第0章

操作系统接口

操作系统的工作是(1)将计算机的资源在多个程序间共享，并且给程序提供一系列比硬件本身更有用的服务。(2)管理并抽象底层硬件，举例来说，一个文字处理软件（比如 word）不用去关心自己使用的是何种硬盘。(3)多路复用硬件，使得多个程序可以(至少看起来是)同时运行的。(4)最后，给程序间提供一种受控的交互方式，使得程序之间可以共享数据、共同工作。

操作系统通过接口向用户程序提供服务。设计一个好的接口实际上是很难的。一方面我们希望接口设计得简单和精准，使其易于正确地实现；另一方面，我们可能忍不住想为应用提供一些更加复杂的功能。解决这种矛盾的办法是让接口的设计依赖于少量的*机制*（*mechanism*），而通过这些机制的组合提供强大、通用的功能。

本书通过 xv6 操作系统来阐述操作系统的概念，它提供 Unix 操作系统中的基本接口（由 Ken Thompson 和 Dennis Ritchie 引入），同时模仿 Unix 的内部设计。Unix 里机制结合良好的窄接口提供了令人吃惊的通用性。这样的接口设计非常成功，使得包括 BSD，Linux，Mac OS X，Solaris （甚至 Microsoft Windows 在某种程度上）都有类似 Unix 的接口。理解 xv6 是理解这些操作系统的一个良好起点。

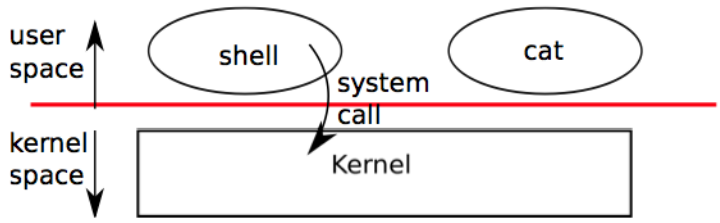

Figure 0-1. A kernel and two user processes.

如图0-1所示，xv6 使用了传统的**内核**概念 - 一个向其他运行中程序提供服务的特殊程序。每一个运行中程序（称之为**进程**）都拥有包含指令、数据、栈的内存空间。指令实现了程序的运算，数据是用于运算过程的变量，栈管理了程序的过程调用。

进程通过**系统调用**使用内核服务。系统调用会进入内核，让内核执行服务然后返回。所以进程总是在用户空间和内核空间之间交替运行。

内核使用了 CPU 的硬件保护机制来保证用户进程只能访问自己的内存空间。内核拥有实现保护机制所需的硬件权限(hardware privileges)，而用户程序没有这些权限。当一个用户程序进行一次系统调用时，硬件会提升特权级并且开始执行一些内核中预定义的功能。

内核提供的一系列系统调用就是用户程序可见的操作系统接口，xv6 内核提供了 Unix 传统系统调用的一部分，它们是：

系统调用	描述
fork()	创建进程
exit()	结束当前进程
wait()	等待子进程结束
kill(pid)	结束 pid 所指进程
getpid()	获得当前进程 pid
sleep(n)	睡眠 n 秒
exec(filename, *argv)	加载并执行一个文件
sbrk(n)	为进程内存空间增加 n 字节
open(filename, flags)	打开文件，flags 指定读/写模式
read(fd, buf, n)	从文件中读 n 个字节到 buf
write(fd, buf, n)	从 buf 中写 n 个字节到文件
close(fd)	关闭打开的 fd
dup(fd)	复制 fd
pipe(p)	创建管道， 并把读和写的 fd 返回到p
chdir(dirname)	改变当前目录
mkdir(dirname)	创建新的目录
mknod(name, major, minor)	创建设备文件
fstat(fd)	返回文件信息
link(f1, f2)	给 f1 创建一个新名字(f2)
unlink(filename)	删除文件

这一章剩下的部分将说明 xv6 系统服务的概貌 —— 进程，内存，文件描述符，管道和文件系统，为了描述他们，我们给出了代码和一些讨论。这些系统调用在 shell 上的应用阐述了他们的设计是多么独具匠心。

shell 是一个普通的程序，它接受用户输入的命令并且执行它们，它也是传统 Unix 系统中最基本的用户界面。shell 作为一个普通程序，而不是内核的一部分，充分说明了系统调用接口的强大：shell 并不是一个特别的用户程序。这也意味着 shell 是很容易被替代的，实际上这导致了现代 Unix 系统有着各种各样的 shell，每一个都有着自己的用户界面和脚本特性。xv6 shell 本质上是一个 Unix Bourne shell 的简单实现。它的实现在第 7850 行。

