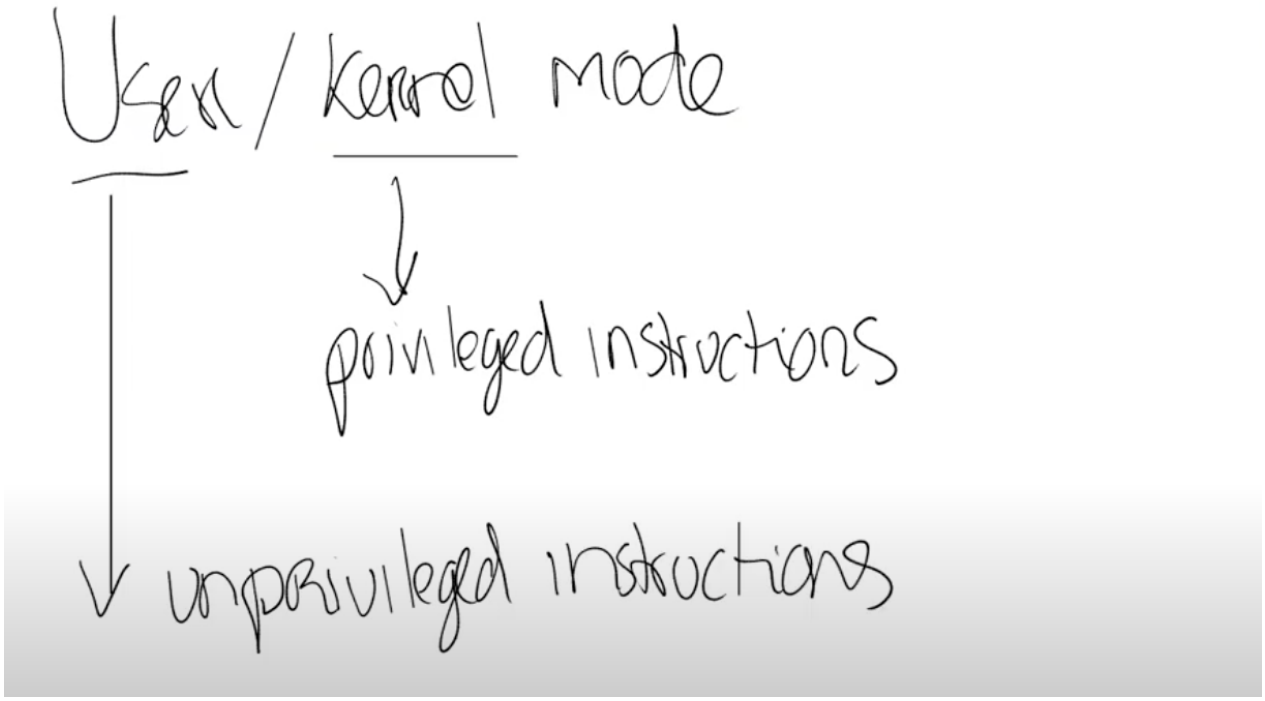
huihongxiao GitBook: [master... Latest commit 540f202 on Oct 17, 2020 History

