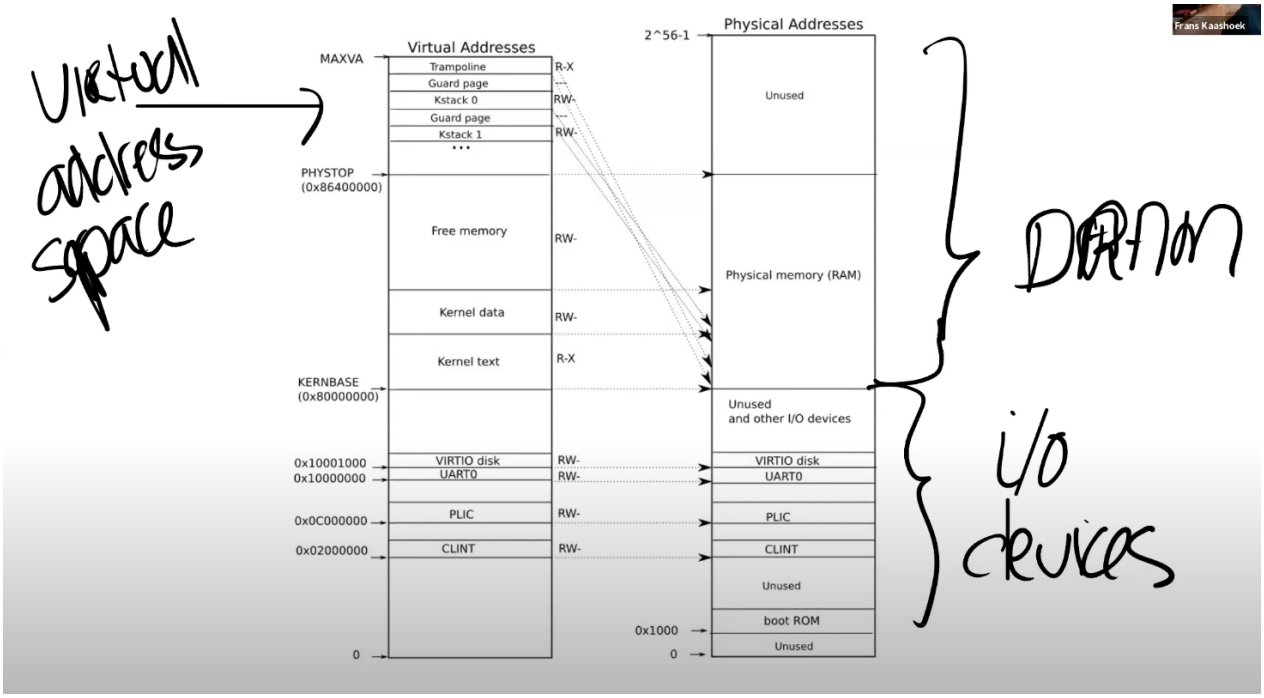
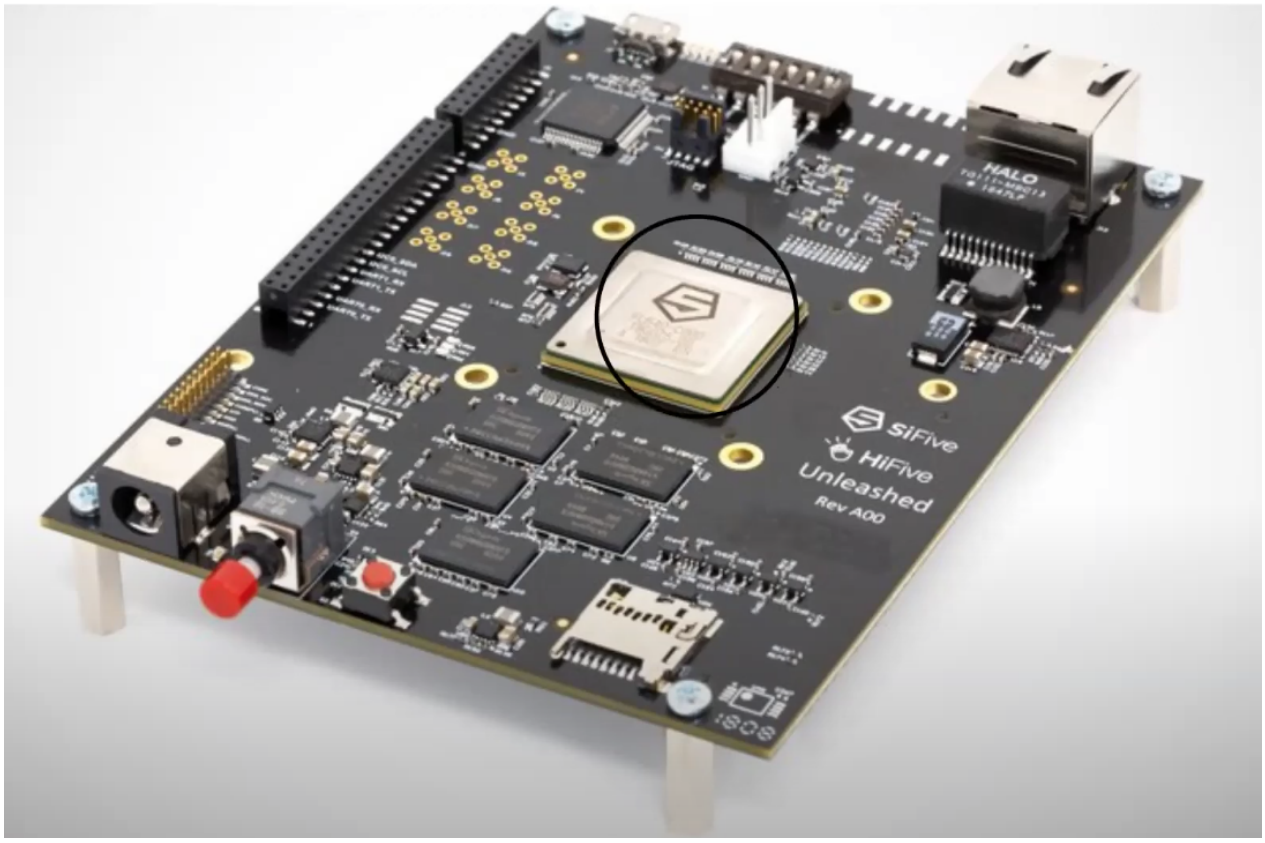
4.5 Kernel Page Table