进程和内存

一个 xv6 进程由两部分组成，一部分是用户内存空间（指令，数据，栈），另一部分是仅对内核可见的进程状态。xv6 提供了分时特性：它在可用 CPU 之间不断切换，决定哪一个等待中的进程被执行。当一个进程不在执行时，xv6 保存它的 CPU 寄存器，当他们再次被执行时恢复这些寄存器的值。内核将每个进程和一个 **pid** (process identifier) 关联起来。

一个进程可以通过系统调用 `fork` 来创建一个新的进程。`fork` 创建的新进程被称为**子进程**，子进程的内存内容同创建它的进程（父进程）一样。`fork` 函数在父进程、子进程中都返回（一次调用两次返回）。对于父进程它返回子进程的 pid，对于子进程它返回 0。考虑下面这段代码：

```
int pid;
pid = fork();
if(pid > 0){
    printf("parent: child=%d\n", pid);
    pid = wait();
    printf("child %d is done\n", pid);
} else if(pid == 0){
    printf("child: exiting\n");
    exit();
} else {
    printf("fork error\n");
}
```

系统调用 `exit` 会导致调用它的进程停止运行，并且释放诸如内存和打开文件在内的资源。系统调用 `wait` 会返回一个当前进程已退出的子进程，如果没有子进程退出，`wait` 会等候直到有一个子进程退出。在上面的例子中，下面的两行输出

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

可能以任意顺序被打印，这种顺序由父进程或子进程谁先结束 `printf` 决定。当子进程退出时，父进程的 `wait` 也就返回了，于是父进程打印：

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

需要留意的是父子进程拥有不同的内存空间和寄存器，改变一个进程中的变量不会影响另一个进程。

系统调用 `exec` 将从某个*文件*（通常是可执行文件）里读取内存镜像，并将其替换到调用它的进程的内存空间。这份文件必须符合特定的格式，规定文件的哪一部分是指令，哪一部分是数据，哪里是指令的开始等等。xv6 使用 ELF 文件格式，第2章将详细介绍它。当 `exec` 执行成功后，它并不返回到原来的调用进程，而是从ELF头中声明的入口开始，执行从文件中加载的指令。`exec` 接受两个参数：可执行文件名和一个字符串参数数组。举例来说：

```
char *argv[3];
argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo", argv);
printf("exec error\n");
```

这段代码将调用程序替换为 `/bin/echo` 这个程序，这个程序的参数列表为 `echo hello`。大部分的程序都忽略第一个参数，这个参数惯例上是程序的名字（此例是 echo）。

xv6 shell 用以上调用为用户执行程序。shell 的主要结构很简单，详见 `main` 的代码（8001）。主循环通过 `getcmd` 读取命令行的输入，然后它调用 `fork` 生成一个 shell 进程的副本。父 shell 调用 `wait`，而子进程执行用户命令。举例来说，用户在命令行输入“echo hello”，`getcmd` 会以 `echo hello` 为参数调用 `runcmd`（7906），由 `runcmd` 执行实际的命令。对于 `echo hello`，`runcmd` 将调用 `exec`。如果 `exec` 成功被调用，子进程就会转而去执行 `echo` 程序里的指令。在某个时刻 `echo`

会调用 `exit`，这会使得其父进程从 `wait` 返回。你可能会疑惑为什么 `fork` 和 `exec` 为什么没有被合并成一个调用，我们之后将会发现，将创建进程——加载程序分为两个过程是一个非常机智的设计。

xv6 通常隐式地分配用户的内存空间。`fork` 在子进程需要装入父进程的内存拷贝时分配空间，`exec` 在需要装入可执行文件时分配空间。一个进程在需要额外内存时可以通过调用 `sbrk(n)` 来增加 n 字节的数据内存。`sbrk` 返回新的内存的地址。

xv6 没有用户这个概念当然更没有不同用户间的保护隔离措施。按照 Unix 的术语来说，所有的 xv6 进程都以 root 用户执行。

I/O 和文件描述符

文件描述符是一个整数，它代表了一个进程可以读写的被内核管理的对象。进程可以通过多种方式获得一个文件描述符，如打开文件、目录、设备，或者创建一个管道（pipe），或者复制已经存在的文件描述符。简单起见，我们常常把文件描述符指向的对象称为“文件”。文件描述符的接口是对文件、管道、设备等的抽象，这种抽象使得它们看上去就是字节流。