1 contributor

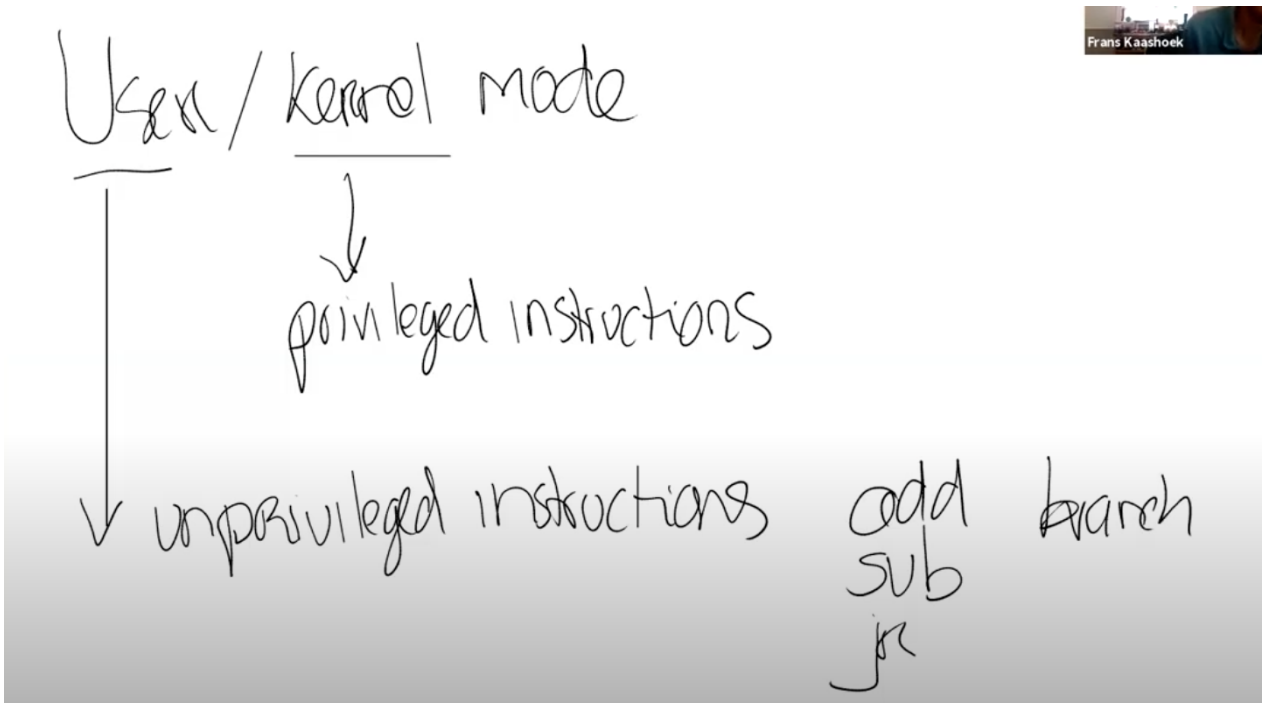
3.4 硬件对于强隔离的支持

硬件对于强隔离的支持包括了：user/kernle mode和虚拟内存。

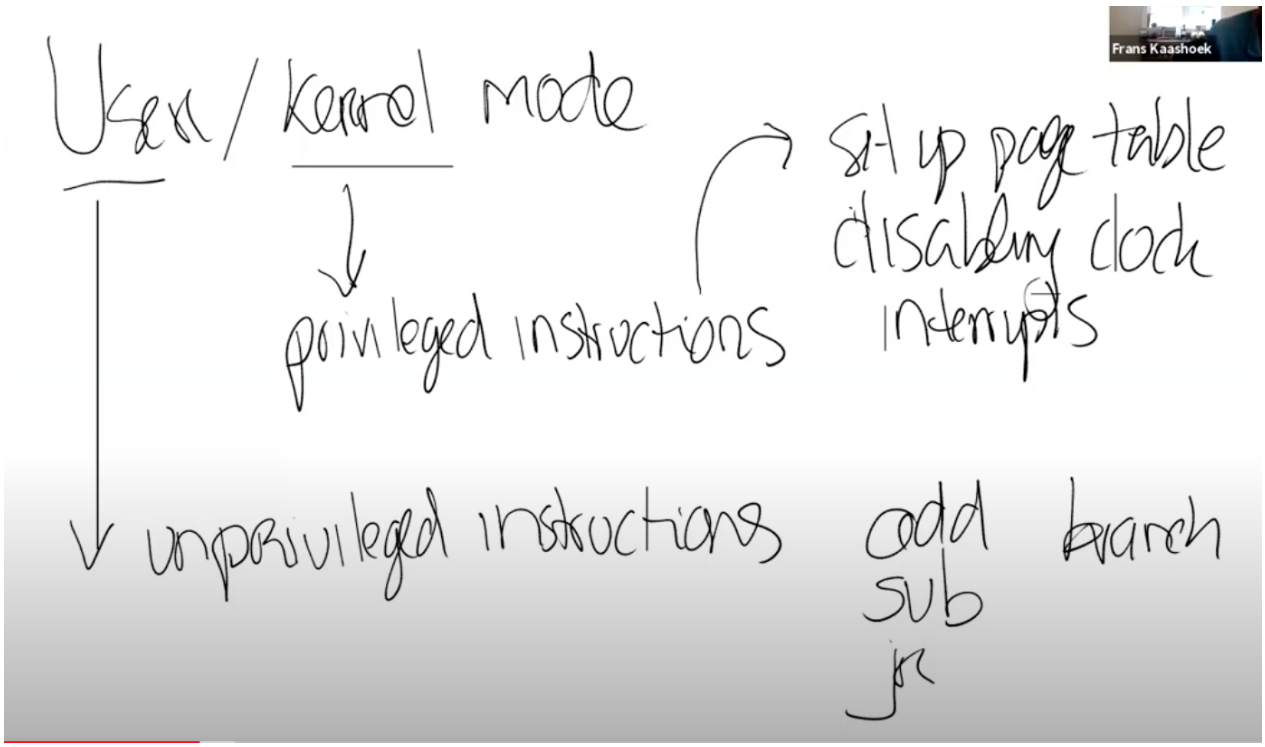
首先，我们来看一下user/kernel mode，这里会以尽可能全局的视角来介绍，有很多重要的细节在这节课中都不会涉及。为了支持user/kernel mode，处理器会有两种操作模式，第一种是user mode，第二种是kernel mode。当运行在kernel mode时，CPU可以运行特定权限的指令（privileged instructions）；当运行在user mode时，CPU只能运行普通权限的指令（unprivileged instructions）。



