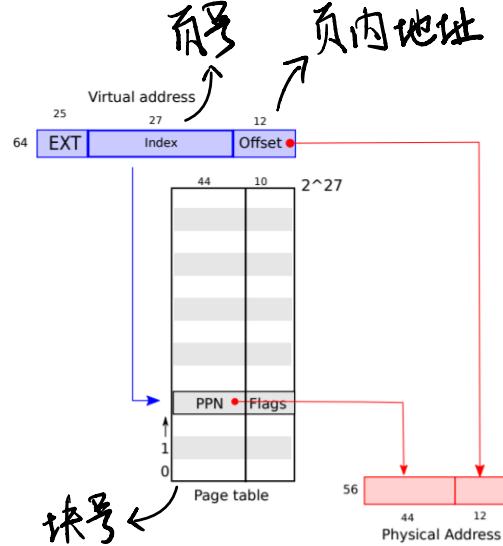


Figure 3.1: RISC-V virtual and physical addresses, with a simplified logical page table.

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 显示了这一切的工作原理。标志位和与页相关的结构体定义在(*kernel/riscv.h*)。

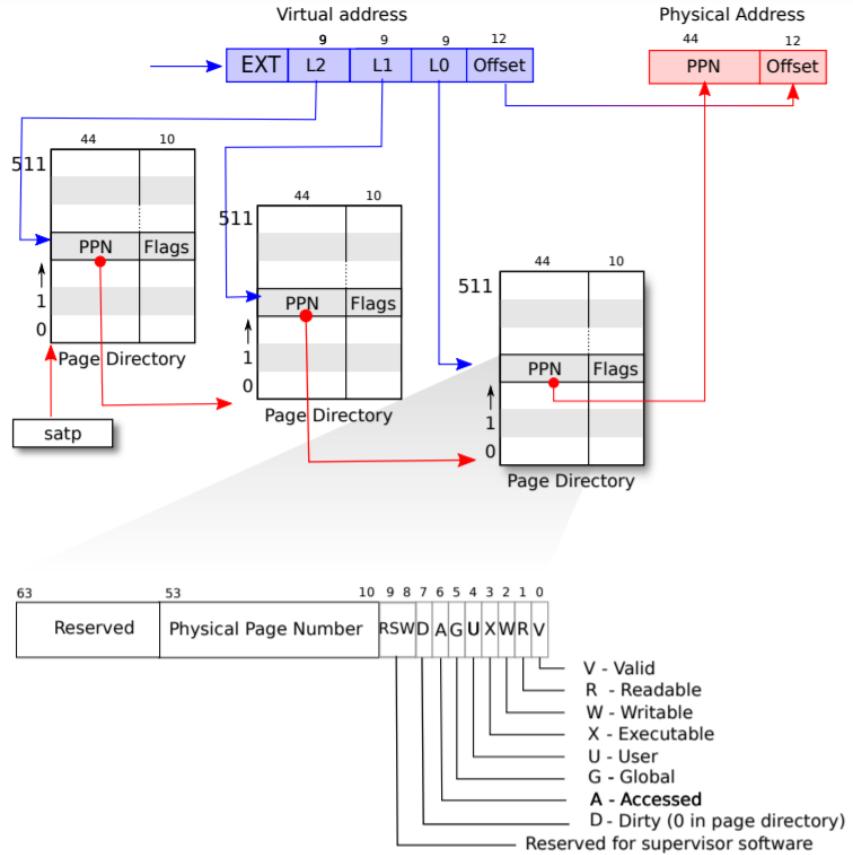


Figure 3.2: RISC-V address translation details.

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。

$\text{MAXVA} = (1L \ll (\beta + \gamma + \delta + 1 - 1))$ 即理论最大虚地址

- 是为了避免对高住虚地址进行符号扩展 (溢出)
(sign-extend)

free memory 用于存放用户进程的页

中断设备

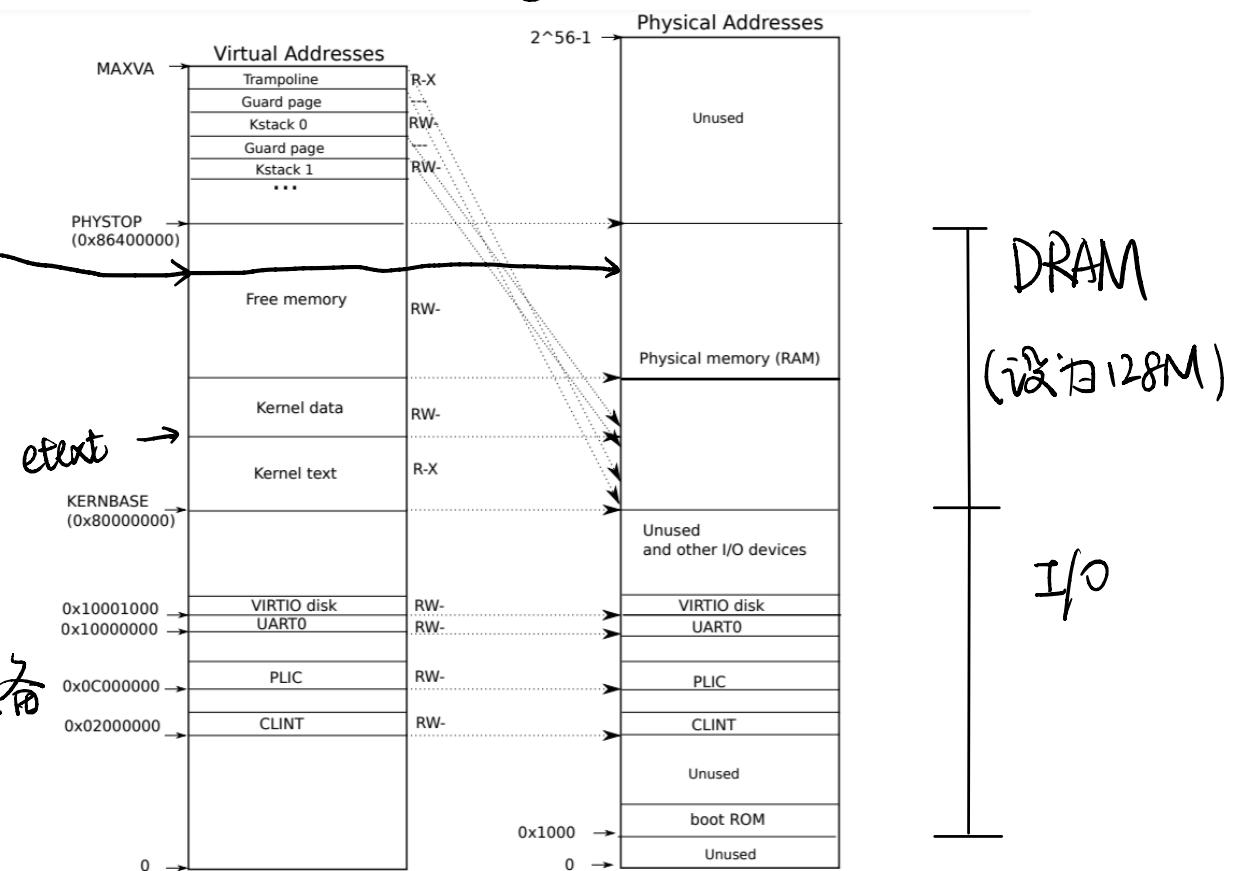


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

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 PHYSSTOP。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。)

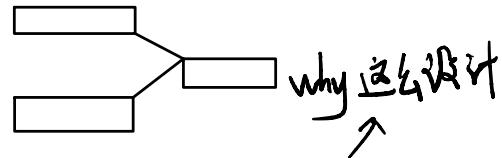
内核使用“直接映射”RAM 和 *内存映射*设备寄存器, 也就是在虚拟地址上映射硬件资源, 这些地址与物理地址相等。例如, 内核本身在虚拟地址空间和物理内存中的位置都是 **KERNBASE=0x80000000**。直接映射简化了读/写物理内存的内核代码。例如, 当 fork 为子进程分配用户内存时, 分配器返回该内存的物理地址; fork 在将父进程的用户内存复制到子进程时, 直接使用该地址作为虚拟地址。

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, 但我们在那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。
→ kstack → Guard page

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, 很可能会引起异常, 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

可复用? 强
化不足?



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(可执行程序的代码段)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);)

译源码

3.3 Code: creating an address space [创建(内核)地址空间]

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 **kvminit hart** (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。**Kvmmap** 栈的虚拟地址映射到申请的物理内存上，然后调用 **kvminit hart** 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

→ kstack 在 kernel 地址
空间中
→ 快表

flush

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 `kvminit hart` 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)`。

避免用户空间可能通过 TLB
访问内核空间

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。
`kmem{freelist}` (`kalloc.c`)

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。

自旋锁:
循环查询

`main` 函数调用 `kinit` 来初始化分配器(`kernel/kalloc.c:27`)。`kinit` 初始化空闲页链表, 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` 将页面预存入空闲列表: 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。

↑ 取用空闲页

3.6 Process address space

进程地址空间

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。

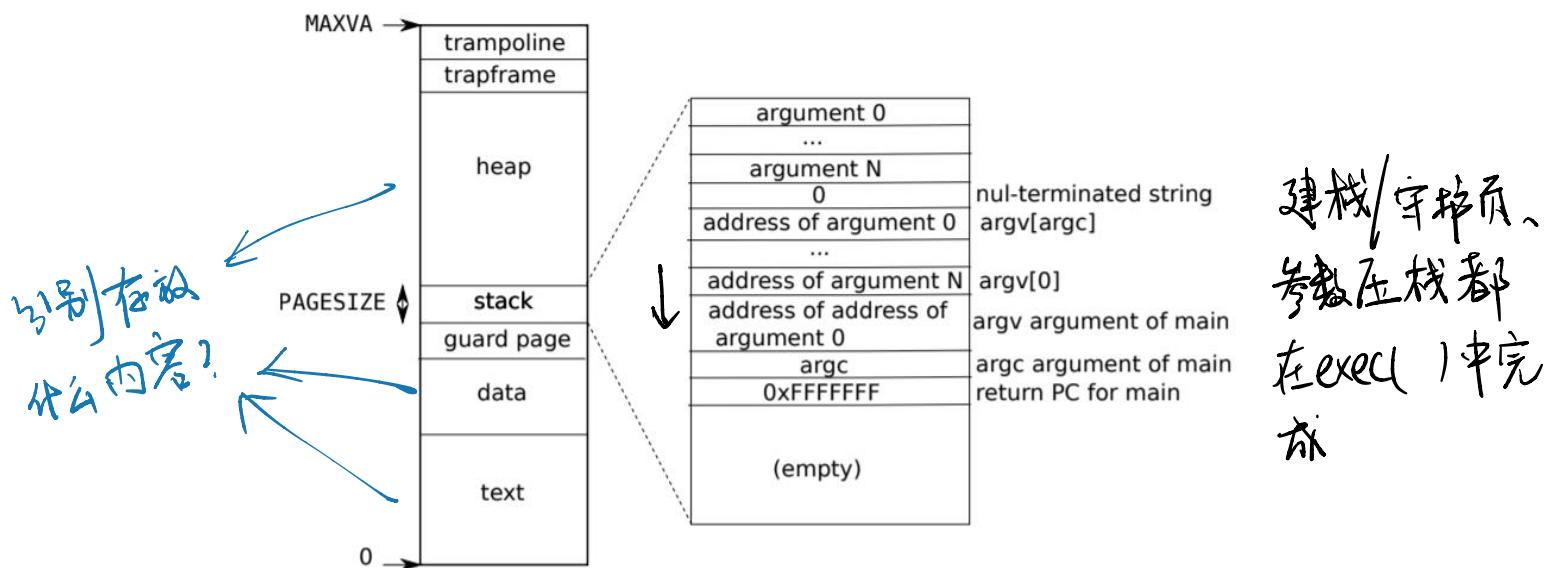


Figure 3.4: A process's user address space, with its initial stack.

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

内存缩减发生在进程用户地址空间的堆 heap? 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。



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(char *path, char **argv)*

`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 0x000000000000840 memsz 0x0000000000000858 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。

程序节头 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 `uvmalloc` 分配了足够的物理内存来容纳 2136 字节，但只从文件/`init` 中读取 2112 字节。

guard page

`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。一旦映像完成，`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。

(历史)

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

本章描述了RV32和RV64程序的C编译器标准和两个调用约定：附加标准通用扩展（RV32G/RV64G）的基础ISA约定，以及缺乏浮点单元（例如RV32I/RV64I）实现的软浮点约定。

使用ISA扩展的实现可能需要扩展调用约定。

18.1 C语言的数据类型和对齐方式

表18.1总结了RISC-V C程序本机支持的数据类型。在RV32和RV64 C编译器中，C中的int类型都是32位。另一方面，long和指针都与整数寄存器位数一致，所以在RV32中，两者都是32位，而在RV64中，两者都是64位。同样，RV32采用ILP32整数模型，而RV64是LP64。在RV32和RV64中，C类型long long是64位整数，float是遵循IEEE754-2008标准的32位浮点数，double是遵循IEEE754-2008标准的64位浮点数，long double是遵循IEEE754-2008标准的128位浮点数。

char被实现为un char

C类型char和unsigned char都是8位无符号整数，当存储在RISC-V整数寄存器中时是零扩展。unsigned short是16位无符号整数，当存储在RISC-V整数寄存器中时是零扩展。signed char是8位有符号整数，当存储在RISC-V整数寄存器中时是符号扩展的，即比特位从(XLEN-1)到7都是相等的。short是16位有符号整数，当存储在寄存器中时是符号扩展的。

在RV64中，32位的数据类型（如int）以合适的符号扩展存储在整数寄存器中；也就是说，比特位从63到31都是相等的。即使是无符号的32位类型，这个限制也适用。

RV32和RV64 C编译器和兼容软件将所有上述数据类型存储在内存中时保持自然对齐。

C数据类型	描述	RV32中字节数	RV64中字节数
char	字符值/字节	1	1
short	短整型	2	2
int	整型	4	4
long	长整型	4	8
long long	超长整型	8	8
void*	指针	4	8
float	单精度浮点型	4	4
double	双精度浮点型	8	8

C数据类型	描述	RV32中字节数	RV64中字节数
long double	扩展精度浮点型	16	16

表18.1：基于RISC-V指令集的C编译器数据类型

18.2 RVG调用协定

RISC-V调用约定尽可能在寄存器中传递参数。为此，最多使用八个整数寄存器[a0-a7](#)和八个浮点寄存器[fa0-fa7](#)。

如果函数的参数被概念化为C结构体的字段，结构体中的每个字段都按指针长度对齐，则参数寄存器是该结构体中前八个指针字长参数的副本。如果第*i*(*i*<8)个参数是浮点类型，则在浮点寄存器fa*i*中传递；否则，在整数寄存器ai*i*中传递。但是，浮点参数如果属于union或结构体中数组字段的一部分，就会在整数寄存器中传递。此外，变参函数的浮点参数（未显式命名参数列表的函数）在整数寄存器中传递。

小于指针字长的参数在参数寄存器的最低有效位(LSB)中传递。相应地，栈上传递的小于指针字长的参数出现在指针字的较低地址中，因为RISC-V有一个小端存储系统。

当在堆栈上传递两倍于指针字大小的基本参数时，它们是自然对齐的。当它们在整数寄存器中传递时，它们驻留在对齐的偶数号-奇数号寄存器对中，偶数寄存器保存最低有效位。例如，在RV32中，函数void foo(int, long long)的第一个参数在a0中传递，第二个参数在a2和a3中传递。a1中不传递任何内容。

大于指针字大小两倍的参数通过引用传递。

结构体中未在参数寄存器中传递的部分在栈上传递。栈指针sp指向未在寄存器中传递的第一个参数。

函数在整数寄存器a0和a1以及浮点寄存器fa0和fa1中返回值。只有当浮点值是原始值（传入时fa0和fa1作为参数寄存器，原始值是指该参数不改变而直接返回）或作为仅有一两个浮点值组成的结构体的成员时，才会从浮点寄存器中返回。长度恰好为两个指针字长的其他返回值将在a0和a1中返回。较大的返回值完全在内存中传递；调用方分配此内存区域，并将指针作为隐式的第一个参数传递给被调用方。

在标准的RISC-V调用约定中，栈向下增长，栈指针始终保持16字节对齐。

除了自变量和返回值寄存器之外，还有在调用中不稳定的七个整数寄存器t0-t6和十二个浮点寄存器ft0-ft11作为临时寄存器，如果之后使用，调用者必须保存它们。十二个整数寄存器s0-s11和十二个浮点寄存器fs0-fs11在调用中受保护，如果使用，被调用者必须保存它们。表18.2显示了调用约定中每个整数和浮点寄存器的作用。

寄存器	ABI名称	描述	保存者
x0	zero	硬布线零	
x1	ra	返回地址	调用者

the diff between
callee-saved &
caller-saved

The diagram illustrates the RISC-V calling convention. It shows a table of registers and their attributes, with handwritten annotations. Brackets on the left group registers x2-x7 and f0-f7 as 'callee-saved'. Brackets on the right group registers x8-x17 and f8-f17 as 'caller-saved'. The table columns are: 寄存器 (Register), ABI名称 (ABI Name), 描述 (Description), and 保存者 (Saver).

寄存器	ABI名称	描述	保存者
x2	sp	栈指针	被调用者
x3	gp	全局指针	
x4	tp	线程指针	
x5-7	t0-2	临时暂存单元	调用者
x8	s0/fp	保留寄存器/帧指针	被调用者
x9	s1	保留寄存器	被调用者
x10-11	a0-1	函数参数/返回值	调用者
x12-17	a2-7	函数参数	调用者
x18-27	s2-11	保留寄存器	被调用者
x28-31	t3-6	临时暂存单元	调用者
f0-7	ft0-7	浮点临时暂存单元	调用者
f8-9	fs0-1	浮点保留寄存器	被调用者
f10-11	fa0-1	浮点参数/返回值	调用者
f12-17	fa2-7	浮点参数	调用者
f18-27	fs2-11	浮点保留寄存器	被调用者
f28-31	ft8-11	浮点临时暂存单元	调用者

表18.2 RISC-V调用协定寄存器的使用

18.3 软浮点数调用协定

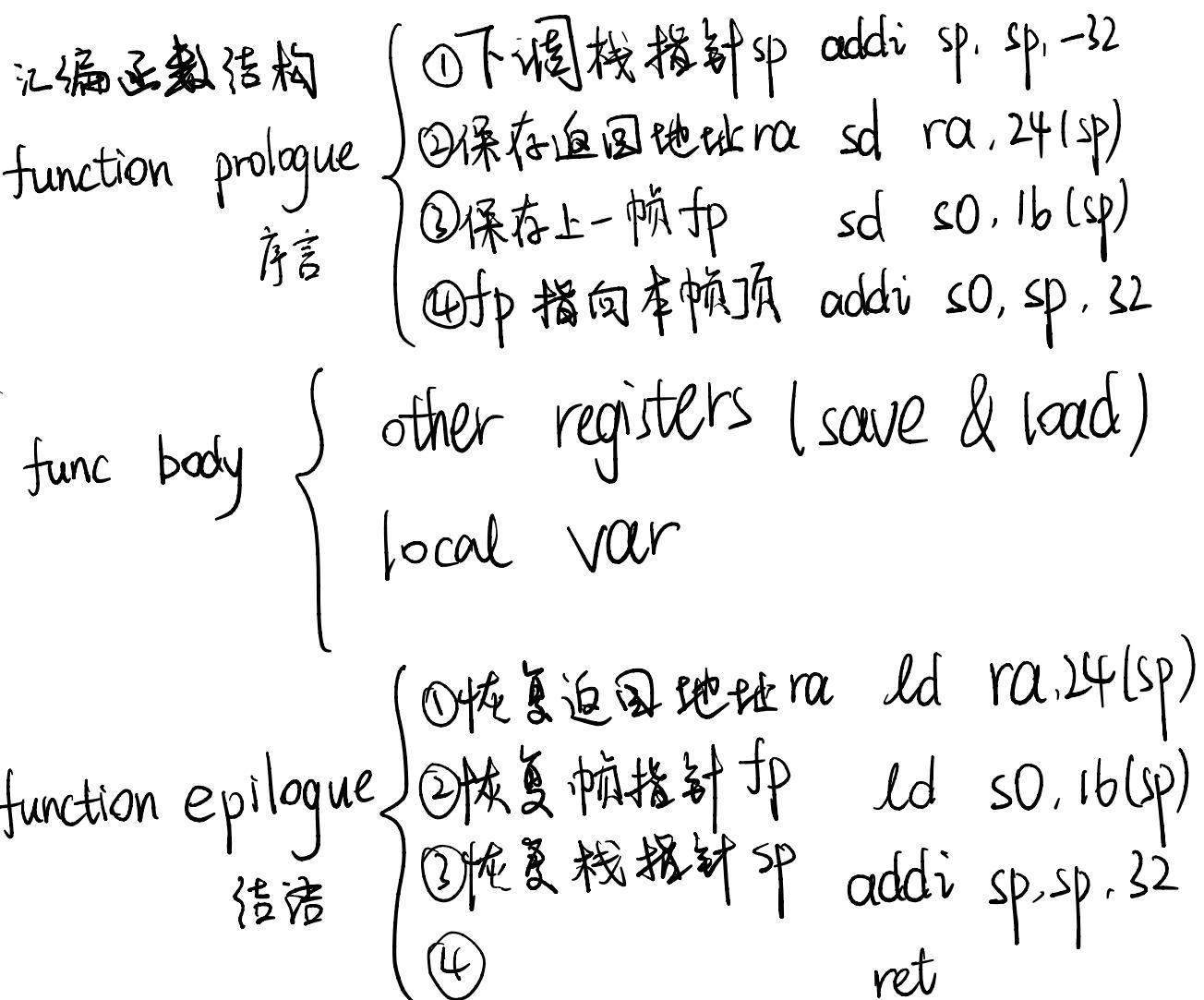
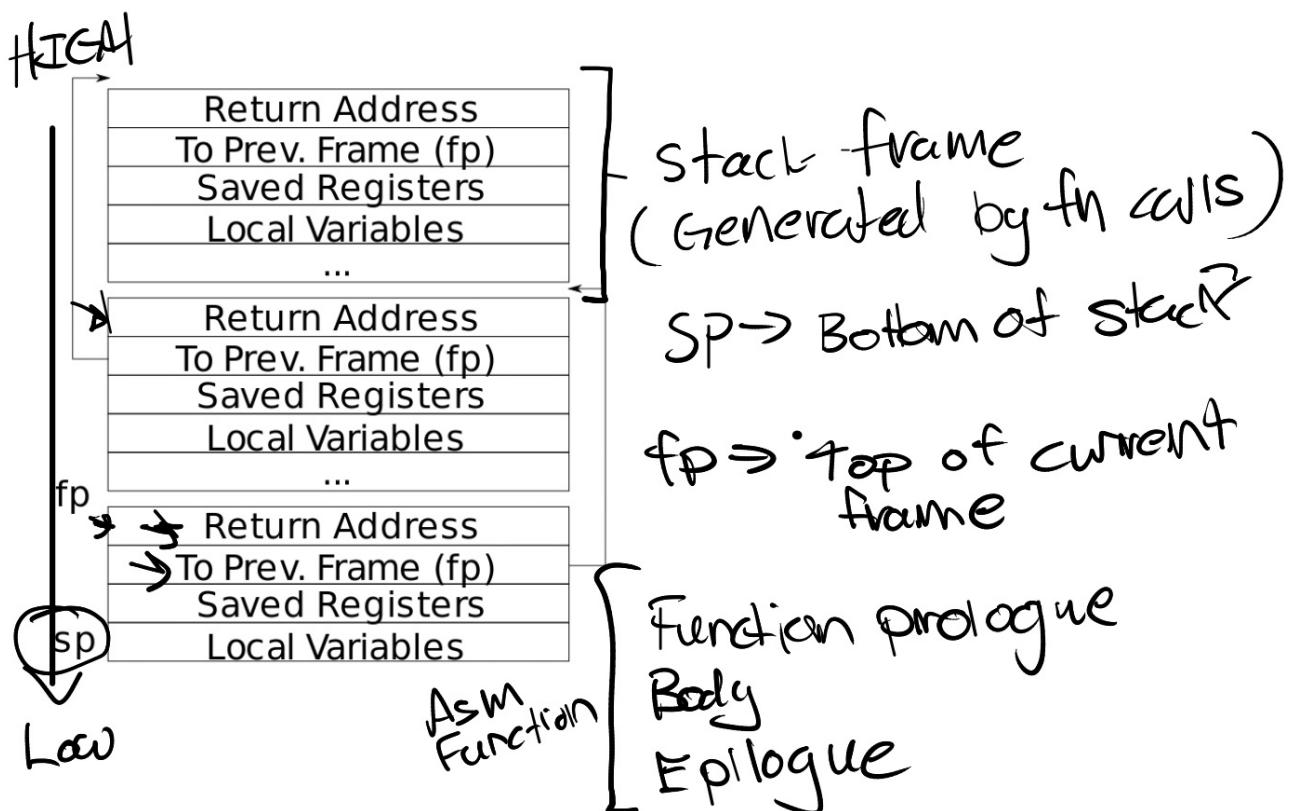
软浮点调用约定用于缺乏浮点硬件的RV32和RV64实现。它避免使用了F、D和Q标准扩展中的所有指令，从而避免使用f寄存器。

