xv6: a simple, Unix-like teaching operating system

Russ Cox Frans Kaashoek Robert Morris

August 31, 2024

xv6: 一个简单、类Unix的教学操作系统

Russ Cox Frans Kaashoek Robert Morris

2024年8月31日

2

Contents

内容

6

Foreword and acknowledgments

This is a draft text intended for a class on operating systems. It explains the main concepts of operating systems by studying an example kernel, named xv6. Xv6 is modeled on Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6) [17]. Xv6 loosely follows the structure and style of v6, but is implemented in ANSI C [7] for a multi-core RISC-V [15].

This text should be read along with the source code for xv6, an approach inspired by John Lions' Commentary on UNIX 6th Edition [11]; the text has hyperlinks to the source code at `https://github.com/mit-pdos/xv6-riscv`. See `https://pdos.csail.mit.edu/6.1810` for additional pointers to on-line resources for v6 and xv6, including several lab assignments using xv6.

We have used this text in 6.828 and 6.1810, the operating system classes at MIT. We thank the faculty, teaching assistants, and students of those classes who have all directly or indirectly contributed to xv6. In particular, we would like to thank Adam Belay, Austin Clements, and Nickolai Zeldovich. Finally, we would like to thank people who emailed us bugs in the text or suggestions for improvements: Abutalib Aghayev, Sebastian Boehm, brandb97, Anton Burtsev, Raphael Carvalho, Tej Chajed,Brendan Davidson, Rasit Eskicioglu, Color Fuzzy, Wojciech Gac, Giuseppe, Tao Guo, Haibo Hao, Naoki Hayama, Chris Henderson, Robert Hilderman, Eden Hochbaum, Wolfgang Keller, Paweł Kraszewski, Henry Laih, Jin Li, Austin Liew, lyazj@github.com, Pavan Maddamsetti, Jacek Masiulaniec, Michael McConville, m3hm00d, miguelgvieira, Mark Morrissey, Muhammed Mourad, Harry Pan, Harry Porter, Siyuan Qian, Zhefeng Qiao, Askar Safin, Salman Shah, Huang Sha, Vikram Shenoy, Adeodato Simó, Ruslan Savchenko, Pawel Szczurko, Warren Toomey, tyfkda, tzerbib, Vanush Vaswani, Xi Wang, and Zou Chang Wei, Sam Whitlock, Qiongsi Wu, LucyShawYang, ykf1114@gmail.com, and Meng Zhou

If you spot errors or have suggestions for improvement, please send email to Frans Kaashoek and Robert Morris (kaashoek,rtm@csail.mit.edu).

前言和致谢

这是一份用于操作系统课程的草稿文本。它通过研究一个名为xv6的示例内核，解释了操作系统的核心概念。Xv6基于丹尼斯·里奇和肯·汤普森的Unix版本6 (v6) [17]建模。Xv6大致遵循v6的结构和风格，但使用ANSI C [7] 为多核RISC-V [15]实现。

本文本应与xv6的源代码一起阅读，这种做法受到约翰·李昂斯对UNIX第六版的评论 [11] 的启发；文本中提供了指向源代码的链接，位于https://github.com/mit-pdos/xv6-riscv。有关v6和xv6的在线资源，包括使用xv6的几个实验作业，请参阅https://pdos.csail.mit.edu/6.1810。

我们曾在MIT的6.828和6.1810操作系统课程中使用过本文本。我们感谢这些课程的所有教师、助教和学生，他们都直接或间接地为xv6做出了贡献。特别是，我们想感谢亚当·贝莱、奥斯汀·克莱门茨和尼科莱·泽尔多维奇。最后，我们想感谢那些通过电子邮件向我们报告文本中的错误或提出建议的人

用于改进：阿布塔利布·阿加耶夫，塞巴斯蒂安·博姆，brandb97，安东·布特谢夫，拉斐尔·卡瓦尔霍，泰·查杰德，布兰登·戴维森，拉斯IT·埃斯基奥格鲁，Color Fuzzy，沃伊切赫·加克，朱塞佩，郭涛，郝海波，林浩，克里斯·亨德森，罗伯特·希尔德曼，伊甸·霍克鲍姆，沃尔夫冈·凯勒，帕维尔·克拉谢夫斯基，亨利·莱，李金，奥斯汀·刘，lyazj@github.com，帕万·马达姆塞蒂，雅采克·马西乌拉涅茨，MichaelMcConville，m3hm00d，miguelgvieira，马克·莫里斯西，穆罕默德·穆拉德，哈里·潘，哈里·波特，钱思远，乔哲峰，阿斯卡·萨芬，萨拉曼·沙，黄沙，维克拉姆·谢诺伊，阿德奥达托·西莫，鲁斯兰·萨夫琴科，帕维尔·斯奇尔科，沃伦·图梅，tyfkda，tzerbib，瓦努什·瓦斯瓦尼，王希，邹长伟，山姆·惠特洛克，吴琼思，LucyShawYang，ykf1114@gmail.com，以及周梦

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

8

Chapter 1

Operating system interfaces

The job of an operating system is to share a computer among multiple programs and to provide a more useful set of services than the hardware alone supports. An operating system manages and abstracts the low-level hardware, so that, for example, a word processor need not concern itself with which type of disk hardware is being used. An operating system shares the hardware among multiple programs so that they run (or appear to run) at the same time. Finally, operating systems provide controlled ways for programs to interact, so that they can share data or work together.

An operating system provides services to user programs through an interface. Designing a good interface turns out to be difficult. On the one hand, we would like the interface to be simple and narrow because that makes it easier to get the implementation right. On the other hand, we may be tempted to offer many sophisticated features to applications. The trick in resolving this tension is to design interfaces that rely on a few mechanisms that can be combined to provide much generality.

This book uses a single operating system as a concrete example to illustrate operating system concepts. That operating system, xv6, provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie's Unix operating system [17], as well as mimicking Unix's internal design. Unix provides a narrow interface whose mechanisms combine well, offering a surprising degree of generality. This interface has been so successful that modern operating systems—BSD, Linux, macOS, Solaris, and even, to a lesser extent, Microsoft Windows—have Unix-like interfaces. Understanding xv6 is a good start toward understanding any of these systems and many others.

As Figure 1.1 shows, xv6 takes the traditional form of a *kernel*, a special program that provides services to running programs. Each running program, called a *process*, has memory containing instructions, data, and a stack. The instructions implement the program's computation. The data are the variables on which the computation acts. The stack organizes the program's procedure calls. A given computer typically has many processes but only a single kernel.

When a process needs to invoke a kernel service, it invokes a *system call*, one of the calls in the operating system's interface. The system call enters the kernel; the kernel performs the service and returns. Thus a process alternates between executing in *user space* and *kernel space*.

As described in detail in subsequent chapters, the kernel uses the hardware protection mechanisms provided by a CPU[1] to ensure that each process executing in user space can access only

[1]This text generally refers to the hardware element that executes a computation with the term *CPU*, an acronym

第一章

操作系统接口

操作系统的任务是让多个程序共享一台计算机，并提供硬件本身所无法支持的更完善的服务集。操作系统管理和抽象低级硬件，因此例如文字处理器不必关心正在使用哪种类型的磁盘硬件。操作系统在多个程序之间共享硬件，以便它们可以同时运行（或看起来像是在同时运行）。最后，操作系统为程序提供受控的交互方式，以便它们可以共享数据或协同工作。

操作系统通过接口为用户程序提供服务。设计一个良好的接口被证明是困难的。一方面，我们希望接口简单且狭窄，因为那样更容易正确实现。另一方面，我们可能会被诱惑向应用程序提供许多复杂的功能。解决这种紧张关系的诀窍是设计依赖于少数可以组合起来提供广泛通用性的机制的接口。

本书以单个操作系统作为具体实例，说明操作系统概念。该操作系统，xv6，提供了肯·汤普森和丹尼斯·里奇所提出的Unix操作系统 [17],的基本接口，以及模拟Unix的内部设计。Unix提供了一种狭窄的接口，其机制结合良好，具有惊人的通用性。这种接口非常成功，以至于现代操作系统——BSD、Linux、macOS、Solaris，甚至在一定程度上Microsoft Windows——都具有类Unix的接口。理解xv6是理解这些系统以及其他许多系统的良好起点。

如图1.1所示，xv6采用内核的传统形式，内核是一个特殊程序，为运行中的程序提供服务。每个运行中的程序，称为进程，具有包含指令、数据和栈的内存。指令实现程序的运算。数据是运算作用的变量。栈组织程序的程序调用。一台给定的计算机通常有多个进程，但只有一个内核。

当进程需要调用内核服务时，它会发起一个系统调用，这是操作系统接口中的一种调用。系统调用进入内核；内核执行该服务并返回。因此，进程在用户空间和内核空间之间交替执行。

如后续章节所述，内核使用CPU[1] 提供的硬件保护机制，以确保在用户空间中执行的每个进程只能访问

[1]这段文字通常指的是执行计算的硬件元素，该元素用CPU这个缩写词表示。

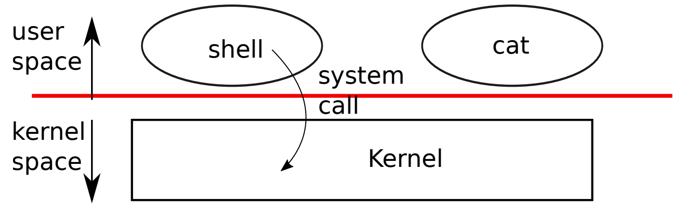

Figure 1.1: A kernel and two user processes.

its own memory. The kernel executes with the hardware privileges required to implement these protections; user programs execute without those privileges. When a user program invokes a system call, the hardware raises the privilege level and starts executing a pre-arranged function in the kernel.

The collection of system calls that a kernel provides is the interface that user programs see. The xv6 kernel provides a subset of the services and system calls that Unix kernels traditionally offer. Figure 1.2 lists all of xv6's system calls.

The rest of this chapter outlines xv6's services—processes, memory, file descriptors, pipes, and a file system—and illustrates them with code snippets and discussions of how the *shell*, Unix's command-line user interface, uses them. The shell's use of system calls illustrates how carefully they have been designed.

The shell is an ordinary program that reads commands from the user and executes them. The fact that the shell is a user program, and not part of the kernel, illustrates the power of the system call interface: there is nothing special about the shell. It also means that the shell is easy to replace; as a result, modern Unix systems have a variety of shells to choose from, each with its own user interface and scripting features. The xv6 shell is a simple implementation of the essence of the Unix Bourne shell. Its implementation can be found at (user/sh.c:1).

1.1 Processes and memory

An xv6 process consists of user-space memory (instructions, data, and stack) and per-process state private to the kernel. Xv6 *time-shares* processes: it transparently switches the available CPUs among the set of processes waiting to execute. When a process is not executing, xv6 saves the process's CPU registers, restoring them when it next runs the process. The kernel associates a process identifier, or `PID`, with each process.

A process may create a new process using the `fork` system call. `fork` gives the new process an exact copy of the calling process's memory: it copies the instructions, data, and stack of the calling process into the new process's memory. `fork` returns in both the original and new processes. In the original process, `fork` returns the new process's PID. In the new process, `fork` returns zero. The original and new processes are often called the *parent* and *child*.

for central processing unit. Other documentation (e.g., the RISC-V specification) also uses the words processor, core, and hart instead of CPU.

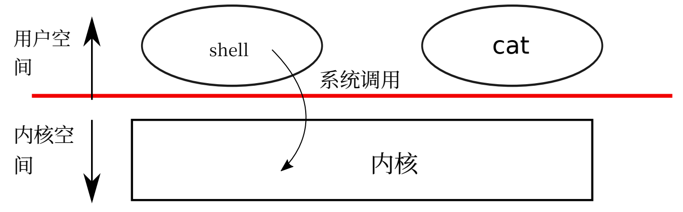

图1.1：一个内核和两个用户进程。

其自己的内存。内核以实现这些保护所需的硬件特权执行；用户程序则没有这些特权。当用户程序调用系统调用时，硬件提升特权级别，并开始执行内核中预先安排的函数。

内核提供的系统调用集合是用户程序看到的接口。xv6 内核提供了一部分 Unix 内核传统上提供的服务和系统调用。图 1.2 列出了 xv6 的所有系统调用。

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

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

1.1 进程和内存

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

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

中央处理单元的缩写。其他文档（例如，RISC-V规范）也使用处理器、核心和hart代替CPU这些词语。

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n zero bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

Figure 1.2: Xv6 system calls. If not otherwise stated, these calls return 0 for no error, and -1 if there's an error.

For example, consider the following program fragment written in the C programming language [7]:

```
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");
}
```

The `exit` system call causes the calling process to stop executing and to release resources such as memory and open files. Exit takes an integer status argument, conventionally 0 to indicate success and 1 to indicate failure. The `wait` system call returns the PID of an exited (or killed) child of the current process and copies the exit status of the child to the address passed to wait; if none of

系统调用	描述
int fork()	创建进程，返回子进程的PID。
int exit(int status)	终止当前进程；状态报告给wait()；无返回值。
int wait(int *status)	等待子进程退出；退出状态在 *status 中返回；返回子进程 PID。
int kill(int pid)	终止进程 PID。返回 0，或 -1 表示错误。
int getpid()	返回当前进程的 PID。
int 睡眠(int n)	暂停 n 时钟滴答。
int exec(char *文件, char *argv[])	加载文件并使用参数执行；仅在出错时返回。
char *sbrk(int n)	通过 n 零字节增长进程的内存。返回新内存的起始地址。
int open(char *file, int flags)	打开文件；标志指示读/写；返回文件描述符 fd。
int 写入(int 文件描述符, char *buf, int n)	将 buf 中的 n 个字节写入文件描述符 fd；返回 n。
int 读取(int 文件描述符, char *buf, int n)	将 n 个字节读入 buf；返回读取的字节数；或文件末尾时返回 0。
int 关闭(int 文件描述符)	释放 打开 文件 文件描述符。
int dup(int 文件描述符)	返回一个新的文件描述符，它引用与 文件描述符 相同的文件。
int 管道(int p[])	创建一个管道，将读/写文件描述符放入 p[0] 和 p[1]。
int chdir(char *dir)	改变当前目录。
int mkdir(char *dir)	创建一个新目录。
int 创建设备文件(char *file, int, int)	创建一个设备文件。
int fstat(int fd, struct stat *st)	将打开文件的信息放入 *st。
int 链接(char *文件1, char *文件2)	为文件 file1 创建另一个名称（file2）。
int unlink(char *file)	删除一个文件。

图1.2：Xv6系统调用。如果未特别说明，这些调用在没有错误时返回0，如果有错误则返回-1 有错误 .

例如，考虑以下用 C 编程语言编写的程序片段 [7]:

```
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");
}
```

退出系统调用导致调用进程停止执行并释放资源，例如内存和打开的文件。退出需要一个整数状态参数，通常为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，并将子进程的退出状态复制到wait传递的地址；如果调用者的所有子进程都没有退出，wait会等待一个子进程退出。如果调用者没有子进程，wait会立即返回-1。如果父进程不关心子进程的退出状态，它可以向wait传递一个0地址。

the caller's children has exited, `wait` waits for one to do so. If the caller has no children, `wait` immediately returns -1. If the parent doesn't care about the exit status of a child, it can pass a 0 address to `wait`.

In the example, the output lines

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

might come out in either order (or even intermixed), depending on whether the parent or child gets to its `printf` call first. After the child exits, the parent's `wait` returns, causing the parent to print

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

Although the child has the same memory contents as the parent initially, the parent and child are executing with separate memory and separate registers: changing a variable in one does not affect the other. For example, when the return value of `wait` is stored into `pid` in the parent process, it doesn't change the variable `pid` in the child. The value of `pid` in the child will still be zero.

The `exec` system call replaces the calling process's memory with a new memory image loaded from a file stored in the file system. The file must have a particular format, which specifies which part of the file holds instructions, which part is data, at which instruction to start, etc. Xv6 uses the ELF format, which Chapter 3 discusses in more detail. Usually the file is the result of compiling a program's source code. When `exec` succeeds, it does not return to the calling program; instead, the instructions loaded from the file start executing at the entry point declared in the ELF header. `exec` takes two arguments: the name of the file containing the executable and an array of string arguments. For example:

```
char *argv[3];

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

This fragment replaces the calling program with an instance of the program `/bin/echo` running with the argument list `echo hello`. Most programs ignore the first element of the argument array, which is conventionally the name of the program.

The xv6 shell uses the above calls to run programs on behalf of users. The main structure of the shell is simple; see `main` (user/sh.c:146). The main loop reads a line of input from the user with `getcmd`. Then it calls `fork`, which creates a copy of the shell process. The parent calls `wait`, while the child runs the command. For example, if the user had typed "`echo hello`" to the shell, `runcmd` would have been called with "`echo hello`" as the argument. `runcmd` (user/sh.c:55) runs the actual command. For "`echo hello`", it would call `exec` (user/sh.c:79). If `exec` succeeds then the child will execute instructions from `echo` instead of `runcmd`. At some point `echo` will call `exit`, which will cause the parent to return from `wait` in `main` (user/sh.c:146).

You might wonder why `fork` and `exec` are not combined in a single call; we will see later that the shell exploits the separation in its implementation of I/O redirection. To avoid the wastefulness

调用者的子进程没有退出，wait会等待一个子进程退出。如果调用者没有子进程，wait会立即返回-1。如果父进程不关心子进程的退出状态，它可以向wait传递一个0地址。

在示例中，输出行

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

可能会以任意顺序（甚至可能交错）出现，这取决于父进程或子进程先到达其 printf 调用。子进程退出后，父进程的 wait 返回，导致父进程打印

父进程：子进程 1234 已完成

尽管子进程初始时与父进程具有相同的内存内容，但父进程和子进程使用的是独立的内存和独立的寄存器：在一个进程中修改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中的 pid 变量时，它不会改变子进程中的变量 pid。子进程中的 pid 值仍然为零。

exec 系统调用用从文件系统加载的新内存镜像替换调用进程的内存。该文件必须具有特定的格式，该格式指定文件中的哪部分包含指令、哪部分是数据、从哪条指令开始执行等。Xv6 使用 ELF 格式，第 3 章将更详细地讨论。通常，该文件是编译程序源代码的结果。当 exec 成功时，它不会返回到调用程序；相反，从文件加载的指令会在 ELF 头中声明的入口点开始执行。exec 接受两个参数：包含可执行文件的文件名和一个字符串参数数组。例如：

```
char *argv[3];

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

这个片段用程序 /bin/echo 的一个实例替换了调用程序，该实例使用参数列表 echohello 来运行。大多数程序会忽略参数数组中的第一个元素，该元素通常是程序的名称。

xv6 shell 使用上述调用代表用户运行程序。shell 的主要结构很简单；参见 main (user/sh.c:146)。主循环使用 getcmd 从用户读取一行输入。然后它调用 fork，创建 shell 进程的副本。父进程调用 wait，而子进程运行命令。例如，如果用户向 shell 输入了"echo hello"，runcmd 将以"echo hello"作为参数被调用。runcmd (user/sh.c:55) 运行实际命令。对于"echo hello"，它将调用 exec (user/sh.c:79)。如果 exec 成功，子进程将执行 echo 的指令而不是 runcmd。在某个时刻 echo 将调用 exit，这将导致父进程在 main (user/sh.c:146) 中从 wait 返回。

你可能想知道为什么 fork 和 exec 不是一个调用；我们稍后会看到 shell 利用其在 I/O 重定向实现中的分离。为了避免浪费

of creating a duplicate process and then immediately replacing it (with `exec`), operating kernels optimize the implementation of `fork` for this use case by using virtual memory techniques such as copy-on-write (see Section 4.6).

Xv6 allocates most user-space memory implicitly: `fork` allocates the memory required for the child's copy of the parent's memory, and `exec` allocates enough memory to hold the executable file. A process that needs more memory at run-time (perhaps for `malloc`) can call `sbrk(n)` to grow its data memory by `n` zero bytes; `sbrk` returns the location of the new memory.

1.2 I/O and File descriptors

A *file descriptor* is a small integer representing a kernel-managed object that a process may read from or write to. A process may obtain a file descriptor by opening a file, directory, or device, or by creating a pipe, or by duplicating an existing descriptor. For simplicity we'll often refer to the object a file descriptor refers to as a "file"; the file descriptor interface abstracts away the differences between files, pipes, and devices, making them all look like streams of bytes. We'll refer to input and output as *I/O*.

Internally, the xv6 kernel uses the file descriptor as an index into a per-process table, so that every process has a private space of file descriptors starting at zero. By convention, a process reads from file descriptor 0 (standard input), writes output to file descriptor 1 (standard output), and writes error messages to file descriptor 2 (standard error). As we will see, the shell exploits the convention to implement I/O redirection and pipelines. The shell ensures that it always has three file descriptors open (user/sh.c:152), which are by default file descriptors for the console.

The `read` and `write` system calls read bytes from and write bytes to open files named by file descriptors. The call `read(fd, buf, n)` reads at most `n` bytes from the file descriptor `fd`, copies them into `buf`, and returns the number of bytes read. Each file descriptor that refers to a file has an offset associated with it. `read` reads data from the current file offset and then advances that offset by the number of bytes read: a subsequent `read` will return the bytes following the ones returned by the first `read`. When there are no more bytes to read, `read` returns zero to indicate the end of the file.

The call `write(fd, buf, n)` writes `n` bytes from `buf` to the file descriptor `fd` and returns the number of bytes written. Fewer than `n` bytes are written only when an error occurs. Like `read`, `write` writes data at the current file offset and then advances that offset by the number of bytes written: each `write` picks up where the previous one left off.

The following program fragment (which forms the essence of the program `cat`) copies data from its standard input to its standard output. If an error occurs, it writes a message to the standard error.

```
char buf[512];
int n;

for(;;){
  n = read(0, buf, sizeof buf);
  if(n == 0)
```

创建一个重复的进程然后立即替换它（使用 exec），操作内核通过使用写时复制等虚拟内存技术优化 fork 的实现以适应这种情况（参见第 4.6 节）。

Xv6 隐式分配大多数用户空间内存：fork 分配子进程所需的父进程内存副本所需的内存，而 exec 分配足够的内存来容纳可执行文件。需要在运行时需要更多内存的进程（可能用于 malloc）可以调用 sbrk(n) 来通过 n 个零字节增长其数据内存；sbrk 返回新内存的位置。

1.2 I/O 和文件描述符

文件描述符是一个表示内核管理的对象的小整数，进程可以从中读取或写入。进程可以通过打开文件、目录或设备，或创建管道，或复制现有描述符来获得文件描述符。为了简单起见，我们将经常将文件描述符引用的对象称为“文件”；文件描述符接口抽象了文件、管道和设备之间的差异，使它们看起来都像字节流。我们将输入和输出称为 I/O。

内部，xv6 内核使用文件描述符作为每个进程表中的索引，以便每个进程都有一个从零开始的私有文件描述符空间。按惯例，进程从文件描述符 0（标准输入）读取，将输出写入文件描述符 1（标准输出），并将错误消息写入文件描述符 2（标准错误）。正如我们将看到的，shell 利用惯例来实现 I/O 重定向和管道。shell 确保它始终有三个文件描述符打开（user/sh.c:152），这些默认是控制台的文件描述符。

读取和写入系统调用从以文件描述符命名的打开文件中读取字节，并将字节写入这些文件。调用 read(fd, buf, n) 从文件描述符 fd 中读取最多 n 个字节，将它们复制到 buf 中，并返回读取的字节数。每个引用文件的文件描述符都与一个偏移量相关联。读取从当前文件偏移量读取数据，然后通过读取的字节数递增该偏移量：后续的读取将返回第一次读取后返回的字节。当没有更多字节可读时，读取返回零以指示文件末尾。

调用 write(fd, buf, n) 将 n 个字节从 buf 写入文件描述符 fd 并返回写入的字节数。只有当发生错误时，才会写入少于 n 个字节。与读取类似，写入在当前文件偏移量写入数据，然后通过写入的字节数递增该偏移量：每次写入都从上一次写入结束的地方开始。

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

```
char buf[512];
int n;

for(;;){
  n = read(0, buf, sizeof buf);
  if(n == 0)
```

```
      break;
    if(n < 0){
      fprintf(2, "read error\n");
      exit(1);
    }
    if(write(1, buf, n) != n){
      fprintf(2, "write error\n");
      exit(1);
    }
  }
```

The important thing to note in the code fragment is that `cat` doesn't know whether it is reading from a file, console, or a pipe. Similarly `cat` doesn't know whether it is printing to a console, a file, or whatever. The use of file descriptors and the convention that file descriptor 0 is input and file descriptor 1 is output allows a simple implementation of `cat`.

The `close` system call releases a file descriptor, making it free for reuse by a future `open`, `pipe`, or `dup` system call (see below). A newly allocated file descriptor is always the lowest-numbered unused descriptor of the current process.

File descriptors and `fork` interact to make I/O redirection easy to implement. `fork` copies the parent's file descriptor table along with its memory, so that the child starts with exactly the same open files as the parent. The system call `exec` replaces the calling process's memory but preserves its file table. This behavior allows the shell to implement *I/O redirection* by forking, re-opening chosen file descriptors in the child, and then calling `exec` to run the new program. Here is a simplified version of the code a shell runs for the command `cat < input.txt`:

```
  char *argv[2];

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

After the child closes file descriptor 0, `open` is guaranteed to use that file descriptor for the newly opened `input.txt`: 0 will be the smallest available file descriptor. `cat` then executes with file descriptor 0 (standard input) referring to `input.txt`. The parent process's file descriptors are not changed by this sequence, since it modifies only the child's descriptors.

The code for I/O redirection in the xv6 shell works in exactly this way (user/sh.c:83). Recall that at this point in the code the shell has already forked the child shell and that `runcmd` will call `exec` to load the new program.

The second argument to `open` consists of a set of flags, expressed as bits, that control what `open` does. The possible values are defined in the file control (fcntl) header (kernel/fcntl.h:1-5): `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREATE`, and `O_TRUNC`, which instruct `open` to open the file

```
      break;
    if(n < 0){
      fprintf(2, "read error\n");
      exit(1);
    }
    if(write(1, buf, n) != n){
      fprintf(2, "write error\n");
      exit(1);
    }
  }
```

在代码片段中需要注意的重要一点是，cat 不知道它是否正在从文件、控制台或管道中读取。同样，cat 也不知道它是否正在向控制台、文件或任何其他地方打印。文件描述符的使用以及文件描述符 0 是输入、文件描述符 1 是输出的约定，使得 cat 的实现变得简单。

关闭系统调用会释放一个文件描述符，使其可供未来的打开、管道或 dup 系统调用（见下文）重用。新分配的文件描述符始终是当前进程的最低编号的未使用描述符。

文件描述符和 fork 协同工作，使 I/O 重定向易于实现。fork 会复制父进程的文件描述符表及其内存，因此子进程会以与父进程完全相同的打开文件开始。系统调用 exec 会替换调用进程的内存，但保留其文件表。这种行为允许 shell 通过 fork、在子进程中重新打开选定的文件描述符，然后调用 exec 来运行新程序来实现 I/O 重定向。以下是 shell 为 cat ＜input.txt 命令运行的代码的简化版本：

```
  char *argv[2];

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

子进程关闭文件描述符 0 后，open 保证会使用该文件描述符打开新的 input.txt：0 将是可用的最小文件描述符。cat 然后以文件描述符 0（标准输入）指向 input.txt 的方式执行。父进程的文件描述符不会因这一序列而改变，因为它只修改了子进程的描述符。

xv6 shell 中的 I/O 重定向代码工作方式完全如此（user/sh.c:83）。回想一下，在代码的这一点上，shell 已经 fork 了子 shell，并且 runcmd 将调用 exec 来加载新程序。

open 的第二个参数是一组标志，以位的形式表示，用于控制 open 的行为。可能的值在文件控制 (fcntl) 头文件中定义（kernel/fcntl.h:1-5）：O_RDONLY, O_WRONLY, O_RDWR, O_CREATE, 以及 O_TRUNC，这些值指示 open 打开文件

for reading, or for writing, or for both reading and writing, to create the file if it doesn't exist, and to truncate the file to zero length.

Now it should be clear why it is helpful that `fork` and `exec` are separate calls: between the two, the shell has a chance to redirect the child's I/O without disturbing the I/O setup of the main shell. One could instead imagine a hypothetical combined `forkexec` system call, but the options for doing I/O redirection with such a call seem awkward. The shell could modify its own I/O setup before calling `forkexec` (and then un-do those modifications); or `forkexec` could take instructions for I/O redirection as arguments; or (least attractively) every program like `cat` could be taught to do its own I/O redirection.

Although `fork` copies the file descriptor table, each underlying file offset is shared between parent and child. Consider this example:

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

At the end of this fragment, the file attached to file descriptor 1 will contain the data `hello world`. The `write` in the parent (which, thanks to `wait`, runs only after the child is done) picks up where the child's `write` left off. This behavior helps produce sequential output from sequences of shell commands, like `(echo hello; echo world) >output.txt`.

The `dup` system call duplicates an existing file descriptor, returning a new one that refers to the same underlying I/O object. Both file descriptors share an offset, just as the file descriptors duplicated by `fork` do. This is another way to write `hello world` into a file:

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

Two file descriptors share an offset if they were derived from the same original file descriptor by a sequence of `fork` and `dup` calls. Otherwise file descriptors do not share offsets, even if they resulted from `open` calls for the same file. `dup` allows shells to implement commands like this: `ls existing-file non-existing-file > tmp1 2>&1`. The `2>&1` tells the shell to give the command a file descriptor 2 that is a duplicate of descriptor 1. Both the name of the existing file and the error message for the non-existing file will show up in the file `tmp1`. The xv6 shell doesn't support I/O redirection for the error file descriptor, but now you know how to implement it.

File descriptors are a powerful abstraction, because they hide the details of what they are connected to: a process writing to file descriptor 1 may be writing to a file, to a device like the console, or to a pipe.

用于读取，或用于写入，或用于读写，如果文件不存在则创建该文件，并将文件截断为零长度。

现在应该很清楚为什么fork和exec是分开的调用是有帮助的：在这两者之间，shell有机会在不干扰主shell的I/O设置的情况下重定向子进程的I/O。人们可以想象一个假设的forkexec系统调用，但使用这种调用进行I/O重定向的选项似乎很笨拙。shell可以在调用forkexec之前修改自己的I/O设置（然后撤销这些修改）；或者forkexec可以作为参数接收I/O重定向指令；或者（最不吸引人）每个像cat这样的程序都可以被教自己做自己的I/O重定向。

尽管fork复制了文件描述符表，但每个底层文件偏移量都在父进程和子进程之间共享。考虑这个例子：

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

在这个片段的末尾，附加到文件描述符1的文件将包含数据hello world。父进程中的写入（多亏了wait，它只在子进程完成后运行）接续了子进程写入停止的地方。这种行为有助于从一系列shell命令中生成顺序输出，例如（echo hello; echo world） >output.txt。

dup系统调用复制一个现有的文件描述符，返回一个新的文件描述符，该文件描述符引用相同的底层I/O对象。这两个文件描述符共享一个偏移量，就像fork复制的文件描述符一样。这是另一种将hello world写入文件的方法：

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

两个文件描述符共享一个偏移量，如果它们是由一系列 fork 和 dup 调用从同一个原始文件描述符派生出来的。否则，文件描述符不共享偏移量，即使它们是针对同一个文件进行的 open 调用的结果。dup 允许 shell 实现类似这样的命令：ls 现有文件 不存在的文件 > t mp1 2>&1。 2>&1 告诉 shell 将命令的文件描述符2设置为文件描述符1的副本。现有文件的名字和不存在文件的错误消息都会出现在文件 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。

文件描述符是一种强大的抽象，因为它们隐藏了它们所连接的细节：一个写入文件描述符1的进程可能会写入一个文件、一个像控制台这样的设备，或是一个管道。

1.3 Pipes

A *pipe* is a small kernel buffer exposed to processes as a pair of file descriptors, one for reading and one for writing. Writing data to one end of the pipe makes that data available for reading from the other end of the pipe. Pipes provide a way for processes to communicate.

The following example code runs the program `wc` with standard input connected to the read end of a pipe.

```
int p[2];
char *argv[2];

argv[0] = "wc";
argv[1] = 0;

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

The program calls `pipe`, which creates a new pipe and records the read and write file descriptors in the array `p`. After `fork`, both parent and child have file descriptors referring to the pipe. The child calls `close` and `dup` to make file descriptor zero refer to the read end of the pipe, closes the file descriptors in `p`, and calls `exec` to run `wc`. When `wc` reads from its standard input, it reads from the pipe. The parent closes the read side of the pipe, writes to the pipe, and then closes the write side.

If no data is available, a `read` on a pipe waits for either data to be written or for all file descriptors referring to the write end to be closed; in the latter case, `read` will return 0, just as if the end of a data file had been reached. The fact that `read` blocks until it is impossible for new data to arrive is one reason that it's important for the child to close the write end of the pipe before executing `wc` above: if one of `wc` 's file descriptors referred to the write end of the pipe, `wc` would never see end-of-file.

The xv6 shell implements pipelines such as `grep fork sh.c | wc -l` in a manner similar to the above code (user/sh.c:101). The child process creates a pipe to connect the left end of the pipeline with the right end. Then it calls `fork` and `runcmd` for the left end of the pipeline and `fork` and `runcmd` for the right end, and waits for both to finish. The right end of the pipeline may be a command that itself includes a pipe (e.g., `a | b | c`), which itself forks two new child processes (one for `b` and one for `c`). Thus, the shell may create a tree of processes. The leaves

1.3 管道

管道是一个暴露给进程的小内核缓冲区，以一对文件描述符的形式出现，一个用于读取，一个用于写入。向管道的一端写入数据会使该数据在管道的另一端可用。管道为进程提供了一种通信方式。

以下示例代码运行 wc 程序，其标准输入连接到管道的读取端。

```
int p[2];
char *argv[2];

argv[0] = "wc";
argv[1] = 0;

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

程序调用 pipe，创建一个新的管道，并将读取和写入文件描述符记录在数组 p 中。f
ork 后，父进程和子进程都有指向管道的文件描述符。子进程调用 close 和 dup，使文件描述符零指向管道的读取端，关闭 inp 的文件描述符，并调用 exec 运行 wc。当 wc 从其标准输入读取时，它从管道读取。父进程关闭管道的读取端，写入管道，然后关闭写入端。

如果没有数据可用，对管道的读取会等待写入数据或所有指向写入端的文件描述符被关闭；在后一种情况下，读取将返回 0，就像数据文件结束一样。读取会阻塞直到不可能再收到新数据，这是子进程在执行上述 wc 之前关闭管道写入端的重要原因：如果 wc 的某个文件描述符指向管道的写入端，wc 将永远不会看到文件结束。

The xv6 shell 实现了与上述代码类似（user/sh.c:101）的管道，例如 grep fork sh.c | wc -l。子进程创建一个管道来连接管道的左侧端和右侧端。然后它调用 fork 和 runcmd 来处理管道的左侧端，以及 fork 和 runcmd 来处理管道的右侧端，并等待它们都完成。管道的右侧端可能是一个本身包含管道的命令（例如，a | b | c），该命令本身会创建两个新的子进程（一个用于 b，一个用于 c）。因此，shell 可能会创建一个进程树。这棵树的叶子

of this tree are commands and the interior nodes are processes that wait until the left and right children complete.

Pipes may seem no more powerful than temporary files: the pipeline

```
echo hello world | wc
```

could be implemented without pipes as

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

Pipes have at least three advantages over temporary files in this situation. First, pipes automatically clean themselves up; with the file redirection, a shell would have to be careful to remove `/tmp/xyz` when done. Second, pipes can pass arbitrarily long streams of data, while file redirection requires enough free space on disk to store all the data. Third, pipes allow for parallel execution of pipeline stages, while the file approach requires the first program to finish before the second starts.

1.4 File system

The xv6 file system provides data files, which contain uninterpreted byte arrays, and directories, which contain named references to data files and other directories. The directories form a tree, starting at a special directory called the *root*. A *path* like `/a/b/c` refers to the file or directory named `c` inside the directory named `b` inside the directory named `a` in the root directory `/`. Paths that don't begin with `/` are evaluated relative to the calling process's *current directory*, which can be changed with the `chdir` system call. Both these code fragments open the same file (assuming all the directories involved exist):

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

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

The first fragment changes the process's current directory to `/a/b`; the second neither refers to nor changes the process's current directory.

There are system calls to create new files and directories: `mkdir` creates a new directory, `open` with the `O_CREATE` flag creates a new data file, and `mknod` creates a new device file. This example illustrates all three:

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

`mknod` creates a special file that refers to a device. Associated with a device file are the major and minor device numbers (the two arguments to `mknod`), which uniquely identify a kernel device. When a process later opens a device file, the kernel diverts `read` and `write` system calls to the kernel device implementation instead of passing them to the file system.

是命令，内部节点是等待左右子进程完成的进程。

管道似乎并不比临时文件更强大：管道

echo hello world | wc

完全可以不使用管道来实现

echo hello 世界 >/tmp/xyz; wc </tmp/xyz

在这种情况下，管道至少比临时文件有三种优势。首先，管道会自动清理自己；使用文件重定向时，shell需要小心地删除 /tmp/xyz。其次，管道可以传递任意长度的数据流，而文件重定向需要磁盘上有足够的空闲空间来存储所有数据。第三，管道允许管道阶段并行执行，而文件方法要求第一个程序完成之前第二个程序才能开始。

1.4 文件系统

xv6文件系统提供数据文件，其中包含未解释的字节数组，以及目录，其中包含对数据文件和其他目录的命名引用。目录形成一个树状结构，从称为根的特殊目录开始。像/a/b/c这样的路径引用根目录中名为a的目录内名为b的目录内名为c的文件或目录。不以/开头的路径相对于调用进程的当前目录进行解析，该目录可以使用chdir系统调用更改。这两个代码片段都打开同一个文件（假设所有涉及的目录都存在）：

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

打开("/a/b/c", O_RDONLY);

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

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

mkdir("/dir"); fd = open("/dir/file", O_CREATE|
O_WRONLY); 关闭(fd); 创建设备文件("/console", 1,
1);

mknod 创建一个指向设备的特殊文件。与设备文件关联的是主设备号和次设备号（mknod 的两个参数），它们唯一标识一个内核设备。当进程后来打开设备文件时，内核将读取和写入系统调用重定向到内核设备实现，而不是将它们传递给文件系统。

A file's name is distinct from the file itself; the same underlying file, called an *inode*, can have multiple names, called *links*. Each link consists of an entry in a directory; the entry contains a file name and a reference to an inode. An inode holds *metadata* about a file, including its type (file or directory or device), its length, the location of the file's content on disk, and the number of links to a file.

The `fstat` system call retrieves information from the inode that a file descriptor refers to. It fills in a `struct stat`, defined in `stat.h` (kernel/stat.h) as:

```
#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device

struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

The `link` system call creates another file system name referring to the same inode as an existing file. This fragment creates a new file named both `a` and `b`.

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

Reading from or writing to `a` is the same as reading from or writing to `b`. Each inode is identified by a unique *inode number*. After the code sequence above, it is possible to determine that `a` and `b` refer to the same underlying contents by inspecting the result of `fstat`: both will return the same inode number (`ino`), and the `nlink` count will be set to 2.

The `unlink` system call removes a name from the file system. The file's inode and the disk space holding its content are only freed when the file's link count is zero and no file descriptors refer to it. Thus adding

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

to the last code sequence leaves the inode and file content accessible as `b`. Furthermore,

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

is an idiomatic way to create a temporary inode with no name that will be cleaned up when the process closes `fd` or exits.

Unix provides file utilities callable from the shell as user-level programs, for example `mkdir`, `ln`, and `rm`. This design allows anyone to extend the command-line interface by adding new user-level programs. In hindsight this plan seems obvious, but other systems designed at the time of Unix often built such commands into the shell (and built the shell into the kernel).

One exception is `cd`, which is built into the shell (user/sh.c:161). `cd` must change the current working directory of the shell itself. If `cd` were run as a regular command, then the shell would

一个文件的名称与其本身是不同的；同一个底层文件，称为 inode，可以有多个名称，称为链接。每个链接都是目录中的一个条目；条目包含一个文件名称和一个指向 inode 的引用。inode 包含有关文件的信息，包括其类型（文件或目录或设备）、其长度、文件内容在磁盘上的位置以及指向文件的链接数。

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

```
#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device

struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

link系统调用创建一个指向现有文件相同inode的另一个文件系统名称。这段代码创建了一个名为a和b的新文件。

打开("a", O_创建 |O_只写); 链接("a", "
b");

从a读取或向a写入与从b读取或向b写入相同。每个inode由一个唯一的inode编号标识。在上述代码序列之后，通过检查fstat的结果，可以确定a和b指向相同的基本内容：两者都将返回相同的inode编号（ino），然后链接计数将被设置为2。

unlink系统调用从文件系统中删除一个名称。当文件的链接计数为零且没有文件描述符引用它时，文件inode及其内容占用的磁盘空间才会被释放。因此添加

unlink("a");

到最后一个代码序列将使 inode 和文件内容作为 b 可访问。此外，

文件描述符 = 创建/open/tmp/xyz, O_CREATE |O_
RDWR); unlink(/tmp/xyz);

这是一种创建无名称的临时 inode 的方法，当进程关闭 fd 或退出时将被清理。

Unix 提供可以从 shell 作为用户级程序调用的文件工具，例如 mkdir、ln 和 rm。这种设计允许任何人通过添加新的用户级程序来扩展命令行界面。回顾起来这个计划很明显，但在 Unix 时代设计的其他系统通常将此类命令集成到 shell（并将 shell 集成到内核）中。

一个例外是 cd，它被内置于 shell 中（user/sh.c:161）。cd 必须改变 shell 自身的当前工作目录。如果 cd 被作为普通命令运行，那么 shell 会

fork a child process, the child process would run `cd`, and `cd` would change the *child* 's working directory. The parent's (i.e., the shell's) working directory would not change.

1.5 Real world

Unix's combination of "standard" file descriptors, pipes, and convenient shell syntax for operations on them was a major advance in writing general-purpose reusable programs. The idea sparked a culture of "software tools" that was responsible for much of Unix's power and popularity, and the shell was the first so-called "scripting language." The Unix system call interface persists today in systems like BSD, Linux, and macOS.

The Unix system call interface has been standardized through the Portable Operating System Interface (POSIX) standard. Xv6 is *not* POSIX compliant: it is missing many system calls (including basic ones such as `lseek`), and many of the system calls it does provide differ from the standard. Our main goals for xv6 are simplicity and clarity while providing a simple UNIX-like system-call interface. Several people have extended xv6 with a few more system calls and a simple C library in order to run basic Unix programs. Modern kernels, however, provide many more system calls, and many more kinds of kernel services, than xv6. For example, they support networking, windowing systems, user-level threads, drivers for many devices, and so on. Modern kernels evolve continuously and rapidly, and offer many features beyond POSIX.

Unix unified access to multiple types of resources (files, directories, and devices) with a single set of file-name and file-descriptor interfaces. This idea can be extended to more kinds of resources; a good example is Plan 9 [16], which applied the "resources are files" concept to networks, graphics, and more. However, most Unix-derived operating systems have not followed this route.

The file system and file descriptors have been powerful abstractions. Even so, there are other models for operating system interfaces. Multics, a predecessor of Unix, abstracted file storage in a way that made it look like memory, producing a very different flavor of interface. The complexity of the Multics design had a direct influence on the designers of Unix, who aimed to build something simpler.

Xv6 does not provide a notion of users or of protecting one user from another; in Unix terms, all xv6 processes run as root.

This book examines how xv6 implements its Unix-like interface, but the ideas and concepts apply to more than just Unix. Any operating system must multiplex processes onto the underlying hardware, isolate processes from each other, and provide mechanisms for controlled inter-process communication. After studying xv6, you should be able to look at other, more complex operating systems and see the concepts underlying xv6 in those systems as well.

1.6 Exercises

1. Write a program that uses UNIX system calls to "ping-pong" a byte between two processes over a pair of pipes, one for each direction. Measure the program's performance, in exchanges per second.

fork 一个子进程，子进程会运行 cd，cd 会改变子进程的工作目录。父进程（即 shell）的工作目录不会改变。

1.5 现实世界

Unix 将“标准”文件描述符、管道以及方便的 shell 语法结合起来，用于操作它们，这是编写通用可重用程序的重大进步。这一理念催生了“软件工具”的文化，Unix 的强大和流行很大程度上归功于此，而 shell 是第一种所谓的“脚本语言”。Unix 系统调用接口至今仍存在于 BSD、Linux 和 macOS 等系统中。

Unix 系统调用接口已通过可移植操作系统接口（POSIX）标准进行了标准化。Xv6 并非 POSIX 兼容：它缺少许多系统调用（包括基本的 lseek 等），并且它提供的许多系统调用与标准不同。我们为 xv6 的主要目标是简单性和清晰性，同时提供一个简单的类 Unix 系统调用接口。许多人通过增加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，现代内核提供的系统调用和内核服务种类远多于 xv6。例如，它们支持网络、窗口系统、用户级线程、许多设备的驱动程序等。现代内核不断快速演进，并提供了许多超越 POSIX 的特性。

Unix 统一访问多种类型的资源（文件、目录和设备），使用一套文件名和文件描述符接口。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 [16], ，它将“资源是文件”的概念应用于网络、图形等。然而，大多数类 Unix 操作系统并没有遵循这条路线。

文件系统和文件描述符都是非常强大的抽象。即便如此，也存在其他操作系统接口模型。Unix 的前身 Multics 以一种使其看起来像内存的方式抽象了文件存储，产生了一种非常不同的接口风格。Multics 设计的复杂性直接影响了 Unix 的设计者，他们旨在构建一个更简单的东西。

Xv6 没有提供用户的概念，也没有保护一个用户免受另一个用户侵害的功能；在 Unix 术语中，所有 xv6 进程都作为 root 运行。

本书探讨了 xv6 如何实现其类 Unix 接口，但思想和概念不仅适用于 Unix。任何操作系统都必须将进程多路复用到底层硬件上，隔离进程彼此，并提供受控的进程间通信机制。学习完 xv6 后，你应该能够查看其他更复杂的操作系统，并看到 xv6 在那些系统中的底层概念。

1.6 练习

1. 编写一个程序，使用 UNIX 系统调用在两个进程之间通过一对管道（一个用于每个方向）来回传递一个字节。测量该程序的性能，以每秒交换次数计。

20

Chapter 2

Operating system organization

A key requirement for an operating system is to support several activities at once. For example, using the system call interface described in Chapter 1 a process can start new processes with `fork`. The operating system must *time-share* the resources of the computer among these processes. For example, even if there are more processes than there are hardware CPUs, the operating system must ensure that all of the processes get a chance to execute. The operating system must also arrange for *isolation* between the processes. That is, if one process has a bug and malfunctions, it shouldn't affect processes that don't depend on the buggy process. Complete isolation, however, is too strong, since it should be possible for processes to intentionally interact; pipelines are an example. Thus an operating system must fulfill three requirements: multiplexing, isolation, and interaction.

This chapter provides an overview of how operating systems are organized to achieve these three requirements. It turns out there are many ways to do so, but this text focuses on mainstream designs centered around a *monolithic kernel*, which is used by many Unix operating systems. This chapter also provides an overview of an xv6 process, which is the unit of isolation in xv6, and the creation of the first process when xv6 starts.

Xv6 runs on a *multi-core*[1] RISC-V microprocessor, and much of its low-level functionality (for example, its process implementation) is specific to RISC-V. RISC-V is a 64-bit CPU, and xv6 is written in "LP64" C, which means long (L) and pointers (P) in the C programming language are 64 bits, but an `int` is 32 bits. This book assumes the reader has done a bit of machine-level programming on some architecture, and will introduce RISC-V-specific ideas as they come up. The user-level ISA [2] and privileged architecture [3] documents are the complete specifications. You may also refer to "The RISC-V Reader: An Open Architecture Atlas" [15].

The CPU in a complete computer is surrounded by support hardware, much of it in the form of I/O interfaces. Xv6 is written for the support hardware simulated by qemu's "-machine virt" option. This includes RAM, a ROM containing boot code, a serial connection to the user's keyboard/screen, and a disk for storage.

[1]By "multi-core" this text means multiple CPUs that share memory but execute in parallel, each with its own set of registers. This text sometimes uses the term *multiprocessor* as a synonym for multi-core, though multiprocessor can also refer more specifically to a computer with several distinct processor chips.

第二章

操作系统组织

操作系统的一个关键要求是同时支持多项活动。例如，使用第一章中描述的系统调用接口，一个进程可以用 fork 启动新进程。操作系统必须分时共享计算机的资源给这些进程。例如，即使进程数量多于硬件 CPU 数量，操作系统必须确保所有进程都有执行的机会。操作系统还必须安排进程之间的隔离。也就是说，如果一个进程存在错误并发生故障，它不应该影响不依赖于该错误进程的其他进程。然而，完全隔离过于严格，因为进程应该能够有意交互；管道就是一个例子。因此，操作系统必须满足三个要求：多路复用、隔离和交互。

本章概述了操作系统如何组织以实现这三个要求。事实证明，有多种方法可以实现，但本文重点介绍以宏内核为中心的主流设计，这种设计被许多 Unix 操作系统使用。本章还概述了 xv6 进程，它是 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[1] RISC-V微处理器上，其大部分低级功能（例如，其进程实现）是针对RISC-V的。RISC-V是一款64位CPU，而xv6是用"LP64" C编写的，这意味着C编程语言中的长整型（L）和指针（P）是64位的，但int是32位的。本书假设读者在某个架构上做过一些机器级编程，并将随着内容的展开介绍RISC-V特有的概念。用户级ISA [2]和特权架构 [3]文档是完整的规范。你也可以参考"RISC-V读者：开放架构地图集" [15]。

一台完整的计算机中的CPU被支持硬件所包围，其中大部分以I/O接口的形式存在。xv6是为qemu的"-machine virt"选项模拟的支持硬件编写的。这包括RAM、一个包含启动代码的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。

[1]在本文本中，"多核"指的是共享内存但并行执行的多个CPU，每个CPU都有自己的寄存器集。本文本有时将"多处理器"作为"多核"的同义词使用，尽管多处理器也可以更具体地指代具有多个独立处理器芯片的计算机。

2.1 Abstracting physical resources

The first question one might ask when encountering an operating system is why have it at all? That is, one could implement the system calls in Figure 1.2 as a library, with which applications link. In this plan, each application could even have its own library tailored to its needs. Applications could directly interact with hardware resources and use those resources in the best way for the application (e.g., to achieve high or predictable performance). Some operating systems for embedded devices or real-time systems are organized in this way.

The downside of this library approach is that, if there is more than one application running, the applications must be well-behaved. For example, each application must periodically give up the CPU so that other applications can run. Such a *cooperative* time-sharing scheme may be OK if all applications trust each other and have no bugs. It's more typical for applications to not trust each other, and to have bugs, so one often wants stronger isolation than a cooperative scheme provides.

To achieve strong isolation it's helpful to forbid applications from directly accessing sensitive hardware resources, and instead to abstract the resources into services. For example, Unix applications interact with storage only through the file system's `open`, `read`, `write`, and `close` system calls, instead of reading and writing the disk directly. This provides the application with the convenience of pathnames, and it allows the operating system (as the implementer of the interface) to manage the disk. Even if isolation is not a concern, programs that interact intentionally (or just wish to keep out of each other's way) are likely to find a file system a more convenient abstraction than direct use of the disk.

Similarly, Unix transparently switches hardware CPUs among processes, saving and restoring register state as necessary, so that applications don't have to be aware of time-sharing. This transparency allows the operating system to share CPUs even if some applications are in infinite loops.

As another example, Unix processes use `exec` to build up their memory image, instead of directly interacting with physical memory. This allows the operating system to decide where to place a process in memory; if memory is tight, the operating system might even store some of a process's data on disk. `exec` also provides users with the convenience of a file system to store executable program images.

Many forms of interaction among Unix processes occur via file descriptors. Not only do file descriptors abstract away many details (e.g., where data in a pipe or file is stored), they are also defined in a way that simplifies interaction. For example, if one application in a pipeline fails, the kernel generates an end-of-file signal for the next process in the pipeline.

The system-call interface in Figure 1.2 is carefully designed to provide both programmer convenience and the possibility of strong isolation. The Unix interface is not the only way to abstract resources, but it has proved to be a good one.

2.2 User mode, supervisor mode, and system calls

Strong isolation requires a hard boundary between applications and the operating system. If the application makes a mistake, we don't want the operating system to fail or other applications to

2.1 抽象物理资源

当遇到一个操作系统时，人们可能会问的第一个问题是为什么要用它？也就是说，人们可以将图1.2中的系统调用作为一个库来实现，应用程序与之链接。在这个计划中，每个应用程序甚至可以拥有一个为其需求量身定制的库。应用程序可以直接与硬件资源交互，并以最适合应用程序的方式使用这些资源（例如，以实现高或可预测的性能）。一些嵌入式设备或实时系统的操作系统就是以这种方式组织的。

这种库方法的缺点是，如果有多个应用程序在运行，这些应用程序必须行为良好。例如，每个应用程序都必须定期放弃 CPU，以便其他应用程序可以运行。如果所有应用程序都互相信任且没有错误，这种协作分时方案可能是可以接受的。更典型的情况是应用程序之间不信任，并且存在错误，因此人们通常希望比协作方案提供的更强的隔离。

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

同样地，Unix 会透明地在进程之间切换硬件 CPU，根据需要保存和恢复寄存器状态，以便应用程序不必知道分时。这种透明性允许操作系统共享 CPU，即使某些应用程序处于无限循环中也是如此。

作为另一个例子，Unix 进程使用 exec 来构建它们的内存映像，而不是直接与物理内存交互。这允许操作系统决定进程在内存中的位置；如果内存紧张，操作系统甚至可以将进程的部分数据存储在磁盘上。exec 还为用户提供了一个文件系统的便利，用于存储可执行程序映像。

许多 Unix 进程之间的交互形式都通过文件描述符进行。文件描述符不仅抽象了许多细节（例如，管道或文件中数据的存储位置），而且其定义方式也简化了交互。例如，如果管道中的一个应用程序失败，内核将为管道中的下一个进程生成文件结束信号。

图1.2中的系统调用接口经过精心设计，旨在提供程序员便利性和强隔离的可能性。Unix接口并非抽象资源的唯一方式，但它已被证明是一个很好的选择。

2.2 用户模式、监督模式和系统调用

强隔离需要在应用程序和操作系统之间设置一个硬边界。如果应用程序出现错误，我们不想让操作系统失败或其他应用程序也

fail. Instead, the operating system should be able to clean up the failed application and continue running other applications. To achieve strong isolation, the operating system must arrange that applications cannot modify (or even read) the operating system's data structures and instructions and that applications cannot access other processes' memory.

