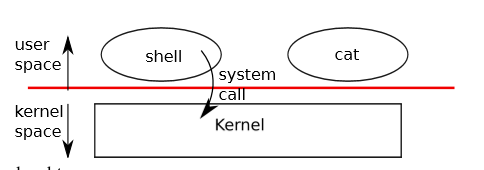
# chap0

## 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.



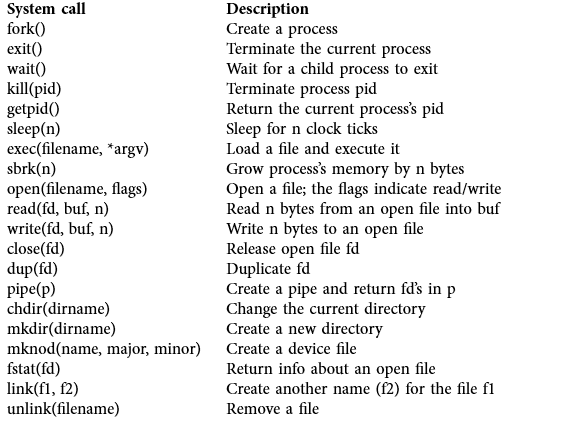
kernel：为进程process提供服务

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（）：

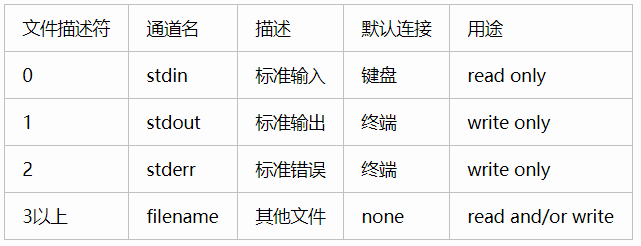
fork仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：  
    1）在父进程中，fork返回新创建子进程的进程ID；  
    2）在子进程中，fork返回0；  
    3）如果出现错误，fork返回一个负值；

（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。



文件描述符是内核为了高效管理已被打开的文件所创建的索引，用于指向被打开的文件，所有执行I/O操作的系统调用都通过文件描述符；文件描述符是一个简单的非负整数，用以表明每个被进程打开的文件。程序刚刚启动时，第一个打开的文件是0，第二个是1，以此类推。也可以理解为文件的身份ID。

## 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

## os要干什么?

* os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，即使有比硬件处理器更多的进程需要运行。
* os必须保证各个进程之间的isolation。即使一个进程有了bug或者运行失败，也不能影响不依赖于该失败进程的进程。但是完全的isolation太过强大，而进程之间可能有所交互，例如pipe。

因此一个os必须满足：复用（multiplexing）, isolation, 交互（interaction.）

## 本章概述

本章概述了操作系统是如何组织来实现这3个需求的。事实证明有很多方法可以实现这一点，但是本文主要关注以许多Unix操作系统都使用的单片内核（monolithic kernel）为中心的主流设计。

本章通过跟踪xv6开始运行时第一个进程的创建来介绍xv6的设计。在此过程中，本文简要介绍了xv6提供的所有主要抽象的实现、它们如何交互，以及如何满足多路复用、隔离和交互这三个需求。大多数xv6避免了对第一个进程进行特殊封装，而是重用了xv6必须为标准操作提供的代码。接下来的章节将更详细地探讨每个抽象。

Xv6运行在PC平台上的Intel 80386或更高版本(“x86”)处理器上，它的许多低级功能(例如，它的进程实现)都是特定于x86的。本书假设读者已经在某些体系结构上进行了一些机器级别的编程，并将在遇到特定于x86的思想时介绍它们。附录A简要概述了PC平台。

## Abstracting physical resources

当人们遇到操作系统，第一个问题可能会问为什么要这么做？也就是说可以将syscall实现为库，将应用与库连接起来。在这个计划中，每个应用程序甚至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交互，并以对应用程序最好的方式使用这些资源(例如，实现高或可预测的性能)。嵌入式设备或实时系统的一些操作系统就是这样组织的。

这个方法的缺点是：如果有多个应用程序在运行，则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处理器，以便其他应用程序能够运行。如果所有的应用程序相互信任并且没有bug，那么这种协作的分时方案可能是可行的。但对于应用程序更典型的情况是，应用程序之间互不信任，并且存在bug，因此通常需要比协作模式（cooperative scheme ）提供更强的isolation。