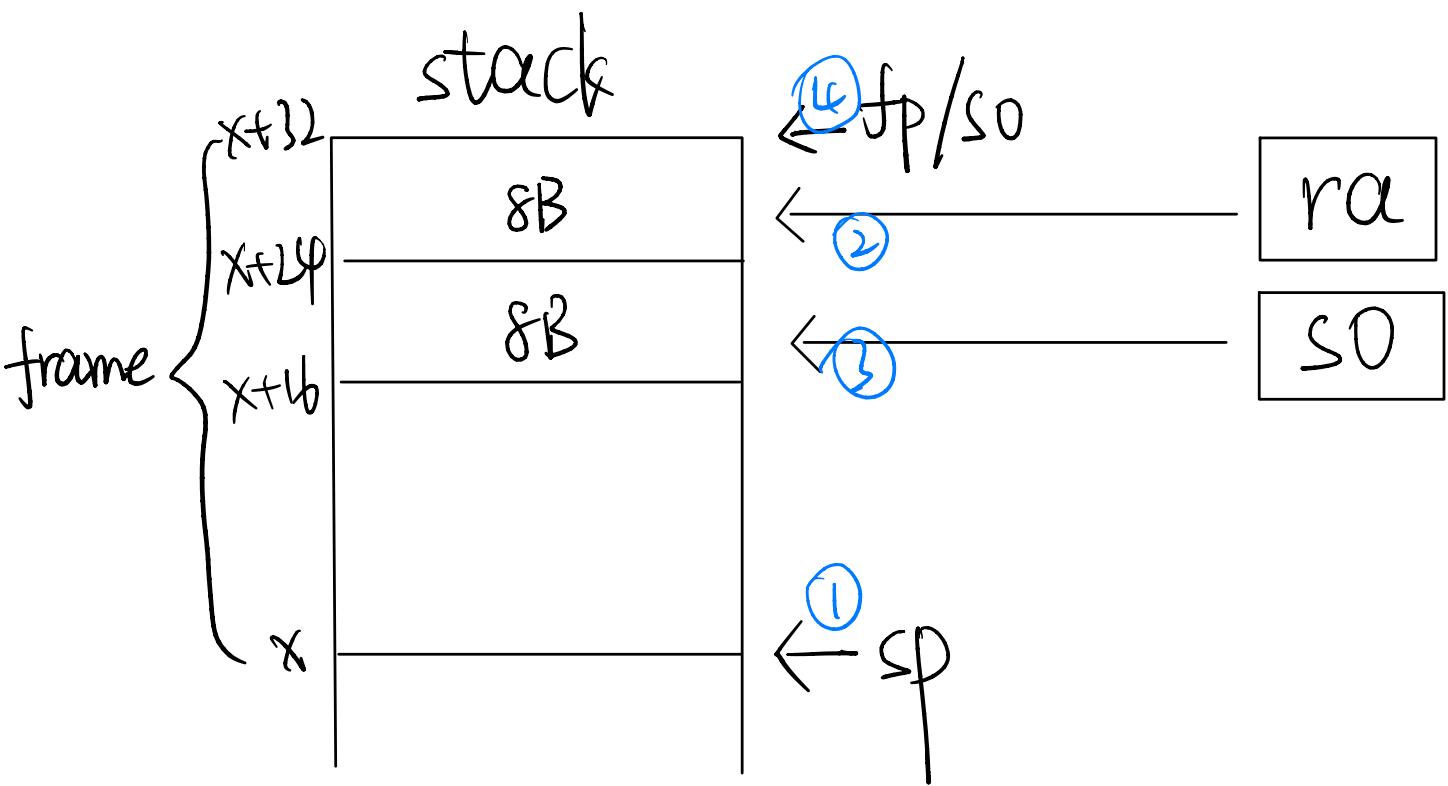
完整参数的传递和返回方式与RVG约定相同，栈规则也相同。浮点参数使用长度相同的整型参数的规则在整数寄存器中传递和返回。例如，在RV32中，函数double `foo(int, double, long double)`的第一个参数在a0中传递，第二个参数在a2和a3中传递，第三个参数通过a4传递引用；其结果在a0和a1中返回。在RV64中，参数以a0、a1和a2-a3对形式传递，结果以a0形式返回。

动态舍入模式和累计异常标志可以通过C99头文件 `fenv.h` 提供的程序访问。

注：为了编写高精度浮点数的运算，编程人员需要控制浮点数环境的各个方面：结果如何舍入，浮点数表达式如何简化与变换，如何处理浮点数异常（如下溢之类的浮点数异常是忽略还是产生错误）等等。C99引入了 `fenv.h` 来控制浮点数环境。

the Stack





{ ra 在棧是 caller-saved
 so 在棧是 callee-saved

参考: <https://danielmangum.com/posts/risc-v-bytes-caller-callee-registers/>

第四章 陷阱指令和系统调用

有三种事件会导致中央处理器搁置普通指令的执行，并强制将控制权转移到处理该事件的特殊代码上。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些什么时；另一种情况是 异常：（用户或内核）指令做了一些非法的事情，例如除以零或使用无效的虚拟地址；第三种情况是 设备中断，一个设备，例如当磁盘硬件完成读或写请求时，向系统表明它需要被关注。

本书使用陷阱（trap）作为这些情况的通用术语。通常，陷阱发生时正在执行的任何代码都需要稍后恢复，并且不需要意识到发生了任何特殊的事情。也就是说，我们经常希望陷阱是透明的；这对于中断尤其重要，中断代码通常难以预料。（通常的顺序是陷阱强制将控制权转移到内核；内核保存寄存器和其他状态，以便可以恢复执行；内核执行适当的处理程序代码（例如，系统调用接口或设备驱动程序）；内核恢复保存的状态并从陷阱中返回；原始代码从它停止的地方恢复。）

`xv6` 内核处理所有陷阱。这对于系统调用来说是顺理成章的。由于隔离性要求用户进程不直接使用设备，而且只有内核具有设备处理所需的状态，因而对中断也是有意义的。因为 `xv6` 通过杀死违规程序来响应用户空间中的所有异常，它也对异常有意义。

`xv6` 陷阱处理分为四个阶段：RISC-V CPU 采取的硬件操作、为内核 C 代码执行而准备的汇编程序集“向量”、决定如何处理陷阱的 C 陷阱处理程序以及系统调用或设备驱动程序服务例程。（虽然三种陷阱类型之间的共性表明内核可以用一个代码路径处理所有陷阱，但对于三种不同的情况：来自用户空间的陷阱、来自内核空间的陷阱和定时器中断，分别使用单独的程序集向量和 C 陷阱处理程序更加方便。）

(硬件操作)

4.1 RISC-V陷入机制

CSR: 控制与状态寄存器
control status

每个RISC-V CPU都有一组控制寄存器，(内核通过向这些寄存器写入内容来告诉CPU如何处理陷阱，内核可以读取这些寄存器来明确已经发生的陷阱。) RISC-V文档包含了完整的内容。**riscv.h**(**kernel/riscv.h:1**)包含在xv6中使用到的内容的定义。以下是最重要的一些寄存器概述：

- **stvec**: 内核在这里写入其陷阱处理程序的地址；RISC-V跳转到这里处理陷阱。
- **sepc**: 当发生陷阱时，RISC-V会在这里保存程序计数器**pc**（因为**pc**会被**stvec**覆盖）。**sret**（从陷阱返回）指令会将**sepc**复制到**pc**。内核可以写入**sepc**来控制**sret**的去向。
- **scause**: RISC-V在这里放置一个描述陷阱原因的数字。
- **sscratch**: 内核在这里放置了一个值，这个值在陷阱处理程序一开始就会派上用场。
- **sstatus**: 其中的**SIE**位控制设备中断是否启用。如果内核清空**SIE**，RISC-V将推迟设备中断，直到内核重新设置**SIE**。**SPP**位指示陷阱是来自用户模式还是管理模式，并控制**sret**返回的模式。

上述寄存器都用于在管理模式下处理陷阱，在用户模式下不能读取或写入。在机器模式下处理陷阱有一组等效的控制寄存器，xv6仅在计时器中断的特殊情况下使用它们。

把S换成m

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

当需要强制执行陷阱时，RISC-V硬件对所有陷阱类型（计时器中断除外）执行以下操作：

1. 如果陷阱是设备中断，并且状态**SIE**位被清空，则不执行以下任何操作。
2. 清除**SIE**以禁用中断。（操作PC需要中断）
3. 将**pc**复制到**sepc**。
4. 将当前模式（用户或管理）保存在状态的**SPP**位中。
5. 设置**scause**以反映产生陷阱的原因。中断类型码
6. 将模式设置为管理模式。S-MODE
7. 将**stvec**复制到**pc**。
8. 在新的**pc**上开始执行。何时开中断（是否允许多级中断）

请注意，CPU不会切换到内核页表，不会切换到内核栈，也不会保存除**pc**之外的任何寄存器。内核软件必须执行这些任务。CPU在陷阱期间执行尽可能少量工作的一个原因是为软件提供灵活性；例如，一些操作系统在某些情况下不需要页表切换，这可以提高性能。

你可能想知道CPU硬件的陷阱处理顺序是否可以进一步简化。例如，假设CPU不切换程序计数器。那么陷阱可以在仍然运行用户指令的情况下切换到管理模式。但因此这些用户指令可以打破用户/内核的隔离机制，例如通过修改**satp**寄存器来指向允许访

问所有物理内存的页表。因此，CPU使用专门的寄存器切换到内核指定的指令地址，即 **stvec**，是很重要的。

管理模式特权：

1. 读写CSR
2. 使用页表中 $PTE_V=0$ 的项



Kazurin Lv1

2020年11月06日 阅读 856

关注

RISC-V 特权指令集入门

CSR

CSR 是支撑 RISC-V 特权指令集的一个重要概念。CSR 的全称为 **控制与状态寄存器** (control and status registers)。

简单来说，CSR 是 CPU 中的一系列特殊的寄存器，这些寄存器能够反映和控制 CPU 当前的状态和执行机制。在 RISC-V 特权指令集手册中定义的一些典型的 CSR 如下：

- `misa`，反映 CPU 对于 RISC-V 指令集的支持情况，如 CPU 所支持的最长的位数 (32 / 64 / 128) 和 CPU 所支持的 RISC-V 扩展。
- `mstatus`，包含很多与 CPU 执行机制有关的状态位，如 `MIE` 是否开启 M-mode 中断等。

CSR 的地址空间有 12 位，因此理论上能够支持最多 4,096 个 CSR。但实际上，这个地址空间大部分是空的，RISC-V 手册中实际只定义了数十个 CSR。访问不存在的 CSR 将触发无效指令异常。关于该地址空间的分配，请参考 RISC-V 特权指令集手册的 2.2 *CSR Listing* 节。

操作 CSR 的指令在 RISC-V 的 `Zicsr` 扩展模块中定义。包括伪指令在内，共有以下 7 种操作类型：

1. `csrr`，读取一个 CSR 的值到通用寄存器。如：`csrr t0, mstatus`，读取 `mstatus` 的值到 `t0` 中。
2. `csrw`，把一个通用寄存器中的值写入 CSR 中。如：`csrw mstatus, t0`，将 `t0` 的值写入 `mstatus`。
3. `csrs`，把 CSR 中指定的 bit 置 1。如：`csrsi mstatus, (1 << 2)`，将 `mstatus` 的右起第 3 位置 1。
4. `csrc`，把 CSR 中指定的 bit 置 0。如：`csrci mstatus, (1 << 2)`，将 `mstatus` 的右起第 3 位置 0。
5. `csrrw`，读取一个 CSR 的值到通用寄存器，然后把另一个值写入该 CSR。如：`csrrw t0, mstatus, t0`，将 `mstatus` 的值与 `t0` 的值交换。
6. `csrrs`，读取一个 CSR 的值到通用寄存器，然后把该 CSR 中指定的 bit 置 1。
7. `csrrc`，读取一个 CSR 的值到通用寄存器，然后把该 CSR 中指定的 bit 置 0。

这些指令都有 R 格式和 I 格式，I 格式的指令名需要在 R 格式的指令名之后附加字母 `i`，如 R 格式指令 `csrr` 对应的 I 格式指令为 `csrri`。具体的指令格式和执行机制请参考 RISC-V 非特权指令手册的 `Zicsr` 节。

前 4 种操作 `csrr` / `csrw` / `csrs` / `csrc` 是伪指令，这些指令会由汇编器翻译成对应的 `csrrw` / `csrrs` / `csrrc` 指令。这样做是为了减少 CPU 需要实现的指令数量，使 CPU 的片上面积利用更高效。具体请参考 RISC-V 非特权指令集手册的 *RISC-V Assembly Programmer's Handbook* 节。

从 `mscratch` CSR 中读出并写入一个值的示例汇编代码如下：

关于作者



Kazurin Lv1

获得点赞 7

文章被阅读 2,394



下载手机APP

一个帮助开发者成长的社区

找对——
属于你的
技术圈子



回复‘进群’加入掘金
微信交流群

相关文章

[Wasm介绍之2：指令集和栈](#)

1 0

[1.2W字 | 了不起的 TypeScript 入门教程](#)

2319 112

[Nginx 从入门到实践，万字详解！](#)

2067 88

[Vue3.0 前的 TypeScript 最佳入门实践](#)

1957 113

[前端必会的 Nginx入门视频教程\(共11集\)](#)

2526 129

目录

• [CSR](#)

• [四种特权模式](#)

• [中断和异常](#)

- RISC-V 中断处理程序
- 其它模式下的中断，及跳转到...

• [可由平台定义的部分](#)

• [可由平台定义的部分](#)

```
csrr    t0, mscratch  
addi    t0, t0, 1  
csrwr   mscratch, t0
```

as 复制代码

四种特权模式

类似于 x86 中的特权模式，RISC-V 特权指令集中也定义了 4 种特权模式（参考 RISC-V 特权指令集手册的 1.2 *Privilege Levels* 节）。它们的名字和代号如下：

- Machine mode (M-mode)，序号为 3；
- Hypervisor mode (H-mode)，序号为 2；
- Supervisor mode (S-mode)，序号为 1；
- User mode (U-mode)，序号为 0。

其中，权限最高的模式为 M-mode，它拥有对机器底层的一切访问。通常来说，M-mode 中运行的代码必须是可信代码。M-mode 可以用于管理机器内的安全执行环境（手册原文：secure execution environments）。

H-mode 用于尚未完成的 RISC-V 硬件虚拟化 (hypervisor) 扩展。虚拟机管理程序将运行在该特权模式下。在现行（2019 年 6 月）的 RISC-V 特权指令集手册稳定版本中，该特权模式是保留 (reserved) 的。

相应地，S-mode 和 U-mode 分别划分给操作系统和用户程序使用。

通常，一个 CPU 需要实现 M, S, U 三种模式，才能够运行完整的类 Unix 操作系统（如 Linux）。

与 x86 不同，RISC-V 并没有提供一种机制来获取当前的特权模式。这是因为 RISC-V 设计者认为，一段代码在编写的时候，就应该能知道它的目标特权模式是什么，因此在代码中手动获取当前特权模式是没有必要的。设计这样一种机制会造成 CSR 状态位冗余。

RISC-V 规范中只提供了一种在不同特权模式间切换的方法，即借助中断返回机制。具体请参考下一章节。

中断和异常

在 RISC-V 特权指令集手册中，能够引起当前程序中断，并使得 CPU 转而执行特定代码的事件统称为 **陷阱** (trap)。陷阱共分为两大类：**中断** (interrupt) 和 **异常** (exception)，其中：

- **中断** 通常指由外部设备引发的事件（当然，也有一类中断是软件中断），具有 **异步处理** 的特点；
- **异常** 指的是 CPU 内部在执行程序时产生的异常事件，具有 **同步处理** 的特点。

(注：以上分类存在一定争议，如 OSDev.org 就对以上的名词有 [不同解释](#)，认为 **陷阱** 指的是在产生异常时，CPU 同步地跳转到异常处理程序的动作。以上分类仅代表作者本人观点。)

RISC-V 中的 **中断** 又分为 3 种主要类型：

- **软件中断**，指不是由外部设备引起的，而是通过软件写入 `mip` CSR 触发的中断。
- **时钟中断**。

- **外部中断**, 指受 RISC-V 平台级中断控制器 (Platform Level Interrupt Controller, PLIC) 控制的, 由外部设备触发的中断。

不将时钟中断归类为外部中断的原因是, 时钟中断不受 PLIC 的控制, 而是由两个独立的寄存器 `mtime` 和 `mtimecmp` 控制。时钟中断也是唯一一种由外部设备引发的, 且在 RISC-V 特权指令集手册中有明确定义的中断。

除时钟中断之外, 由外部设备引发的中断都由 RISC-V 的 PLIC 控制。PLIC 负责捕获中断请求, 按优先级顺序发送给 CPU; 并向 CPU 提供一系列的硬件寄存器, 供程序查询中断相关的信息。

(注 2: 下文中仍根据 x86 的约定, 将 RISC-V 的中断和异常统称为 **中断**, RISC-V 中的中断称为 **外部中断**。)

RISC-V 中断处理程序

在 RISC-V CPU 中, 外部中断必须通过 CSR 手动开启, 否则 CPU 不会接收到任何中断 (除了 non-maskable interrupts, 不可屏蔽中断)。但 CPU 异常只要在异常条件满足时就会触发, 并且无法通过 CSR 屏蔽。

开启中断共需要经过两个步骤, 其中 `mstatus[MIE]` 是中断总开关 (MIE 是 machine interrupt enabled 的缩写), `mie` CSR 是针对每种中断类型的独立开关。只有当两个 CSR 都正确设置时, 才能够触发中断。

中断响应程序的地址需要放在 `mtvec` CSR 中。目前 RISC-V 支持两种类型的中断向量:

- 直接模式 (direct), 所有类型的中断均发送给同一个中断响应程序。
- 向量化模式 (vectored), **外部中断** 将根据 **中断类型** 发送给不同的中断响应程序, 但所有的 **异常** 仍然发送给 **同一个** 异常响应程序。

当有中断发生时, CPU 会做 (包括但不限于) 以下工作:

- 将 **发生异常的指令 或 被中断时的下一条指令** 的 PC 地址放入 `mepc` CSR;
- 将中断类型码放入 `mcause` CSR;
- 如果中断带有附加信息 (如: 非法内存访问时的内存地址), 则该信息将放入 `mtval` CSR;
- 如果是外部引发的中断, 令 `mstatus[MPIE] = mstatus[MIE]`, 并令 `mstatus[MIE] = 0`, 相当于在进入中断响应程序前暂时关闭中断。PIE 是 previous interrupt enabled 的缩写。
- 将当前特权模式代号放入 `mstatus[MPP]` 中, 并将当前模式设为 M-mode;
- 根据 `mtvec` CSR 的值, 决定中断响应程序的地址, 并跳转到该地址。

默认情况下, 所有的中断均在 M-mode 下处理, 无论当前的特权模式是什么。但 M-mode 也可以配置中断**委托** (delegation), 将部分中断直接转发给低权限的模式, 如 S-mode 和 U-mode。这时, 相应的 CSR 的名称则变为 `sepc` / `uepc` 等。有关中断委托的细节, 可以参考 RISC-V 特权指令集手册的 3.1.8 Machine Trap Delegation Registers (`medeleg` and `mideleg`) 节。

M-mode 的中断响应程序通过 `mret` 指令从中断中返回。该指令会进行以下操作:

- 将当前特权模式设置为 `mstatus[MPP]`;
- 令 `mstatus[MIE] = mstatus[MPIE]`, 以还原发生中断前的中断开关;
- `mstatus[MPIE] = 1`;
- `mstatus[MPP]` 将被设为 U (如果 CPU 不支持 U-mode, 则设为 M) ;
- 将 PC 的值设为 `mepc` 的值, 以返回中断前的程序。

具体流程请参考 RISC-V 特权指令集手册的 3.1.6.1 *Privilege and Global Interrupt-Enable Stack in `mstatus` register* 和 3.2.2 *Trap-Return Instructions* 节。

其它模式下的中断，及跳转到低权限模式的方法

S-mode 下的中断和中断返回的机制与 M-mode 有所不同，不详细介绍。具体请参考 RISC-V 特权指令集手册的 4 *Supervisor-Level ISA* 章节。

RISC-V 标准中提供的唯一一种从 M-mode 跳转到低权限模式的方法是：将 `mstatus[MPP]` 设为要跳转的模式，将 `mepc` 设为跳转的目标地址，然后执行 `mret`。

可由平台定义的部分

RISC-V 标准给予了芯片设计者很大的自由发挥的空间。如：

- 物理内存属性 (physical memory attributes, PMAs)：RISC-V 的内存地址空间并不一定要与物理内存一一对应。芯片设计者可以自行决定芯片的物理内存布局，称为 物理内存属性 (PMA)。
- CPU 重置 (reset)：CPU 启动后的 PC 地址可以由芯片设计者定义。
- 不可屏蔽中断 (non-maskable interrupts, NMI)：RISC-V 标准中没有定义任何具体的 NMI 类型，可以由芯片设计者自行定义。NMI 的中断响应程序地址也是由芯片设计者自行定义的。
- 调试模式 (debug mode, D-mode)：拥有比 M-mode 更高的权限，供芯片调试使用。具体行为可由芯片设计者定义。

文章分类 阅读 文章标签  操作系统



Kazurin 
发布了 5 篇文章 · 获得点赞 7 · 获得阅读 2,394





安装掘金浏览器插件
多内容聚合浏览、多引擎快捷搜索、多工具便捷提效、多模式随心畅享，你想要的，这里都有！





输入评论 (Enter换行, Ctrl + Enter发送)



相关推荐

程序猿小卡 | 3年前 | Node.js

WebSocket：5分钟从入门到精通

WebSocket的出现，使得浏览器具备了实时双向通信的能力。本文由浅入深，介绍了...

