xv6: a simple, Unix-like teaching operating system

Russ Cox Frans Kaashoek Robert Morris

August 31, 2020

## 前言和致谢

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

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

我们在麻省理工学院的操作系统课6.828和6.S081中使用过这篇手稿。我们感谢这些课程的教师、助教和学生，他们都对xv6做出了直接或间接的贡献。我们尤其要感谢Adam Belay、Austin Clements和Nickolai Zeldovich。最后，我们要感谢那些通过电子邮件向我们发送文本中的错误或改进建议的人。Abutalib Aghayev、Sebastian Boehm、Anton Burtsev、Raphael Carvalho、Tej Chajed、Rasit Eskicioglu、Color Fuzzy、Giuseppe、Tao Guo、Naoki Hayama、Robert Hilderman、Wolfgang Keller、Austin Liew、Pavan Maddamsetti。Jacek Masiulaniec、Michael McConville、m3hm00d、miguelgvieira、Mark Morrissey、Harry Pan、Askar Safin、Salman Shah、Adeodato Simó、Ruslan Savchenko、Pawel Szczurko、Warren Toomey、tyfkda、zerbib、Xi Wang、Zou Chang Wei。

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

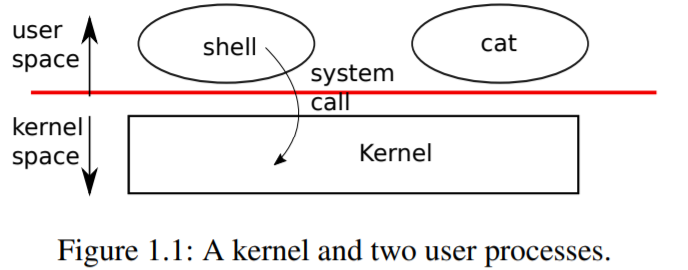
## CHAPTER 1: Operating system interfaces

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

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

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

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



​ 当一个进程需要调用一个内核服务时，它就会调用**系统调用**，这是操作系统接口中的一个调用。系统调用进入内核，内核执行服务并返回。因此，一个进程在用户空间和内核空间中交替执行。

​ 内核使用CPU[[1]](#footnote-1) 提供的硬件保护机制来确保在用户空间中执行的每个进程只能访问其自己的内存。内核运行时拥有硬件特权，可以访问这些受到保护的资源;用户程序执行时没有这些特权。当用户程序调用系统调用(接口)时，硬件提高特权级别并开始执行内核中预先安排的函数。

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

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

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

本章其余部分概述了xv6的服务进程、内存、文件描述符、管道和文件系统，并通过代码片段和讨论**shell**（Unix的命令行用户接口），以及使用它们。shell对系统调用的使用说明了系统调用是如何被精心设计的。

shell是一个普通的程序，它从用户那里读取命令并执行它们。shell是一个用户程序，而不是内核的一部分，这一事实说明了系统调用接口的强大功能:shell没有什么特别之处。这也意味着外壳很容易更换;因此，现代Unix系统有许多shell可供选择，每个shell都有自己的用户界面和脚本特性。xv6 shell是Unix Bourne shell的一个简单实现。它的实现可以在(user/sh.c:1)找到。

### 1.1 Processes and memory

一个xv6进程由用户空间内存（指令、数据和堆栈）和内核私有的进程状态组成。Xv6的进程共享cpu，它透明地切换当前cpu正在执行的进程。当一个进程暂时不使用cpu时，xv6会保存它的CPU寄存器，在下次运行该进程时恢复它们。内核为每个进程关联一个**PID**(进程标识符)。

可以使用**fork**系统调用创建一个新的进程。**Fork**创建的新进程，称为子进程，其内存内容与调用的进程完全相同，原进程被称为父进程。在父进程和子进程中，fork都会返回。在父进程中，fork返回子进程的PID；在子进程中，fork返回0。例如，考虑以下用C编程语言编写的程序片段[6]。

*int* pid = fork();

    if (pid > 0)

    {

        printf("parent: child=%d\n", pid);

        pid = wait((*int* \*)0);

        printf("child %d is done\n", pid);