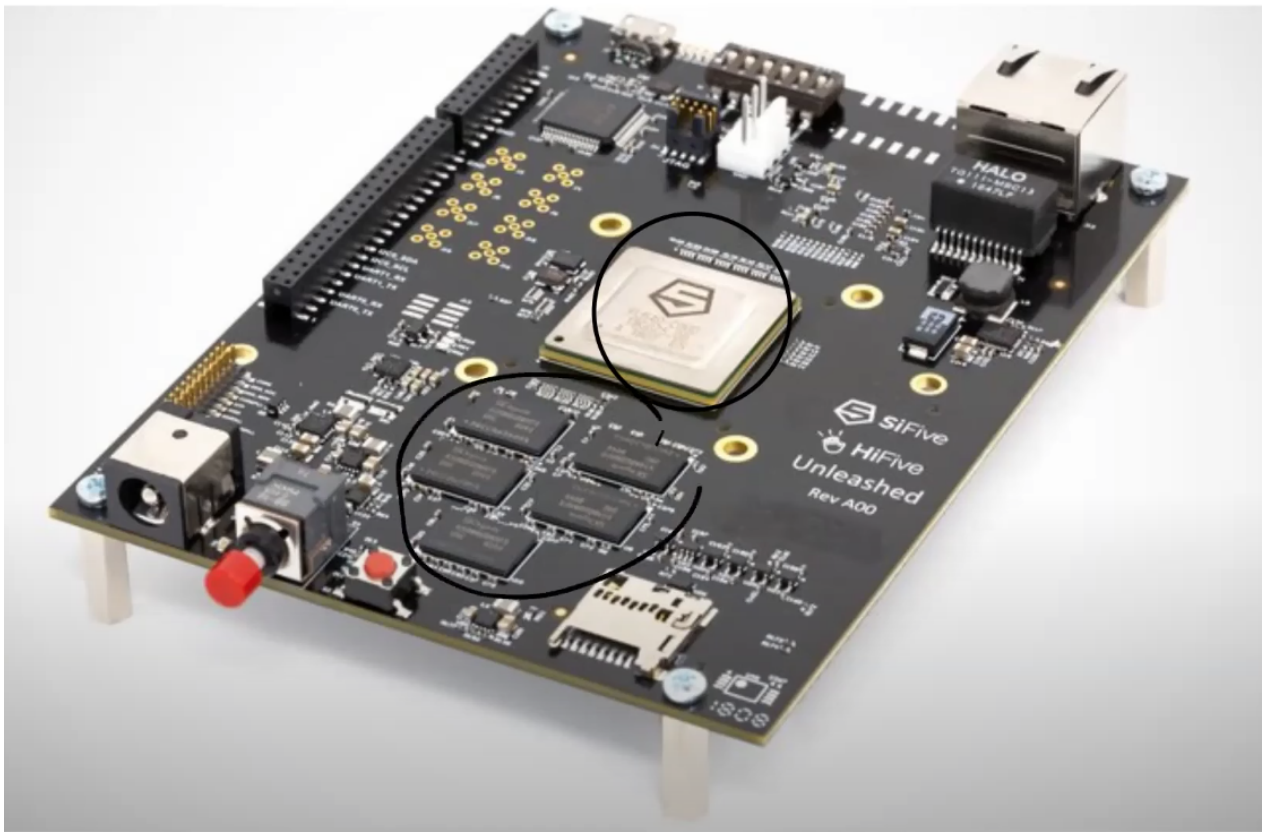
接下来，我们看一下在XV6中，page table是如何工作的？首先我们来看一下kernel page的分布。下图就是内核中地址的对应关系，左边是内核的虚拟地址空间，右边上半部分是物理内存或者说是DRAM，右边下半部分是I/O设备。接下来我会首先介绍右半部分，然后再介绍左半部分。