 2.8w  1323  69

4.2 从用户空间陷入

stvec 指向
uservec, 即
硬件操作
已完成

如果用户程序发出系统调用（`ecall`指令），或者做了一些非法的事情，或者设备中断，那么在用户空间中执行时就可能会产生陷阱。来自用户空间的陷阱的高级路径是`uservec` (`kernel/trampoline.S:16`)，然后是`usertrap` (`kernel/trap.c:37`)；返回时，先是`usertrapret` (`kernel/trap.c:90`)，然后是`userret` (`kernel/trampoline.S:16`)。

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

由于RISC-V硬件在陷阱期间不会切换页表，所以用户页表必须包括`uservec` (`stvec`指向的陷阱向量指令) 的映射。uservec必须切换`satp`以指向内核页表；为了在切换后继续执行指令，`uservec`必须在内核页表中与用户页表中映射相同的地址。
被映射在

xv6使用包含`uservec`的蹦床页面（`trampoline page`）来满足这些约束。xv6将蹦床页面映射到内核页表和每个用户页表中相同的虚拟地址。这个虚拟地址是`TRAMPOLINE`（如图2.3和图3.3所示）。蹦床内容在`trampoline.S`中设置，并且（当执行用户代码时）`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`并生成保存寄存器的地址。RISC-V以`sscratch`寄存器的形式提供了帮助。`uservec`开始时的`csrrw`指令交换了`a0`和`sscratch`的内容。现在用户代码的`a0`被保存了；`uservec`有一个寄存器（`a0`）可以使用；`a0`包含内核以前放在`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，内核先前将`sscratch`设置为指向每个进程的陷阱帧，该帧（除~~外~~之外）具有保存所有用户寄存器的空间(`kernel/proc.h:44`)。因为`satp`仍然指向用户页表，所以`uservec`需要将陷阱帧映射到用户地址空间中。每当创建一个进程时，`xv6`就为该进程的陷阱帧分配一个页面，并安排它映射在用户虚拟地址`TRAPFRAME`，该地址就在`TRAMPOLINE`下面。尽管使用物理地址，该进程的`p->trapframe`也指向陷阱帧，这样内核就可以通过内核页表使用它。
p->trapframe = (struct trapframe*) kalloc()

因此在交换`a0`和`sscratch`之后，`a0`持有指向当前进程陷阱帧的指针。`uservec`现在保存那里的所有用户寄存器，包括从`sscratch`读取的用户的`a0`。
(`tempoline.S:63`)

由核通过
`p->trapframe`
用户空间通过
`TRAPFRAME`

陷阱帧包含指向当前进程内核栈的指针、当前CPU的`hartid`、`usertrap`的地址和内核页表的地址。`uservec`取得这些值，将`satp`切换到内核页表，并调用`usertrap`。
jr to

`usertrap`的任务是确定陷阱的原因，处理并返回(`kernel/trap.c:37`)。如上所述，它首先改变`stvec`，这样内核中的陷阱将由`kernelvec`处理。它保存了`sepc`（保存的用户程序计数器），再保存是因为`usertrap`中可能有进程切换，导致`sepc`被覆盖。如果陷阱来自系统调用，`syscall`会处理它；如果是设备中断，`devintr`会处理；否则它是一个异常，内核会杀死错误进程。系统调用路径在保存的用户程序计数

见 `proc.h`
`struct trapframe`
处理完CSR
和`regif`后`intr-on()`
(开中断)

意味着可以多级中断？
(只有`syscall`可以)

→ ecall
→ ret

指令长度均32位

器pc上加4，因为在系统调用的情况下，RISC-V会留下指向ecall指令的程序指针（返回后需要执行ecall之后的下一条指令）。在退出的过程中，usertrap检查进程是已经被杀死还是应该让出CPU（如果这个陷阱是计时器中断）。

返回用户空间的第一步是调用usertrapret (kernel/trap.c:90)。该函数设置RISC-V控制寄存器，为将来来自用户空间的陷阱做准备。这涉及到将stvec更改为指向uservec，准备uservec所依赖的陷阱帧字段，并将sepc设置为之前保存的用户程序计数器。最后，usertrapret在用户和内核页表中都映射的蹦床页面上调用userret；原因是userret中的汇编代码会切换页表。

usertrapret对userret的调用将指针传递到a0中的进程用户页表和a1中的TRAPFRAME (kernel/trampoline.S:88)。userret将satp切换到进程的用户页表。回想一下，用户页表同时映射蹦床页面和TRAPFRAME，但没有从内核映射其他内容。同样，蹦床页面映射在用户和内核页表中的同一个虚拟地址上的事实允许用户在更改satp后继续执行。userret复制陷阱帧保存的用户a0到sscratch，为以后与TRAPFRAME的交换做准备。从此刻开始，userret可以使用的唯一数据是寄存器内容和陷阱帧的内容。下一个userret从陷阱帧中恢复保存的用户寄存器，做a0与sscratch的最后一次交换来恢复用户a0并为下一个陷阱保存TRAPFRAME，并使用sret返回用户空间。

中断
intr-off()

过程有意思

4.3 代码：调用系统调用

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

(`usys.S / li a7, SYS_exec`)

用户代码将 `exec` 需要的参数放在寄存器 `a0` 和 `a1` 中，并将系统调用号放在 `a7` 中。系统调用号与 `syscalls` 数组中的条目相匹配，`syscalls` 数组是一个函数指针表 (`kernel/syscall.c:108`)。`ecall` 指令陷入 (trap) 到内核中，执行 `uservec`、`usertrap` 和 `syscall`，和我们之前看到的一样。

`syscall` (`kernel/syscall.c:133`) 从陷阱帧 (trapframe) 中保存的 `a7` 中检索系统调用号 (`p->trapframe->a7`)，并用它索引到 `syscalls` 中，对于第一次系统调用，`a7` 中的内容是 `SYS_exec` (`kernel/syscall.h:8`)，导致了对系统调用接口函数 `sys_exec` 的调用。

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

通过 userret
将 trapframe
的内容回到
register

4.4 系统调用参数

trap frame

内核中的系统调用接口需要找到用户代码传递的参数。因为用户代码调用了系统调用封装函数，所以参数最初被放置在RISC-V C调用所约定的地方：寄存器。内核陷阱代码将用户寄存器保存到当前进程的陷阱框架中，内核代码可以在那里找到它们。函数`artint`、`artaddr`和`artfd`从陷阱框架中检索第n个系统调用参数并以整数、指针或文件描述符的形式保存。他们都调用`argraw`来检索相应的保存的用户寄存器（`kernel/syscall.c:35`）。

有些系统调用传递指针作为参数，内核必须使用这些指针来读取或写入用户内存。例如：`exec`系统调用传递给内核一个指向用户空间中字符串参数的指针数组。这些指针带来了两个挑战。首先，用户程序可能有缺陷或恶意，可能会传递给内核一个无效的指针，或者一个旨在欺骗内核访问内核内存而不是用户内存的指针。其次，`xv6`内核页表映射与用户页表映射不同，因此内核不能使用普通指令从用户提供的地址加载或存储。

内核实现了安全地将数据传输到用户提供的地址和从用户提供的地址传输数据的功能。`fetchstr`是一个例子（`kernel/syscall.c:25`）。文件系统调用，如`exec`，使用`fetchstr`从用户空间检索字符串文件名参数。`fetchstr`调用`copyinstr`来完成这项困难的工作。

`copyinstr`（`kernel/vm.c:406`）从用户页表表中的虚拟地址`srcva`复制`max`字节到`dst`。它使用`walkaddr`（它又调用`walk`）在软件中遍历页表，以确定`srcva`的物理地址`pa0`。由于内核将所有物理RAM地址映射到同一个内核虚拟地址，`copyinstr`可以直接将字符串字节从`pa0`复制到`dst`。`walkaddr`（`kernel/vm.c:95`）检查用户提供的虚拟地址是否为进程用户地址空间的一部分，因此程序不能欺骗内核读取其他内存。一个类似的函数`copyout`，将数据从内核复制到用户提供的地址。

4.5 从内核空间陷入

xv6根据执行的是用户代码还是内核代码，对CPU陷阱寄存器的配置有所不同。当在CPU上执行内核时，内核将`stvec`指向`kernelvec(kernel/kernelvec.S:10)`的汇编代码。由于xv6已经在内核中，`kernelvec`可以依赖于设置为内核页表的`satp`，以及指向有效内核栈的栈指针。`kernelvec`保存所有寄存器，以便被中断的代码最终可以不受干扰地恢复。

`kernelvec`将寄存器保存在被中断的内核线程的栈上，这是有意义的，因为寄存器值属于该线程。如果陷阱导致切换到不同的线程，那这一点就显得尤为重要——在这种情况下，陷阱将实际返回到新线程的栈上，将被中断线程保存的寄存器安全地保存在其栈上。

`Kernelvec`在保存寄存器后跳转到`kerneltrap(kernel/trap.c:134)`。`kerneltrap`为两种类型的陷阱做好了准备：设备中断和异常。它调用`devintr(kernel/trap.c:177)`来检查和处理前者。如果陷阱不是设备中断，则必定是一个异常，内核中的异常将是一个致命的错误；内核调用`panic`并停止执行。

如果由于计时器中断而调用了`kerneltrap`，并且一个进程的内核线程正在运行（而不是调度程序线程），`kerneltrap`会调用`yield`，给其他线程一个运行的机会。在某个时刻，其中一个线程会让步，让我们的线程和它的`kerneltrap`再次恢复。第7章解释了`yield`中发生的事情。

当`kerneltrap`的工作完成后，它需要返回到任何被陷阱中断的代码。因为一个`yield`可能已经破坏了保存的`sepc`和在`sstatus`中保存的前一个状态模式，因此`kerneltrap`在启动时保存它们。它现在恢复这些控制寄存器并返回到`kernelvec(kernel/kernelvec.S:48)`。`kernelvec`从栈中弹出保存的寄存器并执行`sret`，将`sepc`复制到`pc`并恢复中断的内核代码。

？值得思考的是，如果内核陷阱由于计时器中断而调用`yield`，陷阱返回是如何发生的。*timervec!*

当CPU从用户空间进入内核时，xv6将CPU的`stvec`设置为`kernelvec`；您可以在`usertrap(kernel/trap.c:29)`中看到这一点。内核执行时有一个时间窗口，但`stvec`设置为`uservec`，在该窗口中禁用设备中断至关重要。幸运的是，RISC-V总是在开始设置陷阱时禁用中断，xv6在设置`stvec`之前不会再次启用中断。

ecall	硬件	中断
usertrap()		开中断 (syscall)
usertrapret()		中断
sret	硬件	开中断

4.6 页面错误异常

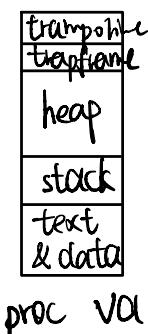
xv6对异常的响应相当无趣:如果用户空间中发生异常,内核将终止故障进程。如果内核中发生异常,则内核会崩溃。真正的操作系统通常以更有趣的方式做出反应。

例如,许多内核使用页面错误来实现写时拷贝版本的fork——copy on write (COW fork)。要解释COW fork,请回忆第3章内容:xv6的fork通过调用uvmcopy(*kernel/vm.c:309*)为子级分配物理内存,并将父级的内存复制到其中,使子级具有与父级相同的内存内容。如果父子进程可以共享父级的物理内存,则效率会更高。然而武断地实现这种方法是行不通的,因为它会导致父级和子级通过对共享栈和堆的写入来中断彼此的执行。

由页面错误驱动的COW fork可以使父级和子级安全地共享物理内存。当CPU无法将scause虚拟地址转换为物理地址时,CPU会生成页面错误异常。Risc-v有三种不同的页面错误:加载页面错误(当加载指令无法转换其虚拟地址时),存储页面错误(当存储指令无法转换其虚拟地址时)和指令页面错误(当指令的地址无法转换时)。scause寄存器中的值指示页面错误的类型,stval寄存器包含无法翻译的地址。

COW fork中的基本计划是让父子最初共享所有物理页面,但将它们映射为只读。因此,当子级或父级执行存储指令时,risc-v CPU引发页面错误异常。为了响应此异常,内核复制了包含错误地址的页面。它在子级的地址空间中映射一个权限为读/写的副本,在父级的地址空间中映射另一个权限为读/写的副本。更新页表后,内核会在导致故障的指令处恢复故障进程的执行。由于内核已经更新了相关的PTE以允许写入,所以错误指令现在将正确执行。

COW策略对fork很有效,因为通常子进程会在fork之后立即调用exec,用新的地址空间替换其地址空间。(在这种常见情况下,子级只会触发很少的页面错误,内核可以避免拷贝父进程内存完整的副本。)此外,COW fork是透明的:无需对应用程序进行任何修改即可使其受益。



除COW fork以外,页表和页面错误的结合还开发出了广泛有趣的可能性。另一个广泛使用的特性叫做惰性分配——*lazy allocation。*它包括两部分内容:首先,当应用程序调用sbrk时,内核增加地址空间,但在页表中将新地址标记为无效。其次,对于包含于其中的地址的页面错误,内核分配物理内存并将其映射到页表中。由于应用程序通常申请比他们需要的更多的内存,惰性分配可以称得上一次胜利:内核仅在应用程序实际使用它时才分配内存。像COW fork一样,内核可以对应用程序透明地实现此功能。

利用页面故障的另一个广泛使用的功能是从磁盘分页。如果应用程序需要比可用物理RAM更多的内存,内核可以换出一些页面:将它们写入存储设备(如磁盘),并将它们的PTE标记为无效。如果应用程序读取或写入被换出的页面,则CPU将触发页面错误。然后内核可以检查故障地址。如果该地址属于磁盘上的页面,则内核分配物理内存页面,将该页面从磁盘读取到该内存,将PTE更新为有效并引用该内存,然后恢复应用程序。为了给页面腾出空间,内核可能需要换出另一个页面。此功能不需要对应用

12: 指令页面错误
13: 加载 ..
15: 存储 ..

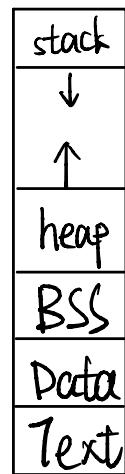
需重新执行写入命令,而非从下一条开始

用时分配

请求分页

程序进行更改，并且如果应用程序具有引用的地址(即，它们在任何给定时间仅使用其内存的子集)，则该功能可以很好地工作。

结合分页和页面错误异常的其他功能包括自动扩展栈空间和内存映射文件。



Linux
User space

4.7 真实世界

如果内核内存被映射到每个进程的用户页表中（带有适当的**PTE**权限标志），就可以消除对特殊蹦床页面的需求。这也将消除在从用户空间捕获到内核时对页表切换的需求。这反过来也将允许内核中的系统调用实现利用当前进程正在映射的用户内存，允许内核代码直接解引用用户指针。许多操作系统已经使用这些想法来提高效率。**xv6**避免了这些漏洞，以减少由于无意中使用用户指针而导致内核中出现安全漏洞的可能性，并降低了确保用户和内核虚拟地址不重叠所需的一些复杂性。

4.8 练习

1. 函数`copyin`和`copyinstr`在软件中遍历用户页表。设置内核页表，使内核拥有用户程序的映射，这样`copyin`和`copyinstr`可以使用`memcpy`将系统调用参数复制到内核空间，依靠硬件进行页表遍历
2. 实现惰性内存分配*(lazy allocation)*
3. 实现写时拷贝版本的`fork` (*copy on write fork*)

第五章 中断和设备驱动

驱动程序是操作系统中管理特定设备的代码：它配置硬件设备，告诉设备执行操作，处理由此产生的中断，并与可能正在等待设备输入/输出的进程进行交互。编写驱动可能很棘手，因为驱动程序与它管理的设备同时运行。此外，驱动程序必须理解设备的硬件接口，这可能很复杂，而且缺乏文档。

需要操作系统关注的设备通常可以被配置为生成中断，这是陷阱的一种。（内核陷阱处理代码识别设备何时引发中断，并调用驱动程序的中断处理程序；在xv6中，这种调度发生在devintr中（kernel/trap.c:177）。

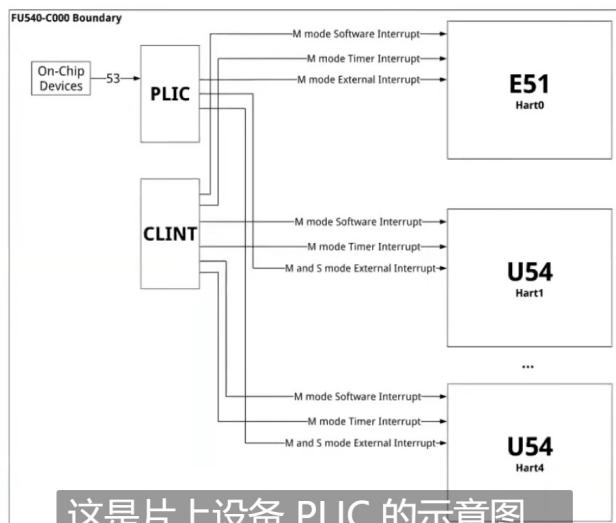
许多设备驱动程序在两种环境中执行代码：上半部分在进程的内核线程中运行，下半部分在中断时执行。上半部分通过系统调用进行调用，如希望设备执行I/O操作的read和write。这段代码可能会要求硬件执行操作（例如，要求磁盘读取块）；然后代码等待操作完成。最终设备完成操作并引发中断。驱动程序的中断处理程序充当下半部分，计算出已经完成的操作，如果合适的话唤醒等待中的进程，并告诉硬件开始执行下一个正在等待的操作。

top

(阻塞态)

PLIC 管理设备中断

platform level interrupt control



这是片上设备 PLIC 的示意图，

here's a diagram of the the PLIC in on-chip devices,

配置设备 (main.c)

- ① 初始化设备(如uart)
- ② 设置PLIC可接收的中断
- ③ 每个CPU core 表明可接收的中断

5.1 代码：控制台输入

控制台驱动程序（**console.c**）是驱动程序结构的简单说明。控制台驱动程序通过连接到RISC-V的UART串口硬件接受人们键入的字符。控制台驱动程序一次累积一行输入，处理如**backspace**和**Ctrl-u**的特殊输入字符。用户进程，如**Shell**，使用**read**系统调用从控制台获取输入行。当您在QEMU中通过键盘输入到xv6时，您的按键将通过QEMU模拟的UART硬件传递到xv6。

驱动程序管理的UART硬件是由QEMU仿真的16550芯片。（在真正的计算机上，16550将管理连接到终端或其他计算机的RS232串行链路。运行QEMU时，它连接到键盘和显示器。）

KERNBASE
以下

UART硬件在软件中看起来是一组内存映射的控制寄存器。也就是说，存在一些RISC-V硬件连接到UART的物理地址，以便载入(**load**)和存储(**store**)操作与设备硬件而不是内存交互。UART的内存映射地址起始于0x10000000或UART0

(**kernel/memlayout.h:21**)。有几个宽度为一字节的UART控制寄存器，它们关于UART0的偏移量在(**kernel/uart.c:22**)中定义。例如，LSR寄存器包含指示输入字符是否正在等待软件读取的位。这些字符（如果有的话）可用从RHR寄存器读取。每次读取一个字符，UART硬件都会从等待字符的内部FIFO寄存器中删除它，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独立于接收硬件；如果软件向THR写入一个字节，则UART传输该字节。

LSR
LCR
FCR
IER
RHR/THR

input output

UART xv6的**main**函数调用**consoleinit** (**kernel/console.c:184**) 来初始化UART硬件。该代码配置UART：UART对接收到的每个字节的输入生成一个接收中断，对发送完的每个字节的输出生成一个发送完成中断 (**kernel/uart.c:53**)。

配置设备的
读写函数
devsw[CONSOLE]
ready/write

filered对不同类型文件调用不同**read**函数
xv6的shell通过**init.c** (**user/init.c:19**) 中打开的文件描述符从控制台读取输入。（对**read**的调用实现了从内核流向**consoleread** (**kernel/console.c:82**) 的数据通路。）
consoleread等待输入到达（通过中断）并在**cons.buf**中缓冲，将输入复制到用户空间，然后（在整行到达后）返回给用户进程。如果用户还没有键入整行，任何读取进程都将在**sleep**系统调用中等待 (**kernel/console.c:98**)（第7章解释了**sleep**的细节）。
上半部分

下半部分

(IB) 当用户输入一个字符时，UART硬件要求RISC-V发出一个中断，从而激活xv6的陷阱处理程序。陷阱处理程序调用**devintr** (**kernel/trap.c:177**)，它查看RISC-V的**scause**寄存器，发现中断来自外部设备。然后它要求一个称为PLIC的硬件单元告诉它哪个设备中断了 (**kernel/trap.c:186**)。如果是UART，**devintr**调用**uartintr**。

uartintr (**kernel/uart.c:180**) 从UART硬件读取所有等待输入的字符，并将它们交给**consoleintr** (**kernel/console.c:138**)；它不会等待字符，因为未来的输入将引发一个新的中断。**consoleintr**的工作是在**cons.buf**中积累输入字符，直到一整行到达。
consoleintr对**backspace**和其他少量字符进行特殊处理。当换行符到达时，
consoleintr唤醒一个等待的**consoleread**（如果有的话）。
↓
wakeup(&cons.r)

'\n' 或 EOF 或 buf 满

驱动程序的
中断处理
程序
(完成操作
发出中断、唤醒)

返回到 read() 的位置

一旦被唤醒，`consoleread` 将监视 `cons.buf` 中的一整行，将其复制到用户空间，
并返回（通过系统调用机制）到用户空间。

`either-copyout`

从某种程度上说，`console read` 与 `console (UART 硬件) 读入字符` 间
不是直接产生关系，而是通过 `cons` 产生（异步）联系。

`uartintr`（驱动程序的中断处理程序）是由用户输入字符产生中断
而被调用，只负责将 `UART` 的字符传入 `cons.buf`（形成整行）。

`consoleread` 只负责将 `cons.buf` 中的字符读出，写入 `dst`。

“异步”是因为 `consoleread` 在 `cons.buf` 无字符时会 `sleep()`
`consoleread` / 会 `wakeup()` 唤醒 `consoleread`.

从val写到console
5.2 代码：控制台输出

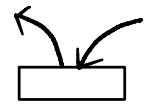
consolewrite
each c { either_copyin(l...)
uartputc() }

在连接到控制台的文件描述符上执行 `write` 系统调用，最终将到达 `uartputc(kernel/kernel.c:87)`。设备驱动程序维护一个输出缓冲区 (`uart_tx_buf`)，这样写进程就不必等待UART完成发送；相反，`uartputc` 将每个字符附加到缓冲区，调用 `uartstart` 来启动设备传输（如果还未启动），然后返回。导致 `uartputc` 等待的唯一情况是缓冲区已满。

URAT I/O 字
符后都会发
出中断请求
处理

每当UART发送完一个字节，它就会产生一个中断。`uartintr` 调用 `uartstart`，检查设备是否真的完成了发送，并将下一个缓冲的输出字符交给设备。（因此，如果一个进程向控制台写入多个字节，通常第一个字节将由 `uartputc` 调用 `uartstart` 发送，而剩余的缓冲字节将由 `uartintr` 调用 `uartstart` 发送，直到传输完成中断到来。）

需要注意，这里的一般模式是通过缓冲区和中断机制将设备活动与进程活动解耦。即使没有进程等待读取输入，控制台驱动程序仍然可以处理输入，而后续的读取将看到这些输入。类似地，进程无需等待设备就可以发送输出。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，当设备很慢（如UART）或需要立即关注（如回声型字符(echoing typed characters)）时，这种解耦尤为重要。这种想法有时被称为I/O并发



虽然 `console` I/O 均采用 I/O 并发思想，但向 `console` 输出会显得
关系更紧密（将字符输入 buf 后立即调用 `uartstart`）。因为输出
可知而输入不可知。不过这种紧密也只存在于 transmit 的
第一个字节。

5.3 驱动中的并发

你或许注意到了在`consoleread`和`consoleintr`中对`acquire`的调用。这些调用获得了一个保护控制台驱动程序的数据结构不受并发访问的锁。这里有三种并发风险：①运行在不同CPU上的两个进程可能同时调用`consoleread`，②硬件或许会在`consoleread`正在执行时要求CPU传递控制台中断③，并且硬件可能在当前CPU正在执行`consoleread`时向其他CPU传递控制台中断。第6章探讨了锁在这些场景中的作用。

在驱动程序中需要注意并发的另一种场景是，一个进程可能正在等待来自设备的输入，但是输入的中断信号可能是在另一个进程（或者根本没有进程）正在运行时到达的。因此中断处理程序不允许考虑它们已经中断的进程或代码。例如，中断处理程序不能安全地使用当前进程的页表调用`copyout`（注：因为你不知道是否发生了进程切换，当前进程可能并不是原先的进程）。中断处理程序通常做相对较少的工作（例如，只需将输入数据复制到缓冲区），并唤醒上半部分代码来完成其余工作。

→ 指中断处理程序 不运行在任何特定进程的上下文中
不能使用发出`syscall`（现在`sleep`中）的进程的内容

5.4 定时器中断

xv6使用定时器中断来维持其时钟，并使其能够在受计算量限制的进程（compute-bound processes）之间切换；`usertrap`和`kerneltrap`中的`yield`调用会导致这种切换。定时器中断来自附加到每个RISC-V CPU上的时钟硬件。`xv6`对该时钟硬件进行编程，以定期中断每个CPU。

RISC-V要求定时器中断在机器模式而不是管理模式下进行。RISC-V机器模式无需分页即可执行，并且有一组单独的控制寄存器，因此在机器模式下运行普通的xv6内核代码是不实际的。因此，xv6处理定时器中断完全不同于上面列出的陷阱机制。

机器模式下执行的代码位于`main`之前的`start.c`中，它设置了接收定时器中断（`kernel/start.c:57`）。工作的一部分是对CLINT（core-local interruptor）硬件编程，以在特定延迟后生成中断。另一部分是设置一个`scratch`区域，类似于`trapframe`，以帮助定时器中断处理程序保存寄存器和CLINT寄存器的地址。最后，`start`将`mtvec`设置为`timervec`，并使能定时器中断。

计时器中断可能发生在用户或内核代码正在执行的任何时候；内核无法在临界区操作期间禁用计时器中断。因此，计时器中断处理程序必须保证不干扰中断的内核代码。基本策略是处理程序要求RISC-V发出“软件中断”并立即返回。RISC-V用普通陷阱机制将软件中断传递给内核，并允许内核禁用它们。处理由定时器中断产生的软件中断的代码可以在`devintr` (`kernel/trap.c:204`)中看到。

机器模式定时器中断向量是`timervec` (`kernel/kernelvec.S:93`)。它在`start`准备的`scratch`区域中保存一些寄存器，以告诉CLINT何时生成下一个定时器中断，要求RISC-V引发软件中断，恢复寄存器，并且返回。定时器中断处理程序中没有C代码。

5.5 真实世界

xv6允许在内核中执行时以及在执行用户程序时触发设备和定时器中断。定时器中断迫使定时器中断处理程序进行线程切换（调用`yield`），即使在内核中执行时也是如此。如果内核线程有时花费大量时间计算而不返回用户空间，则在内核线程之间公平地对CPU进行时间分割的能力非常有用。然而，内核代码需要注意它可能被挂起（由于计时器中断），然后在不同的CPU上恢复，这是xv6中一些复杂性的来源。如果设备和计时器中断只在执行用户代码时发生，内核可以变得简单一些。