CPUs provide hardware support for strong isolation. For example, RISC-V has three modes in which the CPU can execute instructions: *machine mode*, *supervisor mode*, and *user mode*. Instructions executing in machine mode have full privilege; a CPU starts in machine mode. Machine mode is mostly intended for setting up the computer during boot. Xv6 executes a few lines in machine mode and then changes to supervisor mode.

In supervisor mode the CPU is allowed to execute *privileged instructions*: for example, enabling and disabling interrupts, reading and writing the register that holds the address of a page table, etc. If an application in user mode attempts to execute a privileged instruction, then the CPU doesn't execute the instruction, but switches to supervisor mode so that supervisor-mode code can terminate the application, because it did something it shouldn't be doing. Figure 1.1 in Chapter 1 illustrates this organization. An application can execute only user-mode instructions (e.g., adding numbers, etc.) and is said to be running in *user space*, while the software in supervisor mode can also execute privileged instructions and is said to be running in *kernel space*. The software running in kernel space (or in supervisor mode) is called the *kernel*.

An application that wants to invoke a kernel function (e.g., the `read` system call in xv6) must transition to the kernel; an application *cannot* invoke a kernel function directly. CPUs provide a special instruction that switches the CPU from user mode to supervisor mode and enters the kernel at an entry point specified by the kernel. (RISC-V provides the `ecall` instruction for this purpose.) Once the CPU has switched to supervisor mode, the kernel can then validate the arguments of the system call (e.g., check if the address passed to the system call is part of the application's memory), decide whether the application is allowed to perform the requested operation (e.g., check if the application is allowed to write the specified file), and then deny it or execute it. It is important that the kernel control the entry point for transitions to supervisor mode; if the application could decide the kernel entry point, a malicious application could, for example, enter the kernel at a point where the validation of arguments is skipped.

2.3 Kernel organization

A key design question is what part of the operating system should run in supervisor mode. One possibility is that the entire operating system resides in the kernel, so that the implementations of all system calls run in supervisor mode. This organization is called a *monolithic kernel*.

In this organization the entire operating system consists of a single program running with full hardware privilege. This organization is convenient because the OS designer doesn't have to decide which parts of the operating system don't need full hardware privilege. Furthermore, it is easier for different parts of the operating system to cooperate. For example, an operating system might have a buffer cache that can be shared both by the file system and the virtual memory system.

A downside of the monolithic organization is that the interactions among different parts of the operating system are often complex (as we will see in the rest of this text), and therefore it

失败。相反，操作系统应该能够清理失败的应用程序并继续运行其他应用程序。为了实现强隔离，操作系统必须确保应用程序不能修改（甚至读取）操作系统的数据结构和指令，并且应用程序不能访问其他进程的内存。

CPU提供硬件强隔离支持。例如，RISC-V有三种CPU可执行的指令模式：机器模式、监督模式和用户模式。在机器模式下执行的指令具有完全特权；CPU从机器模式启动。机器模式主要用于启动时设置计算机。xv6在机器模式下执行几行指令后，切换到监督模式。

在监督模式下，CPU允许执行特权指令：例如，启用和禁用中断、读取和写入保存页表地址的寄存器等。如果用户模式的应用程序尝试执行特权指令，则CPU不会执行该指令，而是切换到监督模式，以便监督模式代码可以终止应用程序，因为它做了不该做的事情。第一章的图1.1说明了这种结构。应用程序只能执行用户模式指令（例如，加法等），并称其为在用户空间运行，而监督模式中的软件也可以执行特权指令，并称其为在内核空间运行。在内核空间（或在监督模式下）运行的软件称为内核。

一个想要调用内核函数的应用程序（例如，在xv6中的读取系统调用）必须切换到内核；应用程序不能直接调用内核函数。CPU提供了一条特殊指令，该指令将CPU从用户模式切换到监督模式，并进入内核指定的入口点。（RISC-V为此提供了ecall指令。）一旦CPU切换到监督模式，内核就可以验证系统调用的参数（例如，检查传递给系统调用的地址是否是应用程序内存的一部分），决定应用程序是否被允许执行请求的操作（例如，检查应用程序是否被允许写入指定的文件），然后拒绝执行或执行它。重要的是内核控制切换到监督模式的入口点；如果应用程序可以决定内核入口点，恶意应用程序可以，例如，在跳过参数验证的点进入内核。

2.3 内核组织

一个关键的设计问题是操作系统应该以监督模式运行哪部分。一种可能性是整个操作系统驻留在内核中，以便所有系统调用的实现都在监督模式下运行。这种组织方式称为宏内核。

在这种组织方式中，整个操作系统由一个以完整硬件权限运行的单一程序组成。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些部分不需要完整硬件权限。此外，操作系统不同部分之间的协作更容易。例如，操作系统可能有一个缓冲区缓存，该缓存既可以被文件系统共享，也可以被虚拟内存系统共享。

单体组织的缺点在于，操作系统不同部分之间的交互通常复杂（正如我们将在本文其余部分看到的那样），因此它

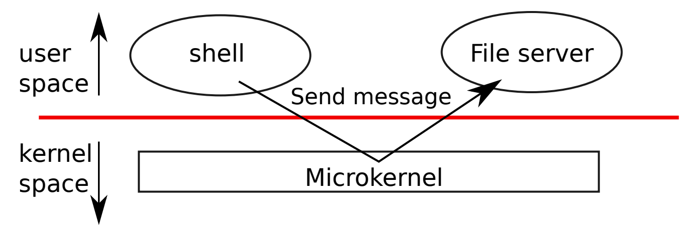

Figure 2.1: A microkernel with a file-system server

is easy for an operating system developer to make a mistake. In a monolithic kernel, a mistake is fatal, because an error in supervisor mode will often cause the kernel to fail. If the kernel fails, the computer stops working, and thus all applications fail too. The computer must reboot to start again.

To reduce the risk of mistakes in the kernel, OS designers can minimize the amount of operating system code that runs in supervisor mode, and execute the bulk of the operating system in user mode. This kernel organization is called a *microkernel*.

Figure 2.1 illustrates this microkernel design. In the figure, the file system runs as a user-level process. OS services running as processes are called servers. To allow applications to interact with the file server, the kernel provides an inter-process communication mechanism to send messages from one user-mode process to another. For example, if an application like the shell wants to read or write a file, it sends a message to the file server and waits for a response.

In a microkernel, the kernel interface consists of a few low-level functions for starting applications, sending messages, accessing device hardware, etc. This organization allows the kernel to be relatively simple, as most of the operating system resides in user-level servers.

In the real world, both monolithic kernels and microkernels are popular. Many Unix kernels are monolithic. For example, Linux has a monolithic kernel, although some OS functions run as user-level servers (e.g., the window system). Linux delivers high performance to OS-intensive applications, partially because the subsystems of the kernel can be tightly integrated.

Operating systems such as Minix, L4, and QNX are organized as a microkernel with servers, and have seen wide deployment in embedded settings. A variant of L4, seL4, is small enough that it has been verified for memory safety and other security properties [8].

There is much debate among developers of operating systems about which organization is better, and there is no conclusive evidence one way or the other. Furthermore, it depends much on what “better” means: faster performance, smaller code size, reliability of the kernel, reliability of the complete operating system (including user-level services), etc.

There are also practical considerations that may be more important than the question of which organization. Some operating systems have a microkernel but run some of the user-level services in kernel space for performance reasons. Some operating systems have monolithic kernels because that is how they started and there is little incentive to move to a pure microkernel organization, because new features may be more important than rewriting the existing operating system to fit a microkernel design.

From this book’s perspective, microkernel and monolithic operating systems share many key ideas. They implement system calls, they use page tables, they handle interrupts, they support

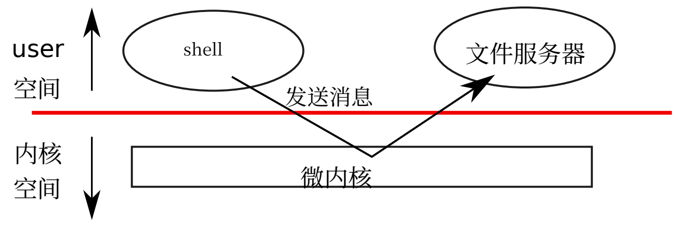

图2.1：一个带有文件系统服务器的微内核

容易让操作系统开发者犯错误。在宏内核中，一个错误是致命的，因为监督模式中的错误通常会导致内核崩溃。如果内核崩溃，计算机停止工作，因此所有应用程序也会失败。计算机必须重新启动才能再次启动。

为了降低内核中的错误风险，操作系统设计者可以最小化在监督模式下运行的操作系统代码量，并在用户模式下执行大部分操作系统代码。这种内核结构称为微内核。

图2.1说明了这种微内核设计。在图中，文件系统作为用户级进程运行。作为进程运行的操作系统服务被称为服务器。为了允许应用程序与文件服务器交互，内核提供了一种进程间通信机制，用于从一个用户模式进程向另一个用户模式进程发送消息。例如，如果像shell这样的应用程序想要读取或写入文件，它将向文件服务器发送消息并等待响应。

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

在现实世界中，单体内核和微内核都很受欢迎。许多Unix内核是单体的。例如，Linux有一个单体内核，尽管一些操作系统功能作为用户级服务器运行（例如，窗口系统）。Linux为操作系统密集型应用程序提供高性能，部分原因是内核的子系统可以紧密集成。

像Minix、L4和QNX这样的操作系统以带有服务器的微内核结构组织，并在嵌入式环境中得到了广泛应用。L4的一个变体seL4足够小，以至于它已经通过了内存安全和其它安全特性的验证 [8]。

操作系统开发者之间关于哪种结构更好的争论很多，但双方都没有确凿的证据。此外，这很大程度上取决于“更好”意味着什么：更快的性能、更小的代码大小、内核的可靠性、完整操作系统的可靠性（包括用户级服务）等等。

还有一些实际考虑可能比哪种结构更重要。一些操作系统有微内核，但出于性能原因，一些用户级服务在内核空间运行。一些操作系统有单体内核，因为它们从一开始就是这样，并且几乎没有动力转向纯微内核结构，因为新功能可能比重写现有操作系统以适应微内核设计更重要。

从本书的角度来看，微内核和单体内核操作系统共享许多关键思想。它们实现系统调用，它们使用页表，它们处理中断，它们支持

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

Figure 2.2: Xv6 kernel source files.

processes, they use locks for concurrency control, they implement a file system, etc. This book focuses on these core ideas.

Xv6 is implemented as a monolithic kernel, like most Unix operating systems. Thus, the xv6 kernel interface corresponds to the operating system interface, and the kernel implements the complete operating system. Since xv6 doesn't provide many services, its kernel is smaller than some microkernels, but conceptually xv6 is monolithic.

2.4 Code: xv6 organization

The xv6 kernel source is in the `kernel/` sub-directory. The source is divided into files, following a rough notion of modularity; Figure 2.2 lists the files. The inter-module interfaces are defined in

File	描述
bio.c	文件系统的磁盘块缓存。
console.c	连接到用户键盘和屏幕。
entry.S	非常最初的启动指令。
exec.c	exec() 系统调用。
file.c	文件描述符支持。
fs.c	文件系统。
kalloc.c	物理页分配器。
kernelvec.S	处理来自内核的陷阱。
log.c	文件系统日志记录和崩溃恢复。
main.c	控制启动期间其他模块的初始化。
pipe.c	管道。
plic.c	RISC-V 中断控制器。
printf.c	格式化输出到控制台。
proc.c	进程和调度。
sleeplock.c	让出CPU的锁。
spinlock.c	不释放CPU的锁。
start.c	早期机器模式的启动代码。
string.c	C字符串和字节数组的库。
swtch.S	线程切换。
syscall.c	调度系统调用至处理函数。
sysfile.c	文件相关系统调用。
sysproc.c	进程相关系统调用。
trampoline.S	用于在用户和内核之间切换的汇编代码。
trap.c	用于处理和从陷阱和中断返回的C代码。
uart.c	串口控制台设备驱动。
virtio 磁盘.c_	磁盘设备驱动程序。
vm.c	管理页表和地址空间。

图2.2：Xv6内核源文件。

进程，它们使用锁进行并发控制，它们实现文件系统，等等。本书关注这些核心思想。

Xv6 作为宏内核实现，与大多数 Unix 操作系统类似。因此，xv6 内核接口对应于操作系统接口，内核实现了完整的操作系统。由于 xv6 不提供许多服务，其内核比一些微内核小，但概念上 xv6 是宏内核。

2.4 代码：xv6 组织

Xv6内核源代码位于kernel/子目录中。源代码被划分为文件，遵循一种大致的模块化概念；图2.2列出了这些文件。模块间接口在

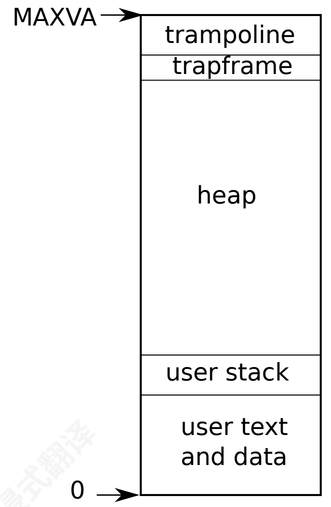

Figure 2.3: Layout of a process's virtual address space

`defs.h` (kernel/defs.h).

2.5 Process overview

The unit of isolation in xv6 (as in other Unix operating systems) is a *process*. The process abstraction prevents one process from wrecking or spying on another process's memory, CPU, file descriptors, etc. It also prevents a process from wrecking the kernel itself, so that a process can't subvert the kernel's isolation mechanisms. The kernel must implement the process abstraction with care because a buggy or malicious application may trick the kernel or hardware into doing something bad (e.g., circumventing isolation). The mechanisms used by the kernel to implement processes include the user/supervisor mode flag, address spaces, and time-slicing of threads.

To help enforce isolation, the process abstraction provides the illusion to a program that it has its own private machine. A process provides a program with what appears to be a private memory system, or *address space*, which other processes cannot read or write. A process also provides the program with what appears to be its own CPU to execute the program's instructions.

Xv6 uses page tables (which are implemented by hardware) to give each process its own address space. The RISC-V page table translates (or "maps") a *virtual address* (the address that an RISC-V instruction manipulates) to a *physical address* (an address that the CPU sends to main memory).

Xv6 maintains a separate page table for each process that defines that process's address space. As illustrated in Figure 2.3, an address space includes the process's *user memory* starting at virtual address zero. Instructions come first, followed by global variables, then the stack, and finally a "heap" area (for malloc) that the process can expand as needed. There are a number of factors that limit the maximum size of a process's address space: pointers on the RISC-V are 64 bits wide; the hardware uses only the low 39 bits when looking up virtual addresses in page tables; and xv6 uses only 38 of those 39 bits. Thus, the maximum address is $2^{38} - 1$ = 0x3fffffffff, which

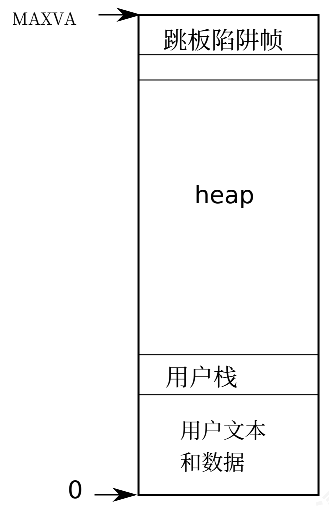

图2.3：进程虚拟地址空间的布局

defs.h (kernel/defs.h)。

2.5 进程概述

在xv6（以及其他Unix操作系统）中，隔离的单位是进程。进程抽象防止一个进程破坏或窥探另一个进程的内存、CPU、文件描述符等。它还防止进程破坏内核本身，因此进程不能绕过内核的隔离机制。内核必须小心地实现进程抽象，因为有错误或恶意的应用程序可能会欺骗内核或硬件做坏事（例如，绕过隔离）。内核用于实现进程的机制包括用户/监督模式标志、地址空间和线程时间片。

为了帮助执行隔离，进程抽象向程序提供了一种它拥有自己的私有机器的错觉。进程向程序提供了一种看似私有的内存系统或地址空间，其他进程无法读取或写入。进程还向程序提供了一种看似它自己的CPU来执行程序的指令。

Xv6 使用页表（由硬件实现）为每个进程提供各自的地址空间。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 发送到主内存的地址）。

Xv6 为每个进程维护一个单独的页表，该页表定义了该进程的地址空间。如图 2.3 所示，地址空间包括进程的用户内存，起始虚拟地址为零。指令首先出现，其次是全局变量，然后是栈，最后是一个"堆"区域（用于 malloc），进程可以根据需要扩展该区域。有多个因素限制了进程地址空间的最大大小：RISC-V 上的指针是 64 位宽；硬件在页表中查找虚拟地址时仅使用低 39 位；而 xv6 仅使用其中的 38 位。因此，最大地址是 $2^{38} - 1$ = 0x3fffffffff，这

is `MAXVA` (kernel/riscv.h:378). At the top of the address space xv6 places a *trampoline* page (4096 bytes) and a *trapframe* page. Xv6 uses these two pages to transition into the kernel and back; the trampoline page contains the code to transition in and out of the kernel, and the trapframe is where the kernel saves the process's user registers, as Chapter 4 explains.

The xv6 kernel maintains many pieces of state for each process, which it gathers into a `struct proc` (kernel/proc.h:85). A process's most important pieces of kernel state are its page table, its kernel stack, and its run state. We'll use the notation `p->xxx` to refer to elements of the `proc` structure; for example, `p->pagetable` is a pointer to the process's page table.

Each process has a thread of control (or *thread* for short) that holds the state needed to execute the process. At any given time, a thread might be executing on a CPU, or suspended (not executing, but capable of resuming executing in the future). To switch a CPU between processes, the kernel suspends the thread currently running on that CPU and saves its state, and restores the state of another process's previously-suspended thread. Much of the state of a thread (local variables, function call return addresses) is stored on the thread's stacks. Each process has two stacks: a user stack and a kernel stack (`p->kstack`). When the process is executing user instructions, only its user stack is in use, and its kernel stack is empty. When the process enters the kernel (for a system call or interrupt), the kernel code executes on the process's kernel stack; while a process is in the kernel, its user stack still contains saved data, but isn't actively used. A process's thread alternates between actively using its user stack and its kernel stack. The kernel stack is separate (and protected from user code) so that the kernel can execute even if a process has wrecked its user stack.

A process can make a system call by executing the RISC-V `ecall` instruction. This instruction raises the hardware privilege level and changes the program counter to a kernel-defined entry point. The code at the entry point switches to the process's kernel stack and executes the kernel instructions that implement the system call. When the system call completes, the kernel switches back to the user stack and returns to user space by calling the `sret` instruction, which lowers the hardware privilege level and resumes executing user instructions just after the system call instruction. A process's thread can "block" in the kernel to wait for I/O, and resume where it left off when the I/O has finished.

`p->state` indicates whether the process is allocated, ready to run, currently running on a CPU, waiting for I/O, or exiting.

`p->pagetable` holds the process's page table, in the format that the RISC-V hardware expects. Xv6 causes the paging hardware to use a process's `p->pagetable` when executing that process in user space. A process's page table also serves as the record of the addresses of the physical pages allocated to store the process's memory.

In summary, a process bundles two design ideas: an address space to give a process the illusion of its own memory, and a thread to give the process the illusion of its own CPU. In xv6, a process consists of one address space and one thread. In real operating systems a process may have more than one thread to take advantage of multiple CPUs.

是 MAXVA (kernel/riscv.h:378)。在地址空间的顶部，xv6 放置一个跳板页（4096 字节）和一个陷阱帧页。Xv6 使用这两个页来进入内核并返回；跳板页包含进入和退出内核的代码，而陷阱帧是内核保存进程用户寄存器的地方，第 4 章对此进行了说明。

xv6 内核为每个进程维护许多状态信息，它将这些信息收集到一个 struct proc (kernel/proc.h:85) 结构中。进程最重要的内核状态信息包括它的页表、内核栈和运行状态。我们将 proc 结构的元素表示为 p->xxx 的形式；例如，p->pagetable 是一个指向进程页表的指针。

每个进程都有一个控制线程（或简称线程），它持有执行进程所需的全部状态信息。在任何给定时刻，一个线程可能正在 CPU 上执行，或者处于挂起状态（不执行，但具备未来恢复执行的能力）。为了在进程之间切换 CPU，内核会挂起当前在该 CPU 上运行的线程并保存其状态，然后恢复另一个进程之前挂起的线程的状态。线程的大部分状态信息（局部变量、函数调用返回地址）都存储在它的栈上。每个进程有两个栈：用户栈和内核栈（p->kstack）。当进程执行用户指令时，只有它的用户栈在使用中，而内核栈是空的。当进程进入内核（执行系统调用或中断时），内核代码在进程的内核栈上执行；当一个进程处于内核中时，它的用户栈仍然包含保存的数据，但不会被主动使用。进程的线程会在用户栈和内核栈之间交替使用。内核栈是独立的（并且受保护，防止用户代码访问），这样即使进程损坏了它的用户栈，内核仍然可以执行。

一个进程可以通过执行RISC-Vecall指令进行系统调用。该指令会提升硬件特权级并将程序计数器切换到内核定义的入口点。入口点的代码会切换到进程的内核栈并执行实现系统调用的内核指令。当系统调用完成后，内核会切换回用户栈并通过调用sret指令返回用户空间，该指令会降低硬件特权级并恢复执行系统调用指令之后的用户指令。一个进程的线程可以在内核中"阻塞"以等待I/O，并在I/O完成后从之前的位置继续执行。

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

p->pagetable持有进程的页表，格式为RISC-V硬件所期望的格式。Xv6在用户空间执行进程时，会指示分页硬件使用该进程的p->pagetable。进程的页表也作为记录分配给进程内存的物理页地址的记录。

总而言之，一个进程捆绑了两种设计思想：一个地址空间来给进程带来自己的内存的错觉，和一个线程来给进程带来自己的CPU的错觉。在xv6中，一个进程由一个地址空间和一个线程组成。在真实操作系统中，一个进程可能有多个线程来利用多个CPU。

2.6 Code: starting xv6, the first process and system call

To make xv6 more concrete, we'll outline how the kernel starts and runs the first process. The subsequent chapters will describe the mechanisms that show up in this overview in more detail.

When the RISC-V computer powers on, it initializes itself and runs a boot loader which is stored in read-only memory. The boot loader loads the xv6 kernel into memory. Then, in machine mode, the CPU executes xv6 starting at `_entry` (kernel/entry.S:7). The RISC-V starts with paging hardware disabled: virtual addresses map directly to physical addresses.

The loader loads the xv6 kernel into memory at physical address `0x80000000`. The reason it places the kernel at `0x80000000` rather than `0x0` is because the address range `0x0:0x80000000` contains I/O devices.

The instructions at `_entry` set up a stack so that xv6 can run C code. Xv6 declares space for an initial stack, `stack0`, in the file `start.c` (kernel/start.c:11). The code at `_entry` loads the stack pointer register `sp` with the address `stack0+4096`, the top of the stack, because the stack on RISC-V grows down. Now that the kernel has a stack, `_entry` calls into C code at `start` (kernel/start.c:15).

The function `start` performs some configuration that is only allowed in machine mode, and then switches to supervisor mode. To enter supervisor mode, RISC-V provides the instruction `mret`. This instruction is most often used to return from a previous call from supervisor mode to machine mode. `start` isn't returning from such a call, but sets things up as if it were: it sets the previous privilege mode to supervisor in the register `mstatus`, it sets the return address to `main` by writing `main`'s address into the register `mepc`, disables virtual address translation in supervisor mode by writing `0` into the page-table register `satp`, and delegates all interrupts and exceptions to supervisor mode.

Before jumping into supervisor mode, `start` performs one more task: it programs the clock chip to generate timer interrupts. With this housekeeping out of the way, `start` "returns" to supervisor mode by calling `mret`. This causes the program counter to change to `main` (kernel/main.c:11), the address previously stored in `mepc`.

After `main` (kernel/main.c:11) initializes several devices and subsystems, it creates the first process by calling `userinit` (kernel/proc.c:233). The first process executes a small program written in RISC-V assembly, which makes the first system call in xv6. `initcode.S` (user/initcode.S:3) loads the number for the `exec` system call, `SYS_EXEC` (kernel/syscall.h:8), into register `a7`, and then calls `ecall` to re-enter the kernel.

The kernel uses the number in register `a7` in `syscall` (kernel/syscall.c:132) to call the desired system call. The system call table (kernel/syscall.c:107) maps `SYS_EXEC` to the function `sys_exec`, which the kernel invokes. As we saw in Chapter 1, `exec` replaces the memory and registers of the current process with a new program (in this case, `/init`).

Once the kernel has completed `exec`, it returns to user space in the `/init` process. `init` (user/init.c:15) creates a new console device file if needed and then opens it as file descriptors 0, 1, and 2. Then it starts a shell on the console. The system is up.

2.6 代码：启动 xv6，第一个进程和系统调用

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

当 RISC-V 计算机开机时，它会初始化自己并运行存储在只读内存中的引导加载程序。引导加载程序将 xv6 内核加载到内存中。然后，在机器模式下，CPU 从 _entry（kernel/entry.S:7）开始执行 xv6。RISC-V 以分页硬件禁用状态启动：虚拟地址直接映射到物理地址。

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

位于 _entry 的指令设置了一个栈，以便 xv6 可以运行 C 代码。Xv6 在文件 start.c（kernel/start.c:11）中声明了初始栈空间 stack0。位于 _entry 的代码将栈指针寄存器 sp 加载为地址 stack0+4096，即栈顶，因为 RISC-V 上的栈向下增长。现在内核有了栈，_entry 调用 kernel/start.c:15 中的 C 代码 start。

函数 start 执行一些仅在机器模式下允许的配置，然后切换到监督模式。要进入监督模式，RISC-V 提供了 mret 指令。此指令通常用于从之前的调用中返回，从监督模式到机器模式。start 并不是从这样的调用中返回，但它像这样设置：它将先前的特权模式设置为监督模式在寄存器 mstatus 中，它将返回地址设置为 main，通过将 main 的地址写入寄存器 mepc，通过将 0 写入页表寄存器 satp 禁用监督模式中的虚拟地址转换，并将所有中断和异常委托给监督模式。

在跳转到监督模式之前，start 执行一项更多任务：它编程时钟芯片以生成定时器中断。处理完这些杂务后，start 通过调用 mret "返回"到监督模式。这会导致程序计数器变为 main (kernel/main.c:11)，先前存储在 mepc 中的地址。

在 main (kernel/main.c:11) 初始化多个设备和子系统后，它通过调用 userinit (kernel/proc.c:233) 创建第一个进程。第一个进程执行一个用 RISC-V 汇编语言编写的小程序，这是在 xv6 中做出的第一个系统调用。initcode.S (user/initcode.S:3) 将 exec 系统调用 SYS_EXEC (kernel/syscall.h:8) 的编号加载到寄存器 a7 中，然后调用 ecall 重新进入内核。

内核使用寄存器 a7 中的编号在 syscall (kernel/syscall.c:132) 中调用所需的系统调用。系统调用表(kernel/syscall.c:107) 将 SYS_EXEC 映射到内核调用的函数 sys_exec。正如我们在第一章中看到的，exec 用一个新程序（在这种情况下是 /init）替换当前进程的内存和寄存器。

一旦内核完成 exec，它返回到 /init 进程的用户空间。init (user/init.c:15) 如果需要会创建一个新的控制台设备文件，然后将其作为文件描述符 0、1 和 2 打开。接着它在控制台上启动一个 shell。系统已启动。

2.7 Security Model

You may wonder how the operating system deals with buggy or malicious code. Because coping with malice is strictly harder than dealing with accidental bugs, it's reasonable to focus mostly on providing security against malice. Here's a high-level view of typical security assumptions and goals in operating system design.

The operating system must assume that a process's user-level code will do its best to wreck the kernel or other processes. User code may try to dereference pointers outside its allowed address space; it may attempt to execute any RISC-V instructions, even those not intended for user code; it may try to read and write any RISC-V control register; it may try to directly access device hardware; and it may pass clever values to system calls in an attempt to trick the kernel into crashing or doing something stupid. The kernel's goal is to restrict each user processes so that all it can do is read/write/execute its own user memory, use the 32 general-purpose RISC-V registers, and affect the kernel and other processes in the ways that system calls are intended to allow. The kernel must prevent any other actions. This is typically an absolute requirement in kernel design.

The expectations for the kernel's own code are quite different. Kernel code is assumed to be written by well-meaning and careful programmers. Kernel code is expected to be bug-free, and certainly to contain nothing malicious. This assumption affects how we analyze kernel code. For example, there are many internal kernel functions (e.g., the spin locks) that would cause serious problems if kernel code used them incorrectly. When examining any specific piece of kernel code, we'll want to convince ourselves that it behaves correctly. We assume, however, that kernel code in general is correctly written, and follows all the rules about use of the kernel's own functions and data structures. At the hardware level, the RISC-V CPU, RAM, disk, etc. are assumed to operate as advertised in the documentation, with no hardware bugs.

Of course in real life things are not so straightforward. It's difficult to prevent clever user code from making a system unusable (or causing it to panic) by consuming kernel-protected resources – disk space, CPU time, process table slots, etc. It's usually impossible to write bug-free kernel code or design bug-free hardware; if the writers of malicious user code are aware of kernel or hardware bugs, they will exploit them. Even in mature, widely-used kernels, such as Linux, people discover new vulnerabilities continuously [1]. It's worthwhile to design safeguards into the kernel against the possibility that it has bugs: assertions, type checking, stack guard pages, etc. Finally, the distinction between user and kernel code is sometimes blurred: some privileged user-level processes may provide essential services and effectively be part of the operating system, and in some operating systems privileged user code can insert new code into the kernel (as with Linux's loadable kernel modules).

2.8 Real world

Most operating systems have adopted the process concept, and most processes look similar to xv6's. Modern operating systems, however, support several threads within a process, to allow a single process to exploit multiple CPUs. Supporting multiple threads in a process involves quite a bit of machinery that xv6 doesn't have, often including interface changes (e.g., Linux's `clone`, a

2.7 安全模型

您可能会想操作系统如何处理有错误或恶意的代码。因为应对恶意行为比处理意外错误要严格得多，因此主要关注提供针对恶意的安全性是合理的。以下是操作系统设计中典型安全假设和目标的概览。

操作系统必须假设一个进程的用户级代码会尽力破坏内核或其他进程。用户代码可能会尝试引用其允许地址空间之外的指针；它可能会尝试执行任何 RISC-V 指令，即使那些并非为用户代码设计的指令；它可能会尝试读取和写入任何 RISC-V 控制寄存器；它可能会尝试直接访问设备硬件；并且它可能会向系统调用传递巧妙的值，试图欺骗内核崩溃或做些愚蠢的事情。内核的目标是限制每个用户进程，使其只能读取/写入/执行自己的用户内存，使用 32 个通用 RISC-V 寄存器，并以系统调用允许的方式影响内核和其他进程。内核必须阻止任何其他操作。这在内核设计中通常是一个绝对的要求。

对内核自身代码的期望则大不相同。假设内核代码是由善意且谨慎的程序员编写的。内核代码被期望没有错误，当然也绝不含恶意内容。这一假设影响了我们如何分析内核代码。例如，有许多内核内部函数（例如自旋锁）如果内核代码使用不当，会导致严重问题。在检查任何特定的内核代码片段时，我们将要确信其行为正确。然而，我们假设内核代码总体上是正确编写的，并遵循所有关于使用内核自身函数和数据结构的规则。在硬件层面，RISC-V CPU、RAM、磁盘等被假设按照文档中所述的方式运行，没有任何硬件错误。

当然在现实生活中事情并不那么简单。很难阻止聪明的用户代码通过消耗内核保护资源（如磁盘空间、CPU 时间、进程表槽位等）使系统无法使用（或导致系统恐慌）。通常不可能编写无错误的内核代码或设计无错误的硬件；如果恶意用户代码的编写者了解内核或硬件错误，他们会利用这些错误。即使在成熟、广泛使用的内核中，例如 Linux，人们会不断发现新的漏洞 [1]。在内核中设计保护措施以防止其存在错误是有价值的：断言、类型检查、栈保护页等。最后，用户代码和内核代码之间的区别有时会变得模糊：一些特权用户级进程可能会提供基本服务，并实际上成为操作系统的一部分，并且在某些操作系统中，特权用户代码可以将新代码插入内核（例如 Linux 的可加载内核模块）。

2.8 现实世界

大多数操作系统都采用了进程概念，并且大多数进程看起来与xv6的类似。然而，现代操作系统支持一个进程内的多个线程，以允许单个进程利用多个CPU。在进程内支持多线程涉及相当多的机制，这些机制xv6没有，通常包括接口变更（例如，Linux的clone，一个fork的变体），以控制进程线程共享哪些方面。

variant of `fork`), to control which aspects of a process threads share.

2.9 Exercises

1. Add a system call to xv6 that returns the amount of free memory available.

变体），以控制进程线程共享哪些方面。

2.9 练习

1. 为xv6添加一个系统调用，返回可用空闲内存的量。

Chapter 3

Page tables

Page tables are the most popular mechanism through which the operating system provides each process with its own private address space and memory. Page tables determine what memory addresses mean, and what parts of physical memory can be accessed. They allow xv6 to isolate different process's address spaces and to multiplex them onto a single physical memory. Page tables are a popular design because they provide a level of indirection that allow operating systems to perform many tricks. Xv6 performs a few tricks: mapping the same memory (a trampoline page) in several address spaces, and guarding kernel and user stacks with an unmapped page. The rest of this chapter explains the page tables that the RISC-V hardware provides and how xv6 uses them.

3.1 Paging hardware

As a reminder, RISC-V instructions (both user and kernel) manipulate virtual addresses. The machine's RAM, or physical memory, is indexed with physical addresses. The RISC-V page table hardware connects these two kinds of addresses, by mapping each virtual address to a physical address.

Xv6 runs on Sv39 RISC-V, which means that only the bottom 39 bits of a 64-bit virtual address are used; the top 25 bits are not used. In this Sv39 configuration, a RISC-V page table is logically an array of 2^{27} (134,217,728) *page table entries (PTEs)*. Each PTE contains a 44-bit physical page number (PPN) and some flags. The paging hardware translates a virtual address by using the top 27 bits of the 39 bits to index into the page table to find a PTE, and making a 56-bit physical address whose top 44 bits come from the PPN in the PTE and whose bottom 12 bits are copied from the original virtual address. Figure 3.1 shows this process with a logical view of the page table as a simple array of PTEs (see Figure 3.2 for a fuller story). A page table gives the operating system control over virtual-to-physical address translations at the granularity of aligned chunks of 4096 (2^{12}) bytes. Such a chunk is called a *page*.

In Sv39 RISC-V, the top 25 bits of a virtual address are not used for translation. The physical address also has room for growth: there is room in the PTE format for the physical page number to grow by another 10 bits. The designers of RISC-V chose these numbers based on technology predictions. 2^{39} bytes is 512 GB, which should be enough address space for applications running

第 3 章

页表

页表是操作系统为每个进程提供其私有地址空间和内存的最常用机制。页表决定了内存地址的含义，以及哪些物理内存可以被访问。它们允许xv6隔离不同进程的地址空间，并将它们多路复用到一个物理内存上。页表是一种流行的设计，因为它们提供了一种间接性，允许操作系统执行许多技巧。xv6执行了一些技巧：在多个地址空间中映射相同的内存（一个跳板页），并用一个未映射的页守护内核和用户栈。本章的其余部分解释了RISC-V硬件提供的页表以及xv6如何使用它们。

3.1 分页硬件

作为提醒，RISC-V 指令（包括用户和内核）操作虚拟地址。Thema- chine 的 RAM，或物理内存，使用物理地址索引。RISC-V 页表硬件通过将每个虚拟地址映射到物理地址来连接这两种地址。

Xv6 运行在 Sv39 RISC-V 上，这意味着 64 位虚拟地址的底部 39 位被使用；顶部 25 位未被使用。在此 Sv39 配置中，RISC-V 页表在逻辑上是一个包含 2^{27} (134,217,728) 个页表项 (PTE) 的数组。每个 PTE 包含一个 44 位的物理页码 (PPN) 和一些标志。分页硬件通过使用 39 位的顶部 27 位来索引页表以查找 PTE，并生成一个 56 位的物理地址，其顶部 44 位来自 PTE 中的 PPN，底部 12 位从原始虚拟地址复制。图 3.1 展示了这个过程，将页表逻辑地视为一个简单的 PTE 数组（有关更详细的信息，请参阅图 3.2）。页表使操作系统能够在 4096 (2^{12}) 字节的对齐块粒度上控制虚拟到物理地址的转换。这样的块称为页。

在Sv39RISC-V中，虚拟地址的最高25位不用于翻译。物理地址也有扩展空间：PTE格式中物理页码可以再扩展10位。RISC-V的设计者根据技术预测选择了这些数字。 2^{39} 字节是512 GB，应该足够应用程序运行

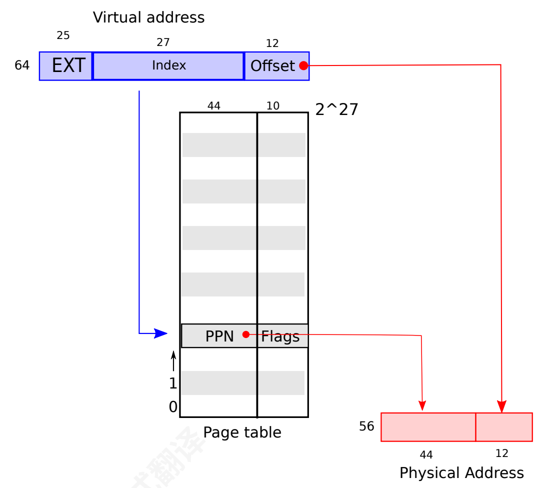

Figure 3.1: RISC-V virtual and physical addresses, with a simplified logical page table.

on RISC-V computers. 2^{56} is enough physical memory space for the near future to fit many I/O devices and RAM chips. If more is needed, the RISC-V designers have defined Sv48 with 48-bit virtual addresses [3].

As Figure 3.2 shows, a RISC-V CPU translates a virtual address into a physical in three steps. A page table is stored in physical memory as a three-level tree. The root of the tree is a 4096-byte page-table page that contains 512 PTEs, which contain the physical addresses for page-table pages in the next level of the tree. Each of those pages contains 512 PTEs for the final level in the tree. The paging hardware uses the top 9 bits of the 27 bits to select a PTE in the root page-table page, the middle 9 bits to select a PTE in a page-table page in the next level of the tree, and the bottom 9 bits to select the final PTE. (In Sv48 RISC-V a page table has four levels, and bits 39 through 47 of a virtual address index into the top-level.)

If any of the three PTEs required to translate an address is not present, the paging hardware raises a *page-fault exception*, leaving it up to the kernel to handle the exception (see Chapter 4).

The three-level structure of Figure 3.2 allows a memory-efficient way of recording PTEs, compared to the single-level design of Figure 3.1. In the common case in which large ranges of virtual addresses have no mappings, the three-level structure can omit entire page directories. For example, if an application uses only a few pages starting at address zero, then the entries 1 through 511 of the top-level page directory are invalid, and the kernel doesn't have to allocate pages those for 511 intermediate page directories. Furthermore, the kernel also doesn't have to allocate pages for the bottom-level page directories for those 511 intermediate page directories. So, in this example, the three-level design saves 511 pages for intermediate page directories and 511×512 pages for bottom-level page directories.

Although a CPU walks the three-level structure in hardware as part of executing a load or store instruction, a potential downside of three levels is that the CPU must load three PTEs from memory to perform the translation of the virtual address in the load/store instruction to a physical address. To avoid the cost of loading PTEs from physical memory, a RISC-V CPU caches page table entries in a *Translation Look-aside Buffer (TLB)*.

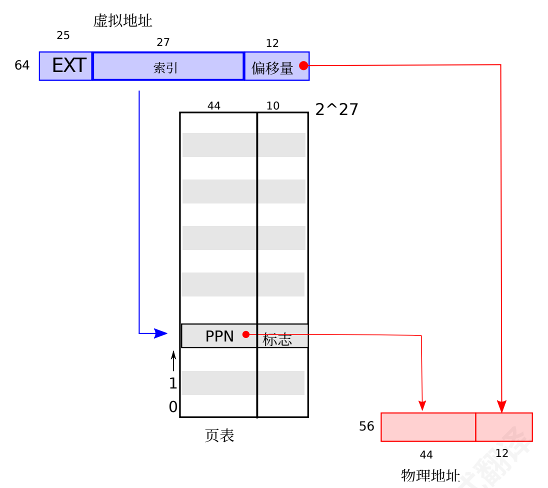

图3.1：RISC-V虚拟和物理地址，以及简化的逻辑页表。

在RISC-V计算机上。 2^{56} 足够近期的物理内存空间来容纳许多I/O设备和RAM芯片。如果需要更多，RISC-V的设计者已经定义了Sv48，使用48位虚拟地址 [3]。

如图3.2所示，RISC-V CPU将虚拟地址转换为物理地址需要三个步骤。页表存储在物理内存中，作为三级树结构。树的根是一个包含512个PTE的4096字节页表页，这些PTE包含下一级树中页表页的物理地址。这些页中的每一页都包含512个用于最终级树的PTE。分页硬件使用27位中的最高9位来选择根页表页中的PTE，中间9位来选择下一级树中页表页的PTE，以及最低9位来选择最终的PTE。（在Sv48 RISC-V中，页表有四级，虚拟地址的位39到47索引到顶层。）

如果用于转换地址的三个PTE中任何一个不存在，分页硬件会引发页错误异常，由内核处理该异常（见第4章）。

图3.2的三级结构允许以内存高效的方式记录PTE，与图3.1的单级设计相比。在常见情况下，如果大量虚拟地址没有映射，三级结构可以省略整个页目录。例如，如果应用程序仅使用从地址零开始的几页，那么顶层页目录的1到511条目无效，内核不必为这511个中间页目录分配页。此外，内核也不必为这511个中间页目录的底层页目录分配页。因此，在这个例子中，三级设计节省了511个中间页目录的页和 511×512 个底层页目录的页。

虽然 CPU 在执行加载或存储指令时，会通过硬件遍历三级结构，但三级结构的一个潜在缺点是，CPU 必须从内存加载三个 PTE 以执行加载/存储指令中的虚拟地址到物理地址的转换。为了避免从物理内存加载 PTE 的成本，RISC-V CPU 会将页表项缓存到转换旁路缓冲器（TLB）中。

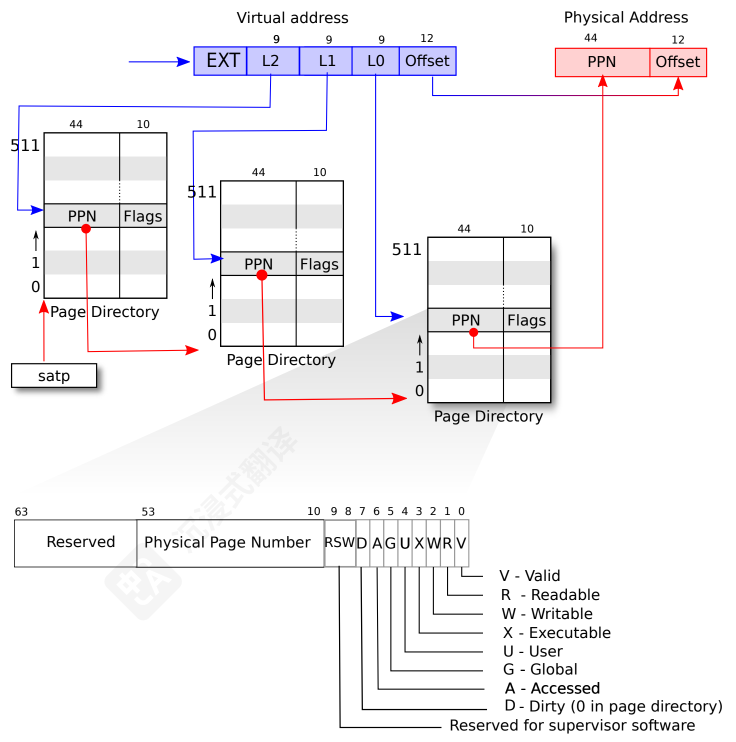

Figure 3.2: RISC-V address translation details.

Each PTE contains flag bits that tell the paging hardware how the associated virtual address is allowed to be used. `PTE_V` indicates whether the PTE is present: if it is not set, a reference to the page causes an exception (i.e., is not allowed). `PTE_R` controls whether instructions are allowed to read to the page. `PTE_W` controls whether instructions are allowed to write to the page. `PTE_X` controls whether the CPU may interpret the content of the page as instructions and execute them. `PTE_U` controls whether instructions in user mode are allowed to access the page; if `PTE_U` is not set, the PTE can be used only in supervisor mode. Figure 3.2 shows how it all works. The flags and all other page hardware-related structures are defined in (kernel/riscv.h)

To tell a CPU to use a page table, the kernel must write the physical address of the root page-table page into the `satp` register. A CPU will translate all addresses generated by subsequent instructions using the page table pointed to by its own `satp`. Each CPU has its own `satp` so that different CPUs can run different processes, each with a private address space described by its own page table.

From the kernel's point of view, a page table is data stored in memory, and the kernel creates and modifies page tables using code much like you might see for any tree-shaped data structure.

A few notes about terms used in this book. *Physical memory* refers to storage cells in RAM. A byte of physical memory has an address, called a *physical address*. Instructions that dereference addresses (such as loads, stores, jumps, and function calls) use only virtual addresses, which the paging hardware translates to physical addresses, and then sends to the RAM hardware to read or write storage. An *address space* is the set of virtual addresses that are valid in a given page table;

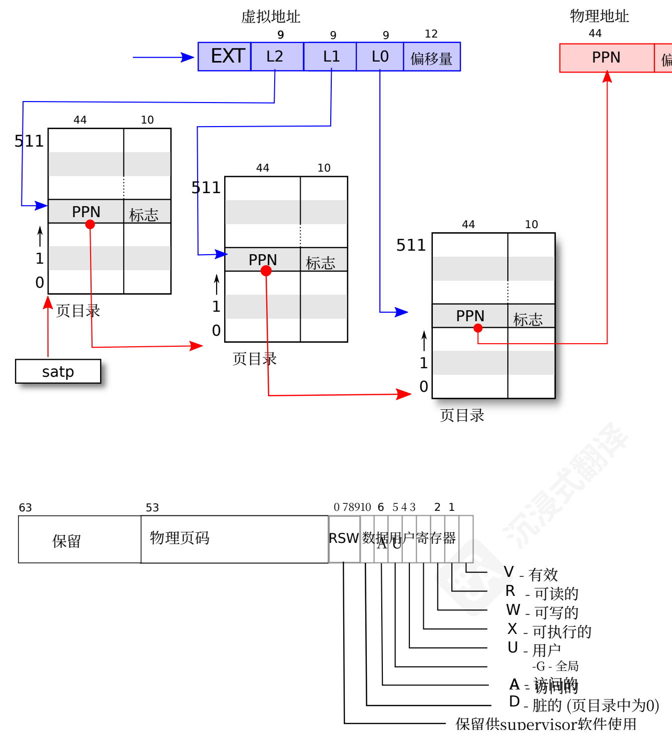

图3.2：RISC-V地址转换细节。

每个 PTE 包含标志位，这些标志位会告诉分页硬件关联的虚拟地址如何被使用。PTE_V 指示 PTE 是否存在：如果未设置，对页面的引用会导致异常（即不允许）。PTE_R 控制是否允许指令读取页面。PTE_W 控制是否允许指令写入页面。PTE_X 控制是否允许 CPU 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 PTE_U 未设置，PTE 只能在监督模式下使用。图 3.2 展示了其工作原理。标志位和所有其他与页硬件相关的结构定义在 (kernel/riscv.h) 中。

要让CPU使用页表，内核必须将根页表页的物理地址写入satp寄存器。CPU会使用其自己的satp指向的页表来转换后续指令生成的所有地址。每个CPU都有自己的satp，以便不同的CPU可以运行不同的进程，每个进程都有其自己的地址空间，该地址空间由其自己的页表描述。

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

本书中使用的术语的一些说明。物理内存是指RAM中的存储单元。物理内存的一个字节有一个地址，称为物理地址。那些对地址进行解引用的指令（如加载、存储、跳转和函数调用）只使用虚拟地址，分页硬件将这些虚拟地址转换为物理地址，然后发送给RAM硬件以读取或写入存储。地址空间是在给定的页表中有效的虚拟地址集；

each xv6 process has a separate user address space, and the xv6 kernel has its own address space as well. *User memory* refers to a process's user address space plus the physical memory that the page table allows the process to access. *Virtual memory* refers to the ideas and techniques associated with managing page tables and using them to achieve goals such as isolation.

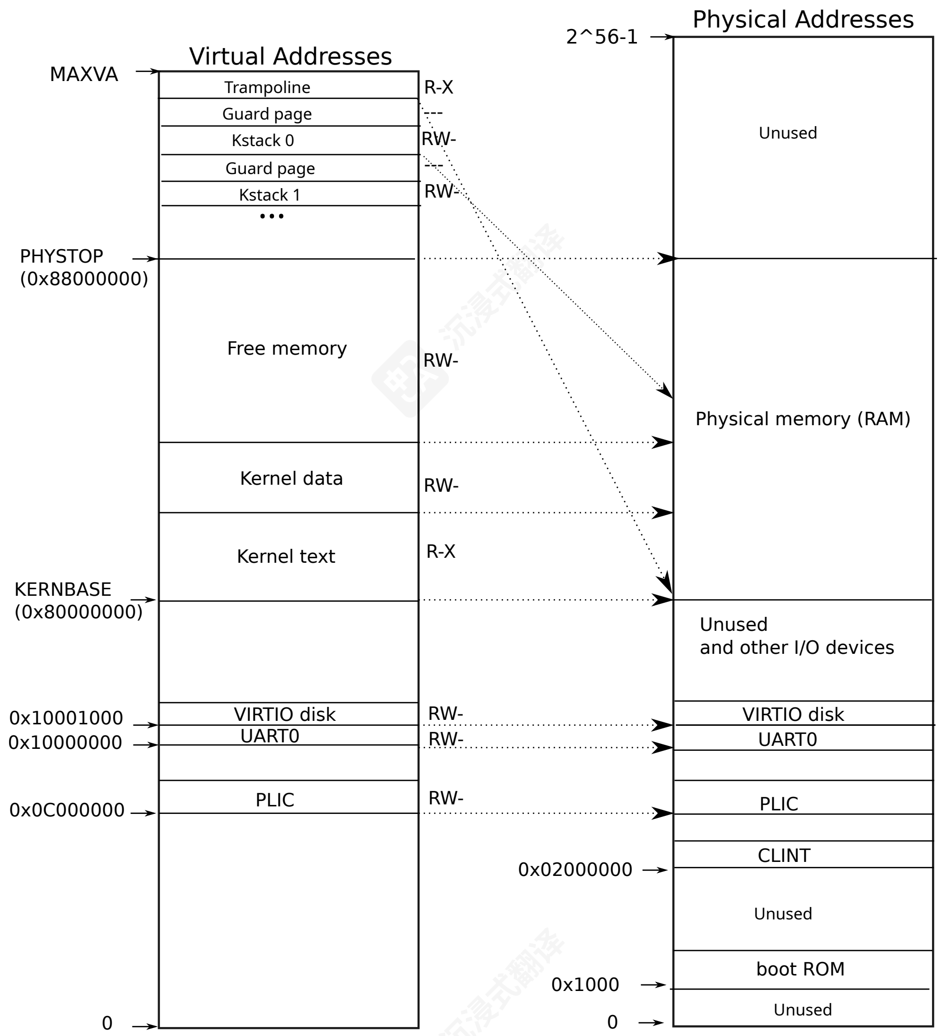

Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

3.2 Kernel address space

Xv6 maintains one page table per process, describing each process's user address space, plus a single page table that describes the kernel's address space. The kernel configures the layout of its address space to give itself access to physical memory and various hardware resources at predictable

每个xv6进程都有自己的用户地址空间，而xv6内核也有自己的地址空间。用户内存是指进程的用户地址空间加上页表允许进程访问的物理内存。虚拟内存是指与管理和使用页表相关联的思想和技术，以及使用它们来实现隔离等目标。

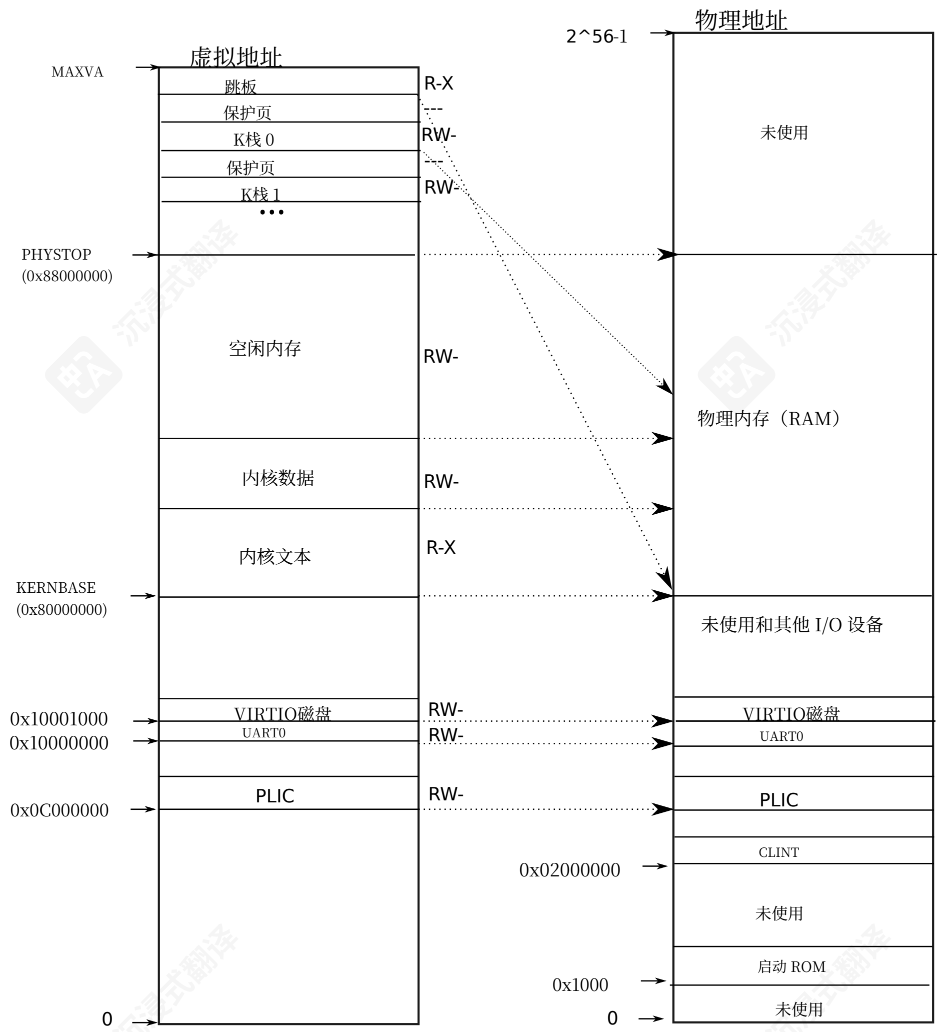

图3.3：左侧是xv6的内核地址空间。RWX指的是PTE读、写和执行权限。右侧是xv6期望看到的RISC-V物理地址空间。

3.2 内核地址空间

Xv6 为每个进程维护一个页表，描述每个进程的用户地址空间，此外还有一个页表描述内核的地址空间。内核配置其地址空间的布局，以使其能够以可预测的方式访问物理内存和各种硬件资源。

virtual addresses. Figure 3.3 shows how this layout maps kernel virtual addresses to physical addresses. The file (kernel/memlayout.h) declares the constants for xv6's kernel memory layout.

