huihongxiao GitBook: [master] 24 pa... Latest commit acec9d0 on Apr 5 History

1 contributor

72 lines (42 sloc) 8.98 KB Raw Blame

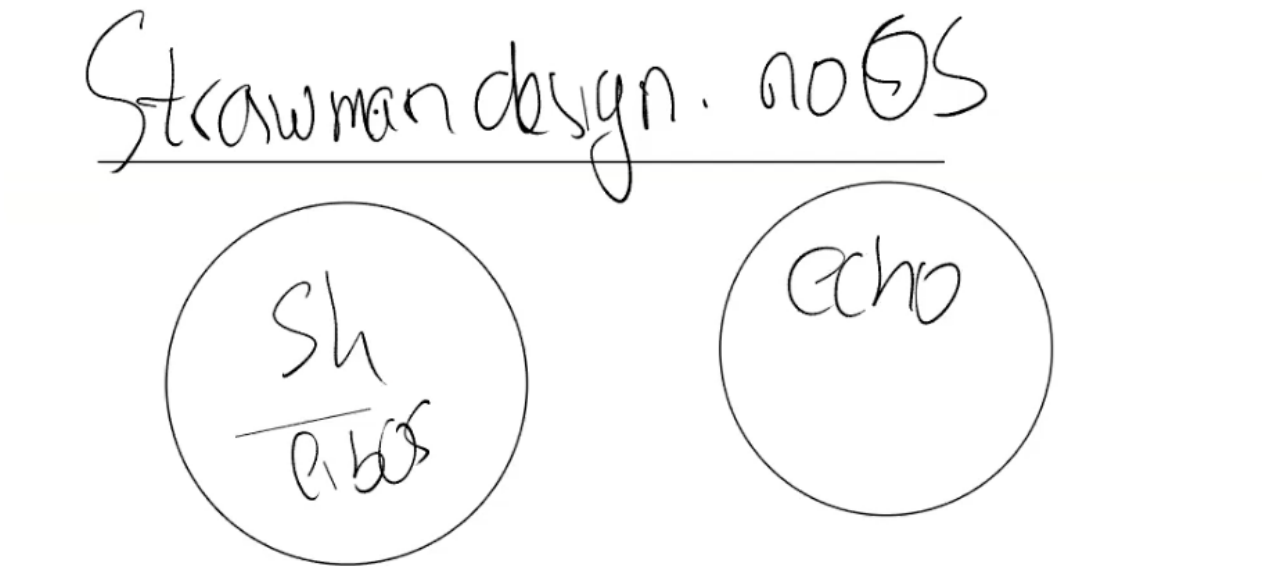
3.2 操作系统隔离性（isolation）

我们首先来简单的介绍一下隔离性（isolation），以及介绍为什么它很重要，为什么我们需要关心它?