要实现强隔离，最好禁止应用程序直接访问敏感的硬件资源，而是将资源抽象到服务中。例如，应用程序仅通过open、read、write和close系统调用与文件系统交互，而不是读写原始磁盘扇区。这为应用程序提供了路径名的便利，并且允许操作系统(作为接口的实现者)管理磁盘。

类似地，Unix透明地在进程之间切换硬件处理器，必要时保存和恢复寄存器状态，因此应用程序不必知道他们之间相互时间共享。这种透明性允许操作系统共享处理器，即使某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，操作系统甚至可能将进程的一些数据存储在磁盘上。**exec**还为用户提供了方便的文件系统来存储可执行程序映像。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。文件描述符不仅抽象了许多细节(例如，管道或文件中的数据存储在哪里)，它们还以一种简化交互的方式定义。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**end-of-ﬁle** 。

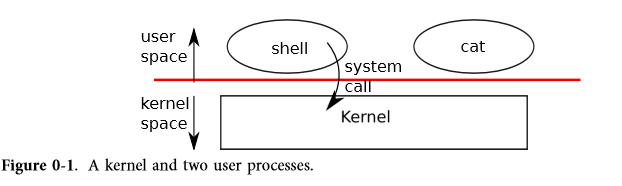
正如您所看到的，图0-2中的系统调用接口经过了精心设计，以便为程序员提供方便，并可能实现强大的隔离。Unix接口并不是抽象资源的唯一方法，但它已被证明是一种非常好的方法。

|  |  |
| --- | --- |
| systemcall | 描述 |
| fork（） | 创建一个进程 |
| exit（） | 终止一个进程 |
| wait（） | 等待一个子进程exit |
| kill（pid） | 终止进程pid |
| getpid（） | 获取当前进程的pid |
| sleep（n） | 睡眠n个时钟 |
| exec（filename，\*argv） | 加载一个文件并且执行它 |
| sbrk(n) | 将进程的内存增长n个字节 |
| open(filename,flags) | 打开一个文件 flags指示读/写 |
| read（fd，buf，n） | 从打开的文件中读取n bytes 到buf中 |
| write（fd,buf,n） | 向打开的文件中写入n bytes |
| close（fd） | 释放文件fd |
| dup（fd） | 复制文件fd |
| pipe（p） | 生成一个管道p并且在p中返回fd's |
| chdir(dirname) | 改变当前目录 |
| mkdir（dirname） | 创建一个新目录 |
| mknod（name,major,minor） | 创建一个磁盘文件 |
| fstat（fd） | 返回一个打开的文件的info |
| link（f1,f2） | 为文件f1创建一个新的名字f2（链接） |
| unlink（filename） | 删除一个文件 |

## User mode, kernel mode, and system calls

强隔离要求应用程序和操作系统之间有一个硬边界。如果应用程序出错，我们不希望操作系统失败或其他应用程序失败。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(甚至读)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。

处理器为强隔离提供硬件支持。例如，与许多其他处理器一样，x86处理器有两种模式来执行指令:内核模式和用户模式。在内核模式下，允许处理器执行特权指令。例如，读取和写入磁盘(或任何其他I/O设备)涉及特权指令。如果用户模式下的应用程序试图执行特权指令，那么处理器不会执行该指令，而是切换到内核模式，以便内核模式下的软件可以清理应用程序，因为它做了一些不应该做的事情。第0章中的图0-1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(例如，数字相加等)，并被称为运行在用户空间中，而内核模式的软件也可以执行特权指令，并被称为运行在内核空间中。在内核空间(或内核模式)中运行的软件称为内核。

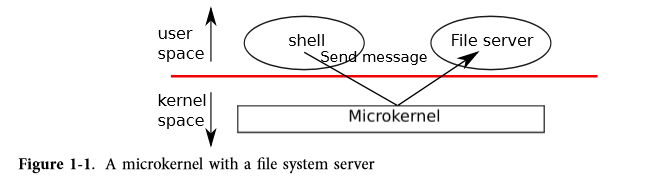


要在磁盘上读取或写入文件的应用程序必须转换到内核，因为应用程序本身不能执行I/O指令。处理器提供了一条特殊的指令，它将处理器从用户模式切换到内核模式，并在内核指定的入口点进入内核。(x86处理器为此提供了**int**指令。)一旦处理器切换到内核模式，内核就可以验证系统调用的参数，决定是否允许应用程序执行请求的操作，然后拒绝或执行它。内核设置转换到内核模式的入口点是很重要的;如果应用程序可以决定内核入口点，那么恶意应用程序可以在跳过参数验证等的点进入内核。