QEMU simulates a computer that includes RAM (physical memory) starting at physical address `0x80000000` and continuing through at least `0x88000000`, which xv6 calls `PHYSTOP`. The QEMU simulation also includes I/O devices such as a disk interface. QEMU exposes the device interfaces to software as *memory-mapped* control registers that sit below `0x80000000` in the physical address space. The kernel can interact with the devices by reading/writing these special physical addresses; such reads and writes communicate with the device hardware rather than with RAM. Chapter 4 explains how xv6 interacts with devices.

The kernel gets at RAM and memory-mapped device registers using "direct mapping;" that is, mapping the resources at virtual addresses that are equal to the physical address. For example, the kernel itself is located at `KERNBASE=0x80000000` in both the virtual address space and in physical memory. Direct mapping simplifies kernel code that reads or writes physical memory. For example, when `fork` allocates user memory for the child process, the allocator returns the physical address of that memory; `fork` uses that address directly as a virtual address when it is copying the parent's user memory to the child.

There are a couple of kernel virtual addresses that aren't direct-mapped:

- The trampoline page. It is mapped at the top of the virtual address space; user page tables have this same mapping. Chapter 4 discusses the role of the trampoline page, but we see here an interesting use case of page tables; a physical page (holding the trampoline code) is mapped twice in the virtual address space of the kernel: once at top of the virtual address space and once with a direct mapping.

- The kernel stack pages. Each process has its own kernel stack, which is mapped high so that below it xv6 can leave an unmapped *guard page*. The guard page's PTE is invalid (i.e., `PTE_V` is not set), so that if the kernel overflows a kernel stack, it will likely cause an exception and the kernel will panic. Without a guard page an overflowing stack would overwrite other kernel memory, resulting in incorrect operation. A panic crash is preferable.

While the kernel uses its stacks via the high-memory mappings, they are also accessible to the kernel through a direct-mapped address. An alternate design might have just the direct mapping, and use the stacks at the direct-mapped address. In that arrangement, however, providing guard pages would involve unmapping virtual addresses that would otherwise refer to physical memory, which would then be hard to use.

The kernel maps the pages for the trampoline and the kernel text with the permissions `PTE_R` and `PTE_X`. The kernel reads and executes instructions from these pages. The kernel maps the other pages with the permissions `PTE_R` and `PTE_W`, so that it can read and write the memory in those pages. The mappings for the guard pages are invalid.

3.3 Code: creating an address space

Most of the xv6 code for manipulating address spaces and page tables resides in `vm.c` (kernel/vm.c:1). The central data structure is `pagetable_t`, which is really a pointer to a RISC-V

虚拟地址。图 3.3 显示了这种布局如何将内核虚拟地址映射到物理地址。文件 (kernel/memlayout.h) 声明了用于 xv6 内核内存布局的常量。

QEMU 模拟一台包含 RAM（物理内存）的计算机，从物理地址 0x80000000 开始，并至少持续到 0x88000000，xv6 称之为 PHYSTOP。QEMU 模拟还包括磁盘接口等 I/O 设备。QEMU 将设备接口作为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/写入这些特殊物理地址与设备交互；这些读取和写入与设备硬件通信，而不是与 RAM 通信。第 4 章解释了 xv6 如何与设备交互。

内核通过"直接映射"方式访问 RAM 和内存映射设备寄存器；也就是说，将资源映射到虚拟地址等于物理地址的位置。例如，内核本身位于虚拟地址空间和物理内存中的 KERNBASE=0x80000000。直接映射简化了内核代码中对物理内存的读取或写入。例如，当 fork 为子进程分配用户内存时，分配器返回该内存的物理地址；fork 在将父进程的用户内存复制到子进程时，直接将该地址用作虚拟地址。

存在一些没有直接映射的内核虚拟地址：

• 跳板页。它在虚拟地址空间的顶部进行映射；用户页表也有相同的映射。第4章讨论了跳板页的作用，但这里我们看到一个有趣的页表用例；一个物理页（包含跳板代码）在内核的虚拟地址空间中映射了两次：一次在虚拟地址空间的顶部，一次通过直接映射。

• 内核栈页。每个进程都有自己的内核栈，它被映射到较高的位置，以便在它下面 xv6 可以留出一个未映射的保护页。保护页的 PTE 是无效的（即 PTE_V 没有被设置），所以如果内核栈溢出，它很可能会引发异常，内核会恐慌。如果没有保护页，溢出的栈会覆盖其他内核内存，导致操作不正确。恐慌崩溃是更可取的。

虽然内核通过高内存映射使用它的栈，但它们也可以通过直接映射的地址被内核访问。另一种设计可能只有直接映射，并在直接映射的地址使用栈。然而，在这种安排下，提供保护页将涉及取消映射原本会引用物理内存的虚拟地址，这会变得难以使用。

内核使用权限 PTE_R 和 PTE_X 映射跳板和内核文本的页。内核从这些页读取并执行指令。内核使用权限 PTE_R 和 PTE_W 映射其他页，以便它能够读取和写入这些页中的内存。保护页的映射是无效的。

3.3 代码：创建地址空间

大部分用于操作地址空间和页表的 xv6 代码位于 vm.c (内核/vm.c:1) 中。核心数据结构是 pagetable_t，它实际上是一个指向 RISC-V 的指针

root page-table page; a pagetable_t may be either the kernel page table, or one of the per-process page tables. The central functions are walk, which finds the PTE for a virtual address, and mappages, which installs PTEs for new mappings. Functions starting with kvm manipulate the kernel page table; functions starting with uvm manipulate a user page table; other functions are used for both. copyout and copyin copy data to and from user virtual addresses provided as system call arguments; they are in vm.c because they need to explicitly translate those addresses in order to find the corresponding physical memory.

Early in the boot sequence, main calls kvminit (kernel/vm.c:54) to create the kernel's page table using kvmmake (kernel/vm.c:20). This call occurs before xv6 has enabled paging on the RISC-V, so addresses refer directly to physical memory. kvmmake first allocates a page of physical memory to hold the root page-table page. Then it calls kvmmap to install the translations that the kernel needs. The translations include the kernel's instructions and data, physical memory up to PHYSTOP, and memory ranges which are actually devices. proc_mapstacks (kernel/proc.c:33) allocates a kernel stack for each process. It calls kvmmap to map each stack at the virtual address generated by KSTACK, which leaves room for the invalid stack-guard pages.

kvmmap (kernel/vm.c:132) calls mappages (kernel/vm.c:144), which installs mappings into a page table for a range of virtual addresses to a corresponding range of physical addresses. It does this separately for each virtual address in the range, at page intervals. For each virtual address to be mapped, mappages calls walk to find the address of the PTE for that address. It then initializes the PTE to hold the relevant physical page number, the desired permissions (PTE_W, PTE_X, and/or PTE_R), and PTE_V to mark the PTE as valid (kernel/vm.c:165).

walk (kernel/vm.c:86) mimics the RISC-V paging hardware as it looks up the PTE for a virtual address (see Figure 3.2). walk descends the page table one level at a time, using each level's 9 bits of virtual address to index into the relevant page directory page. At each level it finds either the PTE of the next level's page directory page, or the PTE of final page (kernel/vm.c:92). If a PTE in a first or second level page directory page isn't valid, then the required directory page hasn't yet been allocated; if the alloc argument is set, walk allocates a new page-table page and puts its physical address in the PTE. It returns the address of the PTE in the lowest layer in the tree (kernel/vm.c:102).

The above code depends on physical memory being direct-mapped into the kernel virtual address space. For example, as walk descends levels of the page table, it pulls the (physical) address of the next-level-down page table from a PTE (kernel/vm.c:94), and then uses that address as a virtual address to fetch the PTE at the next level down (kernel/vm.c:92).

main calls kvminithart (kernel/vm.c:62) to install the kernel page table. It writes the physical address of the root page-table page into the register satp. After this the CPU will translate addresses using the kernel page table. Since the kernel uses a direct mapping, the now virtual address of the next instruction will map to the right physical memory address.

Each RISC-V CPU caches page table entries in a *Translation Look-aside Buffer (TLB)*, and when xv6 changes a page table, it must tell the CPU to invalidate corresponding cached TLB entries. If it didn't, then at some point later the TLB might use an old cached mapping, pointing to a physical page that in the meantime has been allocated to another process, and as a result, a process might be able to scribble on some other process's memory. The RISC-V has an

根页表页；一个 pagetable_t 可能是内核页表，或者是每个进程的页表之一。核心函数是 walk，用于查找虚拟地址的 PTE，以及 mappages，用于为新映射安装 PTE。以 kvm 开头的函数操作内核页表；以 uvm 开头的函数操作用户页表；其他函数用于两者。copyout 和 copyin 将数据复制到和从用户虚拟地址，这些地址作为系统调用参数提供；它们位于 vm.c 中，因为它们需要显式转换这些地址以找到相应的物理内存。

在启动序列的早期，main 调用 kvminit (kernel/vm.c:54) 使用 kvmmake (kernel/vm.c:20) 创建内核的页表。此调用发生在 xv6 在 RISC-V 上启用分页之前，因此地址直接指向物理内存。kvmmake 首先分配一页物理内存来存放根页表页。然后它调用 kvmmap 来安装内核需要的转换。这些转换包括内核的指令和数据、直到 PHYSTOP 的物理内存，以及实际上是设备的内存范围。proc_mapstacks (kernel/proc.c:33) 为每个进程分配一个内核栈。它调用 kvmmap 将每个栈映射到由 KSTACK 生成的虚拟地址，这为无效的栈保护页留出了空间。

kvmmap (kernel/vm.c:132) 调用 mappages (kernel/vm.c:144)，后者将映射安装到页表中，将一系列虚拟地址的范围映射到相应的物理地址范围。它分别针对范围内的每个虚拟地址，以页为单位进行此操作。对于要映射的每个虚拟地址，mappages 调用 walk 来查找该地址的 PTE 地址。然后它将 PTE 初始化为包含相关的物理页号、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:165)。

walk (内核/vm.c:86)模拟了 RISC-V 分页硬件，因为它查找虚拟地址的 PTE（参见图 3.2）。walk 按照页表逐级遍历，使用每一级的 9 位虚拟地址索引到相关的页目录页。在每一级，它找到下一级页目录页的 PTE，或者最终页的 PTE（内核/vm.c:92）。如果一个第一级或第二级页目录页中的 PTE 无效，那么所需的目录页尚未分配；如果 alloc 参数被设置，walk 分配一个新的页表页，并将它的物理地址放入 PTE。它返回树中最低层的 PTE 地址（内核/vm.c:102）。

上述代码依赖于物理内存直接映射到内核虚拟地址空间。例如，当 walk 遍历页表的级时，它从 PTE（内核/vm.c:94）中拉取下一级页表的（物理）地址，然后使用该地址作为虚拟地址来获取下一级的 PTE（内核/vm.c:92）。

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

每个 RISC-V CPU 都在转换旁路缓冲区（TLB）中缓存页表条目，当 xv6 更改页表时，它必须告诉 CPU 使相应的缓存的 TLB 条目无效。如果它没有这样做，那么在稍后的某个时刻，TLB 可能会使用一个旧的缓存映射，指向一个在此期间已被分配给另一个进程的物理页，其结果是一个进程可能会在另一个进程的内存上涂鸦。RISC-V 有一个

instruction `sfence.vma` that flushes the current CPU's TLB. Xv6 executes `sfence.vma` in `kvminithart` after reloading the `satp` register, and in the trampoline code that switches to a user page table before returning to user space (kernel/trampoline.S:89).

It is also necessary to issue `sfence.vma` before changing `satp`, in order to wait for completion of all outstanding loads and stores. This wait ensures that preceding updates to the page table have completed, and ensures that preceding loads and stores use the old page table, not the new one.

To avoid flushing the complete TLB, RISC-V CPUs may support address space identifiers (ASIDs) [3]. The kernel can then flush just the TLB entries for a particular address space. Xv6 does not use this feature.

3.4 Physical memory allocation

The kernel must allocate and free physical memory at run-time for page tables, user memory, kernel stacks, and pipe buffers.

Xv6 uses the physical memory between the end of the kernel and `PHYSTOP` for run-time allocation. It allocates and frees whole 4096-byte pages at a time. It keeps track of which pages are free by threading a linked list through the pages themselves. Allocation consists of removing a page from the linked list; freeing consists of adding the freed page to the list.

3.5 Code: Physical memory allocator

The allocator resides in `kalloc.c` (kernel/kalloc.c:1). The allocator's data structure is a *free list* of physical memory pages that are available for allocation. Each free page's list element is a `struct run` (kernel/kalloc.c:17). Where does the allocator get the memory to hold that data structure? It store each free page's `run` structure in the free page itself, since there's nothing else stored there. The free list is protected by a spin lock (kernel/kalloc.c:21-24). The list and the lock are wrapped in a struct to make clear that the lock protects the fields in the struct. For now, ignore the lock and the calls to `acquire` and `release`; Chapter 6 will examine locking in detail.

The function `main` calls `kinit` to initialize the allocator (kernel/kalloc.c:27). `kinit` initializes the free list to hold every page between the end of the kernel and `PHYSTOP`. Xv6 ought to determine how much physical memory is available by parsing configuration information provided by the hardware. Instead xv6 assumes that the machine has 128 megabytes of RAM. `kinit` calls `freerange` to add memory to the free list via per-page calls to `kfree`. A PTE can only refer to a physical address that is aligned on a 4096-byte boundary (is a multiple of 4096), so `freerange` uses `PGROUNDUP` to ensure that it frees only aligned physical addresses. The allocator starts with no memory; these calls to `kfree` give it some to manage.

The allocator sometimes treats addresses as integers in order to perform arithmetic on them (e.g., traversing all pages in `freerange`), and sometimes uses addresses as pointers to read and write memory (e.g., manipulating the `run` structure stored in each page); this dual use of addresses is the main reason that the allocator code is full of C type casts.

指令sfence.vma用于刷新当前CPU的TLB。Xv6在重新加载satp寄存器后，在kvminithart中执行sfence.vma，以及在切换到用户页表返回用户空间之前（kernel/trampoline.S:89）的trampoline code中执行。

在更改 satp 之前，也需要发出 sfence.vma，以便等待所有未完成的加载和存储操作完成。这种等待确保了页表的前序更新已经完成，并确保前序的加载和存储使用旧的页表，而不是新的页表。

为了避免刷新整个 TLB，RISC-V CPU 可能支持地址空间标识符（ASIDs） [3]。然后内核可以仅刷新特定地址空间的 TLB 条目。Xv6 不使用此功能。

3.4 物理内存分配

内核必须在运行时为页表、用户内存、内核栈和管道缓冲区分配和释放物理内存。

Xv6 使用内核结束和 PHYSTOP 之间的物理内存进行运行时分配。它一次分配和释放整个 4096-byte 页面。它通过在页面本身中穿过一个链表来跟踪哪些页面是空闲的。分配包括从链表中移除一个页面；释放包括将释放的页面添加到列表中。

3.5 代码：物理内存分配器

分配器位于 kalloc.c (kernel/kalloc.c:1) 中。分配器的数据结构是一个空闲的物理内存页列表，这些页面可供分配。每个空闲页面的列表元素是一个 struct run (kernel/kalloc.c:17)。分配器从哪里获取存储该数据结构的内存？它将每个空闲页面的运行结构存储在空闲页面本身中，因为那里没有其他内容存储。空闲列表由一个 spin lock (kernel/kalloc.c:21-24) 保护。列表和锁被包装在一个 struct 中，以明确表示锁保护 struct 中的字段。目前，忽略锁以及获取和释放的调用；第 6 章将详细探讨锁定。

函数 main 调用 kinit 来初始化分配器 (kernel/kalloc.c:27)。kinit 初始化空闲列表，以包含内核结束和 PHYSTOP 之间的所有页面。Xv6 应该通过解析硬件提供的配置信息来确定可用的物理内存量。相反，xv6 假设机器有 128 兆字节的 RAM。kinit 调用 freerange，通过每页调用 kfree 来向空闲列表添加内存。PTE 只能引用一个在 4096 字节边界上对齐的物理地址（是 4096 的倍数），因此 freerange 使用 PGROUNDUP 来确保它只释放对齐的物理地址。分配器最初没有内存；这些调用 kfree 为它提供了一些可以管理的内存。

分配器有时将地址视为整数，以便对它们执行算术运算（例如，遍历空闲范围内的所有页），有时则将地址用作指针来读取和写入内存（例如，操作存储在每个页中的运行结构）；地址的这种双重用途是分配器代码充满C类型转换的主要原因。

The function kfree (kernel/kalloc.c:47) begins by setting every byte in the memory being freed to the value 1. This will cause code that uses memory after freeing it (uses "dangling references") to read garbage instead of the old valid contents; hopefully that will cause such code to break faster. Then kfree prepends the page to the free list: it casts pa to a pointer to struct run, records the old start of the free list in r->next, and sets the free list equal to r. kalloc removes and returns the first element in the free list.

3.6 Process address space

Each process has its own page table, and when xv6 switches between processes, it also changes page tables. Figure 3.4 shows a process's address space in more detail than Figure 2.3. A process's user memory starts at virtual address zero and can grow up to MAXVA (kernel/riscv.h:375), allowing a process to address in principle 256 Gigabytes of memory.

A process's address space consists of pages that contain the text of the program (which xv6 maps with the permissions PTE_R, PTE_X, and PTE_U), pages that contain the pre-initialized data of the program, a page for the stack, and pages for the heap. Xv6 maps the data, stack, and heap with the permissions PTE_R, PTE_W, and PTE_U.

Using permissions within a user address space is a common technique to harden a user process. If the text were mapped with PTE_W, then a process could accidentally modify its own program; for example, a programming error may cause the program to write to a null pointer, modifying instructions at address 0, and then continue running, perhaps creating more havoc. To detect such errors immediately, xv6 maps the text without PTE_W; if a program accidentally attempts to store to address 0, the hardware will refuse to execute the store and raises a page fault (see Section 4.6). The kernel then kills the process and prints out an informative message so that the developer can track down the problem.

Similarly, by mapping data without PTE_X, a user program cannot accidentally jump to an address in the program's data and start executing at that address.

In the real world, hardening a process by setting permissions carefully also aids in defending against security attacks. An adversary may feed carefully-constructed input to a program (e.g., a Web server) that triggers a bug in the program in the hope of turning that bug into an exploit [14]. Setting permissions carefully and other techniques, such as randomizing of the layout of the user address space, make such attacks harder.

The stack is a single page, and is shown with the initial contents as created by exec. Strings containing the command-line arguments, as well as an array of pointers to them, are at the very top of the stack. Just under that are values that allow a program to start at main as if the function main(argc, argv) had just been called.

To detect a user stack overflowing the allocated stack memory, xv6 places an inaccessible guard page right below the stack by clearing the PTE_U flag. If the user stack overflows and the process tries to use an address below the stack, the hardware will generate a page-fault exception because the guard page is inaccessible to a program running in user mode. A real-world operating system might instead automatically allocate more memory for the user stack when it overflows.

When a process asks xv6 for more user memory, xv6 grows the process's heap. Xv6 first uses

函数kfree（kernel/kalloc.c:47）首先将正在释放的内存中的每个字节设置为值1。这将导致在释放内存后使用内存的代码（使用“悬空引用”）读取垃圾数据而不是旧的有效内容；希望这能让这类代码更快地崩溃。然后kfree将页添加到空闲列表的前面：它将pa转换为指向struct run的指针，将空闲列表的旧起始地址记录在r->next中，并将空闲列表设置为r。kalloc移除并返回空闲列表中的第一个元素。

3.6 进程地址空间

每个进程都有自己的页表，当 xv6 在进程之间切换时，它也会切换页表。图3.4比图2.3更详细地显示了进程的地址空间。一个进程的用户内存从虚拟地址零开始，可以增长到MAXVA（kernel/riscv.h:375），原则上允许进程寻址256吉字节的内存。

一个进程的地址空间由包含程序文本的页（xv6 使用权限 PTE_R、PTE_X 和 PTE_U 映射这些页）、包含程序预初始化数据的页、一个栈页以及堆页组成。xv6 使用权限 PTE_R、PTE_W 和 PTE_U 映射数据、栈和堆。

在用户地址空间内使用权限是一种常见的加固用户进程的技术。如果文本映射为PTE_W，那么进程可能会意外地修改自己的程序；例如，编程错误可能导致程序写入空指针，修改地址0处的指令，然后继续运行，可能会造成更多混乱。为了立即检测此类错误，xv6 不使用 PTE_W 映射文本；如果程序意外尝试存储到地址0，硬件将拒绝执行存储并引发页错误（见第4.6节）。内核然后杀死进程并打印出一条信息性消息，以便开发者可以追踪问题。

同样地，通过映射数据而不使用 PTE_X，用户程序不能意外跳转到程序数据中的地址并在该地址处开始执行。

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

栈是一个单页，并以由 exec 创建的初始内容显示。命令行参数的字符串以及指向它们的指针数组位于栈的顶部。紧挨着下面的是允许程序像函数 main(argc, argv) 刚刚被调用一样在 main 处开始的值。

为了检测用户栈是否溢出分配的栈内存，xv6 通过清除 PTE_U 标志在栈正下方放置一个不可访问的保护页。如果用户栈溢出并且进程尝试使用栈下方的地址，硬件将生成一个页错误异常，因为保护页对在用户模式下运行的程序不可访问。实际操作系统在栈溢出时可能会自动为用户栈分配更多内存。

当进程向 xv6 申请更多用户内存时，xv6 会扩展进程的堆。xv6 首先使用

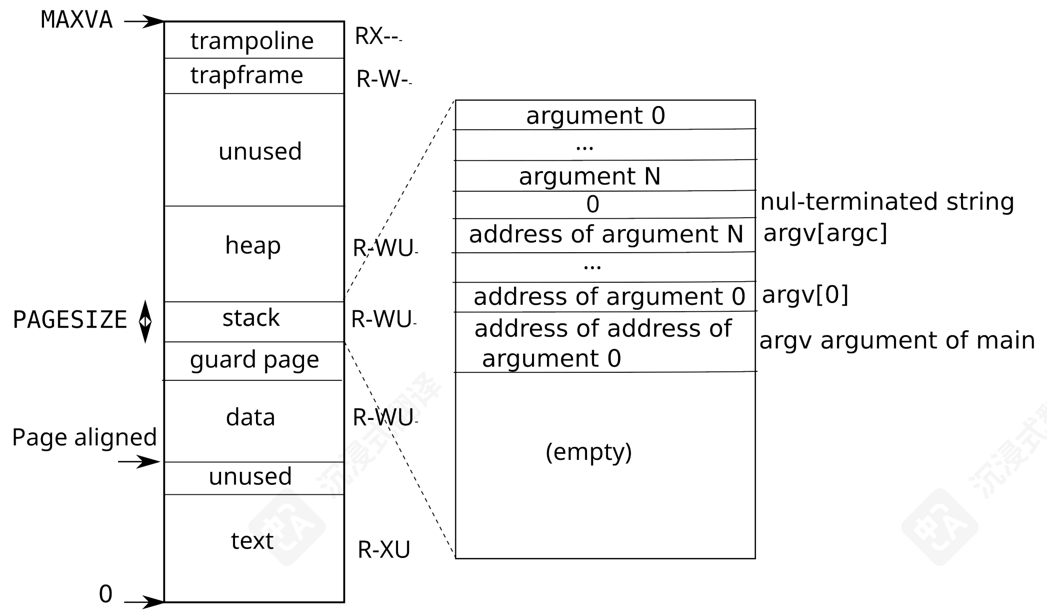

Figure 3.4: A process's user address space, with its initial stack.

kalloc to allocate physical pages. It then adds PTEs to the process's page table that point to the new physical pages. Xv6 sets the PTE_W, PTE_R, PTE_U, and PTE_V flags in these PTEs. Most processes do not use the entire user address space; xv6 leaves PTE_V clear in unused PTEs.

We see here a few nice examples of use of page tables. First, different processes' page tables translate user addresses to different pages of physical memory, so that each process has private user memory. Second, each process sees its memory as having contiguous virtual addresses starting at zero, while the process's physical memory can be non-contiguous. Third, the kernel maps a page with trampoline code at the top of the user address space (without PTE_U), thus a single page of physical memory shows up in all address spaces, but can be used only by the kernel.

3.7 Code: sbrk

sbrk is the system call for a process to shrink or grow its memory. The system call is implemented by the function growproc (kernel/proc.c:260). growproc calls uvmalloc or uvmdealloc, depending on whether n is positive or negative. uvmalloc (kernel/vm.c:233) allocates physical memory with kalloc, zeros the allocated memory, and adds PTEs to the user page table with mappages. uvmdealloc calls uvmunmap (kernel/vm.c:178), which uses walk to find PTEs and kfree to free the physical memory they refer to.

Xv6 uses a process's page table not just to tell the hardware how to map user virtual addresses,

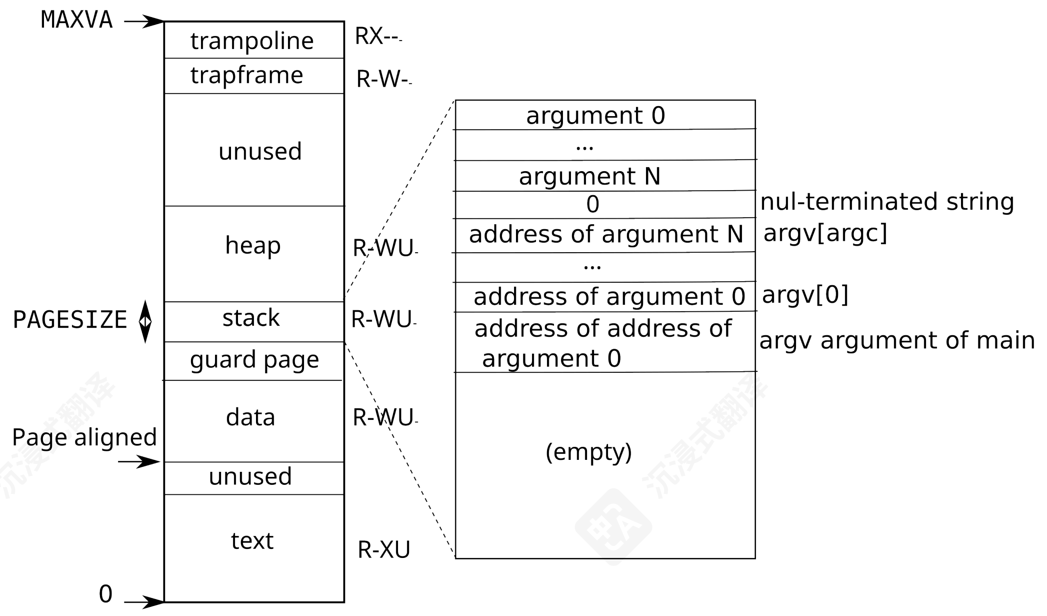

图 3.4：进程的用户地址空间，及其初始栈。

kalloc 来分配物理页。然后向进程的页表添加 PTE，这些 PTE 指向新的物理页。xv6 在这些 PTE 中设置 PTE_W、PTE_R、PTE_U 和 PTE_V 标志。大多数进程不会使用整个用户地址空间；xv6 在未使用的 PTE 中将 PTE_V 清除。

我们看到这里有几个使用页表的优秀示例。首先，不同的进程的页表将用户地址转换为不同的物理内存页，以便每个进程都有私有用户内存。其次，每个进程都将其内存视为从零开始的连续虚拟地址，而进程的物理内存可以是非连续的。第三，内核在用户地址空间顶部映射了一个带有 trampoline 代码的页（没有 PTE_U），因此一个物理内存页在所有地址空间中都会出现，但只能由内核使用。

3.7 代码：sbrk

sbrk 是进程缩小或增长其内存的系统调用。该系统调用由函数 growproc (kernel/proc.c:260) 实现。growproc 调用 uvmalloc 或 uvmdealloc，取决于 n 是正数还是负数。uvmalloc (kernel/vm.c:233) 使用 kalloc 分配物理内存，将分配的内存清零，并使用 mappages 向用户页表添加 PTE。uvmdealloc 调用 uvmunmap (kernel/vm.c:178)，它使用 walk 查找 PTE 并使用 kfree 释放它们所引用的物理内存。

Xv6 使用进程的页表不仅告诉硬件如何映射用户虚拟地址，

but also as the only record of which physical memory pages are allocated to that process. That is the reason why freeing user memory (in `uvmunmap`) requires examination of the user page table.

3.8 Code: exec

`exec` is a system call that replaces a process's user address space with data read from a file, called a binary or executable file. A binary is typically the output of the compiler and linker, and holds machine instructions and program data. `exec` (kernel/exec.c:23) opens the named binary `path` using `namei` (kernel/exec.c:36), which is explained in Chapter 8. Then, it reads the ELF header. Xv6 binaries are formatted in the widely-used *ELF format*, defined in (kernel/elf.h). An ELF binary consists of an ELF header, `struct elfhdr` (kernel/elf.h:6), followed by a sequence of program section headers, `struct proghdr` (kernel/elf.h:25). Each `progvhdr` describes a section of the application that must be loaded into memory; xv6 programs have two program section headers: one for instructions and one for data.

The first step is a quick check that the file probably contains an ELF binary. An ELF binary starts with the four-byte "magic number" `0x7F`, `'E'`, `'L'`, `'F'`, or `ELF_MAGIC` (kernel/elf.h:3). If the ELF header has the right magic number, `exec` assumes that the binary is well-formed.

`exec` allocates a new page table with no user mappings with `proc_pagetable` (kernel/exec.c:49), allocates memory for each ELF segment with `uvmalloc` (kernel/exec.c:65), and loads each segment into memory with `loadseg` (kernel/exec.c:10). `loadseg` uses `walkaddr` to find the physical address of the allocated memory at which to write each page of the ELF segment, and `readi` to read from the file.

The program section header for `/init`, the first user program created with `exec`, looks like this:

```
# objdump -p user/_init

user/_init:     file format elf64-little

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

We see that the text segment should be loaded at virtual address 0x0 in memory (without write permissions) from content at offset 0x1000 in the file. We also see that the data should be loaded at address 0x1000, which is at a page boundary, and without executable permissions.

而且也是记录哪些物理内存页分配给该进程的唯一记录。这就是为什么释放用户内存（inuvmunmap）需要检查用户页表的原因。

3.8 代码：exec

exec 是一个系统调用，用从文件（称为二进制文件或可执行文件）读取的数据替换进程的用户地址空间。二进制文件通常是编译器和链接器的输出，包含机器指令和程序数据。exec (kernel/exec.c:23) 使用 namei (kernel/exec.c:36) 打开命名的二进制路径，这在第 8 章中解释。然后，它读取 ELF 头。Xv6 二进制文件采用广泛使用的 ELF 格式，定义在 (kernel/elf.h) 中。ELF 二进制文件由 ELF 头、struct elfhdr (kernel/elf.h:6) 以及一系列程序段头部、struct proghdr (kernel/elf.h:25) 组成。每个 progvhdr 描述了必须加载到内存中的应用程序部分；xv6 程序有两个程序段头部：一个用于指令，一个用于数据。

第一步是一个快速检查，以确定文件可能包含 ELF 二进制文件。ELF 二进制文件以四字节的"魔数" 0x7F、'E'、'L'、'F' 或 ELF_MAGIC (kernel/elf.h:3) 开头。如果 ELF 头具有正确的魔数，exec 假定该二进制文件格式良好。

exec 为 proc_pagetable (kernel/exec.c:49) 分配一个新的页表，其中没有用户映射，使用 uvmalloc 为每个 ELF 段分配内存 (kernel/exec.c:65)，并使用 loadseg (kernel/exec.c:10) 将每个段加载到内存中。loadseg 使用 walkaddr 找到分配内存的物理地址，用于写入 ELF 段的每一页，并使用 readi 从文件中读取。

/init 的程序段头，exec 创建的第一个用户程序，看起来像这样：

```
objdump -p user/_init

user/_init:     file format elf64-little

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

我们看到文本段应该被加载到内存中的虚拟地址 0x0（无写入权限）处，内容来自文件中偏移量 0x1000 的位置。我们还看到数据应该被加载到地址 0x1000 处，该地址位于页边界，且无可执行权限。

A program section header's `filesz` may be less than the `memsz`, indicating that the gap between them should be filled with zeroes (for C global variables) rather than read from the file. For `/init`, the data `filesz` is 0x10 bytes and `memsz` is 0x30 bytes, and thus `uvmalloc` allocates enough physical memory to hold 0x30 bytes, but reads only 0x10 bytes from the file `/init`.

Now `exec` allocates and initializes the user stack. It allocates just one stack page. `exec` copies the argument strings to the top of the stack one at a time, recording the pointers to them in `ustack`. It places a null pointer at the end of what will be the `argv` list passed to `main`. The values for `argc` and `argv` are passed to `main` through the system-call return path: `argc` is passed via the system call return value, which goes in `a0`, and `argv` is passed through the `a1` entry of the process's trapframe.

`exec` places an inaccessible page just below the stack page, so that programs that try to use more than one page will fault. This inaccessible page also allows `exec` to deal with arguments that are too large; in that situation, the `copyout` (kernel/vm.c:359) function that `exec` uses to copy arguments to the stack will notice that the destination page is not accessible, and will return -1.

During the preparation of the new memory image, if `exec` detects an error like an invalid program segment, it jumps to the label `bad`, frees the new image, and returns -1. `exec` must wait to free the old image until it is sure that the system call will succeed: if the old image is gone, the system call cannot return -1 to it. The only error cases in `exec` happen during the creation of the image. Once the image is complete, `exec` can commit to the new page table (kernel/exec.c:125) and free the old one (kernel/exec.c:129).

`exec` loads bytes from the ELF file into memory at addresses specified by the ELF file. Users or processes can place whatever addresses they want into an ELF file. Thus `exec` is risky, because the addresses in the ELF file may refer to the kernel, accidentally or on purpose. The consequences for an unwary kernel could range from a crash to a malicious subversion of the kernel's isolation mechanisms (i.e., a security exploit). Xv6 performs a number of checks to avoid these risks. For example `if(ph.vaddr + ph.memsz < ph.vaddr)` checks for whether the sum overflows a 64-bit integer. The danger is that a user could construct an ELF binary with a `ph.vaddr` that points to a user-chosen address, and `ph.memsz` large enough that the sum overflows to 0x1000, which will look like a valid value. In an older version of xv6 in which the user address space also contained the kernel (but not readable/writable in user mode), the user could choose an address that corresponded to kernel memory and would thus copy data from the ELF binary into the kernel. In the RISC-V version of xv6 this cannot happen, because the kernel has its own separate page table; `loadseg` loads into the process's page table, not in the kernel's page table.

It is easy for a kernel developer to omit a crucial check, and real-world kernels have a long history of missing checks whose absence can be exploited by user programs to obtain kernel privileges. It is likely that xv6 doesn't do a complete job of validating user-level data supplied to the kernel, which a malicious user program might be able to exploit to circumvent xv6's isolation.

3.9 Real world

Like most operating systems, xv6 uses the paging hardware for memory protection and mapping. Most operating systems make far more sophisticated use of paging than xv6 by combining paging

程序段头部的 filesz 可能小于 memsz，这表示它们之间的间隙应使用零填充（用于 C 全局变量），而不是从文件中读取。对于 /init，数据文件 filesz 是 0x10 字节，memsz 是 0x30 字节，因此 uvmalloc 分配足够的物理内存来容纳 0x30 字节，但仅从文件/init 中读取 0x10 字节。

现在 exec 分配并初始化用户栈。它只分配一个栈页。exec 逐个将参数字符串复制到栈顶，并将它们的指针记录在 ustack 中。它在即将传递给 main 的 argv 列表的末尾放置一个空指针。argc 和 argv 的值通过系统调用返回路径传递给 main：argc 通过系统调用返回值传递，该返回值存储在 a0 中，argv 通过进程的陷阱帧的 a1 条目传递。

exec 在栈页下方放置一个不可访问的页，以便尝试使用超过一页的程序发生故障。这个不可访问的页也允许 exec 处理过大的参数；在这种情况下，exec 用来将参数复制到栈中的 copyout (kernel/vm.c:359) 函数会注意到目标页不可访问，并返回 -1。

在新内存图像的准备过程中，如果 exec 检测到错误（如无效的程序段），它会跳转到标签 bad，释放新图像，并返回 -1。exec 必须等到确信系统调用会成功后才能释放旧图像：如果旧图像已经消失，系统调用无法向它返回 -1。exec 中的唯一错误情况发生在图像创建过程中。一旦图像完成，exec 可以提交新页表 (kernel/exec.c:125) 并释放旧页表 (kernel/exec.c:129)。

exec 将 ELF 文件中的字节加载到内存中，地址由 ELF 文件指定。用户或进程可以将他们想要的任何地址放入 ELF 文件中。因此 exec 很危险，因为 ELF 文件中的地址可能指向内核，可能是意外或故意。对于粗心的内核，后果可能从崩溃到恶意破坏内核的隔离机制（即，一个安全漏洞）。Xv6 执行了许多检查以避免这些风险。例如，如果 (ph.vaddr + ph.memsz < ph.vaddr) 检查总和是否溢出 64 位整数。危险在于用户可以构造一个 ELF 二进制文件，其中 ph.vaddr 指向用户选择的地址，并且 ph.memsz 足够大，使得总和溢出到 0x1000，这看起来像是一个有效值。在较旧版本的 xv6 中，用户地址空间也包含内核（但在用户模式下不可读/不可写），用户可以选择一个地址，该地址对应于内核内存，因此会将数据从 ELF 二进制文件复制到内核。在 RISC-V 版本的 xv6 中，这是不可能发生的，因为内核有自己的单独页表；loadseg 加载到进程的页表中，而不是内核的页表中。

内核开发者容易遗漏关键检查，而现实世界的内核长期以来一直存在遗漏检查的情况，这些检查的缺失可能被用户程序利用以获取内核权限。xv6很可能没有完全验证提供给内核的用户级数据，恶意用户程序可能会利用这一点来绕过xv6的隔离。

3.9 现实世界

像大多数操作系统一样，xv6 使用分页硬件进行内存保护和映射。大多数操作系统比 xv6 更复杂地使用分页，通过结合分页和页错误异常，我们将在第4章中讨论这些内容。

and page-fault exceptions, which we will discuss in Chapter 4.

Xv6 is simplified by the kernel's use of a direct map between virtual and physical addresses, and by its assumption that there is physical RAM at address 0x80000000, where the kernel expects to be loaded. This works with QEMU, but on real hardware it turns out to be a bad idea; real hardware places RAM and devices at unpredictable physical addresses, so that (for example) there might be no RAM at 0x80000000, where xv6 expect to be able to store the kernel. More serious kernel designs exploit the page table to turn arbitrary hardware physical memory layouts into predictable kernel virtual address layouts.

RISC-V supports protection at the level of physical addresses, but xv6 doesn't use that feature.

On machines with lots of memory it might make sense to use RISC-V's support for "super pages." Small pages make sense when physical memory is small, to allow allocation and page-out to disk with fine granularity. For example, if a program uses only 8 kilobytes of memory, giving it a whole 4-megabyte super-page of physical memory is wasteful. Larger pages make sense on machines with lots of RAM, and may reduce overhead for page-table manipulation.

The xv6 kernel's lack of a `malloc`-like allocator that can provide memory for small objects prevents the kernel from using sophisticated data structures that would require dynamic allocation. A more elaborate kernel would likely allocate many different sizes of small blocks, rather than (as in xv6) just 4096-byte blocks; a real kernel allocator would need to handle small allocations as well as large ones.

Memory allocation is a perennial hot topic, the basic problems being efficient use of limited memory and preparing for unknown future requests [9]. Today people care more about speed than space efficiency.

3.10 Exercises

1. Parse RISC-V's device tree to find the amount of physical memory the computer has.
2. Write a user program that grows its address space by one byte by calling `sbrk(1)`. Run the program and investigate the page table for the program before the call to `sbrk` and after the call to `sbrk`. How much space has the kernel allocated? What does the PTE for the new memory contain?
3. Modify xv6 to use super pages for the kernel.
4. Unix implementations of `exec` traditionally include special handling for shell scripts. If the file to execute begins with the text `#!`, then the first line is taken to be a program to run to interpret the file. For example, if `exec` is called to run `myprog arg1` and `myprog` 's first line is `#!/interp`, then `exec` runs `/interp` with command line `/interp myprog arg1`. Implement support for this convention in xv6.
5. Implement address space layout randomization for the kernel.

和页错误异常，我们将在第4章中讨论。

Xv6 通过内核在虚拟地址和物理地址之间使用直接映射以及假设在地址 0x80000000 处存在物理内存而得以简化，内核期望在此处加载。这在 QEMU 上可以工作，但在真实硬件上却是个坏主意；真实硬件将 RAM 和设备放置在不确定的物理地址上，因此（例如）在 0x80000000 处可能没有 RAM，而 xv6 期望能够在此存储内核。更严谨的内核设计利用页表将任意的硬件物理内存布局转换为可预测的内核虚拟地址布局。

RISC-V 支持物理地址级别的保护，但 xv6 没有使用该功能。在内存充足的机器上使用 RISC-V 对"超级页"的支持可能是有意义的。当物理内存较小时，小页是有意义的，以允许以细粒度进行分配和页面换出到磁盘。例如，如果程序只使用 8 字节的内存，给它一个完整的 4MB 超级页的物理内存是浪费的。大页在内存充足的机器上是有意义的，并且可能减少页表操作的开销。

xv6 内核缺乏能够为小对象提供内存的类似malloc的分配器，这阻止了内核使用需要动态分配的复杂数据结构。一个更完善的内核可能会分配许多不同大小的小块，而不是（如xv6那样）仅4096字节块；真实的内核分配器需要处理大小和大的分配。

内存分配是一个永恒的热门话题，基本问题是有效利用有限内存和准备未知未来请求 [9]。今天人们更关心速度而不是空间效率。

3.10 练习

1. 解析RISC-V的设备树以找到计算机的物理内存量。
2. 编写一个用户程序，通过调用sbrk(1)使地址空间增长一个字节。运行该程序，并在调用sbrk之前和之后调查程序的页表。内核分配了多少空间？新内存的PTE包含什么？
3. 修改xv6以使用超级页为内核。
4. Unix的exec实现传统上包括对shell脚本的特殊处理。如果可执行文件以文本#!开头，则第一行被视为要运行的程序以解释该文件。例如，如果exec被调用以运行 myprog arg1，并且myprog的第一行是#!/interp，那么exec会运行/interp，命令行为/interp myprog arg1。在xv6中实现对此约定的支持。
5. 为内核实现地址空间布局随机化。

Chapter 4

Traps and system calls

There are three kinds of event which cause the CPU to set aside ordinary execution of instructions and force a transfer of control to special code that handles the event. One situation is a system call, when a user program executes the `ecall` instruction to ask the kernel to do something for it. Another situation is an *exception*: an instruction (user or kernel) does something illegal, such as divide by zero or use an invalid virtual address. The third situation is a device *interrupt*, when a device signals that it needs attention, for example when the disk hardware finishes a read or write request.

This book uses *trap* as a generic term for these situations. Typically whatever code was executing at the time of the trap will later need to resume, and shouldn't need to be aware that anything special happened. That is, we often want traps to be transparent; this is particularly important for device interrupts, which the interrupted code typically doesn't expect. The usual sequence is that a trap forces a transfer of control into the kernel; the kernel saves registers and other state so that execution can be resumed; the kernel executes appropriate handler code (e.g., a system call implementation or device driver); the kernel restores the saved state and returns from the trap; and the original code resumes where it left off.

Xv6 handles all traps in the kernel; traps are not delivered to user code. Handling traps in the kernel is natural for system calls. It makes sense for interrupts since isolation demands that only the kernel be allowed to use devices, and because the kernel is a convenient mechanism with which to share devices among multiple processes. It also makes sense for exceptions since xv6 responds to all exceptions from user space by killing the offending program.

Xv6 trap handling proceeds in four stages: hardware actions taken by the RISC-V CPU, some assembly instructions that prepare the way for kernel C code, a C function that decides what to do with the trap, and the system call or device-driver service routine. While commonality among the three trap types suggests that a kernel could handle all traps with a single code path, it turns out to be convenient to have separate code for two distinct cases: traps from user space, and traps from kernel space. Kernel code (assembler or C) that processes a trap is often called a *handler*; the first handler instructions are usually written in assembler (rather than C) and are sometimes called a *vector*.

第4章

陷阱和系统调用

有三种事件会导致CPU暂停普通指令的执行，并强制将控制权转移到处理该事件的特殊代码。一种情况是系统调用，当用户程序执行ecall指令请求内核为其做某事时会发生这种情况。另一种情况是异常：指令（用户或内核）执行了非法操作，例如除零或使用无效虚拟地址。第三种情况是设备中断，当设备发出需要关注的信号时，例如磁盘硬件完成读取或写入请求时。

这本书使用陷阱作为这些情况的通用术语。通常，在陷阱发生时正在执行的代码稍后需要恢复，并且不需要知道发生了什么特别的事情。也就是说，我们通常希望陷阱是透明的；这对于设备中断尤其重要，因为被中断的代码通常不会期望这种情况。通常的顺序是，陷阱强制控制转移到内核；内核保存寄存器和其他状态，以便可以恢复执行；内核执行适当的处理程序代码（例如，系统调用实现或设备驱动程序）；内核恢复保存的状态并从陷阱返回；然后原始代码在它停止的地方继续执行。

Xv6 在内核中处理所有陷阱；陷阱不会传递给用户代码。在内核中处理陷阱对于系统调用来说是自然的。对于中断来说也是合理的，因为隔离要求只有内核才能使用设备，并且因为内核是多进程之间共享设备的方便机制。对于异常来说也是合理的，因为 xv6 通过杀死有问题的程序来响应用户空间的所有异常。

Xv6 陷阱处理分为四个阶段：RISC-V CPU 执行的硬件操作、为内核C代码做准备的一些汇编指令、决定如何处理陷阱的C函数，以及系统调用或设备驱动服务例程。虽然三种陷阱类型之间的共性表明内核可以使用单个代码路径处理所有陷阱，但事实证明，为两种不同的情况分别编写单独的代码更方便：来自用户空间的陷阱和来自内核空间的陷阱。处理陷阱的内核代码（汇编或C）通常称为handler；第一个handler指令通常用汇编（而不是C）编写，有时称为vector。

4.1 RISC-V trap machinery

Each RISC-V CPU has a set of control registers that the kernel writes to tell the CPU how to handle traps, and that the kernel can read to find out about a trap that has occurred. The RISC-V documents contain the full story [3]. `riscv.h` (kernel/riscv.h:1) contains definitions that xv6 uses. Here's an outline of the most important registers:

- `stvec`: The kernel writes the address of its trap handler here; the RISC-V jumps to the address in `stvec` to handle a trap.
- `sepc`: When a trap occurs, RISC-V saves the program counter here (since the `pc` is then overwritten with the value in `stvec`). The `sret` (return from trap) instruction copies `sepc` to the `pc`. The kernel can write `sepc` to control where `sret` goes.
- `scause`: RISC-V puts a number here that describes the reason for the trap.
- `sscratch`: The trap handler code uses `sscratch` to help it avoid overwriting user registers before saving them.
- `sstatus`: The SIE bit in `sstatus` controls whether device interrupts are enabled. If the kernel clears SIE, the RISC-V will defer device interrupts until the kernel sets SIE. The SPP bit indicates whether a trap came from user mode or supervisor mode, and controls to what mode `sret` returns.

The above registers relate to traps handled in supervisor mode, and they cannot be read or written in user mode.

Each CPU on a multi-core chip has its own set of these registers, and more than one CPU may be handling a trap at any given time.

When it needs to force a trap, the RISC-V hardware does the following for all trap types:

1. If the trap is a device interrupt, and the `sstatus` SIE bit is clear, don't do any of the following.
2. Disable interrupts by clearing the SIE bit in `sstatus`.
3. Copy the `pc` to `sepc`.
4. Save the current mode (user or supervisor) in the SPP bit in `sstatus`.
5. Set `scause` to reflect the trap's cause.
6. Set the mode to supervisor.
7. Copy `stvec` to the `pc`.
8. Start executing at the new `pc`.

4.1 RISC-V 陷阱机制

每个RISC-V CPU有一组控制寄存器，内核写入这些寄存器以告诉CPU如何处理陷阱，内核也可以读取这些寄存器以了解已发生的陷阱。RISC-V文档包含完整的故事 [3]。riscv.h (kernel/riscv.h:1) 包含xv6使用的定义。以下是最重要的寄存器概述：

- stvec: 内核将异常处理程序的地址写入此处；RISC-V 跳转到 stvec 中的地址以处理异常。地址以处理异常。
- sepc: 当异常发生时，RISC-V 将程序计数器保存于此处（因为 pc 会被 stvec 中的值覆盖）。sret（从异常返回）指令将 sepc 复制到 pc。内核可以写入 sepc 以控制 sret 的去向。
- scause: RISC-V 在此处放置一个数字，描述异常的原因。
- sscratch: 异常处理程序代码使用 sscratch 以帮助其避免覆盖用户寄存器在保存它们之前。
- sstatus: sstatus中的SIE位控制设备中断是否启用。如果内核清除SIE，RISC-V将延迟设备中断，直到内核设置SIE。SPP位指示陷阱是否来自用户模式或监督模式，并控制sret返回的模式。

上述寄存器与在监督模式下处理的陷阱相关，它们不能在用户模式下读取或写入。

多核芯片上的每个CPU都有自己的这些寄存器集，并且可能有多于一个CPU在任何给定时间处理一个陷阱。

当需要强制陷阱时，RISC-V硬件对所有陷阱类型执行以下操作：

1. 如果陷阱是设备中断，并且sstatus SIE位为清除，则不要执行以下任何操作。
2. 通过清除 sstatus 中的 SIE 位禁用中断。
3. 将 pc 复制到 sepc。
4. 将当前模式（用户或 supervisor）保存在 sstatus 中的 SPP 位中。
5. 将 scause 设置为反映陷阱的原因。
6. 将模式设置为 supervisor。
7. 将 stvec 复制到 pc。
8. 在新的 pc 处开始执行。

Note that the CPU doesn't switch to the kernel page table, doesn't switch to a stack in the kernel, and doesn't save any registers other than the `pc`. Kernel software must perform these tasks. One reason that the CPU does minimal work during a trap is to provide flexibility to software; for example, some operating systems omit a page table switch in some situations to increase trap performance.

It's worth thinking about whether any of the steps listed above could be omitted, perhaps in search of faster traps. Though there are situations in which a simpler sequence can work, many of the steps would be dangerous to omit in general. For example, suppose that the CPU didn't switch program counters. Then a trap from user space could switch to supervisor mode while still running user instructions. Those user instructions could break user/kernel isolation, for example by modifying the `satp` register to point to a page table that allowed accessing all of physical memory. It is thus important that the CPU switch to a kernel-specified instruction address, namely `stvec`.