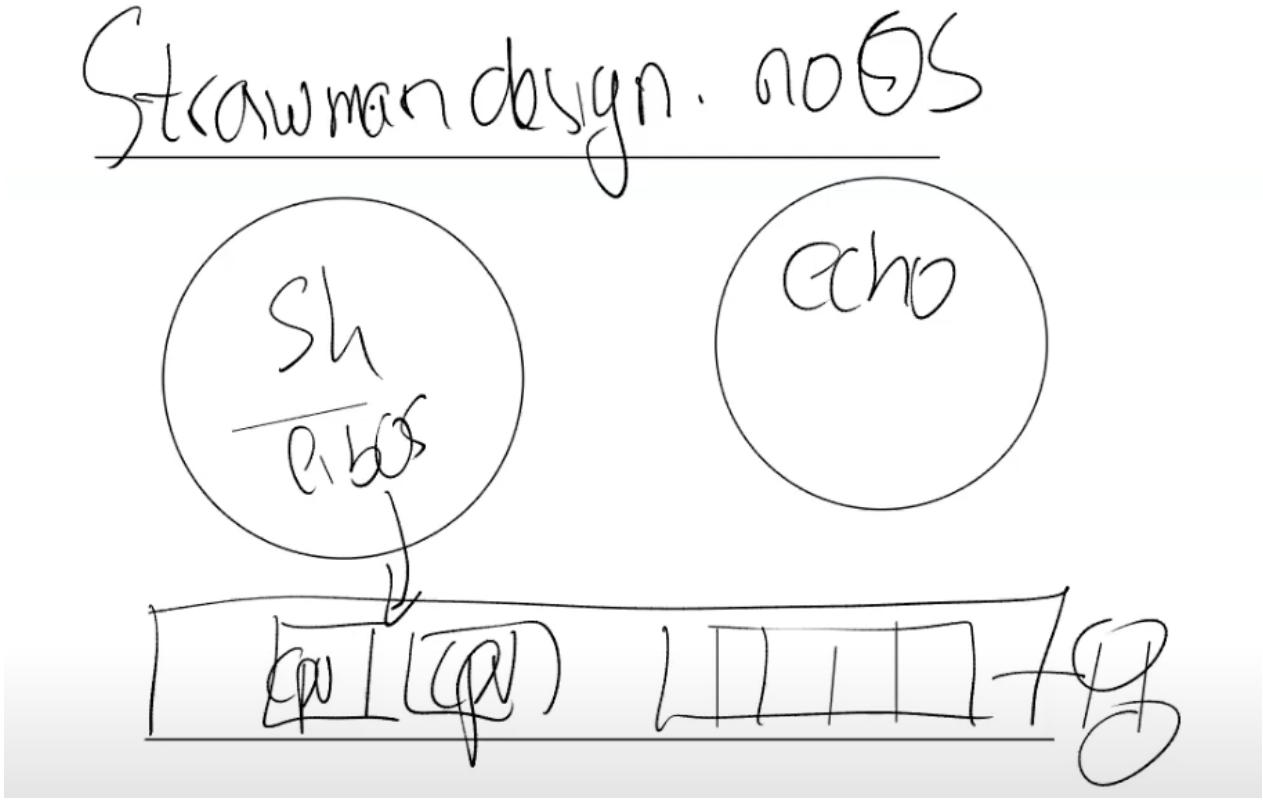
这里的核心思想相对来说比较简单。我们在用户空间有多个应用程序，例如Shell，echo，find。但是，如果你通过Shell运行你们的Prime代码（lab1中的一个部分）时，假设你们的代码出现了问题，Shell不应该会影响到其他的应用程序。举个反例，如果Shell出现问题时，杀掉了其他的进程，这将会非常糟糕。所以你需要在不同的应用程序之间有强隔离性。

类似的，操作系统某种程度上为所有的应用程序服务。当你的应用程序出现问题时，你会希望操作系统不会因此而崩溃。比如说你向操作系统传递了一些奇怪的参数，你会希望操作系统仍然能够很好的处理它们（能较好的处理异常情况）。所以，你也需要在应用程序和操作系统之间有强隔离性。

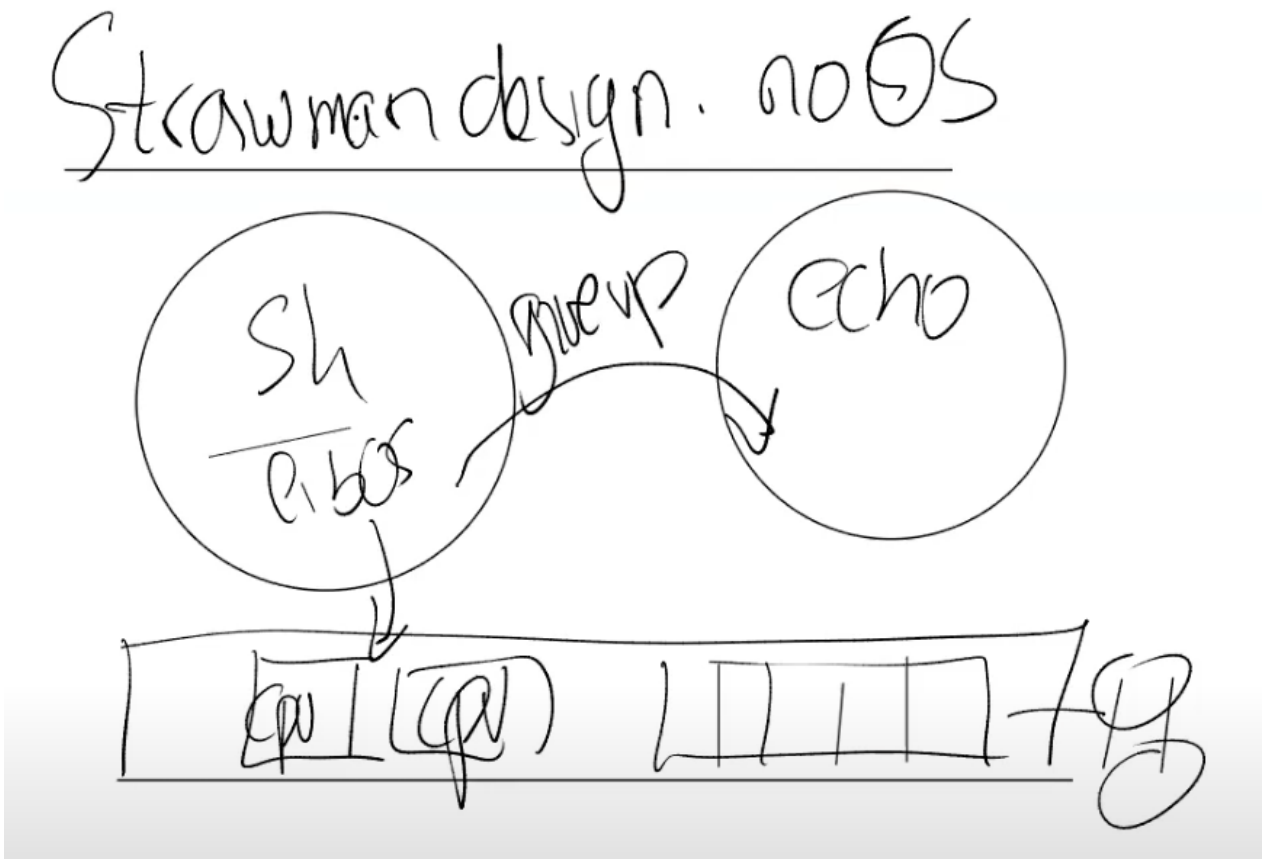
这里我们可以这样想，如果没有操作系统会怎样？我们可以用稻草人提案法（就是通过头脑风暴找出缺点）来考虑一下，如果没有操作系统，或者操作系统只是一些库文件，比如说你在使用Python，通过import os你就可以将整个操作系统加载到你的应用程序中。那么现在，我们有一个Shell，并且我们引用了代表操作系统的库。同时，我们有一些其他的应用程序例如，echo。