    }

    else if (pid == 0)

    {

        printf("child: exiting\n");

        exit(0);

    }

    else

    {

        printf("fork error\n");

    }

exit系统调用使调用进程停止执行，并释放资源，如内存和打开的文件。exit需要一个整数状态参数，通常0表示成功，1表示失败。wait系统调用返回当前进程的一个已退出（或被杀死）的子进程的PID，并将该子进程的退出状态码复制到一个地址，该地址由wait参数提供

；如果调用者的子进程都没有退出，则wait等待一个子进程退出。如果调用者没有子进程，wait立即返回-1。如果父进程不关心子进程的退出状态，可以传递一个0地址给wait。

在上面的例子中，输出为:

parent: child=3884

child: exiting

child 3884 is done

可能会以任何一种顺序输出，这取决于是父进程还是子进程先进入它的printf调用。在子程序退出后，父程序的wait返回，父程序打印。

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

exec系统调用用从文件系统中存储的文件加载的新内存映像替换调用进程的内存。这个文件必须有特定的格式，它指定了文件中哪部分存放指令，哪部分是数据，在哪条指令开始等。xv6使用ELF格式，第3章将详细讨论。当exec成功时，它并不返回到调用程序；相反，从文件中加载的指令在ELF头声明的入口点开始执行。exec需要两个参数：包含可执行文件的文件名和一个字符串参数数组。例如：

*char* \*argv[3];

    argv[0] = "echo";

    argv[1] = "hello";

    argv[2] = 0;

    exec("/bin/echo", argv);

    printf("exec error\n");

这个片段将调用的程序替换为运行参数列表为echo hello的程序/bin/echo的实例。大多数程序都会忽略参数数组的第一个元素，也就是传统的程序名称。

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:145)。主循环用getcmd读取用户的一行输入，然后调用fork，创建shell的副本。然后调用fork，创建一个shell进程的副本。父循环调用wait，而子循环则运行命令。例如，如果用户向shell输入了echo hello，那么就会调用runcmd，参数为echo hello。runcmd (user/sh.c:58) 运行实际的命令。对于echo hello，它会调用exec (user/sh.c:78)。如果exec成功，那么子程序将执行来自echo的指令，而不是runcmd。在某些时候，echo会调用exit，这将导致父程序从main(user/sh.c:145)中的wait返回。

你可能会奇怪为什么fork和exec没有结合在一次调用中，我们后面会看到shell在实现I/O重定向时利用了这种分离。为了避免创建一个一样的进程并立即替换它（用exec）的浪费，操作内核通过使用虚拟内存技术（如copy-on-write）来优化这种用例的fork实现（见4.6节）。

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

### 1.2 I/O and File descriptors

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

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

读/写系统调用可以从文件描述符指向的文件读写数据。调用read(fd, buf, n)从文件描述符fd中读取不超过n个字节的数据，将它们复制到buf中，并返回读取的字节数。每个引用文件的文件描述符都有一个与之相关的偏移量。读取从当前文件偏移量中读取数据，然后按读取的字节数推进偏移量：随后的读取将返回第一次读取所返回的字节之后的字节。当没有更多的字节可读时，读返回零，表示文件的结束。

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

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

*char* buf[512];

*int* n;

    for (;;)

    {

        n = read(0, buf, sizeof buf);

        if (n == 0)

            break;

        if (n < 0)

        {

            fprintf(2, "read error\n");

            exit(1);

        }

        if (write(1, buf, n) != n)

        {

            fprintf(2, "write error\n");

            exit(1);

        }

    }

在这个代码片段中，需要注意的是，cat不知道它是从文件、控制台还是管道中读取的。同样，cat也不知道它是在打印到控制台、文件还是其他什么地方。使用文件描述符和文件描述符0是输入，文件描述符1是输出的约定，使得cat可以简单的实现。

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

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

*char* \*argv[2];

    argv[0] = "cat";

    argv[1] = 0;

    if (fork() == 0)