## Kernel organization

一个关键的设计问题是操作系统的哪一部分应该在内核模式下运行。一种可能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以内核模式运行。这种组织被称为单一内核。在这种组织中，整个操作系统在完全硬件特权下运行。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个部分不需要完整的硬件特权。此外，操作系统的不同部分很容易进行协作。例如，操作系统可能有一个可以由文件系统和虚拟内存系统共享的缓冲区缓存。

整体组织的一个缺点是操作系统不同部分之间的接口通常很复杂(我们将在本文的其余部分中看到)，因此操作系统开发人员很容易犯错误。在单片内核中，错误是致命的，因为内核模式中的错误通常会导致内核失败。如果内核失败，计算机会停止工作，所有应用程序也会失败。计算机必须重新启动才能重新开始。为了减少内核出错的风险，操作系统设计人员可以最小化在内核模式下运行的操作系统代码的数量，并在用户模式下执行操作系统的大部分内容。这种内核组织称为微内核。



如图为微内核的design。在图中，文件系统作为用户级进程运行。以进程形式运行的OS服务称为服务器。为了允许应用程序与文件服务器交互，内核提供了进程间通信机制，以便将消息从一个用户模式进程发送到另一个用户模式进程。例如，如果像shell之类的应用程序希望读取或写入文件，它将向文件服务器发送消息并等待响应。

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

遵循大多数Unix操作系统，Xv6是作为一个单片内核实现的。因此，在xv6中，内核接口对应于操作系统接口，内核实现完整的操作系统。由于xv6不提供很多服务，它的内核比一些微内核还要小。

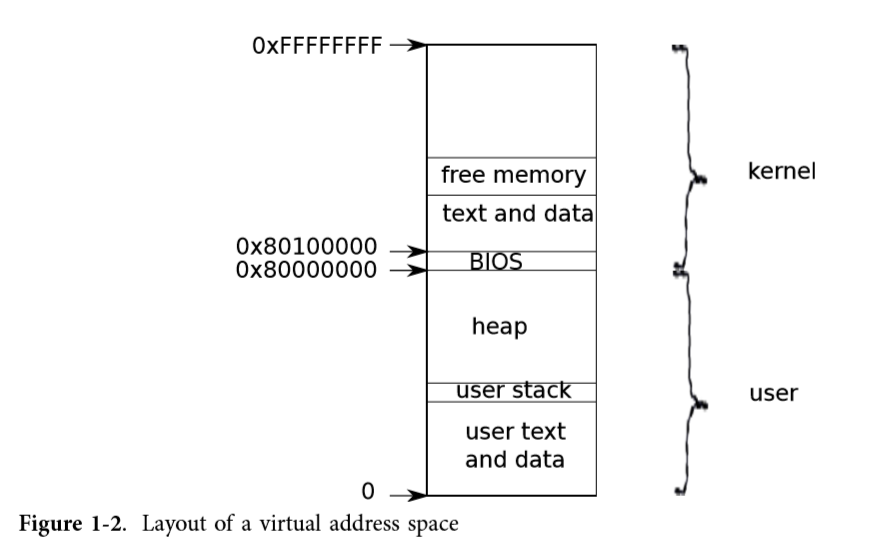
## Process overview

xv6(与其他Unix操作系统一样)中的隔离单元是一个进程。进程抽象防止一个进程破坏或监视另一个进程的内存、CPU、文件描述符等。它还可以防止进程破坏内核本身，这样进程就不能破坏内核的隔离机制。内核必须小心地实现进程抽象，因为一个有bug或恶意的应用程序可能欺骗内核或硬件做一些不好的事情(例如，绕过强制隔离)。内核用于实现进程的机制包括用户/内核模式标志（the user/kernel mode ﬂag）,地址空间（address spaces）, 线程的时分（ time-slicing ）。