通常来说，如果没有操作系统，应用程序会直接与硬件交互。比如，应用程序可以直接看到CPU的多个核，看到磁盘，内存。所以现在应用程序和硬件资源之间没有一个额外的抽象层，如下图所示。

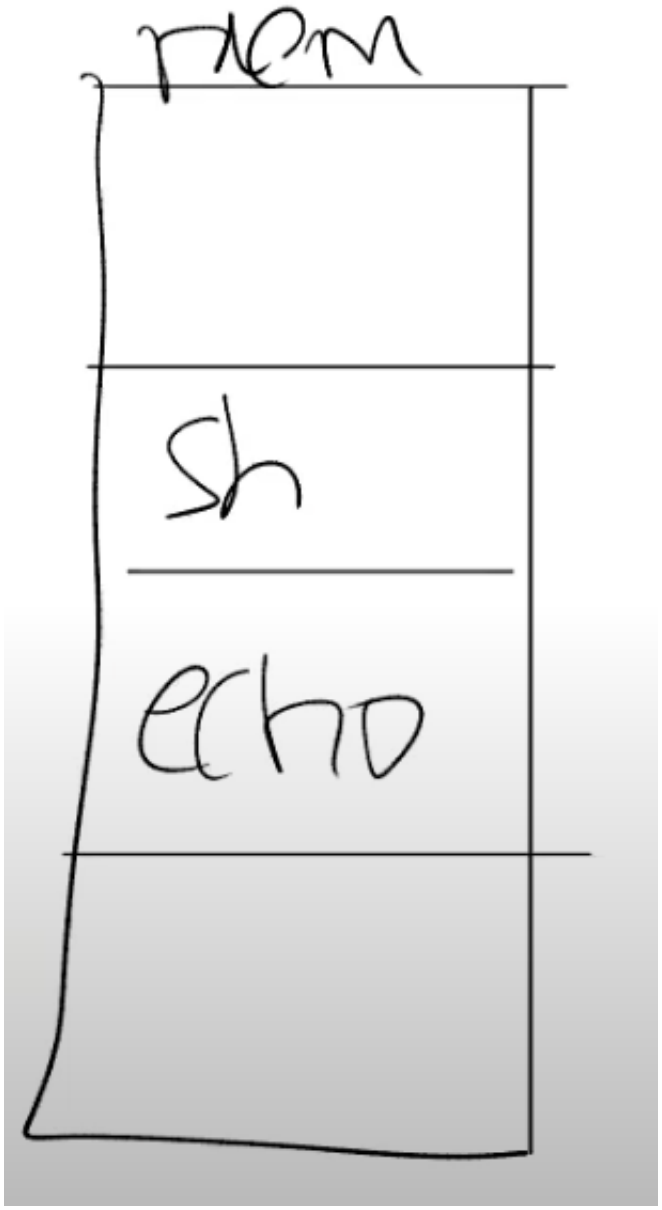


实际上，从隔离性的角度来看，这并不是一个很好的设计。这里你可以看到这种设计是如何破坏隔离性的。使用操作系统的一个目的是为了同时运行多个应用程序，所以时不时的，CPU会从一个应用程序切换到另一个应用程序。我们假设硬件资源里只有一个CPU核，并且我们现在在这个CPU核上运行Shell。但是时不时的，也需要让其他的应用程序也可以运行。现在我们没有操作系统来帮我们完成切换，所以Shell就需要时不时的释放CPU资源。