    {

        close(0); // 释放标准输入的文件描述符

        open("input.txt", O\_RDONLY); // 这时input.txt的文件描述符为0

// 即标准输入为input.txt

        exec("cat", argv); // cat从0读取，并输出到1，见上个代码段

    }

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

xv6 shell中的I/O重定向代码正是以这种方式工作的（user/sh.c:82）。回想一下，在代码的这一点上，shell已经fork子shell，runcmd将调用exec来加载新的程序。

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

现在应该清楚为什么fork和exec是分开调用的：在这两个调用之间，shell有机会重定向子shell的I/O，而不干扰主shell的I/O设置。我们可以假设一个由fork和exec组成的系统调用forkexec，但是用这种调用来做I/O重定向似乎很笨拙。shell在调用forkexec之前修改自己的I/O设置（然后取消这些修改），或者forkexec可以将I/O重定向的指令作为参数，或者（最糟糕的方案）每个程序（比如cat）都需要自己做I/O重定向[[2]](#footnote-2)。

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

    if (fork() == 0)

    {

        write(1, "hello ", 6);

        exit(0);

    }

    else

    {

        wait(0);

        write(1, "world\n", 6);

    }

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

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

    fd = dup(1);

    write(1, "hello ", 6);

    write(fd, "world\n", 6);

如果两个文件描述符是通过一系列的fork和dup调用从同一个原始文件描述符衍生出来的，那么这两个文件描述符共享一个偏移量。否则，文件描述符不共享偏移量，即使它们是由同一个文件的打开调用产生的。Dup允许shell实现这样的命令：ls existing-file non-existing-file > tmp1 2>&1。2>&1表示2是1的复制，即重定向错误信息到标准输出，现有文件的名称和不存在文件的错误信息都会显示在文件tmp1中。xv6 shell不支持错误文件描述符的I/O重定向，但现在你知道如何实现它了。

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

### 1.3 Pipes

管道是一个小的内核缓冲区，作为一对文件描述符暴露给进程，一个用于读，一个用于写。将数据写入管道的一端就可以从管道的另一端读取数据。管道为进程提供了一种通信方式。

下面的示例代码运行程序wc，标准输入连接到管道的读取端。

    int p[2];

    char \*argv[2];

    argv[0] = "wc";

    argv[1] = 0;

    pipe(p);

    if (fork() == 0)

    {

        close(0); // 释放文件描述符0

        dup(p[0]); // 复制一个p[0](管道读端)，此时文件描述符0（标准输入）也引用管道读端，故改变了标准输入。

        close(p[0]);

        close(p[1]);

        exec("/bin/wc", argv); // wc 从标准输入读取数据，并写入到参数中的每// 一个文件

    }

    else

    {

        close(p[0]);

        write(p[1], "hello world\n", 12);

        close(p[1]);

    }

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

如果没有数据可用，管道上的读会等待数据被写入，或者等待所有指向写端的文件描述符被关闭；在后一种情况下，读将返回0，就像数据文件的结束一样。事实上，如果没有数据写入，读会无限阻塞到新数据不可能到达为止（写端被关闭），这也是子进程在执行上面的wc之前关闭管道的写端很重要的一个原因：如果wc的一个文件描述符任然引用了管道的写端，那么wc将永远看不到文件的结束（被自己阻塞）。

xv6的shell实现了管道，如grep fork sh.c | wc -l，shell的实现类似于上面的代码（user/sh.c:100）。执行shell的子进程创建一个管道来连接管道的左端和右端（去看源码，不看难懂）。然后，它为管道左端（写入端）调用fork和runcmd，为右端（读取端）调用fork和runcmd，并等待两者的完成[[3]](#footnote-3)。管道的右端（读取端）可以是一个命令，也可以是包含管道的多个命令（例如，a | b | c），它又会分叉为两个新的子进程（一个是b，一个是c）。因此，shell可以创建一棵进程树。这棵树的叶子是命令，内部（非叶子）节点是等待左右子进程完成的进程。