为了帮助实现隔离，进程抽象为程序提供了一种具有自己私有机器的错觉。一个进程提供一个看起来像是私有内存系统或地址空间的程序，其他进程无法读或写它。进程还提供程序一个看起来属于它自己的执行程序指令的CPU。

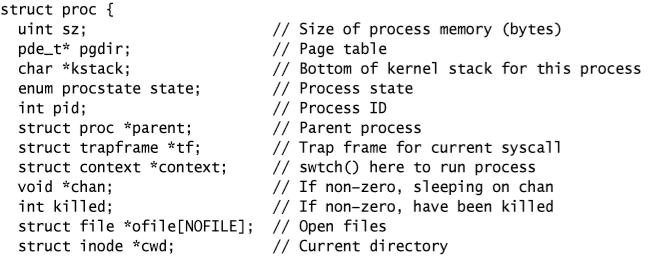
Xv6使用页表(由硬件实现)为每个进程提供自己的地址空间。x86页表转换(map)一个虚拟地址(x86指令操作的地址)到一个物理地址(处理器芯片发送到主存的地址)。

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1-2所示，地址空间包括从虚拟地址0开始的进程的用户内存。首先是指令，然后是全局变量，然后是堆栈，最后是进程可以根据需要展开的“堆”区域(用于malloc)。



每个进程的地址空间映射内核的指令和数据以及用户程序的内存。当进程调用系统调用时，系统调用在进程地址空间的内核映射中执行。这种安排使得内核的系统调用代码可以直接引用用户内存。为了给用户内存留出足够的空间，xv6的地址空间将内核映射为高地址，从0x80100000开始。

xv6内核为每个进程维护许多状态，并将它们收集到一个 **proc结构体**(2337)中。进程最重要的内核状态是它的页表、内核堆栈和运行状态。我们将使用符号p->xxx来引用**proc**结构的元素。





每个进程都有一个执行线程(简称线程)，执行进程的指令。线程可以被挂起，稍后恢复。为了在进程之间透明地切换，内核挂起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)存储在线程堆栈中。每个进程有两个堆栈:一个用户堆栈和一个内核堆栈(p->kstack)。当进程执行用户指令时，只使用它的用户堆栈，它的内核堆栈是空的。当进程进入内核(用于系统调用或中断)时，内核代码在进程的内核堆栈上执行;当进程在内核中时，它的用户堆栈仍然包含已保存的数据，但不被活跃使用。进程的线程在其用户堆栈和内核堆栈之间交替的活跃使用。内核堆栈是独立的(并与用户代码隔离)，因此即使进程破坏了它的用户堆栈，内核也可以执行。

当进程进行系统调用时，处理器切换到内核堆栈，提高硬件特权级别，并开始执行实现系统调用的内核指令。当系统调用完成时，内核返回到用户空间:硬件降低其特权级别，切换回用户堆栈，并在系统调用指令之后继续执行用户指令。进程的线程可以在内核中“阻塞”以等待I/O，并在I/O完成后从它停止的位置恢复。

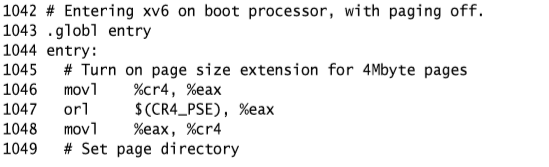
p->state表示进程是否已分配、准备运行、正在运行、等待I/O或退出。

p->pgdir以x86硬件期望的格式保存进程的页表。xv6中，分页硬件在执行进程时使用进程的p->pgdir。进程的页表还可以作为物理页地址的记录，这些物理页被用来存储进程的内存。

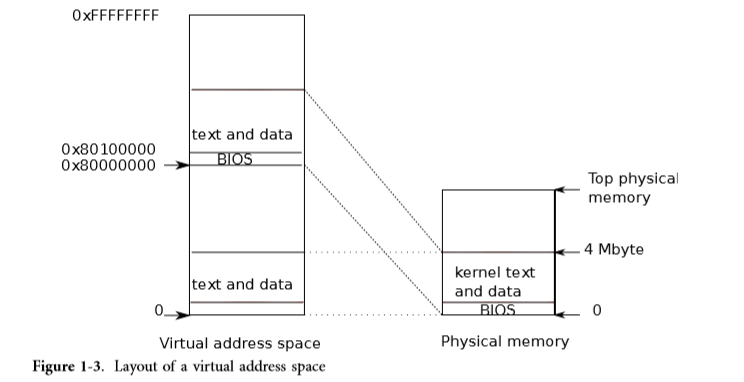
## Code: the ﬁrst address space

为了使xv6组织更加具体，我们将了解内核如何为自己创建第一个地址空间，内核如何创建和启动第一个进程，以及该进程如何执行第一个系统调用。通过跟踪这些操作，我们可以详细了解xv6如何为进程提供强大的隔离。提供强隔离的第一步是将内核设置为在自己的地址空间中运行。