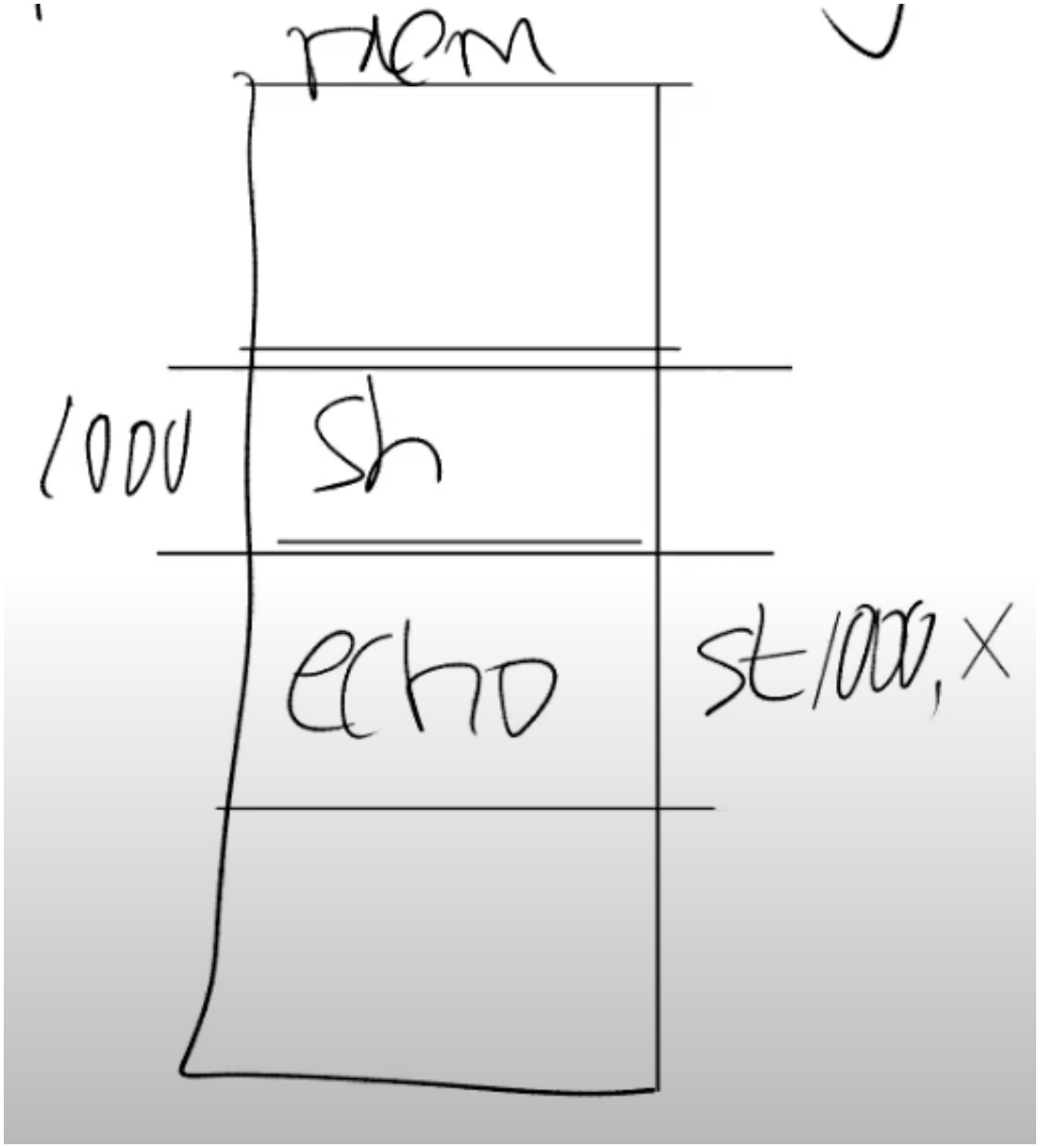


为了不变成一个恶意程序，Shell在发现自己运行了一段时间之后，需要让别的程序也有机会能运行。这种机制有时候称为协同调度（Cooperative Scheduling）。但是这里的场景并没有很好的隔离性，比如说Shell中的某个函数有一个死循环，那么Shell永远也不会释放CPU，进而其他的应用程序也不能够运行，甚至都不能运行一个第三方的程序来停止或者杀死Shell程序。所以这种场景下，我们基本上得不到真正的multiplexing（CPU在多进程同分时复用）。而这个特性是非常有用的，不论应用程序在执行什么操作，multiplexing都会迫使应用程序时不时的释放CPU，这样其他的应用程序才能运行。