原则上，我们可以让内部节点（非叶节点）运行管道的左端，但这样的实现会更加复杂。考虑只做以下修改：修改sh.c，使其不为runcmd(p->left) fork进程，直接递归运行runcmd(p->left)。像这样，echo hi | wc不会产生输出，因为当echo hi在runcmd中退出时，内部进程会退出，而不会调用fork来运行管道的右端。这种不正确的行为可以通过不在runcmd中为内部进程调用exit来修正，但是这种修正使代码变得复杂：runcmd需要知道该进程是否是内部进程（非叶节点）。当不为runcmd(p->right) fork进程，也会出现复杂的情况。像这样这样的修改，sleep 10 | echo hi就会立即打印出hi，而不是10秒后，因为echo会立即运行并退出，而不是等待sleep结束。由于sh.c的目标是尽可能的简单，所以它并没有试图避免创建内部进程[[4]](#footnote-4)。

管道似乎没有比临时文件更多的功能：

    echo hello world | wc

不用管道：

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

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

### 1.4 File system

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

    chdir("/a");

    chdir("b");

    open("c", O\_RDONLY);

    open("/a/b/c", O\_RDONLY);

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

有一些系统调用来创建新的文件和目录：mkdir创建一个新的目录，用O\_CREATE标志打开创建一个新的数据文件，以及mknod创建一个新的设备文件。这个例子说明了这两个系统调用的使用。

    mkdir("/dir");

    fd = open("/dir/file", O\_CREATE | O\_WRONLY);

    close(fd);

    mknod("/console", 1, 1);

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

一个文件的名称与文件本身是不同的；底层文件（非磁盘上的文件）被称为inode[[5]](#footnote-5)，一个inode可以有多个名称，称为链接。每个链接由目录中的一个条目组成；该条目包含一个文件名和对inode的引用。inode保存着一个文件的元数据（非原数据），包括它的类型（文件或目录或设备），它的长度，文件内容在磁盘上的位置，以及文件的链接数量。

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

#define T\_DIR 1    // Directory

#define T\_FILE 2   // File

#define T\_DEVICE 3 // Device

    struct stat

    {

        int dev;     // File system’s disk device

        uint ino;    // Inode number

        short type;  // Type of file

        short nlink; // Number of links to file

        uint64 size; // Size of file in bytes

    };

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

    open("a", O\_CREATE | O\_WRONLY);

    link("a", "b");

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

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

    unlink("a");

这行代码会删除a，此时只有b会引用inode。

    fd = open("/tmp/xyz", O\_CREATE | O\_RDWR);

    unlink("/tmp/xyz");

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

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

### 1.5 Real world

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

标准化。Xv6 并不符合 POSIX 标准：它缺少许多系统调用（包括基本的系统调用，如 lseek），而且它提供的许多系统调用与标准不同。我们对xv6的主要目标是简单明了，同时提供一个简单的类似UNIX的系统调用接口。一些人已经添加了一些系统调用和一个简单的C库扩展了xv6，以便运行基本的Unix程序。然而，现代内核比xv6提供了更多的系统调用和更多种类的内核服务。例如，它们支持网络、窗口系统、用户级线程、许多设备的驱动程序等等。现代内核不断快速发展，并提供了许多超越POSIX的功能。

Unix用一套文件名和文件描述符接口统一了对多种类型资源（文件、目录和设备）的访问。这个思想可以扩展到更多种类的资源，一个很好的例子是Plan 9[13]，它把资源就是文件的概念应用到网络、图形等方面。然而，大多数Unix衍生的操作系统都没有遵循这一路线。

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

Xv6没有提供用户的概念，也没有提供保护一个用户与另一个用户的概念；用Unix的术语来说，所有的xv6进程都以root身份运行。

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

### 1.6 Exercises

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

## Chapter 2 Operating system organization

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

本章概述了如何组织操作系统来实现这三个要求。现实中有很多方法，但本文主要介绍以宏内核[[6]](#footnote-6)为中心的主流设计，很多Unix操作系统都采用这种设计。本章还介绍了xv6进程的概述，xv6进程是xv6中的隔离单元，以及xv6启动时第一个进程的创建。