每个进程都有一张表，而 xv6 内核就以文件描述符作为这张表的索引，所以每个进程都有一个从0开始的文件描述符空间。按照惯例，进程从文件描述符0读入（标准输入），从文件描述符1输出（标准输出），从文件描述符2输出错误（标准错误输出）。我们会看到 shell 正是利用了这种惯例来实现 I/O 重定向。shell 保证在任何时候都有3个打开的文件描述符（8007），他们是控制台（console）的默认文件描述符。

系统调用 `read` 和 `write` 从文件描述符所指的文件中读或者写 n 个字节。`read(fd, buf, n)` 从 `fd` 读最多 n 个字节（`fd` 可能没有 n 个字节），将它们拷贝到 `buf` 中，然后返回读出的字节数。每一个指向文件的文件描述符都和一个偏移关联。`read` 从当前文件偏移处读取数据，然后把偏移增加读出字节数。紧随其后的 `read` 会从新的起点开始读数据。当没有数据可读时，`read` 就会返回0，这就表示文件结束了。

`write(fd, buf, n)` 写 `buf` 中的 n 个字节到 `fd` 并且返回实际写出的字节数。如果返回值小于 n 那么只可能是发生了错误。就像 `read` 一样，`write` 也从当前文件的偏移处开始写，在写的过程中增加这个偏移。

下面这段程序（实际上就是 `cat` 的本质实现）将数据从标准输入复制到标准输出，如果遇到了错误，它会在标准错误输出输出一条信息。

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

这段代码中值得一提的是 `cat` 并不知道它是从文件、控制台或者管道中读取数据的。同样地 `cat` 也不知道它是写到文件、控制台或者别的什么地方。文件描述符的使用和一些惯例（如0是标准输入，1是标准输出）使得我们可以轻松实现 `cat`。

系统调用 `close` 会释放一个文件描述符，使得它未来可以被 `open`，`pipe`，`dup` 等调用重用。一个新分配的文件描述符永远都是当前进程的最小的未被使用的文件描述符。

文件描述符和 `fork` 的交叉使用使得 I/O 重定向能够轻易实现。`fork` 会复制父进程的文件描述符和内存，所以子进程和父进程的文件描述符一模一样。`exec` 会替换调用它的进程的内存但是会保留它的文件描述符表。这种行为使得 shell 可以这样实现重定向：`fork` 一个进程，重新打开指定文件的文件描述符，然后执行新的程序。下面是一个简化版的 shell 执行 `cat<input.txt` 的代码:

```
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，我们可以保证 `open` 会使用0作为新打开的文件 `input.txt` 的文件描述符（因为0是 `open` 执行时的最小可用文件描述符）。之后 `cat` 就会在标准输入指向 `input.txt` 的情况下运行。

xv6 的 shell 正是这样实现 I/O 重定位的（7930）。在 shell 的代码中，记得这时 `fork` 出了子进程，在子进程中 `runcmd` 会调用 `exec` 加载新的程序。现在你应该很清楚为何 `fork` 和 `exec` 是单独的两种系统调用了吧。这种区分使得 shell 可以在子进程执行指定程序之前对子进程进行修改。

虽然 `fork` 复制了文件描述符，但每一个文件当前的偏移仍然是在父子进程之间共享的，考虑下面这个例子:

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

在这段代码的结尾，绑定在文件描述符1上的文件有数据"hello world"，父进程的 `write` 会从子进程 `write` 结束的地方继续写 (因为 `wait` ,父进程只在子进程结束之后才运行 `write`)。这种行为有利于顺序执行的 shell 命令的顺序输出，例如 `(echo hello; echo world)>output.txt`。

`dup` 复制一个已有的文件描述符，返回一个指向同一个输入/输出对象的新描述符。这两个描述符共享一个文件偏移，正如被 `fork` 复制的文件描述符一样。这里有另一种打印 “hello world” 的办法:

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

从同一个原初文件描述符通过一系列 `fork` 和 `dup` 调用产生的文件描述符都共享同一个文件偏移，而其他情况下产生的文件描述符就不是这样了，即使他们打开的都是同一份文件。`dup` 允许 shell 像这样实现命令：`ls existing-file non-exsiting-file > tmp1 2>&1`. `2>&1` 告诉 shell 给这条命令一个复制描述符1的描述符2。这样 `existing-file` 的名字和 `non-exsiting-file` 的错误输出都将出现在 `tmp1` 中。xv6 shell 并未实现标准错误输出的重定向，但现在你知道该怎么去实现它。

文件描述符是一个强大的抽象，因为他们将他们所连接的细节隐藏起来了：一个进程向描述符1写出，它有可能是写到一份文件，一个设备（如控制台），或一个管道。

管道

管道是一个小的内核缓冲区，它以文件描述符对的形式提供给进程，一个用于写操作，一个用于读操作。从管道的一端写的数据可以从管道的另一端读取。管道提供了一种进程间交互的方式。

接下来的示例代码运行了程序 `wc`，它的标准输出绑定到了一个管道的读端口。