4.2 Traps from user space

Xv6 handles traps differently depending on whether the trap occurs while executing in the kernel or in user code. Here is the story for traps from user code; Section 4.5 describes traps from kernel code.

A trap may occur while executing in user space if the user program makes a system call (`ecall` instruction), or does something illegal, or if a device interrupts. The high-level path of a trap from user space is `uservec` (kernel/trampoline.S:22), then `usertrap` (kernel/trap.c:37); and when returning, `usertrapret` (kernel/trap.c:90) and then `userret` (kernel/trampoline.S:101).

A major constraint on the design of xv6's trap handling is the fact that the RISC-V hardware does not switch page tables when it forces a trap. This means that the trap handler address in `stvec` must have a valid mapping in the user page table, since that's the page table in force when the trap handling code starts executing. Furthermore, xv6's trap handling code needs to switch to the kernel page table; in order to be able to continue executing after that switch, the kernel page table must also have a mapping for the handler pointed to by `stvec`.

Xv6 satisfies these requirements using a *trampoline* page. The trampoline page contains `uservec`, the xv6 trap handling code that `stvec` points to. The trampoline page is mapped in every process's page table at address `TRAMPOLINE`, which is at the top of the virtual address space so that it will be above memory that programs use for themselves. The trampoline page is also mapped at address `TRAMPOLINE` in the kernel page table. See Figure 2.3 and Figure 3.3. Because the trampoline page is mapped in the user page table, traps can start executing there in supervisor mode. Because the trampoline page is mapped at the same address in the kernel address space, the trap handler can continue to execute after it switches to the kernel page table.

The code for the `uservec` trap handler is in `trampoline.S` (kernel/trampoline.S:22). When `uservec` starts, all 32 registers contain values owned by the interrupted user code. These 32 values need to be saved somewhere in memory, so that later on the kernel can restore them before returning to user space. Storing to memory requires use of a register to hold the address, but at this point there are no general-purpose registers available! Luckily RISC-V provides a helping hand in the form of the `sscratch` register. The `csrw` instruction at the start of `uservec` saves `a0` in

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 在陷阱期间工作量最小的另一个原因是为软件提供灵活性；例如，某些操作系统在某些情况下省略页表切换以提高陷阱性能。

值得思考的是，上述步骤中是否有可能被省略，或许是为了追求更快的陷阱。尽管在某些情况下，更简单的序列可以奏效，但在一般情况下，省略许多步骤可能是危险的。例如，假设CPU没有切换程序计数器。那么，来自用户空间的陷阱可以在仍在运行用户指令的情况下切换到监督模式。这些用户指令可能会破坏用户/内核隔离，例如通过修改satp寄存器以指向一个允许访问所有物理内存的页表。因此，CPU切换到内核指定的指令地址，即stvec，非常重要。

4.2 来自用户空间的陷阱

Xv6根据陷阱是在内核中执行还是在用户代码中执行而以不同的方式处理陷阱。以下是来自用户代码的陷阱的故事；第4.5节描述了来自内核代码的陷阱。

如果用户程序进行系统调用（ecall指令）、执行非法操作，或者设备中断，则可能在用户空间发生陷阱。来自用户空间陷阱的高级路径是uservec（kernel/trampoline.S:22），然后是usertrap（kernel/trap.c:37）；在返回时，是usertrapret（kernel/trap.c:90）然后是userret（kernel/trampoline.S:101）。

xv6的异常处理设计的一个主要约束是，RISC-V硬件在强制异常时不会切换页表。这意味着stvec中的异常处理程序地址必须在用户页表中有一个有效映射，因为当异常处理代码开始执行时，该页表是当前生效的页表。此外，xv6的异常处理代码需要切换到内核页表；为了能够在切换后继续执行，内核页表也必须包含stvec指向的处理程序的映射。

Xv6使用一个跳板页来满足这些要求。跳板页包含uservec，即stvec指向的xv6异常处理代码。跳板页在每个进程的页表中映射在地址TRAMPOLINE，该地址位于虚拟地址空间的最顶端，以便它位于程序用于自身的内存之上。跳板页也在内核页表的地址TRAMPOLINE处映射。参见图2.3和图3.3。由于跳板页在用户页表中映射，异常可以在监督模式下从那里开始执行。由于跳板页在内核地址空间中相同的地址处映射，异常处理程序可以在切换到内核页表后继续执行。

用户vec异常处理程序的代码位于 trampoline.S (kernel/trampoline.S:22)。当用户vec启动时，所有32个寄存器都包含被中断的用户代码所拥有的值。这32个值需要保存在内存中的某个地方，以便稍后内核可以在返回到用户空间之前将它们恢复。存储到内存需要使用一个寄存器来保存地址，但在这一点上没有可用的通用寄存器！幸运的是，RISC-V 提供了 sscratch 寄存器这种帮助。用户vec开头的 csrw 指令将 a0 保存在

sscratch. Now uservec has one register (a0) to play with.

uservec's next task is to save the 32 user registers. The kernel allocates, for each process, a page of memory for a trapframe structure that (among other things) has space to save the 32 user registers (kernel/proc.h:43). Because satp still refers to the user page table, uservec needs the trapframe to be mapped in the user address space. Xv6 maps each process's trapframe at virtual address TRAPFRAME in that process's user page table; TRAPFRAME is just below TRAMPOLINE. The process's p->trapframe also points to the trapframe, though at its physical address so the kernel can use it through the kernel page table.

Thus uservec loads address TRAPFRAME into a0 and saves all the user registers there, including the user's a0, read back from sscratch.

The trapframe contains the address of the current process's kernel stack, the current CPU's hartid, the address of the usertrap function, and the address of the kernel page table. uservec retrieves these values, switches satp to the kernel page table, and jumps to usertrap.

The job of usertrap is to determine the cause of the trap, process it, and return (kernel/-trap.c:37). It first changes stvec so that a trap while in the kernel will be handled by kernelvec rather than uservec. It saves the sepc register (the saved user program counter), because usertrap might call yield to switch to another process's kernel thread, and that process might return to user space, in the process of which it will modify sepc. If the trap is a system call, usertrap calls syscall to handle it; if a device interrupt, devintr; otherwise it's an exception, and the kernel kills the faulting process. The system call path adds four to the saved user program counter because RISC-V, in the case of a system call, leaves the program pointer pointing to the ecall instruction but user code needs to resume executing at the subsequent instruction. On the way out, usertrap checks if the process has been killed or should yield the CPU (if this trap is a timer interrupt).

The first step in returning to user space is the call to usertrapret (kernel/trap.c:90). This function sets up the RISC-V control registers to prepare for a future trap from user space: setting stvec to uservec and preparing the trapframe fields that uservec relies on. usertrapret sets sepc to the previously saved user program counter. At the end, usertrapret calls userret on the trampoline page that is mapped in both user and kernel page tables; the reason is that assembly code in userret will switch page tables.

usertrapret's call to userret passes a pointer to the process's user page table in a0 (kernel/trampoline.S:101). userret switches satp to the process's user page table. Recall that the user page table maps both the trampoline page and TRAPFRAME, but nothing else from the kernel. The trampoline page mapping at the same virtual address in user and kernel page tables allows userret to keep executing after changing satp. From this point on, the only data userret can use is the register contents and the content of the trapframe. userret loads the TRAPFRAME address into a0, restores saved user registers from the trapframe via a0, restores the saved user a0, and executes sret to return to user space.

sscratch。现在用户vec有一个寄存器 (a0) 可以使用了。

uservec 的下一个任务是保存 32 个用户寄存器。内核为每个进程分配一页内存用于陷阱帧结构，该结构（除其他功能外）有空间保存 32 个用户寄存器（kernel/proc.h:43）。因为 satp 仍然指向用户页表，uservec 需要将陷阱帧映射到用户地址空间。Xv6 将每个进程的陷阱帧映射到该进程用户页表中的虚拟地址 TRAPFRAME，TRAPFRAME 紧邻 TRAMPOLINE。进程的 p->trapframe 也指向陷阱帧，但使用其物理地址，以便内核可以通过内核页表使用它。

因此 uservec 将地址 TRAPFRAME 加载到 a0 并将所有用户寄存器保存到那里，包括从 sscratch 读取的用户 a0。

陷阱帧包含当前进程的内核栈地址、当前 CPU 的 hartid、用户陷阱函数的地址以及内核页表的地址。uservec 获取这些值，将 satp 切换到内核页表，并跳转到用户陷阱。

用户陷阱的任务是确定陷阱的原因，进行处理，并返回（内核/- trap.c:37）。它首先更改stvec，以便在内核中发生陷阱时由kernelvec而不是uservec进行处理。它保存sepc寄存器（保存的用户程序计数器），因为用户陷阱可能会调用yield以切换到另一个进程的内核线程，而该进程可能会返回到用户空间，在这个过程中它会修改sepc。如果陷阱是一个系统调用，用户陷阱会调用syscall来处理它；如果是设备中断，则调用devintr；否则它是一个异常，内核会杀死故障进程。系统调用路径将保存的用户程序计数器加四，因为RISC-V在系统调用的情况下会留下程序指针指向ecall指令，但用户代码需要在后续指令处继续执行。在退出时，用户陷阱检查进程是否已被杀死或应放弃CPU（如果这个陷阱是定时器中断）。

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

usertrapret 对 userret 的调用在 a0 中传递了进程的用户页表的指针（kernel/trampoline.S:101）。userret 将 satp 切换到进程的用户页表。回想一下，用户页表映射了跳板页和 TRAPFRAME，但内核中的其他内容则不映射。用户和内核页表中相同虚拟地址的跳板页映射允许 userret 在更改 satp 后继续执行。从这一点开始，userret 只能使用的唯一数据是寄存器内容和陷阱帧的内容。userret 将 TRAPFRAME 地址加载到 a0 中，通过 a0 从陷阱帧中恢复保存的用户寄存器，恢复保存的用户 a0，并执行 sret 以返回用户空间。

4.3 Code: Calling system calls

Chapter 2 ended with initcode.S invoking the exec system call (user/initcode.S:11). Let's look at how the user call makes its way to the exec system call's implementation in the kernel.

initcode.S places the arguments for exec in registers a0 and a1, and puts the system call number in a7. System call numbers match the entries in the syscalls array, a table of function pointers (kernel/syscall.c:107). The ecall instruction traps into the kernel and causes uservec, usertrap, and then syscall to execute, as we saw above.

syscall (kernel/syscall.c:132) retrieves the system call number from the saved a7 in the trapframe and uses it to index into syscalls. For the first system call, a7 contains SYS_exec (kernel/syscall.h:8), resulting in a call to the system call implementation function sys_exec.

When sys_exec returns, syscall records its return value in p->trapframe->a0. This will cause the original user-space call to exec() to return that value, since the C calling convention on RISC-V places return values in a0. System calls conventionally return negative numbers to indicate errors, and zero or positive numbers for success. If the system call number is invalid, syscall prints an error and returns −1.

4.4 Code: System call arguments

System call implementations in the kernel need to find the arguments passed by user code. Because user code calls system call wrapper functions, the arguments are initially where the RISC-V C calling convention places them: in registers. The kernel trap code saves user registers to the current process's trap frame, where kernel code can find them. The kernel functions argint, argaddr, and argfd retrieve the *n* 'th system call argument from the trap frame as an integer, pointer, or a file descriptor. They all call argraw to retrieve the appropriate saved user register (kernel/syscall.c:34).

Some system calls pass pointers as arguments, and the kernel must use those pointers to read or write user memory. The exec system call, for example, passes the kernel an array of pointers referring to string arguments in user space. These pointers pose two challenges. First, the user program may be buggy or malicious, and may pass the kernel an invalid pointer or a pointer intended to trick the kernel into accessing kernel memory instead of user memory. Second, the xv6 kernel page table mappings are not the same as the user page table mappings, so the kernel cannot use ordinary instructions to load or store from user-supplied addresses.

The kernel implements functions that safely transfer data to and from user-supplied addresses. fetchstr is an example (kernel/syscall.c:25). File system calls such as exec use fetchstr to retrieve string file-name arguments from user space. fetchstr calls copyinstr to do the hard work.

copyinstr (kernel/vm.c:415) copies up to max bytes to dst from virtual address srcva in the user page table pagetable. Since pagetable is *not* the current page table, copyinstr uses walkaddr (which calls walk) to look up srcva in pagetable, yielding physical address pa0. The kernel's page table maps all of physical RAM at virtual addresses that are equal to the RAM's physical address. This allows copyinstr to directly copy string bytes from pa0 to dst. walkaddr (kernel/vm.c:109) checks that the user-supplied virtual address is part of the process's

4.3 代码：调用系统调用

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

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

syscall (kernel/syscall.c:132) 从陷阱帧中保存的 a7 中检索系统调用号，并使用它来索引 syscalls。对于第一个系统调用，a7 包含 SYS_exec（kernel/syscall.h:8），导致调用系统调用实现函数 sys_exec。

当 sys_exec 返回时，syscall 将其返回值记录在 p->trapframe->a0 中。这将导致原始用户空间调用 exec() 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来指示错误，以及零或正数表示成功。如果系统调用号无效，syscall 打印错误并返回 −1。

4.4 代码：系统调用参数

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

一些系统调用将指针作为参数传递，内核必须使用这些指针来读取或写入用户内存。例如，exec 系统调用将一个指向用户空间中字符串参数的指针数组传递给内核。这些指针带来了两个挑战。首先，用户程序可能有错误或恶意，可能会传递给内核一个无效的指针，或者一个意图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，所以内核不能使用普通指令从用户提供的地址加载或存储。

内核实现了将数据安全地传输到和从用户提供的地址的功能。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 来完成这项艰巨的工作。

copyinstr (内核/vm.c:415) 将最多 max 字节从用户页表 pagetable 中的虚拟地址 srcva 复制到 dst。由于 pagetable 不是当前页表，copyinstr 使用 walkaddr（调用 walk）来在 pagetable 中查找 srcva，从而得到物理地址 pa0。内核的页表将所有物理内存映射到虚拟地址，这些虚拟地址等于内存的物理地址。这允许 copyinstr 直接从 pa0 复制字符串字节到 dst。walkaddr (内核/vm.c:109) 检查用户提供的虚拟地址是否是进程的用户地址空间的一部分，因此程序不能欺骗内核去读取其他内存。一个类似的函数 copyout 将数据从内核复制到用户提供的地址。

user address space, so programs cannot trick the kernel into reading other memory. A similar function, `copyout`, copies data from the kernel to a user-supplied address.

4.5 Traps from kernel space

Xv6 handles traps from kernel code in a different way than traps from user code. When entering the kernel, `usertrap` points `stvec` to the assembly code at `kernelvec` (kernel/kernelvec.S:12). Since `kernelvec` only executes if xv6 was already in the kernel, `kernelvec` can rely on `satp` being set to the kernel page table, and on the stack pointer referring to a valid kernel stack. `kernelvec` pushes all 32 registers onto the stack, from which it will later restore them so that the interrupted kernel code can resume without disturbance.

`kernelvec` saves the registers on the stack of the interrupted kernel thread, which makes sense because the register values belong to that thread. This is particularly important if the trap causes a switch to a different thread – in that case the trap will actually return from the stack of the new thread, leaving the interrupted thread's saved registers safely on its stack.

`kernelvec` jumps to `kerneltrap` (kernel/trap.c:135) after saving registers. `kerneltrap` is prepared for two types of traps: device interrupts and exceptions. It calls `devintr` (kernel/trap.c:185) to check for and handle the former. If the trap isn't a device interrupt, it must be an exception, and that is always a fatal error if it occurs in the xv6 kernel; the kernel calls `panic` and stops executing.

If `kerneltrap` was called due to a timer interrupt, and a process's kernel thread is running (as opposed to a scheduler thread), `kerneltrap` calls `yield` to give other threads a chance to run. At some point one of those threads will yield, and let our thread and its `kerneltrap` resume again. Chapter 7 explains what happens in `yield`.

When `kerneltrap`'s work is done, it needs to return to whatever code was interrupted by the trap. Because a `yield` may have disturbed `sepc` and the previous mode in `sstatus`, `kerneltrap` saves them when it starts. It now restores those control registers and returns to `kernelvec` (kernel/kernelvec.S:38). `kernelvec` pops the saved registers from the stack and executes `sret`, which copies `sepc` to `pc` and resumes the interrupted kernel code.

It's worth thinking through how the trap return happens if `kerneltrap` called `yield` due to a timer interrupt.

Xv6 sets a CPU's `stvec` to `kernelvec` when that CPU enters the kernel from user space; you can see this in `usertrap` (kernel/trap.c:29). There's a window of time when the kernel has started executing but `stvec` is still set to `uservec`, and it's crucial that no device interrupt occur during that window. Luckily the RISC-V always disables interrupts when it starts to take a trap, and `usertrap` doesn't enable them again until after it sets `stvec`.

4.6 Page-fault exceptions

Xv6's response to exceptions is quite boring: if an exception happens in user space, the kernel kills the faulting process. If an exception happens in the kernel, the kernel panics. Real operating

用户地址空间，所以程序不能欺骗内核去读取其他内存。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 来自内核空间的陷阱

xv6 处理来自内核代码的陷阱的方式与来自用户代码的陷阱不同。当进入内核时，usertrap 将 stvec 指向 kernelvec 中的汇编代码（kernel/kernelvec.S:12）。由于 kernelvec 只有在 xv6 已经在内核中时才会执行，因此 kernelvec 可以依赖 satp 被设置为内核页表，并且栈指针指向一个有效的内核栈。kernelvec 将所有 32 个寄存器推入栈中，稍后将从栈中恢复它们，以便被中断的内核代码可以无干扰地继续执行。

kernelvec 将被中断的内核线程的寄存器保存在栈上，这是有道理的，因为寄存器值属于该线程。如果陷阱导致切换到不同的线程，这一点尤其重要——在这种情况下，陷阱实际上将从新线程的栈中返回，而被中断线程的保存寄存器将安全地留在其栈上。

kernelvec 保存寄存器后跳转到 kerneltrap（kernel/trap.c:135）。kerneltrap 准备处理两种类型的陷阱：设备中断和异常。它调用 devintr（kernel/trap.c:185）来检查和处理前者。如果陷阱不是设备中断，那么它必须是异常，如果它发生在 xv6 内核中，那么这始终是一个致命错误；内核调用 panic 并停止执行。

如果内核陷阱是由于定时器中断被调用，并且一个进程的内核线程正在运行（而不是调度线程），内核陷阱会调用 yield 以给其他线程运行的机会。在某个时刻，其中一个线程会 yield，并允许我们的线程及其内核陷阱再次恢复。第 7 章解释了 yield 中发生的情况。

当内核陷阱的工作完成后，它需要返回到被陷阱中断的代码。由于 yield 可能会扰乱 sepc 和 sstatus 中的先前模式，内核陷阱在开始时保存它们。现在它恢复这些控制寄存器并返回到 kernelvec（kernel/kernelvec.S:38）。kernelvec 从栈中弹出保存的寄存器并执行 sret，sret 将 sepc 复制到 pc 并恢复中断的内核代码。

值得思考一下如果内核陷阱由于定时器中断调用 yield，陷阱返回是如何发生的。

Xv6 在 CPU 从用户空间进入内核时将 stvec 设置为 kernelvec；你可以在 usertrap（kernel/trap.c:29）中看到这一点。有一个内核已经开始执行但 stvec 仍然设置为 uservec 的时间窗口，在这个时间窗口内，必须确保不会发生设备中断。幸运的是，RISC-V 在开始捕获陷阱时始终禁用中断，并且 usertrap 在设置 stvec 之前不会再次启用它们。

4.6 页错误异常

Xv6 对异常的处理相当无聊：如果异常发生在用户空间，内核杀死故障进程。如果异常发生在内核中，内核崩溃。真实操作系统

systems often respond in much more interesting ways.

As an example, many kernels use page faults to implement *copy-on-write (COW) fork*. To explain copy-on-write fork, consider xv6's `fork`, described in Chapter 3. `fork` causes the child's initial memory content to be the same as the parent's at the time of the fork. Xv6 implements fork with `uvmcopy` (kernel/vm.c:313), which allocates physical memory for the child and copies the parent's memory into it. It would be more efficient if the child and parent could share the parent's physical memory. A straightforward implementation of this would not work, however, since it would cause the parent and child to disrupt each other's execution with their writes to the shared stack and heap.

Parent and child can safely share physical memory by appropriate use of page-table permissions and page faults. The CPU raises a *page-fault exception* when a virtual address is used that has no mapping in the page table, or has a mapping whose `PTE_V` flag is clear, or a mapping whose permission bits (`PTE_R`, `PTE_W`, `PTE_X`, `PTE_U`) forbid the operation being attempted. RISC-V distinguishes three kinds of page fault: load page faults (caused by load instructions), store page faults (caused by store instructions), and instruction page faults (caused by fetches of instructions to be executed). The `scause` register indicates the type of the page fault and the `stval` register contains the address that couldn't be translated.

The basic plan in COW fork is for the parent and child to initially share all physical pages, but for each to map them read-only (with the `PTE_W` flag clear). Parent and child can read from the shared physical memory. If either writes a given page, the RISC-V CPU raises a page-fault exception. The kernel's trap handler responds by allocating a new page of physical memory and copying into it the physical page that the faulted address maps to. The kernel changes the relevant PTE in the faulting process's page table to point to the copy and to allow writes as well as reads, and then resumes the faulting process at the instruction that caused the fault. Because the PTE now allows writes, the re-executed instruction will execute without a fault. Copy-on-write requires book-keeping to help decide when physical pages can be freed, since each page can be referenced by a varying number of page tables depending on the history of forks, page faults, execs, and exits. This book-keeping allows an important optimization: if a process incurs a store page fault and the physical page is only referred to from that process's page table, no copy is needed.

Copy-on-write makes `fork` faster, since `fork` need not copy memory. Some of the memory will have to be copied later, when written, but it's often the case that most of the memory never has to be copied. A common example is `fork` followed by `exec`: a few pages may be written after the `fork`, but then the child's `exec` releases the bulk of the memory inherited from the parent. Copy-on-write `fork` eliminates the need to ever copy this memory. Furthermore, COW fork is transparent: no modifications to applications are necessary for them to benefit.

The combination of page tables and page faults opens up a wide range of interesting possibilities in addition to COW fork. Another widely-used feature is called *lazy allocation*, which has two parts. First, when an application asks for more memory by calling `sbrk`, the kernel notes the increase in size, but does not allocate physical memory and does not create PTEs for the new range of virtual addresses. Second, on a page fault on one of those new addresses, the kernel allocates a page of physical memory and maps it into the page table. Like COW fork, the kernel can implement lazy allocation transparently to applications.

通常以更有趣的方式进行处理。

例如，许多内核使用页错误来实现写时复制 (COW) 分叉。要解释写时复制分叉，可以考虑第 3 章中描述的 xv6 的 fork。fork 会导致子进程的初始内存内容在分叉时与父进程相同。xv6 使用 uvmcopy (内核/vm.c:313) 实现 fork，该函数为子进程分配物理内存，并将父进程的内存复制到其中。如果子进程和父进程能够共享父进程的物理内存，这将更高效。然而，这种直接的实现方式行不通，因为它会导致父进程和子进程通过对共享栈和堆的写入而相互干扰执行。

父进程和子进程可以通过适当使用页表权限和页错误来安全地共享物理内存。当使用一个在页表中没有映射的虚拟地址，或者有一个映射的 PTE_V 标志为清除，或者有一个映射的权限位 (PTE_R、PTE_W、PTE_X、PTE_U) 禁止尝试的操作时，CPU 会引发页错误异常。RISC-V 区分三种类型的页错误：加载页错误（由加载指令引起）、存储页错误（由存储指令引起）和指令页错误（由执行指令的获取引起）。scause 寄存器指示页错误的类型，而 stval 寄存器包含无法转换的地址。

写时复制分叉的基本方案是父进程和子进程最初共享所有物理页，但各自将它们映射为只读（清除 PTE_W 标志）。父进程和子进程可以读取共享的物理内存。如果任一进程写入某个页，RISC-V CPU 会引发页错误异常。内核的异常处理程序通过分配新的物理内存页并将故障地址映射到的物理页复制到其中来响应。内核将故障进程的页表中的相关 PTE 修改为指向副本，并允许写入和读取，然后继续执行引发故障的指令。由于 PTE 现在允许写入，重新执行的指令将不会引发故障。写时复制需要记录信息来帮助决定何时可以释放物理页，因为每个页可能被不同数量的页表引用，具体取决于分叉、页错误、exec 和退出的历史。这种记录信息允许一个重要的优化：如果进程发生存储页错误，并且物理页仅被该进程的页表引用，则无需复制。

写时复制使得分叉更快，因为分叉不需要复制内存。一些内存稍后必须被复制，当写入时，但通常大多数内存永远不需要被复制。一个常见的例子是分叉后跟exec：在分叉后可能只有几页会被写入，然后子进程的exec释放了从父进程继承的大部分内存。写时复制分叉消除了永远复制这部分内存的需要。此外，写时复制分叉是透明的：应用程序无需任何修改即可从中受益。

页表和页错误的组合除了写时复制分叉之外，还开辟了广泛的有趣可能性。另一个广泛使用的特性称为懒加载，它有两个部分。首先，当应用程序通过调用 sbrk 请求更多内存时，内核注意到大小的增加，但不会分配物理内存，也不会为新范围的虚拟地址创建 PTE。其次，在其中一个新地址上发生页错误时，内核分配一页物理内存并将其映射到页表。像写时复制分叉一样，内核可以对应用程序透明地实现懒加载。

Since applications often ask for more memory than they need, lazy allocation is a win: the kernel doesn't have to do any work at all for pages that the application never uses. Furthermore, if the application is asking to grow the address space by a lot, then `sbrk` without lazy allocation is expensive: if an application asks for a gigabyte of memory, the kernel has to allocate and zero 262,144 4096-byte pages. Lazy allocation allows this cost to be spread over time. On the other hand, lazy allocation incurs the extra overhead of page faults, which involve a user/kernel transition. Operating systems can reduce this cost by allocating a batch of consecutive pages per page fault instead of one page and by specializing the kernel entry/exit code for such page-faults.

Yet another widely-used feature that exploits page faults is *demand paging*. In `exec`, xv6 loads all of an application's text and data into memory before starting the application. Since applications can be large and reading from disk takes time, this startup cost can be noticeable to users. To decrease startup time, a modern kernel doesn't initially load the executable file into memory, but just creates the user page table with all PTEs marked invalid. The kernel starts the program running; each time the program uses a page for the first time, a page fault occurs, and in response the kernel reads the content of the page from disk and maps it into the user address space. Like COW fork and lazy allocation, the kernel can implement this feature transparently to applications.

The programs running on a computer may need more memory than the computer has RAM. To cope gracefully, the operating system may implement *paging to disk*. The idea is to store only a fraction of user pages in RAM, and to store the rest on disk in a *paging area*. The kernel marks PTEs that correspond to memory stored in the paging area (and thus not in RAM) as invalid. If an application tries to use one of the pages that has been *paged out* to disk, the application will incur a page fault, and the page must be *paged in*: the kernel trap handler will allocate a page of physical RAM, read the page from disk into the RAM, and modify the relevant PTE to point to the RAM.

What happens if a page needs to be paged in, but there is no free physical RAM? In that case, the kernel must first free a physical page by paging it out or *evicting* it to the paging area on disk, and marking the PTEs referring to that physical page as invalid. Eviction is expensive, so paging performs best if it's infrequent: if applications use only a subset of their memory pages and the union of the subsets fits in RAM. This property is often referred to as having good locality of reference. As with many virtual memory techniques, kernels usually implement paging to disk in a way that's transparent to applications.

Computers often operate with little or no *free* physical memory, regardless of how much RAM the hardware provides. For example, cloud providers multiplex many customers on a single machine to use their hardware cost-effectively. As another example, users run many applications on smart phones in a small amount of physical memory. In such settings allocating a page may require first evicting an existing page. Thus, when free physical memory is scarce, allocation is expensive.

Lazy allocation and demand paging are particularly advantageous when free memory is scarce and programs actively use only a fraction of their allocated memory. These techniques can also avoid the work wasted when a page is allocated or loaded but either never used or evicted before it can be used.

Other features that combine paging and page-fault exceptions include automatically extending stacks and *memory-mapped files*, which are files that a program mapped into its address space using the `mmap` system call so that the program can read and write them using load and store

由于应用程序通常请求比它们需要的更多的内存，懒加载是一种优势：对于应用程序从未使用的页面，内核根本不需要做任何工作。此外，如果应用程序请求通过大量扩展地址空间，那么没有懒加载的 sbrk 是昂贵的：如果应用程序请求一吉字节的内存，内核必须分配并清零 262,144 个 4096-byte 页面。懒加载允许将此成本分摊到时间上。另一方面，懒加载会产生页错误的额外开销，这涉及用户/内核转换。操作系统可以通过每次页错误分配一批连续的页面而不是一个页面来减少此成本，并通过为这种页错误专门定制内核入口/退出代码来减少此成本。

又一个广泛使用的利用页错误的功能是按需分页。在exec中，xv6在启动应用程序之前将应用程序的所有文本和数据加载到内存中。由于应用程序可能很大，而从磁盘读取需要时间，因此这种启动成本对用户来说可能很明显。为了减少启动时间，现代内核不会最初将可执行文件加载到内存中，而是仅创建用户页表，并将所有PTE标记为无效。内核开始运行程序；每次程序第一次使用一页时，都会发生页错误，作为响应，内核从磁盘读取该页的内容并将其映射到用户地址空间。像写时复制分叉和懒加载一样，内核可以透明地实现此功能。

计算机上运行的程序可能需要比计算机拥有的RAM更多的内存。为了优雅地处理这种情况，操作系统可以实现磁盘分页。其思想是只在RAM中存储用户页的一部分，并将其余部分存储在磁盘的分页区域中。内核将对应于存储在分页区域（因此不在RAM中）的内存的PTE标记为无效。如果应用程序尝试使用已从磁盘分页出去的一页，应用程序将发生页错误，并且该页必须被分页回来：内核陷阱处理程序将分配一页物理内存，将页从磁盘读取到RAM中，并修改相关的PTE以指向RAM。

如果一页需要被调入，但没有空闲的物理内存会怎样？在这种情况下，内核必须首先通过将一页分页出去或驱逐到磁盘上的分页区域来释放物理页，并将引用该物理页的 PTE 标记为无效。驱逐很昂贵，因此分页在很少发生时表现最佳：如果应用程序只使用其内存页面的子集，并且这些子集的并集适合在 RAM 中。这种特性通常被称为具有良好的引用局部性。与许多虚拟内存技术一样，内核通常以对应用程序透明的方式实现磁盘分页。

计算机通常在几乎没有或没有空闲物理内存的情况下运行，无论硬件提供的 RAM 多少。例如，云提供商在单个机器上多路复用许多客户以有效利用其硬件成本。作为另一个例子，用户在智能手机上以少量物理内存运行许多应用程序。在这种设置下，分配一页可能需要首先驱逐一个现有的页。因此，当空闲物理内存稀缺时，分配很昂贵。

懒加载和按需分页在空闲内存稀缺且程序仅活跃使用其分配内存的一小部分时尤其有利。这些技术还可以避免当页被分配或加载但从未使用或被驱逐之前被浪费的工作。

其他结合分页和页错误异常的功能包括自动扩展栈和内存映射文件，内存映射文件是程序使用mmap系统调用映射到其地址空间中的文件，以便程序可以使用加载和存储指令读取和写入它们

instructions.

4.7 Real world

The trampoline and trapframe may seem excessively complex. A driving force is that the RISC-V intentionally does as little as it can when forcing a trap, to allow the possibility of very fast trap handling, which turns out to be important. As a result, the first few instructions of the kernel trap handler effectively have to execute in the user environment: the user page table, and user register contents. And the trap handler is initially ignorant of useful facts such as the identity of the process that's running or the address of the kernel page table. A solution is possible because RISC-V provides protected places in which the kernel can stash away information before entering user space: the `sscratch` register, and user page table entries that point to kernel memory but are protected by lack of `PTE_U`. Xv6's trampoline and trapframe exploit these RISC-V features.

The need for special trampoline pages could be eliminated if kernel memory were mapped into every process's user page table (with `PTE_U` clear). That would also eliminate the need for a page table switch when trapping from user space into the kernel. That in turn would allow system call implementations in the kernel to take advantage of the current process's user memory being mapped, allowing kernel code to directly dereference user pointers. Many operating systems have used these ideas to increase efficiency. Xv6 avoids them in order to reduce the chances of security bugs in the kernel due to inadvertent use of user pointers, and to reduce some complexity that would be required to ensure that user and kernel virtual addresses don't overlap.

Production operating systems implement copy-on-write fork, lazy allocation, demand paging, paging to disk, memory-mapped files, etc. Furthermore, production operating systems try to store something useful in all areas of physical memory, typically caching file content in memory that isn't used by processes.

Production operating systems also provide applications with system calls to manage their address spaces and implement their own page-fault handling through the `mmap`, `munmap`, and `sigaction` system calls, as well as providing calls to pin memory into RAM (see `mlock`) and to advise the kernel how an application plans to use its memory (see `madvise`).

4.8 Exercises

1. The functions `copyin` and `copyinstr` walk the user page table in software. Set up the kernel page table so that the kernel has the user program mapped, and `copyin` and `copyinstr` can use `memcpy` to copy system call arguments into kernel space, relying on the hardware to do the page table walk.

2. Implement lazy memory allocation.

3. Implement COW fork.

指令。

4.7 现实世界

跳板和陷阱帧可能看起来过于复杂。一个驱动力是 RISC-V 在强制陷阱时尽可能少做，以允许非常快的陷阱处理，事实证明这很重要。因此，内核陷阱处理程序的前几条指令实际上必须在用户环境中执行：用户页表和用户寄存器内容。并且陷阱处理程序最初不知道有用的信息，例如正在运行的进程的身份或内核页表的地址。这是可能的解决方案，因为 RISC-V 提供了受保护的地方，内核可以在进入用户空间之前将信息存放在那里：sscratch 寄存器和指向内核内存但受缺乏 PTE_U 保护的用户页表条目。Xv6 的跳板和陷阱帧利用了这些 RISC-V 功能。

如果内核内存映射到每个进程的用户页表（PTE_U 清除），则可以消除对特殊跳板页的需求。这也将消除从用户空间陷入内核时对页表切换的需求。反过来，这将允许内核中的系统调用实现利用当前进程的用户内存被映射，允许内核代码直接解除引用用户指针。许多操作系统已经使用这些想法来提高效率。Xv6 避免它们是为了减少由于意外使用用户指针而导致内核中安全错误的机会，并减少一些复杂性，以确保用户和内核虚拟地址不重叠。

生产操作系统实现了写时复制分叉、懒加载、按需分页、磁盘分页、内存映射文件等。此外，生产操作系统尝试在所有物理内存区域存储有用信息，通常将文件内容缓存到未被进程使用的内存中。

生产操作系统还向应用程序提供系统调用以管理其地址空间，并通过 mmap、munmap 和 sigaction 系统调用实现其自身的页错误处理，同时提供将内存固定到 RAM 的调用（参见 mlock）以及向内核建议应用程序如何使用其内存的调用（参见 madvise）。

4.8 练习

1. 函数 copyin 和 copyinstr 在软件中遍历用户页表。设置内核页表，使内核能够映射用户程序，并使 copyin 和 copyinstr 可以使用 memcpy 将系统调用参数复制到内核空间，依赖硬件执行页表遍历。

2. 实现懒加载内存分配。

3. 实现 COW 分叉。

4. Is there a way to eliminate the special `TRAPFRAME` page mapping in every user address space? For example, could `uservec` be modified to simply push the 32 user registers onto the kernel stack, or store them in the `proc` structure?
5. Could xv6 be modified to eliminate the special `TRAMPOLINE` page mapping?
6. Implement `mmap`.

4. 是否有办法消除每个用户地址空间中的特殊 TRAPFRAME 页面映射？例如，uservec 是否可以修改为将 32 个用户寄存器直接推入内核栈，或存储在 proc 结构中？

5. xv6 是否可以修改以消除特殊的 TRAMPOLINE 页面映射？

6. 实现 mmap。

Chapter 5

Interrupts and device drivers

A *driver* is the code in an operating system that manages a particular device: it configures the device hardware, tells the device to perform operations, handles the resulting interrupts, and interacts with processes that may be waiting for I/O from the device. Driver code can be tricky because a driver executes concurrently with the device that it manages. In addition, the driver must understand the device's hardware interface, which can be complex and poorly documented.

Devices that need attention from the operating system can usually be configured to generate interrupts, which are one type of trap. The kernel trap handling code recognizes when a device has raised an interrupt and calls the driver's interrupt handler; in xv6, this dispatch happens in `devintr` (kernel/trap.c:185).

Many device drivers execute code in two contexts: a *top half* that runs in a process's kernel thread, and a *bottom half* that executes at interrupt time. The top half is called via system calls such as `read` and `write` that want the device to perform I/O. This code may ask the hardware to start an operation (e.g., ask the disk to read a block); then the code waits for the operation to complete. Eventually the device completes the operation and raises an interrupt. The driver's interrupt handler, acting as the bottom half, figures out what operation has completed, wakes up a waiting process if appropriate, and tells the hardware to start work on any waiting next operation.

5.1 Code: Console input

The console driver (kernel/console.c) is a simple illustration of driver structure. The console driver accepts characters typed by a human, via the *UART* serial-port hardware attached to the RISC-V. The console driver accumulates a line of input at a time, processing special input characters such as backspace and control-u. User processes, such as the shell, use the `read` system call to fetch lines of input from the console. When you type input to xv6 in QEMU, your keystrokes are delivered to xv6 by way of QEMU's simulated UART hardware.

The UART hardware that the driver talks to is a 16550 chip [13] emulated by QEMU. On a real computer, a 16550 would manage an RS232 serial link connecting to a terminal or other computer. When running QEMU, it's connected to your keyboard and display.

The UART hardware appears to software as a set of *memory-mapped* control registers. That

第5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：它配置设备硬件，告诉设备执行操作，处理产生的中断，并与可能等待设备 I/O 的进程交互。驱动程序代码可能很复杂，因为驱动程序与其管理的设备并发执行。此外，驱动程序必须理解设备的硬件接口，这可能是复杂且文档不完善的。

需要操作系统关注的设备通常可以配置为生成中断，中断是陷阱的一种类型。内核陷阱处理代码识别设备何时触发中断，并调用驱动程序的中断处理程序；在xv6中，这次调度发生在devintr（内核/trap.c:185）。

许多设备驱动程序在两个上下文中执行代码：一个在上半部分，它在进程的内核线程中运行，另一个在下半部分，它在中断时执行。上半部分通过read和write等系统调用被调用，这些调用希望设备执行I/O。这段代码可以要求硬件启动一个操作（例如，要求磁盘读取一个块）；然后代码等待操作完成。最终设备完成操作并触发中断。驱动程序的中断处理程序作为下半部分，弄清楚哪个操作已完成，如果合适，则唤醒等待的进程，并告诉硬件开始处理任何等待的下一个操作。

5.1 代码：控制台输入

控制台驱动程序（kernel/console.c）是驱动程序结构的简单示例。控制台驱动程序通过连接到 RISC-V 的 UART 串行端口硬件接收人类输入的字符。控制台驱动程序一次累积一行输入，处理 backspace 和 control-u 等特殊输入字符。用户进程（如 shell）使用 read 系统调用从控制台获取输入行。当你在 QEMU 中输入 xv6 时，你的按键输入通过 QEMU 的模拟 UART 硬件传递给 xv6。

驱动程序与之交互的 UART 硬件是由 QEMU 模拟的 16550 芯片 [13]。在真实计算机上，16550 会管理连接到终端或其他计算机的 RS232 串行链路。运行 QEMU 时，它会连接到你的键盘和显示器。

UART 硬件对软件表现为一组内存映射控制寄存器。

is, there are some physical addresses that RISC-V hardware connects to the UART device, so that loads and stores interact with the device hardware rather than RAM. The memory-mapped addresses for the UART start at 0x10000000, or `UART0` (kernel/memlayout.h:21). There are a handful of UART control registers, each the width of a byte. Their offsets from `UART0` are defined in (kernel/uart.c:22). For example, the `LSR` register contains bits that indicate whether input characters are waiting to be read by the software. These characters (if any) are available for reading from the `RHR` register. Each time one is read, the UART hardware deletes it from an internal FIFO of waiting characters, and clears the "ready" bit in `LSR` when the FIFO is empty. The UART transmit hardware is largely independent of the receive hardware; if software writes a byte to the `THR`, the UART transmits that byte.

Xv6's `main` calls `consoleinit` (kernel/console.c:182) to initialize the UART hardware. This code configures the UART to generate a receive interrupt when the UART receives each byte of input, and a *transmit complete* interrupt each time the UART finishes sending a byte of output (kernel/uart.c:53).

The xv6 shell reads from the console by way of a file descriptor opened by `init.c` (user/init.c:19). Calls to the `read` system call make their way through the kernel to `consoleread` (kernel/console.c:80). `consoleread` waits for input to arrive (via interrupts) and be buffered in `cons.buf`, copies the input to user space, and (after a whole line has arrived) returns to the user process. If the user hasn't typed a full line yet, any reading processes will wait in the `sleep` call (kernel/console.c:96) (Chapter 7 explains the details of `sleep`).

When the user types a character, the UART hardware asks the RISC-V to raise an interrupt, which activates xv6's trap handler. The trap handler calls `devintr` (kernel/trap.c:185), which looks at the RISC-V `scause` register to discover that the interrupt is from an external device. Then it asks a hardware unit called the PLIC [3] to tell it which device interrupted (kernel/trap.c:193). If it was the UART, `devintr` calls `uartintr`.

`uartintr` (kernel/uart.c:177) reads any waiting input characters from the UART hardware and hands them to `consoleintr` (kernel/console.c:136); it doesn't wait for characters, since future input will raise a new interrupt. The job of `consoleintr` is to accumulate input characters in `cons.buf` until a whole line arrives. `consoleintr` treats backspace and a few other characters specially. When a newline arrives, `consoleintr` wakes up a waiting `consoleread` (if there is one).

Once woken, `consoleread` will observe a full line in `cons.buf`, copy it to user space, and return (via the system call machinery) to user space.

5.2 Code: Console output

A `write` system call on a file descriptor connected to the console eventually arrives at `uartputc` (kernel/uart.c:87). The device driver maintains an output buffer (`uart_tx_buf`) so that writing processes do not have to wait for the UART to finish sending; instead, `uartputc` appends each character to the buffer, calls `uartstart` to start the device transmitting (if it isn't already), and returns. The only situation in which `uartputc` waits is if the buffer is already full.

Each time the UART finishes sending a byte, it generates an interrupt. `uartintr` calls `uartstart`,

UART硬件对软件表现为一组内存映射控制寄存器。也就是说，有一些物理地址是RISC-V硬件连接到UART设备的，因此加载和存储操作是与设备硬件交互而不是与RAM交互。UART的内存映射地址从0x10000000开始，即UART0（kernel/memlayout.h:21）。UART有一些控制寄存器，每个寄存器的宽度为一个字节。它们的偏移量从UART0开始定义在(kernel/uart.c:22)。例如，LSR寄存器包含指示输入字符是否准备好供软件读取的位。这些字符（如果有）可以从RHR寄存器读取。每次读取一个字符时，UART硬件会从等待字符的内部FIFO中删除它，当FIFO为空时，会在LSR中清除“就绪”位。UART的发送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，UART会发送该字节。

Xv6 的 main 调用 consoleinit (kernel/console.c:182) 来初始化 UART 硬件。这段代码配置 UART，使其在每次接收到输入的每个字节时生成接收中断，并在每次 UART 完成发送一个输出字节时生成传输完成中断 (kernel/uart.c:53)。

xv6 shell 通过 init.c (user/init.c:19) 打开的控制台文件描述符读取控制台。对 read 系统调用的调用会通过内核到达 consoleread (kernel/con- sole.c:80)。consoleread 等待输入到达（通过中断）并缓冲在 cons.buf 中，将输入复制到用户空间，并在（整行到达后）返回用户进程。如果用户尚未输入完整行，任何读取进程将在 sleep 调用 (kernel/con- sole.c:96) 中等待（第 7 章解释了 sleep 的详细信息）。

当用户输入一个字符时，UART 硬件请求 RISC-V 触发中断，这会激活 xv6 的异常处理程序。异常处理程序调用 devintr (kernel/trap.c:185)，它查看 RISC-V scause 寄存器，发现中断来自外部设备。然后它请求一个称为 PLIC [3] 的硬件单元告诉它哪个设备触发了中断 (kernel/trap.c:193)。如果是 UART，devintr 会调用 uartintr。

uartintr (kernel/uart.c:177) 从 UART 硬件读取任何等待的输入字符，并将它们交给 consoleintr (kernel/console.c:136)；它不等待字符，因为未来的输入将引发新的中断。consoleintr 的任务是累积输入字符到 cons.buf，直到整行到达。consoleintr 特别处理 backspace 和其他几个字符。当收到 newline 时，consoleintr 唤醒等待的 consoleread（如果有）。

一旦被唤醒，consoleread 将观察 cons.buf 中的整行，将其复制到用户空间，并通过系统调用机制返回到用户空间。

5.2 代码：控制台输出

对连接到控制台 的文件描述符发起的写系统调用最终到达 uartputc (kernel/uart.c:87)。设备驱动程序维护一个输出缓冲区 (uart_tx_buf)，以便写入进程不必等待 UART 完成发送；相反，uartputc 将每个字符追加到缓冲区，调用 uartstart 启动设备传输（如果它尚未启动），然后返回。uartputc 唯一等待的情况是缓冲区已经满了。

每次 UART 完成发送一个字节时，它都会产生一个中断。uartintr 调用 uartstart， which checks that the device really has finished sending, and hands the device the next buffered output character. Thus if a process writes multiple bytes to the console, typically the first byte will be sent by uartputc's call to uartstart, and the remaining buffered bytes will be sent by uartstart calls from uartintr as transmit complete interrupts arrive.

A general pattern to note is the decoupling of device activity from process activity via buffering and interrupts. The console driver can process input even when no process is waiting to read it; a subsequent read will see the input. Similarly, processes can send output without having to wait for the device. This decoupling can increase performance by allowing processes to execute concurrently with device I/O, and is particularly important when the device is slow (as with the UART) or needs immediate attention (as with echoing typed characters). This idea is sometimes called *I/O concurrency*.

5.3 Concurrency in drivers

You may have noticed calls to acquire in consoleread and in consoleintr. These calls acquire a lock, which protects the console driver's data structures from concurrent access. There are three concurrency dangers here: two processes on different CPUs might call consoleread at the same time; the hardware might ask a CPU to deliver a console (really UART) interrupt while that CPU is already executing inside consoleread; and the hardware might deliver a console interrupt on a different CPU while consoleread is executing. Chapter 6 explains how to use locks to ensure that these dangers don't lead to incorrect results.

Another way in which concurrency requires care in drivers is that one process may be waiting for input from a device, but the interrupt signaling arrival of the input may arrive when a different process (or no process at all) is running. Thus interrupt handlers are not allowed to think about the process or code that they have interrupted. For example, an interrupt handler cannot safely call copyout with the current process's page table. Interrupt handlers typically do relatively little work (e.g., just copy the input data to a buffer), and wake up top-half code to do the rest.

5.4 Timer interrupts

Xv6 uses timer interrupts to maintain its idea of the current time and to switch among compute-bound processes. Timer interrupts come from clock hardware attached to each RISC-V CPU. Xv6 programs each CPU's clock hardware to interrupt the CPU periodically.

Code in start.c (kernel/start.c:53) sets some control bits that allow supervisor-mode access to the timer control registers, and then asks for the first timer interrupt. The time control register contains a count that the hardware increments at a steady rate; this serves as a notion of the current time. The stimecmp register contains a time at which the the CPU will raise a timer interrupt; setting stimecmp to the current value of time plus *x* will schedule an interrupt *x* time units in the future. For qemu's RISC-V emulation, 1000000 time units is roughly a tenth of second.

Timer interrupts arrive via usertrap or kerneltrap and devintr, like other device interrupts. Timer interrupts arrive with scause's low bits set to five; devintr in trap.c detects

该函数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，通常第一个字节会通过 uartputc 对 uartstart 的调用发送，而剩余的缓冲字节会通过 uartintr 在传输完成中断到达时由 uartstart 调用发送。

一个需要注意的通用模式是设备活动与进程活动通过缓冲区和中断进行解耦。控制台驱动程序可以在没有进程等待读取的情况下处理输入；后续的读取将看到输入。类似地，进程可以发送输出而无需等待设备。这种解耦可以通过允许进程与设备I/O并发执行来提高性能，并且在设备较慢（如UART）或需要立即关注（如回显输入字符）时尤其重要。这个概念有时被称为I/O并发。

5.3 驱动程序中的并发

您可能已经注意到在consoleread和consoleintr中调用了acquire。这些调用会获取一个锁，该锁保护控制台驱动程序的数据结构免受并发访问。这里有三个并发风险：两个不同的CPU上的进程可能会同时调用consoleread；硬件可能会在CPU已经在consoleread内部执行时要求该CPU传递一个控制台（实际上是UART）中断；并且硬件可能会在consoleread执行时在不同的CPU上传递一个控制台中断。第6章解释了如何使用锁来确保这些风险不会导致不正确的结果。

并发需要驱动程序在另一个方面的注意事项是，一个进程可能正在等待来自设备的输入，但中断信号输入到达时，可能是一个不同的进程（甚至没有任何进程）正在运行。因此，中断处理程序不允许考虑它们中断的进程或代码。例如，中断处理程序不能安全地使用当前进程的页表调用 copyout。中断处理程序通常只做很少的工作（例如，只需将输入数据复制到缓冲区），然后唤醒上半代码来完成其余工作。

5.4 定时器中断

Xv6 使用定时器中断来维护其当前时间的概念，并在计算受限的进程之间进行切换。定时器中断来自每个 RISC-V CPU 上连接的时钟硬件。Xv6 程序每个 CPU 的时钟硬件，使其定期中断 CPU。

start.c (kernel/start.c:53) 中的代码设置了一些控制位，允许超级用户模式访问定时器控制寄存器，然后请求第一个定时器中断。时间控制寄存器包含一个硬件以稳定速率递增的计数；这作为当前时间的概念。stimecmp 寄存器包含 CPU 将触发定时器中断的时间；将 stimecmp 设置为当前时间值加上 x 将在未来 x 时间单位内调度一个中断。对于 qemu 的 RISC-V 模拟，1000000 时间单位大致是 1/10 秒。

定时器中断通过用户陷阱或内核陷阱以及devintr到达，与其他设备中断类似。定时器中断的scause的低位被设置为五；在trap.c中检测devintr this situation and calls `clockintr` (kernel/trap.c:164). The latter function increments `ticks`, allowing the kernel to track the passage of time. The increment occurs on only one CPU, to avoid time passing faster if there are multiple CPUs. `clockintr` wakes up any processes waiting in the `sleep` system call, and schedules the next timer interrupt by writing `stimecmp`.

`devintr` returns 2 for a timer interrupt in order to indicate to `kerneltrap` or `usertrap` that they should call `yield` so that CPUs can be multiplexed among runnable processes.

The fact that kernel code can be interrupted by a timer interrupt that forces a context switch via `yield` is part of the reason why early code in `usertrap` is careful to save state such as `sepc` before enabling interrupts. These context switches also mean that kernel code must be written in the knowledge that it may move from one CPU to another without warning.

5.5 Real world

Xv6, like many operating systems, allows interrupts and even context switches (via `yield`) while executing in the kernel. The reason for this is to retain quick response times during complex system calls that run for a long time. However, as noted above, allowing interrupts in the kernel is the source of some complexity; as a result, a few operating systems allow interrupts only while executing user code.

Supporting all the devices on a typical computer in its full glory is much work, because there are many devices, the devices have many features, and the protocol between device and driver can be complex and poorly documented. In many operating systems, the drivers account for more code than the core kernel.

The UART driver retrieves data a byte at a time by reading the UART control registers; this pattern is called *programmed I/O*, since software is driving the data movement. Programmed I/O is simple, but too slow to be used at high data rates. Devices that need to move lots of data at high speed typically use *direct memory access (DMA)*. DMA device hardware directly writes incoming data to RAM, and reads outgoing data from RAM. Modern disk and network devices use DMA. A driver for a DMA device would prepare data in RAM, and then use a single write to a control register to tell the device to process the prepared data.

Interrupts make sense when a device needs attention at unpredictable times, and not too often. But interrupts have high CPU overhead. Thus high speed devices, such as network and disk controllers, use tricks that reduce the need for interrupts. One trick is to raise a single interrupt for a whole batch of incoming or outgoing requests. Another trick is for the driver to disable interrupts entirely, and to check the device periodically to see if it needs attention. This technique is called *polling*. Polling makes sense if the device performs operations at a high rate, but it wastes CPU time if the device is mostly idle. Some drivers dynamically switch between polling and interrupts depending on the current device load.

The UART driver copies incoming data first to a buffer in the kernel, and then to user space. This makes sense at low data rates, but such a double copy can significantly reduce performance for devices that generate or consume data very quickly. Some operating systems are able to directly move data between user-space buffers and device hardware, often with DMA.