普通权限的指令都是一些你们熟悉的指令，例如将两个寄存器相加的指令ADD、将两个寄存器相减的指令SUB、跳转指令JRC、BRANCH指令等等。这些都是普通权限指令，所有的应用程序都允许执行这些指令。

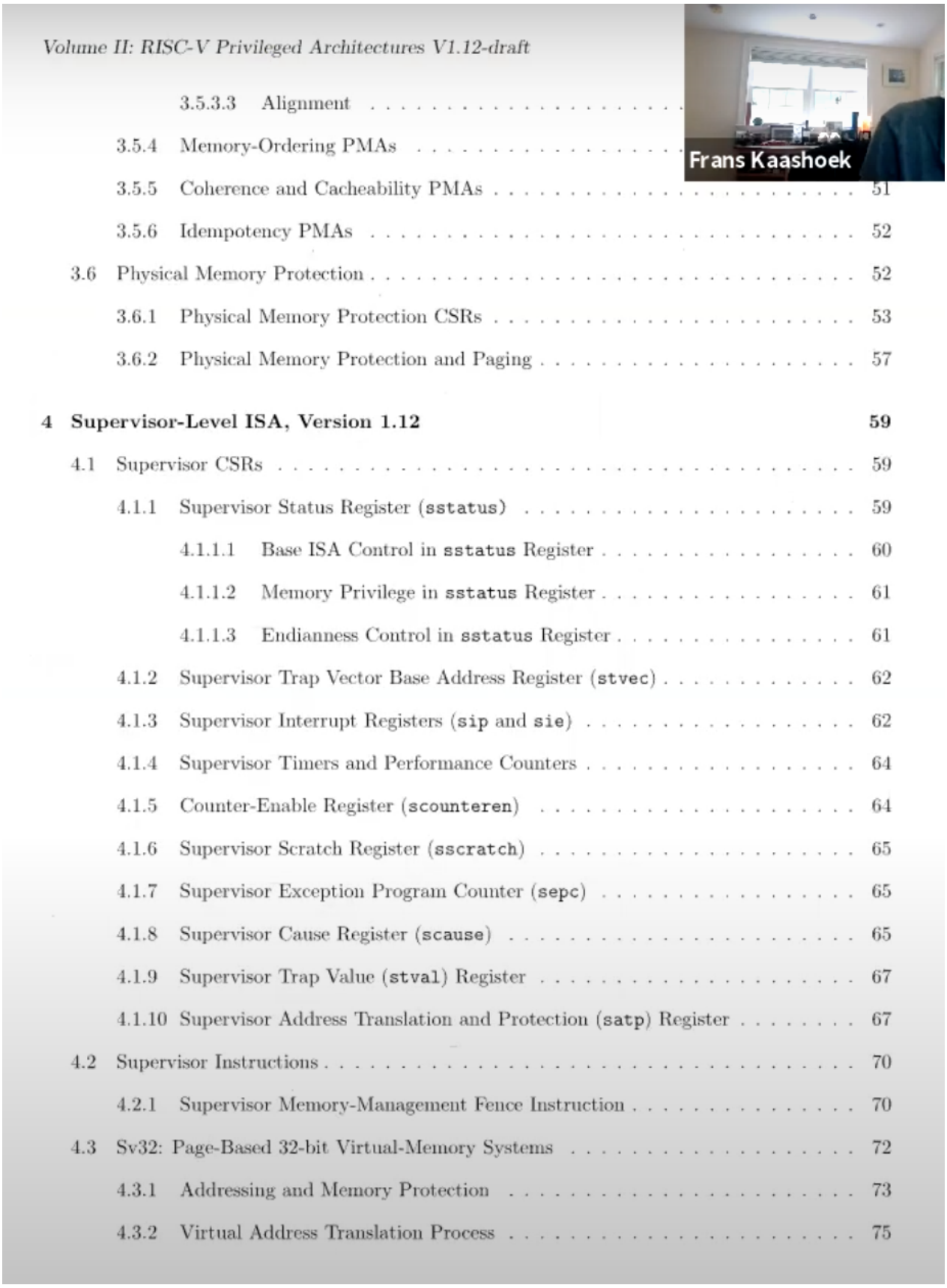


特殊权限指令主要是一些直接操纵硬件的指令和设置保护的指令，例如设置page table寄存器、关闭时钟中断。在处理器上有各种各样的状态，操作系统会使用这些状态，但是只能通过特殊权限指令来变更这些状态。



举个例子，当一个应用程序尝试执行一条特殊权限指令，因为不允许在user mode执行特殊权限指令，处理器会拒绝执行这条指令。通常来说，这时会将控制权限从user mode切换到kernel mode，当操作系统拿到控制权之后，或许会杀掉进程，因为应用程序执行了不该执行的指令。

下图是RISC-V privilege架构的文档，这个文档包括了所有的特殊权限指令。在接下来的一个月，你们都会与这些特殊权限指令打交道。我们下节课就会详细介绍其中一些指令。这里我们先对这些指令有一些初步的认识：应用程序不应该执行这些指令，这些指令只能被内核执行。