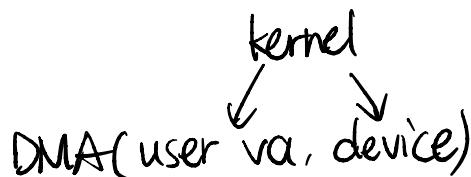
（在一台典型的计算机上支持所有设备是一项艰巨的工作，因为有许多设备，这些设备有许多特性，设备和驱动程序之间的协议可能很复杂，而且缺乏文档。在许多操作系统中，驱动程序比核心内核占用更多的代码。）

UART驱动程序读取UART控制寄存器，一次检索一字节的数据；因为软件驱动数据移动，这种模式被称为程序I/O (Programmed I/O)。程序I/O很简单，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内存，并从内存中读取传出数据。现代磁盘和网络设备使用DMA。DMA设备的驱动程序将在RAM中准备数据，然后使用对控制寄存器的单次写入来告诉设备处理准备好的数据。

保存大量 regi ↗ 当一个设备在不可预知的时间需要注意时，中断是有意义的，而且不是太频繁。但是中断有很高的CPU开销。因此，如网络和磁盘控制器的高速设备，使用一些技巧减少中断需求。（一个技巧是对整批传入或传出的请求发出单个中断。另一个技巧是驱动程序完全禁用中断，并定期检查设备是否需要注意。这种技术被称为轮询 (polling)。如果设备执行操作非常快，轮询是有意义的，但是如果设备大部分空闲，轮询会浪费CPU时间。一些驱动程序根据当前设备负载在轮询和中断之间动态切换。

UART驱动程序首先将传入的数据复制到内核中的缓冲区，然后复制到用户空间。这在低数据速率下是可行的，但是这种双重复制会显著降低快速生成或消耗数据的设备的性能。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，通常带有DMA。

使用



5.6 练习

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

第六章 锁

大多数内核，包括xv6，交错执行多个活动。交错的一个来源是多处理器硬件：计算机的多个CPU之间独立执行，如xv6的RISC-V。多个处理器共享物理内存，xv6利用共享（sharing）来维护所有CPU进行读写的数据结构。这种共享增加了一种可能性，即一个CPU读取数据结构，而另一个CPU正在更新它，甚至多个CPU同时更新相同的数据；如果不仔细设计，这种并行访问可能会产生不正确的结果或损坏数据结构。即使在单处理器上，内核也可能在许多线程之间切换CPU，导致它们的执行交错。最后，如果中断发生在错误的时间，设备中断处理程序修改与某些可中断代码相同的数据，可能导致数据损坏。~~单~~并发（concurrency）是指由于多处理器并行、线程切换或中断，多个指令流交错的情况。

内核中充满了并发访问数据（concurrently-accessed data）。例如，两个CPU可以同时调用kalloc，从而从空闲列表的头部弹出。内核设计者希望允许大量的并发，因为这样可通过并行性提高性能，并提高响应能力。然而，结果是，尽管存在这种并发性，内核设计者还是花费了大量的精力来使其正确运行。有许多方法可以得到正确的代码，有些方法比其他方法更容易。以并发下的正确性为目标的策略和支持它们的抽象称为并发控制技术（concurrency control techniques）。

xv6使用了许多并发控制技术，这取决于不同的情况。本章重点介绍了一种广泛使用的技术：**锁**。锁提供了互斥，确保一次只有一个CPU可以持有锁。如果程序员将每个共享数据项关联一个锁，并且代码在使用一个数据项时总是持有相关联的锁，那么该项一次将只被一个CPU使用。在这种情况下，我们说锁保护数据项。（尽管锁是一种易于理解的并发控制机制，但锁的缺点是它们会扼杀性能，因为它们会串行化并发操作。）

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 竞态条件

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`释放了子进程的内存。因此，在每个CPU上，内核将调用`kfree`来释放子进程的页面。内核分配器维护一个链接列表：`kalloc()`(kernel/kalloc.c:69)从空闲页面列表中取出（pop）一个内存页面；`kfree()`(kernel/kalloc.c:47)将一个内存页面添加（push）到空闲列表上。为了获得最佳性能，我们可能希望两个父进程的`kfree`可以并行执行，而不必等待另一个进程，但是考虑到xv6的`kfree`实现，这将导致错误。

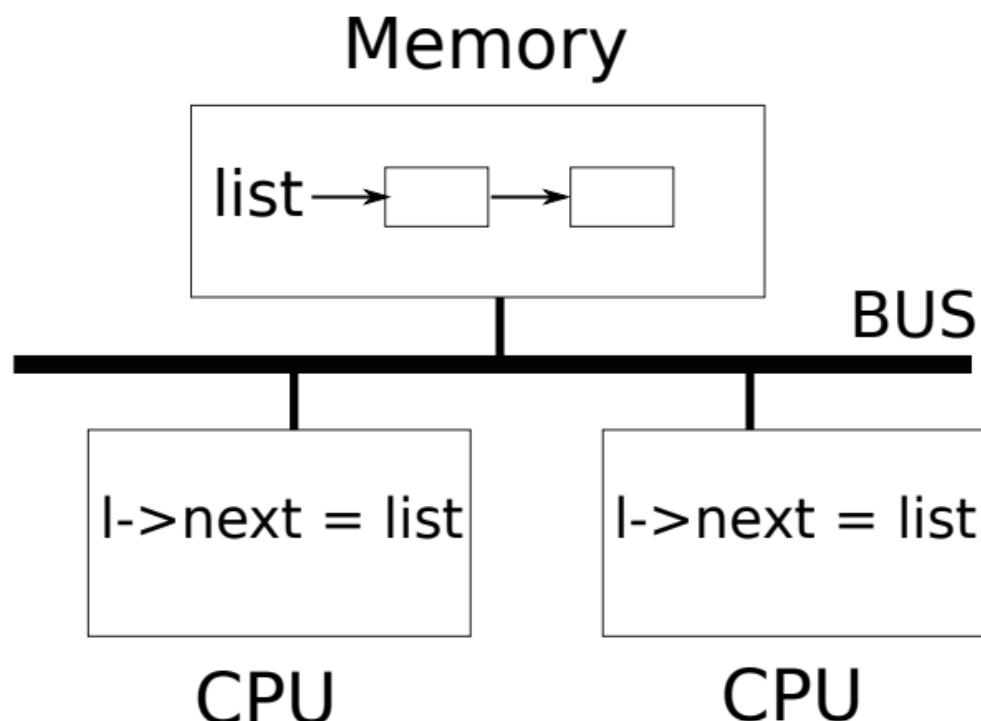


Figure 6.1: Simplified SMP architecture

图6.1更详细地说明了这项设定：链表位于两个CPU共享的内存中，这两个CPU使用`load`和`store`指令操作链表。（实际上，每个处理器都有`cache`，但从概念上讲，多处理器系统的行为就像所有CPU共享一块单独的内存一样）如果没有并发请求，您可能以如下方式实现列表push操作：

```
struct element {
    int data;
    struct element *next;
};

struct element *list = 0;

void
```

```

push(int data)
{
    struct element *l;

    l = malloc(sizeof *l);
    l->data = data;
    l->next = list;
    list = l;
}

```

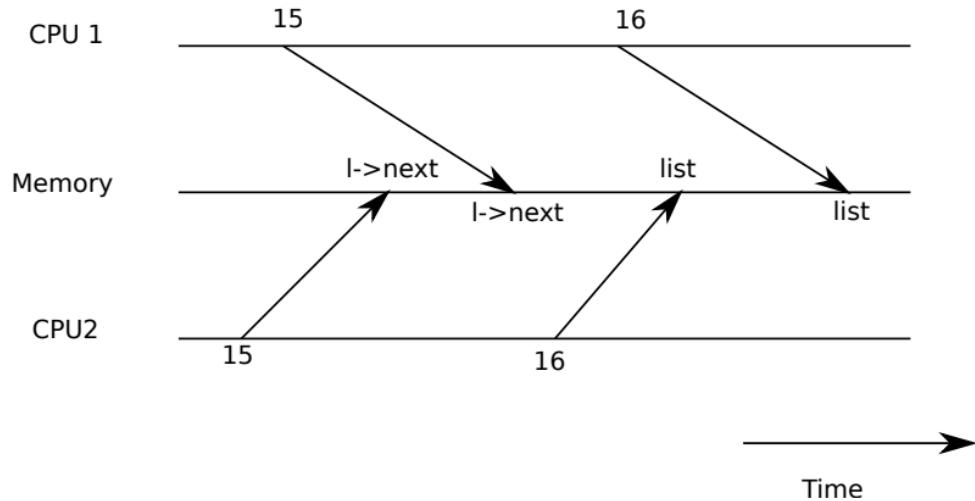


Figure 6.2: Example race

如果存在隔离性，那么这个实现是正确的。但是，如果多个副本并发执行，代码就会出错。如果两个CPU同时执行push，如图6.1所示，两个CPU都可能在执行第16行之前执行第15行，这会导致如图6.2所示的不正确的结果。然后会有两个类型为element的列表元素使用next指针设置为list的前一个值。当两次执行位于第16行的对list的赋值时，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。

第16行丢失的更新是竞态条件（race condition）的一个例子。竞态条件是指多个进程读写某些共享数据（至少有一个访问是写入）的情况。竞态通常包含bug，要么丢失更新（如果访问是写入的），要么读取未完成更新的数据结构。竞争的结果取决于进程在处理器运行的确切时机以及内存系统如何排序它们的内存操作，这可能会使竞争引起的错误难以复现和调试。例如，在调试push时添加printf语句可能会改变执行的时间，从而使竞争消失。

避免竞争的通常方法是使用锁。锁确保互斥，这样一次只有一个CPU可以执行push中敏感的代码行；这使得上述情况不可能发生。上面代码的正确上锁版本只添加了几行（用黄色突出显示）：

```

struct element {
    int data;
    struct element *next;
}

```

```

};

struct element *list = 0;
struct lock listlock;

void
push(int data)
{
    struct element *l;

    l = malloc(sizeof *l);
    l->data = data;
    acquire(&listlock);
    l->next = list;
    list = l;
    release(&listlock);
}

```

`acquire`和`release`之间的指令序列通常被称为**临界区域**(critical section)。锁的作用通常被称为**保护**`list`。

类似算法中的循环不变式

当我们说锁保护数据时，我们实际上是指锁保护适用于数据的某些**不变量集合**。不变量是**跨操作维护**的数据结构的属性。**(通常，操作的正确行为取决于操作开始时不~~变量~~是否为真。操作可能暂时违反不变量，但必须在完成之前重新建立它们。)**例如，在链表的例子中，不变量是`list`指向列表中的第一个元素，以及每个元素的`next`字段指向下一个元素。`push`的实现暂时违反了这个不变量：在第17行，`l->next`指向`list`（注：则此时`list`不再指向列表中的第一个元素，即违反了不变量），但是`list`还没有指向`l`（在第18行重新建立）。我们上面检查的竞态条件发生了，因为第二个CPU执行了依赖于列表不变量的代码，而这些代码（暂时）被违反了。**(正确使用锁可以确保每次只有一个CPU可以对临界区域中的数据结构进行操作，因此当数据结构的不变量不成立时，将没有其他CPU对数据结构执行操作。)**

您可以将锁视为串行化（serializing）并发的临界区域，以便同时只有一个进程在运行这部分代码，从而维护不变量（假设临界区域设定了正确的隔离性）。您还可以将由同一锁保护的临界区域视为彼此之间的**原子**，即彼此之间只能看到之前临界区域的完整更改集，而永远看不到部分完成的更新。

尽管正确使用锁可以改正不正确的代码，但锁限制了性能。例如，如果两个进程并发调用`kfree`，锁将串行化这两个调用，我们在不同的CPU上运行它们没有任何好处。如果多个进程同时想要相同的锁或者锁经历了争用，则称之为发生冲突（conflict）。**内核设计中的一个主要挑战是避免锁争用。**`xv6`为此几乎没做任何工作，但是复杂的内核会精心设计数据结构和算法来避免锁的争用。在链表示例中，内核可能会为每个CPU维护一个**空闲列表**，并且只有当CPU的列表为空并且必须从另一个CPU挪用内存时才会触及另一个CPU的空闲列表。其他用例可能需要更复杂的设计。

锁的位置对性能也很重要。例如，在`push`中把`acquire`的位置提前也是正确的：将`acquire`移动到第13行之前完全没问题。但这样对`malloc`的调用也会被串行化，从而降

低了性能。下面的《使用锁》一节提供了一些关于在哪里插入`acquire`和`release`调用的指导方针。

6.2 代码: Locks

自旋 $\left\{ \begin{array}{l} \text{acquire (spinlock *)} \\ \text{release (spinlock *)} \end{array} \right.$

低性能
无阻塞

xv6有两种类型的锁：自旋锁（spinlocks）和睡眠锁（sleep-locks）。我们将从自旋锁（注：自旋，即循环等待）开始。xv6将自旋锁表示为`struct spinlock` (`kernel/spinlock.h:2`)。结构体中的重要字段是`locked`，当锁可用时为零，当它被持有时为非零。从逻辑上讲，xv6应该通过执行以下代码来获取锁

```
void
acquire(struct spinlock* lk) // does not work!
{
    for(;;) {
        if(lk->locked == 0) {
            lk->locked = 1;
            break;
        }
    }
}
```

不幸的是，这种实现不能保证多处理器上的互斥。可能会发生两个CPU同时到达第5行，看到`lk->locked`为零，然后都通过执行第6行占有锁。此时就有两个不同的CPU持有锁，从而违反了互斥属性。我们需要的是一种方法，使第5行和第6行作为原子（即不可分割）步骤执行。

由CPU提供保证原生性的硬件指令

因为锁被广泛使用，多核处理器通常提供实现第5行和第6行的原子版本的指令。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，它交换寄存器和指定内存地址的内容。它原子地执行这个指令序列，使用特殊的硬件来防止任何其他CPU在读取和写入之间使用内存地址。

看一下OS书的swap硬件指令模拟

Test & set

xv6的`acquire`(`kernel/spinlock.c:22`)使用可移植的C库调用归结为`amoswap`的指令`_sync_lock_test_and_set`；返回值是`lk->locked`的旧（交换了的）内容。`acquire`函数将`swap`包装在一个循环中，直到它获得了锁前一直重试（自旋）。每次迭代将`1`与`lk->locked`进行`swap`操作，并检查`lk->locked`之前的值。如果之前为`0`，`swap`已经把`lk->locked`设置为`1`，那么我们就获得了锁；如果前一个值是`1`，那么另一个CPU持有锁，我们原子地将`1`与`lk->locked`进行`swap`的事实并没有改变它的值。

获取锁后，用于调试，`acquire`将记录下来获取锁的CPU。`lk->cpu`字段受锁保护，只能在保持锁时更改。

函数`release`(`kernel/spinlock.c:47`)与`acquire`相反：它清除`lk->cpu`字段，然后释放锁。从概念上讲，`release`只需要将`0`分配给`lk->locked`。C标准允许编译器用多个存储指令实现赋值，因此对于并发代码，C赋值可能是非原子的。因此`release`使用

`while (_sync_lock_test_and_set(&lk->lock, 1) != 0)`

$\rightarrow \text{amoswap.w.aq } a5, a5, (\$1)$

执行原子赋值的C库函数`_sync_lock_release`。该函数也可以归结为RISC-V的`amoswap`指令。

6.3 代码：使用锁

xv6在许多地方使用锁来避免竞争条件（race conditions）。如上所述，`kalloc(kernel/kalloc.c:69)`和`kfree(kernel/kalloc.c:47)`就是一个很好的例子。尝试练习1和练习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，这表明很难可靠地测试代码是否经历了锁错误和竞争后被释放。xv6有一些竞争是有可能发生的。

使用锁的一个困难部分是决定要使用多少锁，以及每个锁应该保护哪些数据和不变量。有几个基本原则。首先，任何时候可以被一个CPU写入，同时又可以被另一个CPU读写的变量，都应该使用锁来防止两个操作重叠。其次，请记住锁保护不变量（invariants）：如果一个不变量涉及多个内存位置，通常所有这些位置都需要由一个锁来保护，以确保不变量不被改变。

上面的规则说什么时候需要锁，但没有说什么时候不需要锁。为了提高效率，不要向太多地方上锁是很重要的，因为锁会降低并行性。如果并行性不重要，那么可以安排只拥有一个线程，而不用担心锁。一个简单的内核可以在多处理器上做到这一点，方法是拥有一个锁，这个锁必须在进入内核时获得，并在退出内核时释放（尽管如管道读取或`wait`的系统调用会带来问题）。许多单处理器操作系统已经被转换为使用这种方法在多处理器上运行，有时被称为“大内核锁（big kernel lock）”，但是这种方法牺牲了并行性：一次只能有一个CPU运行在内核中。如果内核做一些繁重的计算，使用一组更细粒度的锁的集合会更有效率，这样内核就可以同时在多个处理器上执行。

作为粗粒度锁的一个例子，xv6的kalloc.c分配器有一个由单个锁保护的空闲列表。如果不同CPU上的多个进程试图同时分配页面，每个进程在获得锁之前将必须在acquire中自旋等待。自旋会降低性能，因为它只是无用的等待。如果对锁的竞争浪费了很大一部分CPU时间，也许可以通过改变分配器的设计来提高性能，使其拥有多个空闲列表，每个列表都有自己的锁，以允许真正的并行分配。

作为细粒度锁定的一个例子，xv6对每个文件都有一个单独的锁，这样操作不同文件的进程通常可以不需等待彼此的锁而继续进行。文件锁的粒度可以进一步细化，以允许进程同时写入同一个文件的不同区域。最终的锁粒度决策需要由性能测试和复杂性考量来驱动。

在后面的章节解释xv6的每个部分时，他们将提到xv6使用锁来处理并发的例子。作为预览，表6.3列出了xv6中的所有锁。

锁	描述
<code>bcache.lock</code>	保护块缓冲区缓存项（block buffer cache entries）的分配
<code>cons.lock</code>	串行化对控制台硬件的访问，避免混合输出
<code>ftable.lock</code>	串行化文件表中文件结构体的分配
<code>icache.lock</code>	保护索引结点缓存项（inode cache entries）的分配

锁	描述
<code>vdisk_lock</code>	串行化对磁盘硬件和DMA描述符队列的访问
<code>kmem.lock</code>	串行化内存分配
<code>log.lock</code>	串行化事务日志操作
管道的 <code>pi->lock</code>	串行化每个管道的操作
<code>pid_lock</code>	串行化 <code>next_pid</code> 的增量
进程的 <code>p->lock</code>	串行化进程状态的改变
<code>tickslock</code>	串行化时钟计数操作
索引结点的 <code>ip->lock</code>	串行化索引结点及其内容的操作
缓冲区的 <code>b->lock</code>	串行化每个块缓冲区的操作

Figure 6.3: Locks in xv6

6.4 死锁和锁排序

如果在内核中执行的代码路径必须同时持有数个锁，那么所有代码路径以相同的顺序获取这些锁是很重要的。如果它们不这样做，就有死锁的风险。假设xv6中的两个代码路径需要锁A和B，但是代码路径1按照先A后B的顺序获取锁，另一个路径按照先B后A的顺序获取锁。假设线程T1执行代码路径1并获取锁A，线程T2执行代码路径2并获取锁B。接下来T1将尝试获取锁B，T2将尝试获取锁A。两个获取都将无限期阻塞，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。为了避免这种死锁，所有代码路径必须以相同的顺序获取锁。(全局锁获取顺序的需求意味着锁实际上是每个函数规范的一部分：调用者必须以一种使锁按照约定顺序被获取的方式调用函数。)

由于sleep的工作方式（见第7章），xv6有许多包含每个进程的锁（每个struct proc中的锁）在内的长度为2的锁顺序链。例如，consoleintr (kernel/console.c:138)是处理键入字符的中断例程。当换行符到达时，任何等待控制台输入的进程都应该被唤醒。为此，consoleintr在调用wakeup时持有cons.lock，wakeup获取等待进程的锁以唤醒它。因此，全局避免死锁的锁顺序包括必须在任何进程锁之前获取cons.lock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上的锁、新文件inode上的锁、磁盘块缓冲区上的锁、磁盘驱动程序的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前一句中提到的顺序获取锁。

遵守全局死锁避免的顺序可能会出人意料地困难。有时锁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，但是锁顺序要求在M1中的锁之前获取M2中的锁。有时锁的身份是事先不知道的，也许是因为必须持有一个锁才能发现下一个要获取的锁的身份。这种情况在文件系统中出现，因为它在路径名称中查找连续的组件，也在wait和exit代码中出现，因为它们在进程表中寻找子进程。最后，死锁的危险通常是对细粒度锁定方案的限制，因为更多的锁通常意味着更多的死锁可能性。避免死锁的需求通常是内核实现中的一个主要因素。

要求锁的全局获取方式统一

6.5 锁和中断处理函数

一些xv6自旋锁保护线程和中断处理程序共用的数据。例如，`clockintr`定时器中断处理程序在增加`ticks`(`kernel/trap.c:163`)的同时内核线程可能在`sys_sleep`(`kernel/sysproc.c:64`)中读取`ticks`。锁`tickslock`串行化这两个访问。

自旋锁和中断的交互引发了潜在的危险。假设`sys_sleep`持有`tickslock`，并且它的CPU被计时器中断中断。`clockintr`会尝试获取`tickslock`，意识到它被持有后等待释放。在这种情况下，`tickslock`永远不会被释放：只有`sys_sleep`可以释放它，但是`sys_sleep`直到`clockintr`返回前不能继续运行。所以CPU会死锁，任何需要锁的代码也会冻结。

为了避免这种情况，如果一个自旋锁被中断处理程序所使用，那么CPU必须保证在启用中断的情况下永远不能持有该锁。xv6更保守：当CPU获取任何锁时，xv6总是禁用该CPU上的中断。中断仍然可能发生在其他CPU上，此时中断的`acquire`可以等待线程释放自旋锁；由于不在同一CPU上，不会造成死锁。

保守也是
简化

当CPU未持有自旋锁时，xv6重新启用中断；它必须做一些记录来处理嵌套的临界区域。`acquire`调用`push_off`(`kernel/spinlock.c:89`)并且`release`调用`pop_off`(`kernel/spinlock.c:100`)来跟踪当前CPU上锁的嵌套级别。当计数达到零时，`pop_off`恢复最外层临界区域开始时存在的中断使能状态。`intr_off`和`intr_on`函数执行RISC-V指令分别用来禁用和启用中断。

严格的在设置`lk->locked`(`kernel/spinlock.c:28`)之前让`acquire`调用`push_off`是很重要的。如果两者颠倒，会存在一个既持有锁又启用了中断的短暂窗口期，不幸的话定时器中断会使系统死锁。同样，只有在释放锁之后，`release`才调用`pop_off`也是很重要的(`kernel/spinlock.c:66`)。

记录字段
`struct cpu_nooff`

6.6 指令和内存访问排序

人们很自然地会想到程序是按照源代码语句出现的顺序执行的。然而，许多编译器和中央处理器为了获得更高的性能而不按顺序执行代码。如果一条指令需要许多周期才能完成，中央处理器可能会提前发出指令，这样它就可以与其他指令重叠，避免中央处理器停顿。例如，中央处理器可能会注意到在顺序指令序列A和B中彼此不存在依赖。CPU也许首先启动指令B，或者是因为它的输入先于A的输入准备就绪，或者是为了重叠执行A和B。编译器可以执行类似的重新排序，方法是在源代码中一条语句的指令发出之前，先发出另一条语句的指令。

编译器和CPU在重新排序时需要遵循一定规则，以确保它们不会改变正确编写的串行代码的结果。然而，规则确实允许重新排序后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。CPU的排序规则称为内存模型（memory model）。

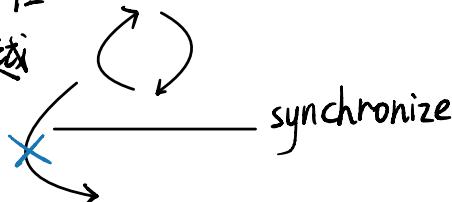
例如，在push的代码中，如果编译器或CPU将对应于第4行的存储指令移动到第6行release后的某个地方，那将是一场灾难：

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

如果发生这样的重新排序，将会有一个窗口期，另一个CPU可以获取锁并查看更新后的list，但却看到一个未初始化的list->next。

(为了告诉硬件和编译器不要执行这样的重新排序，xv6在
acquire(kernel/spinlock.c:22) 和 release(kernel/spinlock.c:47) 中都使用了
__sync_synchronize()。) __sync_synchronize() 是一个内存障碍：它告诉编译器
和 CPU 不要跨障碍重新排序 load 或 store 指令。因为 xv6 在访问共享数据时使用了锁，
xv6 的 acquire 和 release 中的障碍在几乎所有重要的情况下都会强制顺序执行。第 9 章
讨论了一些例外。