当PC启动时，内核初始化自己，然后从磁盘加载一个引导加载程序到内存中并执行它。附录B解释了细节。Xv6的引导加载程序从磁盘加载Xv6内核，并从**entry**(1044)开始执行它。当内核启动时，不启用x86分页硬件;虚拟地址直接映射到物理地址。

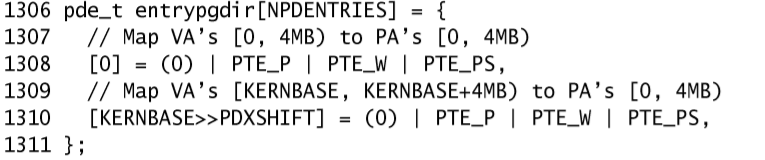


引导加载程序将xv6内核加载到物理地址0x100000的内存中。它没有在0x80100000处加载内核(内核希望在这里找到它的指令和数据)的原因是，在一台小机器上，在这么高的地址上可能没有任何物理内存。它将内核放在0x100000而不是0x0的原因是，地址范围0xa0000:0x100000包含I/O设备。



为了允许内核的其余部分运行，**entry**设置了一个页表，将虚拟地址从0x80000000开始(称为**KERNBASE**(0207))映射到从0x0开始的物理地址(参见图1-2)。将两个范围的虚拟地址映射到相同物理内存范围是页表的常见用法，我们将看到更多类似的示例。

entry page在main.c(1306)中定义。我们在第2章中会查看页表的详细信息，但是简单地说，entry 0将虚拟地址0:0x400000映射到物理地址0:0x400000。只要**entry**在低地址处执行，就需要此映射，但最终将被删除。



entry 512将虚拟地址KERNBASE（0x80000000）:KERNBASE+0x400000（4M）映射到物理地址0:0x400000。该entry将在 **entry** 完成后由内核使用;它将内核期望在其中找到指令和数据的高虚拟地址映射到引导加载程序加载它们的低物理地址。这个映射将内核指令和数据限制为 4 Mbytes 。

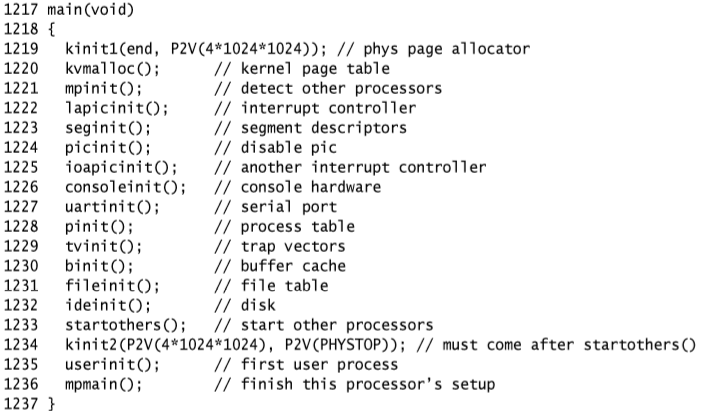
返回到**entry**，它将**entrypgdir**的物理地址加载到控制寄存器%cr3中。%cr3中的值必须是物理地址。让%cr3保存**entrypgdir**的虚拟地址没有意义，因为分页硬件还不知道如何转换虚拟地址;它还没有页表。符号**entrypgdir**引用高内存中的一个地址，宏V2P\_WO(0213)减去**KERNBASE**以找到物理地址（因为**KERNBASE**映射的物理地址为0）。为了启用分页硬件，xv6在控制寄存器%cr0中设置标志CR0\_PG。



启用分页后，处理器仍然在低地址处执行指令，这是因为**entrypgdir**映射低地址。如果xv6省略了了**entrypgdir**中的entry0，那么当尝试在启用分页的指令之后执行该指令时，计算机将会崩溃。