Xv6运行在多核[[7]](#footnote-7)RISC-V微处理器上，它的许多低级功能（例如，它的进程实现）是RISC-V所特有的。RISC-V是一个64位的CPU，xv6是用 "LP64 "C语言编写的，这意味着C编程语言中的long(L)和指针(P)是64位的，但int是32位的。本书假定读者在某种架构上做过一点机器级的编程，并懂一些RISC-V特有的思想。RISC-V有用的参考资料是 "The RISC-V Reader,An Open Architecture Attlas"［12］。用户级ISA[2]和特权架构[1]是官方规范。

一台完整的计算机中的CPU周围都是支持硬件，其中大部分是I/O接口的形式。Xv6是为qemu的"-machine virt "选项模拟的支持硬件编写的。这包括RAM、包含启动代码的ROM、与用户键盘/屏幕的串行连接以及用于存储的磁盘。

一台完整的计算机中的CPU周围都是硬件，其中大部分是I/O接口的形式连接cpu。Xv6是为qemu的"-machine virt "选项模拟的支持硬件编写的。这包括RAM、包含启动代码的ROM、与用户键盘/屏幕的串行连接以及用于存储的磁盘。

### 2.1 Abstracting physical resources

遇到一个操作系统，人们可能会问的第一个问题是为什么要有它呢？答案是，我们可以把图1.2中的系统调用作为一个库来实现，应用程序与之连接。在这个想法中，每个应用程序可以根据自己的需要定制自己的库。应用程序可以直接与硬件资源进行交互，并以最适合应用程序的方式使用这些资源（例如，实现高或可预测的性能）。一些用于嵌入式设备或实时系统的操作系统就是以这种方式组织的。

这种系统库方式的缺点是，如果有多个应用程序在运行，这些应用程序必须表现良好。例如，每个应用程序必须定期放弃CPU，以便其他应用程序能够运行。如果所有的应用程序都相互信任并且没有bug，这样的合作分时方案可能是OK的。更典型的情况是，应用程序之间互不信任，并且有bug，所以人们通常希望比合作方案提供更强的隔离性。

为了实现强隔离，禁止应用程序直接访问敏感的硬件资源，而将资源抽象为服务是很有帮助的。例如，Unix应用程序只通过文件系统的打开、读取、写入和关闭系统调用与存储进行交互，而不是直接读写磁盘。这为应用程序提供了路径名的便利，而且它允许操作系统（作为接口的实现者）管理磁盘。即使不考虑隔离问题，那些有意交互的程序（或者只是希望互不干扰）很可能会发现文件系统是一个比直接使用磁盘更方便的抽象。

同样，Unix在进程之间透明地切换硬件CPU，必要时保存和恢复寄存器状态，这样应用程序就不必意识到时间共享。这种透明性允许操作系统共享CPU，即使一些应用程序处于无限循环中。

另一个例子是，Unix进程使用exec来建立它们的内存映像，而不是直接与物理内存交互。这使得操作系统可以决定将进程放在内存的什么位置；如果内存紧张，操作系统甚至可能将进程的部分数据存储在磁盘上。Exec还允许用户将可执行文件储存在文件系统中。

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

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

### 2.2 User mode, supervisor mode, and system calls

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

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

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

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

### 2.3 Kernel organization

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

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

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

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

图2.1说明了这种微内核设计。在图中，文件系统作为一个用户级进程运行。作为进程运行的OS服务称为服务器。为了让应用程序与文件服务器进行交互，内核提供了一种进程间通信机制，用于从一个用户模式进程向另一个进程发送消息。例如，如果一个像shell这样的应用程序想要读取或写入一个文件，它就会向文件服务器发送一个消息，并等待响应。