在屏障前访存指令调整都沒問題，但調整不可越過屏障



6.7 睡眠锁

有时xv6需要长时间保持锁。例如，文件系统（第8章）在磁盘上读写文件内容时保持文件锁定，这些磁盘操作可能需要几十毫秒。如果另一个进程想要获取自旋锁，那么长时间保持自旋锁会导致获取进程在自旋时浪费很长时间的CPU。自旋锁的另一个缺点是，一个进程在持有自旋锁的同时不能让出（yield）CPU，然而我们希望持有锁的进程等待磁盘I/O的时候其他进程可以使用CPU。持有自旋锁时让步是非法的，因为如果第二个线程试图获取自旋锁，就可能导致死锁：因为acquire不会让出CPU，第二个线程的自旋可能会阻止第一个线程运行并释放锁。在持有锁时让步也违反了在持有自旋锁时中断必须关闭的要求。因此，我们想要一种锁，它在等待获取锁时让出CPU，并允许在持有锁时让步（以及中断）。

第二个CPU的
acquire

sleeplock
{ uint locked
spinlock lk
int pid
}

xv6以睡眠锁（sleep-locks）的形式提供了这种锁。`acquiresleep` (`kernel/sleeplock.c:22`) 在等待时让步CPU，使用的技术将在第7章中解释。在更高层次上，睡眠锁有一个被自旋锁保护的锁定字段，`acquiresleep`对`sleep`的调用原子地让出CPU并释放自旋锁。结果是其他线程可以在`acquiresleep`等待时执行。

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

因为等待会浪费CPU时间，所以自旋锁最适合短的临界区域；睡眠锁对于冗长的操作效果很好。

6.8 真实世界

会有时间了
并行计算吗

尽管对并发原语和并行性进行了多年的研究，但使用锁进行编程仍然具有挑战性。通常最好将锁隐藏在更高级别的结构中，如同步队列，尽管xv6没有这样做。如果您使用锁进行编程，明智的做法是使用试图识别竞争条件（race conditions）的工具，因为很容易错过需要锁的不变量。

大多数操作系统都支持POSIX线程（Pthread），它允许一个用户进程在不同的CPU上同时运行几个线程。Pthread支持用户级锁（user-level locks）、障碍（barriers）等。支持Pthread需要操作系统的支持。例如，应该是这样的情况，如果一个Pthread在系统调用中阻塞，同一进程的另一个Pthread应当能够在该CPU上运行。另一个例子是，如果一个线程改变了其进程的地址空间（例如，映射或取消映射内存），内核必须安排运行同一进程下的线程的其他CPU更新其硬件页表，以反映地址空间的变化。

没有原子指令实现锁是可能的，但是代价昂贵，并且大多数操作系统使用原子指令。

如果许多CPU试图同时获取相同的锁，可能会付出昂贵的开销。如果一个CPU在其本地cache中缓存了一个锁，而另一个CPU必须获取该锁，那么更新保存该锁的cache行的原子指令必须将该行从一个CPU的cache移动到另一个CPU的cache中，并且可能会使cache行的任何其他副本无效。从另一个CPU的cache中获取cache行可能比从本地cache中获取一行的代价要高几个数量级。

为了避免与锁相关的开销，许多操作系统使用无锁的数据结构和算法。例如，可以实现一个像本章开头那样的链表，在列表搜索期间不需要锁，并且使用一个原子指令在一个列表中插入一个条目。然而，无锁编程比有锁编程更复杂；例如，人们必须担心指令和内存重新排序。有锁编程已经很难了，所以xv6避免了无锁编程的额外复杂性。

6.9 练习

1. 注释掉在 `kalloc` 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；你希望看到什么症状？当你运行 `xv6` 时，你看到这些症状了吗？运行 `usertests` 时呢？如果你没有看到问题是为什么呢？看看你是否可以通过在 `kalloc` 的临界区域插入虚拟循环来引发问题。
2. 假设您将 `kfree` 中的锁注释掉（在 `kalloc` 中恢复锁之后）。现在可能会出什么问题？`kfree` 中缺少锁比 `kalloc` 中缺少锁的危害小吗？
3. 如果两个CPU同时调用 `kalloc`，则其中一个不得不等待另一个，这对性能不利。修改 `kalloc.c` 以具有更多的并行性，这样不同CPU对 `kalloc` 的同时调用就可以进行，而不需要相互等待。
4. 使用POSIX线程编写一个并行程序，大多数操作系统都支持这种程序。例如，实现一个并行哈希表，并测量 `puts/gets` 的数量是否随着内核数量的增加而缩放。
5. 在 `xv6` 中实现 `Pthread` 的一个子集。也就是说，实现一个用户级线程库，这样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。想出一个正确处理线程发出阻塞系统调用并改变其共享地址空间的方案。

第七章 调度

任何操作系统都可能运行比CPU数量更多的进程，所以需要一个进程间分时共享CPU的方案。这种共享最好对用户进程透明。一种常见的方法是，通过将进程多路复用到硬件CPU上，使每个进程产生一种错觉，即它有自己的虚拟CPU。本章解释了xv6如何实现这种多路复用。

7.1 多路复用

xv6通过在两种情况下将每个CPU从一个进程切换到另一个进程来实现多路复用（Multiplexing）。第一：当进程等待设备或管道I/O完成，或等待子进程退出，或在sleep系统调用中等待时，xv6使用睡眠（sleep）和唤醒（wakeup）机制切换。第二：xv6周期性地强制切换以处理长时间计算而不睡眠的进程。这种多路复用产生了每个进程都有自己的CPU的错觉，就像xv6使用内存分配器和硬件页表来产生每个进程都有自己内存的错觉一样。

资源求而不得
及进程同步
(进程无法佳
体执行)

实现多路复用带来了一些挑战。首先，如何从一个进程切换到另一个进程？尽管上下文切换的思想很简单，但它的实现是xv6中最不透明的代码之一。第二，如何以对用户进程透明的方式强制切换？xv6使用标准技术，通过定时器中断驱动上下文切换。第三，许多CPU可能同时在进程之间切换，使用一个用锁方案来避免争用是很有必要的。第四，进程退出时必须释放进程的内存以及其他资源，但它不能自己完成所有这一切，因为（例如）它不能在仍然使用自己内核栈的情况下释放它。第五，多核机器的每个核心必须记住它正在执行哪个进程，以便系统调用正确影响对应进程的内核状态。最后，sleep允许一个进程放弃CPU，wakeup允许另一个进程唤醒第一个进程。需要小心避免导致唤醒通知丢失的竞争。xv6试图尽可能简单地解决这些问题，但结果代码很复杂。

争用的资源
有哪些？

不透明指与
硬件密切！

由kernel完成
proc资源回收！

7.2 代码：上下文切换

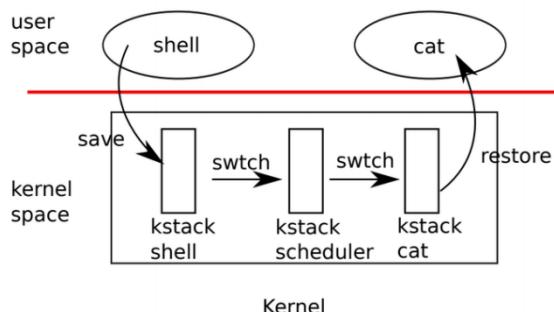


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

图7.1概述了从一个用户进程（旧进程）切换到另一个用户进程（新进程）所涉及的步骤：[一个到旧进程内核线程的用户-内核转换（系统调用或中断），一个到当前CPU调度程序线程的上下文切换，一个到新进程内核线程的上下文切换，以及一个返回到用户级进程的陷阱。] 调度程序在旧进程的内核栈上执行是不安全的：其他一些核心可能会唤醒进程并运行它，而在两个不同的核心上使用同一个栈将是一场灾难，因此xv6调度程序在每个CPU上都有一个专用线程（保存寄存器和栈）。在本节中，我们将研究在内核线程和调度程序线程之间切换的机制。

CPU调度程序
线程是CPU专用
(专用线程和其
它线程有何不同)

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，并恢复新线程先前保存的寄存器；栈指针和程序计数器被保存和恢复的事实意味着CPU将切换栈和执行中的代码。

线程切换不
涉及内存页
来管理内容)

(swtch.S/swtch)

函数 `swtch` 为内核线程切换执行保存和恢复操作。`swtch` 对线程没有直接的了解；它只是保存和恢复寄存器集，称为上下文 (contexts)。当某个进程要放弃CPU时，该进程的内核线程调用 `swtch` 来保存自己的上下文并返回到调度程序的上下文。每个上下文都包含在一个 `struct context` (`kernel/proc.h:2`) 中，这个结构体本身包含在一个进程的 `struct proc` 或一个CPU的 `struct cpu` 中。`Swtch` 接受两个参数：`struct context *old` 和 `struct context *new`。它将当前寄存器保存在 `old` 中，从 `new` 中加载寄存器，然后返回。

ra
sp
so->sl
↓
callee saved

让我们跟随一个进程通过 `swtch` 进入调度程序。我们在第4章中看到，中断结束时的一种可能性是 `usertrap` 调用了 `yield`。依次地：`Yield` 调用 `sched`，`sched` 调用 `swtch` 将当前上下文保存在 `p->context` 中，并切换到先前保存在 `cpu->scheduler` (`kernel/proc.c:517`) 中的调度程序上下文。

注：当前版本的xv6中调度程序上下文是 `cpu->context`

swtch函数ret
后，和其它被
调用函数一样
pc从ra值处
继续执行

`Swtch` (`kernel/swtch.S:3`) 只保存被调用方保存的寄存器 (`callee-saved registers`)；调用方保存的寄存器 (`caller-saved registers`) 通过调用C代码保存在栈上 (如果需要)。`Swtch` 知道 `struct context` 中每个寄存器字段的偏移量。它不保存程序计数器。但 `swtch` 保存 `ra` 寄存器，该寄存器保存调用 `swtch` 的返回地址。 现在，`swtch` 从新进程的上下文中恢复寄存器，该上下文保存前一个 `swtch` 保存的寄存器值。当

`swtch`返回时，它返回到由`ra`寄存器指定的指令，即新线程以前调用`swtch`的指令。另外，它在新线程的栈上返回。

注：关于callee-saved registers和caller-saved registers请回看视频课程LEC5以及文档《Calling Convention》

这里不太容易理解，这里举个课程视频中的例子：

以`cc`切换到`ls`为例，且`ls`此前运行过

1. `xv6`将`cc`程序的内核线程的内核寄存器保存在一个`context`对象中
2. 因为要切换到`ls`程序的内核线程，那么`ls`程序现在的状态必然是`RUNABLE`，表明`ls`程序之前运行了一半。这同时也意味着：
 - a. `ls`程序的用户空间状态已经保存在了对应的`trapframe`中
 - b. `ls`程序的内核线程对应的内核寄存器已经保存在对应的`context`对象中
- 所以接下来，`xv6`会恢复`ls`程序的内核线程的`context`对象，也就是恢复内核线程的寄存器。
3. 之后`ls`会继续在它的内核线程栈上，完成它的中断处理程序
4. 恢复`ls`程序的`trapframe`中的用户进程状态，返回到用户空间的`ls`程序中
5. 最后恢复执行`ls`

在我们的示例中，`sched`调用`swtch`切换到`cpu->scheduler`，即每个CPU的调度程序上下文。调度程序上下文之前通过`scheduler`对`swtch`（`kernel/proc.c:475`）的调用进行了保存。当我们追踪`swtch`到返回时，他返回到`scheduler`而不是`sched`，并且它的栈指针指向当前CPU的调用程序栈（`scheduler stack`）。

对于进程而言，进程切换是一个透明的过程。

进程切换只是将旧进程的状态停止

{ 用户状态由 trapframe 保存，因为进程切换必然要 trap
核心状态由 swtch 保存

7.3 代码：调度

上一节介绍了`swtch`的底层细节；现在，让我们以`swtch`为给定对象，检查从一个进程的内核线程通过调度程序切换到另一个进程的情况。调度器（`scheduler`）以每个

特殊线程？

CPU上一个特殊线程的形式存在，每个线程都运行`scheduler`函数。此函数负责选择下一个要运行的进程。（想要放弃CPU的进程必须先获得自己的进程锁`p->lock`，并释放它

CPU → context

持有的任何其他锁，更新自己的状态（`p->state`），然后调用`sched`。）

中是 scheduler()

`yield`（`kernel/proc.c:515`）遵循这个约定，`sleep`和`exit`也遵循这个约定，我们将在后面进行研究。`Sched`对这些条件再次进行检查（`kernel/proc.c:499-504`），并检查这些条件的隐含条件：由于锁被持有，中断应该被禁用。最后，`sched`调用`swtch`将当前上下文保存在`p->context`中，并切换到`cpu->scheduler`中的调度程序上下文。

入口？在哪

`Swtch`在调度程序的栈上返回，就像是`scheduler`的`swtch`返回一样。`scheduler`继续

设置的？

for循环，找到要运行的进程，切换到该进程，重复循环。有趣

我们刚刚看到，`xv6`在对`swtch`的调用中持有`p->lock`：`swtch`的调用者必须已经持有了锁，并且锁的控制权传递给切换到的代码。这种约定在锁上是不寻常的；通常，获取锁的线程还负责释放锁，这使得对正确性进行推理更加容易。对于上下文切换，有必要打破这个惯例，因为`p->lock`保护进程`state`和`context`字段上的不变量，而这些不变量在`swtch`中执行时不成立。如果在`swtch`期间没有保持`p->lock`，可能会出现一个问题：在`yield`将其状态设置为`RUNNABLE`之后，但在`swtch`使其停止使用自己的内核栈之前，另一个CPU可能会决定运行该进程。结果将是两个CPU在同一栈上运行，这不可能是正确的。

（内核线程总是在`sched`中放弃其CPU，并总是切换到调度程序中的同一位置，而调度程序（几乎）总是切换到以前调用`sched`的某个内核线程。）因此，如果要打印`xv6`切换线程处的行号，将观察到以下简单模式：（`kernel/proc.c:475`），

（`kernel/proc.c:509`），（`kernel/proc.c:475`），（`kernel/proc.c:509`）等等。在两个线程之间进行这种样式化切换的过程有时被称为协程（coroutines）；在本例中，`sched`和`scheduler`是彼此的协同程序。

存在一种情况使得调度程序对`swtch`的调用没有以`sched`结束。（一个新进程第一次被调度时，它从`forkret`（`kernel/proc.c:527`）开始。`Forkret`存在以释放`p->lock`；否则，新进程可以从`usertrapret`开始。）

没懂

`scheduler`（`kernel/proc.c:457`）运行一个简单的循环：找到要运行的进程，运行它直到它让步，然后重复循环。`scheduler`在进程表上循环查找可运行的进程，该进程具有`p->state == RUNNABLE`。一旦找到一个进程，它将设置CPU当前进程变量`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`）。

角度

考虑调度代码结构的一种方法是，它为每个进程强制维持一个不变量的集合，并在这些不变量不成立时持有`p->lock`。其中一个不变量是：如果进程是`RUNNING`状态，计时器中断的`yield`必须能够安全地从进程中切换出去；这意味着CPU寄存器必须保存进程的寄存器值（即`swtch`没有将它们移动到`context`中），并且`c->proc`必须指向进

还

意味着 xv6 不
存在进程代
先级“一视”

关于`p->lock`的
`acquire`和
`release`很有意思

也正是因为
不成立，所以
才要保持

$p \rightarrow lock$

程。另一个不变量是：如果进程是RUNNABLE状态，空闲CPU的调度程序必须安全地运行它；这意味着 $p \rightarrow context$ 必须保存进程的寄存器（即，它们实际上不在实际寄存器中），没有CPU在进程的内核栈上执行，并且没有CPU的 $c \rightarrow proc$ 引用进程。请注意，在保持 $p \rightarrow lock$ 时，这些属性通常不成立。

维护上述不变量是xv6经常在一个线程中获取 $p \rightarrow lock$ 并在另一个线程中释放它的原因，例如在yield中获取并在scheduler中释放。一旦yield开始修改一个RUNNING进程的状态为RUNNABLE，锁必须保持被持有状态，直到不变量恢复：最早的正确释放点是scheduler（在其自身栈上运行）清除 $c \rightarrow proc$ 之后。类似地，一旦scheduler开始将RUNNABLE进程转换为RUNNING，在内核线程完全运行之前（在swtch之后，例如在yield中）绝不能释放锁。

$p \rightarrow lock$ 还保护其他东西：`exit`和`wait`之间的相互作用，避免丢失`wakeup`的机制（参见第7.5节），以及避免一个进程退出和其他进程读写其状态之间的争用（例如，`exit`系统调用查看 $p \rightarrow pid$ 并设置 $p \rightarrow killed$ (kernel/proc.c:611)）。为了清晰起见，也许为了性能起见，有必要考虑一下 $p \rightarrow lock$ 的不同功能是否可以拆分。

显然， $p \rightarrow lock$
的获取和释
放间是“原
子操作”。

7.4 代码： mycpu和myproc

xv6通常需要指向当前进程的`proc`结构体的指针。在单处理器系统上，可以有一个指向当前`proc`的全局变量。但这不能用于多核系统，因为每个核执行的进程不同。解决这个问题的方法是基于每个核心都有自己的寄存器集，从而使用其中一个寄存器来帮助查找每个核心的信息。

xv6为每个CPU维护一个`struct cpu`，它记录当前在该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的`struct cpu`的指针。RISC-V给它的CPU编号，给每个CPU一个`hartid`。xv6确保每个CPU的`hartid`在内核中存储在该CPU的`tp`寄存器中。这允许`mycpu`使用`tp`对一个`cpu`结构体数组（即`cpus`数组，`kernel/proc.c:9`）进行索引，以找到正确的那个。

确保CPU的`tp`始终保存CPU的`hartid`有点麻烦。`mstart`在CPU启动次序的早期设置`tp`寄存器，此时仍处于机器模式（`kernel/start.c:46`）。因为用户进程可能会修改`tp`，`usertrapret`在蹦床页面（trampoline page）中保存`tp`。最后，`uservec`在从用户空间（`kernel/trampoline.S:70`）进入内核时恢复保存的`tp`。编译器保证永远不会使用`tp`寄存器。如果RISC-V允许xv6直接读取当前hartid会更方便，但这只允许在机器模式下，而不允许在管理模式下。

`cpuid`和`mycpu`的返回值很脆弱：如果定时器中断并导致线程让步（`yield`），然后移动到另一个CPU，以前返回的值将不再正确。为了避免这个问题，xv6要求调用者禁用中断，并且只有在使用完返回的`struct cpu`后才重新启用。

函数`myproc` (`kernel/proc.c:68`)返回当前CPU上运行进程`struct proc`的指针。`myproc`禁用中断，调用`mycpu`，从`struct cpu`中取出当前进程指针（`c->proc`），然后启用中断。即使启用中断，`myproc`的返回值也可以安全使用：如果计时器中断将调用进程移动到另一个CPU，其`struct proc`指针不会改变。

7.5 sleep与wakeup

此处“交互”
该指的是“同
步”

调度和锁有助于隐藏一个进程对另一个进程的存在，但到目前为止，我们还没有帮助进程进行有意交互的抽象。为解决这个问题已经发明了许多机制。**xv6**使用了一种称为**sleep**和**wakeup**的方法，它允许一个进程在等待事件时休眠，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）或条件同步机制（conditional synchronization mechanisms）。

为了说明，让我们考虑一个称为信号量（semaphore）的同步机制，它可以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（对于生产者）增加计数。“P”操作（对于使用者）等待计数为非零，然后递减并返回。如果只有一个生产者线程和一个消费者线程，并且它们在不同的CPU上执行，并且编译器没有进行过积极的优化，那么此实现将是正确的：

```
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);
}
```

轮回(忙式等待)

上面的实现代价昂贵。如果生产者很少采取行动，消费者将把大部分时间花在**while**循环中，希望得到非零计数。消费者的CPU可以找到比通过反复轮询**s->count**繁忙等待更有成效的工作。要避免繁忙等待，消费者需要一种方法来释放CPU，并且只有在**V**增加计数后才能恢复。

这是朝着这个方向迈出的一步，尽管我们将看到这是不够的。让我们想象一对调用，**sleep**和**wakeup**，工作流程如下。**Sleep(chan)**在任意值**chan**上睡眠，称为等待通道（wait channel）。**Sleep**将调用进程置于睡眠状态，释放CPU用于其他工作。

Wakeup(chan)唤醒所有在**chan**上睡眠的进程（如果有），使其**sleep**调用返回。如果没有进程在**chan**上等待，则**wakeup**不执行任何操作。我们可以将信号量实现更改为使用**sleep**和**wakeup**（更改的行添加了注释）：

是的。因为chan只是数字，没有数据结构
作为sleep channel。只要p->chan==chan
的进程都会被wakeup

唤醒所有？
chan上所有
进程进行争
夺，而非由chan决定优先级吗

```

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

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

```

P现在放弃CPU而不是自旋，这很好。然而，事实证明，使用此接口设计sleep和wakeup而不遭受所谓的丢失唤醒（lost wake-up）问题并非易事。假设P在第9行发现s->count==0。当P在第9行和第10行之间时，V在另一个CPU上运行：它将s->count更改为非零，并调用wakeup，这样就不会发现进程处于休眠状态，因此不会执行任何操作。现在P继续在第10行执行：它调用sleep并进入睡眠。这会导致一个问题：P正在休眠，等待调用V，而V已经被调用。除非我们运气好，生产者再次呼叫V，否则消费者将永远等待，即使count为非零。

这个问题的根源是V在错误的时刻运行，违反了P仅在s->count==0时才休眠的不变量。保护不变量的一种不正确的方法是将锁的获取（下面以黄色突出显示）移动到P中，以便其检查count和调用sleep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); // !pay attention
    while (s->count == 0)
        sleep(s);
    s->count -= 1;
    release(&s->lock);
}

```

人们可能希望这个版本的P能够避免丢失唤醒，因为锁阻止V在第10行和第11行之间执行。它确实这样做了，但它会导致死锁：P在睡眠时持有锁，因此V将永远阻塞等待锁。

我们将通过更改sleep的接口来修复前面的方案：调用方必须将条件锁（condition lock）传递给sleep，以便在调用进程被标记为asleep并在睡眠通道上等待后sleep可以释放锁。如果有一个并发的V操作，锁将强制它在P将自己置于睡眠状态前一直等待，因此wakeup将找到睡眠的消费者并将其唤醒。一旦消费者再次醒来，sleep会在返回前重新获得锁。我们新的正确的sleep/wakeup方案可用如下（更改以黄色突出显示）：

此处 P.V 在
该处是理论
课学的。那么
所谓“原生性”
就是由 s->lock
的获取与释放
保证的吗？