现在entry需要传输到内核的C代码，并在高内存中运行它。首先，它使堆栈指针%esp指向作为堆栈(1058)使用的内存。所有符号都有高地址，包括**stack**，所以即使低映射被删除，堆栈仍然有效。最后，**entry**转换至**main**，**main**也是高地址。之所以需要间接跳转，是因为汇编程序会生成一个相对于pc的直接跳转，它将执行**main**的低内存版本。因为堆栈上没有返回PC，所以**Main**不能返回。现在内核在**main**(1217)函数的高地址中运行。

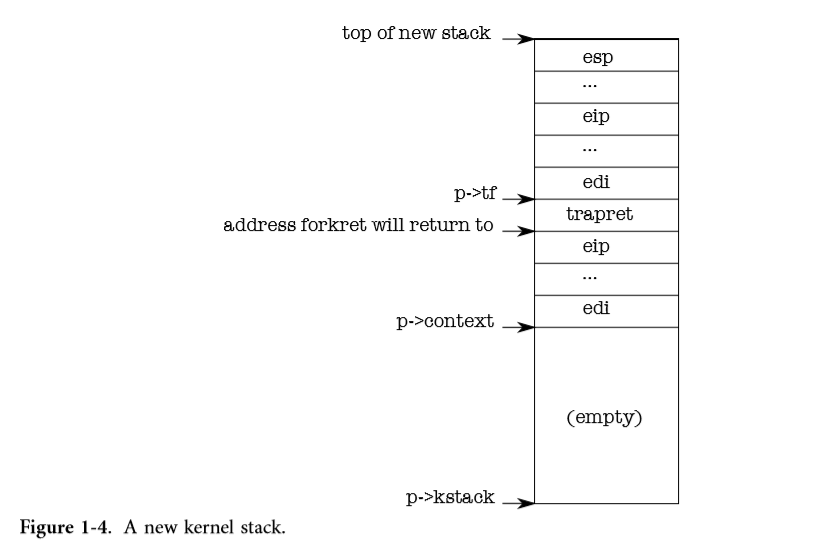




## Code: creating the ﬁrst process

现在，我们将了解内核如何创建用户级进程并确保它们是强隔离的。

在**main(**1217)初始化几个设备和子系统之后，它通过调用**userinit**(2520)创建第一个进程。**Userinit**的第一个动作是调用**allocproc**。**allocproc**(2473)的任务是在进程表中分配一个slot(一个struct **proc**)，并初始化进程内核线程执行所需的部分进程状态。对每个新进程调用**Allocproc**，而只对第一个进程调用**userinit**。**Allocproc**扫描**proc**表中状态为**UNUSED**的slot(2480-2482)。当发现一个未使用的slot时，**allocproc**将状态设置为**EMBRYO**，将其标记为used，并给进程一个惟一的**pid**(2469-2489)。接下来，它尝试为进程的内核线程分配一个内核堆栈。如果内存分配失败，**allocproc**将状态更改为**UNUSED**，并返回0表示失败。



现在，**allocproc**必须设置新进程的内核堆栈。编写**allocproc**是为了让fork和创建第一个进程时都可以使用它。**allocproc**用一个特别准备的内核堆栈和一组内核寄存器来设置新进程，这些寄存器会使它在第一次运行时“返回”到用户空间。准备好的内核堆栈的布局如图1-4所示。**allocproc**通过设置返回程序计数器值来完成这项工作的一部分，这些值将使新进程的内核线程首先在**forkret**中执行，然后在**trapret**(2507-2512)中执行。内核线程将开始执行从 p->context复制来的寄存器的内容。然后，设置**forkret** 为p->context->eip 使内核线程在**forkret**(2853)开始时执行。这个函数将返回堆栈底部的任何地址。内容环境切换代码(3059)将堆栈指针设置为刚好指向 p->context结束之后的位置。 **allocproc**将 p->context放在堆栈上，并将一个指向**trapret**的指针放在它上面;这就是 p->context将返回的地方。**trapret**从存储在内核堆栈顶部的值中恢复用户寄存器，并跳转到进程中(3324)。对于普通**fork**和创建第一个进程，这个设置是相同的，不过在后一种情况下，进程将在用户空间位置0开始执行，而不是在从**fork**返回时开始执行。