从内存的角度来说，如果应用程序直接运行在硬件资源之上，那么每个应用程序的文本，代码和数据都直接保存在物理内存中。物理内存中的一部分被Shell使用，另一部分被echo使用。



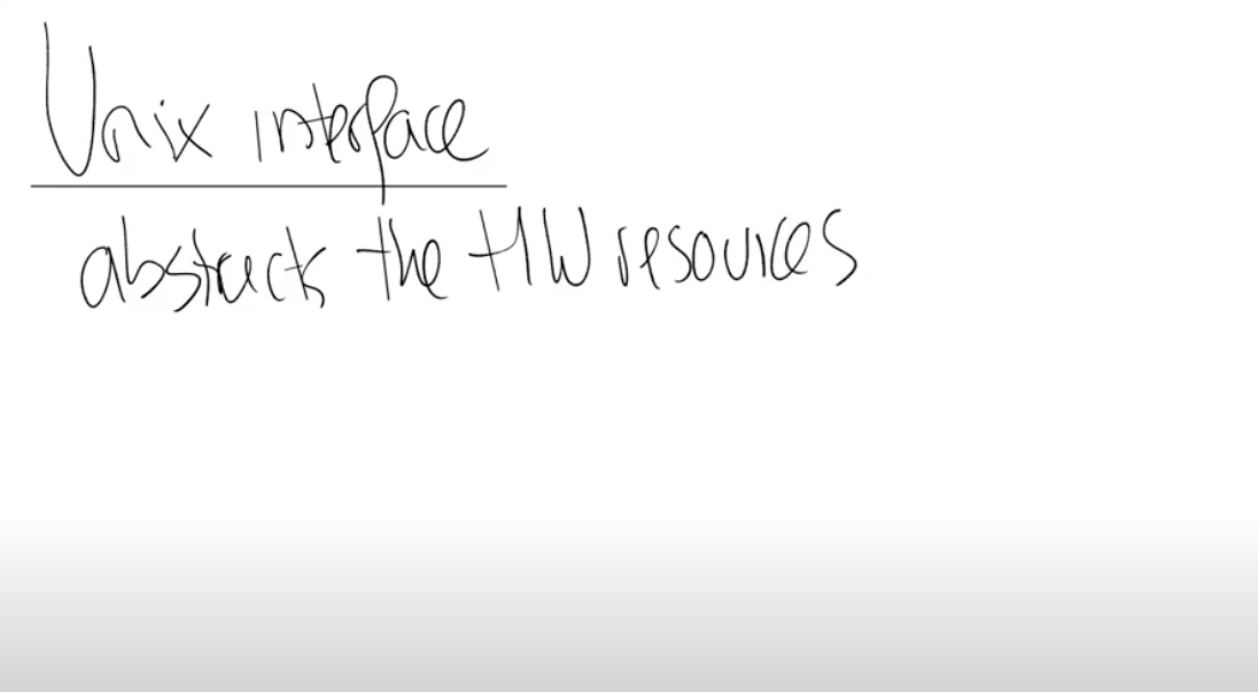
即使在这么简单的例子中，因为两个应用程序的内存之间没有边界，如果echo程序将数据存储在属于Shell的一个内存地址中（下图中的1000），那么就echo就会覆盖Shell程序内存中的内容。



这是非常不想看到的场景，因为echo现在渗透到了Shell中来，并且这类的问题是非常难定位的。所以这里也没有为我们提供好的隔离性。我们希望不同应用程序之间的内存是隔离的，这样一个应用程序就不会覆盖另一个应用程序的内存。