```
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); // !pay attention
    s->count -= 1;
    release(&s->lock);
}
```

P持有s->lock的事实阻止V在P检查s->count和调用sleep之间试图唤醒它。然而请注意，我们需要sleep释放s->lock并使消费者进程进入睡眠状态的操作是原子的。

sleep(void *chan, struct spinlock *lk)
wakeup(void *chan)

7.6 代码： sleep和wakeup

通常会用某个地址确保chan不重叠

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` 释放 CPU; `wakeup` 查找在给定等待通道上休眠的进程，并将其标记为 `RUNNABLE`。`sleep` 和 `wakeup` 的调用者可以使用任何相互间方便的数字作为通道。`xv6` 通常使用等待过程中涉及的内核数据结构的地址。

`sleep` 获得 `p->lock` (`kernel/proc.c:559`)。要进入睡眠的进程现在同时持有 `p->lock` 和 `lk`。在调用者 (示例中为 `P`) 中持有 `lk` 是必要的：它确保没有其他进程 (在示例中指一个运行的 `V`) 可以启动 `wakeup(chan)` 调用。既然 `sleep` 持有 `p->lock`, 那么释放 `lk` 是安全的：其他进程可能会启动对 `wakeup(chan)` 的调用，但是 `wakeup` 将等待获取 `p->lock`, 因此将等待 `sleep` 把进程置于睡眠状态的完成，以防止 `wakeup` 错过 `sleep`。

SLEEPING状态时不变量和RUNNABLE相同？

还有一个小问题：如果 `lk` 和 `p->lock` 是同一个锁，那么如果 `sleep` 试图获取 `p->lock` 就会自身死锁。但是，如果调用 `sleep` 的进程已经持有 `p->lock`, 那么它不需要做更多的事情来避免错过并发的 `wakeup`。当 `wait` (`kernel/proc.c:582`) 持有 `p->lock` 调用 `sleep` 时，就会出现这种情况。

此时 sleep 和 wakeup
的 caller (r1P.vh 例) 就无需 s->lock

由于 `sleep` 只持有 `p->lock` 而无其他，它可以通过记录睡眠通道、将进程状态更改为 `SLEEPING` 并调用 `sched` (`kernel/proc.c:564-567`) 将进程置于睡眠状态。过一会儿，我们就会明白为什么在进程被标记为 `SLEEPING` 之前不将 `p->lock` 释放 (由 `scheduler`) 是至关重要的。

在某个时刻，一个进程将获取条件锁，设置睡眠者正在等待的条件，并调用 `wakeup(chan)`。在持有状态锁时调用 `wakeup` 非常重要*^{[注]*}。`wakeup` 遍历进程表 (`kernel/proc.c:582`)。它获取它所检查的每个进程的 `p->lock`，这既是因为它可能会操纵该进程的状态，也是因为 `p->lock` 确保 `sleep` 和 `wakeup` 不会彼此错过。当 `wakeup` 发现一个 `SLEEPING` 的进程且 `chan` 相匹配时，它会将该进程的状态更改为 `RUNNABLE`。调度器下次运行时，将看到进程已准备好运行。

所谓“唤醒”
也就是改变
进程状态

注：严格地说，`wakeup` 只需跟在 `acquire` 之后就足够了（也就是说，可以在 `release` 之后调用 `wakeup`）

二者不变量相
同的猜测

为什么 `sleep` 和 `wakeup` 的用锁规则能确保睡眠进程不会错过唤醒？休眠进程从检查条件之前的某处到标记为休眠之后的某处，要么持有条件锁，要么持有其自身的 `p->lock` 或同时持有两者。调用 `wakeup` 的进程在 `wakeup` 的循环中同时持有这两个锁。因此，要么唤醒器 (`waker`) 在消费者线程检查条件之前使条件为真；要么唤醒器的 `wakeup` 在睡眠线程标记为 `SLEEPING` 后对其进行严格检查。然后 `wakeup` 将看到睡眠进程并将其唤醒（除非有其他东西首先将其唤醒）。

有进程在 sleep 时
无法 wakeup,
wakeup 时无法
sleep

有时，多个进程在同一个通道上睡眠；例如，多个进程读取同一个管道。一个单独的 `wakeup` 调用就能把它们全部唤醒。其中一个将首先运行并获取与 `sleep` 一同调用的锁，并且（在管道例子中）读取在管道中等待的任何数据。尽管被唤醒，其他进程

在若干竞争者中（可能含有非 chan 的新竞争者），只有与新增资源
相等的进程可争得资
源，其它进程又将 sleep

wakeup 只负责
修改进程状态
为 RUNNABLE，
也即争夺资源
的“资格”。

将发现没有要读取的数据。从他们的角度来看，醒来是“虚假的”，他们必须再次睡眠。因此，在检查条件的循环中总是调用 `sleep`。

这也解释了为什么
P操作中是 while
而非 if

如果两次使用 `sleep/wakeup` 时意外选择了相同的通道，则不会造成任何伤害：它们将看到虚假的唤醒，但如上所述的循环将容忍此问题。`sleep/wakeup` 的魅力在于它既轻量级（不需要创建特殊的数据结构来充当睡眠通道），又提供了一层抽象（调用者不需要知道他们正在与哪个特定进程进行交互）。

sleep/wakeup 参考流程

lock source

channel

proc1 (类 V 操作)

```
{ acquire(lock)
  op(source)
  if(condition == True)
    sleep(channel, lock)
  op(source)
  release(lock)
}
```

{
 acquire(p→lock)
 release(p→lock)
 // go to sleep
 sched()
 switch(~)
 :
 scheduler()
 release(p→lock)
 acquire(p→lock)}

wakeup 发生在此处

{
 release(p→lock)
 acquire(p→lock)
 p→state = RUNNING
 switch(~)}

proc2 (类 P 操作)