正如我们将在第3章看到的，通过一个中断机制,控制从用户软件转移到内核。中断机制被系统调用、中断和异常所使用。当进程运行时，只要将控制转移到内核，硬件和xv6 trap entry代码就会将用户寄存器保存在进程的内核堆栈上。**userinit**在新堆栈的顶部写入一些值，这些值与进程通过中断(2533-2539)进入内核时的值类似，这样从内核返回到进程的用户代码的普通代码就可以工作了。这些值是一个存储用户寄存器的**struct trapframe**。现在，新进程的内核堆栈已经完全准备好了，如图1-4所示。

第一个进程将执行一个小程序(**initcode.S**;(8400))。进程需要物理内存来存储这个程序，程序需要被复制到那个内存中，而且进程需要一个页表来将用户空间地址映射到那个内存中。

**userinit**调用**setupkvm**(1818)来为进程创建一个页表，其中(首先)仅对内核使用的内存进行映射。我们将在第2章中详细研究这个功能，但是在高层次上**setupkvm**和**userinit**创建一个如图1-2所示的地址空间。

第一个进程的用户空间内存的初始内容是**initcode.S**的编译形式;作为内核构建过程的一部分，链接器将二进制文件嵌入到内核中，并定义了两个特殊符号**\_binary\_initcode\_start**和**\_binary\_initcode\_size**，表示二进制文件的位置和大小。**Userinit**通过调用**inituvm**将二进制文件复制到新进程的内存中，**inituvm**将分配一页物理内存，将虚拟地址0映射到该内存，并将二进制文件复制到该页(1886)。

然后**userinit**使用初始的用户模式状态设置trap frame(0602): %cs寄存器包含一个针对SEG\_UCODE段的段选择器（segment selector），该段运行在特权级别**DPL\_USER**(用户模式而不是内核模式)，类似地，%ds、%es和%ss通过特权**DPL\_USER**来使用**SEG\_UDATA**。**%eflags FL\_IF**位被设置为允许硬件中断;我们将在第三章重新讨论这个问题。

堆栈指针**%esp**被设置为进程的最大有效虚拟地址(p->sz)。指令指针被设置为initcode的entry，地址0。

函数**DPL\_USER**主要为了调试将p->name设置为initcode。设置p->cwd来设置进程的当前工作目录;我们将在第6章详细介绍**namei**。

进程初始化后，  **userinit** 通过将p->state设置为**RUNNABLE**来标记进程可以被调度。

## Code: Running the ﬁrst process

现在已经准备好了第一个线程的状态，可以运行它了。在**main**调用**userinit**之后，**mpmain**调用**scheduler**来开始运行进程(1257)。**scheduler**(2758)查找一个p->state 状态设置为**RUNNABLE**的进程，并且只有一个进程:**initproc**。它将per-cpu变量**proc**设置为它找到的进程，并调用**switchuvm**来告诉硬件开始使用目标进程的页表(1879)。在内核中执行时更改页表是可行的，因为**setupkvm**使所有进程的页表对内核代码和数据具有相同的映射。**switchuvm**还设置了一个任务状态段**SEG\_TSS**，指示硬件在进程的内核堆栈上执行系统调用和中断。我们将在第3章重新学习任务状态部分。

**scheduler**现在将p->state设置为**RUNNING**，并调用**swtch**(3059)对目标进程的内核线程执行上下文切换。**swtch**首先保存当前寄存器。当前上下文不是一个进程，而是一个特殊的per-cpu调度程序环境，因此**scheduler**告诉**swtch**将当前硬件寄存器保存per-cpu storage (**cpu->scheduler**) 中，而不是保存在任何进程的内核线程环境中。**swtch**然后将保存的目标内核线程寄存器(**p->context**)加载到x86硬件寄存器中，包括堆栈指针和指令指针。我们将在第5章更详细地研究**swtch**。最后一条**ret**指令(3078)从堆栈中pop目标进程的**%eip**，完成环境切换。现在处理器运行在进程**p**的内核堆栈上。