这种情况，并调用clockintr（内核/trap.c:164）。该函数递增ticks，使内核能够跟踪时间的流逝。递增仅在单个CPU上发生，以避免在多个CPU的情况下时间流逝过快。clockintr唤醒在睡眠系统调用中等待的任何进程，并通过写入stimecmp来调度下一个定时器中断。

devintr 返回 2 以表示定时器中断，以便内核陷阱或用户陷阱知道应该调用 yield，以便 CPU 可以在可运行进程之间多路复用。

内核代码可以被定时器中断中断，该中断通过 yield 强制上下文切换，这是早期用户陷阱代码需要小心保存状态（如 sepc）并在启用中断之前调用 yield 的原因之一。这些上下文切换还意味着内核代码必须在知道它可能会在没有警告的情况下从一个 CPU 移动到另一个 CPU 的知识下编写。

5.5 现实世界

Xv6 与许多操作系统一样，在内核中执行时允许中断，甚至允许通过 yield 进行上下文切换。这样做的原因是在长时间运行的复杂系统调用期间保持快速响应时间。然而，如上所述，在内核中允许中断是某些复杂性的来源；因此，一些操作系统仅在执行用户代码时允许中断。

支持典型计算机上所有设备在其全部辉煌中的工作量很大，因为有许多设备，设备有许多功能，设备和驱动程序之间的协议可能复杂且文档不完善。在许多操作系统中，驱动程序的代码量比核心内核更多。

UART驱动程序通过读取UART控制寄存器逐字节检索数据；这种模式称为程序化I/O，因为软件正在驱动数据移动。程序化I/O很简单，但在高速数据传输时太慢，无法使用。需要高速移动大量数据的设备通常使用直接内存访问（DMA）。DMA设备硬件直接将传入数据写入RAM，并从RAM读取传出数据。现代磁盘和网络设备使用DMA。DMA设备的驱动程序会在RAM中准备数据，然后通过向控制寄存器写入单个数据来告诉设备处理准备好的数据。

当中断需要在不可预测的时间需要设备关注且频率不高时才有意义。但中断具有高CPU开销。因此，高速设备（如网络和磁盘控制器）使用技巧来减少中断的需求。一种技巧是为一批传入或传出请求发起单个中断。另一种技巧是驱动程序完全禁用中断，并定期检查设备是否需要关注。这种技术称为轮询。如果设备以高频率执行操作，轮询才有意义，但如果设备大部分时间处于空闲状态，则会浪费CPU时间。一些驱动程序会根据当前设备负载在轮询和中断之间动态切换。

UART驱动程序首先将接收到的数据复制到内核中的缓冲区，然后复制到用户空间。在低数据速率下，这样做是有道理的，但对于快速生成或消耗数据的设备，这种双重复制会显著降低性能。一些操作系统能够直接在用户空间缓冲区和设备硬件之间移动数据，通常使用DMA。

As mentioned in Chapter 1, the console appears to applications as a regular file, and applications read input and write output using the `read` and `write` system calls. Applications may want to control aspects of a device that cannot be expressed through the standard file system calls (e.g., enabling/disabling line buffering in the console driver). Unix operating systems support the `ioctl` system call for such cases.

Some usages of computers require that the system must respond in a bounded time. For example, in safety-critical systems missing a deadline can lead to disasters. Xv6 is not suitable for hard real-time settings. Operating systems for hard real-time tend to be libraries that link with the application in a way that allows for an analysis to determine the worst-case response time. Xv6 is also not suitable for soft real-time applications, when missing a deadline occasionally is acceptable, because xv6's scheduler is too simplistic and it has kernel code path where interrupts are disabled for a long time.

5.6 Exercises

1. Modify `uart.c` to not use interrupts at all. You may need to modify `console.c` as well.
2. Add a driver for an Ethernet card.

如第一章所述，控制台对应用程序表现为普通文件，应用程序使用read和write系统调用读取输入和写入输出。应用程序可能希望控制无法通过标准文件系统调用表达的设备方面（例如，在控制台驱动程序中启用/禁用行缓冲）。Unix操作系统支持ioctl系统调用以用于此类情况。

某些计算机应用要求系统必须在有界时间内响应。例如，在安全关键系统中，错过截止时间可能导致灾难。Xv6不适用于硬实时设置。硬实时操作系统通常是与应用程序链接的库，允许进行以确定最坏情况响应时间。Xv6也不适用于软实时应用程序，偶尔错过截止时间是可接受的，因为Xv6的调度器过于简单，并且它有内核代码路径，其中中断长时间被禁用。

5.6 练习

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

Chapter 6

Locking

Most kernels, including xv6, interleave the execution of multiple activities. One source of interleaving is multiprocessor hardware: computers with multiple CPUs executing independently, such as xv6's RISC-V. These multiple CPUs share physical RAM, and xv6 exploits the sharing to maintain data structures that all CPUs read and write. This sharing raises the possibility of one CPU reading a data structure while another CPU is mid-way through updating it, or even multiple CPUs updating the same data simultaneously; without careful design such parallel access is likely to yield incorrect results or a broken data structure. Even on a uniprocessor, the kernel may switch the CPU among a number of threads, causing their execution to be interleaved. Finally, a device interrupt handler that modifies the same data as some interruptible code could damage the data if the interrupt occurs at just the wrong time. The word *concurrency* refers to situations in which multiple instruction streams are interleaved, due to multiprocessor parallelism, thread switching, or interrupts.

Kernels are full of concurrently-accessed data. For example, two CPUs could simultaneously call `kalloc`, thereby concurrently popping from the head of the free list. Kernel designers like to allow for lots of concurrency, since it can yield increased performance through parallelism, and increased responsiveness. However, as a result kernel designers must convince themselves of correctness despite such concurrency. There are many ways to arrive at correct code, some easier to reason about than others. Strategies aimed at correctness under concurrency, and abstractions that support them, are called *concurrency control* techniques.

Xv6 uses a number of concurrency control techniques, depending on the situation; many more are possible. This chapter focuses on a widely used technique: the *lock*. A lock provides mutual exclusion, ensuring that only one CPU at a time can hold the lock. If the programmer associates a lock with each shared data item, and the code always holds the associated lock when using an item, then the item will be used by only one CPU at a time. In this situation, we say that the lock protects the data item. Although locks are an easy-to-understand concurrency control mechanism, the downside of locks is that they can limit performance, because they serialize concurrent operations.

The rest of this chapter explains why xv6 needs locks, how xv6 implements them, and how it uses them.

第6章

锁定

大多数内核，包括xv6，会交错执行多个活动。交错执行的一个来源是多处理器硬件：具有多个独立执行的CPU的计算机，例如xv6的RISC-V。这些多个CPU共享物理内存，并且xv6利用这种共享来维护所有CPU读取和写入的数据结构。这种共享可能会出现一个CPU正在读取数据结构，而另一个CPU正在中途更新它的情况，甚至多个CPU同时更新相同的数据；如果没有仔细设计，这种并行访问很可能导致错误的结果或损坏的数据结构。即使在单处理器上，内核也可能在多个线程之间切换CPU，导致它们的执行交错。最后，如果中断恰好在错误的时间发生，修改相同数据的设备中断处理程序可能会损坏数据。并发一词指的是由于多处理器并行性、线程切换或中断而导致多个指令流交错的情况。

内核充满了并发访问的数据。例如，两个 CPU 可能同时调用 kalloc，从而并发地从空闲列表头部弹出。内核设计者喜欢允许大量并发，因为并行性可以带来性能提升，并提高响应能力。然而，作为结果，内核设计者必须在不考虑这种并发的情况下证明其正确性。有许多方法可以编写正确的代码，有些比其他方法更容易推理。旨在并发环境下保证正确性的策略，以及支持它们的抽象，称为并发控制技术。

Xv6 根据情况使用多种并发控制技术，还有更多可能。本章重点介绍一种广泛使用的技术：锁。锁提供互斥，确保同一时间只有一个 CPU 可以持有锁。如果程序员为每个共享数据项关联一个锁，并且在使用该数据项时始终持有关联的锁，那么该数据项将同一时间只被一个 CPU 使用。在这种情况下，我们称锁保护了数据项。尽管锁是一种易于理解的并发控制机制，但锁的缺点是它们可能会限制性能，因为它们会序列化并发操作。

本章其余部分解释了为什么xv6需要锁，它是如何实现锁的，以及它是如何使用锁的。

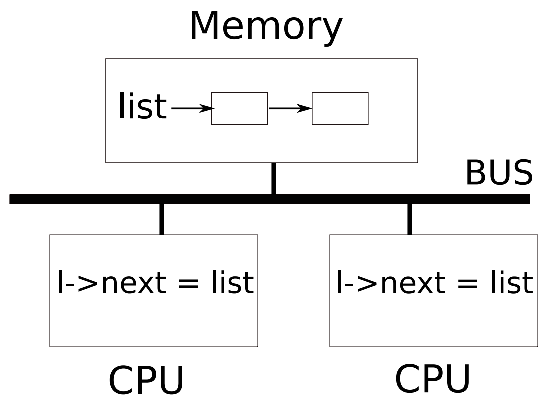

Figure 6.1: Simplified SMP architecture

6.1 Races

As an example of why we need locks, consider two processes with exited children calling `wait` on two different CPUs. `wait` frees the child's memory. Thus on each CPU, the kernel will call `kfree` to free the children's memory pages. The kernel allocator maintains a linked list: `kalloc()` (kernel/kalloc.c:69) pops a page of memory from a list of free pages, and `kfree()` (kernel/kalloc.c:47) pushes a page onto the free list. For best performance, we might hope that the `kfree`s of the two parent processes would execute in parallel without either having to wait for the other, but this would not be correct given xv6's `kfree` implementation.

Figure 6.1 illustrates the setting in more detail: the linked list of free pages is in memory that is shared by the two CPUs, which manipulate the list using load and store instructions. (In reality, the processors have caches, but conceptually multiprocessor systems behave as if there were a single, shared memory.) If there were no concurrent requests, you might implement a list `push` operation as follows:

```
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;
```

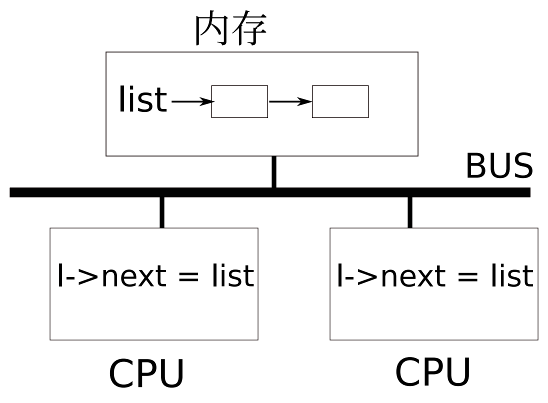

图 6.1：简化的 SMP 架构

6.1 竞争条件

以我们需要锁为例，考虑两个进程在两个不同的CPU上调用wait，并且这两个进程都有已退出的子进程。wait会释放子进程的内存。因此，在每个CPU上，内核都会调用kfree来释放子进程的内存页。内核分配器维护一个链表：kalloc()（kernel/kalloc.c:69）从空闲页面的列表中弹出一页内存，而kfree()（kernel/kalloc.c:47）将一页内存推入空闲列表。为了最佳性能，我们希望两个父进程的kfree操作能够并行执行，而无需彼此等待，但考虑到xv6的kfree实现，这是不正确的。

图6.1更详细地说明了设置：空闲页面的链表位于两个CPU共享的内存中，CPU使用加载和存储指令来操作该链表。（实际上，处理器具有缓存，但从概念上讲，多处理器系统表现得好像有一个单一的、共享的内存。）如果没有并发请求，你可以按如下方式实现列表推送操作：

```
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;
```

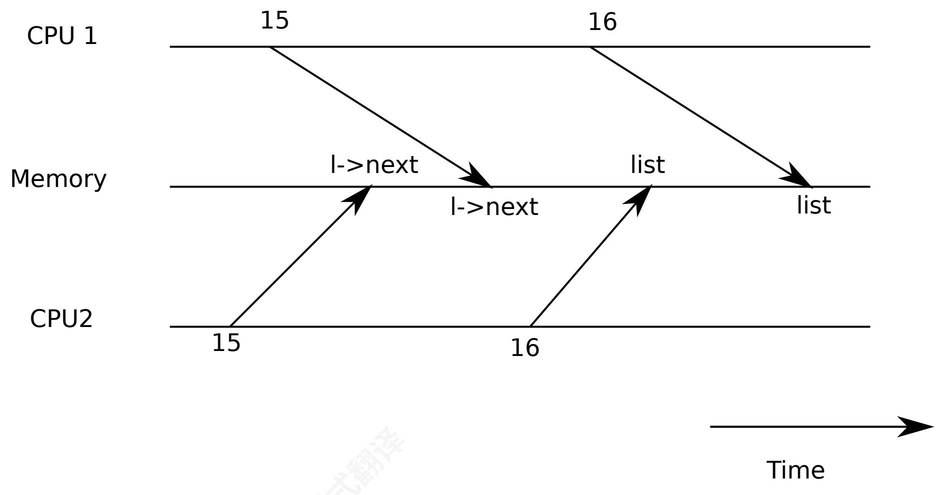

Figure 6.2: Example race

```
17      }
```

This implementation is correct if executed in isolation. However, the code is not correct if more than one copy executes concurrently. If two CPUs execute `push` at the same time, both might execute line 15 as shown in Fig 6.1, before either executes line 16, which results in an incorrect outcome as illustrated by Figure 6.2. There would then be two list elements with `next` set to the former value of `list`. When the two assignments to `list` happen at line 16, the second one will overwrite the first; the element involved in the first assignment will be lost.

The lost update at line 16 is an example of a *race*. A race is a situation in which a memory location is accessed concurrently, and at least one access is a write. A race is often a sign of a bug, either a lost update (if the accesses are writes) or a read of an incompletely-updated data structure. The outcome of a race depends on the machine code generated by the compiler, the timing of the two CPUs involved, and how their memory operations are ordered by the memory system, which can make race-induced errors difficult to reproduce and debug. For example, adding print statements while debugging `push` might change the timing of the execution enough to make the race disappear.

The usual way to avoid races is to use a lock. Locks ensure *mutual exclusion*, so that only one CPU at a time can execute the sensitive lines of `push`; this makes the scenario above impossible. The correctly locked version of the above code adds just a few lines (highlighted in yellow):

```
6       struct element *list = 0;
7       struct lock listlock;
8
9       void
10      push(int data)
11      {
12        struct element *l;
13        l = malloc(sizeof *l);
14        l->data = data;
```

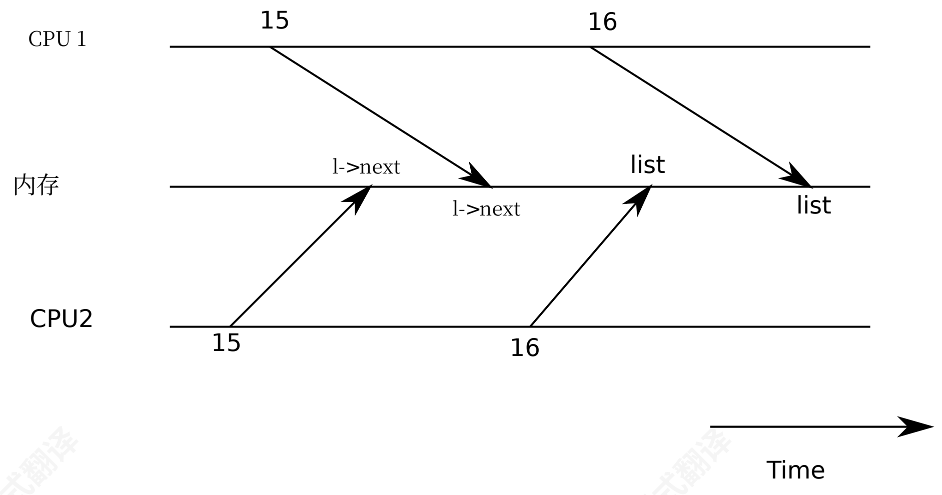

图6.2：示例竞争条件

```
17      }
```

如果该实现被隔离执行，则是正确的。但是，如果有多于一个副本并发执行，代码就不正确。如果两个CPU同时执行push，它们都可能按图6.1所示执行行15，在任何一个执行行16之前，这会导致不正确的结果，如图6.2所示。然后会有两个列表元素，它们的next被设置为列表的旧值。当两个对list的赋值发生在行16时，第二个赋值会覆盖第一个；第一个赋值涉及的元素将被丢失。

行16处的丢失更新是竞争条件的一个示例。竞争条件是一种内存位置被并发访问的情况，并且至少有一个访问是写入操作。竞争条件通常是错误的迹象，要么是丢失更新（如果访问是写入操作），要么是对未完全更新的数据结构的读取。竞争条件的后果取决于编译器生成的机器码、涉及的两个CPU的时间，以及内存系统如何对它们的内存操作进行排序，这可能导致由竞争条件引起的错误难以重现和调试。例如，在调试时添加打印语句可能会改变执行的时间，足以使竞争条件消失。

避免竞争条件的常用方法是为锁。锁确保互斥，因此一次只有一个CPU可以执行push的敏感代码行；这使得上述场景不可能。上述代码的正确加锁版本增加了一些行（黄色突出显示）：

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

The sequence of instructions between `acquire` and `release` is often called a *critical section*. The lock is typically said to be protecting `list`.

When we say that a lock protects data, we really mean that the lock protects some collection of invariants that apply to the data. Invariants are properties of data structures that are maintained across operations. Typically, an operation's correct behavior depends on the invariants being true when the operation begins. The operation may temporarily violate the invariants but must reestablish them before finishing. For example, in the linked list case, the invariant is that `list` points at the first element in the list and that each element's `next` field points at the next element. The implementation of `push` violates this invariant temporarily: in line 17, `l` points to the next list element, but `list` does not point at `l` yet (reestablished at line 18). The race we examined above happened because a second CPU executed code that depended on the list invariants while they were (temporarily) violated. Proper use of a lock ensures that only one CPU at a time can operate on the data structure in the critical section, so that no CPU will execute a data structure operation when the data structure's invariants do not hold.

You can think of a lock as *serializing* concurrent critical sections so that they run one at a time, and thus preserve invariants (assuming the critical sections are correct in isolation). You can also think of critical sections guarded by the same lock as being atomic with respect to each other, so that each sees only the complete set of changes from earlier critical sections, and never sees partially-completed updates.

Though useful for correctness, locks inherently limit performance. For example, if two processes call `kfree` concurrently, the locks will serialize the two critical sections, so that there is no benefit from running them on different CPUs. We say that multiple processes *conflict* if they want the same lock at the same time, or that the lock experiences *contention*. A major challenge in kernel design is avoidance of lock contention in pursuit of parallelism. Xv6 does little of that, but sophisticated kernels organize data structures and algorithms specifically to avoid lock contention. In the list example, a kernel may maintain a separate free list per CPU and only touch another CPU's free list if the current CPU's list is empty and it must steal memory from another CPU. Other use cases may require more complicated designs.

The placement of locks is also important for performance. For example, it would be correct to move `acquire` earlier in `push`, before line 13. But this would likely reduce performance because then the calls to `malloc` would be serialized. The section "Using locks" below provides some guidelines for where to insert `acquire` and `release` invocations.

```
15
16        acquire(&listlock);
17        l->next = list;
18        list = l;
19        release(&listlock);
20    }
```

acquire和release之间的指令序列通常称为临界区。锁通常被说是在保护list。

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一些不变式集合。不变式是数据结构在操作过程中保持的属性。通常，一个操作的正确行为取决于操作开始时不变式为真。操作可能会暂时违反不变式，但在完成前必须重新建立它们。例如，在链表的情况下，不变式是列表指向列表中的第一个元素，并且每个元素的 next 字段指向下一个元素。push 的实现暂时违反了这一不变式：在第 17 行，l 指向下一个列表元素，但 list 尚未指向 l（在第 18 行重新建立）。我们上面分析的竞争条件发生是因为第二个 CPU 在列表不变式（暂时）被违反时执行了依赖于列表不变式的代码。正确使用锁确保同一时间只有一个 CPU 可以在临界区中对数据结构进行操作，这样就不会有 CPU 在数据结构的不变式不成立时执行数据结构操作。

你可以将锁视为序列化并发临界区，以便它们一次运行一个，从而保持不变式（假设临界区在隔离状态下是正确的）。你也可以将受相同锁保护的临界区视为彼此原子，以便每个只能看到早期临界区的完整变更集，并且永远不会看到部分完成的更新。

虽然锁对正确性很有用，但它们本质上会限制性能。例如，如果两个进程并发调用 kfree，锁将序列化两个临界区，因此无法从在不同 CPU 上运行它们中获益。我们说多个进程冲突，如果它们同时想要同一个锁，或者锁经历了竞争。内核设计中的一个主要挑战是在追求并行性的同时避免锁竞争。xv6 很少这样做，但复杂的内核会专门组织数据结构和算法来避免锁竞争。在列表示例中，内核可以为每个 CPU 维护一个单独的空闲列表，并且只有当当前 CPU 的列表为空并且必须从另一个 CPU 偷窃内存时才接触另一个 CPU 的空闲列表。其他用例可能需要更复杂的设计。

锁的位置对性能也很重要。例如，将 acquire 提前到 push 中，在行 13 之前移动是正确的。但这可能会降低性能，因为那样 malloc 的调用将被序列化。下面的“使用锁”部分提供了一些关于在哪里插入 acquire 和 release 调用的指南。

6.2 Code: Locks

Xv6 has two types of locks: spinlocks and sleep-locks. We'll start with spinlocks. Xv6 represents a spinlock as a `struct spinlock` (kernel/spinlock.h:2). The important field in the structure is `locked`, a word that is zero when the lock is available and non-zero when it is held. Logically, xv6 should acquire a lock by executing code like

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

Unfortunately, this implementation does not guarantee mutual exclusion on a multiprocessor. It could happen that two CPUs simultaneously reach line 25, see that `lk->locked` is zero, and then both grab the lock by executing line 26. At this point, two different CPUs hold the lock, which violates the mutual exclusion property. What we need is a way to make lines 25 and 26 execute as an *atomic* (i.e., indivisible) step.

Because locks are widely used, multi-core processors usually provide instructions that implement an atomic version of lines 25 and 26. On the RISC-V this instruction is `amoswap r, a`. `amoswap` reads the value at the memory address `a`, writes the contents of register `r` to that address, and puts the value it read into `r`. That is, it swaps the contents of the register and the memory address. It performs this sequence atomically, using special hardware to prevent any other CPU from using the memory address between the read and the write.

Xv6's `acquire` (kernel/spinlock.c:22) uses the portable C library call `__sync_lock_test_and_set`, which boils down to the `amoswap` instruction; the return value is the old (swapped) contents of `lk->locked`. The `acquire` function wraps the swap in a loop, retrying (spinning) until it has acquired the lock. Each iteration swaps one into `lk->locked` and checks the previous value; if the previous value is zero, then we've acquired the lock, and the swap will have set `lk->locked` to one. If the previous value is one, then some other CPU holds the lock, and the fact that we atomically swapped one into `lk->locked` didn't change its value.

Once the lock is acquired, `acquire` records, for debugging, the CPU that acquired the lock. The `lk->cpu` field is protected by the lock and must only be changed while holding the lock.

The function `release` (kernel/spinlock.c:47) is the opposite of `acquire`: it clears the `lk->cpu` field and then releases the lock. Conceptually, the release just requires assigning zero to `lk->locked`. The C standard allows compilers to implement an assignment with multiple store instructions, so a C assignment might be non-atomic with respect to concurrent code. Instead, `release` uses the C library function `__sync_lock_release` that performs an atomic assignment. This function also boils down to a RISC-V `amoswap` instruction.

6.2 代码：锁

xv6 有两种类型的锁：自旋锁和睡眠锁。我们将从自旋锁开始。xv6 将自旋锁表示为 struct spinlock (kernel/spinlock.h:2)。结构中的重要字段是 locked，一个字，当锁可用时为零，当它被持有时为非零。从逻辑上讲，xv6 应该通过执行像

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

不幸的是，这种实现方式在多处理器上不能保证互斥。可能会发生两个 CPU 同时到达第 25 行，看到 lk->locked 是零，然后都通过执行第 26 行来获取锁。此时，两个不同的 CPU 都持有锁，这违反了互斥属性。我们需要的是一种方法，让第 25 行和第 26 行作为一个原子（即不可分割）的步骤来执行。

因为锁被广泛使用，多核处理器通常提供指令来实现第 25 行和第 26 行的原子版本。在 RISC-V 上，这条指令是 amoswap r, a。amoswap 读取内存地址 a 的值，将寄存器 r 的内容写入该地址，并将读取的值放入 r 中。也就是说，它交换了寄存器和内存地址的内容。它以原子序列执行这个操作，使用特殊的硬件来防止其他 CPU 在读取和写入之间使用内存地址。

Xv6的acquire（kernel/spinlock.c:22）使用可移植C库调用__sync_lock_test_和_set，这归结为amoswap指令；返回值是lk->锁定时的旧（交换）内容。acquire函数将交换包装在一个循环中，重试（自旋），直到它获取了锁。每次迭代将一个交换到lk->锁定，并检查前一个值；如果前一个值是零，那么我们已经获取了锁，并且交换将设置lk->锁定为1。如果前一个值是1，那么其他CPU持有锁，并且我们原子性地将1交换到lk->锁定的事实并没有改变它的值。

一旦获取了锁，acquire会记录，用于调试，获取锁的CPU。lk->cpu字段受锁保护，并且只能在持有锁时更改。

函数 release (kernel/spinlock.c:47) 是 acquire 的相反操作：它会清除 lk->cpu 字段，然后释放锁。从概念上讲，release 只需要将零赋值给 tolk->locked。C 标准允许编译器使用多个存储指令来实现赋值，因此 C 赋值可能相对于并发代码是非原子的。相反，release 使用 C 库函数 __sync_lock_release 来执行原子赋值。这个函数也归结为一条 RISC-V amoswap 指令。

6.3 Code: Using locks

Xv6 uses locks in many places to avoid races. As described above, `kalloc` (kernel/kalloc.c:69) and `kfree` (kernel/kalloc.c:47) form a good example. Try Exercises 1 and 2 to see what happens if those functions omit the locks. You'll likely find that it's difficult to trigger incorrect behavior, suggesting that it's hard to reliably test whether code is free from locking errors and races. Xv6 may well have as-yet-undiscovered races.

A hard part about using locks is deciding how many locks to use and which data and invariants each lock should protect. There are a few basic principles. First, any time a variable can be written by one CPU at the same time that another CPU can read or write it, a lock should be used to keep the two operations from overlapping. Second, remember that locks protect invariants: if an invariant involves multiple memory locations, typically all of them need to be protected by a single lock to ensure the invariant is maintained.

The rules above say when locks are necessary but say nothing about when locks are unnecessary, and it is important for efficiency not to lock too much, because locks reduce parallelism. If parallelism isn't important, then one could arrange to have only a single thread and not worry about locks. A simple kernel can do this on a multiprocessor by having a single lock that must be acquired on entering the kernel and released on exiting the kernel (though blocking system calls such as pipe reads or `wait` would pose a problem). Many uniprocessor operating systems have been converted to run on multiprocessors using this approach, sometimes called a "big kernel lock," but the approach sacrifices parallelism: only one CPU can execute in the kernel at a time. If the kernel does any heavy computation, it would be more efficient to use a larger set of more fine-grained locks, so that the kernel could execute on multiple CPUs simultaneously.

As an example of coarse-grained locking, xv6's `kalloc.c` allocator has a single free list protected by a single lock. If multiple processes on different CPUs try to allocate pages at the same time, each will have to wait for its turn by spinning in `acquire`. Spinning wastes CPU time, since it's not useful work. If contention for the lock wasted a significant fraction of CPU time, perhaps performance could be improved by changing the allocator design to have multiple free lists, each with its own lock, to allow truly parallel allocation.

As an example of fine-grained locking, xv6 has a separate lock for each file, so that processes that manipulate different files can often proceed without waiting for each other's locks. The file locking scheme could be made even more fine-grained if one wanted to allow processes to simultaneously write different areas of the same file. Ultimately lock granularity decisions need to be driven by performance measurements as well as complexity considerations.

As subsequent chapters explain each part of xv6, they will mention examples of xv6's use of locks to deal with concurrency. As a preview, Figure 6.3 lists all of the locks in xv6.

6.4 Deadlock and lock ordering

If a code path through the kernel must hold several locks at the same time, it is important that all code paths acquire those locks in the same order. If they don't, there is a risk of *deadlock*. Let's say two code paths in xv6 need locks A and B, but code path 1 acquires locks in the order A then

6.3 代码：使用锁定

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

使用锁定的一个难点在于决定使用多少个锁定，以及每个锁定应该保护哪些数据和不变式。有一些基本原则。首先，任何时间一个变量可以被一个CPU写入，同时另一个CPU可以读取或写入它，都应该使用锁定来防止两个操作重叠。其次，记住锁定保护不变式：如果不变式涉及多个内存位置，通常所有这些位置都需要由一个锁定来保护，以确保不变式被维护。

上述规则说明了何时需要锁定，但没有说明何时不需要锁定，并且为了效率，不应该锁定太多，因为锁定会减少并行性。如果并行性不重要，那么可以安排只有一个线程，而不必担心锁定。一个简单的内核可以通过在进入内核时必须获取一个锁定，在退出内核时释放这个锁定，在多处理器上实现这一点（尽管阻塞系统调用如管道读取或wait会存在问题）。许多单处理器操作系统已经通过这种方法转换为在多处理器上运行，有时称为“大内核锁”，但这种方法牺牲了并行性：一次只有一个CPU可以在内核中执行。如果内核进行任何重量级计算，使用更大的一组更细粒度的锁定会更有效率，这样内核就可以同时在一个以上的CPU上执行。

以粗粒度锁为例，xv6的kalloc.c分配器有一个由单个锁保护的空闲列表。如果多个进程在不同的CPU上同时尝试分配页，每个进程都必须通过在acquire.Spinning中旋转来等待其轮次。旋转浪费了CPU时间，因为它不是有用的工作。如果锁的竞争浪费了相当一部分CPU时间，也许可以通过将分配器设计更改为由多个空闲列表组成，每个空闲列表都有自己的锁，以允许真正的并行分配来提高性能。

以细粒度锁为例，xv6为每个文件都有一个单独的锁，这样操作不同文件的进程通常可以彼此不等待而继续进行。如果想要允许进程同时写入同一文件的不同区域，文件锁方案可以做得更加细粒度。最终锁粒度决策需要由性能测量以及复杂性考虑来驱动。

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

6.4 死锁和锁顺序

如果一个内核中的代码路径必须同时持有多个锁，那么所有代码路径以相同的顺序获取这些锁非常重要。如果不是这样，就有死锁的风险。假设在xv6中有两个代码路径需要锁A和B，但代码路径1以A然后...的顺序获取锁

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

Figure 6.3: Locks in xv6

B, and the other path acquires them in the order B then A. Suppose thread T1 executes code path 1 and acquires lock A, and thread T2 executes code path 2 and acquires lock B. Next T1 will try to acquire lock B, and T2 will try to acquire lock A. Both acquires will block indefinitely, because in both cases the other thread holds the needed lock, and won't release it until its acquire returns. To avoid such deadlocks, all code paths must acquire locks in the same order. The need for a global lock acquisition order means that locks are effectively part of each function's specification: callers must invoke functions in a way that causes locks to be acquired in the agreed-on order.

Xv6 has many lock-order chains of length two involving per-process locks (the lock in each `struct proc`) due to the way that `sleep` works (see Chapter 7). For example, `consoleintr` (kernel/console.c:136) is the interrupt routine which handles typed characters. When a newline arrives, any process that is waiting for console input should be woken up. To do this, `consoleintr` holds `cons.lock` while calling `wakeup`, which acquires the waiting process's lock in order to wake it up. In consequence, the global deadlock-avoiding lock order includes the rule that `cons.lock` must be acquired before any process lock. The file-system code contains xv6's longest lock chains. For example, creating a file requires simultaneously holding a lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, the disk driver's `vdisk_lock`, and the calling process's `p->lock`. To avoid deadlock, file-system code always acquires locks in the order mentioned in the previous sentence.

Honoring a global deadlock-avoiding order can be surprisingly difficult. Sometimes the lock order conflicts with logical program structure, e.g., perhaps code module M1 calls module M2, but the lock order requires that a lock in M2 be acquired before a lock in M1. Sometimes the identities of locks aren't known in advance, perhaps because one lock must be held in order to discover the identity of the lock to be acquired next. This kind of situation arises in the file system as it looks up successive components in a path name, and in the code for `wait` and `exit` as they search the table

Lock	描述
bcache.lock	保护块缓冲区缓存条目的分配
cons.lock	序列化对控制台硬件的访问，避免输出混合
ftable.lock	序列化 struct file 在文件表中的分配
itable.lock	保护内存inode条目的分配
vdisk lock_	序列化对磁盘硬件和DMA描述符队列的访问
kmem.lock	序列化 内存分配
日志锁	序列化 事务日志操作
管道的 π->锁	序列化 每个管道的操作
pid_锁	序列化 next_pid 的增量
进程的 p->锁	序列化进程状态的变化
等待锁_	帮助等待避免丢失唤醒
tickslock	序列化 ticks计数器的操作
inode 的 ip->锁	序列化对每个 inode 及其内容的操作
buf 的 b->锁	序列化对每个块缓冲区的操作

图6.3：xv6 中的锁

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要意味着锁实际上是每个函数规范的一部分：调用者必须以导致锁以商定的顺序被获取的方式调用函数。

Xv6 由于 sleep 的工作方式（参见第 7 章）导致存在许多长度为二的锁顺序链，其中涉及每个进程的锁（即每个 struct proc 中的锁）。例如，consoleintr (kernel/console.c:136) 是处理输入字符的中断例程。当收到换行符时，任何正在等待控制台输入的进程都应该被唤醒。为此，consoleintr 在调用 wakeup 时持有 cons.lock，以获取等待进程的锁来唤醒它。因此，全局死锁避免锁顺序包括规则：必须先获取 cons.lock，然后才能获取任何进程锁。文件系统代码包含 xv6 最长的锁链。例如，创建文件需要同时持有对目录的锁、新文件的 inode 锁、磁盘块缓冲区的锁、磁盘驱动程序的 vdisk_锁，以及调用进程的 p->锁。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。

遵循全局死锁避免顺序可能会出人意料地困难。有时锁顺序与逻辑程序结构冲突，例如，可能代码模块 M1 调用模块 M2，但锁顺序要求必须先获取 M2 中的锁，才能获取 M1 中的锁。有时锁的身份无法提前知道，可能是因为必须持有某个锁才能发现下一个要获取的锁的身份。这种情况会在文件系统查找路径名 successive components 时出现，以及 wait 和 exit 的代码在搜索进程 table 时出现。

of processes looking for child processes. Finally, the danger of deadlock is often a constraint on how fine-grained one can make a locking scheme, since more locks often means more opportunity for deadlock. The need to avoid deadlock is often a major factor in kernel implementation.

6.5 Re-entrant locks

It might appear that some deadlocks and lock-ordering challenges could be avoided by using *re-entrant locks*, which are also called *recursive locks*. The idea is that if the lock is held by a process and if that process attempts to acquire the lock again, then the kernel could just allow this (since the process already has the lock), instead of calling panic, as the xv6 kernel does.

It turns out, however, that re-entrant locks make it harder to reason about concurrency: re-entrant locks break the intuition that locks cause critical sections to be atomic with respect to other critical sections. Consider the following functions `f` and `g`, and a hypothetical function `h`:

```
struct spinlock lock;
int data = 0; // protected by lock

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

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

h() {
    ...
}
```

Looking at this code fragment, the intuition is that `call_once` will be called only once: either by `f`, or by `g`, but not by both.

But if re-entrant locks are allowed, and `h` happens to call `g`, `call_once` will be called *twice*.

If re-entrant locks aren't allowed, then `h` calling `g` results in a deadlock, which is not great either. But, assuming it would be a serious error to call `call_once`, a deadlock is preferable. The

最后，死锁的风险通常是对锁定方案粒度的一种约束，因为更多的锁往往意味着更多的死锁机会。避免死锁的需要通常是内核实现中的一个主要因素。

6.5 可重入锁

看起来使用可重入锁（也称为递归锁）可以避免一些死锁和锁顺序问题。其思路是：如果锁被一个进程持有，并且该进程尝试再次获取该锁，那么内核可以允许这样做（因为该进程已经持有锁），而不是像 xv6 内核那样调用 panic。

然而，可重入锁使得并发推理更加困难：可重入锁破坏了锁导致临界区相对于其他临界区具有原子性的直觉。考虑以下函数 f 和 g，以及一个假设的函数 h：

```
struct spinlock lock;
int data = 0; // protected by lock

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

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

h() {
    ...
}
```

查看这段代码片段，直觉是 call_once 会被调用一次：要么由 f 调用，要么由 g 调用，但不会同时被两者调用。

但如果允许使用可重入锁，并且 h 恰好调用 g，那么 call_once 会被调用两次。

如果不可重入锁不被允许，那么 h 调用 g 会导致死锁，这也不太好。但是，假设调用 call_ 一次就是严重的错误，那么死锁是更可取的。The kernel developer will observe the deadlock (the kernel panics) and can fix the code to avoid it, while calling `call_once` twice may silently result in an error that is difficult to track down.

For this reason, xv6 uses the simpler to understand non-re-entrant locks. As long as programmers keep the locking rules in mind, however, either approach can be made to work. If xv6 were to use re-entrant locks, one would have to modify `acquire` to notice that the lock is currently held by the calling thread. One would also have to add a count of nested acquires to struct spinlock, in similar style to `push_off`, which is discussed next.

6.6 Locks and interrupt handlers

Some xv6 spinlocks protect data that is used by both threads and interrupt handlers. For example, the `clockintr` timer interrupt handler might increment `ticks` (kernel/trap.c:164) at about the same time that a kernel thread reads `ticks` in `sys_sleep` (kernel/sysproc.c:61). The lock `tickslock` serializes the two accesses.

The interaction of spinlocks and interrupts raises a potential danger. Suppose `sys_sleep` holds `tickslock`, and its CPU is interrupted by a timer interrupt. `clockintr` would try to acquire `tickslock`, see it was held, and wait for it to be released. In this situation, `tickslock` will never be released: only `sys_sleep` can release it, but `sys_sleep` will not continue running until `clockintr` returns. So the CPU will deadlock, and any code that needs either lock will also freeze.

To avoid this situation, if a spinlock is used by an interrupt handler, a CPU must never hold that lock with interrupts enabled. Xv6 is more conservative: when a CPU acquires any lock, xv6 always disables interrupts on that CPU. Interrupts may still occur on other CPUs, so an interrupt's `acquire` can wait for a thread to release a spinlock; just not on the same CPU.

Xv6 re-enables interrupts when a CPU holds no spinlocks; it must do a little book-keeping to cope with nested critical sections. `acquire` calls `push_off` (kernel/spinlock.c:89) and `release` calls `pop_off` (kernel/spinlock.c:100) to track the nesting level of locks on the current CPU. When that count reaches zero, `pop_off` restores the interrupt enable state that existed at the start of the outermost critical section. The `intr_off` and `intr_on` functions execute RISC-V instructions to disable and enable interrupts, respectively.

It is important that `acquire` call `push_off` strictly before setting `lk->locked` (kernel/spinlock.c:28). If the two were reversed, there would be a brief window when the lock was held with interrupts enabled, and an unfortunately timed interrupt would deadlock the system. Similarly, it is important that `release` call `pop_off` only after releasing the lock (kernel/spinlock.c:66).

6.7 Instruction and memory ordering

It is natural to think of programs executing in the order in which source code statements appear. That's a reasonable mental model for single-threaded code, but is incorrect when multiple threads interact through shared memory [2, 4]. One reason is that compilers emit load and store instructions in orders different from those implied by the source code, and may entirely omit them (for example by caching data in registers). Another reason is that the CPU may execute instructions out of order

内核开发者会观察到死锁（内核崩溃）并可以修复代码来避免它，而调用 call_ 一次两次可能会静默地导致一个难以追踪的错误。

因此，xv6 使用更容易理解的不可重入锁。只要程序员记住锁定规则，不过，两种方法都可以使其工作。如果 xv6 使用可重入锁，则必须修改 acquire 以注意到锁当前由调用线程持有。还必须在 struct spinlock 中添加嵌套 acquire 的计数，类似于接下来讨论的 push_off。

6.6 锁和中断处理程序

一些 xv6 自旋锁保护由线程和中断处理程序共同使用的数据。例如，clockintr 定时器中断处理程序可能在内核线程在 sys_sleep 中读取 ticks（kernel/trap.c:164）大约相同的时间增加 ticks。locks tickslock 对这两种访问进行序列化。

自旋锁和中断的交互引发了一个潜在的危险。假设 sys_sleep 持有 tickslock，并且其 CPU 被定时器中断中断。clockintr 会尝试获取 tickslock，看到它被持有，并等待它被释放。在这种情况下，tickslock 将永远不会被释放：只有 sys_sleep 可以释放它，但 sys_sleep 将不会继续运行，直到 clockintr 返回。因此，CPU 将死锁，任何需要这两种锁的代码也会冻结。

为了避免这种情况，如果一个中断处理程序使用了自旋锁，那么一个 CPU 绝不能在启用中断的情况下持有该锁。Xv6 更加保守：当一个 CPU 获取任何锁时，xv6 总是在该 CPU 上禁用中断。中断仍然可能发生在其他 CPU 上，所以一个中断的获取可以等待一个线程释放自旋锁；只是不能在同一个 CPU 上。

当 CPU 没有持有任何自旋锁时，Xv6 会重新启用中断；它必须做一点账目管理来处理嵌套的临界区。acquire 调用 push_off (kernel/spinlock.c:89) 而 release 调用 pop_off (kernel/spinlock.c:100) 来跟踪当前 CPU 上锁的嵌套级别。当计数达到零时，pop_off 恢复在最外层临界区开始时存在的中断启用状态。intr off 和 intr on 函数分别执行 RISC-V 指令来 _ _禁用和启用中断。

在设置lk->锁定之前，acquire调用必须严格推入_off（kernel/spinlock.c:28）。如果顺序颠倒，会出现一个锁被持有且中断启用的短暂窗口，而一个不幸时机到来的中断会死锁系统。类似地，在释放锁之后，release调用才应该弹出_off（kernel/spinlock.c:66）。

6.7 指令和内存排序

人们自然地认为程序按照源代码语句出现的顺序执行。这是单线程代码的一个合理心智模型，但在多个线程通过共享内存 [2, 4] 交互时是不正确的。一个原因是编译器生成的加载和存储指令的顺序与源代码隐含的顺序不同，并且可能会完全省略它们（例如通过将数据缓存到寄存器中）。另一个原因是 CPU 可能会为了提高性能而乱序执行指令

to increase performance. For example, a CPU may notice that in a serial sequence of instructions A and B are not dependent on each other. The CPU may start instruction B first, either because its inputs are ready before A's inputs, or in order to overlap execution of A and B.

As an example of what could go wrong, in this code for `push`, it would be a disaster if the compiler or CPU moved the store corresponding to line 4 to a point after the `release` on line 6:

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

If such a re-ordering occurred, there would be a window during which another CPU could acquire the lock and observe the updated `list`, but see an uninitialized `list->next`.

The good news is that compilers and CPUs help concurrent programmers by following a set of rules called the *memory model*, and by providing some primitives to help programmers control re-ordering.

To tell the hardware and compiler not to re-order, xv6 uses `__sync_synchronize()` in both `acquire` (kernel/spinlock.c:22) and `release` (kernel/spinlock.c:47). `__sync_synchronize()` is a *memory barrier*: it tells the compiler and CPU to not reorder loads or stores across the barrier. The barriers in xv6's `acquire` and `release` force order in almost all cases where it matters, since xv6 uses locks around accesses to shared data. Chapter 9 discusses a few exceptions.

6.8 Sleep locks

Sometimes xv6 needs to hold a lock for a long time. For example, the file system (Chapter 8) keeps a file locked while reading and writing its content on the disk, and these disk operations can take tens of milliseconds. Holding a spinlock that long would lead to waste if another process wanted to acquire it, since the acquiring process would waste CPU for a long time while spinning. Another drawback of spinlocks is that a process cannot yield the CPU while retaining a spinlock; we'd like to do this so that other processes can use the CPU while the process with the lock waits for the disk. Yielding while holding a spinlock is illegal because it might lead to deadlock if a second thread then tried to acquire the spinlock; since `acquire` doesn't yield the CPU, the second thread's spinning might prevent the first thread from running and releasing the lock. Yielding while holding a lock would also violate the requirement that interrupts must be off while a spinlock is held. Thus we'd like a type of lock that yields the CPU while waiting to acquire, and allows yields (and interrupts) while the lock is held.

Xv6 provides such locks in the form of *sleep-locks*. `acquiresleep` (kernel/sleeplock.c:22) yields the CPU while waiting, using techniques that will be explained in Chapter 7. At a high level, a sleep-lock has a `locked` field that is protected by a spinlock, and `acquiresleep` 's call to `sleep` atomically yields the CPU and releases the spinlock. The result is that other threads can execute while `acquiresleep` waits.

。例如，CPU 可能会注意到在一系列串行指令 A 和 B 中它们之间没有依赖关系。CPU 可能会先开始执行指令 B，要么是因为它的输入在 A 的输入之前就准备好了，要么是为了重叠 A 和 B 的执行。

以示例说明可能出错的情况，在这个 push 代码中，如果编译器或 CPU 将对应第 4 行的存储操作移动到第 6 行的 release 之后，那将是灾难性的：

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

如果发生这种重排序，会有一个窗口期，另一个 CPU 可能会获取锁并观察到更新的 list，但看到未初始化的 list->next。

好消息是，编译器和 CPU 通过遵循一套称为内存模型的规则，并提供一些原语来帮助程序员控制重排序，来帮助并发程序员。

为了告诉硬件和编译器不要重排序，xv6 在 acquire (kernel/spinlock.c:22) 和 release (kernel/spinlock.c:47) 中都使用 __sync_synchronize()。__sync_synchronize() 是一个内存屏障：它告诉编译器和 CPU 不要跨过屏障重排序加载或存储。xv6 的 acquire 和 release 中的屏障在几乎所有重要情况下都强制顺序，因为 xv6 在访问共享数据时使用锁。第 9 章讨论了一些例外。

6.8 睡眠锁

有时 xv6 需要长时间持有锁。例如，文件系统（第 8 章）在磁盘上读取和写入其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它，长时间持有自旋锁会导致浪费，因为获取进程会花费很长时间进行自旋。自旋锁的另一个缺点是，进程在持有自旋锁时不能让出 CPU；我们希望这样，以便其他进程可以在锁等待磁盘时使用 CPU。在持有自旋锁时让出 CPU 是非法的，因为如果第二个线程尝试获取自旋锁，这可能会导致死锁；由于 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。在持有锁时让出 CPU 也会违反自旋锁被持有时中断必须关闭的要求。因此，我们需要一种锁，它在等待获取时让出 CPU，并且在持有锁时允许让出（和中断）。

Xv6 以睡眠锁的形式提供这些锁。acquiresleep (kernel/sleeplock.c:22) 在等待时让出 CPU，使用将在第 7 章中解释的技术。从高层次来看，一个睡眠锁有一个由自旋锁保护的锁定字段，而 acquiresleep 对 sleep 的调用会原子性地让出 CPU 并释放自旋锁。结果是，其他线程可以在 acquiresleep 等待时执行。

Because sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers. Because `acquiresleep` may yield the CPU, sleep-locks cannot be used inside spinlock critical sections (though spinlocks can be used inside sleep-lock critical sections).

Spin-locks are best suited to short critical sections, since waiting for them wastes CPU time; sleep-locks work well for lengthy operations.

6.9 Real world

Programming with locks remains challenging despite years of research into concurrency primitives and parallelism. It is often best to conceal locks within higher-level constructs like synchronized queues, although xv6 does not do this. If you program with locks, it is wise to use a tool that attempts to identify races, because it is easy to miss an invariant that requires a lock.

Most operating systems support POSIX threads (Pthreads), which allow a user process to have several threads running concurrently on different CPUs. Pthreads has support for user-level locks, barriers, etc. Pthreads also allows a programmer to optionally specify that a lock should be re-entrant.

Supporting Pthreads at user level requires support from the operating system. For example, it should be the case that if one pthread blocks in a system call, another pthread of the same process should be able to run on that CPU. As another example, if a pthread changes its process's address space (e.g., maps or unmaps memory), the kernel must arrange that other CPUs that run threads of the same process update their hardware page tables to reflect the change in the address space.

It is possible to implement locks without atomic instructions [10], but it is expensive, and most operating systems use atomic instructions.

Locks can be expensive if many CPUs try to acquire the same lock at the same time. If one CPU has a lock cached in its local cache, and another CPU must acquire the lock, then the atomic instruction to update the cache line that holds the lock must move the line from the one CPU's cache to the other CPU's cache, and perhaps invalidate any other copies of the cache line. Fetching a cache line from another CPU's cache can be orders of magnitude more expensive than fetching a line from a local cache.

To avoid the expenses associated with locks, many operating systems use lock-free data structures and algorithms [6, 12]. For example, it is possible to implement a linked list like the one in the beginning of the chapter that requires no locks during list searches, and one atomic instruction to insert an item in a list. Lock-free programming is more complicated, however, than programming locks; for example, one must worry about instruction and memory reordering. Programming with locks is already hard, so xv6 avoids the additional complexity of lock-free programming.

6.10 Exercises

1. Comment out the calls to `acquire` and `release` in `kalloc` (kernel/kalloc.c:69). This seems like it should cause problems for kernel code that calls `kalloc`; what symptoms do you expect to see? When you run xv6, do you see these symptoms? How about when running

由于睡眠锁会保持中断启用，因此它们不能用于中断处理程序。因为 acquiresleep 可能会让出 CPU，所以睡眠锁不能用于自旋锁临界区内部（尽管自旋锁可以用于睡眠锁临界区内部）。

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

6.9 现实世界

尽管对并发原语和并行性进行了多年的研究，使用锁编程仍然具有挑战性。通常最好将锁隐藏在高级结构（如同步队列）中，尽管 xv6 没有这样做。如果你使用锁编程，最好使用一个尝试识别竞争条件的工具，因为很容易错过需要锁的不变量。

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

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

使用原子指令以外的锁实现是可能的 [10],，但这很昂贵，大多数操作系统使用原子指令。

如果许多 CPU 同时尝试获取相同的锁，锁可能会很昂贵。如果一个 CPU 在其本地缓存中缓存了锁，而另一个 CPU 必须获取锁，那么必须移动持有锁的缓存行，将行从一个 CPU 的缓存移动到另一个 CPU 的缓存，并可能使缓存行的任何其他副本无效。从另一个 CPU 的缓存中获取缓存行可能比从本地缓存中获取行贵几个数量级。

为了避免与锁相关的开销，许多操作系统使用无锁数据结构和算法 [6, 12]。例如，可以实现像本章开头那样的链表，它在列表搜索期间不需要锁，并且只需一条原子指令即可在列表中插入一个项目。然而，无锁编程比锁编程更复杂；例如，必须担心指令和内存重排序。使用锁编程已经很难了，所以 xv6 避免了无锁编程的额外复杂性。

6.10 练习

1. 在 kalloc (kernel/kalloc.c:69) 中注释掉 acquire 和 release 的调用。这似乎应该会导致调用 kalloc 的内核代码出现问题；你期望看到什么症状？当你运行 xv6 时，你会看到这些症状吗？运行时呢

usertests? If you don't see a problem, why not? See if you can provoke a problem by inserting dummy loops into the critical section of kalloc.

2. Suppose that you instead commented out the locking in kfree (after restoring locking in kalloc). What might now go wrong? Is lack of locks in kfree less harmful than in kalloc?

3. If two CPUs call kalloc at the same time, one will have to wait for the other, which is bad for performance. Modify kalloc.c to have more parallelism, so that simultaneous calls to kalloc from different CPUs can proceed without waiting for each other.

4. Write a parallel program using POSIX threads, which is supported on most operating systems. For example, implement a parallel hash table and measure if the number of puts/gets scales with increasing number of CPUs.

5. Implement a subset of Pthreads in xv6. That is, implement a user-level thread library so that a user process can have more than 1 thread and arrange that these threads can run in parallel on different CPUs. Come up with a design that correctly handles a thread making a blocking system call and changing its shared address space.

用户测试？如果你看不到问题，为什么不呢？尝试通过在 kalloc 的临界区插入虚拟循环来引发一个问题。

2. 假设你取消了对 kfree 中的锁定（在 kalloc 中恢复锁定之后）。现在可能会发生什么问题？在 kfree 中缺少锁是否比在 kalloc 中更不严重？

3. 如果两个 CPU 同时调用 kalloc，其中一个必须等待另一个，这对性能不利。修改 kalloc.c 以增加并行性，以便来自不同 CPU 的 kalloc 并发调用可以彼此不等待地继续进行。

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

5. 在 xv6 中实现 Pthreads 的一部分。也就是说，实现一个用户级线程库，以便用户进程可以有多个线程，并安排这些线程可以在不同的 CPU 上并行运行。提出一个设计，该设计可以正确处理线程执行阻塞系统调用并更改其共享地址空间的情况。

Chapter 7

Scheduling

Any operating system is likely to run with more processes than the computer has CPUs, so a plan is needed to time-share the CPUs among the processes. Ideally the sharing would be transparent to user processes. A common approach is to provide each process with the illusion that it has its own virtual CPU by *multiplexing* the processes onto the hardware CPUs. This chapter explains how xv6 achieves this multiplexing.

7.1 Multiplexing

Xv6 multiplexes by switching each CPU from one process to another in two situations. First, xv6's `sleep` and `wakeup` mechanism switches when a process makes a system call that blocks (has to wait for an event), typically in `read`, `wait`, or `sleep`. Second, xv6 periodically forces a switch to cope with processes that compute for long periods without blocking. The former are voluntary switches; the latter are called involuntary. This multiplexing creates the illusion that each process has its own CPU.