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

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

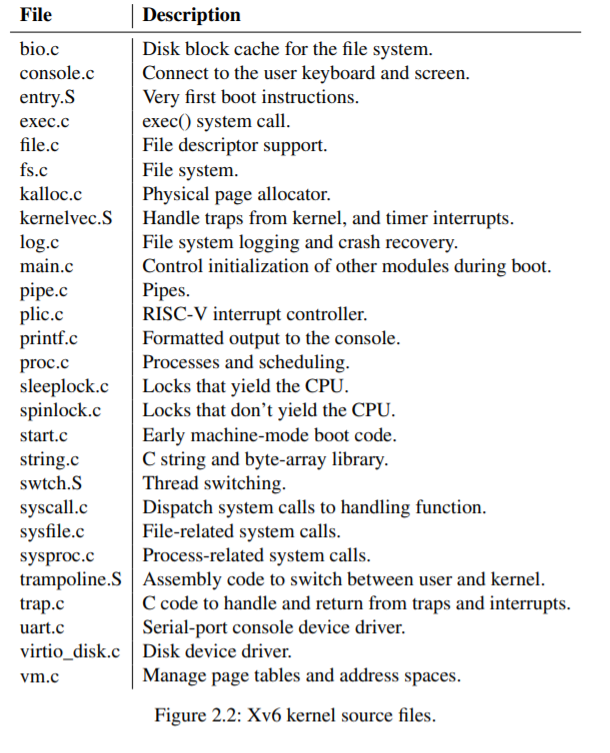
### 2.4 Code: xv6 organization

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

### 2.5 Process overview

xv6中的隔离单位（和其他Unix操作系统一样）是一个进程。进程抽象可以防止一个进程破坏或监视另一个进程的内存、CPU、文件描述符等。它还可以防止一个进程破坏内核，所以一个进程不能破坏内核的隔离机制。内核必须小心翼翼地实现进程抽象，因为一个错误的或恶意的应用程序可能会欺骗内核或硬件做一些不好的事情（例如，规避隔离）。内核用来实现进程的机制包括用户/监督模式标志、地址空间和线程的时间分割。

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



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

Xv6为每个进程维护一个单独的页表，定义该进程的地址空间。如图2.3所示，一个地址空间包括从虚拟地址0开始的进程的用户内存。指令排在第一位，其次是全局变量，然后是栈，最后是一个堆区（用于malloc），进程可以根据需要扩展。有一些因素限制了进程地址空间的最大大小：RISC-V上的指针是64位宽；硬件在页表中查找虚拟地址时只使用低的39位；xv6只使用39位中的38位。因此，最大地址是238-1 = 0x3fffffffff，也就是MAXVA（kernel/riscv.h:348）。在地址空间的顶端，xv6保留了一个页，用于trampoline和一个映射进程的trapframe的页，以便切换到内核，我们将在第4章中解释。

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

每个进程都有一个执行线程（简称线程），执行进程的指令。一个线程可以被暂停，然后再恢复。为了在进程之间透明地切换，内核会暂停当前运行的线程，并恢复另一个进程的线程。线程的大部分状态（局部变量、函数调用返回地址）都存储在线程的栈中。每个进程有两个栈：用户栈和内核栈（p->kstack）。当进程在执行用户指令时，只有它的用户栈在使用，而它的内核栈是空的。当进程进入内核时（为了系统调用或中断），内核代码在进程的内核栈上执行；当进程在内核中时，它的用户栈仍然包含保存的数据，但不被主动使用。一个进程的线程在主动使用它的用户栈和内核栈之间交替进行。内核栈是独立的（并且受到保护，不受用户代码的影响），所以即使一个进程用户栈被破坏了，内核也可以执行。

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

p->state表示进程是创建、就绪、运行、等待I/O，还是退出。

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

### 2.6 Code: starting xv6 and the first process

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

当RISC-V计算机开机时，它会初始化自己，并运行一个存储在只读存储器中的boot loader。Boot loader将xv6内核加载到内存中。然后，在机器模式下，CPU从 \_entry（kernel/entry.S:6）开始执行xv6。RISC-V在禁用分页硬件的情况下启动：虚拟地址直接映射到物理地址。

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

\_entry处的指令设置了一个栈，这样xv6就可以运行C代码。Xv6在文件start.c(kernel/start.c:11)中声明了初始栈的空间，即stack0。在\_entry处的代码加载堆栈指针寄存器sp，地址为stack0+4096，也就是堆栈的顶部，因为RISC-V的堆栈是向下生长的。现在内核有了栈，\_entry在start(kernel/start.c:21)处调用到C代码。