这里是硬件支持强隔离的一个方面。

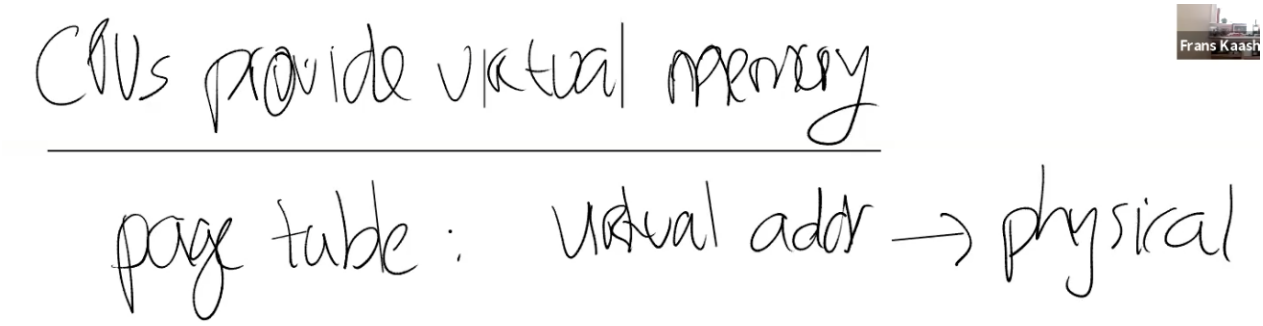
> 学生提问：如果kernel mode允许一些指令的执行，user mode不允许一些指令的执行，那么是谁在检查当前的mode并实际运行这些指令，并且怎么知道当前是不是kernel mode？是有什么标志位吗？
>
> Frans教授：是的，在处理器里面有一个flag。在处理器的一个bit，当它为1的时候是user mode，当它为0时是kernel mode。当处理器在解析指令时，如果指令是特殊权限指令，并且该bit被设置为1，处理器会拒绝执行这条指令，就像在运算时不能除以0一样。
>
> 同一个学生继续问：所以，唯一的控制方式就是通过某种方式更新了那个bit？
>
> Frans教授：你认为是什么指令更新了那个bit位？是特殊权限指令还是普通权限指令？（等了一会，那个学生没有回答）。很明显，设置那个bit位的指令必须是特殊权限指令，因为应用程序不应该能够设置那个bit到kernel mode，否则的话应用程序就可以运行各种特殊权限指令了。所以那个bit是被保护的，这样回答了你的问题吗？