图中的右半部分的结构完全由硬件设计者决定。如你们上节课看到的一样，当操作系统启动时，会从地址0x80000000开始运行，这个地址其实也是由硬件设计者决定的。具体的来说，如果你们看一个主板，



中间是RISC-V处理器，我们现在知道了处理器中有4个核，每个核都有自己的MMU和TLB。处理器旁边就是DRAM芯片。



主板的设计人员决定了，在完成了虚拟到物理地址的翻译之后，如果得到的物理地址大于0x80000000会走向DRAM芯片，如果得到的物理地址低于0x80000000会走向不同的I/O设备。这是由这个主板的设计人员决定的物理结构。如果你想要查看这里的物理结构，你可以阅读主板的手册，手册中会一一介绍物理地址对应关系。

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_00FF		Reserved	Debug Address Space
0x0000_0100	0x0000_0FFF	RWX A	Debug	
0x0000_1000	0x0000_1FFF	R X	Mode Select	On-Chip Peripherals
0x0000_2000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_7FFF	R X	Mask ROM (32 KiB)	
0x0001_8000	0x00FF_FFFF		Reserved	
0x0100_0000	0x0100_1FFF	RWX A	E51 DTIM (8 KiB)	
0x0100_2000	0x017F_FFFF		Reserved	
0x0180_0000	0x0180_1FFF	RWX A	E51 Hart 0 ITIM (8 KiB)	
0x0180_2000	0x0180_7FFF		Reserved	
0x0180_8000	0x0180_EFFF	RWX A	U54 Hart 1 ITIM (28 KiB)	
0x0180_F000	0x0180_FFFF		Reserved	
0x0181_0000	0x0181_6FFF	RWX A	U54 Hart 2 ITIM (28 KiB)	
0x0181_7000	0x0181_7FFF		Reserved	
0x0181_8000	0x0181_EFFF	RWX A	U54 Hart 3 ITIM (28 KiB)	
0x0181_F000	0x0181_FFFF		Reserved	
0x0182_0000	0x0182_6FFF	RWX A	U54 Hart 4 ITIM (28 KiB)	
0x0182_7000	0x01FF_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	
0x0201_0000	0x0201_0FFF	RW A	Cache Controller	
0x0201_1000	0x0201_FFFF		Reserved	
0x0202_0000	0x0202_0FFF	RW A	MSI	
0x0202_1000	0x02FF_FFFF		Reserved	
0x0300_0000	0x030F_FFFF	RW A	DMA Controller	
0x0310_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x09FF_FFFF	RWX A	L2 LIM (32 MiB)	
0x0A00_0000	0x0BFF_FFFF	RWXCA	L2 Zero device	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	PRCI	
0x1000_1000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	UART 0	
0x1001_1000	0x1001_1FFF	RW A	UART 1	
0x1001_2000	0x1001_FFFF		Reserved	
0x1002_0000	0x1002_0FFF	RW A	PWM 0	
0x1002_1000	0x1002_1FFF	RW A	PWM 1	