Implementing multiplexing poses a few challenges. First, how to switch from one process to another? The basic idea is to save and restore CPU registers, though the fact that this cannot be expressed in C makes it tricky. Second, how to force switches in a way that is transparent to user processes? Xv6 uses the standard technique in which a hardware timer's interrupts drive context switches. Third, all of the CPUs switch among the same set of processes, so a locking plan is necessary to avoid races. Fourth, a process's memory and other resources must be freed when the process exits, but it cannot do all of this itself because (for example) it can't free its own kernel stack while still using it. Fifth, each CPU of a multi-core machine must remember which process it is executing so that system calls affect the correct process's kernel state. Finally, `sleep` and `wakeup` allow a process to give up the CPU and wait to be woken up by another process or interrupt. Care is needed to avoid races that result in the loss of wakeup notifications.

第7章

调度

任何操作系统都可能运行比计算机拥有的CPU更多的进程，因此需要一种计划来在进程之间分时共享CPU。理想情况下，共享对用户进程应该是透明的。一种常见的方法是通过多路复用将进程映射到硬件CPU上，从而给每个进程提供它拥有自己的虚拟CPU的错觉。本章解释了xv6如何实现这种多路复用。

7.1 多路复用

Xv6通过在两种情况下切换每个CPU从进程到另一个进程来多路复用。首先，当进程执行一个会阻塞（必须等待事件）的系统调用时，xv6的睡眠和唤醒机制会切换，这通常发生在读取、等待或睡眠操作中。其次，xv6定期强制切换以应对那些长时间计算而不阻塞的进程。前者是自愿切换；后者被称为非自愿切换。这种多路复用创造了每个进程拥有自己的CPU的错觉。

实现多路复用会带来一些挑战。首先，如何从一个进程切换到另一个进程？基本思路是保存和恢复 CPU 寄存器，但由于这无法用 C 语言表达，因此比较棘手。其次，如何以对用户进程透明的方式强制切换？Xv6 使用标准技术，即硬件计时器的中断驱动上下文切换。第三，所有 CPU 都在相同的一组进程之间切换，因此需要锁定计划来避免竞争条件。第四，当进程退出时，其内存和其他资源必须被释放，但它不能自己完成所有这些操作，因为（例如）它不能在仍在使用时释放自己的内核栈。第五，多核机器的每个 CPU 必须记住它正在执行的进程，以便系统调用影响正确的进程的内核状态。最后，睡眠和唤醒允许进程放弃 CPU 并等待另一个进程或中断唤醒它。需要小心避免导致唤醒通知丢失的竞争条件。

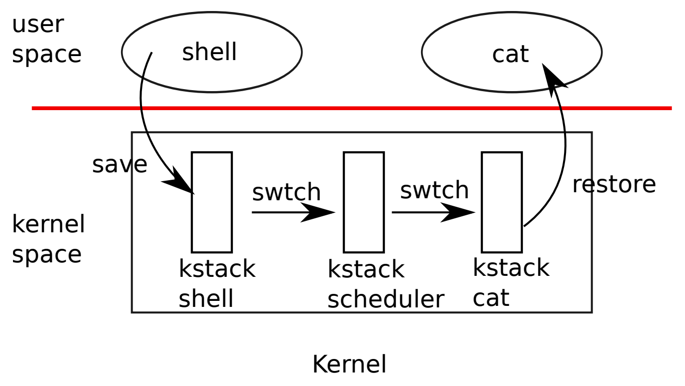

Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

7.2 Code: Context switching

Figure 7.1 outlines the steps involved in switching from one user process to another: a trap (system call or interrupt) from user space to the old process's kernel thread, a context switch to the current CPU's scheduler thread, a context switch to a new process's kernel thread, and a trap return to the user-level process. Xv6 has separate threads (saved registers and stacks) in which to execute the scheduler because it is not safe for the scheduler to execute on any process's kernel stack: some other CPU might wake the process up and run it, and it would be a disaster to use the same stack on two different CPUs. There is a separate scheduler thread for each CPU to cope with situations in which more than one CPU is running a process that wants to give up the CPU. In this section we'll examine the mechanics of switching between a kernel thread and a scheduler thread.

Switching from one thread to another involves saving the old thread's CPU registers, and restoring the previously-saved registers of the new thread; the fact that the stack pointer and program counter are saved and restored means that the CPU will switch stacks and switch what code it is executing.

The function `swtch` saves and restores registers for a kernel thread switch. `swtch` doesn't directly know about threads; it just saves and restores sets of RISC-V registers, called *contexts*. When it is time for a process to give up the CPU, the process's kernel thread calls `swtch` to save its own context and restore the scheduler's context. Each context is contained in a `struct context` (kernel/proc.h:2), itself contained in a process's `struct proc` or a CPU's `struct cpu`. `swtch` takes two arguments: `struct context *old` and `struct context *new`. It saves the current registers in `old`, loads registers from `new`, and returns.

Let's follow a process through `swtch` into the scheduler. We saw in Chapter 4 that one possibility at the end of an interrupt is that `usertrap` calls `yield`. `yield` in turn calls `sched`, which calls `swtch` to save the current context in `p->context` and switch to the scheduler context previously saved in `cpu->context` (kernel/proc.c:506).

`swtch` (kernel/swtch.S:3) saves only callee-saved registers; the C compiler generates code in the caller to save caller-saved registers on the stack. `swtch` knows the offset of each register's field in `struct context`. It does not save the program counter. Instead, `swtch` saves the `ra` register,

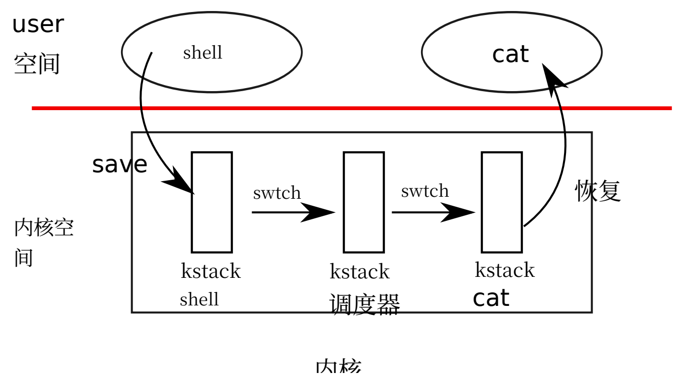

图 7.1：从一个用户进程切换到另一个用户进程。在这个示例中，xv6 使用一个 CPU（因此只有一个调度线程）运行。

7.2 代码：上下文切换

图 7.1 概述了从一个用户进程切换到另一个进程所涉及的步骤：从用户空间到旧进程的内核线程的陷阱（系统调用或中断）、到当前 CPU 的调度器线程的上下文切换、到新进程的内核线程的上下文切换，以及返回到用户级进程的陷阱返回。Xv6 有单独的线程（保存的寄存器和栈）来执行调度器，因为调度器在任何进程的内核栈上执行是不安全的：其他 CPU 可能唤醒该进程并运行它，而在两个不同的 CPU 上使用相同的栈将是一场灾难。每个 CPU 都有一个单独的调度器线程，以应对多个 CPU 正在运行希望放弃 CPU 的进程的情况。在本节中，我们将检查内核线程和调度器线程之间切换的机制。

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

函数 swtch 保存和恢复寄存器以进行内核线程切换。swtch 不会直接了解线程；它只是保存和恢复一组 RISC-V 寄存器，称为上下文。当进程需要放弃 CPU 时，进程的内核线程调用 swtch 保存其自己的上下文并恢复调度器的上下文。每个上下文都包含在一个 struct context (kernel/proc.h:2) 中，该 struct context 本身包含在进程的 struct proc 或 CPU 的 struct cpu 中。swtch 接受两个参数：struct context *old 和 struct context *new。它将当前寄存器保存到 old 中，从 new 加载寄存器，并返回。

让我们跟随一个进程通过 swtch 进入调度器。我们在第4章中看到，中断结束时的一种可能性是 usertrap 调用 yield。yield 反过来调用 sched，它调用 swtch 以在 p->上下文中保存当前上下文，并切换到先前保存在 cpu->上下文中的调度器上下文（kernel/proc.c:506）。

swtch (内核/swtch.S:3) 仅保存被调用者保存的寄存器；C 编译器在调用者中生成代码以在栈上保存调用者保存的寄存器。swtch 知道每个寄存器字段在 struct context 中的偏移量。它不保存程序计数器。相反，swtch 保存 ra 寄存器，

which holds the return address from which swtch was called. Now swtch restores registers from the new context, which holds register values saved by a previous swtch. When swtch returns, it returns to the instructions pointed to by the restored ra register, that is, the instruction from which the new thread previously called swtch. In addition, it returns on the new thread's stack, since that's where the restored sp points.

In our example, sched called swtch to switch to cpu->context, the per-CPU scheduler context. That context was saved at the point in the past when scheduler called swtch (kernel/proc.c:466) to switch to the process that's now giving up the CPU. When the swtch we have been tracing returns, it returns not to sched but to scheduler, with the stack pointer in the current CPU's scheduler stack.

7.3 Code: Scheduling

The last section looked at the low-level details of swtch; now let's take swtch as a given and examine switching from one process's kernel thread through the scheduler to another process. The scheduler exists in the form of a special thread per CPU, each running the scheduler function. This function is in charge of choosing which process to run next. A process that wants to give up the CPU must acquire its own process lock p->lock, release any other locks it is holding, update its own state (p->state), and then call sched. You can see this sequence in yield (kernel/proc.c:512), sleep and exit. sched double-checks some of those requirements (kernel/proc.c:496-501) and then checks an implication: since a lock is held, interrupts should be disabled. Finally, sched calls swtch to save the current context in p->context and switch to the scheduler context in cpu->context. swtch returns on the scheduler's stack as though scheduler's swtch had returned (kernel/proc.c:466). The scheduler continues its for loop, finds a process to run, switches to it, and the cycle repeats.

We just saw that xv6 holds p->lock across calls to swtch: the caller of swtch must already hold the lock, and control of the lock passes to the switched-to code. This arrangement is unusual: it's more common for the thread that acquires a lock to also release it. Xv6's context switching must break this convention because p->lock protects invariants on the process's state and context fields that are not true while executing in swtch. For example, if p->lock were not held during swtch, a different CPU might decide to run the process after yield had set its state to RUNNABLE, but before swtch caused it to stop using its own kernel stack. The result would be two CPUs running on the same stack, which would cause chaos. Once yield has started to modify a running process's state to make it RUNNABLE, p->lock must remain held until the invariants are restored: the earliest correct release point is after scheduler (running on its own stack) clears c->proc. Similarly, once scheduler starts to convert a RUNNABLE process to RUNNING, the lock cannot be released until the process's kernel thread is completely running (after the swtch, for example in yield).

The only place a kernel thread gives up its CPU is in sched, and it always switches to the same location in scheduler, which (almost) always switches to some kernel thread that previously called sched. Thus, if one were to print out the line numbers where xv6 switches threads, one would observe the following simple pattern: (kernel/proc.c:466), (kernel/proc.c:506),

该寄存器保存了调用 swtch 的返回地址。现在 swtch 从新上下文中恢复寄存器，该上下文保存了先前 swtch 保存的寄存器值。当 swtch 返回时，它返回到由恢复的 ra 寄存器指向的指令，即新线程先前调用 swtch 的指令。此外，它在新线程的栈上返回，因为恢复的 sp 指向那里。

在我们的示例中，调度器调用了 swtch 以切换到 cpu->上下文，即每个 CPU 的调度器上下文。该上下文是在调度器调用 swtch（kernel/proc.c:466）以切换到当前正在释放 CPU 的进程的那个过去的时刻保存的。当我们在追踪的 swtch 返回时，它返回的不是调度器，而是调度器，当前 CPU 的调度器栈中的栈指针。

7.3 代码：调度

上一节介绍了 swtch 的低级细节；现在让我们将 swtch 视为既定，并检查从某个进程的内核线程通过调度器切换到另一个进程。调度器以每个 CPU 的特殊线程的形式存在，每个线程都在运行调度器函数。该函数负责选择下一个要运行的进程。一个想要释放 CPU 的进程必须获取自己的进程锁 p->锁，释放它持有的任何其他锁，更新自己的状态（p->状态），然后调用调度器。你可以在 yield（kernel/proc.c:512）、睡眠和退出中看到这个序列。调度器会再次检查其中一些要求（kernel/proc.c:496-501），然后检查一个推论：由于持有锁，中断应该被禁用。最后，调度器调用 swtch 以在 p->上下文中保存当前上下文，并切换到 cpu->上下文中的调度器上下文。swtch 像调度器的 swtch 返回一样返回调度器的栈上（kernel/proc.c:466）。调度器继续它的 for 循环，找到一个要运行的进程，切换到它，然后循环重复。

我们刚刚看到 xv6 在 swtch 的调用中持有 p->锁：swtch 的调用者必须已经持有该锁，并且锁的控制权会传递给切换到的代码。这种安排不常见：通常情况下，获取锁的线程也会释放它。xv6 的上下文切换必须打破这一惯例，因为 p->锁保护着进程状态和上下文字段中的不变式，而这些字段在执行 swtch 时并不成立。例如，如果在 swtch 期间没有持有 p->锁，另一个 CPU 可能会在 yield 将其状态设置为 RUNNABLE 之后、但在 swtch 使其停止使用自己的内核栈之前决定运行该进程。结果将是两个 CPU 在同一个栈上运行，这将导致混乱。一旦 yield 开始修改正在运行进程的状态使其变为 RUNNABLE，p->锁必须保持持有，直到不变式被恢复：最早的正确释放点是调度器（在其自己的栈上运行）清除 c->proc 之后。类似地，一旦调度器开始将 RUNNABLE 进程转换为 RUNNING，锁不能释放，直到进程的内核线程完全运行（例如在 yield 之后，通过 swtch）。

内核线程只有在 sched 中才会放弃其 CPU，并且它总是切换到调度器的同一个位置，而调度器（几乎）总是切换到之前调用过 sched 的某个内核线程。因此，如果有人打印出 xv6 切换线程的行号，就会观察到以下简单模式：(kernel/proc.c:466), (kernel/proc.c:506), (kernel/proc.c:466), (kernel/proc.c:506), and so on. Procedures that intentionally transfer control to each other via thread switch are sometimes referred to as *coroutines*; in this example, `sched` and `scheduler` are co-routines of each other.

There is one case when the scheduler's call to `swtch` does not end up in `sched`. `allocproc` sets the context `ra` register of a new process to `forkret` (kernel/proc.c:524), so that its first `swtch` "returns" to the start of that function. `forkret` exists to release the `p->lock`; otherwise, since the new process needs to return to user space as if returning from `fork`, it could instead start at `usertrapret`.

`scheduler` (kernel/proc.c:445) runs a loop: find a process to run, run it until it yields, repeat. The scheduler loops over the process table looking for a runnable process, one that has `p->state == RUNNABLE`. Once it finds a process, it sets the per-CPU current process variable `c->proc`, marks the process as `RUNNING`, and then calls `swtch` to start running it (kernel/proc.c:461-466).

7.4 Code: mycpu and myproc

Xv6 often needs a pointer to the current process's `proc` structure. On a uniprocessor one could have a global variable pointing to the current `proc`. This doesn't work on a multi-core machine, since each CPU executes a different process. The way to solve this problem is to exploit the fact that each CPU has its own set of registers.

While a given CPU is executing in the kernel, xv6 ensures that the CPU's `tp` register always holds the CPU's hartid. RISC-V numbers its CPUs, giving each a unique *hartid*. `mycpu` (kernel/proc.c:74) uses `tp` to index an array of `cpu` structures and return the one for the current CPU. A `struct cpu` (kernel/proc.h:22) holds a pointer to the `proc` structure of the process currently running on that CPU (if any), saved registers for the CPU's scheduler thread, and the count of nested spinlocks needed to manage interrupt disabling.

Ensuring that a CPU's `tp` holds the CPU's hartid is a little involved, since user code is free to modify `tp`. `start` sets the `tp` register early in the CPU's boot sequence, while still in machine mode (kernel/start.c:45). `usertrapret` saves `tp` in the trampoline page, in case user code modifies it. Finally, `uservec` restores that saved `tp` when entering the kernel from user space (kernel/trampoline.S:78). The compiler guarantees never to modify `tp` in kernel code. It would be more convenient if xv6 could ask the RISC-V hardware for the current hartid whenever needed, but RISC-V allows that only in machine mode, not in supervisor mode.

The return values of `cpuid` and `mycpu` are fragile: if the timer were to interrupt and cause the thread to yield and later resume execution on a different CPU, a previously returned value would no longer be correct. To avoid this problem, xv6 requires that callers disable interrupts, and only enable them after they finish using the returned `struct cpu`.

The function `myproc` (kernel/proc.c:83) returns the `struct proc` pointer for the process that is running on the current CPU. `myproc` disables interrupts, invokes `mycpu`, fetches the current process pointer (`c->proc`) out of the `struct cpu`, and then enables interrupts. The return value of `myproc` is safe to use even if interrupts are enabled: if a timer interrupt moves the calling process to a different CPU, its `struct proc` pointer will stay the same.

(kernel/proc.c:466), (kernel/proc.c:506), 等等。通过线程切换有意地将控制权传递给彼此的进程有时被称为协程；在这个例子中，sched 和 scheduler 是彼此的协程。

有一个情况，调度器的 swtch 调用没有结束在 sched 中。allocproc 将新进程的上下文 ra 寄存器设置为 forkret (kernel/proc.c:524)，以便其第一个 swtch "返回"到该函数的开始处。forkret 存在是为了释放 p-> 锁；否则，由于新进程需要像从 fork 返回一样返回到用户空间，它可能会从 usertrapret 开始。

调度器 (kernel/proc.c:445) 运行一个循环：找到要运行的过程，运行它直到它 yield，重复。调度器遍历进程表，寻找一个可运行的过程，即具有 p-> 状态 == RUNNABLE 的过程。一旦找到过程，它设置每个 CPU 的当前过程变量 c->proc，将过程标记为 RUNNING，然后调用 swtch 来开始运行它 (kernel/proc.c:461-466)。

7.4 代码：mycpu 和 myproc

Xv6经常需要一个指向当前进程的proc结构的指针。在单处理器上，可以有一个全局变量指向当前进程。但在多核机器上，这行不通，因为每个CPU都在执行不同的进程。解决这个问题的方法是通过利用每个CPU都有自己的寄存器集这一事实。

当某个CPU在内核中执行时，xv6确保该CPU的tp寄存器始终持有该CPU的hartid。RISC-V为它的CPU编号，给每个分配一个唯一的hartid。mycpu（kernel/proc.c:74）使用tp来索引一个cpu结构数组，并返回当前CPU的那个结构。一个struct cpu（kernel/proc.h:22）包含一个指向该CPU上当前正在运行的进程的proc结构的指针（如果有），该CPU调度线程的保存寄存器，以及管理中断禁用所需的嵌套自旋锁计数。

确保CPU的tp持有该CPU的hartid有点复杂，因为用户代码可以自由修改tp。start在CPU的启动序列早期设置tp寄存器，此时仍在机器模式（kernel/start.c:45）。usertrapret将tp保存在跳板页中，以防用户代码修改它。最后，uservec在从用户空间进入内核时恢复那个保存的tp（kernel/trampoline.S:78）。编译器保证内核代码永远不会修改tp。如果xv6能够在需要时向RISC-V硬件请求当前的hartid，那将更方便，但RISC-V只允许在机器模式中这样做，不允许在监督模式中。

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

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

7.5 Sleep and wakeup

Scheduling and locks help conceal the actions of one thread from another, but we also need abstractions that help threads intentionally interact. For example, the reader of a pipe in xv6 may need to wait for a writing process to produce data; a parent's call to `wait` may need to wait for a child to exit; and a process reading the disk needs to wait for the disk hardware to finish the read. The xv6 kernel uses a mechanism called sleep and wakeup in these situations (and many others). Sleep allows a kernel thread to wait for a specific event; another thread can call wakeup to indicate that threads waiting for a specified event should resume. Sleep and wakeup are often called *sequence coordination* or *conditional synchronization* mechanisms.

Sleep and wakeup provide a relatively low-level synchronization interface. To motivate the way they work in xv6, we'll use them to build a higher-level synchronization mechanism called a *semaphore* [5] that coordinates producers and consumers (xv6 does not use semaphores). A semaphore maintains a count and provides two operations. The "V" operation (for the producer) increments the count. The "P" operation (for the consumer) waits until the count is non-zero, and then decrements it and returns. If there were only one producer thread and one consumer thread, and they executed on different CPUs, and the compiler didn't optimize too aggressively, this implementation would be correct:

```
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);
}
```

The implementation above is expensive. If the producer acts rarely, the consumer will spend most of its time spinning in the `while` loop hoping for a non-zero count. The consumer's CPU could probably find more productive work than *busy waiting* by repeatedly *polling* `s->count`.

7.5 睡眠和唤醒

调度和锁有助于隐藏一个线程对另一个线程的操作，但我们还需要帮助线程有意交互的抽象。例如，xv6 中的管道读者可能需要等待写入进程产生数据；父进程的等待调用可能需要等待子进程退出；读取磁盘的进程需要等待磁盘硬件完成读取。xv6 内核在这些情况下（以及其他许多情况下）使用一种称为睡眠和唤醒的机制。睡眠允许内核线程等待特定事件；另一个线程可以调用唤醒来指示等待特定事件的线程应该恢复。睡眠和唤醒通常被称为序列协调或条件同步机制。

睡眠和唤醒提供一种相对底层的同步接口。为了说明它们在 xv6 中的工作方式，我们将使用它们来构建一种更高级的同步机制，称为信号量 [5]，它协调生产者和消费者（xv6 不使用信号量）。信号量维护一个计数，并提供两种操作。对于生产者的“V”操作（Increment）会递增计数。对于消费者的“P”操作（Proberen，荷兰语意为“测试”）会等待计数非零，然后递减计数并返回。如果只有一个生产者线程和一个消费者线程，并且它们在不同的 CPU 上执行，而编译器没有过度优化，这种实现将是正确的：

```
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);
}
```

上述实现很昂贵。如果生产者很少动作，消费者将花费大部分时间在while循环中旋转，希望计数不为零。消费者CPU可能比忙等待更能找到更有生产性的工作，通过反复轮询s->count。

Avoiding busy waiting requires a way for the consumer to yield the CPU and resume only after V increments the count.

Here's a step in that direction, though as we will see it is not enough. Let's imagine a pair of calls, `sleep` and `wakeup`, that work as follows. `sleep(chan)` waits for an event designated by the value of `chan`, called the *wait channel*. `sleep` puts the calling process to sleep, releasing the CPU for other work. `wakeup(chan)` wakes all processes that are in calls to `sleep` with the same `chan` (if any), causing their `sleep` calls to return. If no processes are waiting on `chan`, `wakeup` does nothing. We can change the semaphore implementation to use `sleep` and `wakeup` (changes highlighted in yellow):

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

P now gives up the CPU instead of spinning, which is nice. However, it turns out not to be straightforward to design `sleep` and `wakeup` with this interface without suffering from what is known as the *lost wake-up* problem. Suppose that P finds that `s->count == 0` on line 212. While P is between lines 212 and 213, V runs on another CPU: it changes `s->count` to be nonzero and calls `wakeup`, which finds no processes sleeping and thus does nothing. Now P continues executing at line 213: it calls `sleep` and goes to sleep. This causes a problem: P is asleep waiting for a V call that has already happened. Unless we get lucky and the producer calls V again, the consumer will wait forever even though the count is non-zero.

The root of this problem is that the invariant that P sleeps only when `s->count == 0` is violated by V running at just the wrong moment. An incorrect way to protect the invariant would be to move the lock acquisition (highlighted in yellow below) in P so that its check of the count and its call to `sleep` are atomic:

```
300  void
301  V(struct semaphore *s)
302  {
303    acquire(&s->lock);
```

避免忙等待需要一个方法，让消费者让出CPU，并且只在V增加计数后才恢复。

这是一个方向上的步骤，尽管正如我们将看到的，这还不够。让我们想象一对调用，睡眠和唤醒，它们的工作方式如下。sleep(chan) 等待由 chan 值指定的事件，称为等待通道。sleep 将调用进程置于睡眠状态，释放 CPU 以供其他工作使用。wakeup(chan) 唤醒所有在具有相同 chan 的 sleep 调用中等待的进程（如果有），使其 sleep 调用返回。如果没有进程在等待 chan，wakeup 什么也不做。我们可以更改信号量实现以使用睡眠和唤醒（黄色突出显示的更改）：

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

P 现在放弃 CPU 而不是旋转，这很好。然而，事实证明，使用此接口设计睡眠和唤醒而不遭受所谓的丢失唤醒问题并不容易。假设 P 发现 s->count == 0 在行 212 上。当 P 在行 212 和 213 之间时，V 在另一个 CPU 上运行：它将 s->count 更改为非零并调用 wakeup，wakeup 发现没有进程在睡眠并因此什么也不做。现在 P 在行 213 继续执行：它调用 sleep 并进入睡眠状态。这会导致一个问题：P 在等待已经发生的 V 调用时处于睡眠状态。除非我们走运并且生产者再次调用 V，否则消费者将永远等待，即使计数非零。

这个问题根源在于，当 V 在不恰当的时刻运行时，P 仅在 s->count == 0 时睡眠的不变量被违反。保护不变量的错误方法是将锁获取（下方黄色高亮）在 P 中移动，以便其对计数的检查和对其睡眠的调用是原子性的：

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

One might hope that this version of `P` would avoid the lost wakeup because the lock prevents `V` from executing between lines 313 and 314. It does that, but it also deadlocks: `P` holds the lock while it sleeps, so `V` will block forever waiting for the lock.

We'll fix the preceding scheme by changing `sleep`'s interface: the caller must pass the *condition lock* to `sleep` so it can release the lock after the calling process is marked as asleep and waiting on the sleep channel. The lock will force a concurrent `V` to wait until `P` has finished putting itself to sleep, so that the `wakeup` will find the sleeping consumer and wake it up. Once the consumer is awake again `sleep` reacquires the lock before returning. Our new correct sleep/wakeup scheme is usable as follows (change highlighted in yellow):

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

The fact that `P` holds `s->lock` prevents `V` from trying to wake it up between `P`'s check of `s->count` and its call to `sleep`. However, `sleep` must release `s->lock` and put the consuming

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

人们可能希望这个版本的 P 会避免丢失唤醒，因为锁防止 V 在行 313 和 314 之间执行。它确实做到了这一点，但它也会导致死锁：P 在睡眠时持有锁，因此 V 将永远等待锁而阻塞。

我们将通过更改 sleep 的接口来修复前面的方案：调用者必须将条件锁传递给 sleep，以便在调用进程被标记为睡眠并等待在睡眠通道上时释放锁。锁将迫使并发 V 等待 P 完成将自己置于睡眠状态，以便唤醒操作能够找到睡眠的消费者并将其唤醒。一旦消费者再次醒来，sleep 会在返回之前重新获取锁。我们新的正确 sleep/唤醒方案如下所示（黄色高亮部分为更改内容）：

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

P 持有 s->锁的事实阻止 V 在 P 检查 s->计数和其调用 sleep 之间尝试唤醒它。然而，sleep 必须释放 s->锁并以从唤醒的角度来看是原子的方式将消耗进程置于睡眠状态，以避免丢失唤醒。

process to sleep in a way that's atomic from the point of view of `wakeup`, in order to avoid lost wakeups.

7.6 Code: Sleep and wakeup

Xv6's `sleep` (kernel/proc.c:548) and `wakeup` (kernel/proc.c:579) provide the interface used in the last example above. The basic idea is to have `sleep` mark the current process as `SLEEPING` and then call `sched` to release the CPU; `wakeup` looks for a process sleeping on the given wait channel and marks it as `RUNNABLE`. Callers of `sleep` and `wakeup` can use any mutually convenient number as the channel. Xv6 often uses the address of a kernel data structure involved in the waiting.

`sleep` acquires `p->lock` (kernel/proc.c:559) and *only then* releases `lk`. As we'll see, the fact that `sleep` holds one or the other of these locks at all times is what prevents a concurrent `wakeup` (which must acquire and hold both) from acting. Now that `sleep` holds just `p->lock`, it can put the process to sleep by recording the sleep channel, changing the process state to `SLEEPING`, and calling `sched` (kernel/proc.c:563-566). In a moment it will be clear why it's critical that `p->lock` is not released (by `scheduler`) until after the process is marked `SLEEPING`.

At some point, a process will acquire the condition lock, set the condition that the sleeper is waiting for, and call `wakeup(chan)`. It's important that `wakeup` is called while holding the condition lock[1]. `wakeup` loops over the process table (kernel/proc.c:579). It acquires the `p->lock` of each process it inspects. When `wakeup` finds a process in state `SLEEPING` with a matching `chan`, it changes that process's state to `RUNNABLE`. The next time `scheduler` runs, it will see that the process is ready to be run.

Why do the locking rules for `sleep` and `wakeup` ensure that a process that's going to sleep won't miss a concurrent wakeup? The going-to-sleep process holds either the condition lock or its own `p->lock` or both from *before* it checks the condition until *after* it is marked `SLEEPING`. The process calling `wakeup` holds *both* locks in `wakeup`'s loop. Thus the waker either makes the condition true before the consuming thread checks the condition; or the waker's `wakeup` examines the sleeping thread strictly after it has been marked `SLEEPING`. Then `wakeup` will see the sleeping process and wake it up (unless something else wakes it up first).

Sometimes multiple processes are sleeping on the same channel; for example, more than one process reading from a pipe. A single call to `wakeup` will wake them all up. One of them will run first and acquire the lock that `sleep` was called with, and (in the case of pipes) read whatever data is waiting. The other processes will find that, despite being woken up, there is no data to be read. From their point of view the wakeup was "spurious," and they must sleep again. For this reason `sleep` is always called inside a loop that checks the condition.

No harm is done if two uses of sleep/wakeup accidentally choose the same channel: they will see spurious wakeups, but looping as described above will tolerate this problem. Much of the charm of sleep/wakeup is that it is both lightweight (no need to create special data structures to act as sleep channels) and provides a layer of indirection (callers need not know which specific process they are interacting with).

[1]Strictly speaking it is sufficient if `wakeup` merely follows the `acquire` (that is, one could call `wakeup` after the `release`).

以原子性方式从唤醒的角度让进程睡眠，以避免丢失唤醒。

7.6 代码：睡眠和唤醒

Xv6 的睡眠（kernel/proc.c:548）和唤醒（kernel/proc.c:579）提供了上述最后一个示例中使用的接口。基本思想是让睡眠将当前进程标记为 SLEEPING，然后调用 sched 释放 CPU；唤醒会查找在给定等待通道上睡眠的进程并将其标记为 RUNNABLE。睡眠和唤醒的调用者可以使用任何互方便宜的数字作为通道。Xv6 经常使用参与等待的内核数据结构的地址。

睡眠获取 p->锁（kernel/proc.c:559）然后才释放 lk。正如我们将看到的，睡眠始终持有这两种锁中的一种的事实是防止并发唤醒（必须获取并持有这两种锁）起作用的原因。现在睡眠只持有 p->锁，它可以通过记录睡眠通道、将进程状态更改为 SLEEPING 并调用 sched（kernel/proc.c:563-566）来使进程睡眠。稍后将会清楚为什么 p->锁直到进程被标记为 SLEEPING 后（由调度器）不会释放。

在某个时刻，一个进程将获取条件锁，设置睡眠者等待的条件，并调用 wakeup(chan)。重要的是，在持有条件锁[1]时调用 wakeup。wakeup 遍历进程表 (kernel/proc.c:579)。它获取它检查的每个进程的 p->锁。当 wakeup 找到一个状态为 SLEEPING 且 chan 匹配的进程时，它会将该进程的状态更改为 RUNNABLE。下次调度器运行时，它会发现该进程准备运行。

为什么睡眠和唤醒的锁定规则确保将要睡眠的进程不会错过并发唤醒？将要睡眠的进程在检查条件之前和被标记为 SLEEPING 之后，会持有条件锁或其自己的 p->锁或两者。调用 wakeup 的进程在 wakeup 的循环中持有这两个锁。因此，唤醒者要么在消耗线程检查条件之前使条件为真；要么唤醒者的 wakeup 在睡眠线程被标记为 SLEEPING 后严格检查它。然后 wakeup 将看到睡眠进程并将其唤醒（除非有其他东西先唤醒它）。

有时多个进程会睡眠在同一个通道上；例如，多个进程从同一个管道读取。一个 wakeup 调用会唤醒它们全部。其中一个会首先运行并获取 sleep 调用时使用的锁，并且在（管道的情况下）读取等待中的任何数据。其他进程会发现，尽管被唤醒了，但没有数据可读取。从它们的角度来看，wakeup 是“虚假的”，它们必须再次睡眠。出于这个原因，slee p 总是调用在一个检查条件的循环内部。

如果两个 sleep/wakeup 的使用意外地选择了同一个通道，则不会造成任何伤害：它们会看到虚假唤醒，但如上所述的循环可以容忍这个问题。sleep/wakeup 的魅力之一在于它既轻量级（无需创建特殊的数据结构来充当睡眠通道），又提供了一层间接性（调用者无需知道它们正在与哪个具体进程交互）。

[1]严格来说，wakeup 仅需在 acquire 之后即可（也就是说，可以在 release 之后调用 wakeup）。

7.7 Code: Pipes

A more complex example that uses sleep and wakeup to synchronize producers and consumers is xv6's implementation of pipes. We saw the interface for pipes in Chapter 1: bytes written to one end of a pipe are copied to an in-kernel buffer and then can be read from the other end of the pipe. Future chapters will examine the file descriptor support surrounding pipes, but let's look now at the implementations of pipewrite and piperead.

Each pipe is represented by a struct pipe, which contains a lock and a data buffer. The fields nread and nwrite count the total number of bytes read from and written to the buffer. The buffer wraps around: the next byte written after buf[PIPESIZE-1] is buf[0]. The counts do not wrap. This convention lets the implementation distinguish a full buffer (nwrite == nread+PIPESIZE) from an empty buffer (nwrite == nread), but it means that indexing into the buffer must use buf[nread % PIPESIZE] instead of just buf[nread] (and similarly for nwrite).

Let's suppose that calls to piperead and pipewrite happen simultaneously on two different CPUs. pipewrite (kernel/pipe.c:77) begins by acquiring the pipe's lock, which protects the counts, the data, and their associated invariants. piperead (kernel/pipe.c:106) then tries to acquire the lock too, but cannot. It spins in acquire (kernel/spinlock.c:22) waiting for the lock. While piperead waits, pipewrite loops over the bytes being written (addr[0..n-1]), adding each to the pipe in turn (kernel/pipe.c:95). During this loop, it could happen that the buffer fills (kernel/pipe.c:88). In this case, pipewrite calls wakeup to alert any sleeping readers to the fact that there is data waiting in the buffer and then sleeps on &pi->nwrite to wait for a reader to take some bytes out of the buffer. sleep releases the pipe's lock as part of putting pipewrite's process to sleep.

piperead now acquires the pipe's lock and enters its critical section: it finds that pi->nread != pi->nwrite (kernel/pipe.c:113) (pipewrite went to sleep because pi->nwrite == pi->nread + PIPESIZE (kernel/pipe.c:88)), so it falls through to the for loop, copies data out of the pipe (kernel/pipe.c:120), and increments nread by the number of bytes copied. That many bytes are now available for writing, so piperead calls wakeup (kernel/pipe.c:127) to wake any sleeping writers before it returns. wakeup finds a process sleeping on &pi->nwrite, the process that was running pipewrite but stopped when the buffer filled. It marks that process as RUNNABLE.

The pipe code uses separate sleep channels for reader and writer (pi->nread and pi->nwrite); this might make the system more efficient in the unlikely event that there are lots of readers and writers waiting for the same pipe. The pipe code sleeps inside a loop checking the sleep condition; if there are multiple readers or writers, all but the first process to wake up will see the condition is still false and sleep again.

7.8 Code: Wait, exit, and kill

sleep and wakeup can be used for many kinds of waiting. An interesting example, introduced in Chapter 1, is the interaction between a child's exit and its parent's wait. At the time of the child's death, the parent may already be sleeping in wait, or may be doing something else; in the latter case, a subsequent call to wait must observe the child's death, perhaps long after it calls

7.7 代码：管道

一个更复杂的例子是使用 sleep 和 wakeup 来同步生产者和消费者的 xv6 的管道实现。我们在第一章看到了管道的接口：写入管道一端的字节会被复制到内核缓冲区，然后可以从管道的另一端读取。未来的章节将研究围绕管道的文件描述符支持，但让我们现在看看 pipewrite 和 piperead 的实现。

每个管道由一个 struct pipe 表示，其中包含一个锁和一个数据缓冲区。字段 nread 和 nwrite 计数从缓冲区读取和写入的总字节数。缓冲区会回绕：在 buf[PIPESIZE-1] 后写入的下一个字节是 buf[0]。计数不会回绕。这种约定允许实现区分满缓冲区（nwrite == nread+PIPESIZE）和空缓冲区（nwrite == nread），但这意味着在缓冲区中进行索引必须使用 buf[nread % PIPESIZE] 而不是仅仅使用 buf[nread]（nwrite 也类似）。

假设 piperead 和 pipewrite 调用在不同的 CPU 上同时发生。pipewrite (kernel/pipe.c:77) 首先获取管道的锁，该锁保护计数、数据和它们相关的不变式。piperead (kernel/pipe.c:106) 然后尝试获取锁，但无法获取。它在 acquire (kernel/spinlock.c:22) 中自旋等待锁。在 piperead 等待时，pipewrite 遍历正在写入的字节（addr[0..n-1]），依次将每个字节添加到管道中（kernel/pipe.c:95）。在此循环期间，可能会发生缓冲区已满的情况（kernel/pipe.c:88）。在这种情况下，pipewrite 调用 wakeup 来通知任何睡眠的读者缓冲区中有数据等待，然后睡眠在 &pi->nwrite 上等待读者从缓冲区中取出一些字节。sleep 在将 pipewrite 的进程置于睡眠状态时释放管道的锁。

piperead 现在获取管道的锁并进入其临界区：它发现 pi->nread !=pi->nwrite (kernel/pipe.c:113) (pipewrite 进入睡眠状态，因为 pi->nwrite == pi->nread+ PIPESIZE (kernel/pipe.c:88))，所以它进入 for 循环，将数据从管道中复制出来 (kernel/pipe.c:120)，并将复制的字节数加到 nread 上。现在有这么多字节可供写入，所以 piperead 调用 wakeup (kernel/pipe.c:127) 来唤醒任何睡眠的写者，然后它返回。wakeup 发现一个进程正在睡眠在 &pi->nwrite 上，即运行 pipewrite 但在缓冲区满时停止的进程。它将该进程标记为 RUNNABLE。

管道代码使用独立的睡眠通道为读者和写者 (pi->nreadand pi->nwrite)；在不太可能的情况下，如果有很多读者和写者在等待同一个管道，这可能使系统更高效。管道代码在检查睡眠条件的循环中睡眠；如果有多个读者或写者，除了第一个唤醒的进程外，其他进程都会看到条件仍然为假并再次睡眠。

7.8 代码：wait,退出,和杀死

睡眠和唤醒可用于多种等待。第一章介绍的一个有趣例子是子进程退出与其父进程等待之间的交互。在子进程死亡时，父进程可能已经在等待中睡眠，或者可能在做其他事情；在后一种情况下，随后的等待调用必须观察到子进程的死亡，可能是在它调用很久之后。

exit. The way that xv6 records the child's demise until wait observes it is for exit to put the caller into the ZOMBIE state, where it stays until the parent's wait notices it, changes the child's state to UNUSED, copies the child's exit status, and returns the child's process ID to the parent. If the parent exits before the child, the parent gives the child to the init process, which perpetually calls wait; thus every child has a parent to clean up after it. A challenge is to avoid races and deadlock between simultaneous parent and child wait and exit, as well as simultaneous exit and exit.

wait starts by acquiring wait_lock (kernel/proc.c:391), which acts as the condition lock that helps ensure that wait doesn't miss a wakeup from an exiting child. Then wait scans the process table. If it finds a child in ZOMBIE state, it frees that child's resources and its proc structure, copies the child's exit status to the address supplied to wait (if it is not 0), and returns the child's process ID. If wait finds children but none have exited, it calls sleep to wait for any of them to exit (kernel/proc.c:433), then scans again. wait often holds two locks, wait_lock and some process's pp->lock; the deadlock-avoiding order is first wait_lock and then pp->lock.

exit (kernel/proc.c:347) records the exit status, frees some resources, calls reparent to give its children to the init process, wakes up the parent in case it is in wait, marks the caller as a zombie, and permanently yields the CPU. exit holds both wait_lock and p->lock during this sequence. It holds wait_lock because it's the condition lock for the wakeup(p->parent), preventing a parent in wait from losing the wakeup. exit must hold p->lock for this sequence also, to prevent a parent in wait from seeing that the child is in state ZOMBIE before the child has finally called swtch. exit acquires these locks in the same order as wait to avoid deadlock.

It may look incorrect for exit to wake up the parent before setting its state to ZOMBIE, but that is safe: although wakeup may cause the parent to run, the loop in wait cannot examine the child until the child's p->lock is released by scheduler, so wait can't look at the exiting process until well after exit has set its state to ZOMBIE (kernel/proc.c:379).

While exit allows a process to terminate itself, kill (kernel/proc.c:598) lets one process request that another terminate. It would be too complex for kill to directly destroy the victim process, since the victim might be executing on another CPU, perhaps in the middle of a sensitive sequence of updates to kernel data structures. Thus kill does very little: it just sets the victim's p->killed and, if it is sleeping, wakes it up. Eventually the victim will enter or leave the kernel, at which point code in usertrap will call exit if p->killed is set (it checks by calling killed (kernel/proc.c:627)). If the victim is running in user space, it will soon enter the kernel by making a system call or because the timer (or some other device) interrupts.

If the victim process is in sleep, kill's call to wakeup will cause the victim to return from sleep. This is potentially dangerous because the condition being waited for for may not be true. However, xv6 calls to sleep are always wrapped in a while loop that re-tests the condition after sleep returns. Some calls to sleep also test p->killed in the loop, and abandon the current activity if it is set. This is only done when such abandonment would be correct. For example, the pipe read and write code (kernel/pipe.c:84) returns if the killed flag is set; eventually the code will return back to trap, which will again check p->killed and exit.

Some xv6 sleep loops do not check p->killed because the code is in the middle of a multi-step system call that should be atomic. The virtio driver (kernel/virtio_disk.c:285) is an example: it

退出。xv6记录子进程死亡直到等待观察到它的方式是，退出将调用者置于僵尸状态，直到父进程的等待注意到它，将子进程的状态更改为未使用，复制子进程的退出状态，并将子进程的进程ID返回给父进程。如果父进程在子进程之前退出，父进程将子进程交给init进程，init进程不断调用等待；因此每个子进程都有一个父进程来清理它。一个挑战是避免父进程和子进程同时等待和退出以及同时退出和退出的竞争条件和死锁。

wait 开始获取 wait_锁（kernel/proc.c:391），该锁作为条件锁，有助于确保 wait 不会错过退出子进程的唤醒。然后 wait 扫描进程表。如果找到一个处于僵尸状态（ZOMBIE state）的子进程，它会释放该子进程的资源及其 proc 结构，将子进程的退出状态复制到传递给 wait 的地址（如果它不是 0），然后返回子进程的进程 ID。如果 wait 找到子进程但没有子进程退出，它会调用 sleep 等待它们中的任何一个退出（kernel/proc.c:433），然后再次扫描。wait 通常持有两个锁，wait_锁和某个进程的 pp->锁；避免死锁的顺序是先 wait_锁，然后是 pp->锁。

退出（kernel/proc.c:347）记录退出状态，释放一些资源，调用重新父进程将它的子进程交给init进程，唤醒父进程以防它处于等待状态，将调用者标记为僵尸状态，并永久让出CPU。退出在此序列中持有wait_锁和p->锁。它持有wait_锁，因为它是唤醒(p->父进程)的条件锁，防止处于等待状态的父进程丢失唤醒。退出在此序列中也必须持有p->锁，以防止处于等待状态的父进程在子进程最终调用swtch之前看到子进程处于僵尸状态。退出以与wait相同的顺序获取这些锁以避免死锁。

退出在设置其状态为僵尸状态之前唤醒父进程可能看起来不正确，但这很安全：虽然唤醒可能会使父进程运行，但在调度器释放子进程的 p->锁之前，wait 中的循环无法检查子进程，因此 wait 直到退出将状态设置为僵尸状态很久之后才能查看退出进程 (kernel/proc.c:379)。

虽然退出允许进程自我终止，但杀死（kernel/proc.c:598）允许一个进程请求另一个进程终止。杀死直接销毁受害者进程过于复杂，因为受害者可能正在另一个 CPU 上执行，也许在内核数据结构的敏感更新序列中间。因此杀死几乎什么也不做：它只是设置受害者的 p->killed，如果它在睡眠中，就唤醒它。最终受害者将进入或离开内核，此时 usertrap 中的代码会调用退出，如果 p->killed 被设置（它通过调用 killed (kernel/proc.c:627) 来检查）。如果受害者正在用户空间运行，它将通过系统调用或因为定时器（或其他设备）中断而很快进入内核。

如果受害者进程处于睡眠状态，kill 对 wakeup 的调用将导致受害者从睡眠中返回。这可能是危险的，因为等待的条件可能并不成立。然而，xv6 对 sleep 的调用总是被包裹在一个 while 循环中，该循环在 sleep 返回后会重新测试条件。一些对 sleep 的调用也在循环中测试 p->killed，如果它被设置，则放弃当前活动。这只有在这种放弃是正确的情况下才会执行。例如，管道读写代码（kernel/pipe.c:84）如果 killed 标志被设置则返回；最终代码将返回到 trap，trap 会再次检查 p->killed 并退出。

一些 xv6 睡眠循环不会检查 p->killed，因为代码位于一个多步骤的系统调用中间，该系统调用应该是原子性的。virtio 驱动程序（kernel/virtio_disk.c:285）是一个例子：它

does not check p->killed because a disk operation may be one of a set of writes that are all needed in order for the file system to be left in a correct state. A process that is killed while waiting for disk I/O won't exit until it completes the current system call and usertrap sees the killed flag.

7.9 Process Locking

The lock associated with each process (p->lock) is the most complex lock in xv6. A simple way to think about p->lock is that it must be held while reading or writing any of the following struct proc fields: p->state, p->chan, p->killed, p->xstate, and p->pid. These fields can be used by other processes, or by scheduler threads on other CPUs, so it's natural that they must be protected by a lock.

However, most uses of p->lock are protecting higher-level aspects of xv6's process data structures and algorithms. Here's the full set of things that p->lock does:

- Along with p->state, it prevents races in allocating proc[] slots for new processes.
- It conceals a process from view while it is being created or destroyed.
- It prevents a parent's wait from collecting a process that has set its state to ZOMBIE but has not yet yielded the CPU.
- It prevents another CPU's scheduler from deciding to run a yielding process after it sets its state to RUNNABLE but before it finishes swtch.
- It ensures that only one CPU's scheduler decides to run a RUNNABLE processes.
- It prevents a timer interrupt from causing a process to yield while it is in swtch.
- Along with the condition lock, it helps prevent wakeup from overlooking a process that is calling sleep but has not finished yielding the CPU.
- It prevents the victim process of kill from exiting and perhaps being re-allocated between kill's check of p->pid and setting p->killed.
- It makes kill's check and write of p->state atomic.

The p->parent field is protected by the global lock wait_lock rather than by p->lock. Only a process's parent modifies p->parent, though the field is read both by the process itself and by other processes searching for their children. The purpose of wait_lock is to act as the condition lock when wait sleeps waiting for any child to exit. An exiting child holds either wait_lock or p->lock until after it has set its state to ZOMBIE, woken up its parent, and yielded the CPU. wait_lock also serializes concurrent exits by a parent and child, so that the init process (which inherits the child) is guaranteed to be woken up from its wait. wait_lock is a global lock rather than a per-process lock in each parent, because, until a process acquires it, it cannot know who its parent is.

不会检查 p->killed，因为磁盘操作可能是文件系统需要保持正确状态的一系列写入操作中的一员。一个在等待磁盘I/O时被杀死的进程不会退出，直到它完成当前的系统调用和用户陷阱看到被杀死的标志。

7.9 进程锁

与每个进程（p->lock）关联的锁是 xv6 中最复杂的锁。关于 p->lock 的一个简单理解是，在读取或写入以下任何 struct proc 字段时必须持有该锁：p->state、p->chan、p->killed、p->xstate 和 p->pid。这些字段可以被其他进程或其他 CPU 上的调度线程使用，因此它们自然需要被锁保护。

然而，p->lock 的大多数使用都是保护 xv6 的进程数据结构和算法的高级方面。以下是 p->lock 完整的功能集：

- 与 p->state 一起，它防止了为新进程分配 proc[] 槽位的竞争条件。
- 它在进程被创建或销毁时将其隐藏。
- 它防止父进程等待收集状态已设置为 ZOMBIE 但尚未释放 CPU 的进程。
- 它防止另一个CPU的调度器在将其状态设置为RUNNABLE之后但在完成swtch之前决定运行一个yielding进程。
- 它确保只有一个CPU的调度器决定运行一个RUNNABLE进程。
- 它防止定时器中断在进程处于swtch时导致其yield。
- 与条件锁一起，它有助于防止唤醒忽略一个正在调用sleep但尚未完成CPU yield的进程。
- 它防止kill的受害者进程在kill检查p->pid和设置p->killed之间退出并可能被重新分配。
- 它使kill的检查和写入p->state的操作原子化。

p->父进程字段由全局锁 wait_锁保护，而不是由 p->锁保护。只有进程的父进程会修改 p->父进程，尽管该字段既被进程本身读取，也被其他搜索其子进程的进程读取。wait_锁的作用是在 wait 睡眠等待任何子进程退出时作为条件锁。退出的子进程会持有 wait_锁或 p->锁，直到它将状态设置为 ZOMBIE、唤醒其父进程并让出 CPU。wait_锁还通过父进程和子进程的并发退出进行序列化，以确保 init 进程（继承子进程）能够从其 wait 中被唤醒。wait_锁是全局锁，而不是每个父进程的每个进程锁，因为在进程获取它之前，它无法知道谁是它的父进程。

7.10 Real world

The xv6 scheduler implements a simple scheduling policy, which runs each process in turn. This policy is called *round robin*. Real operating systems implement more sophisticated policies that, for example, allow processes to have priorities. The idea is that a runnable high-priority process will be preferred by the scheduler over a runnable low-priority process. These policies can become complex quickly because there are often competing goals: for example, the operating system might also want to guarantee fairness and high throughput. In addition, complex policies may lead to unintended interactions such as *priority inversion* and *convoys*. Priority inversion can happen when a low-priority and high-priority process both use a particular lock, which when acquired by the low-priority process can prevent the high-priority process from making progress. A long convoy of waiting processes can form when many high-priority processes are waiting for a low-priority process that acquires a shared lock; once a convoy has formed it can persist for long time. To avoid these kinds of problems additional mechanisms are necessary in sophisticated schedulers.

`sleep` and `wakeup` are a simple and effective synchronization method, but there are many others. The first challenge in all of them is to avoid the "lost wakeups" problem we saw at the beginning of the chapter. The original Unix kernel's `sleep` simply disabled interrupts, which sufficed because Unix ran on a single-CPU system. Because xv6 runs on multiprocessors, it adds an explicit lock to `sleep`. FreeBSD's `msleep` takes the same approach. Plan 9's `sleep` uses a callback function that runs with the scheduling lock held just before going to sleep; the function serves as a last-minute check of the sleep condition, to avoid lost wakeups. The Linux kernel's `sleep` uses an explicit process queue, called a wait queue, instead of a wait channel; the queue has its own internal lock.

Scanning the entire set of processes in `wakeup` is inefficient. A better solution is to replace the `chan` in both `sleep` and `wakeup` with a data structure that holds a list of processes sleeping on that structure, such as Linux's wait queue. Plan 9's `sleep` and `wakeup` call that structure a rendezvous point. Many thread libraries refer to the same structure as a condition variable; in that context, the operations `sleep` and `wakeup` are called `wait` and `signal`. All of these mechanisms share the same flavor: the sleep condition is protected by some kind of lock dropped atomically during sleep.

The implementation of `wakeup` wakes up all processes that are waiting on a particular channel, and it might be the case that many processes are waiting for that particular channel. The operating system will schedule all these processes and they will race to check the sleep condition. Processes that behave in this way are sometimes called a *thundering herd*, and it is best avoided. Most condition variables have two primitives for `wakeup`: `signal`, which wakes up one process, and `broadcast`, which wakes up all waiting processes.

Semaphores are often used for synchronization. The count typically corresponds to something like the number of bytes available in a pipe buffer or the number of zombie children that a process has. Using an explicit count as part of the abstraction avoids the "lost wakeup" problem: there is an explicit count of the number of wakeups that have occurred. The count also avoids the spurious wakeup and thundering herd problems.