函数start执行一些只有在机器模式下才允许的配置，然后切换到监督者模式。为了进入监督者模式，RISC-V提供了指令mret。为了进入监督者模式，RISC-V提供了指令mret。这条指令最常用来从上一次的调用中返回，上一次调用从supervisor模式到机器模式[[8]](#footnote-8)。start并不是从这样的调用中返回，而是把事情设置得像有过这样的调用一样：它在寄存器mstatus中把上一次的特权模式设置为supervisor，它把main的地址写入寄存器mepc中，把返回地址设置为main，在supervisor模式中把0写入页表寄存器satp中，禁用虚拟地址转换，并把所有中断和异常委托给supervisor模式。

在跳入主管模式之前，start还要执行一项任务：对时钟芯片进行编程以产生定时器中断。在完成了这些内务管理后，start通过调用mret "返回 "到监督模式。这将导致程序计数器变为main（kernel/main.c:11）。

在main(kernel/main.c:11)初始化几个设备和子系统后，它通过调用userinit(kernel/proc.c:212)来创建第一个进程。第一个进程执行一个用RISC-V汇编编写的小程序initcode.S（user/initcode.S:1），它通过调用exec系统调用重新进入内核。正如我们在第一章中所看到的，exec用一个新的程序（本例中是/init）替换当前进程的内存和寄存器。一旦内核完成exec，它就会在/init进程中返回到用户空间。init (user/init.c:15)在需要时会创建一个新的控制台设备文件，然后以文件描述符0、1和2的形式打开它。然后它在控制台上启动一个shell。这样系统就启动了。

### 2.7 Real world

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

然而，现代操作系统支持一个进程中的多个线程，以允许一个进程利用多个CPU。在一个进程中支持多个线程涉及到不少xv6没有的机制，包括潜在的接口变化(如Linux的克隆，fork的变种)，以控制线程共享进程的哪些方面。

### 2.8 Exercises

1.你可以使用gdb来观察kernel mode到user mode的第一次转换。运行 make qemu-gdb。在同一目录下的另一个窗口中，运行gdb。输入gdb命令 break \*0x3ffffff10e，这将在内核中跳转到用户空间的sret指令处设置一个断点。输入continue gdb命令，gdb应该在断点处停止，并即将执行sret。gdb现在应该显示它正在地址0x0处执行，该地址在initcode.S的用户空间开始处。

1. 1本文一般用CPU中央处理单元的缩写来指代执行计算的硬件元素。其他文本(如RISC-V规范)也使用处理器、内核和hart等词代替CPU。 [↑](#footnote-ref-1)
2. 原文为 every program like cat could be taught to do its own I/O redirection. [↑](#footnote-ref-2)
3. 读取端会因为管道无数据且输入端未关闭而阻塞，即只能等待左边的命令执行完才能执行右边的命令，写入端需要将自己的输出写入到管道，或者关闭管道，右边的命令读取管道并将其作为自己的输入（可以没有参数）， [↑](#footnote-ref-3)
4. 多创建了进程，但其设计和实现更加优雅了。 [↑](#footnote-ref-4)
5. Inode是linux和类unix操作系统用来储存除了文件名和实际数据的数据结构，它是用来连接实际数据和文件名的。 [↑](#footnote-ref-5)
6. 与微内核设计理念相对应的理念，这也是一个源自操作系统级别的概念。对于宏内核来说，整个操作系统就是一个整体，包括了进程管理、内存管理、文件系统等等 [↑](#footnote-ref-6)
7. 本文所说的 "多核 "是指多个共享内存但并行执行的CPU，每个CPU都有自己的一套寄存器。本文有时使用多处理器一词作为多核的同义词，但多处理器也可以更具体地指具有多个不同处理器芯片的计算机。 [↑](#footnote-ref-7)
8. 大概意思是 mret通常用于返回到上一种模式，xv6通过把上一种模式设置为supervisor（实际上没有），而现在的模式为machine模式，使用mret指令就能改变模式为supervisor [↑](#footnote-ref-8)