使用操作系统的一个原因，甚至可以说是主要原因就是为了实现multiplexing和内存隔离。如果你不使用操作系统，并且应用程序直接与硬件交互，就很难实现这两点。所以，将操作系统设计成一个库，并不是一种常见的设计。你或许可以在一些实时操作系统中看到这样的设计，因为在这些实时操作系统中，应用程序之间彼此相互信任。但是在大部分的其他操作系统中，都会强制实现硬件资源的隔离。

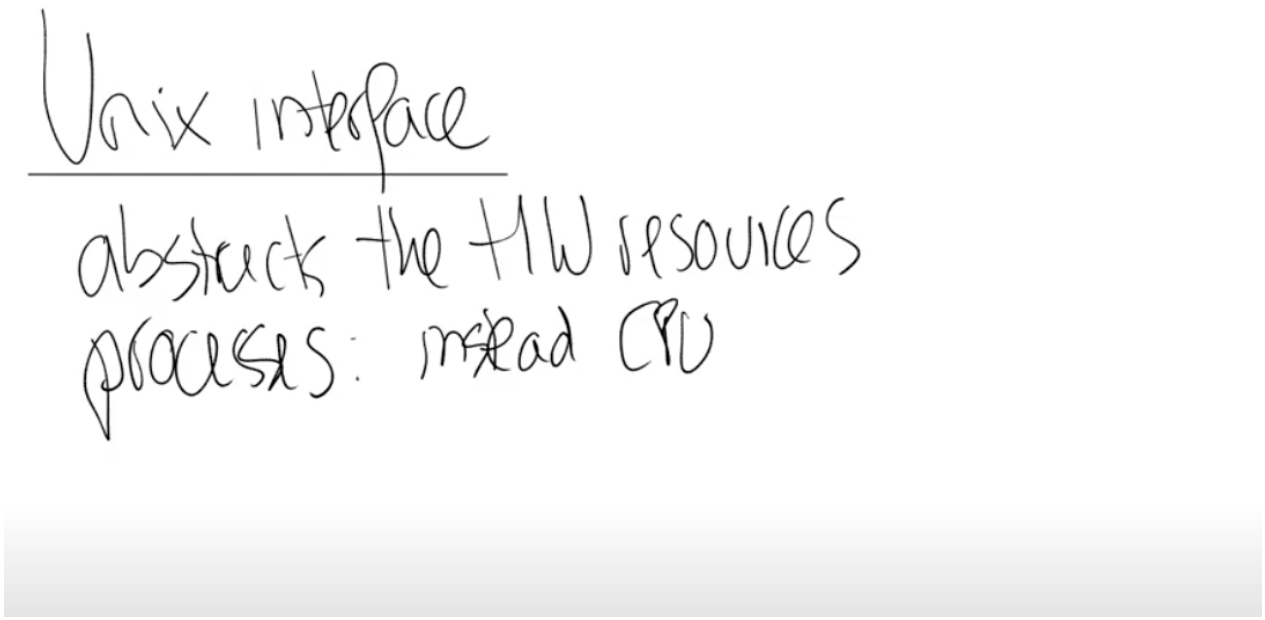
如果我们从隔离的角度来稍微看看Unix接口，那么我们可以发现，接口被精心设计以实现资源的强隔离，也就是multiplexing和物理内存的隔离。接口通过抽象硬件资源，从而使得提供强隔离性成为可能。