Terminating processes and cleaning them up introduces much complexity in xv6. In most operating systems it is even more complex, because, for example, the victim process may be deep

7.10 现实世界

xv6 调度器实现了一种简单的调度策略，该策略按顺序运行每个进程。这种策略称为轮询。真实操作系统实现了更复杂的策略，例如，允许进程具有优先级。其思想是，可运行的优先级高的进程会被调度器优先于可运行的优先级低的进程。这些策略很快会变得复杂，因为通常存在竞争目标：例如，操作系统可能还想保证公平性和高吞吐量。此外，复杂的策略可能导致意外的交互，如优先级反转和护航队。优先级反转可能发生在低优先级和高优先级进程都使用特定的锁时，当低优先级进程获取该锁时，可能会阻止高优先级进程取得进展。当许多高优先级进程等待获取共享锁的低优先级进程时，可能会形成长长的等待护航队；一旦护航队形成，它可能会持续很长时间。为了避免这类问题，复杂的调度器需要额外的机制。

睡眠和唤醒是一种简单有效的同步方法，但也有许多其他方法。所有这些方法的首要挑战都是避免我们在章节开头看到的“丢失唤醒”问题。原始Unix内核的睡眠只是禁用了中断，这足够了，因为Unix运行在单CPU系统上。由于xv6运行在多处理器上，它为睡眠添加了一个显式的锁。FreeBSD的msleep采取了相同的方法。Plan 9的睡眠使用一个回调函数，在进入睡眠之前持有调度锁运行；该函数作为睡眠条件的最后一刻检查，以避免丢失唤醒。Linux内核的睡眠使用一个显式的进程队列，称为等待队列，而不是await channel；该队列有自己的内部锁。

在唤醒中扫描整个进程集是低效的。更好的解决方案是用一个数据结构替换睡眠和唤醒中的chan，该结构保存了在结构上睡眠的进程列表，例如Linux的等待队列。Plan 9的睡眠和唤醒称该结构为汇合点。许多线程库将同一结构称为条件变量；在这种情况下，操作sleep和wakeup被称为wait和signal。所有这些机制都有相同的特点：睡眠条件由某种锁保护，该锁在睡眠期间原子性地释放。

唤醒的实现会唤醒所有正在等待特定通道的进程，而且可能有许多进程正在等待该特定通道。操作系统将调度所有这些进程，它们会竞争检查睡眠条件。以这种方式行为的进程有时被称为雷声阵阵，最好避免这种情况。大多数条件变量有两个用于唤醒的原语：信号，它唤醒一个进程，以及广播，它唤醒所有等待的进程。

信号量通常用于同步。计数通常对应于诸如管道缓冲区中可用字节数或进程拥有的僵尸子进程数量之类的东西。使用显式计数作为抽象的一部分可以避免“丢失唤醒”问题：有发生唤醒次数的显式计数。计数还可以避免虚假唤醒和雷声阵阵问题。

终止进程并清理它们会引入大量复杂性到 xv6 中。在大多数操作系统中，它甚至更复杂，因为，例如，受害者进程可能很深

inside the kernel sleeping, and unwinding its stack requires care, since each function on the call stack may need to do some clean-up. Some languages help out by providing an exception mechanism, but not C. Furthermore, there are other events that can cause a sleeping process to be woken up, even though the event it is waiting for has not happened yet. For example, when a Unix process is sleeping, another process may send a `signal` to it. In this case, the process will return from the interrupted system call with the value -1 and with the error code set to EINTR. The application can check for these values and decide what to do. Xv6 doesn't support signals and this complexity doesn't arise.

Xv6's support for `kill` is not entirely satisfactory: there are sleep loops which probably should check for `p->killed`. A related problem is that, even for `sleep` loops that check `p->killed`, there is a race between `sleep` and `kill`; the latter may set `p->killed` and try to wake up the victim just after the victim's loop checks `p->killed` but before it calls `sleep`. If this problem occurs, the victim won't notice the `p->killed` until the condition it is waiting for occurs. This may be quite a bit later or even never (e.g., if the victim is waiting for input from the console, but the user doesn't type any input).

A real operating system would find free `proc` structures with an explicit free list in constant time instead of the linear-time search in `allocproc`; xv6 uses the linear scan for simplicity.

7.11 Exercises

1. Implement semaphores in xv6 without using `sleep` and `wakeup` (but it is OK to use spin locks). Choose a few of xv6's uses of sleep and wakeup and replace them with semaphores. Judge the result.

2. Fix the race mentioned above between `kill` and `sleep`, so that a `kill` that occurs after the victim's sleep loop checks `p->killed` but before it calls `sleep` results in the victim abandoning the current system call.

3. Design a plan so that every sleep loop checks `p->killed` so that, for example, a process that is in the virtio driver can return quickly from the while loop if it is killed by another process.

4. Modify xv6 to use only one context switch when switching from one process's kernel thread to another, rather than switching through the scheduler thread. The yielding thread will need to select the next thread itself and call `swtch`. The challenges will be to prevent multiple CPUs from executing the same thread accidentally; to get the locking right; and to avoid deadlocks.

5. Modify xv6's `scheduler` to use the RISC-V `WFI` (wait for interrupt) instruction when no processes are runnable. Try to ensure that, any time there are runnable processes waiting to run, no CPUs are pausing in `WFI`.

在里面内核睡眠，并且展开它的栈需要小心，因为调用栈上的每个函数可能需要做一些清理。一些语言通过提供异常机制来提供帮助，但不是 C。此外，还有其他事件可以导致睡眠进程被唤醒，即使它正在等待的事件还没有发生。例如，当一个 Unix 进程在睡眠时，另一个进程可以给它发送一个信号。在这种情况下，进程会从被中断的系统调用返回，值为 -1，并且错误代码设置为 EINTR。应用程序可以检查这些值并决定要做什么。Xv6 不支持信号，这种复杂性不会出现。

Xv6 对 kill 的支持并不完全令人满意：存在一些睡眠循环，可能应该检查 p->killed。一个相关的问题是，即使对于检查 p->killed 的睡眠循环，睡眠和杀死之间存在竞争条件；后者可能在受害者检查 p->killed 后但调用 sleep 前设置 p->killed 并尝试唤醒受害者。如果这个问题发生，受害者直到它等待的条件发生时才不会注意到 p->killed。这可能要晚很多，甚至永远不会（例如，如果受害者正在等待控制台的输入，但用户没有输入任何内容）。

一个真正的操作系统会使用显式的空闲列表以常数时间查找空闲的进程结构，而不是 allocproc 中的线性时间搜索；xv6 使用线性扫描是为了简化。

7.11 练习

1. 在 xv6 中实现信号量，但不使用睡眠和唤醒（但可以使用自旋锁）。选择一些 xv6 中使用睡眠和唤醒的用例，并用信号量替换它们。评估结果。

2. 修复上述kill和sleep之间的竞争条件，以便在受害者睡眠循环检查p->killed之后发生kill，但在调用sleep之前发生kill时，受害者会放弃当前的系统调用。

3. 设计一个计划，以便每个睡眠循环检查p->killed，以便例如，如果另一个进程杀死了一个处于virtio驱动的进程，该进程可以快速从while循环返回。

4. 修改xv6，以便在从一个进程的内核线程切换到另一个进程时只使用一个上下文切换，而不是通过调度线程切换。让出线程需要自己选择下一个线程并调用swtch。挑战在于防止多个CPU意外执行相同的线程；正确地实现锁定；以及避免死锁。

5. 修改xv6的调度器，以便在没有可运行进程时使用RISC-V WFI（等待中断）指令。尽量确保，只要可运行进程等待运行，没有CPU会暂停在WFI中。

84

Chapter 8

File system

The purpose of a file system is to organize and store data. File systems typically support sharing of data among users and applications, as well as *persistence* so that data is still available after a reboot.

The xv6 file system provides Unix-like files, directories, and pathnames (see Chapter 1), and stores its data on a virtio disk for persistence. The file system addresses several challenges:

- The file system needs on-disk data structures to represent the tree of named directories and files, to record the identities of the blocks that hold each file's content, and to record which areas of the disk are free.
- The file system must support *crash recovery*. That is, if a crash (e.g., power failure) occurs, the file system must still work correctly after a restart. The risk is that a crash might interrupt a sequence of updates and leave inconsistent on-disk data structures (e.g., a block that is both used in a file and marked free).
- Different processes may operate on the file system at the same time, so the file-system code must coordinate to maintain invariants.
- Accessing a disk is orders of magnitude slower than accessing memory, so the file system must maintain an in-memory cache of popular blocks.

The rest of this chapter explains how xv6 addresses these challenges.

8.1 Overview

The xv6 file system implementation is organized in seven layers, shown in Figure 8.1. The disk layer reads and writes blocks on an virtio hard drive. The buffer cache layer caches disk blocks and synchronizes access to them, making sure that only one kernel process at a time can modify the data stored in any particular block. The logging layer allows higher layers to wrap updates to several blocks in a *transaction*, and ensures that the blocks are updated atomically in the face of crashes (i.e., all of them are updated or none). The inode layer provides individual files, each

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，以便在重启后数据仍然可用。

xv6文件系统提供类Unix文件、目录和路径名（见第一章），并将其数据存储在virtio磁盘上进行持久化。文件系统解决了以下几个挑战：

• 文件系统需要磁盘数据结构来表示命名目录和文件的树，记录每个文件内容的块标识，以及记录磁盘的空闲区域。

• 文件系统必须支持崩溃恢复。也就是说，如果发生崩溃（例如电源故障），文件系统在重启后仍然必须能正常工作。风险在于崩溃可能会中断更新序列，并留下不一致的磁盘数据结构（例如，一个既被文件使用又被标记为空闲的块）。

• 不同的进程可能同时操作文件系统，因此文件系统代码
必须协调以维护不变式。

• 访问磁盘比访问内存慢几个数量级，因此文件系统必须维护一个包含常用块的内存缓存。

本章的其余部分解释了xv6如何应对这些挑战。

8.1 概述

xv6文件系统实现由七层组织，如图8.1所示。磁盘层在virtio硬盘上读取和写入块。缓冲区缓存层缓存磁盘块并同步它们的使用，确保一次只有一个内核进程可以修改任何特定块中的数据。日志层允许更高层将多个块的更新包装在一个事务中，并确保在崩溃的情况下（即全部更新或全部不更新）原子性地更新这些块。i节点层提供单个文件，每个文件表示为一个具有唯一i号和一些包含文件数据的块的i节点。

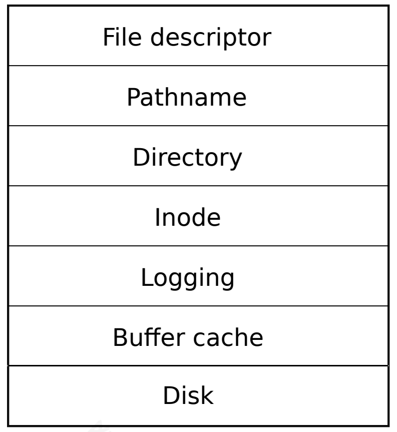

Figure 8.1: Layers of the xv6 file system.

represented as an *inode* with a unique i-number and some blocks holding the file's data. The directory layer implements each directory as a special kind of inode whose content is a sequence of directory entries, each of which contains a file's name and i-number. The pathname layer provides hierarchical path names like `/usr/rtm/xv6/fs.c`, and resolves them with recursive lookup. The file descriptor layer abstracts many Unix resources (e.g., pipes, devices, files, etc.) using the file system interface, simplifying the lives of application programmers.

Disk hardware traditionally presents the data on the disk as a numbered sequence of 512-byte *blocks* (also called *sectors*): sector 0 is the first 512 bytes, sector 1 is the next, and so on. The block size that an operating system uses for its file system maybe different than the sector size that a disk uses, but typically the block size is a multiple of the sector size. Xv6 holds copies of blocks that it has read into memory in objects of type `struct buf` (kernel/buf.h:1). The data stored in this structure is sometimes out of sync with the disk: it might have not yet been read in from disk (the disk is working on it but hasn't returned the sector's content yet), or it might have been updated by software but not yet written to the disk.

The file system must have a plan for where it stores inodes and content blocks on the disk. To do so, xv6 divides the disk into several sections, as Figure 8.2 shows. The file system does not use block 0 (it holds the boot sector). Block 1 is called the *superblock*; it contains metadata about the file system (the file system size in blocks, the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold the log. After the log are the inodes, with multiple inodes per block. After those come bitmap blocks tracking which data blocks are in use. The remaining blocks are data blocks; each is either marked free in the bitmap block, or holds content for a file or directory. The superblock is filled in by a separate program, called `mkfs`, which builds an initial file system.

The rest of this chapter discusses each layer, starting with the buffer cache. Look out for situations where well-chosen abstractions at lower layers ease the design of higher ones.

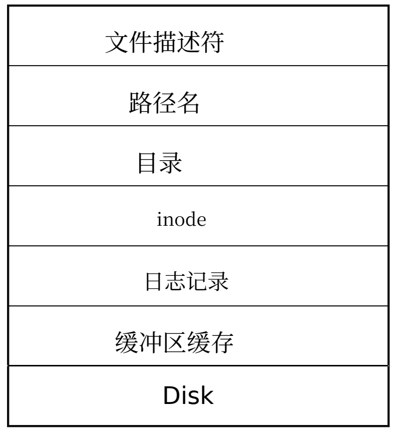

图8.1：xv6文件系统的层次结构。

目录层将每个目录实现为一种特殊的i节点，其内容是一系列目录条目，每个条目都包含一个文件名和i号。路径名层提供分层路径名，如/usr/rtm/xv6/fs.c，并使用递归查找解决它们。文件描述符层使用文件系统接口抽象许多Unix资源（例如管道、设备、文件等），简化了应用程序开发人员的生活。

磁盘硬件传统上将磁盘上的数据呈现为编号的512字节块序列（也称为扇区）：扇区0是前512字节，扇区1是下一个，以此类推。操作系统用于其文件系统的块大小可能与磁盘使用的扇区大小不同，但通常块大小是扇区大小的倍数。xv6在类型为struct buf（kernel/buf.h:1）的对象中保存已读入内存的块副本。此结构中存储的数据有时与磁盘不同步：它可能尚未从磁盘读取（磁盘正在处理但尚未返回扇区的内容），或者可能已被软件更新但尚未写入磁盘。

文件系统必须有一个计划，用于在磁盘上存储inode和内容块。为此，xv6将磁盘分为几个区域，如图8.2所示。文件系统不使用块0（它包含引导扇区）。块1称为超级块；它包含有关文件系统的元数据（文件系统的大小（以块为单位）、数据块的数量、inode的数量以及日志中的块数量）。从块2开始的块包含日志。日志之后是inode，每个块包含多个inode。之后是跟踪哪些数据块正在使用的位图块。剩余的块是数据块；每个块要么在位图块中标记为空闲，要么包含文件或目录的内容。超级块由一个单独的程序填充，该程序称为mkfs，它构建初始文件系统。

本章的其余部分将讨论每一层，从缓冲区缓存开始。注意那些在较低层选择良好的抽象可以简化较高层设计的场景。

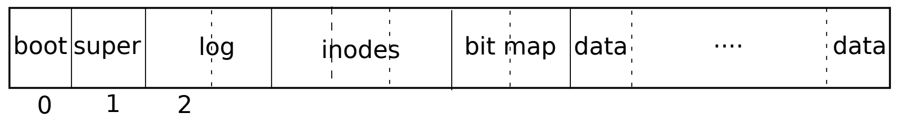

Figure 8.2: Structure of the xv6 file system.

8.2 Buffer cache layer

The buffer cache has two jobs: (1) synchronize access to disk blocks to ensure that only one copy of a block is in memory and that only one kernel thread at a time uses that copy; (2) cache popular blocks so that they don't need to be re-read from the slow disk. The code is in `bio.c`.

The main interface exported by the buffer cache consists of `bread` and `bwrite`; the former obtains a *buf* containing a copy of a block which can be read or modified in memory, and the latter writes a modified buffer to the appropriate block on the disk. A kernel thread must release a buffer by calling `brelse` when it is done with it. The buffer cache uses a per-buffer sleep-lock to ensure that only one thread at a time uses each buffer (and thus each disk block); `bread` returns a locked buffer, and `brelse` releases the lock.

Let's return to the buffer cache. The buffer cache has a fixed number of buffers to hold disk blocks, which means that if the file system asks for a block that is not already in the cache, the buffer cache must recycle a buffer currently holding some other block. The buffer cache recycles the least recently used buffer for the new block. The assumption is that the least recently used buffer is the one least likely to be used again soon.

8.3 Code: Buffer cache

The buffer cache is a doubly-linked list of buffers. The function `binit`, called by `main` (kernel/-main.c:27), initializes the list with the `NBUF` buffers in the static array `buf` (kernel/bio.c:43-52). All other access to the buffer cache refer to the linked list via `bcache.head`, not the `buf` array.

A buffer has two state fields associated with it. The field `valid` indicates that the buffer contains a copy of the block. The field `disk` indicates that the buffer content has been handed to the disk, which may change the buffer (e.g., write data from the disk into `data`).

`bread` (kernel/bio.c:93) calls `bget` to get a buffer for the given sector (kernel/bio.c:97). If the buffer needs to be read from disk, `bread` calls `virtio_disk_rw` to do that before returning the buffer.

`bget` (kernel/bio.c:59) scans the buffer list for a buffer with the given device and sector numbers (kernel/bio.c:65-73). If there is such a buffer, `bget` acquires the sleep-lock for the buffer. `bget` then returns the locked buffer.

If there is no cached buffer for the given sector, `bget` must make one, possibly reusing a buffer that held a different sector. It scans the buffer list a second time, looking for a buffer that is not in use (`b->refcnt = 0`); any such buffer can be used. `bget` edits the buffer metadata to record the new device and sector number and acquires its sleep-lock. Note that the assignment `b->valid = 0` ensures that `bread` will read the block data from disk rather than incorrectly using the buffer's

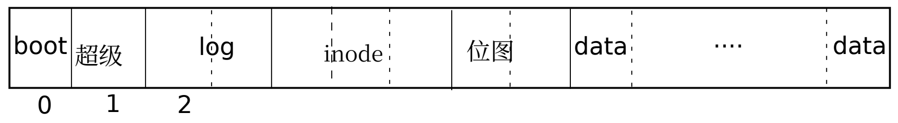

图8.2：xv6文件系统的结构。

8.2 缓冲缓存层

缓冲区缓存有两个任务：(1) 同步对磁盘块的访问，以确保一个块只有一个副本在内存中，并且一次只有一个内核线程使用该副本；(2) 缓存常用块，以便它们不需要从慢速磁盘重新读取。代码在 bio.c 中。

缓冲缓存的主要接口由bread和bwrite组成；前者获取一个包含块副本的buf，该副本可以在内存中读取或修改，后者将修改后的缓冲区写入磁盘上的相应块。当内核线程完成使用缓冲区时，必须通过调用brelse来释放它。缓冲缓存使用每个缓冲区的睡眠锁来确保同一时间只有一个线程使用每个缓冲区（因此每个磁盘块）；bread返回一个锁定的缓冲区，brelse释放锁。

让我们回到缓冲区缓存。缓冲区缓存有固定数量的缓冲区来存储磁盘块，这意味着如果文件系统请求一个不在缓存中的块，缓冲区缓存必须回收当前持有其他块的缓冲区。缓冲区缓存将最不常用的缓冲区回收以用于新块。假设最不常用的缓冲区是最不可能很快再次使用的。

8.3 代码：缓冲区缓存

缓冲区缓存是一个双向链表的缓冲区。由 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 个缓冲区初始化该链表。所有对缓冲区缓存的访问都引用 viabcache.head 链表，而不是 buf 数组。

一个缓冲区有两个与之关联的状态字段。字段 valid 表示缓冲区包含一个块的副本。字段 disk 表示缓冲区内容已交给磁盘，这可能改变缓冲区（例如，从磁盘写入数据到数据）。

bread (kernel/bio.c:93) 调用 bget 来为给定的扇区获取缓冲区 (kernel/bio.c:97)。如果缓冲区需要从磁盘读取，bread 会调用 virtio_disk_rw 来执行该操作，然后再返回缓冲区。

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

如果给定扇区没有缓存的缓冲区，bget 必须创建一个，可能重用曾经持有不同扇区的缓冲区。它再次扫描缓冲区列表，寻找一个未使用的缓冲区（b->refcnt = 0）；任何这样的缓冲区都可以使用。bget 编辑缓冲区元数据以记录新的设备和扇区号，并获取其睡眠锁。注意，赋值 b->valid = 0 确保 bread 会从磁盘读取块数据，而不是错误地使用缓冲区的前一个内容。

previous contents.

It is important that there is at most one cached buffer per disk sector, to ensure that readers see writes, and because the file system uses locks on buffers for synchronization. `bget` ensures this invariant by holding the `bache.lock bcache.lock` continuously from the first loop's check of whether the block is cached through the second loop's declaration that the block is now cached (by setting `dev`, `blockno`, and `refcnt`). This causes the check for a block's presence and (if not present) the designation of a buffer to hold the block to be atomic.

It is safe for `bget` to acquire the buffer's sleep-lock outside of the `bcache.lock` critical section, since the non-zero `b->refcnt` prevents the buffer from being re-used for a different disk block. The sleep-lock protects reads and writes of the block's buffered content, while the `bcache.lock` protects information about which blocks are cached.

If all the buffers are busy, then too many processes are simultaneously executing file system calls; `bget` panics. A more graceful response might be to sleep until a buffer became free, though there would then be a possibility of deadlock.

Once `bread` has read the disk (if needed) and returned the buffer to its caller, the caller has exclusive use of the buffer and can read or write the data bytes. If the caller does modify the buffer, it must call `bwrite` to write the changed data to disk before releasing the buffer. `bwrite` (kernel/bio.c:107) calls `virtio_disk_rw` to talk to the disk hardware.

When the caller is done with a buffer, it must call `brelse` to release it. (The name `brelse`, a shortening of b-release, is cryptic but worth learning: it originated in Unix and is used in BSD, Linux, and Solaris too.) `brelse` (kernel/bio.c:117) releases the sleep-lock and moves the buffer to the front of the linked list (kernel/bio.c:128-133). Moving the buffer causes the list to be ordered by how recently the buffers were used (meaning released): the first buffer in the list is the most recently used, and the last is the least recently used. The two loops in `bget` take advantage of this: the scan for an existing buffer must process the entire list in the worst case, but checking the most recently used buffers first (starting at `bcache.head` and following `next` pointers) will reduce scan time when there is good locality of reference. The scan to pick a buffer to reuse picks the least recently used buffer by scanning backward (following `prev` pointers).

8.4 Logging layer

One of the most interesting problems in file system design is crash recovery. The problem arises because many file-system operations involve multiple writes to the disk, and a crash after a subset of the writes may leave the on-disk file system in an inconsistent state. For example, suppose a crash occurs during file truncation (setting the length of a file to zero and freeing its content blocks). Depending on the order of the disk writes, the crash may either leave an inode with a reference to a content block that is marked free, or it may leave an allocated but unreferenced content block.

The latter is relatively benign, but an inode that refers to a freed block is likely to cause serious problems after a reboot. After reboot, the kernel might allocate that block to another file, and now we have two different files pointing unintentionally to the same block. If xv6 supported multiple users, this situation could be a security problem, since the old file's owner would be able to read

前一个内容。

每个磁盘扇区最多有一个缓存缓冲区很重要，以确保读者能看到写入，并且因为文件系统使用锁来同步缓冲区。bget通过从第一个循环检查块是否缓存开始，一直持有bache.lock bcache.lock，直到第二个循环声明块现在已缓存（通过设置dev、blockno和refcnt），从而确保这个不变量。这导致检查块是否存在以及（如果不存在）指定一个缓冲区来持有块的操作是原子的。

bget在bache.lock临界区外部获取缓冲区的睡眠锁是安全的，因为非零的b->refcnt防止缓冲区被用于不同的磁盘块。睡眠锁保护块缓冲区内容的读取和写入，而bcache.lock保护有关哪些块被缓存的 信息。

如果所有缓冲区都忙，那么同时执行文件系统调用的进程太多；bget会恐慌。更优雅的响应可能是睡眠直到缓冲区变为空闲，但那样可能会有死锁的可能性。

一旦 bread 读取了磁盘（如果需要）并将缓冲区返回给调用者，调用者将独占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，它必须在释放缓冲区之前调用 bwrite 将更改的数据写入磁盘。bwrite (kernel/bio.c:107)调用 virtio_disk_rw 以与磁盘硬件通信。

当调用者完成使用一个缓冲区时，它必须调用 brelse 来释放它。（brelse 的名称，b-release 的缩写，虽然晦涩但值得学习：它起源于 Unix，并在 BSD、Linux 和 Solaris 中使用。）brelse (kernel/bio.c:117) 释放睡眠锁并将缓冲区移动到链表的前面 (kernel/bio.c:128-133)。移动缓冲区会导致链表按缓冲区最近使用顺序排序（即释放顺序）：列表中的第一个缓冲区是最常用的，最后一个是最不常用的。bget 中的两个循环利用了这一点：查找现有缓冲区的扫描必须在最坏情况下处理整个列表，但首先检查最常用的缓冲区（从 bcache.head 开始并跟随后向指针）将减少扫描时间，当引用局部性良好时。选择要重用的缓冲区的扫描通过向后扫描（跟随前向指针）选择最不常用的缓冲区。

8.4 日志层

文件系统设计中一个最有趣的问题是崩溃恢复。这个问题产生的原因是许多文件系统操作涉及对磁盘的多次写入，而在写入子集后崩溃可能会使磁盘上的文件系统处于不一致状态。例如，假设在文件截断（将文件长度设置为零并释放其内容块）期间发生崩溃。根据磁盘写入的顺序，崩溃可能会留下一个引用了被标记为空闲的内容块的 inode，或者留下一个已分配但未被引用的内容块。

后者相对温和，但一个引用已释放块的inode在重启后可能会引发严重问题。重启后，内核可能会将那个块分配给另一个文件，现在我们就有两个不同的文件无意中指向同一个块。如果xv6支持多用户，这种情况可能是一个安全问题，因为旧文件的所有者将能够读取和写入新文件中的块，而新文件属于不同的用户。

and write blocks in the new file, owned by a different user.

Xv6 solves the problem of crashes during file-system operations with a simple form of logging. An xv6 system call does not directly write the on-disk file system data structures. Instead, it places a description of all the disk writes it wishes to make in a *log* on the disk. Once the system call has logged all of its writes, it writes a special *commit* record to the disk indicating that the log contains a complete operation. At that point the system call copies the writes to the on-disk file system data structures. After those writes have completed, the system call erases the log on disk.

If the system should crash and reboot, the file-system code recovers from the crash as follows, before running any processes. If the log is marked as containing a complete operation, then the recovery code copies the writes to where they belong in the on-disk file system. If the log is not marked as containing a complete operation, the recovery code ignores the log. The recovery code finishes by erasing the log.

Why does xv6's log solve the problem of crashes during file system operations? If the crash occurs before the operation commits, then the log on disk will not be marked as complete, the recovery code will ignore it, and the state of the disk will be as if the operation had not even started. If the crash occurs after the operation commits, then recovery will replay all of the operation's writes, perhaps repeating them if the operation had started to write them to the on-disk data structure. In either case, the log makes operations atomic with respect to crashes: after recovery, either all of the operation's writes appear on the disk, or none of them appear.

8.5 Log design

The log resides at a known fixed location, specified in the superblock. It consists of a header block followed by a sequence of updated block copies ("logged blocks"). The header block contains an array of sector numbers, one for each of the logged blocks, and the count of log blocks. The count in the header block on disk is either zero, indicating that there is no transaction in the log, or non-zero, indicating that the log contains a complete committed transaction with the indicated number of logged blocks. Xv6 writes the header block when a transaction commits, but not before, and sets the count to zero after copying the logged blocks to the file system. Thus a crash midway through a transaction will result in a count of zero in the log's header block; a crash after a commit will result in a non-zero count.

Each system call's code indicates the start and end of the sequence of writes that must be atomic with respect to crashes. To allow concurrent execution of file-system operations by different processes, the logging system can accumulate the writes of multiple system calls into one transaction. Thus a single commit may involve the writes of multiple complete system calls. To avoid splitting a system call across transactions, the logging system only commits when no file-system system calls are underway.

The idea of committing several transactions together is known as *group commit*. Group commit reduces the number of disk operations because it amortizes the fixed cost of a commit over multiple operations. Group commit also hands the disk system more concurrent writes at the same time, perhaps allowing the disk to write them all during a single disk rotation. Xv6's virtio driver doesn't support this kind of *batching*, but xv6's file system design allows for it.

和写入块在新文件中，而新文件属于不同的用户。

Xv6 通过一种简单的日志记录方式解决了文件系统操作期间崩溃的问题。一个 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，它会将其希望进行的所有磁盘写入描述放在磁盘上的一个日志中。一旦系统调用记录了所有写入，它就会向磁盘写入一个特殊的提交记录，表明该日志包含一个完整的操作。在这一点上，系统调用将写入复制到磁盘上的文件系统数据结构中。在那些写入完成后，系统调用会擦除磁盘上的日志。

如果系统应该崩溃并重启，文件系统代码在运行任何进程之前会从崩溃中恢复，如下所示。如果日志被标记为包含一个完整的操作，那么恢复代码会将写入复制到它们在磁盘上的文件系统中所属的位置。如果日志没有被标记为包含一个完整的操作，恢复代码会忽略该日志。恢复代码最后会擦除该日志。

为什么 xv6 的日志能解决文件系统操作期间崩溃的问题？如果崩溃发生在操作提交之前，那么磁盘上的日志不会被标记为完成，恢复代码会忽略它，而磁盘的状态将好像操作根本没有开始。如果崩溃发生在操作提交之后，那么恢复会重放操作的所有写入，如果操作已经开始写入磁盘数据结构，可能会重复它们。在任一情况下，日志使操作相对于崩溃变得原子性：恢复后，要么操作的所有写入都出现在磁盘上，要么一个都不出现。

8.5 日志设计

日志位于一个已知的固定位置，在超级块中指定。它由一个头部块和一系列更新的块副本（“已记录块”）组成。头部块包含一个扇区号数组，每个已记录块一个，以及日志块的计数。磁盘上头部块中的计数要么为零，表示日志中没有事务，要么为非零，表示日志包含一个完整的已提交事务，以及指示的已记录块数量。xv6 在事务提交时写入头部块，但在提交之前不写入，并且在将已记录块复制到文件系统后将其计数设置为零。因此，事务中途崩溃会导致日志的头部块计数为零；提交后崩溃会导致计数为非零。

每个系统调用的代码都指示了必须与崩溃保持原子性的写入序列的开始和结束。为了允许不同进程并发执行文件系统操作，日志系统可以将多个系统调用的写入累积到一个事务中。因此，单个提交可能涉及多个完整系统调用的写入。为了避免将系统调用分割到多个事务中，日志系统仅在没有任何文件系统系统调用正在进行时才提交。

将多个事务一起提交的概念称为组提交。组提交减少了磁盘操作的数量，因为它将提交的固定成本分摊到多个操作中。组提交还让磁盘系统在同一时间处理更多的并发写入，也许允许磁盘在单个磁盘旋转期间将它们全部写入。xv6的virtio驱动程序不支持这种批处理，但xv6的文件系统设计允许这样做。

Xv6 dedicates a fixed amount of space on the disk to hold the log. The total number of blocks written by the system calls in a transaction must fit in that space. This has two consequences. No single system call can be allowed to write more distinct blocks than there is space in the log. This is not a problem for most system calls, but two of them can potentially write many blocks: `write` and `unlink`. A large file write may write many data blocks and many bitmap blocks as well as an inode block; unlinking a large file might write many bitmap blocks and an inode. Xv6's write system call breaks up large writes into multiple smaller writes that fit in the log, and `unlink` doesn't cause problems because in practice the xv6 file system uses only one bitmap block. The other consequence of limited log space is that the logging system cannot allow a system call to start unless it is certain that the system call's writes will fit in the space remaining in the log.

8.6 Code: logging

A typical use of the log in a system call looks like this:

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

`begin_op` (kernel/log.c:127) waits until the logging system is not currently committing, and until there is enough unreserved log space to hold the writes from this call. `log.outstanding` counts the number of system calls that have reserved log space; the total reserved space is `log.outstanding` times `MAXOPBLOCKS`. Incrementing `log.outstanding` both reserves space and prevents a commit from occurring during this system call. The code conservatively assumes that each system call might write up to `MAXOPBLOCKS` distinct blocks.

`log_write` (kernel/log.c:215) acts as a proxy for `bwrite`. It records the block's sector number in memory, reserving it a slot in the log on disk, and pins the buffer in the block cache to prevent the block cache from evicting it. The block must stay in the cache until committed: until then, the cached copy is the only record of the modification; it cannot be written to its place on disk until after commit; and other reads in the same transaction must see the modifications. `log_write` notices when a block is written multiple times during a single transaction, and allocates that block the same slot in the log. This optimization is often called *absorption*. It is common that, for example, the disk block containing inodes of several files is written several times within a transaction. By absorbing several disk writes into one, the file system can save log space and can achieve better performance because only one copy of the disk block must be written to disk.

`end_op` (kernel/log.c:147) first decrements the count of outstanding system calls. If the count is now zero, it commits the current transaction by calling `commit()`. There are four stages in this process. `write_log()` (kernel/log.c:179) copies each block modified in the transaction from the buffer cache to its slot in the log on disk. `write_head()` (kernel/log.c:103) writes the header block to disk: this is the commit point, and a crash after the write will result in recovery replaying the

xv6在磁盘上分配了固定数量的空间来保存日志。一个事务中系统调用写入的块总数必须适合该空间。这有两个后果。不允许单个系统调用写入比日志空间更多的不同块。这对大多数系统调用不是问题，但有两个系统调用可能写入许多块：写入和unlink。大文件写入可能会写入许多数据块和许多位图块，以及一个索引节点块；删除大文件可能会写入许多位图块和一个索引节点。xv6的写入系统调用将大写入拆分成多个适合日志的小写入，而unlink不会造成问题，因为实际上xv6文件系统只使用一个位图块。有限日志空间的另一个后果是，日志系统不能允许系统调用开始，除非它确定系统调用的写入将适合日志中剩余的空间。

8.6 代码：日志记录

系统调用中日志的典型用法如下：

begin_op(); ... bp =
bread(...); bp->data[...
] = ...; 日志_写入(bp); ...
end_op();

begin_op (kernel/log.c:127) 等待日志系统当前未提交，并且有足够的未保留日志空间来保存此调用的写入。log.outstanding 计数已保留日志空间的系统调用数量；总保留空间是 log.outstanding 乘以 MAXOPBLOCKS。增加 log.outstanding 既是保留空间，也防止在此系统调用期间发生提交。代码保守地假设每个系统调用可能写入最多 MAXOPBLOCKS 个不同的块。

log_write (kernel/log.c:215) 作为 bwrite 的代理。它将块的扇区号记录在内存中，在磁盘上的日志中为其保留一个槽位，并将缓冲区固定在块缓存中，以防止块缓存将其驱逐。块必须保留在缓存中直到提交：在此之前，缓存的副本是修改的唯一记录；它不能在提交后写入其磁盘上的位置；同一事务中的其他读取必须看到修改。log_write 注意到在单个事务期间块被多次写入，并为该块分配相同的日志槽位。这种优化通常称为吸收。例如，包含多个文件 inode 的磁盘块在事务中多次写入是很常见的。通过将多个磁盘写入吸收为一个，文件系统可以节省日志空间并实现更好的性能，因为只需要将磁盘块的一个副本写入磁盘。

end_op (kernel/log.c:147) 首先减少未完成系统调用的计数。如果计数现在为零，它会通过调用 commit() 提交当前事务。这个过程分为四个阶段。write_log() (kernel/log.c:179) 将事务中修改的每个块从缓冲区缓存复制到磁盘日志上的相应槽位。write_head() (kernel/log.c:103) 将头部块写入磁盘：这是提交点，写入后的崩溃会导致恢复重新播放事务的写入。

transaction's writes from the log. install_trans (kernel/log.c:69) reads each block from the log and writes it to the proper place in the file system. Finally end_op writes the log header with a count of zero; this has to happen before the next transaction starts writing logged blocks, so that a crash doesn't result in recovery using one transaction's header with the subsequent transaction's logged blocks.

recover_from_log (kernel/log.c:117) is called from initlog (kernel/log.c:55), which is called from fsinit(kernel/fs.c:42) during boot before the first user process runs (kernel/proc.c:535). It reads the log header, and mimics the actions of end_op if the header indicates that the log contains a committed transaction.

An example use of the log occurs in filewrite (kernel/file.c:135). The transaction looks like this:

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

This code is wrapped in a loop that breaks up large writes into individual transactions of just a few sectors at a time, to avoid overflowing the log. The call to writei writes many blocks as part of this transaction: the file's inode, one or more bitmap blocks, and some data blocks.

8.7 Code: Block allocator

File and directory content is stored in disk blocks, which must be allocated from a free pool. Xv6's block allocator maintains a free bitmap on disk, with one bit per block. A zero bit indicates that the corresponding block is free; a one bit indicates that it is in use. The program mkfs sets the bits corresponding to the boot sector, superblock, log blocks, inode blocks, and bitmap blocks.

The block allocator provides two functions: balloc allocates a new disk block, and bfree frees a block. balloc The loop in balloc at (kernel/fs.c:72) considers every block, starting at block 0 up to sb.size, the number of blocks in the file system. It looks for a block whose bitmap bit is zero, indicating that it is free. If balloc finds such a block, it updates the bitmap and returns the block. For efficiency, the loop is split into two pieces. The outer loop reads each block of bitmap bits. The inner loop checks all Bits-Per-Block (BPB) bits in a single bitmap block. The race that might occur if two processes try to allocate a block at the same time is prevented by the fact that the buffer cache only lets one process use any one bitmap block at a time.

bfree (kernel/fs.c:92) finds the right bitmap block and clears the right bit. Again the exclusive use implied by bread and brelse avoids the need for explicit locking.

As with much of the code described in the remainder of this chapter, balloc and bfree must be called inside a transaction.

install_trans (kernel/log.c:69) 从日志中读取每个块并将其写入文件系统的正确位置。最后 end_op 将带有零计数的日志头写入；这必须在下一个事务开始写入已记录块之前发生，以防止崩溃导致恢复使用一个事务的头部与后续事务的已记录块。

从_日志 (kernel/log.c:117)中恢复_是被initlog (kernel/log.c:55)调用的，而initlog是在启动时在第一个用户进程运行之前 (kernel/proc.c:535)从fsinit(kernel/fs.c:42)中调用的。它读取日志头，如果头指示日志包含一个已提交的事务，则会模拟end_op的行为。

日志的一个示例用法出现在filewrite (kernel/file.c:135)中。事务看起来像这样：

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

这段代码被包裹在一个循环中，该循环将大写入分解为单个事务，每次只处理少数几个扇区，以避免日志溢出。对writei的调用作为此事务的一部分写入了许多块：文件的inode、一个或多个位图块和一些数据块。

8.7 代码：块分配器

文件和目录内容存储在磁盘块中，这些块必须从一个空闲池中分配。Xv6的块分配器在磁盘上维护一个空闲位图，每个块有一个位。一个零位表示相应的块是空闲的；一个一位表示它正在使用。程序mkfs设置了对应于引导扇区、超级块、日志块、inode块和位图块的位。

块分配器提供两个函数：balloc 分配一个新的磁盘块，和 bfree 释放一个块。balloc 在 (kernel/fs.c:72) 中的循环考虑每一个块，从块 0 开始直到 sb.size，即文件系统中的块数。它查找 bitmap 位为零的块，表示该块是空闲的。如果 balloc 找到这样的块，它更新 bitmap 并返回该块。为了效率，循环被分成两部分。外层循环读取每个 bitmap 位块。内层循环检查单个 bitmap 块中的所有每块位数 (BPB) 位。如果两个进程尝试同时分配一个块可能发生的竞争条件，是通过缓冲区缓存一次只允许一个进程使用任何一个 bitmap 块来防止的。

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

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

8.8 Inode layer

The term *inode* can have one of two related meanings. It might refer to the on-disk data structure containing a file's size and list of data block numbers. Or "inode" might refer to an in-memory inode, which contains a copy of the on-disk inode as well as extra information needed within the kernel.

The on-disk inodes are packed into a contiguous area of disk called the inode blocks. Every inode is the same size, so it is easy, given a number n, to find the nth inode on the disk. In fact, this number n, called the inode number or i-number, is how inodes are identified in the implementation.

The on-disk inode is defined by a `struct dinode` (kernel/fs.h:32). The `type` field distinguishes between files, directories, and special files (devices). A type of zero indicates that an on-disk inode is free. The `nlink` field counts the number of directory entries that refer to this inode, in order to recognize when the on-disk inode and its data blocks should be freed. The `size` field records the number of bytes of content in the file. The `addrs` array records the block numbers of the disk blocks holding the file's content.

The kernel keeps the set of active inodes in memory in a table called `itable`; `struct inode` (kernel/file.h:17) is the in-memory copy of a `struct dinode` on disk. The kernel stores an inode in memory only if there are C pointers referring to that inode. The `ref` field counts the number of C pointers referring to the in-memory inode, and the kernel discards the inode from memory if the reference count drops to zero. The `iget` and `iput` functions acquire and release pointers to an inode, modifying the reference count. Pointers to an inode can come from file descriptors, current working directories, and transient kernel code such as `exec`.

There are four lock or lock-like mechanisms in xv6's inode code. `itable.lock` protects the invariant that an inode is present in the inode table at most once, and the invariant that an in-memory inode's `ref` field counts the number of in-memory pointers to the inode. Each in-memory inode has a `lock` field containing a sleep-lock, which ensures exclusive access to the inode's fields (such as file length) as well as to the inode's file or directory content blocks. An inode's `ref`, if it is greater than zero, causes the system to maintain the inode in the table, and not re-use the table entry for a different inode. Finally, each inode contains a `nlink` field (on disk and copied in memory if in memory) that counts the number of directory entries that refer to a file; xv6 won't free an inode if its link count is greater than zero.

A `struct inode` pointer returned by `iget()` is guaranteed to be valid until the corresponding call to `iput()`; the inode won't be deleted, and the memory referred to by the pointer won't be re-used for a different inode. `iget()` provides non-exclusive access to an inode, so that there can be many pointers to the same inode. Many parts of the file-system code depend on this behavior of `iget()`, both to hold long-term references to inodes (as open files and current directories) and to prevent races while avoiding deadlock in code that manipulates multiple inodes (such as pathname lookup).

The `struct inode` that `iget` returns may not have any useful content. In order to ensure it holds a copy of the on-disk inode, code must call `ilock`. This locks the inode (so that no other process can `ilock` it) and reads the inode from the disk, if it has not already been read. `iunlock` releases the lock on the inode. Separating acquisition of inode pointers from locking helps avoid deadlock in some situations, for example during directory lookup. Multiple processes can hold a

8.8 i节点层

术语 inode 可以有两种相关的含义。它可能指的是包含文件大小和数据块号列表的磁盘数据结构。或者，“inode”可能指的是内存 inode，它包含磁盘 inode 的副本以及内核中所需的额外信息。

磁盘上的i节点被压缩到一个连续的磁盘区域中，称为索引节点块。每个i节点的大小相同，因此给定一个数字n，很容易找到磁盘上的第n个i节点。事实上，这个数字n，称为i节点编号或ino，显示了i节点在实现中的标识方式。

磁盘上的i节点由struct dinode (kernel/fs.h:32)定义。类型字段区分文件、目录和特殊文件（设备）。类型为零表示磁盘上的i节点是空闲的。nlink字段计算引用此i节点的目录条目数量，以便识别何时应释放磁盘上的i节点及其数据块。size字段记录文件中内容的字节数。addrs数组记录存储文件内容的磁盘块的块号。

内核将活动i节点集保存在内存中的一个称为itable的表中；struct inode (kernel/file.h:17)是磁盘上struct dinode的内存副本。内核只有在有C指针引用该i节点时才会将i节点存储在内存中。ref字段计算引用内存中i节点的C指针数量，如果引用计数降至零，内核会丢弃内存中的i节点。iget和iput函数获取和释放对i节点的指针，修改引用计数。对i节点的指针可以来自文件描述符、当前工作目录和exec等瞬态内核代码。

xv6的inode代码中有四个锁或类似锁的机制。itable.lock保护两个不变量：一个inode最多一次出现在inode表中，以及内存中inode的ref字段计数内存中指向该inode的指针数量。每个内存中inode都有一个包含睡眠锁的锁字段，这确保了对inode字段（如文件长度）以及inode的文件或目录内容块的独占访问。如果一个inode的ref大于零，system会维护该inode在表中，并且不重用表项用于不同的inode。最后，每个inode包含一个nlink字段（磁盘上和如果内存中也复制），该字段计数引用文件的目录条目数量；如果其链接计数大于零，xv6不会释放该inode。

由iget()返回的struct inode指针在对应的iput()调用期间保证有效；该inode不会被删除，并且指针所引用的内存不会被重用于不同的inode。iget()提供对inode的非独占访问，因此可以有多个指针指向同一个inode。文件系统代码的许多部分依赖于iget()的这种行为，既要持有对inode的长期引用（如打开的文件和当前目录），也要在避免在操作多个inode的代码（如路径名查找）中出现死锁时防止竞争。

iget返回的struct inode可能没有任何有用内容。为了确保它包含磁盘inode的副本，代码必须调用ilock。这会锁定inode（以便其他进程不能ilock它），并且如果ithas尚未已读，则从磁盘读取inode。iunlock释放对inode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

C pointer to an inode returned by `iget`, but only one process can lock the inode at a time.

The inode table only stores inodes to which kernel code or data structures hold C pointers. Its main job is synchronizing access by multiple processes. The inode table also happens to cache frequently-used inodes, but caching is secondary; if an inode is used frequently, the buffer cache will probably keep it in memory. Code that modifies an in-memory inode writes it to disk with `iupdate`.

8.9 Code: Inodes

To allocate a new inode (for example, when creating a file), xv6 calls `ialloc` (kernel/fs.c:199). `ialloc` is similar to `balloc`: it loops over the inode structures on the disk, one block at a time, looking for one that is marked free. When it finds one, it claims it by writing the new `type` to the disk and then returns an entry from the inode table with the tail call to `iget` (kernel/fs.c:213). The correct operation of `ialloc` depends on the fact that only one process at a time can be holding a reference to `bp`: `ialloc` can be sure that some other process does not simultaneously see that the inode is available and try to claim it.

`iget` (kernel/fs.c:247) looks through the inode table for an active entry (`ip->ref > 0`) with the desired device and inode number. If it finds one, it returns a new reference to that inode (kernel/fs.c:256-260). As `iget` scans, it records the position of the first empty slot (kernel/fs.c:261-262), which it uses if it needs to allocate a table entry.

Code must lock the inode using `ilock` before reading or writing its metadata or content. `ilock` (kernel/fs.c:293) uses a sleep-lock for this purpose. Once `ilock` has exclusive access to the inode, it reads the inode from disk (more likely, the buffer cache) if needed. The function `iunlock` (kernel/fs.c:321) releases the sleep-lock, which may cause any processes sleeping to be woken up.

`iput` (kernel/fs.c:337) releases a C pointer to an inode by decrementing the reference count (kernel/fs.c:360). If this is the last reference, the inode's slot in the inode table is now free and can be re-used for a different inode.

If `iput` sees that there are no C pointer references to an inode and that the inode has no links to it (occurs in no directory), then the inode and its data blocks must be freed. `iput` calls `itrunc` to truncate the file to zero bytes, freeing the data blocks; sets the inode type to 0 (unallocated); and writes the inode to disk (kernel/fs.c:342).

The locking protocol in `iput` in the case in which it frees the inode deserves a closer look. One danger is that a concurrent thread might be waiting in `ilock` to use this inode (e.g., to read a file or list a directory), and won't be prepared to find that the inode is no longer allocated. This can't happen because there is no way for a system call to get a pointer to an in-memory inode if it has no links to it and `ip->ref` is one. That one reference is the reference owned by the thread calling `iput`. The other main danger is that a concurrent call to `ialloc` might choose the same inode that `iput` is freeing. This can happen only after the `iupdate` writes the disk so that the inode has type zero. This race is benign; the allocating thread will politely wait to acquire the inode's sleep-lock before reading or writing the inode, at which point `iput` is done with it.

`iput()` can write to the disk. This means that any system call that uses the file system may write to the disk, because the system call may be the last one having a reference to the file. Even

由iget返回的指向inode的C指针，但一次只有一个进程可以锁定inode。

inode表仅存储哪些内核代码或数据结构持有C指针的inode。它的主要工作是同步多个进程的访问。inode表还会缓存频繁使用的inode，但缓存是次要的；如果一个inode被频繁使用，缓冲区缓存可能会将其保留在内存中。修改内存inode的代码会使用iupdate将其写入磁盘。

8.9 代码：inode

要分配一个新的inode（例如，在创建文件时），xv6会调用ialloc（kernel/fs.c:199）。ialloc与balloc类似：它会逐块遍历磁盘上的inode结构，寻找一个标记为空闲的。找到后，它会通过将新类型写入磁盘来声明它，然后返回inode表中的一个条目，并使用尾调用到iget（kernel/fs.c:213）。ialloc的正确操作依赖于这样一个事实：一次只有一个进程可以持有对bp的引用：ialloc可以确信没有其他进程同时看到inode是可用的并试图声明它。

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

在读取或写入其元数据或内容之前，代码必须使用 ilock 锁定 inode。ilock (kernel/fs.c:293) 使用睡眠锁为此目的。一旦 ilock 对 inode 具有独占访问权，如果需要，它就会从磁盘（更可能是缓冲区缓存）读取 inode。函数 iunlock (kernel/fs.c:321) 释放睡眠锁，这可能会唤醒任何正在睡眠的进程。

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

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

iput 在释放 inode 时的锁定协议值得更仔细地查看。一个风险是，一个并发线程可能正在 ilock 中等待使用这个 inode（例如，读取文件或列出目录），并且不会准备好发现该 inode 已不再分配。这不可能发生，因为如果它没有链接并且 ip->ref 是一个，系统调用没有办法获取内存中 inode 的指针。这个引用是调用 iput 的线程拥有的引用。另一个主要风险是，一个并发调用到 ialloc 可能会选择 iput 正在释放的同一个 inode。这只能发生在 iupdate 将磁盘写入类型为零的 inode 之后。这种竞争条件是无害的；分配线程会礼貌地等待获取 inode 的睡眠锁，然后再读取或写入 inode，此时 iput 已经处理完它了。

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。甚至

calls like `read()` that appear to be read-only, may end up calling `iput()`. This, in turn, means that even read-only system calls must be wrapped in transactions if they use the file system.

There is a challenging interaction between `iput()` and crashes. `iput()` doesn't truncate a file immediately when the link count for the file drops to zero, because some process might still hold a reference to the inode in memory: a process might still be reading and writing to the file, because it successfully opened it. But, if a crash happens before the last process closes the file descriptor for the file, then the file will be marked allocated on disk but no directory entry will point to it.

File systems handle this case in one of two ways. The simple solution is that on recovery, after reboot, the file system scans the whole file system for files that are marked allocated, but have no directory entry pointing to them. If any such file exists, then it can free those files.

The second solution doesn't require scanning the file system. In this solution, the file system records on disk (e.g., in the super block) the inode inumber of a file whose link count drops to zero but whose reference count isn't zero. If the file system removes the file when its reference count reaches 0, then it updates the on-disk list by removing that inode from the list. On recovery, the file system frees any file in the list.

Xv6 implements neither solution, which means that inodes may be marked allocated on disk, even though they are not in use anymore. This means that over time xv6 runs the risk that it may run out of disk space.

8.10 Code: Inode content

The on-disk inode structure, `struct dinode`, contains a size and an array of block numbers (see Figure 8.3). The inode data is found in the blocks listed in the `dinode`'s `addrs` array. The first `NDIRECT` blocks of data are listed in the first `NDIRECT` entries in the array; these blocks are called *direct blocks*. The next `NINDIRECT` blocks of data are listed not in the inode but in a data block called the *indirect block*. The last entry in the `addrs` array gives the address of the indirect block. Thus the first 12 kB (`NDIRECT x BSIZE`) bytes of a file can be loaded from blocks listed in the inode, while the next `256` kB (`NINDIRECT x BSIZE`) bytes can only be loaded after consulting the indirect block. This is a good on-disk representation but a complex one for clients. The function `bmap` manages the representation so that higher-level routines, such as `readi` and `writei`, which we will see shortly, do not need to manage this complexity. `bmap` returns the disk block number of the `bn`'th data block for the inode `ip`. If `ip` does not have such a block yet, `bmap` allocates one.

The function `bmap` (kernel/fs.c:383) begins by picking off the easy case: the first `NDIRECT` blocks are listed in the inode itself (kernel/fs.c:388-396). The next `NINDIRECT` blocks are listed in the indirect block at `ip->addrs[NDIRECT]`. `bmap` reads the indirect block (kernel/fs.c:407) and then reads a block number from the right position within the block (kernel/fs.c:408). If the block number exceeds `NDIRECT+NINDIRECT`, `bmap` panics; `writei` contains the check that prevents this from happening (kernel/fs.c:513).

`bmap` allocates blocks as needed. An `ip->addrs[]` or indirect entry of zero indicates that no block is allocated. As `bmap` encounters zeros, it replaces them with the numbers of fresh blocks, allocated on demand (kernel/fs.c:389-390) (kernel/fs.c:401-402).