许多同学都已经知道了，实际上RISC-V还有第三种模式称为machine mode。在大多数场景下，我们会忽略这种模式，所以我也不太会介绍这种模式。 所以实际上我们有三级权限（user/kernel/machine），而不是两级(user/kernel)。

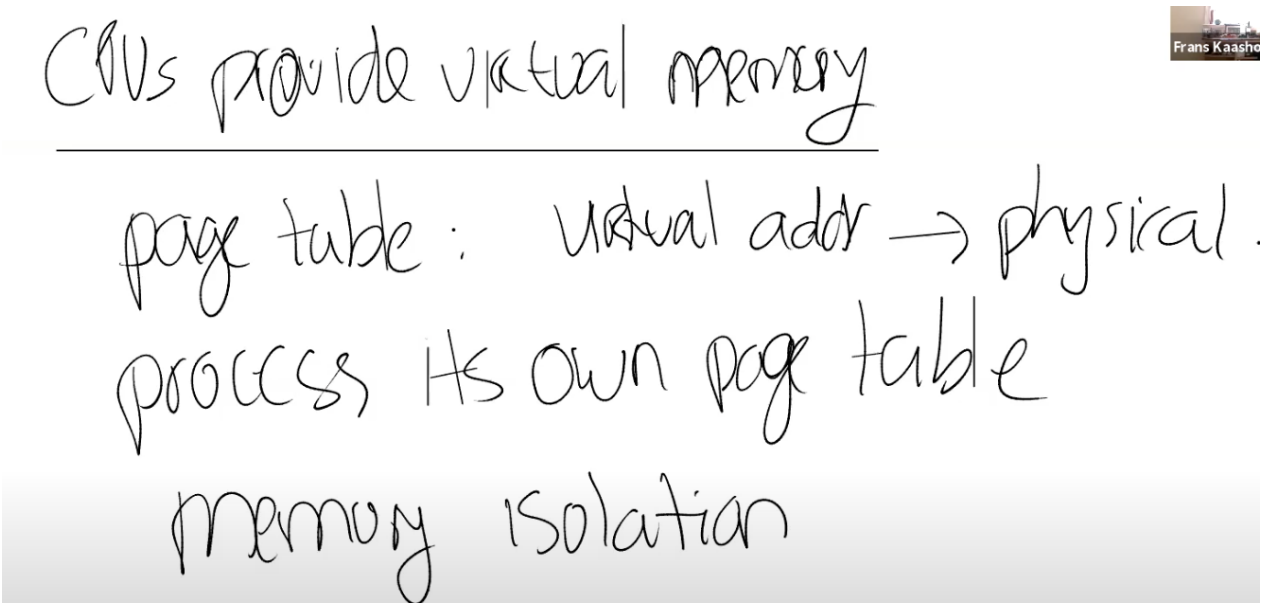
> 学生提问：考虑到安全性，所有的用户代码都会通过内核访问硬件，但是有没有可能一个计算机的用户可以随意的操纵内核？
>
> Frans教授：并不会，至少小心的设计就不会发生这种事。或许一些程序会有额外的权限，操作系统也会认可这一点。但是这些额外的权限并不会给每一个用户，比如只有root用户有特定的权限来完成安全相关的操作。
>
> 同一个学生提问：那BIOS呢？BIOS会在操作系统之前运行还是之后？
>
> Frans教授：BIOS是一段计算机自带的代码，它会先启动，之后它会启动操作系统，所以BIOS需要是一段可被信任的代码，它最好是正确的，且不是恶意的。
>
> 学生提问：之前提到，设置处理器中kernel mode的bit位的指令是一条特殊权限指令，那么一个用户程序怎么才能让内核执行任何内核指令？因为现在切换到kernel mode的指令都是一条特殊权限指令了，对于用户程序来说也没法修改那个bit位。