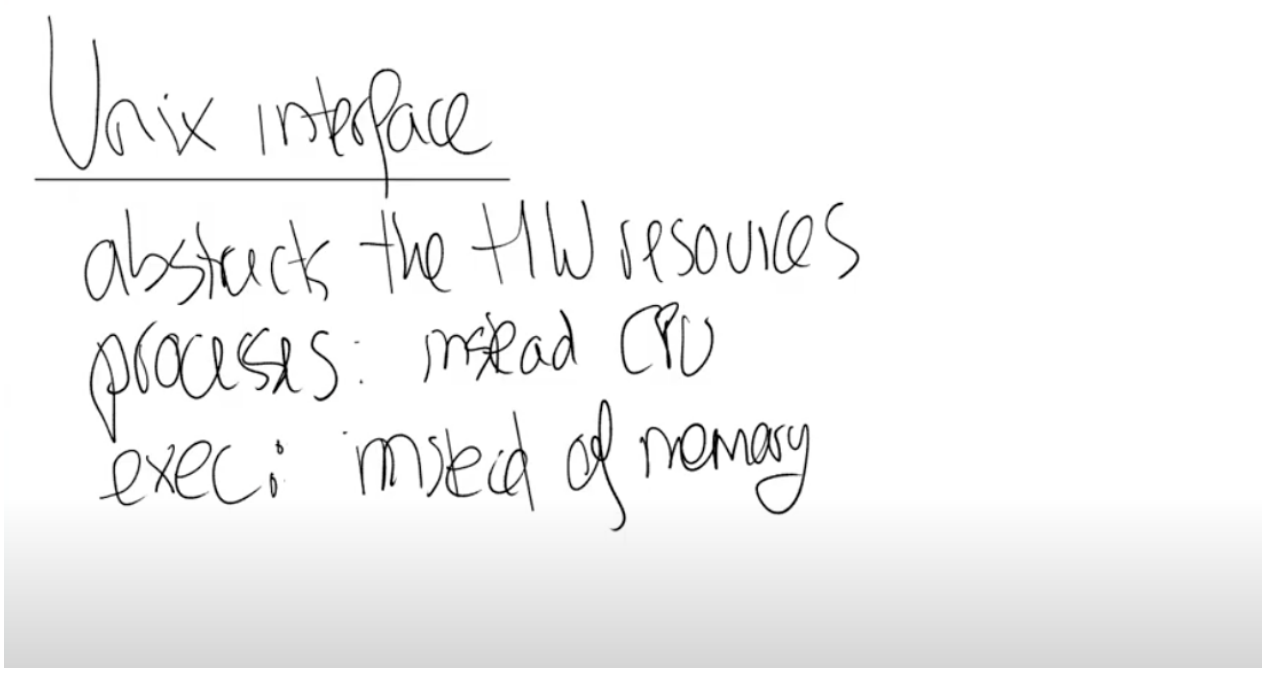
接下来我们举几个例子。

之前通过fork创建了进程。进程本身不是CPU，但是它们对应了CPU，它们使得你可以在CPU上运行计算任务。所以你懂的，应用程序不能直接与CPU交互，只能与进程交互。操作系统内核会完成不同进程在CPU上的切换。所以，操作系统不是直接将CPU提供给应用程序，而是向应用程序提供“进程”，进程抽象了CPU，这样操作系统才能在多个应用程序之间复用一个或者多个CPU。