```
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 {
    write(p[1], "hello world\n", 12);
    close(p[0]);
    close(p[1]);
}
```

这段程序调用 `pipe`，创建一个新的管道并且将读写描述符记录在数组 `p` 中。在 `fork` 之后，父进程和子进程都有了指向管道的文件描述符。子进程将管道的读端口拷贝在描述符0上，关闭 `p` 中的描述符，然后执行 `wc`。当 `wc` 从标准输入读取时，它实际上是从管道读取的。父进程向管道的写端口写入然后关闭它的两个文件描述符。

如果数据没有准备好，那么对管道执行的`read`会一直等待，直到有数据了或者其他绑定在这个管道写端口的描述符都已经关闭了。在后一种情况中，`read` 会返回 0，就像是一份文件读到了最后。读操作会一直阻塞直到不可能再有新数据到来了，这就是为什么我们在执行 `wc` 之前要关闭子进程的写端口。如果 `wc` 指向了一个管道的写端口，那么 `wc` 就永远看不到 eof 了。

xv6 shell 对管道的实现（比如 `fork sh.c | wc -l`）和上面的描述是类似的（7950行）。子进程创建一个管道连接管道的左右两端。然后它为管道左右两端都调用 `runcmd`，然后通过两次 `wait` 等待左右两端结束。管道右端可能也是一个带有管道的指令，如 `a | b | c`，它 `fork` 两个新的子进程（一个 `b` 一个 `c`），因此，shell 可能创建出一颗进程树。树的叶子节点是命令，中间节点是进程，它们会等待左子和右子执行结束。理论上，你可以让中间节点都运行在管道的左端，但做的如此精确会使得实现变得复杂。

pipe 可能看上去和临时文件没有什么两样：命令

`echo hello world | wc`

可以用无管道的方式实现：

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

但管道和临时文件起码有三个关键的不同点。首先，管道会进行自我清扫，如果是 shell 重定向的话，我们必须要在任务完成后删除 `/tmp/xyz`。第二，管道可以传输任意长度的数据。第三，管道允许同步：两个进程可以使用一对管道来进行二者之间的信息传递，每一个读操作都阻塞调用进程，直到另一个进程用 `write` 完成数据的发送。

文件系统

xv6 文件系统提供文件和目录，文件就是一个简单的字节数组，而目录包含指向文件和其他目录的引用。xv6 把目录实现为一种特殊的文件。目录是一棵树，它的根节点是一个特殊的目录 `root`。`/a/b/c` 指向一个在目录 `b` 中的文件 `c`，而 b 本身又是在目录 `a` 中的，`a` 又是处在 `root` 目录下的。不从 `/` 开始的目录表示的是相对调用进程当前目录的目录，调用进程的当前目录可以通过 `chdir` 这个系统调用进行改变。下面的这些代码都打开同一个文件（假设所有涉及到的目录都是存在的）。

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

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

第一个代码段将当前目录切换到 `/a/b`；第二个代码片段则对当前目录不做任何改变。

有很多的系统调用可以创建一个新的文件或者目录：`mkdir` 创建一个新的目录，`open` 加上 `O_CREATE` 标志打开一个新的文件，`mknod` 创建一个新的设备文件。下面这个例子说明了这3种调用：

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