```
{ acquire(lock)
  op(source)
  wakeup(channel)
  release(lock)
}
```

{
 acquire(p→lock)
 p→state = RUNNABLE
 release(p→lock)}

7.7 代码：Pipes

使用睡眠和唤醒来同步生产者和消费者的一个更复杂的例子是xv6的管道实现。我们在第1章中看到了管道接口：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。以后的章节将研究围绕管道的文件描述符支持，但现在让我们看看和的实现。

每个管道都由一个`struct pipe`表示，其中包含一个锁`lock`和一个数据缓冲区`data`。字段`nread`和`nwrite`统计从缓冲区读取和写入缓冲区的总字节数。缓冲区是环形的：在`buf[PIPESIZE-1]`之后写入的下一个字节是`buf[0]`。而计数不是环形。此约定允许实现区分完整缓冲区（`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]`，而不仅仅是`buf[nread]`（对于`nwrite`也是如此）。

环形缓冲
直线计数
这种缓冲区的
实现方法在
console中也出
现过

让我们假设对和的调用同时发生在两个不同的CPU上。
`Pipewrite` (`kernel/pipe.c:77`) 从获取管道锁开始，它保护计数、数据及其相关不变量。`Piperead` (`kernel/pipe.c:103`) 然后也尝试获取锁，但无法实现。它在`acquire` (`kernel/spinlock.c:22`) 中旋转等待锁。当等待时，`pipewrite`遍历被写入的字节 (`addr[0..n-1]`)，依次将每个字节添加到管道中 (`kernel/pipe.c:95`)。在这个循环中缓冲区可能会被填满 (`kernel/pipe.c:85`)。在这种情况下，`pipewrite`调用wakeup来提醒所有处于睡眠状态的读进程缓冲区中有数据等待，然后在`&pi->nwrite`上睡眠，等待读进程从缓冲区中取出一些字节。作为使进程进入睡眠状态的一部分，`Sleep`释放`pi->lock`。

现在`pi->lock`可用，`piperead`设法获取它并进入其临界区域：它发现`pi->nread != pi->nwrite` (`kernel/pipe.c:110`) (`pipewrite`进入睡眠状态是因为`pi->nwrite == pi->nread+PIPESIZE` (`kernel/pipe.c:85`))，因此它进入`for`循环，从管道中复制数据 (`kernel/pipe.c:117`)，并根据复制的字节数增加`nread`。那些读出的字节就可供写入，因此调用wakeup (`kernel/pipe.c:124`) 返回之前唤醒所有休眠的写进程。`Wakeup`寻找一个在`&pi->nwrite`上休眠的进程，该进程正在运行，但在缓冲区填满时停止。它将该进程标记为RUNNABLE。

管道代码为读者和写者使用单独的睡眠通道 (`pi->nread`和`pi->nwrite`)；这可能使系统在有许多读者和写者等待同一管道这种不太可能的情况下更加高效。管道代码在检查休眠条件的循环中休眠；如果有多个读者或写者，那么除了第一个醒来的进程之外，所有进程都会看到条件仍然错误，并再次睡眠。

↓
告诉GUN编译器函数不会返回

system call	int wait(int *) int exit (int) __attribute__((noreturn)) int kill (int)
----------------	--

7.8 代码：wait, exit和kill

Sleep和wakeup可用于多种等待。第一章介绍的一个有趣的例子是子进程exit和父进程wait之间的交互。在子进程死亡时，父进程可能已经在wait中休眠，或者正在做其他事情；在后一种情况下，随后的wait调用必须观察到子进程的死亡，可能是在子进程调用exit后很久。xv6记录子进程终止直到wait观察到它的方式是让exit将调用方置于ZOMBIE状态，在那里它一直保持到父进程的wait注意到它，将子进程的状态更改为UNUSED，复制子进程的exit状态码，并将子进程ID返回给父进程。如果父进程在子进程之前退出，则父进程将子进程交给init进程，init进程将永久调用wait；因此，每个子进程退出后都有一个父进程进行清理。（主要的实现挑战是父级和子级wait和exit，以及exit和exit之间可能存在竞争和死锁。）

calling (即np->lock)

Wait使用调用进程的p->lock作为条件锁，以避免丢失唤醒，并在开始时获取该锁(kernel/proc.c:398)。然后它扫描进程表。如果它发现一个子进程处于ZOMBIE状态，它将释放该子进程的资源及其proc结构体，将该子进程的退出状态码复制到提供给wait的地址（如果addr!=0）并返回该子进程的进程ID。如果wait找到子进程但没有子进程退出，它将调用sleep以等待其中一个退出（kernel/proc.c:445），然后再次扫描。这里，sleep中释放的条件锁是等待进程的p->lock，这是上面提到的特例。注意，wait通常持有两个锁：它在试图获得任何子进程的锁之前先获得自己的锁；因此，整个xv6都必须遵守相同的锁定顺序（父级，然后是子级），以避免死锁。

遍历每个子进程

freeproc(np)

sleep(p, &wait_lock)

Wait查看每个进程的np->parent以查找其子进程。它使用np->parent时不持有np->lock，这违反了通常的规则，即共享变量必须受到锁的保护。np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，因为这将违反上述顺序。这种情况下无锁检查np->parent似乎是安全的：进程的parent字段仅由其父进程更改，因此如果np->parent==p为true，除非当前流程更改它，否则该值无法被更改，

xstate

Exit (kernel/proc.c:333) 记录退出状态码，释放一些资源，将所有子进程提供给init进程，在父进程处于等待状态时唤醒父进程，将调用方标记为僵尸进程(zombie)，并永久地让出CPU。最后的顺序有点棘手。退出进程必须在将其状态设置为ZOMBIE并唤醒父进程时持有其父进程的锁，因为父进程的锁是防止在wait中丢失唤醒的条件锁。子级还必须持有自己的p->lock，否则父级可能会看到它处于ZOMBIE状态，并在它仍运行时释放它。锁获取顺序对于避免死锁很重要：因为wait先获取父锁再获取子锁，所以exit必须使用相同的顺序。

wait()在新版
本中作了简化，
使用了全局锁
wait_lock，避
免了索取父进程
的p->lock。也沒
有专门的唤醒
函数 wakeup!

Exit调用一个专门的唤醒函数wakeup1，该函数仅唤醒父进程，且父进程必须正在wait中休眠（kernel/proc.c:598）。在将自身状态设置为ZOMBIE之前，子进程唤醒父进程可能看起来不正确，但这是安全的：虽然wakeup1可能会导致父进程运行，但wait中的循环在scheduler释放子进程的p->lock之前无法检查子进程，所以wait在exit将其状态设置为ZOMBIE（kernel/proc.c:386）之前不能查看退出进程。

exit允许进程自行终止，而kill (kernel/proc.c:611) 允许一个进程请求另一个进程终止。对于kill来说，直接销毁受害者进程（即要杀死的进程）太复杂了，因为受害者可能在另一个CPU上执行，也许是在更新内核数据结构的敏感序列中间。因此，

离开内核时也判断
p->killed吗？

不是指 wakeup)
是 if (p->state == SLEEPING)
p->state == RUNNABLE

具体被kill的时
刻其实不明确。
有一种“不涉及内
核就放任进程”
的意味

kill的工作量很小：它只是设置受害者的p->killed，如果它正在睡眠，则唤醒它。
受害者进程终将进入或离开内核，此时，如果设置了p->killed，usertrap中的代码将调用exit。如果受害者在用户空间中运行，它将很快通过进行系统调用或由于计时器（或其他设备）中断而进入内核。

如果受害者进程在sleep中，kill对wakeup的调用将导致受害者从sleep中返回。这存在潜在的危险，因为等待的条件可能不为真。但是，xv6对sleep的调用总是封装在while循环中，该循环在sleep返回后重新测试条件。一些对sleep的调用还在循环中测试p->killed，如果它被设置，则放弃当前活动。只有在这种放弃是正确的情况下才能这样做。例如，如果设置了killed标志，则管道读写代码返回；最终代码将返回到陷阱，陷阱将再次检查标志并退出。

一些xv6的sleep循环不检查p->killed，因为代码在应该是原子操作的多步系统调用的中间。virtio驱动程序（kernel/virtio_disk.c:242）就是一个例子：它不检查p->killed，因为一个磁盘操作可能是文件系统保持正确状态所需的一组写入操作之一。等待磁盘I/O时被杀死的进程将不会退出，直到它完成当前系统调用并且usertrap看到killed标志

7.9 真实世界

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。这一策略被称为轮询调度（round robin）。真实的操作系统实施更复杂的策略，例如，允许进程具有优先级。其思想是调度器将优先选择可运行的高优先级进程，而不是可运行的低优先级进程。这些策略可能变得很复杂，因为常常存在相互竞争的目标：例如，操作系统可能希望保证公平性和高吞吐量。此外，复杂的策略可能会导致意外的交互，例如优先级反转（priority inversion）和航队（convoys）。当低优先级进程和高优先级进程共享一个锁时，可能会发生优先级反转，当低优先级进程持有该锁时，可能会阻止高优先级进程前进。当许多高优先级进程正在等待一个获得共享锁的低优先级进程时，可能会形成一个长的等待进程航队；一旦航队形成，它可以持续很长时间。为了避免此类问题，在复杂的调度器中需要额外的机制。**可抢占？**

睡眠和唤醒是一种简单有效的同步方法，但还有很多其他方法。所有这些问题中的第一个挑战是避免我们在本章开头看到的“丢失唤醒”问题。原始Unix内核的sleep只是禁用了中断，这就足够了，因为Unix运行在单CPU系统上。因为xv6在多处理器上运行，所以它为sleep添加了一个显式锁。FreeBSD的msleep采用了同样的方法。Plan 9的sleep使用一个回调函数，该函数在马上睡眠时获取调度锁，并在运行中持有；该函数用于在最后时刻检查睡眠条件，以避免丢失唤醒。Linux内核的sleep使用一个显式的进程队列，称为等待队列，而不是等待通道；队列有自己内部的锁。

（在wakeup中扫描整个进程列表以查找具有匹配chan的进程效率低下。一个更好的解决方案是用一个数据结构替换sleep和wakeup中的chan，该数据结构包含在该结构上休眠的进程列表，例如Linux的等待队列。）Plan 9的sleep和wakeup将该结构称为集结点（rendezvous point）或Rendez。许多线程库引用与条件变量相同的结构；在这种情况下，sleep和wakeup操作称为wait和signal。所有这些机制都有一个共同的特点：睡眠条件受到某种在睡眠过程中原子级释放的锁的保护。

wakeup的实现会唤醒在特定通道上等待的所有进程，可能有许多进程在等待该特定通道。操作系统将安排所有这些进程，它们将竞相检查睡眠条件。进程的这种行为有时被称为惊群效应（thundering herd），最好避免。大多数条件变量都有两个用于唤醒的原语：signal用于唤醒一个进程；broadcast用于唤醒所有等待进程。

信号量（Semaphores）通常用于同步。计数count通常对应于管道缓冲区中可用的字节数或进程具有的僵尸子进程数。使用显式计数作为抽象的一部分可以避免“丢失唤醒”问题：使用显式计数记录已经发生wakeup的次数。计数还避免了虚假唤醒和惊群效应问题。

终止进程并清理它们在xv6中引入了很多复杂性。在大多数操作系统中甚至更复杂，因为，例如，受害者进程可能在内核深处休眠，而展开其栈空间需要非常仔细的编程。许多操作系统使用显式异常处理机制（如longjmp）来展开栈。此外，还有其他事件可能导致睡眠进程被唤醒，即使它等待的事件尚未发生。例如，当一个Unix进程处于休眠状态时，另一个进程可能会向它发送一个signal。在这种情况下，进程将

从中断的系统调用返回，返回值为-1，错误代码设置为EINTR。应用程序可以检查这些值并决定执行什么操作。xv6不支持信号，因此不会出现这种复杂性。

xv6对kill的支持并不完全令人满意：有一些sleep循环可能应该检查p->killed。一个相关的问题是，即使对于检查p->killed的sleep循环，sleep和kill 后者可能在受害者之间也存在竞争；后者可能会设置p->killed，并试图在受害者的循环检查p->killed之后但在调用sleep之前尝试唤醒受害者。如果出现此问题，受害者将不会注意到p->killed，直到其等待的条件发生。这可能比正常情况要晚一点（例如，当virtio驱动程序返回受害者正在等待的磁盘块时）或永远不会发生（例如，如果受害者正在等待来自控制台的输入，但用户没有键入任何输入）。

注：上节中说到kill的工作方式，kill设置p->killed，如果遇到进程正在休眠，则会唤醒它，此后在usertrap中检测p->killed，并使进程退出

而如果像上面说的，在检查p->killed之后调用sleep之前唤醒受害者进程，那么接下来执行sleep就会导致进程无法进入内核，无法在usertrap中退出，而必须等待所需事件的发生再次唤醒

一个实际的操作系统将在固定时间内使用空闲列表找到自由的proc结构体，而不是allocproc中的线性时间搜索；xv6使用线性扫描是为了简单起见。

7.10 练习

1. Sleep 必须检查 `lk != &p->lock` 来避免死锁(**kernel/proc.c:558-561**). 假设通过将

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

替换为

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

来消除特殊情况，这样做将会破坏 sleep。是如何破坏的呢？

1. 大多数进程清理可以通过 `exit` 或 `wait` 来完成。事实证明，必须是 `exit` 作为关闭打开的文件的那个。为什么？答案涉及管道。
2. 在 xv6 中实现信号量而不使用 `sleep` 和 `wakeup`（但可以使用自旋锁）。用信号量取代 xv6 中 `sleep` 和 `wakeup` 的使用。判断结果。
3. 修复上面提到的 `kill` 和 `sleep` 之间的竞争，这样在受害者的 `sleep` 循环检查 `p->killed` 之后但在调用 `sleep` 之前发生的 `kill` 会导致受害者放弃当前系统调用。
4. 设计一个计划，使每个睡眠循环检查 `p->killed`，这样，例如，`virtio` 驱动程序中的一个进程可以在被另一个进程终止时从 `while` 循环快速返回。
5. 修改 xv6，使其在从一个进程的内核线程切换到另一个线程时仅使用一次上下文切换，而不是通过调度器线程进行切换。屈服 (`yield`) 线程需要选择下一个线程本身并调用 `swtch`。挑战在于：防止多个内核意外执行同一个线程；获得正确的锁；避免死锁。
6. 修改 xv6 的调度程序，以便在没有进程可运行时使用 RISC-V 的 `WFI`（`wait for interrupt`，等待中断）指令。尽量确保在任何时候有可运行的进程等待运行时，没有核心在 `WFI` 中暂停。
7. 锁 `p->lock` 保护许多不变量，当查看受 `p->lock` 保护的特定 xv6 代码段时，可能很难确定保护的是哪个不变量。通过将 `p->lock` 拆分为多个锁，设计一个更清晰的计划。

第八章 文件系统

文件系统的目的是组织和存储数据。文件系统通常支持用户和应用程序之间的数据共享，以及持久性，以便在重新启动后数据仍然可用。

xv6文件系统提供类似于**Unix**的文件、目录和路径名（参见第1章），并将其数据存储在**virtio**磁盘上以便持久化（参见第4章）。文件系统解决了几个难题：

注：完整计算机中的CPU被支撑硬件包围，其中大部分是以I/O接口的形式。**xv6**是以**qemu**的“-machine virt”选项模拟的支撑硬件编写的。这包括**RAM**、包含引导代码的**ROM**、一个到用户键盘/屏幕的串行连接，以及一个用于存储的**磁盘**。

- 文件系统需要**磁盘**上的数据结构来表示目录和文件名称树，记录保存每个文件内容的块的标识，以及记录磁盘的哪些区域是空闲的。
- 文件系统必须支持崩溃恢复（crash recovery）。也就是说，如果发生崩溃（例如，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一系列更新，并使磁盘上的数据结构不一致（例如，一个块在某个文件中使用但同时仍被标记为空闲）。
- 不同的进程可能同时在文件系统上运行，因此文件系统代码必须协调以保持不变量。
- 访问**磁盘**的速度比访问内存慢几个数量级，因此文件系统必须保持常用块的内存缓存。

本章的其余部分将解释**xv6**如何应对这些挑战。

8.1 概述

xv6文件系统实现分为七层，如图8.1所示。磁盘层读取和写入virtio硬盘上的块。缓冲区高速缓存层缓存磁盘块并同步对它们的访问，确保每次只有一个内核进程可以修改存储在任何特定块中的数据。日志记录层允许更高层在一次事务（transaction）中将更新包装到多个块，并确保在遇到崩溃时自动更新这些块（即，所有块都已更新或无更新）。索引结点层提供单独的文件，每个文件表示为一个索引结点，其中包含唯一的索引号（i-number）和一些保存文件数据的块。目录层将每个目录实现为一种特殊的索引结点，其内容是一系列目录项，每个目录项包含一个文件名和索引号。路径名层提供了分层路径名，如**/usr/rtm/xv6/fs.c**，并通过递归查找来解析它们。文件描述符层使用文件系统接口抽象了许多Unix资源（例如，管道、设备、文件等），简化了应用程序员的工作。



图8.1 xv6文件系统的层级

文件系统必须有将索引节点和内容块存储在磁盘上哪些位置的方案。为此，xv6将磁盘划分为几个部分，如图8.2所示。文件系统不使用块0（它保存引导扇区）。块1称为超级块：它包含有关文件系统的元数据（文件系统大小（以块为单位）、数据块数、索引节点数和日志中的块数）。从2开始的块保存日志。日志之后是索引节点，每个块有多个索引节点。然后是位图块，跟踪正在使用的数据块。其余的块是数据块：每个要么在位图块中标记为空闲，要么保存文件或目录的内容。超级块由一个名为mkfs的单独的程序填充，该程序构建初始文件系统。

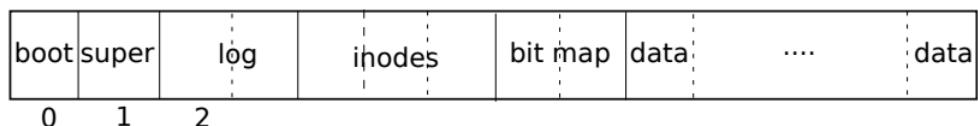


Figure 8.2: Structure of the xv6 file system.

本章的其余部分将从缓冲区高速缓存层开始讨论每一层。注意那些在较低层次上精心选择的抽象可以简化较高层次的设计的情况。

8.2 Buffer cache层

磁盘块的
内存副本

Buffer cache有两个任务：

1. 同步对磁盘块的访问，以确保磁盘块在内存中只有一个副本，并且一次只有一个内核线程使用该副本
2. 缓存常用块，以便不需要从慢速磁盘重新读取它们。代码在**bio.c**中。

Buffer cache层导出的主接口主要是**bread**和**bwrite**；前者获取一个buf，其中包含一个可以在内存中读取或修改的块的副本，后者将修改后的缓冲区写入磁盘上的相应块。内核线程必须通过调用**brelse**释放缓冲区。Buffer cache每个缓冲区使用一个睡眠锁，以确保每个缓冲区（因此也是每个磁盘块）每次只被一个线程使用；**bread**返回一个上锁的缓冲区，**brelse**释放该锁。

让我们回到Buffer cache。Buffer cache中保存磁盘块的缓冲区数量固定，这意味着如果文件系统请求还未存放在缓存中的块，Buffer cache必须回收当前保存其他块内容的缓冲区。Buffer cache为新块回收最近使用最少的缓冲区。这样做的原因是认为最近使用最少的缓冲区是最不可能近期再次使用的缓冲区。LRU

思想和“固定分配局部置换的请求页相同”

8.3 代码：Buffer cache

Buffer cache是以双链表表示的缓冲区。`main` (`kernel/main.c:27`) 调用的函数 `binit` 使用静态数组 `buf` (`kernel/bio.c:43-52`) 中的 `NBUF` 个缓冲区初始化列表。对 Buffer cache 的所有其他访问都通过 `bcache.head` 引用链表，而不是 `buf` 数组。

缓冲区有两个与之关联的状态字段。字段 `valid` 表示缓冲区是否包含块的副本。字段 `disk` 表示缓冲区内容是否已交给磁盘，这可能会更改缓冲区（例如，将数据从磁盘写入 `data`）。

`Bread` (`kernel/bio.c:93`) 调用 `bget` 为给定扇区 (`kernel/bio.c:97`) 获取缓冲区。如果缓冲区需要从磁盘进行读取，`bread` 会在返回缓冲区之前调用 `virtio_disk_rw` 来执行此操作。

`Bget` (`kernel/bio.c:59`) 扫描缓冲区列表，查找具有给定设备和扇区号 (`kernel/bio.c:65-73`) 的缓冲区。如果存在这样的缓冲区，`bget` 将获取缓冲区的睡眠锁。然后 `Bget` 返回锁定的缓冲区。

如果对于给定的扇区没有缓冲区，`bget` 必须创建一个，这可能会重用包含其他扇区的缓冲区。它再次扫描缓冲区列表，查找未在使用中的缓冲区 (`b->refcnt = 0`)：任何这样的缓冲区都可以使用。`Bget` 编辑缓冲区元数据以记录新设备和扇区号，并获取其睡眠锁。注意，`b->valid = 0` 的布置确保了 `bread` 将从磁盘读取块数据，而不是错误地使用缓冲区以前的内容。

每个磁盘扇区最多有一个缓存缓冲区是非常重要的，并且因为文件系统使用缓冲区上的锁进行同步，可以确保读者看到写操作。`Bget` 的从第一个检查块是否缓存的循环到第二个声明块现在已缓存（通过设置 `dev`、`blockno` 和 `refcnt`）的循环，一直持有 `bcache.lock` 来确保此不变量。这会导致检查块是否存在以及（如果不存在）指定一个缓冲区来存储块具有原子性。

(`bget` 在 `bcache.lock` 临界区域之外获取缓冲区的睡眠锁是安全的，因为非零 `b->refcnt` 防止缓冲区被重新用于不同的磁盘块。) 睡眠锁保护块缓冲内容的读写，而 `bcache.lock` 保护有关缓存哪些块的信息。

如果所有缓冲区都处于忙碌，那么太多进程同时执行文件系统调用；`bget` 将会 `panic`。（一个更优雅的响应可能是在缓冲区空闲之前休眠，尽管这样可能会出现死锁。）

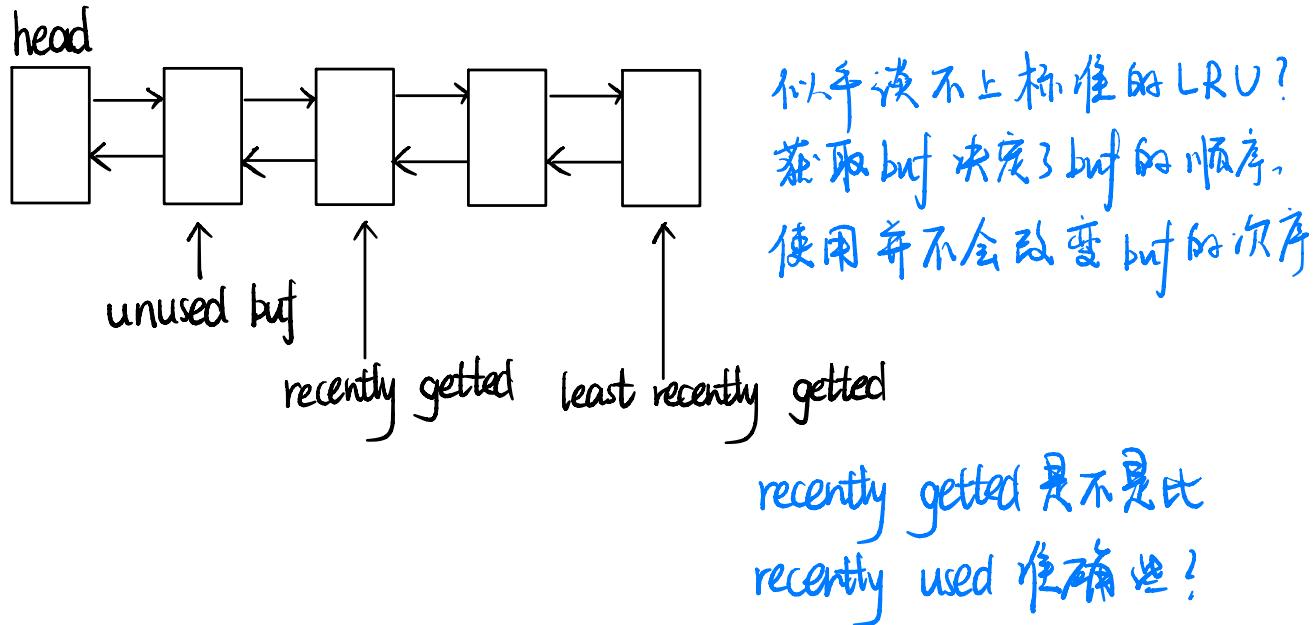
经典的“持有-觊觎”死锁

一旦 `bread` 读取了磁盘（如果需要）并将缓冲区返回给其调用者，调用者就可以独占使用缓冲区，并可以读取或写入数据字节。如果调用者确实修改了缓冲区，则必须在释放缓冲区之前调用 `bwrite` 将更改的数据写入磁盘。`Bwrite` (`kernel/bio.c:107`) 调用 `virtio_disk_rw` 与磁盘硬件对话。

当调用方使用完缓冲区后，它必须调用 `brelease` 来释放缓冲区 (`brelease` 是 `b-release` 的缩写，这个名字很隐晦，但值得学习：它起源于 Unix，也用于 BSD、Linux 和

若修改为
`bget`：
`panic` → `sleep`
`brelease`：
`wakeup`
proc A 和 proc B
均持有 `buf`，且
试图获取新
`buf`。此刻 A &
B 都 `sleep`，若无 proc
C 释放 `buf`，则
A & B 发生死锁

Solaris）。`brelse` (`kernel/bio.c:117`) 释放睡眠锁并将缓冲区移动到链表的前面（`kernel/bio.c:128-133`）。移动缓冲区会使列表按缓冲区的使用频率排序（意思是释放）：列表中的第一个缓冲区是最近使用的，最后一个是最少使用的。`bget`中的两个循环利用了这一点：在最坏的情况下，对现有缓冲区的扫描必须处理整个列表，但首先检查最新使用的缓冲区（从`bcache.head`开始，然后是下一个指针），在引用局部性良好的情况下将减少扫描时间。~~选择要重用的缓冲区时，通过自后向前扫描（跟随`prev`指针）选择最近使用最少的缓冲区。~~



8.4 日志层

比如文件写入会
涉及 data block
inode, bitmap
的写入

文件系统设计中最有趣的问题之一是崩溃恢复。出现此问题的原因是，许多文件系统操作都涉及到对磁盘的多次写入，并且在完成写操作的部分子集后崩溃可能会使磁盘上的文件系统处于不一致的状态。例如，假设在文件截断（将文件长度设置为零并释放其内容块）期间发生崩溃。根据磁盘写入的顺序，崩溃可能会留下对标记为空闲的内容块的引用的inode，也可能留下已分配但未引用的内容块。

后者相对来说是良性的，但引用已释放块的inode在重新启动后可能会导致严重问题。重新启动后，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同一块。如果xv6支持多个用户，这种情况可能是一个安全问题，因为旧文件的所有者将能够读取和写入新文件中的块，而新文件的所有者是另一个用户。

- ① logwrite
- ② commits
- ③ install
- ④ clean log

xv6通过简单的日志记录形式解决了文件系统操作期间的崩溃问题。xv6系统调用不会直接写入磁盘上的文件系统数据结构。相反，它会在磁盘上的log（日志）中放置它希望进行的所有磁盘写入的描述。①一旦系统调用记录了它的所有写入操作，②它就会向磁盘写入一条特殊的commit（提交）记录，表明日志包含一个完整的操作。此时，③系统调用将操作复制到磁盘上的文件系统数据结构。④完成这些写入后，系统调用将擦除磁盘上的日志。
→ 写入(logged block)

如果系统崩溃并重新启动，则在运行任何进程之前，文件系统代码将按如下方式从崩溃中恢复。（如果日志标记为包含完整操作，则恢复代码会将操作复制到磁盘文件系统中它们所属的位置。如果日志没有标记为包含完整操作，则恢复代码将忽略该日志。恢复代码通过擦除日志完成。）

为什么xv6的日志解决了文件系统操作期间的崩溃问题？如果崩溃发生在操作提交之前，那么磁盘上的~~log~~将不会被标记为已完成，恢复代码将忽略它，并且磁盘的状态将如同操作尚未启动一样。如果崩溃发生在操作提交之后，则恢复将重播操作的所有写入操作，如果操作已开始将它们写入磁盘数据结构，则~~可能会~~重复这些操作。在任何一种情况下，日志都会使操作在崩溃时成为原子操作：恢复后，要么操作的所有写入都显示在磁盘上，要么都不显示。

1024B/block

boot	super block	log	inode	bitmap	data block	
0	1	2	31 32	44 45	46	

8.5 日志设计

日志驻留在超级块中指定的已知固定位置。它由一个头块（header block）和一系列更新块的副本（logged block）组成。头块包含一个扇区号数组（每个logged block对应一个扇区号）以及日志块的计数。（磁盘上的头块中的计数或者为零，表示日志中没有事务；或者为非零，表示日志包含一个完整的已提交事务，并具有指定数量的logged block。）在事务提交（commit）时xv6才向头块写入数据，在此之前不会写入，并在将logged blocks复制到文件系统后将计数设置为零。因此，事务中途崩溃将导致日志头块中的计数为零；提交后的崩溃将导致非零计数。

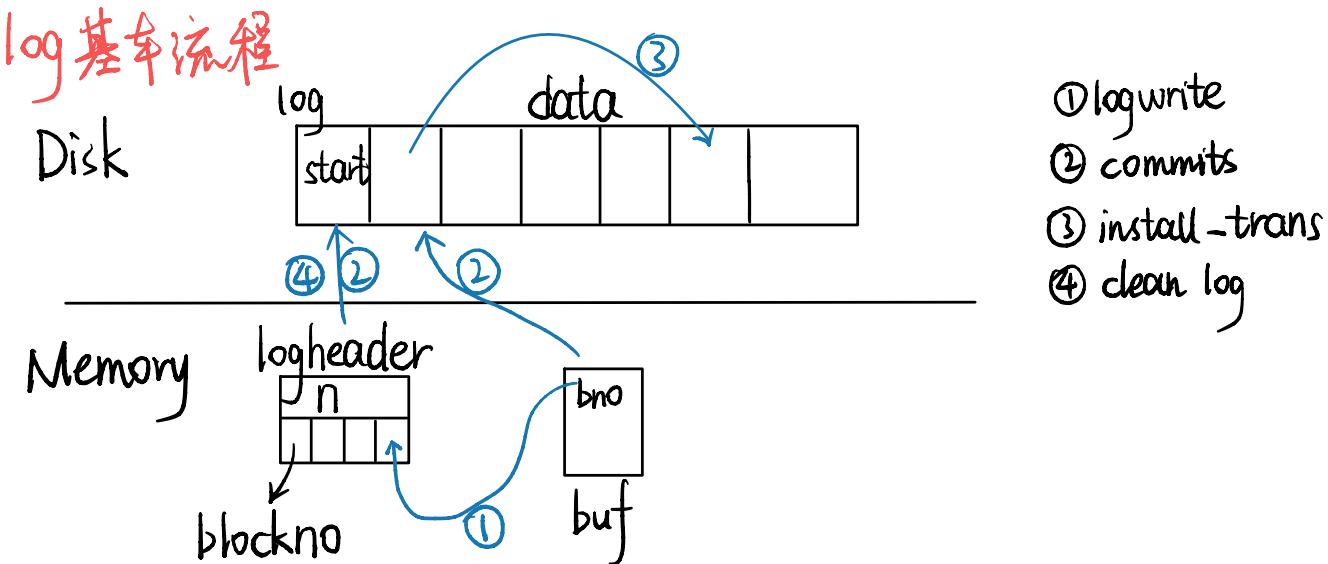
注：logged block表示已经记录了操作信息的日志块，而log block仅表示日志块

sector not block? 因为 xv6 没有实际的 disk, 所以这里的 sector number = block number?

每个系统调用的代码都指示写入序列的起止，考虑到崩溃，写入序列必须具有原子性。为了允许不同进程并发执行文件系统操作，日志系统可以将多个系统调用的写入累积到一个事务中。因此，单个提交可能涉及多个完整系统调用的写入。为了避免在事务之间拆分系统调用，日志系统仅在没有文件系统调用进行时提交。把一个系统调用拆分到不同事务中

同时提交多个事务的想法称为组提交（group commit）。组提交减少了磁盘操作的数量，因为成本固定的一次提交分摊了多个操作。组提交还同时为磁盘系统提供更多并发写操作，可能允许磁盘在一个磁盘旋转时间内写入所有这些操作。xv6的virtio驱动程序不支持这种批处理，但是xv6的文件系统设计允许这样做。

xv6在磁盘上留出固定的空间来保存日志。事务中系统调用写入的块总数必须可容纳于该空间。这导致两个后果：任何单个系统调用都不允许写入超过日志空间的不同块。这对于大多数系统调用来说都不是问题，但其中两个可能会写入许多块：`write`和`unlink`。一个大文件的`write`可以写入多个数据块和多个位图块以及一个inode块；`unlink`大文件可能会写入许多位图块和inode。`xv6`的`write`系统调用将大的写入分解为适合日志的多个较小的写入，`unlink`不会导致此问题，因为实际上xv6文件系统只使用一个位图块。日志空间有限的另一个后果是，除非确定系统调用的写入将可容纳于日志中剩余的空间，否则日志系统无法允许启动系统调用。



8.6 代码：日志

在系统调用中一个典型的日志使用就像这样：

每个xv6的文件
系统调用都是
这个结构

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

`begin_op` (`kernel/log.c:126`) 等待直到日志系统当前未处于提交中，并且直到有足够的未被占用的日志空间来保存此调用的写入。`log.outstanding` 统计预定了日志空间的系统调用数；为此保留的总空间为 `log.outstanding` 乘以 `MAXOPBLOCKS`。递增 `log.outstanding` 会预定空间并防止在此系统调用期间发生提交。代码保守地假设每个系统调用最多可以写入 `MAXOPBLOCKS` 个不同的块。

`blockno`

`log_write` (`kernel/log.c:214`) 充当 `bwrite` 的代理。它将块的扇区号记录在内存中，在磁盘上的日志中预定一个槽位，并调用 `bpin` 将缓存固定在 `block cache` 中，以防止 `block cache` 将其逐出。

注：固定在 `block cache` 是指在缓存不足需要考虑替换时，不会将这个 `block` 换出，因为事务具有原子性：假设块 45 被写入，将其换出的话需要写入磁盘中文件系统对应的位置，而日志系统要求所有内存必须都存入日志，最后才能写入文件系统。

`bpin` 是通过增加引用计数防止块被换出的，之后需要再调用 `bunpin`

在提交之前，块必须留在缓存中：在提交之前，缓存的副本是修改的唯一记录；只有在提交后才能将其写入磁盘上的位置；同一事务中的其他读取必须看到修改。[?] `log_write` 会注意到在单个事务中多次写入一个块的情况，并在日志中为该块分配相同的槽位。这种优化通常称为合并（absorption）。例如，包含多个文件 `inode` 的磁盘块在一个事务中被多次写入是很常见的。通过将多个磁盘写入合并到一个磁盘中，文件系统可以节省日志空间并实现更好的性能，因为只有一个磁盘块副本必须写入磁盘。

注：日志需要写入磁盘，以便重启后读取，但日志头块和日志数据块也会在 `block cache` 中有一个副本

`end_op` (`kernel/log.c:146`) 首先减少未完成系统调用的计数。如果计数现在为零，则通过调用 `commit()` 提交当前事务。这一过程分为四个阶段。`write_log()` (`kernel/log.c:178`) 将事务中修改的每个块从缓冲区缓存复制到磁盘上日志槽位中。`write_head()` (`kernel/log.c:102`) 将头块写入磁盘：这是提交点，写入后的崩溃将导致从日志恢复重演事务的写入操作。`install_trans` (`kernel/log.c:69`) 从日志中读取

把计数为0的日志头（in memory）
写入 block [log_start] (on disk)

每个块，并将其写入文件系统中的适当位置。最后，`end_op`写入计数为零的日志头；这必须在下一个事务开始写入日志块之前发生，以便崩溃不会导致使用一个事务的头块和后续事务的日志块进行恢复。

~~install-tran~~ `recover_from_log` (`kernel/log.c:116`) 是由 `initlog` (`kernel/log.c:55`) 调用的，而它又是在第一个用户进程运行 (`kernel/proc.c:539`) 之前的引导期间由 `fsinit` (`kernel/fs.c:42`) 调用的。它读取日志头，如果头中指示日志包含已提交的事务，则模拟 `end_op` 的操作。
~~的宾率为1.1~~
~~为不需要~~