Table 6: FU540-C000 Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics

Base	Top	Attr.	Description	Notes
0x1002_2000	0x1002_FFFF		Reserved	
0x1003_0000	0x1003_0FFF	RW A	I2C	
0x1003_1000	0x1003_FFFF		Reserved	
0x1004_0000	0x1004_0FFF	RW A	QSPI 0	
0x1004_1000	0x1004_1FFF	RW A	QSPI 1	
0x1004_2000	0x1004_FFFF		Reserved	
0x1005_0000	0x1005_0FFF	RW A	QSPI 2	
0x1005_1000	0x1005_FFFF		Reserved	
0x1006_0000	0x1006_0FFF	RW A	GPIO	
0x1006_1000	0x1006_FFFF		Reserved	
0x1007_0000	0x1007_0FFF	RW A	OTP	
0x1007_1000	0x1007_FFFF		Reserved	
0x1008_0000	0x1008_0FFF	RW A	Pin Control	
0x1008_1000	0x1008_FFFF		Reserved	
0x1009_0000	0x1009_1FFF	RW A	Ethernet MAC	
0x1009_2000	0x1009_FFFF		Reserved	
0x100A_0000	0x100A_0FFF	RW A	Ethernet Management	
0x100A_1000	0x100A_FFFF		Reserved	
0x100B_0000	0x100B_3FFF	RW A	DDR Control	
0x100B_4000	0x100B_FFFF		Reserved	
0x100C_0000	0x100C_3FFF	RW A	DDR Management	
0x100C_4000	0x17FF_FFFF		Reserved	
0x1800_0000	0x1FFF_FFFF	RW CA	Error Device	
0x2000_0000	0x2FFF_FFFF	R X A	QSPI 0 Flash (256 MiB)	Off-Chip Non-Volatile Memory
0x3000_0000	0x3FFF_FFFF	R X A	QSPI 1 Flash (256 MiB)	
0x4000_0000	0x5FFF_FFFF	RWX A	ChipLink (512 MiB)	ChipLink
0x6000_0000	0x7FFF_FFFF	RWXCA	ChipLink (512 MiB)	
0x8000_0000	0x1F_FFFF_FFFF	RWX A	DDR Memory (126 GiB)	Off-Chip Volatile Memory
0x20_0000_0000	0x2F_FFFF_FFFF	RWX A	ChipLink (64 GiB)	ChipLink
0x30_0000_0000	0x3F_FFFF_FFFF	RWXCA	ChipLink (64 GiB)	