`itrunc` frees a file's blocks, resetting the inode's size to zero. `itrunc` (kernel/fs.c:426) starts by

像 read() 这样看似只读的调用，最终可能会调用 iput()。这反过来又意味着，如果它们使用文件系统，即使是只读的系统调用也必须用事务包装。

iput() 和崩溃之间存在一个具有挑战性的交互。当文件的链接计数降至零时，i put() 不会立即截断 afile，因为某些进程可能仍在内存中持有对 inode 的引用：因为进程成功打开了文件，所以它可能仍在读取和写入文件。但是，如果在最后一个进程关闭文件描述符之前发生崩溃，那么文件将在磁盘上被标记为分配的，但没有目录项指向它。

文件系统以两种方式处理此情况。简单的解决方案是，在恢复后重新启动时，文件系统会扫描整个文件系统，查找被标记为分配但没有目录项指向它们的文件。如果存在任何此类文件，则可以释放这些文件。

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

Xv6 实现了上述两种解决方案中的任何一种，这意味着 i节点可能会在磁盘上被标记为已分配，即使它们不再使用。这意味着随着时间的推移，xv6 运行的风险是磁盘空间可能用尽。

8.10 代码：i节点内容

磁盘上的 i节点结构 struct dinode 包含一个大小和一个块号数组（见图 8.3）。i节点数据位于 dinode 的 addrs 数组中列出的块中。前 NDIRECT 个数据块列在数组的前 NDIRECT 个条目中；这些块称为直接块。接下来的 NINDIRECT 个数据块不在 i节点中，而是在一个称为间接块的块中列出。addrs 数组的最后一个条目给出间接块的地址。因此，文件的前 12 kB（NDIRECT x BSIZE）字节可以从 i节点中列出的块中加载，而接下来的 256 kB （ NINDIRECT x BSIZE ）字节只能在使用间接块后加载。这是一种好的磁盘表示，但对客户端来说很复杂。函数 bmap 管理这种表示，以便更高级别的例程（如我们很快将看到的 readi 和 writei）不需要管理这种复杂性。bmap 返回 i节点 ip 的第 bn 个数据块的磁盘块号。如果 ip 尚未具有这样的块，bmap 将为其分配一个。

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

bmap 按需分配块。一个 ip->addrs[] 或间接条目为零表示没有块被分配。当 bmap 遇到零时，它会用新分配的块号替换它们，这些块按需分配 (kernel/fs.c:389-390) (kernel/fs.c:401-402)。

itrunc 释放文件的块，将 inode 的大小重置为零。itrunc (kernel/fs.c:426) 首先开始

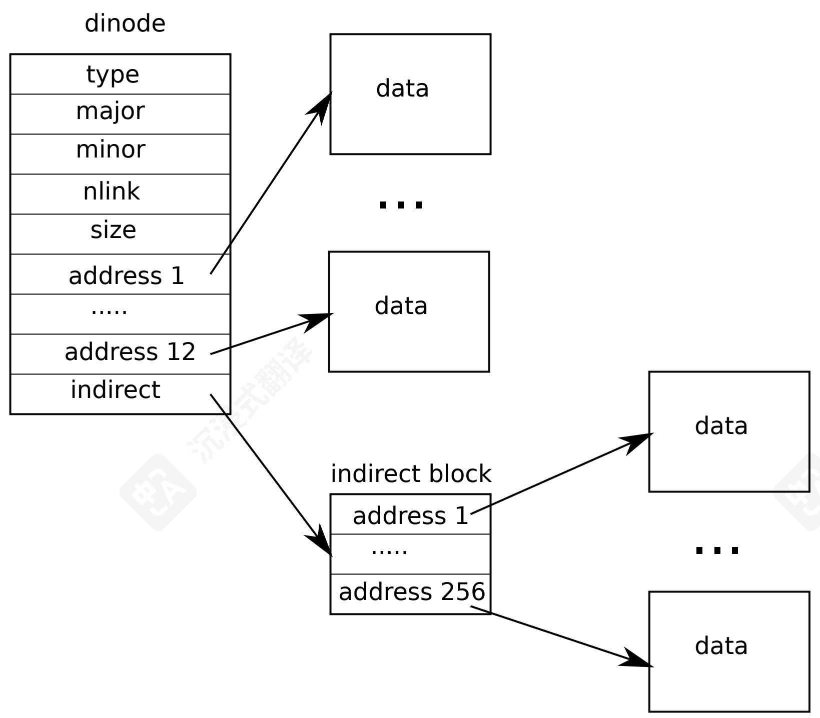

Figure 8.3: The representation of a file on disk.

freeing the direct blocks (kernel/fs.c:432-437), then the ones listed in the indirect block (kernel/fs.c:442-445), and finally the indirect block itself (kernel/fs.c:447-448).

bmap makes it easy for readi and writei to get at an inode's data. readi (kernel/fs.c:472) starts by making sure that the offset and count are not beyond the end of the file. Reads that start beyond the end of the file return an error (kernel/fs.c:477-478) while reads that start at or cross the end of the file return fewer bytes than requested (kernel/fs.c:479-480). The main loop processes each block of the file, copying data from the buffer into dst (kernel/fs.c:482-494). writei (kernel/fs.c:506) is identical to readi, with three exceptions: writes that start at or cross the end of the file grow the file, up to the maximum file size (kernel/fs.c:513-514); the loop copies data into the buffers instead of out (kernel/fs.c:522); and if the write has extended the file, writei must update its size (kernel/fs.c:530-531).

The function stati (kernel/fs.c:458) copies inode metadata into the stat structure, which is exposed to user programs via the stat system call.

8.11 Code: directory layer

A directory is implemented internally much like a file. Its inode has type T_DIR and its data is a sequence of directory entries. Each entry is a struct dirent (kernel/fs.h:56), which contains a

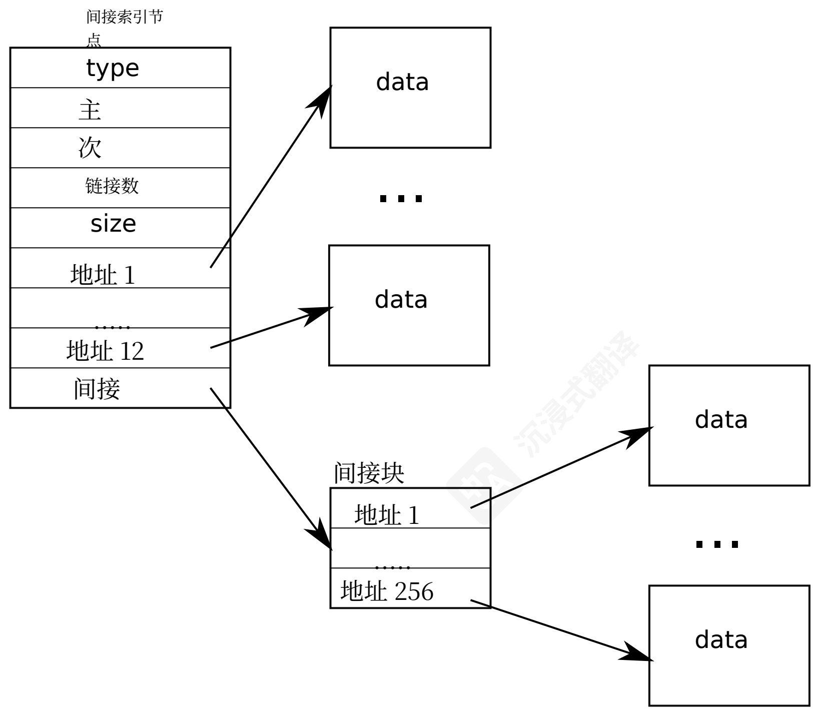

图8.3：磁盘上的文件表示。

释放直接块 (kernel/fs.c:432-437)，然后是间接块中列出的块(kernel/fs.c:442- 445)，最后是间接块本身 (kernel/fs.c:447-448)。

bmap使得readi和writei能够轻松访问inode的数据。readi (kernel/fs.c:472) 首先确保偏移量和计数不超过文件末尾。从文件末尾开始或跨越文件末尾的读取会返回错误 (kernel/fs.c:477-478)，而从文件末尾开始或跨越文件末尾的读取会返回少于请求的字节数 (kernel/fs.c:479-480)。主循环处理文件的每个块，将数据从缓冲区复制到dst (kernel/fs.c:482-494)。writei (kernel/fs.c:506) 与readi相同，有三个例外：从文件末尾开始或跨越文件末尾的写入会扩展文件，直到最大文件大小 (kernel/fs.c:513-514)；循环将数据复制到缓冲区而不是外部 (kernel/fs.c:522)；如果写入扩展了文件，writei必须更新其大小 (kernel/fs.c:530-531)。

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

8.11 代码：目录层

目录在内部实现与文件非常相似。其inode的类型为T_DIR，其数据是一系列目录条目。每个条目是一个struct dirent (kernel/fs.h:56)，其中包含一个

name and an inode number. The name is at most DIRSIZ (14) characters; if shorter, it is terminated by a NULL (0) byte. Directory entries with inode number zero are free.

The function dirlookup (kernel/fs.c:552) searches a directory for an entry with the given name. If it finds one, it returns a pointer to the corresponding inode, unlocked, and sets *poff to the byte offset of the entry within the directory, in case the caller wishes to edit it. If dirlookup finds an entry with the right name, it updates *poff and returns an unlocked inode obtained via iget. dirlookup is the reason that iget returns unlocked inodes. The caller has locked dp, so if the lookup was for ., an alias for the current directory, attempting to lock the inode before returning would try to re-lock dp and deadlock. (There are more complicated deadlock scenarios involving multiple processes and .., an alias for the parent directory; . is not the only problem.) The caller can unlock dp and then lock ip, ensuring that it only holds one lock at a time.

The function dirlink (kernel/fs.c:580) writes a new directory entry with the given name and inode number into the directory dp. If the name already exists, dirlink returns an error (kernel/fs.c:586-590). The main loop reads directory entries looking for an unallocated entry. When it finds one, it stops the loop early (kernel/fs.c:592-597), with off set to the offset of the available entry. Otherwise, the loop ends with off set to dp->size. Either way, dirlink then adds a new entry to the directory by writing at offset off (kernel/fs.c:602-603).

8.12 Code: Path names

Path name lookup involves a succession of calls to dirlookup, one for each path component. namei (kernel/fs.c:687) evaluates path and returns the corresponding inode. The function nameiparent is a variant: it stops before the last element, returning the inode of the parent directory and copying the final element into name. Both call the generalized function namex to do the real work.

namex (kernel/fs.c:652) starts by deciding where the path evaluation begins. If the path begins with a slash, evaluation begins at the root; otherwise, the current directory (kernel/fs.c:656-659). Then it uses skipelem to consider each element of the path in turn (kernel/fs.c:661). Each iteration of the loop must look up name in the current inode ip. The iteration begins by locking ip and checking that it is a directory. If not, the lookup fails (kernel/fs.c:662-666). (Locking ip is necessary not because ip->type can change underfoot—it can't—but because until ilock runs, ip->type is not guaranteed to have been loaded from disk.) If the call is nameiparent and this is the last path element, the loop stops early, as per the definition of nameiparent; the final path element has already been copied into name, so namex need only return the unlocked ip (kernel/fs.c:667-671). Finally, the loop looks for the path element using dirlookup and prepares for the next iteration by setting ip = next (kernel/fs.c:672-677). When the loop runs out of path elements, it returns ip.

The procedure namex may take a long time to complete: it could involve several disk operations to read inodes and directory blocks for the directories traversed in the pathname (if they are not in the buffer cache). Xv6 is carefully designed so that if an invocation of namex by one kernel thread is blocked on a disk I/O, another kernel thread looking up a different pathname can proceed concurrently. namex locks each directory in the path separately so that lookups in different directories can proceed in parallel.

This concurrency introduces some challenges. For example, while one kernel thread is looking

名称和一个inode编号。名称最多为DIRSIZ (14) 个字符；如果更短，则由一个NULL (0) 字节终止。具有零inode编号的目录条目是空闲的。

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

函数 dirlink (kernel/fs.c:580) 将具有给定名称和 inode 编号的新的目录条目写入目录 dp。如果名称已存在，dirlink 返回错误 (kernel/fs.c:586-590)。主循环读取目录条目，寻找未分配的条目。当找到时，它会提前停止循环 (kernel/fs.c:592-597)，并将 off 设置为可用条目的偏移量。否则，循环结束，off 设置为 dp->size。无论如何，dirlink 然后通过在偏移量 off 处写入来向目录添加一个新条目 (kernel/fs.c:602-603)。

8.12 代码：路径名

路径名查找涉及一系列对 dirlookup 的调用，每个路径组件调用一次。namei (kernel/fs.c:687) 评估路径并返回相应的 inode。函数 nameiparent 是一个变体：它在最后一个元素之前停止，返回父目录的 inode 并将最后一个元素复制到 name 中。两者都调用通用的函数 namex 来执行实际工作。

namex (kernel/fs.c:652) 首先决定路径评估的起始位置。如果路径以斜杠开头，评估从根开始；否则，从当前目录 (kernel/fs.c:656-659)。然后它使用 skipelem 依次考虑路径的每个元素 (kernel/fs.c:661)。循环的每次迭代都必须在当前 inode ip 中查找 name。迭代开始时锁定 ip 并检查它是否为目录。如果不是，查找失败 (kernel/fs.c:662-666)。(锁定 ip 不是因为它->类型可能会突然改变——它不会——而是因为在 ilock 运行之前，ip->type 不保证已从磁盘加载。) 如果调用是 nameiparent 且这是最后一个路径元素，循环会根据 nameiparent 的定义提前停止；最后一个路径元素已经复制到 name 中，因此 namex 只需返回未锁定的 ip (kernel/fs.c:667-671)。最后，循环使用 dirlookup 查找路径元素，并通过设置 ip = next 为下一次迭代做准备 (kernel/fs.c:672-677)。当循环用完路径元素时，它返回 ip。

The procedure namex 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遍历的目录的 inode 和目录块（如果它们不在缓冲区缓存中）。Xv6 经过精心设计，使得如果一个内核线程的 namex 调用被磁盘 I/O 阻塞，另一个查找不同路径名的内核线程可以并发进行。namex 分别锁定路径中的每个目录，以便不同目录中的查找可以并行进行。

这种并发引入了一些挑战。例如，当一个内核线程正在查找...

up a pathname another kernel thread may be changing the directory tree by unlinking a directory. A potential risk is that a lookup may be searching a directory that has been deleted by another kernel thread and its blocks have been re-used for another directory or file.

Xv6 avoids such races. For example, when executing `dirlookup` in `namex`, the lookup thread holds the lock on the directory and `dirlookup` returns an inode that was obtained using `iget`. `iget` increases the reference count of the inode. Only after receiving the inode from `dirlookup` does `namex` release the lock on the directory. Now another thread may unlink the inode from the directory but xv6 will not delete the inode yet, because the reference count of the inode is still larger than zero.

Another risk is deadlock. For example, `next` points to the same inode as `ip` when looking up ".". Locking `next` before releasing the lock on `ip` would result in a deadlock. To avoid this deadlock, `namex` unlocks the directory before obtaining a lock on `next`. Here again we see why the separation between `iget` and `ilock` is important.

8.13 File descriptor layer

A cool aspect of the Unix interface is that most resources in Unix are represented as files, including devices such as the console, pipes, and of course, real files. The file descriptor layer is the layer that achieves this uniformity.

Xv6 gives each process its own table of open files, or file descriptors, as we saw in Chapter 1. Each open file is represented by a `struct file` (kernel/file.h:1), which is a wrapper around either an inode or a pipe, plus an I/O offset. Each call to `open` creates a new open file (a new `struct file`): if multiple processes open the same file independently, the different instances will have different I/O offsets. On the other hand, a single open file (the same `struct file`) can appear multiple times in one process's file table and also in the file tables of multiple processes. This would happen if one process used `open` to open the file and then created aliases using `dup` or shared it with a child using `fork`. A reference count tracks the number of references to a particular open file. A file can be open for reading or writing or both. The `readable` and `writable` fields track this.

All the open files in the system are kept in a global file table, the `ftable`. The file table has functions to allocate a file (`filealloc`), create a duplicate reference (`filedup`), release a reference (`fileclose`), and read and write data (`fileread` and `filewrite`).

The first three follow the now-familiar form. `filealloc` (kernel/file.c:30) scans the file table for an unreferenced file (`f->ref == 0`) and returns a new reference; `filedup` (kernel/file.c:48) increments the reference count; and `fileclose` (kernel/file.c:60) decrements it. When a file's reference count reaches zero, `fileclose` releases the underlying pipe or inode, according to the type.

The functions `filestat`, `fileread`, and `filewrite` implement the `stat`, `read`, and `write` operations on files. `filestat` (kernel/file.c:88) is only allowed on inodes and calls `stati`. `fileread` and `filewrite` check that the operation is allowed by the open mode and then pass the call through to either the pipe or inode implementation. If the file represents an inode, `fileread` and `filewrite` use the I/O offset as the offset for the operation and then advance it (kernel/file.c:122-123) (kernel/file.c:153-154). Pipes have no concept of offset. Recall that the inode functions require

这种并发引入了一些挑战。例如，当一个内核线程正在查找一个路径名时，另一个内核线程可能会通过删除一个目录来更改目录树。一个潜在的风险是，查找操作可能会在一个已被另一个内核线程删除的目录中进行搜索，而该目录的块已被重新用于另一个目录或文件。

Xv6 避免了这种竞争条件。例如，当在 namex 中执行 dirlookup 时，查找线程会持有目录的锁，并且 dirlookup 会返回一个使用 iget 获取的 inode。iget 会增加 i node 的引用计数。只有在从 dirlookup 接收到 inode 之后，namex 才会释放目录的锁。现在另一个线程可以 unlink 目录中的 inode，但 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，在查找 "." 时，next 指向与 ip 相同的 inode。在释放 i p 的锁之前锁定 next 会导致死锁。为了避免这种死锁，namex 在获取 next 的锁之前会解锁目录。这里再次看到为什么 iget 和 ilock 之间的分离很重要。

8.13 文件描述符层

Unix 接口的一个酷的方面是 Unix 中的大多数资源都表示为文件，包括控制台、管道以及当然的普通文件。文件描述符层是实现这种统一性的层。

Xv6 为每个进程提供其自己的打开文件表，或文件描述符表，正如我们在第一章中看到的。每个打开的文件由一个 struct file (kernel/file.h:1) 表示，这是一个围绕 inode 或管道的包装器，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）：如果多个进程独立打开同一个文件，不同的实例将会有不同的 I/O 偏移量。另一方面，一个单一的打开文件（相同的 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。这会发生在进程使用 open 打开文件后，然后使用 dup 创建别名，或使用 fork 与子进程共享文件时。一个引用计数跟踪对特定打开文件的引用数量。文件可以用于读取、写入或两者。可读和可写字段跟踪这一点。

系统中的所有打开文件都保存在全局文件表 ftable 中。文件表有分配文件（filealloc）、创建重复引用（filedup）、释放引用（fileclose）以及读写数据（filereadand filewrite）的函数。

前三个遵循了现在熟悉的格式。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；而 fileclose (kernel/file.c:60) 减少它。当文件的引用计数达到零时，fileclose 根据类型释放底层的管道或 inode。

函数filestat、fileread和filewrite实现了对文件的stat、读取和写入操作。filestat (kernel/file.c:88) 仅限于inode，并调用callsstati。fileread和filewrite检查操作是否根据打开模式允许，然后将调用传递给管道或inode实现。如果文件表示一个inode，fileread和filewrite使用I/O偏移量作为操作的偏移量，然后将其前进（kernel/file.c:122-123）（kernel/file.c:153-154）。管道没有偏移量的概念。回想一下，inode函数需要

the caller to handle locking (kernel/file.c:94-96) (kernel/file.c:121-124) (kernel/file.c:163-166). The inode locking has the convenient side effect that the read and write offsets are updated atomically, so that multiple writing to the same file simultaneously cannot overwrite each other's data, though their writes may end up interlaced.

8.14 Code: System calls

With the functions that the lower layers provide, the implementation of most system calls is trivial (see (kernel/sysfile.c)). There are a few calls that deserve a closer look.

The functions `sys_link` and `sys_unlink` edit directories, creating or removing references to inodes. They are another good example of the power of using transactions. `sys_link` (kernel/sysfile.c:124) begins by fetching its arguments, two strings `old` and `new` (kernel/sysfile.c:129). Assuming `old` exists and is not a directory (kernel/sysfile.c:133-136), `sys_link` increments its `ip->nlink` count. Then `sys_link` calls `nameiparent` to find the parent directory and final path element of `new` (kernel/sysfile.c:149) and creates a new directory entry pointing at `old`'s inode (kernel/sysfile.c:152). The new parent directory must exist and be on the same device as the existing inode: inode numbers only have a unique meaning on a single disk. If an error like this occurs, `sys_link` must go back and decrement `ip->nlink`.

Transactions simplify the implementation because it requires updating multiple disk blocks, but we don't have to worry about the order in which we do them. They either will all succeed or none. For example, without transactions, updating `ip->nlink` before creating a link, would put the file system temporarily in an unsafe state, and a crash in between could result in havoc. With transactions we don't have to worry about this.

`sys_link` creates a new name for an existing inode. The function `create` (kernel/sysfile.c:246) creates a new name for a new inode. It is a generalization of the three file creation system calls: `open` with the `O_CREATE` flag makes a new ordinary file, `mkdir` makes a new directory, and `mkdev` makes a new device file. Like `sys_link`, `create` starts by calling `nameiparent` to get the inode of the parent directory. It then calls `dirlookup` to check whether the name already exists (kernel/sysfile.c:256). If the name does exist, `create`'s behavior depends on which system call it is being used for: `open` has different semantics from `mkdir` and `mkdev`. If `create` is being used on behalf of `open` (`type == T_FILE`) and the name that exists is itself a regular file, then `open` treats that as a success, so `create` does too (kernel/sysfile.c:260). Otherwise, it is an error (kernel/sysfile.c:261-262). If the name does not already exist, `create` now allocates a new inode with `ialloc` (kernel/sysfile.c:265). If the new inode is a directory, `create` initializes it with `.` and `..` entries. Finally, now that the data is initialized properly, `create` can link it into the parent directory (kernel/sysfile.c:278). `create`, like `sys_link`, holds two inode locks simultaneously: `ip` and `dp`. There is no possibility of deadlock because the inode `ip` is freshly allocated: no other process in the system will hold `ip`'s lock and then try to lock `dp`.

Using `create`, it is easy to implement `sys_open`, `sys_mkdir`, and `sys_mknod`. `sys_open` (kernel/sysfile.c:305) is the most complex, because creating a new file is only a small part of what it can do. If `open` is passed the `O_CREATE` flag, it calls `create` (kernel/sysfile.c:320). Otherwise, it calls `namei` (kernel/sysfile.c:326). `create` returns a locked inode, but `namei` does not, so `sys_open`

调用者处理锁定 (kernel/file.c:94-96) (kernel/file.c:121-124) (kernel/file.c:163-166)。inode锁定有一个方便的副作用，即读取和写入偏移量原子性更新，因此多个同时写入同一文件不会互相覆盖数据，尽管它们的写入可能会交错。

8.14 代码：系统调用

凭借下层提供的函数，大多数系统调用的实现都很简单（参见(kernel/sysfile.c)）。有几个调用值得更仔细地研究。

函数sys_link和sys_unlink编辑目录，创建或删除对inode的引用。它们是使用事务强大功能的另一个好例子。sys_link (kernel/sys-file.c:124) 首先获取其参数，两个字符串old和new (kernel/sysfile.c:129)。假设old存在且不是目录 (kernel/sysfile.c:133-136)，sys_link增加其ip->nlink计数。然后sys_link调用nameiparent来找到new的父目录和最终路径元素 (kernel/sysfile.c:149)，并创建一个指向old的inode的新目录条目 (kernel/sys-file.c:152)。新的父目录必须存在，并且与现有的inode在同一设备上：inode编号只在单个磁盘上才有唯一意义。如果发生这种错误，sys_link必须回去并减少ip->nlink。

事务简化了实现，因为它需要更新多个磁盘块，但我们不必担心执行它们的顺序。它们要么全部成功，要么全部失败。例如，如果没有事务，在创建链接之前更新 ip->nlink，会使文件系统暂时处于不安全状态，而中间发生的崩溃可能会导致混乱。有了事务，我们就不必担心这一点。

sys_link 为现有的 inode 创建一个新名称。函数 create (kernel/sysfile.c:246) 为新的 inode 创建一个新名称。它是三个文件创建系统调用的一般化：使用 O_CREATE 标志的 open 创建一个新的普通文件，mkdir 创建一个新的目录，mkdev 创建一个新的设备文件。与 sys_link 类似，create 首先调用 nameiparent 获取父目录的 inode。然后调用 dirlookup 检查名称是否已存在 (kernel/sysfile.c:256)。如果名称已存在，create 的行为取决于它正在为哪个系统调用使用：open 与 mkdir 和 mkdev 具有不同的语义。如果 create 是代表 open (类型 == T_FILE) 使用的，并且存在的名称本身是一个普通文件，那么 open 将其视为成功，因此 create 也这样 (kernel/sysfile.c:260)。否则，它是一个错误 (kernel/sysfile.c:261-262)。如果名称不存在，create 现在用 ialloc (kernel/sysfile.c:265) 分配一个新的 inode。如果新的 inode 是目录，create 用 . 和 .. 条目初始化它。最后，现在数据已经正确初始化，create 可以将其链接到父目录 (kernel/sysfile.c:278)。create，与 sys_link 类似，同时持有两个 inode 锁：ip 和 dp。由于 inode ip 是新鲜分配的，因此没有死锁的可能性：系统中的其他进程不会持有 ip 的锁，然后尝试锁定 dp。

使用创建，可以轻松实现 sys_open、sys_mkdir 和 sys_mknod。sys_open (kernel/sysfile.c:305) 是最复杂的，因为创建新文件只是它能做的事情的一小部分。如果 open 传递了 O_CREATE 标志，它会调用创建 (kernel/sysfile.c:320)。否则，它会调用 namei (kernel/sysfile.c:326)。创建返回一个锁定的 inode，但 namei 不返回，所以 sys_open must lock the inode itself. This provides a convenient place to check that directories are only opened for reading, not writing. Assuming the inode was obtained one way or the other, `sys_open` allocates a file and a file descriptor (kernel/sysfile.c:344) and then fills in the file (kernel/sysfile.c:356-361). Note that no other process can access the partially initialized file since it is only in the current process's table.

Chapter 7 examined the implementation of pipes before we even had a file system. The function `sys_pipe` connects that implementation to the file system by providing a way to create a pipe pair. Its argument is a pointer to space for two integers, where it will record the two new file descriptors. Then it allocates the pipe and installs the file descriptors.

8.15 Real world

The buffer cache in a real-world operating system is significantly more complex than xv6's, but it serves the same two purposes: caching and synchronizing access to the disk. Xv6's buffer cache, like V6's, uses a simple least recently used (LRU) eviction policy; there are many more complex policies that can be implemented, each good for some workloads and not as good for others. A more efficient LRU cache would eliminate the linked list, instead using a hash table for lookups and a heap for LRU evictions. Modern buffer caches are typically integrated with the virtual memory system to support memory-mapped files.

Xv6's logging system is inefficient. A commit cannot occur concurrently with file-system system calls. The system logs entire blocks, even if only a few bytes in a block are changed. It performs synchronous log writes, a block at a time, each of which is likely to require an entire disk rotation time. Real logging systems address all of these problems.

Logging is not the only way to provide crash recovery. Early file systems used a scavenger during reboot (for example, the UNIX `fsck` program) to examine every file and directory and the block and inode free lists, looking for and resolving inconsistencies. Scavenging can take hours for large file systems, and there are situations where it is not possible to resolve inconsistencies in a way that causes the original system calls to be atomic. Recovery from a log is much faster and causes system calls to be atomic in the face of crashes.

Xv6 uses the same basic on-disk layout of inodes and directories as early UNIX; this scheme has been remarkably persistent over the years. BSD's UFS/FFS and Linux's ext2/ext3 use essentially the same data structures. The most inefficient part of the file system layout is the directory, which requires a linear scan over all the disk blocks during each lookup. This is reasonable when directories are only a few disk blocks, but is expensive for directories holding many files. Microsoft Windows's NTFS, macOS's HFS, and Solaris's ZFS, just to name a few, implement a directory as an on-disk balanced tree of blocks. This is complicated but guarantees logarithmic-time directory lookups.

Xv6 is naive about disk failures: if a disk operation fails, xv6 panics. Whether this is reasonable depends on the hardware: if an operating systems sits atop special hardware that uses redundancy to mask disk failures, perhaps the operating system sees failures so infrequently that panicking is okay. On the other hand, operating systems using plain disks should expect failures and handle them more gracefully, so that the loss of a block in one file doesn't affect the use of the rest of the

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用于读取，而不是写入。假设无论如何都获得了 inode，sys_open 分配一个文件和一个文件描述符 (kernel/sysfile.c:344)，然后填充文件 (kernel/sysfile.c:356-361)。请注意，由于它仅在当前进程的表中，因此没有其他进程可以访问部分初始化的文件。

第7章在文件系统出现之前就考察了管道的实现。函数sys_pipe通过提供创建管道对的方式将这一实现与文件系统连接起来。它的参数是指向两个整数空间的指针，用于记录两个新的文件描述符。然后它分配管道并安装文件描述符。

8.15 现实世界

现实世界操作系统的缓冲区缓存比xv6的复杂得多，但它服务于同样的两个目的：缓存和同步对磁盘的访问。xv6的缓冲区缓存与V6的类似，使用简单的最近最少使用（LRU）驱逐策略；可以实现许多更复杂的策略，每种策略对某些工作负载有效，对另一些则不那么有效。一个更高效的LRU缓存将消除链表，改用哈希表进行查找，并使用堆进行LRU驱逐。现代缓冲区缓存通常与虚拟内存系统集成，以支持内存映射文件。

Xv6 的日志系统效率低下。提交操作不能与文件系统系统调用并发发生。系统记录整个块，即使块中只有几个字节被更改。它执行同步日志写入，一次一个块，每个块很可能需要整个磁盘旋转时间。真实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式。早期的文件系统在重启期间使用清理程序（例如 UNIX fsck 程序）来检查每个文件和目录以及块和 inode 空闲列表，查找并解决不一致性。对于大型文件系统，清理可能需要数小时，并且存在无法以导致原始系统调用原子性的方式解决不一致性的情况。从日志中恢复要快得多，并且在崩溃面前使系统调用原子性。

Xv6 使用与早期 UNIX 相同的基本磁盘 inode 和目录布局；这种方案多年来一直非常持久。BSD 的 UFS/FFS 和 Linux 的 ext2/ext3 使用本质上相同的数据结构。文件系统布局中最低效的部分是目录，它在每次查找时都需要对所有磁盘块进行线性扫描。当目录只有几个磁盘块时这是合理的，但对于包含许多文件的目录来说很昂贵。Microsoft Windows 的 NTFS、macOS 的 HFS 和 Solaris 的 ZFS，仅举几例，将目录实现为磁盘上的块平衡树。这很复杂，但保证了对数时间的目录查找。

Xv6 对磁盘故障很天真：如果磁盘操作失败，xv6 会恐慌。这是否合理取决于硬件：如果操作系统运行在特殊硬件上，该硬件使用冗余来掩盖磁盘故障，那么操作系统可能很少看到故障，因此恐慌是可接受的。另一方面，使用普通磁盘的操作系统应该预期到故障，并更优雅地处理它们，以便一个文件中丢失的块不会影响其余部分的使用

file system.

Xv6 requires that the file system fit on one disk device and not change in size. As large databases and multimedia files drive storage requirements ever higher, operating systems are developing ways to eliminate the “one disk per file system” bottleneck. The basic approach is to combine many disks into a single logical disk. Hardware solutions such as RAID are still the most popular, but the current trend is moving toward implementing as much of this logic in software as possible. These software implementations typically allow rich functionality like growing or shrinking the logical device by adding or removing disks on the fly. Of course, a storage layer that can grow or shrink on the fly requires a file system that can do the same: the fixed-size array of inode blocks used by xv6 would not work well in such environments. Separating disk management from the file system may be the cleanest design, but the complex interface between the two has led some systems, like Sun’s ZFS, to combine them.

Xv6’s file system lacks many other features of modern file systems; for example, it lacks support for snapshots and incremental backup.

Modern Unix systems allow many kinds of resources to be accessed with the same system calls as on-disk storage: named pipes, network connections, remotely-accessed network file systems, and monitoring and control interfaces such as `/proc`. Instead of xv6’s `if` statements in `fileread` and `filewrite`, these systems typically give each open file a table of function pointers, one per operation, and call the function pointer to invoke that inode’s implementation of the call. Network file systems and user-level file systems provide functions that turn those calls into network RPCs and wait for the response before returning.

8.16 Exercises

1. Why panic in `balloc` ? Can xv6 recover?
2. Why panic in `ialloc` ? Can xv6 recover?
3. Why doesn’t `filealloc` panic when it runs out of files? Why is this more common and therefore worth handling?
4. Suppose the file corresponding to `ip` gets unlinked by another process between `sys_link` ’s calls to `iunlock(ip)` and `dirlink`. Will the link be created correctly? Why or why not?
5. `create` makes four function calls (one to `ialloc` and three to `dirlink`) that it requires to succeed. If any doesn’t, `create` calls `panic`. Why is this acceptable? Why can’t any of those four calls fail?
6. `sys_chdir` calls `iunlock(ip)` before `iput(cp->cwd)`, which might try to lock `cp->cwd`, yet postponing `iunlock(ip)` until after the `iput` would not cause deadlocks. Why not?
7. Implement the `lseek` system call. Supporting `lseek` will also require that you modify `filewrite` to fill holes in the file with zero if `lseek` sets `off` beyond `f->ip->size`.

文件系统。

Xv6要求文件系统适合在一个磁盘设备上，并且大小不能改变。随着大型数据库和多媒体文件推动存储需求不断提高，操作系统正在开发消除“每个文件系统一个磁盘”瓶颈的方法。基本方法是将许多磁盘组合成一个逻辑磁盘。硬件解决方案如RAID仍然是最流行的，但当前趋势是尽可能将这部分逻辑实现为软件。这些软件实现通常允许丰富的功能，如通过动态添加或移除磁盘来扩展或缩小逻辑设备。当然，一个可以动态扩展或缩小的存储层需要一个能够同样做到这一点的文件系统：xv6使用的固定大小的索引节点块数组在这样的环境中表现不佳。将磁盘管理与文件系统分离可能是最干净的设计，但两者之间的复杂接口导致一些系统，如Sun的ZFS，将它们结合起来。

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

现代Unix系统允许许多种类的资源通过相同的系统调用进行访问，就像磁盘存储一样：命名管道、网络连接、远程访问的网络文件系统，以及像/proc这样的监控和控制接口。这些系统通常在fileread和filewrite中的if语句之外，为每个打开的文件提供一个函数指针表，每个操作一个，然后调用函数指针来调用该inode的调用实现。网络文件系统和用户级文件系统提供函数，将这些调用转换为网络RPC，并在返回之前等待响应。

8.16 练习

1. 为什么balloc会恐慌？xv6能恢复吗？
2. 为什么在ialloc中恐慌？xv6能恢复吗？
3. 为什么filealloc在用完文件时不会恐慌？为什么这种情况更常见，因此值得处理？
4. 假设在sys_link之间，另一个进程将ip对应的文件unlinked了
   对 iunlock(ip) 和 dirlink 的调用。链接会创建正确吗？为什么或为什么不？
5. create 需要四个函数调用（一个到 ialloc 和三个到 dirlink）才能成功。如果任何一个失败，create 会调用 panic。为什么这是可以接受的？为什么这四个调用中不能有任何失败？
6. sys_chdir 在调用 iput(cp->cwd) 之前调用 iunlock(ip)，这可能尝试锁定 cp->cwd，但推迟 iunlock(ip) 直到 iput 之后不会导致死锁。为什么不会？
7. 实现 lseek 系统调用。支持 lseek 还需要修改 filewrite，以便在 lseek 设置 off 超出 f->ip->size 时用零填充文件中的空洞。

8. Add `O_TRUNC` and `O_APPEND` to `open`, so that > and >> operators work in the shell.
9. Modify the file system to support symbolic links.
10. Modify the file system to support named pipes.
11. Modify the file and VM system to support memory-mapped files.

8. 在open中添加O_TRUNC和O_APPEND，以便 > 和 >> 运算符在shell中工作。
9. 修改文件系统以支持符号链接。
10. 修改文件系统以支持命名管道。
11. 修改文件和虚拟机系统以支持内存映射文件。

Chapter 9

Concurrency revisited

Simultaneously obtaining good parallel performance, correctness despite concurrency, and understandable code is a big challenge in kernel design. Straightforward use of locks is the best path to correctness, but is not always possible. This chapter highlights examples in which xv6 is forced to use locks in an involved way, and examples where xv6 uses lock-like techniques but not locks.

9.1 Locking patterns

Cached items are often a challenge to lock. For example, the file system's block cache (kernel/bio.c:26) stores copies of up to `NBUF` disk blocks. It's vital that a given disk block have at most one copy in the cache; otherwise, different processes might make conflicting changes to different copies of what ought to be the same block. Each cached block is stored in a `struct buf` (kernel/buf.h:1). A `struct buf` has a lock field which helps ensure that only one process uses a given disk block at a time. However, that lock is not enough: what if a block is not present in the cache at all, and two processes want to use it at the same time? There is no `struct buf` (since the block isn't yet cached), and thus there is nothing to lock. Xv6 deals with this situation by associating an additional lock (`bcache.lock`) with the set of identities of cached blocks. Code that needs to check if a block is cached (e.g., `bget` (kernel/bio.c:59)), or change the set of cached blocks, must hold `bcache.lock`; after that code has found the block and `struct buf` it needs, it can release `bcache.lock` and lock just the specific block. This is a common pattern: one lock for the set of items, plus one lock per item.

Ordinarily the same function that acquires a lock will release it. But a more precise way to view things is that a lock is acquired at the start of a sequence that must appear atomic, and released when that sequence ends. If the sequence starts and ends in different functions, or different threads, or on different CPUs, then the lock acquire and release must do the same. The function of the lock is to force other uses to wait, not to pin a piece of data to a particular agent. One example is the `acquire` in `yield` (kernel/proc.c:512), which is released in the scheduler thread rather than in the acquiring process. Another example is the `acquiresleep` in `ilock` (kernel/fs.c:293); this code often sleeps while reading the disk; it may wake up on a different CPU, which means the lock may be acquired and released on different CPUs.

第 9 章

并发回顾

同时获得良好的并行性能、并发下的正确性以及易于理解的代码是内核设计中的一个重大挑战。直接使用锁是获得正确性的最佳途径，但并不总是可行。本章重点介绍了 xv6 被迫以复杂方式使用锁的例子，以及 xv6 使用类似锁的技术但未使用锁的例子。

9.1 锁定模式

缓存的项通常对锁是一个挑战。例如，文件系统的块缓存（kernel/bio.c:26）存储了多达 NBUF 个磁盘块的副本。确保每个磁盘块在缓存中最多只有一个副本至关重要；否则，不同的进程可能会对不同副本进行冲突的修改，而这些副本本应是同一个块。每个缓存的块存储在一个 struct buf（kernel/buf.h:1）中。struct buf 有一个锁字段，这有助于确保在任何时候只有一个进程使用一个给定的磁盘块。然而，这还不够：如果块根本不在缓存中，而两个进程同时想使用它怎么办？没有 struct buf（因为块尚未缓存），因此没有东西可以锁。xv6 通过将一个额外的锁（bcache.lock）与缓存块的身份集关联来处理这种情况。需要检查块是否缓存（例如，bget（kernel/bio.c:59））或更改缓存块集的代码必须持有 bcache.lock；之后，该代码找到了它需要的块和 struct buf，它可以释放 bcache.lock 并仅锁定特定的块。这是一个常见的模式：一个锁用于项集，每个项一个锁。

通常，获取锁的同一个函数会释放它。但更精确地看待问题是，锁是在必须以原子方式出现的序列开始时获取的，并在该序列结束时释放。如果序列在不同的函数、不同的线程或不同的 CPU 中开始和结束，那么锁的获取和释放必须相同。锁的作用是强制其他使用者等待，而不是将数据固定到特定的代理。一个例子是 yield (kernel/proc.c:512) 中的获取，它在调度线程中释放，而不是在获取进程释放。另一个例子是 ilock (kernel/fs.c:293) 中的 acquiresleep；这段代码在读取磁盘时经常睡眠；它可能会在不同的 CPU 上唤醒，这意味着锁可能会在不同的 CPU 上获取和释放。

Freeing an object that is protected by a lock embedded in the object is a delicate business, since owning the lock is not enough to guarantee that freeing would be correct. The problem case arises when some other thread is waiting in `acquire` to use the object; freeing the object implicitly frees the embedded lock, which will cause the waiting thread to malfunction. One solution is to track how many references to the object exist, so that it is only freed when the last reference disappears. See `pipeclose` (kernel/pipe.c:59) for an example; `pi->readopen` and `pi->writeopen` track whether the pipe has file descriptors referring to it.

Usually one sees locks around sequences of reads and writes to sets of related items; the locks ensure that other threads see only completed sequences of updates (as long as they, too, lock). What about situations where the update is a simple write to a single shared variable? For example, `setkilled` and `killed` (kernel/proc.c:619) lock around their simple uses of `p->killed`. If there were no lock, one thread could write `p->killed` at the same time that another thread reads it. This is a race, and the C language specification says that a race yields *undefined behavior*, which means the program may crash or yield incorrect results[1]. The locks prevent the race and avoid the undefined behavior.

One reason races can break programs is that, if there are no locks or equivalent constructs, the compiler may generate machine code that reads and writes memory in ways quite different than the original C code. For example, the machine code of a thread calling `killed` could copy `p->killed` to a register and read only that cached value; this would mean that the thread might never see any writes to `p->killed`. The locks prevent such caching.

9.2 Lock-like patterns

In many places xv6 uses a reference count or a flag in a lock-like way to indicate that an object is allocated and should not be freed or re-used. A process's `p->state` acts in this way, as do the reference counts in `file`, `inode`, and `buf` structures. While in each case a lock protects the flag or reference count, it is the latter that prevents the object from being prematurely freed.

The file system uses `struct inode` reference counts as a kind of shared lock that can be held by multiple processes, in order to avoid deadlocks that would occur if the code used ordinary locks. For example, the loop in `namex` (kernel/fs.c:652) locks the directory named by each pathname component in turn. However, `namex` must release each lock at the end of the loop, since if it held multiple locks it could deadlock with itself if the pathname included a dot (e.g., `a/./b`). It might also deadlock with a concurrent lookup involving the directory and `..`. As Chapter 8 explains, the solution is for the loop to carry the directory inode over to the next iteration with its reference count incremented, but not locked.

Some data items are protected by different mechanisms at different times, and may at times be protected from concurrent access implicitly by the structure of the xv6 code rather than by explicit locks. For example, when a physical page is free, it is protected by `kmem.lock` (kernel/kalloc.c:24). If the page is then allocated as a pipe (kernel/pipe.c:23), it is protected by a different lock (the embedded `pi->lock`). If the page is re-allocated for a new process's user memory, it is not protected

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

释放受对象中嵌入的锁保护的对象是一项微妙的工作，因为拥有锁并不足以保证释放是正确的。问题情况发生在某个其他线程在 acquire 中等待使用对象时；释放对象会隐式地释放嵌入的锁，这会导致等待线程出现故障。一个解决方案是跟踪对象存在的引用数量，以便只有在最后一个引用消失时才释放它。参见 pipeclose (kernel/pipe.c:59) 的示例；pi->readopen 和 pi->writeopentrack 是否有文件描述符引用它。

通常，人们会在对相关项集合的读取和写入序列周围看到锁；锁确保其他线程只能看到完成的更新序列（只要它们也加锁）。那么，在更新是单个共享变量的简单写入的情况下呢？例如，setkilled 和 killed (kernel/proc.c:619) 在它们对 p->killed 的简单使用周围加锁。如果没有锁，一个线程可以在另一个线程读取它时写入 p->killed。这是一个竞争条件，C 语言规范说竞争条件会产生未定义行为，这意味着程序可能会崩溃或产生错误结果[1]。锁防止了竞争条件并避免了未定义行为。

竞争条件会破坏程序的一个原因是，如果没有锁或等效结构，编译器可能会生成与原始 C 代码截然不同的机器码来读取和写入内存。例如，调用 killed 的线程的机器码可能会将 p->killed 复制到寄存器，并且只读取该缓存的值；这意味着该线程可能永远不会看到对 top->killed 的任何写入。锁防止了这种缓存。

9.2 类似锁模式

在许多地方，xv6 使用引用计数或标志以类似锁的方式指示对象已被分配且不应被释放或重用。进程的 p->状态以此方式运作，文件、inode 和 buf 结构中的引用计数也是如此。虽然在每个情况下锁都保护标志或引用计数，但正是后者阻止对象被过早释放。

文件系统使用 struct inode 引用计数作为一种可以被多个进程持有的共享锁，以避免如果代码使用普通锁会发生的死锁。例如，namex（kernel/fs.c:652）中的循环依次锁定路径名组件命名的目录。然而，namex 必须在循环结束时释放每个锁，因为如果它持有多个锁，如果路径名包含点（例如，a/./b），它可能会与自身死锁。它还可能与涉及目录和...的并发查找死锁。正如第 8 章解释的那样，解决方案是让循环将目录 inode 带着增加的引用计数传递到下一次迭代，但不要锁定。

一些数据项在不同时间由不同的机制保护，有时可能由 xv6 代码的结构隐式地保护以防止并发访问，而不是通过显式锁。例如，当物理页是空闲时，它由 kmem.lock (kernel/kalloc.c:24) 保护。如果该页随后被分配为管道 (kernel/pipe.c:23)，它由不同的锁（嵌入的 pi->锁）保护。如果该页被重新分配为新进程的用户内存，则它没有任何锁保护

[1]“线程和数据竞争条件” in https://en.cppreference.com/w/c/language/memory_模型

by a lock at all. Instead, the fact that the allocator won't give that page to any other process (until it is freed) protects it from concurrent access. The ownership of a new process's memory is complex: first the parent allocates and manipulates it in `fork`, then the child uses it, and (after the child exits) the parent again owns the memory and passes it to `kfree`. There are two lessons here: a data object may be protected from concurrency in different ways at different points in its lifetime, and the protection may take the form of implicit structure rather than explicit locks.

A final lock-like example is the need to disable interrupts around calls to `mycpu()` (kernel/proc.c:83). Disabling interrupts causes the calling code to be atomic with respect to timer interrupts that could force a context switch, and thus move the process to a different CPU.

9.3 No locks at all

There are a few places where xv6 shares mutable data with no locks at all. One is in the implementation of spinlocks, although one could view the RISC-V atomic instructions as relying on locks implemented in hardware. Another is the `started` variable in `main.c` (kernel/main.c:7), used to prevent other CPUs from running until CPU zero has finished initializing xv6; the `volatile` ensures that the compiler actually generates load and store instructions.

Xv6 contains cases in which one CPU or thread writes some data, and another CPU or thread reads the data, but there is no specific lock dedicated to protecting that data. For example, in `fork`, the parent writes the child's user memory pages, and the child (a different thread, perhaps on a different CPU) reads those pages; no lock explicitly protects those pages. This is not strictly a locking problem, since the child doesn't start executing until after the parent has finished writing. It is a potential memory ordering problem (see Chapter 6), since without a memory barrier there's no reason to expect one CPU to see another CPU's writes. However, since the parent releases locks, and the child acquires locks as it starts up, the memory barriers in `acquire` and `release` ensure that the child's CPU sees the parent's writes.

9.4 Parallelism

Locking is primarily about suppressing parallelism in the interests of correctness. Because performance is also important, kernel designers often have to think about how to use locks in a way that both achieves correctness and allows parallelism. While xv6 is not systematically designed for high performance, it's still worth considering which xv6 operations can execute in parallel, and which might conflict on locks.

Pipes in xv6 are an example of fairly good parallelism. Each pipe has its own lock, so that different processes can read and write different pipes in parallel on different CPUs. For a given pipe, however, the writer and reader must wait for each other to release the lock; they can't read/write the same pipe at the same time. It is also the case that a read from an empty pipe (or a write to a full pipe) must block, but this is not due to the locking scheme.

Context switching is a more complex example. Two kernel threads, each executing on its own CPU, can call `yield`, `sched`, and `swtch` at the same time, and the calls will execute in parallel.

。相反，分配器不会将该页分配给任何其他进程（直到它被释放）这一事实保护了它免受并发访问。新进程内存的所有权很复杂：首先父进程在 fork 中分配和操作它，然后子进程使用它，并且在（子进程退出后）父进程再次拥有内存并将其传递给 kfree。这里有两条教训：数据对象在其生命周期中的不同时间点可能以不同的方式防止并发，并且保护可能以隐式结构的形式而不是显式锁的形式出现。

一个最终的类似锁的例子是需要在使用 mycpu() (kernel/proc.c:83) 时不启用中断。禁用中断会导致调用代码相对于定时器中断是原子的，这些中断可能会强制上下文切换，从而导致进程移动到不同的 CPU。

9.3 完全无锁

xv6 有几个地方在没有锁的情况下共享可变数据。一个是自旋锁的实现，尽管可以认为 RISC-V 原子指令依赖于硬件实现的锁。另一个是 main.c (kernel/main.c:7) 中的 started 变量，用于防止其他 CPU 在 CPU 零完成初始化 xv6 之前运行；volatile 确保编译器实际生成加载和存储指令。

xv6 包含一些情况，其中一个 CPU 或线程写入某些数据，而另一个 CPU 或线程读取这些数据，但没有特定的锁用于保护这些数据。例如，在 fork 中，父进程写入子进程的用户内存页，而子进程（可能是不同的线程，可能在不同的 CPU 上）读取这些页；没有锁明确保护这些页。这不是严格意义上的锁问题，因为子进程不会开始执行，直到父进程完成写入。这是一个潜在的内存顺序问题（见第 6 章），因为没有内存屏障，没有理由期望一个 CPU 能看到另一个 CPU 的写入。然而，由于父进程释放锁，而子进程在启动时获取锁，acquire 和 release 中的内存屏障确保子进程的 CPU 能看到父进程的写入。

9.4 并行性

锁定主要是在正确性的前提下抑制并行性。由于性能也很重要，内核设计者经常必须考虑如何使用锁，以同时实现正确性和允许并行性。虽然 xv6 没有系统地设计用于高性能，但它仍然值得考虑哪些 xv6 操作可以并行执行，哪些可能会在锁上发生冲突。

xv6 中的管道是相当好的并行性的一个例子。每个管道都有自己的锁，以便不同的进程可以在不同的 CPU 上并行地读取和写入不同的管道。然而，对于给定的管道，写入者和读者必须等待对方释放锁；他们不能同时读取/写入同一个管道。从空管道读取（或向满管道写入）也必须阻塞，但这并不是由于锁定方案。

上下文切换是一个更复杂的例子。两个内核线程，每个都在自己的CPU上执行，可以同时调用yield、sched和swtch，调用将执行并行。

The threads each hold a lock, but they are different locks, so they don't have to wait for each other. Once in `scheduler`, however, the two CPUs may conflict on locks while searching the table of processes for one that is `RUNNABLE`. That is, xv6 is likely to get a performance benefit from multiple CPUs during context switch, but perhaps not as much as it could.

Another example is concurrent calls to `fork` from different processes on different CPUs. The calls may have to wait for each other for `pid_lock` and `kmem.lock`, and for per-process locks needed to search the process table for an `UNUSED` process. On the other hand, the two forking processes can copy user memory pages and format page-table pages fully in parallel.

The locking scheme in each of the above examples sacrifices parallel performance in certain cases. In each case it's possible to obtain more parallelism using a more elaborate design. Whether it's worthwhile depends on details: how often the relevant operations are invoked, how long the code spends with a contended lock held, how many CPUs might be running conflicting operations at the same time, whether other parts of the code are more restrictive bottlenecks. It can be difficult to guess whether a given locking scheme might cause performance problems, or whether a new design is significantly better, so measurement on realistic workloads is often required.

9.5 Exercises

1. Modify xv6's pipe implementation to allow a read and a write to the same pipe to proceed in parallel on different CPUs.
2. Modify xv6's `scheduler()` to reduce lock contention when different CPUs are looking for runnable processes at the same time.
3. Eliminate some of the serialization in xv6's `fork()`.

这些线程各自持有锁，但它们是不同的锁，所以它们不必等待彼此。然而，一旦进入调度器，两个CPU可能会在搜索进程表以找到一个RUNNABLE的进程时发生锁冲突。也就是说，xv6在上下文切换期间可能会从多个CPU中获得性能收益，但可能没有它本可以获得的那么多。

另一个例子是不同进程在不同CPU上并发调用fork。这些调用可能需要为pid_锁和kmem.lock互相等待，以及为搜索进程表以查找未使用进程所需的每个进程的锁。另一方面，两个创建进程可以并行复制用户内存页并完全格式化页表页。

上述每个例子中的锁方案在某些情况下牺牲了并行性能。在每个情况下，使用更复杂的设计可以获得更多的并行性。是否值得取决于细节：相关操作被调用的频率、代码持有争用锁的时间、可能同时运行冲突操作的CPU数量、代码的其他部分是否是更严格的瓶颈。很难猜测给定的锁方案是否会导致性能问题，或者新的设计是否有显著改善，因此通常需要在真实工作负载上进行测量。

9.5 练习

1. 修改xv6的管道实现以允许同一管道的读取和写入在不同CPU上并行进行。
2. 修改xv6的scheduler()以减少不同CPU同时查找可运行进程时的锁竞争。
3. 消除一些xv6的fork()中的序列化。

Chapter 10

Summary

This text introduced the main ideas in operating systems by studying one operating system, xv6, line by line. Some code lines embody the essence of the main ideas (e.g., context switching, user/kernel boundary, locks, etc.) and each line is important; other code lines provide an illustration of how to implement a particular operating system idea and could easily be done in different ways (e.g., a better algorithm for scheduling, better on-disk data structures to represent files, better logging to allow for concurrent transactions, etc.). All the ideas were illustrated in the context of one particular, very successful system call interface, the Unix interface, but those ideas carry over to the design of other operating systems.

第十章

摘要

这段文本通过逐行研究一个操作系统xv6，介绍了操作系统中的主要思想。一些代码行体现了主要思想的精髓（例如，上下文切换、用户/内核边界、锁等），并且每一行都很重要；其他代码行则展示了如何实现特定的操作系统思想，并且可以很容易地用不同的方式完成（例如，更好的调度算法、更好的磁盘数据结构来表示文件、更好的日志记录以允许并发事务等）。所有这些思想都是在Unix接口这个特定的、非常成功的系统调用接口的上下文中进行说明的，但这些思想也适用于其他操作系统的设计。

108

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

参考文献

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

110

Index

索引