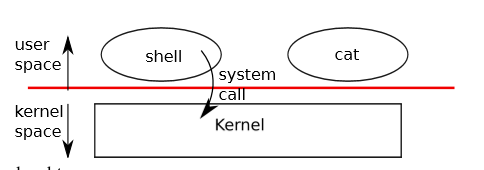
# chap0

## operating system interface

1. 接口应该具有的特性：简单；为应用提供更加sophisticated的特性

2. 大多数os采用 Unix-like接口

3.



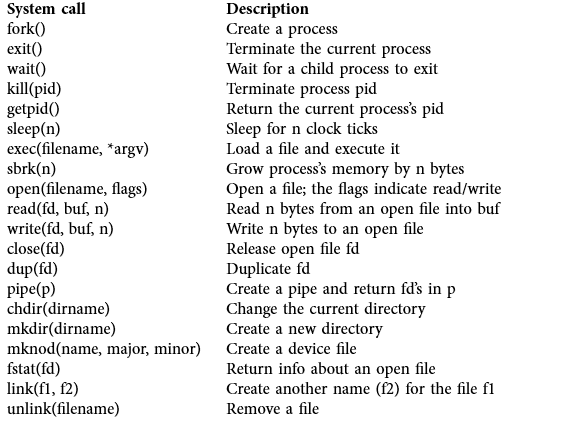
kernel：为进程process提供服务

进程：指令+数据+stack（组织procedure calls）

当进程请求一个系统服务时，使用系统调用system call完成从用户态到系统态的转变

shell：一个用户程序，可被替代

xv6 系统的系统调用：



## process and memory

关于fork（）：

fork仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：  
    1）在父进程中，fork返回新创建子进程的进程ID；  
    2）在子进程中，fork返回0；  
    3）如果出现错误，fork返回一个负值；

（其实就相当于链表，进程形成了链表，父进程的fpid(p 意味point)指向子进程的进程id, 因为子进程没有子进程，所以其fpid为0.）  
    fork出错可能有两种原因：  
    1）当前的进程数已经达到了系统规定的上限，这时errno的值被设置为EAGAIN。  
    2）系统内存不足，这时errno的值被设置为ENOMEM。  
    创建新进程成功后，系统中出现两个基本完全相同的进程，这两个进程执行没有固定的先后顺序，哪个进程先执行要看系统的进程调度策略。  
    每个进程都有一个独特（互不相同）的进程标识符（process ID），可以通过getpid（）函数获得，还有一个记录父进程pid的变量，可以通过getppid（）函数获得变量的值。

## I/O and File descriptors

file descriptors：small integer描述是否能读写

在xv6中，每一个进程都有一个私有的空间存放一个从0开始的file descriptor。

|  |  |
| --- | --- |
| 0 | standard input |
| 1 | standard output |
| 2 | (standard error |

## 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。

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。

## File system

* 树形结构
* link

文件的名称不同于文件本身;同一个底层文件(称为inode)可以有多个名称(称为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");

链接是一个起别名的过程，当一个文件被链接时，nlink数会增加，其中nlink是描述文件信息的结构体的一个成员，结构体定义如下：

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

};

用于文件系统操作的Shell命令被实现为用户级程序，如mkdir、ln、rm等。这种设计允许任何人通过添加一个新的用户级程序来使用新的用户命令来扩展shell。事后看来，这个计划似乎是显而易见的，但是在Unix时期设计的其他系统经常将这样的命令构建到shell中(并将shell构建到内核中)。

shell在死循环中调用getcmd()来获取用户输入的命令，读取到命令之后，fork一个子进程，并在子进程中运行runcmd()，runcmd()根据传入的参数用exec()函数在该子进程中加载命令对应的程序，待执行结束后子进程退出，期间父进程一直等待（父进程执行wait()），直到子进程结束。

一个例外是cd，它内置于shell(8716)中。cd必须更改shell本身的当前工作目录。如果cd作为常规命令运行，那么shell将派生一个子进程，该子进程将运行cd，而cd将更改子进程的工作目录。父程序(即shell的)的工作目录不会改变。

## real world

# chap1 Operating system organization

o's的一个关键需求是同时支持多个活动。例如，使用第0章中描述的系统调用接口，进程可以使用fork启动新进程。os必须在多个进程中时分复用计算机资源，