> Frans教授：你说的对，这也是我们想要看到的结果。可以这么来看这个问题，首先这里不是完全按照你说的方式工作，在RISC-V中，如果你在用户空间（user space）尝试执行一条特殊权限指令（后面Frans那边的Zoom就断了，等他重新接入，他也没有再继续回答，所以后半段回答是我补充的）用户程序会通过系统调用来切换到kernel mode。当用户程序执行系统调用，会通过ECALL触发一个软中断（software interrupt），软中断会查询操作系统预先设定的中断向量表，并执行中断向量表中包含的中断处理程序。中断处理程序在内核中，这样就完成了user mode到kernel mode的切换，并执行用户程序想要执行的特殊权限指令。

我们接下来看看硬件对于支持强隔离性的第二个特性，基本上所有的CPU都支持虚拟内存。我下节课会更加深入的讨论虚拟内存，这里先简单看一下。基本上来说，处理器包含了page table，而page table将虚拟内存地址与物理内存地址做了对应。

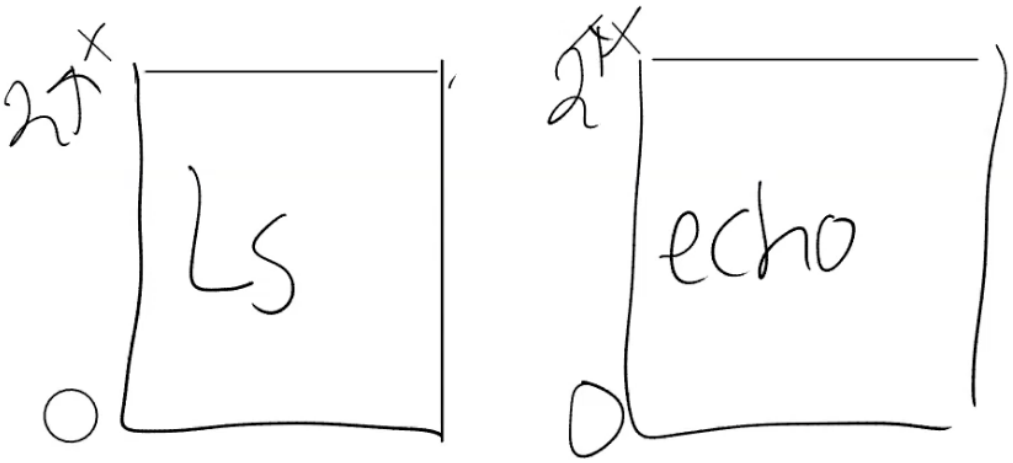


每一个进程都会有自己独立的page table，这样的话，每一个进程只能访问出现在自己page table中的物理内存。操作系统会设置page table，使得每一个进程都有不重合的物理内存，这样一个进程就不能访问其他进程的物理内存，因为其他进程的物理内存都不在它的page table中。一个进程甚至都不能随意编造一个内存地址，然后通过这个内存地址来访问其他进程的物理内存。这样就给了我们内存的强隔离性。



基本上来说，page table定义了对于内存的视图，而每一个用户进程都有自己对于内存的独立视图。这给了我们非常强的内存隔离性。

基于硬件的支持，我们可以重新画一下之前的一张图，我们先画一个矩形，ls程序位于这个矩形中；再画一个矩形，echo程序位于这个矩形中。每个矩形都有一个虚拟内存地址，从0开始到2的n次方。



这样，ls程序有了一个内存地址0，echo程序也有了一个内存地址0。但是操作系统会将两个程序的内存地址0映射到不同的物理内存地址，所以ls程序不能访问echo程序的内存，同样echo程序也不能访问ls程序的内存。

类似的，内核位于应用程序下方，假设是XV6，那么它也有自己的内存地址空间，并且与应用程序完全独立。