**Allocproc**之前已经将**initproc**的**p-->context->eip** 设置为**forkret**，因此**ret**开始执行**forkret**。在第一次调用时(就是这一次)，**forkret**(2853)运行初始化函数。而这些初始化函数不能从**main**运行，因为它们必须在具有自己内核堆栈的常规进程的环境中运行。然后,**forkret**返回。**Allocproc**将 p->context 从栈中pop后的栈顶处理为**trapret**，所以现在**trapret**开始执行，将**%esp**设置为**p->tf**。**Trapret**(3324)使用pop指令从 trap frame (0602)恢复寄存器，就像swtch对内核环境所做的那样: **popal** 恢复常规寄存器，然后**popl**指令恢复**%gs、%fs、%es**和**%ds**。**addl**跳过了两个字段**trapno**和**errcode**。最后，**iret**指令从堆栈中pop **%cs、%eip、%flags, %esp**和**%ss**。trap frame 的环境已经转移到CPU状态，因此处理器继续以trap frame 中指定的**%eip**运行。对于**initproc**，这意味着虚拟地址0，**initcode.S**的第一个指令。

此时，**%eip为**0，**%esp为**4096。这些是进程地址空间中的虚拟地址。处理器的分页硬件将它们转换为物理地址。**allocuvm**已经设置了进程的页表，以便虚拟地址0引用为该进程分配到的的物理内存，并设置了一个标志(**PTE\_U**)，**PTE\_U**告诉分页硬件允许用户代码访问该内存。**userinit**(2533)设置了**%cs**的低位，以CPL=3运行进程的用户代码，这意味着用户代码只能使用设置了**PTE\_U**的页面，并且不能修改敏感的硬件寄存器，如%cr3。因此进程只能使用自己的内存。

## The ﬁrst system call: exec

现在我们已经了解了内核如何为进程提供强大的隔离，接下来让我们看看用户级进程如何重新进入内核，以请求它自己不能执行的服务。

**initcode.S** 的第一个动作是调用**exec**系统调用。正如我们在第0章中看到的，**exec**将当前进程的内存和寄存器替换为新的程序，但它保留文件描述符、进程id和父进程不变。

**Initcode.S**(8409)首先在堆栈上push三个值——**$argv、$init**和**$0**，然后将**%eax**设置为**SYS\_exec**，并执行**int T\_SYSCALL**:它要求内核运行**exec**系统调用。如果一切顺利，**exec**永远不会返回:它开始运行由**$init**命名的程序，它是一个指向以空结尾的字符串**/init**(8422-8424)的指针。另一个参数是命令行参数的**argv**数组;数组末尾的0表示数组的结束。如果**exec**失败并返回，initcode循环调用**exit**系统调用，它肯定不应该返回(8416-8420)。

这段代码手工起草了第一个系统调用，看起来像一个普通的系统调用，我们将在第3章看到。与前面一样，这种设置避免了对第一个进程进行特殊封装(在本例中，它的第一个系统调用)，而是重用xv6必须为标准操作提供的代码。

第2章将详细介绍**exec**的实现，但在较高的层次上，它将**initcode**替换为从文件系统中加载的**/init**二进制文件。现在**initcode**(8400)已经完成，进程将运行**/init**。**Init**(8510)在需要时创建一个新的控制台设备文件，然后将其根据文件描述符0、1和2打开。然后循环，启动一个控制台shell，处理孤立的僵尸，直到shell退出，并重复。

系统终于启动啦！！！！！

## Real word

在现实世界中，单片内核和微内核都有。许多Unix内核都是单片的。例如，Linux有一个单片内核，尽管一些操作系统功能是作为用户级服务器运行的(例如，窗口系统)。像L4、Minix、QNX这样的内核被组织为带有服务器的微内核，并在嵌入式设置中得到了广泛的应用。

大多数操作系统都采用了进程的概念，而且大多数进程看起来与xv6的类似。一个真正的操作系统会在o(1)内使用显式的空闲列表找到空闲的**proc**结构，而不是用**allocproc**进行线性时间搜索;为了简单起见，xv6使用线性扫描(许多方法中的第一种)。

Exercises

1. 在swtch设置断点。使用gdb的**stepi**从**re**t一步到**forkret**，然后使用gdb的**finish**继续到**trapret**，然后**stepi**直到你到达虚拟地址0处的**initcode**。

2. **KERNBASE**限制了单个进程可以使用的内存量，这对于一台拥有全部4 GB RAM的机器来说可能很恼人。提高**KERNBASE**是否允许进程使用更多内存?