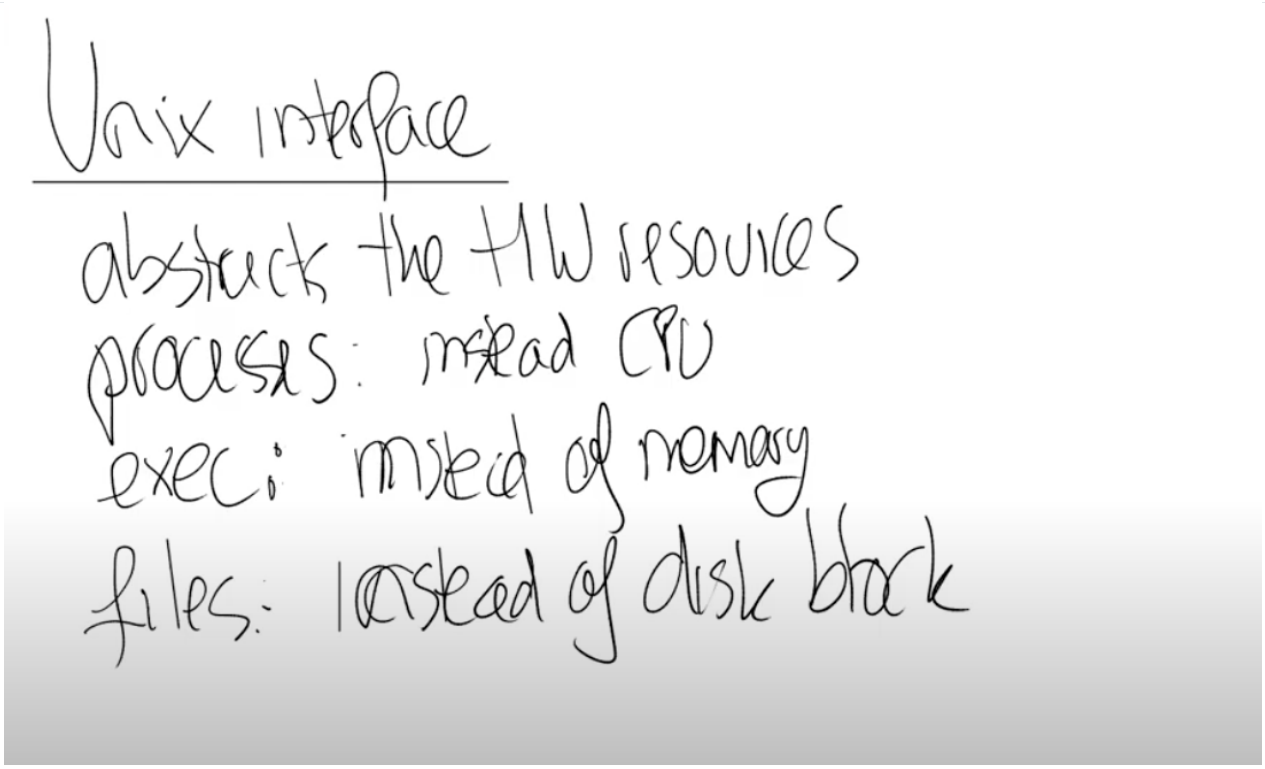
> 学生提问：这里说进程抽象了CPU，是不是说一个进程使用了部分的CPU，另一个进程使用了CPU的另一部分？这里CPU和进程的关系是什么？
>
> Frans教授：我这里真实的意思是，我们在实验中使用的RISC-V处理器实际上是有4个核。所以你可以同时运行4个进程，一个进程占用一个核。但是假设你有8个应用程序，操作系统会分时复用这些CPU核，比如说对于一个进程运行100毫秒，之后内核会停止运行并将那个进程从CPU中卸载，再加载另一个应用程序并再运行100毫秒。通过这种方式使得每一个应用程序都不会连续运行超过100毫秒。这里只是一些基本概念，我们在接下来的几节课中会具体的看这里是如何实现的。
>
> 学生提问：好的，但是多个进程不能在同一时间使用同一个CPU核，对吧？
>
> Frans教授：是的，这里是分时复用。CPU运行一个进程一段时间，再运行另一个进程。



我们可以认为exec抽象了内存。当我们在执行exec系统调用的时候，我们会传入一个文件名，而这个文件名对应了一个应用程序的内存镜像。内存镜像里面包括了程序对应的指令，全局的数据。应用程序可以逐渐扩展自己的内存，但是应用程序并没有直接访问物理内存的权限，例如应用程序不能直接访问物理内存的1000-2000这段地址。不能直接访问的原因是，操作系统会提供内存隔离并控制内存，操作系统会在应用程序和硬件资源之间提供一个中间层。exec是这样一种系统调用，它表明了应用程序不能直接访问物理内存。



另一个例子是files，files基本上来说抽象了磁盘。应用程序不会直接读写挂在计算机上的磁盘本身，并且在Unix中这也是不被允许的。在Unix中，与存储系统交互的唯一方式就是通过files。Files提供了非常方便的磁盘抽象，你可以对文件命名，读写文件等等。之后，操作系统会决定如何将文件与磁盘中的块对应，确保一个磁盘块只出现在一个文件中，并且确保用户A不能操作用户B的文件。通过files的抽象，可以实现不同用户之间和同一个用户的不同进程之间的文件强隔离。



这些都是你们在上一个lab中使用的Unix系统调用接口，或许你们已经看出来了，这些接口看起来像是经过精心的设计以抽象计算机资源，这样这些接口的实现，或者说操作系统本身可以在多个应用程序之间复用计算机硬件资源，同时还提供了强隔离性。

这里有什么问题吗?

> 学生提问：更复杂的内核会不会尝试将进程调度到同一个CPU核上来减少Cache Miss?
>
> Frans教授：是的。有一种东西叫做Cache affinity。现在的操作系统的确非常复杂，并且会尽量避免Cache miss和类似的事情来提升性能。我们在这门课程后面介绍高性能网络的时候会介绍更多相关的内容。
>
> 学生提问：XV6的代码中，哪一部分可以看到操作系统为多个进程复用了CPU?
>
> Frans教授：有挺多文件与这个相关，但是proc.c应该是最相关的一个。两三周之后的课程中会有一个话题介绍这个内容。我们会看大量的细节，并展示操作系统的multiplexing是如何发生的。所以可以这么看待这节课，这节课的内容是对许多不同内容的初始介绍，因为我们总得从某个地方开始吧。