`mknod` 在文件系统中创建一个文件，但是这个文件没有任何内容。相反，这个文件的元信息标志它是一个设备文件，并且记录主设备号和辅设备号（`mknod` 的两个参数），这两个设备号唯一确定一个内核设备。当一个进程之后打开这个文件的时候，内核将读、写的系统调用转发到内核设备的实现上，而不是传递给文件系统。

`fstat` 可以获取一个文件描述符指向的文件的信息。它填充一个名为 `stat` 的结构体，它在 `stat.h` 中定义为：

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

文件名和这个文件本身是有很大的区别。同一个文件（称为 `inode`）可能有多个名字，称为**连接**（`links`）。系统调用 `link` 创建另一个文件系统的名称，它指向同一个 `inode`。下面的代码创建了一个既叫做 `a` 又叫做 `b` 的新文件。

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

读写 `a` 就相当于读写 `b`。每一个 inode 都由一个唯一的 `inode 号` 直接确定。在上面这段代码中，我们可以通过 `fstat` 知道 `a` 和 `b` 都指向同样的内容：`a` 和 `b` 都会返回同样的 inode 号（`ino`），并且 `nlink` 数会设置为2。

系统调用 `unlink` 从文件系统移除一个文件名。一个文件的 inode 和磁盘空间只有当它的链接数变为 0 的时候才会被清空，也就是没有一个文件再指向它。因此在上面的代码最后加上

`unlink("a")`，

我们同样可以通过 `b` 访问到它。另外，

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

是创建一个临时 inode 的最佳方式，这个 inode 会在进程关闭 `fd` 或者退出的时候被清空。

xv6 关于文件系统的操作都被实现为用户程序，诸如 `mkdir`，`ln`，`rm` 等等。这种设计允许任何人都可以通过用户命令拓展 shell 。现在看起来这种设计是很显然的，但是 Unix 时代的其他系统的设计都将这样的命令内置在了 shell 中，而 shell 又是内置在内核中的。

有一个例外，那就是 `cd`，它是在 shell 中实现的（8016）。`cd` 必须改变 shell 自身的当前工作目录。如果 `cd` 作为一个普通命令执行，那么 shell 就会 `fork` 一个子进程，而子进程会运行 `cd`，`cd` 只会改变*子进程*的当前工作目录。父进程的工作目录保持原样。

现实情况

UNIX 将“标准”的文件描述符，管道，和便于操作它们的 shell 命令整合在一起，这是编写通用、可重用程序的重大进步。这个想法激发了 UNIX 强大和流行的“软件工具”文化，而且 shell 也是首个所谓的“脚本语言”。UNIX 的系统调用接口在今天仍然存在于许多操作系统中，诸如 BSD，Linux，以及 Mac OS X。

现代内核提供了比 xv6 要多得多的系统调用和内核服务。最重要的一点，现代基于 Unix 的操作系统并不遵循早期 Unix 将设备暴露为特殊文件的设计，比如刚才所说的控制台文件。Unix 的作者继续打造 Plan 9 项目，它将“资源是文件”的概念应用到现代设备上，将网络、图形和其他资源都视作文件或者文件树。

文件系统抽象是一个强大的想法，它被以万维网的形式广泛的应用在互联网资源上。即使如此，也存在着其他的操作系统接口的模型。Multics，一个 Unix 的前辈，将文件抽象为一种类似内存的概念，产生了十分不同的系统接口。Multics 的设计的复杂性对 Unix 的设计者们产生了直接的影响，他们因此想把文件系统的设计做的更简单。

这本书考察 xv6 是如何实现类似 Unix 的接口的，但涉及的想法和概念可以运用到 Unix 之外很多地方上。任何一个操作系统都需要让多个进程复用硬件，实现进程之间的相互隔离，并提供进程间通讯的机制。在学习 xv6 之后，你应该了解一些其他的更加复杂的操作系统，看一下他们当中蕴含的 xv6 的概念。