`bunpin`

日志的一个示例使用发生在 `filewrite` (`kernel/file.c:135`) 中。事务如下所示：

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

这段代码被包装在一个循环中，该循环一次将大的写操作分解为几个扇区的单个事务，以避免日志溢出。作为此事务的一部分，对 `writei` 的调用写入许多块：文件的 `inode`、一个或多个位图块以及一些数据块。

8.7 代码：块分配器 Block Allocator

文件和目录内容存储在磁盘块中，磁盘块必须从空闲池中分配。`xv6`的块分配器在磁盘上维护一个空闲位图，每一位代表一个块。0表示对应的块是空闲的；1表示它正在使用中。程序`mkfs`设置对应于引导扇区、超级块、日志块、`inode`块和位图块的比特位。

块分配器提供两个功能：`alloc`分配一个新的磁盘块，`free`释放一个块。
`Balloc`中位于`kernel/fs.c:71`的循环从块0到`sb.size`（文件系统中的块数）遍历每个块。它查找位图中位为零的空闲块。如果`alloc`找到这样一个块，它将更新位图并返回该块。为了提高效率，循环被分成两部分。外部循环读取位图中的每个块。内部循环检查单个位图块中的所有BPB位。由于任何一个位图块在`buffer cache`中一次只允许一个进程使用，因此，如果两个进程同时尝试分配一个块，可能会发生争用。

`Bfree` (`kernel/fs.c:90`) 找到正确的位图块并清除正确的位。同样，`bread`和`brelse`隐含的独占使用避免了显式锁定的需要。

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

提高效率是指
可以块再行？

8.8 索引结点层

术语inode（即索引结点）可以具有两种相关含义之一。它可能是指包含文件大小和数据块编号列表的磁盘上的数据结构。或者“inode”可能指内存中的inode，它包含磁盘上inode的副本以及内核中所需的额外信息。

{
磁盘 inode
内存 inode

计算inode所在 blockno:
$$32 + \frac{64n}{1024}$$

32~44 block 64B

磁盘上的inode都被打包到一个称为inode块的连续磁盘区域中。每个inode的大小都相同，因此在给定数字n的情况下，很容易在磁盘上找到第n个inode。事实上，这个编号n，称为inode number或i-number，是在具体实现中标识inode的方式。

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

内核将活动的inode集合保存在内存中；`struct inode` (`kernel/file.h:17`) 是磁盘上`struct dinode`的内存副本。只有当有C指针引用某个inode时，内核才会在内存中存储该inode。（`ref`字段统计引用内存中inode的C指针的数量，如果引用计数降至零，内核将从内存中丢弃该inode。）`iget`和`iput`函数分别获取和释放指向inode的指针，修改引用计数。指向inode的指针可以来自文件描述符、当前工作目录和如`exec`的瞬态内核代码。

itable lock?

xv6的inode代码中有四种锁或类似锁的机制。`icache.lock`保护以下两个不变量：`inode`最多在缓存中出现一次；缓存`inode`的`ref`字段记录指向缓存`inode`的内存指针数量。每个内存中的`inode`都有一个包含睡眠锁的`lock`字段，它确保以独占方式访问`inode`的字段（如文件长度）以及`inode`的文件或目录内容块。如果`inode`的`ref`大于零，则会导致系统在cache中维护`inode`，而不会对其他`inode`重用此缓存项。最后，每个`inode`都包含一个`nlink`字段（在磁盘上，如果已被缓存则已被拷贝到内存中），该字段统计引用文件的目录项的数量；如果`inode`的链接计数大于零，xv6将不会释放`inode`。→ i.e. delete file

iget 只负责在
itable 中占位，
不读取 Disk

`iget()`返回的`struct inode`指针在相应的`iput()`调用之前保证有效：`inode`不会被删除，指针引用的内存也不会被其他`inode`重用。`iget()`提供对`inode`的非独占访问，因此可以有许多指向同一`inode`的指针。文件系统代码的许多部分都依赖于`iget()`的这种行为，既可以保存对`inode`的长期引用（如打开的文件和当前目录），也可以防止争用，同时避免操纵多个`inode`（如路径名查找）的代码产生死锁。

`iget`返回的`struct inode`可能没有任何有用的内容。为了确保它保存磁盘`inode`的副本，代码必须调用`ilock`。这将锁定`inode`（以便没有其他进程可以对其进行`ilock`），并从磁盘读取尚未读取的`inode`。`iunlock`释放`inode`上的锁。（将`inode`指针的获取与锁定分离有助于在某些情况下避免死锁，例如在目录查找期间。多个进程可以持有指向`iget`返回的`inode`的C指针，但一次只能有一个进程锁定`inode`。）

inode缓存只缓存内核代码或数据结构持有C指针的inode。它的主要工作实际上是同步多个进程的访问；缓存是次要的。如果经常使用inode，在inode缓存不保留它的情

况下buffer cache可能会将其保留在内存中。inode缓存是直写的，这意味着修改已缓存inode的代码必须立即使用*iupdate*将其写入磁盘。

8.9 代码：Inodes

为了分配新的inode（例如，在创建文件时），xv6调用 **ialloc** (**kernel/fs.c:196**)。**Ialloc**类似于**balloc**：它一次一个块地遍历磁盘上的索引节点结构体，查找标记为空闲的一个。当它找到一个时，它通过将新**type**写入磁盘来声明它，然后末尾通过调用 **iget** (**kernel/fs.c:210**) 从inode缓存返回一个条目。**ialloc**的正确操作取决于这样一个事实：一次只有一个进程可以保存对bp的引用：**ialloc**可以确保其他进程不会同时看到inode可用并尝试声明它。

Iget (**kernel/fs.c:243**) 在inode缓存中查找具有所需设备和inode编号的活动条目 (**ip->ref > 0**)。如果找到一个，它将返回对该inode的新引用 (**kernel/fs.c:252-256**)。在**iget**扫描时，它会记录第一个空槽 (**kernel/fs.c:257-258**) 的位置，如果需要分配缓存项，它会使用这个槽。**slot**

(在读取或写入inode的元数据或内容之前，代码必须使用**ilock**锁定inode。)
Illock (**kernel/fs.c:289**) 为此使用睡眠锁。一旦**ilock**以独占方式访问inode，它将根据需要从磁盘（更可能是buffer cache）读取inode。函数**iunlock** (**kernel/fs.c:317**) 释放睡眠锁，这可能会导致任何睡眠进程被唤醒。

Iput (**kernel/fs.c:333**) 通过减少引用计数 (**kernel/fs.c:356**) 释放指向inode的C指针。如果这是最后一次引用，inode缓存中该inode的槽现在将是空闲的，可以重用于其他inode。

如果**iput**发现没有指向inode的C指针引用，并且inode没有指向它的链接（发生于无目录），则必须释放inode及其数据块。**Iput**调用**itrunc**将文件截断为零字节，释放数据块；将索引节点类型设置为0（未分配）；并将inode写入磁盘
(**kernel/fs.c:338**)。

iput中释放inode的锁定协议值得仔细研究。一个危险是并发线程可能正在**ilock**中等待使用该inode（例如，读取文件或列出目录），并且不会做好该inode已不再被分配的准备。这不可能发生，因为如果缓存的inode没有链接，并且**ip->ref**为1，那么系统调用就无法获取指向该inode的指针。那一个引用是调用**iput**的线程所拥有的引用。的确，**iput**在**icache.lock**的临界区域之外检查引用计数是否为1，但此时已知链接计数为零，因此没有线程会尝试获取新引用。另一个主要危险是，对**ialloc**的并发调用可能会选择**iput**正在释放的同一个inode。这只能在**iupdate**写入磁盘以使inode的**type**为零后发生。这个争用是良性的：分配线程将客气地等待获取inode的睡眠锁，然后再读取或写入inode，此时**iput**已完成。

iput()可以写入磁盘。这意味着任何使用文件系统的系统调用都可能写入磁盘，因为系统调用可能是最后一个引用该文件的系统调用。即使像**read()**这样看起来是只读的调用，也可能最终调用**iput()**。这反过来意味着，即使是只读系统调用，如果它们使用文件系统，也必须在事务中进行包装。

iput()和崩溃之间存在一种具有挑战性的交互。 **iput()**不会在文件的链接计数降至零时立即截断文件，因为某些进程可能仍在内存中保留对inode的引用：进程可能

仍在读取和写入该文件，因为它已成功打开该文件。但是，如果在最后一个进程关闭该文件的文件描述符之前发生崩溃，则该文件将被标记为已在磁盘上分配，但没有目录项指向它。

文件系统以两种方式之一处理这种情况。简单的解决方案用于恢复时：重新启动后，文件系统会扫描整个文件系统，以查找标记为已分配但没有指向它们的目录项的文件。如果存在任何此类文件，接下来可以将其释放。

第二种解决方案不需要扫描文件系统。在此解决方案中，文件系统在磁盘（例如在超级块中）上记录链接计数降至零但引用计数不为零的文件的*i-number*。如果文件系统在其引用计数达到0时删除该文件，则会通过从列表中删除该*inode*来更新磁盘列表。恢复时，文件系统将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*可能被标记为已在磁盘上分配，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不足的风险。

8.10 代码： Inode包含内容

磁盘上的inode结构体`struct dinode`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`数组列出的块中找到。前面的`NDIRECT`个数据块被列在数组中的前`NDIRECT`个元素中；这些块称为直接块（direct blocks）。接下来的`NINDIRECT`个数据块不在inode中列出，而是在称为间接块（indirect block）的数据块中列出。`addrs`数组中的最后一个元素给出了间接块的地址。因此，可以从inode中列出的块加载文件的前12 kB (`NDIRECT x BSIZE`) 字节，而只有在查阅间接块后才能加载下一个256 kB (`NINDIRECT x BSIZE`) 字节。这是一个很好的磁盘表示，但对于客户端来说较复杂。函数`bmap`管理这种表示，以便实现我们将很快看到的如`readi`和`writei`这样的更高级例程。`bmap(struct inode *ip, uint bn)`返回索引结点`ip`的第`bn`个数据块的磁盘块号。如果`ip`还没有这样的块，`bmap`会分配一个。

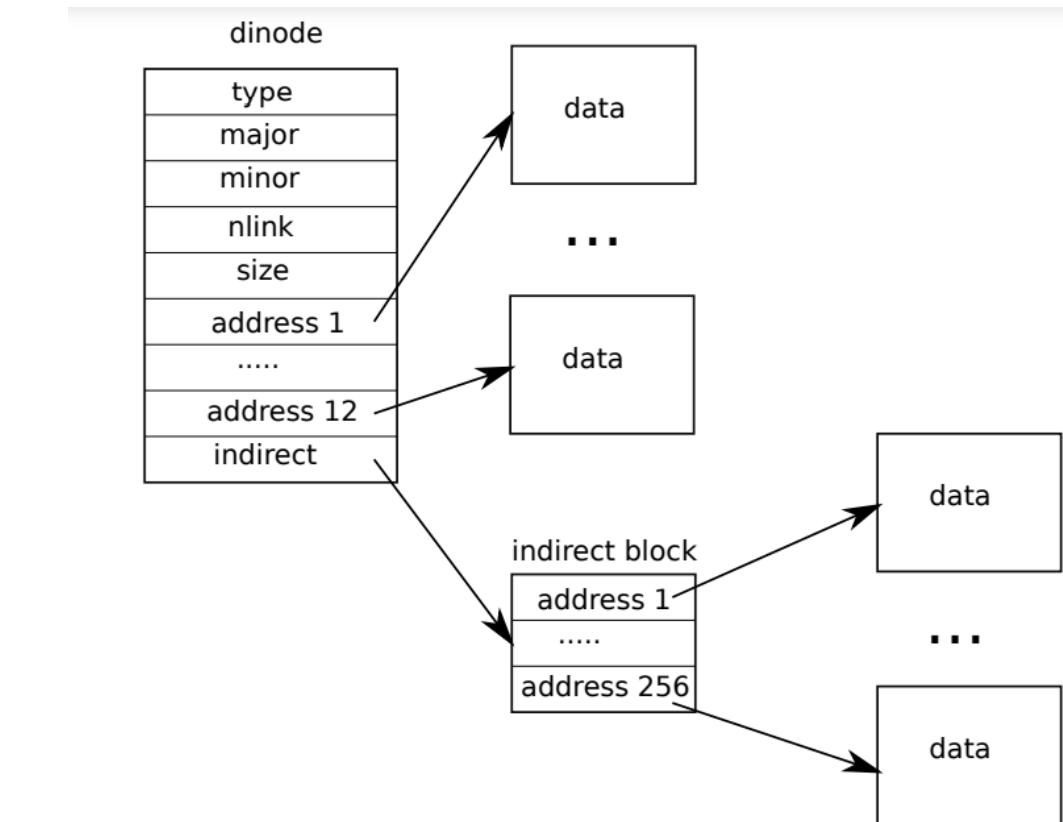


Figure 8.3: The representation of a file on disk.

函数`bmap`（`kernel/fs.c:378`）从简单的情况开始：前面的`NDIRECT`个块在`inode`本身中列出（`kernel/fs.c:383-387`）中。下面`NINDIRECT`个块在`ip->addrs[NDIRECT]`的间接块中列出。`Bmap`读取间接块（`kernel/fs.c:394`），然后从块内的正确位置（`kernel/fs.c:395`）读取块号。如果块号超过`NDIRECT+NINDIRECT`，则`bmap`调用`panic`崩溃；`writei`包含防止这种情况发生的检查（`kernel/fs.c:490`）。

`Bmap`根据需要分配块。`ip->addrs[]`或间接块中条目为零表示未分配块。当`bmap`遇到零时，它会用按需分配的新块（`kernel/fs.c:384-385`）（`kernel/fs.c:392-393`）替换它

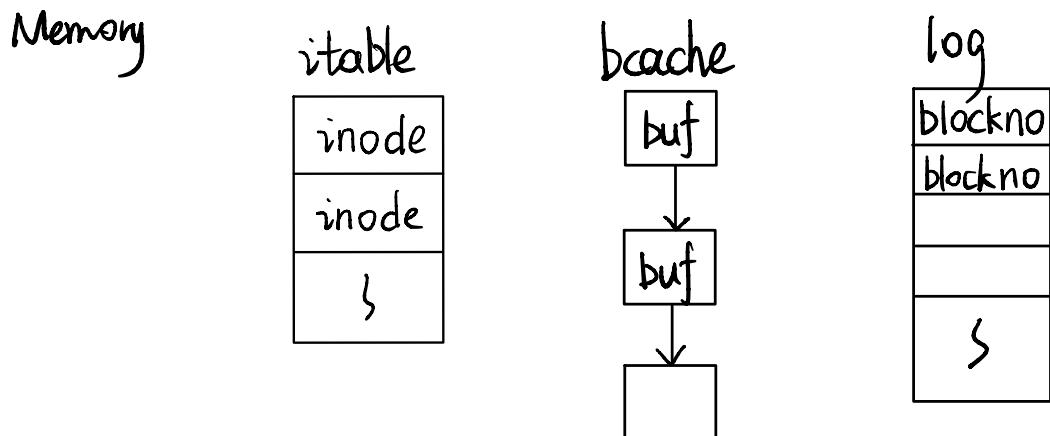
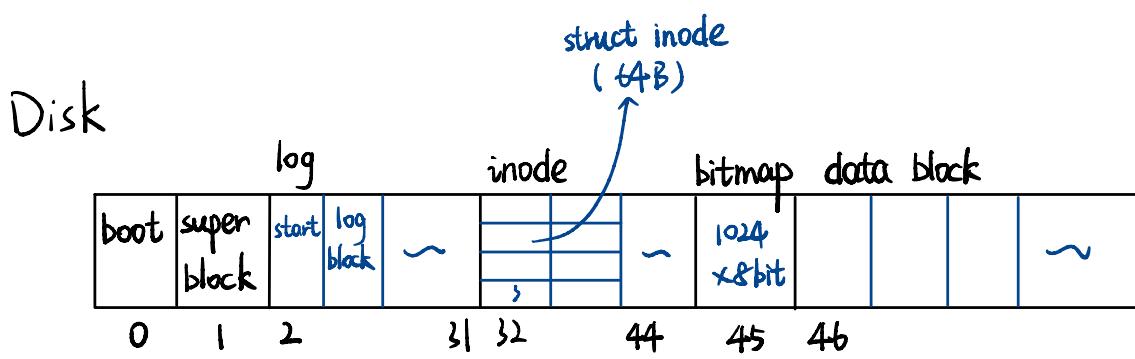
这是否是一种常见的设计方法，没有时就截取，
且截取是以独立函数存在的
们。

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

`Bmap`使`readi`和`writei`很容易获取inode的数据。`Readi` (`kernel/fs.c:456`) 首先确保偏移量和计数不超过文件的末尾。开始于超过文件末尾的地方读取将返回错误 (`kernel/fs.c:461-462`)，而从文件末尾开始或穿过文件末尾的读取返回的字节数少于请求的字节数 (`kernel/fs.c:463-464`)。主循环处理文件的每个块，将数据从缓冲区复制到`dst` (`kernel/fs.c:466-474`)。`writei` (`kernel/fs.c:483`) 与`readi`相同，但有三个例外：从文件末尾开始或穿过文件末尾的写操作会使文件增长到最大文件大小 (`kernel/fs.c:490-491`)；循环将数据复制到缓冲区而不是输出 (`kernel/fs.c:36`)；如果写入扩展了文件，`writei`必须更新其大小 (`kernel/fs.c:504-511`)。

`readi`和`writei`都是从检查`ip->type == T_DEV`开始的。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返回到这种情况。

函数`statt` (`kernel/fs.c:442`) 将inode元数据复制到`stat`结构体中，该结构通过`stat`系统调用向用户程序公开。



直接与Disk发生block交互的是`buf`
`itable`、`log`是信息的记录，用以挂带 block 的交互

8.11 代码：目录层

目录的内部实现很像文件。其inode的type为T_DIR，其数据是一系列目录条目(directory entries)。每个条目(entry)都是一个struct dirent(**kernel/fs.h:56**)，其中包含一个名称name和一个inode编号inum。名称最多为DIRSIZ(14)个字符；如果较短，则以NUL(0)字节终止。inode编号为零的条目是空的。

函数dirlookup(**kernel/fs.c:527**)在目录中搜索具有给定名称的条目。如果找到一个，它将返回一个指向相应inode的指针，解开锁定，并将*poff设置为目录中条目的字节偏移量，以满足调用方希望对其进行编辑的情形。如果dirlookup找到具有正确名称的条目，它将更新*poff并返回通过`iget`获得的未锁定的inode。`Dirlookup`是`iget`返回未锁定inode的原因。调用者已锁定dp，因此，如果对..，当前目录的别名，进行查找，则在返回之前尝试锁定inode将导致重新锁定dp并产生死锁(还有更复杂的死锁场景，涉及多个进程和..，父目录的别名。.不是唯一的问题。)调用者可以解锁dp，然后锁定ip，确保它一次只持有一个锁。

函数dirlink(**kernel/fs.c:554**)将给定名称和inode编号的新目录条目写入目录dp。如果名称已经存在，`dirlink`将返回一个错误(**kernel/fs.c:560-564**)。主循环读取目录条目，查找未分配的条目。当找到一个时，它会提前停止循环(**kernel/fs.c:538-539**)，并将off设置为可用条目的偏移量。否则，循环结束时会将off设置为dp->size。无论哪种方式，`dirlink`都会通过在偏移off处写入(**kernel/fs.c:574-577**)来向目录添加一个新条目。

默认目录不会写满？
似乎没有判断处理

8.12 代码：路径名 `open("/foo/bar", O_RDONLY)`

路径名查找涉及一系列对`dirlookup`的调用，每个路径组件调用一个。
`Namei` (`kernel/fs.c:661`) 计算`path`并返回相应的`inode`。函数`nameiparent`是一个变体：它在最后一个元素之前停止，返回父目录的`inode`并将最后一个元素复制到`name`中。两者都调用通用函数`namex`来完成实际工作。

`Namex` (`kernel/fs.c:626`) 首先决定路径计算的开始位置。如果路径以斜线开始，则计算从根目录开始；否则，从当前目录开始 (`kernel/fs.c:630-633`)。然后，它使用`skipelem`依次考察路径的每个元素 (`kernel/fs.c:635`)。循环的每次迭代都必须在当前索引结点`ip`中查找`name`。迭代首先给`ip`上锁并检查它是否是一个目录。如果不是，则查找失败 (`kernel/fs.c:636-640`) (锁定`ip`是必要的，不是因为`ip->type`可以被更改，而是在`ilock`运行之前，`ip->type`不能保证已从磁盘加载。) 如果调用是`nameiparent`，并且这是最后一个路径元素，则根据`nameiparent`的定义，循环会提前停止；最后一个路径元素已经复制到`name`中，因此`namex`只需返回解锁的`ip` (`kernel/fs.c:641-645`)。最后，循环将使用`dirlookup`查找路径元素，并通过设置`ip = next` (`kernel/fs.c:646-651`) 为下一次迭代做准备。当循环用完路径元素时，它返回`ip`。

`namex`过程可能需要很长时间才能完成：它可能涉及多个磁盘操作来读取路径名中所遍历目录的索引节点和目录块（如果它们不在`buffer cache`中）。xv6经过精心设计，如果一个内核线程对`namex`的调用在磁盘I/O上阻塞，另一个查找不同路径名的内核线程可以同时进行。Namex分别锁定路径中的每个目录，以便在不同目录中进行并行查找。

这种并发性带来了一些挑战。例如，当一个内核线程正在查找路径名时，另一个内核线程可能正在通过取消目录链接来更改目录树。（一个潜在的风险是，查找可能正在搜索已被另一个内核线程删除且其块已被重新用于另一个目录或文件的目录。）

xv6避免了这种竞争。例如，在`namex`中执行`dirlookup`时，`lookup`线程持有目录上的锁，`dirlookup`返回使用`iget`获得的`inode`。`Iget`增加索引节点的引用计数。只有在从`dirlookup`接收`inode`之后，`namex`才会释放目录上的锁。现在，另一个线程可以从目录中取消`inode`的链接，但是xv6还不会删除`inode`，因为`inode`的引用计数仍然大于零。

另一个风险是死锁。例如，查找“.”时，`next`指向与`ip`相同的`inode`。在释放`ip`上的锁之前锁定`next`将导致死锁。为了避免这种死锁，`namex`在获得下一个目录的锁之前解锁该目录。这里我们再次看到为什么`iget`和`ilock`之间的分离很重要。

8.13 文件描述符层

Unix界面的一个很酷的方面是，Unix中的大多数资源都表示为文件，包括控制台、管道等设备，当然还有真实文件。文件描述符层是实现这种一致性的层。

正如我们在第1章中看到的，`xv6`为每个进程提供了自己的打开文件表或文件描述符。每个打开的文件都由一个`struct file` (`kernel/file.h:1`) 表示，它是`inode`或管道的封装，加上一个I/O偏移量。每次调用`open`都会创建一个新的打开文件（一个新的`struct file`）：如果多个进程独立地打开同一个文件，那么不同的实例将具有不同的I/O偏移量。另一方面，单个打开的文件（同一个`struct file`）可以多次出现在一个进程的文件表中，也可以出现在多个进程的文件表中。如果一个进程使用`open`打开文件，然后使用`dup`创建别名，或使用`fork`与子进程共享，就会发生这种情况。引用计数跟踪对特定打开文件的引用数。可以打开文件进行读取或写入，也可以同时进行读取和写入。`readable`和`writable`字段可跟踪此操作。

系统中所有打开的文件都保存在全局文件表`ftable`中。文件表具有分配文件 (`filealloc`)、创建重复引用 (`filedup`)、释放引用 (`fclose`) 以及读取和写入数据 (`fileread`和`filewrite`) 的函数。

前三个函数遵循现在熟悉的形式。`Filealloc` (`kernel/file.c:30`) 扫描文件表以查找未引用的文件 (`f->ref == 0`)，并返回一个新的引用；`filedup` (`kernel/file.c:48`) 增加引用计数；`fclose` (`kernel/file.c:60`) 将其递减。当文件的引用计数达到零时，`fclose`会根据`type`释放底层管道或`inode`。

函数`filestat`、`fileread`和`filewrite`实现对文件的`stat`、`read`和`write`操作。`Filestat` (`kernel/file.c:88`) 只允许在`inode`上操作并且调用了`stati`。`Fileread`和`filewrite`检查打开模式是否允许该操作，然后将调用传递给管道或`inode`的实现。如果文件表示`inode`，`fileread`和`filewrite`使用I/O偏移量作为操作的偏移量，然后将文件指针前进该偏移量 (`kernel/file.c:122-123`) (`kernel/file.c:153-154`)。管道没有偏移的概念。回想一下，`inode`的函数要求调用方处理锁 (`kernel/file.c:94-96`) (`kernel/file.c:121-124`) (`kernel/file.c:163-166`)。`inode`锁定有一个方便的副作用，即读取和写入偏移量以原子方式更新，因此，对同一文件的同时多次写入不能覆盖彼此的数据，尽管他们的写入最终可能是交错的。

8.14 代码：系统调用

通过使用底层提供的函数，大多数系统调用的实现都很简单（请参阅 `kernel/sysfile.c`）。有几个调用值得仔细看看。

函数 `sys_link` 和 `sys_unlink` 编辑目录，创建或删除索引节点的引用。它们是使用事务能力的另一个很好的例子。`sys_link` (`kernel/sysfile.c:120`) 从获取其参数开始，两个字符串分别是 `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` 会增加其 `ip->nlink` 计数。然后 `sys_link` 调用 `nameiparent` 来查找 `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` (`kernel/sysfile.c:148`) 的新目录条目。`new` 的父目录必须存在并且与现有 `inode` 位于同一设备上：`inode` 编号在一个磁盘上只有唯一的含义。如果出现这样的错误，`sys_link` 必须返回并减少 `ip->nlink`。

(事务简化了实现，因为它需要更新多个磁盘块，但我们不必担心更新的顺序。他们要么全部成功，要么什么都不做。) 例如在没有事务的情况下，在创建一个链接之前更新 `ip->nlink` 会使文件系统暂时处于不安全状态，而在这两者之间发生的崩溃可能会造成严重破坏。对于事务，我们不必担心这一点

`Sys_link` 为现有 `inode` 创建一个新名称。函数 `create` (`kernel/sysfile.c:242`) 为新 `inode` 创建一个新名称。(它是三个文件创建系统调用的泛化：带有 `O_CREATE` 标志的 `open` 生成一个新的普通文件，`mkdir` 生成一个新目录，`mkdev` 生成一个新的设备文件。) 与 `sys_link` 一样，`create` 从调用 `nameiparent` 开始，以获取父目录的 `inode`。然后调用 `dirlookup` 检查名称是否已经存在 (`kernel/sysfile.c:252`)。如果名称确实存在，`create` 的行为取决于它用于哪个系统调用：`open` 的语义与 `mkdir` 和 `mkdev` 不同。如果 `create` 是代表 `open` (`type == T_FILE`) 使用的，并且存在的名称本身是一个常规文件，那么 `open` 会将其视为成功，`create` 也会这样做 (`kernel/sysfile.c:256`)。否则，这是一个错误 (`kernel/sysfile.c:257-258`)。如果名称不存在，`create` 现在将使用 `ialloc` (`kernel/sysfile.c:261`) 分配一个新的 `inode`。如果新 `inode` 是目录，`create` 将使用 `.` 和 `..` 条目对它进行初始化。最后，既然数据已正确初始化，`create` 可以将其链接到父目录 (`kernel/sysfile.c:274`)。`Create` 与 `sys_link` 一样，同时持有两个 `inode` 锁：`ip` 和 `dp`。不存在死锁的可能性，因为索引结点 `ip` 是新分配的：系统中没有其他进程会持有 `ip` 的锁，然后尝试锁定 `dp`。

使用 `create`，很容易实现 `sys_open`、`sys_mkdir` 和 `sys_mknod`。
`Sys_open` (`kernel/sysfile.c:287`) 是最复杂的，因为创建一个新文件只是它能做的一小部分。如果 `open` 被传递了 `O_CREATE` 标志，它将调用 `create` (`kernel/sysfile.c:301`)。否则，它将调用 `namei` (`kernel/sysfile.c:307`)。`Create` 返回一个锁定的 `inode`，但 `namei` 不锁定，因此 `sys_open` 必须锁定 `inode` 本身。这提供了一个方便的地方来检查目录是否仅为读取打开，而不是写入。假设 `inode` 是以某种方式获得的，`sys_open` 分配一个文件和一个文件描述符 (`kernel/sysfile.c:325`)，然后填充该文件 (`kernel/sysfile.c:337-342`)。请注意，没有其他进程可以访问部分初始化的文件，因为它仅位于当前进程的表中。

在我们还没有文件系统之前，第7章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该实现连接到文件系统。它的参数是一个指向两个整数的指针，它将在其中记录两个新的文件描述符。然后分配管道并安装文件描述符。

8.15 真实世界

实际操作系统中的buffer cache比xv6复杂得多，但它有两个相同的用途：缓存和同步对磁盘的访问。与UNIX V6一样，xv6的buffer cache使用简单的最近最少使用（LRU）替换策略；有许多更复杂的策略可以实现，每种策略都适用于某些工作场景，而不适用于其他工作场景。更高效的LRU缓存将消除链表，而改为使用哈希表进行查找，并使用堆进行LRU替换。现代buffer cache通常与虚拟内存系统集成，以支持内存映射文件。

xv6的日志系统效率低下。提交不能与文件系统调用同时发生。系统记录整个块，即使一个块中只有几个字节被更改。它执行同步日志写入，每次写入一个块，每个块可能需要整个磁盘旋转时间。真正的日志系统解决了所有这些问题。

日志记录不是提供崩溃恢复的唯一方法。（早期的文件系统在重新启动期间使用了一个清道夫程序（例如，UNIX的fsck程序）来检查每个文件和目录以及块和索引节点空闲列表，查找并解决不一致的问题。）清理大型文件系统可能需要数小时的时间，而且在某些情况下，无法以导致原始系统调用原子化的方式解决不一致问题。从日志中恢复要快得多，并且在崩溃时会导致系统调用原子化。

xv6使用的索引节点和目录的基础磁盘布局与早期UNIX相同；这一方案多年来经久不衰。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，它要求在每次查找期间对所有磁盘块进行线性扫描。当目录只有几个磁盘块时，这是合理的，但对于包含许多文件的目录来说，开销巨大。Microsoft Windows的NTFS、Mac OS X的HFS和Solaris的ZFS（仅举几例）将目录实现为磁盘上块的平衡树。这很复杂，但可以保证目录查找在对数时间内完成（即时间复杂度为O(logn)）。

xv6对于磁盘故障的解决很初级：如果磁盘操作失败，xv6就会调用panic。这是否合理取决于硬件：如果操作系统位于使用冗余屏蔽磁盘故障的特殊硬件之上，那么操作系统可能很少看到故障，因此panic是可以的。另一方面，使用普通磁盘的操作系统应该预料到会出现故障，并能更优雅地处理它们，这样一个文件中的块丢失不会影响文件系统其余部分的使用。

xv6要求文件系统安装在单个磁盘设备上，且大小不变。随着大型数据库和多媒体文件对存储的要求越来越高，操作系统正在开发各种方法来消除“每个文件系统一个磁盘”的瓶颈。基本方法是将多个物理磁盘组合成一个逻辑磁盘。RAID等硬件解决方案仍然是最流行的，但当前的趋势是在软件中尽可能多地实现这种逻辑。这些软件实现通常允许通过动态添加或删除磁盘来扩展或缩小逻辑设备等丰富功能。当然，一个能够动态增长或收缩的存储层需要一个能够做到这一点的文件系统：xv6使用的固定大小的inode块阵列在这样的环境中无法正常工作。（将磁盘管理与文件系统分离可能是最干净的设计，但两者之间复杂的接口导致了一些系统（如Sun的ZFS）将它们结合起来。）

xv6的文件系统缺少现代文件系统的许多其他功能；例如，它缺乏对快照和增量备份的支持。

现代Unix系统允许使用与磁盘存储相同的系统调用访问多种资源：命名管道、网络连接、远程访问的网络文件系统以及监视和控制接口，如/**proc**（注：Linux内核提供了一种通过/**proc**文件系统，在运行时访问内核内部数据结构、改变内核设置的机制。**proc**文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。）。不同于xv6中**fileread**和**filewrite**的**if**语句，这些系统通常为每个打开的文件提供一个函数指针表，每个操作一个，并通过函数指针来援引**inode**的调用实现。网络文件系统和用户级文件系统提供了将这些调用转换为网络RPC并在返回之前等待响应的函数。

8.16 练习

1. 为什么要在**balloc**中**panic**? xv6可以恢复吗?
2. 为什么要在**ialloc**中**panic**? xv6可以恢复吗?
3. 当文件用完时, **filealloc**为什么不**panic**? 为什么这更常见, 因此值得处理?
4. 假设在**sys_link**调用**iunlock(ip)**和**dirlink**之间, 与**ip**对应的文件被另一个进程解除链接。链接是否正确创建? 为什么?
5. **create**需要四个函数调用都成功 (一次调用**ialloc**, 三次调用**dirlink**)。如果未成功, **create**调用**panic**。为什么这是可以接受的? 为什么这四个调用都不能失败?
6. **sys_chdir**在**input(cp->cwd)**之前调用**iunlock(ip)**, 这可能会尝试锁定**cp->cwd**, 但将**iunlock(ip)**延迟到**input**之后不会导致死锁。为什么不这样做?
7. 实现**lseek**系统调用。支持**lseek**还需要修改**filewrite**, 以便在**lseek**设置**off**超过**f->ip->size**时, 用零填充文件中的空缺。
8. 将**O_TRUNC**和**O_APPEND**添加到**open**, 以便**>**和**>>**操作符在**shell**中工作。
9. 修改文件系统以支持符号链接。
10. 修改文件系统以支持命名管道。
11. 修改文件和VM系统以支持内存映射文件。

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。但更准确的看法是，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；它可能在不同的 CPU 上醒来，这意味着锁可能在不同的 CPU 上获取和释放。

agent 代理
在此可能是
func, proc,
cpu 的总称

object
lock

释放一个被锁保护的对象时，若该锁是嵌入在对象里的，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，为子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` ~~就是这样一个~~，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。