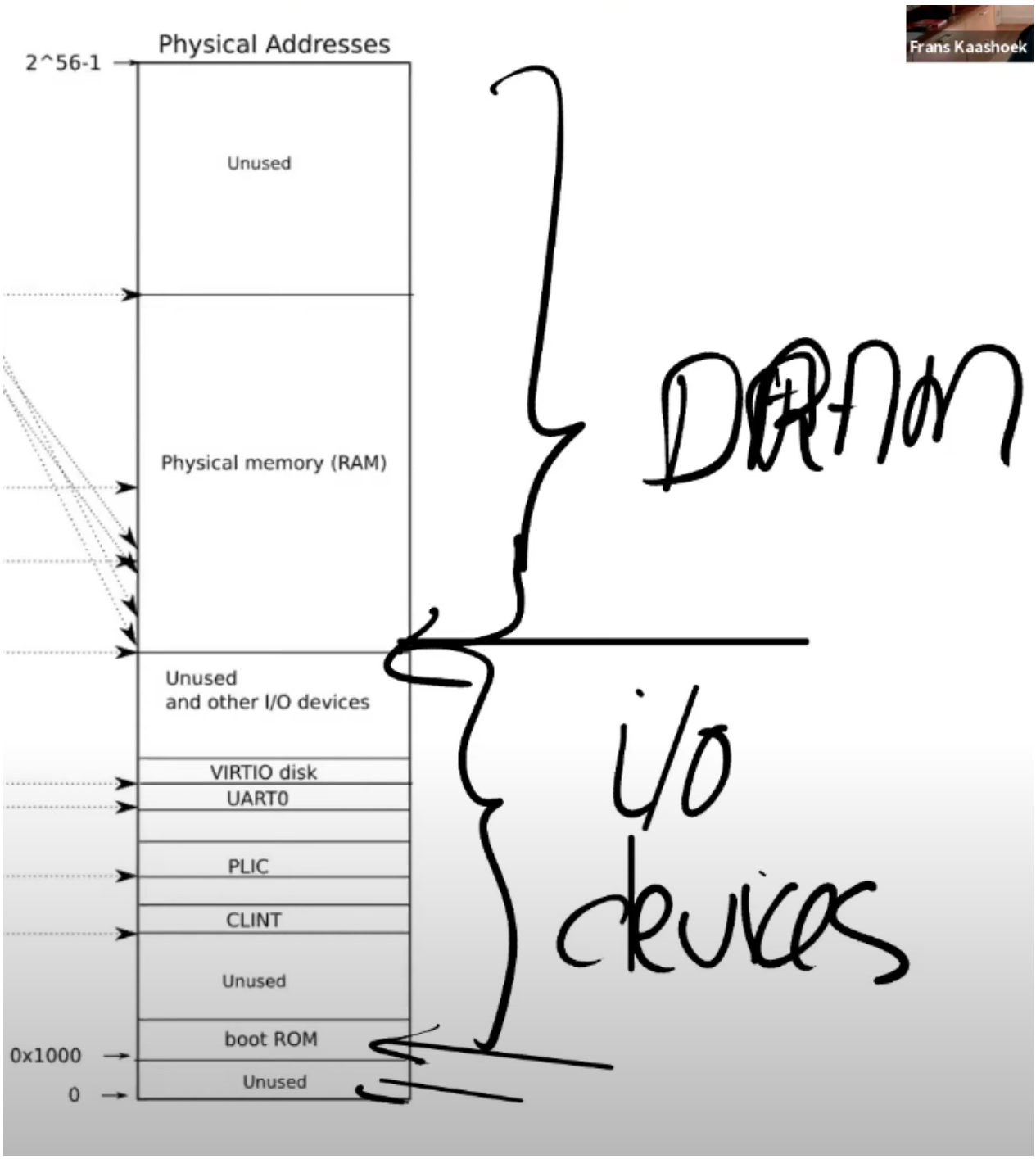
Table 6: FU540-C000 Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics

首先，地址0是保留的，地址0x10090000对应以太网，地址0x80000000对应DDR内存，处理器外的易失存储（Off-Chip Volatile Memory），也就是主板上的DRAM芯片。所以，在你们的脑海里应该要记住这张主板的图片，即使我们接下来会基于你们都知道的C语言程序---QEMU来做介绍，但是最终所有的事情都是由主板硬件决定的。

> 学生提问：当你说这里是由硬件决定的，硬件是特指CPU还是说CPU所在的主板？

> Frans教授：CPU所在的主板。CPU只是主板的一小部分，DRAM芯片位于处理器之外。是主板设计者将处理器，DRAM和许多I/O设备汇总在一起。对于一个操作系统来说，CPU只是一个部分，I/O设备同样也很重要。所以当你在写一个操作系统时，你需要同时处理CPU和I/O设备，比如你需要向互联网发送一个报文，操作系统需要调用网卡驱动和网卡来实际完成这个工作。

回到最初那张图的右侧：物理地址的分布。可以看到最下面是未被使用的地址，这与主板文档内容是一致的（地址为0）。地址0x1000是boot ROM的物理地址，当你对主板上电，主板做的第一件事情就是运行存储在boot ROM中的代码，当boot完成之后，会跳转到地址0x80000000，操作系统需要确保那个地址有一些数据能够接着启动操作系统。



这里还有一些其他的I/O设备：

- PLIC是中断控制器（Platform-Level Interrupt Controller）我们下周的课会

讲。

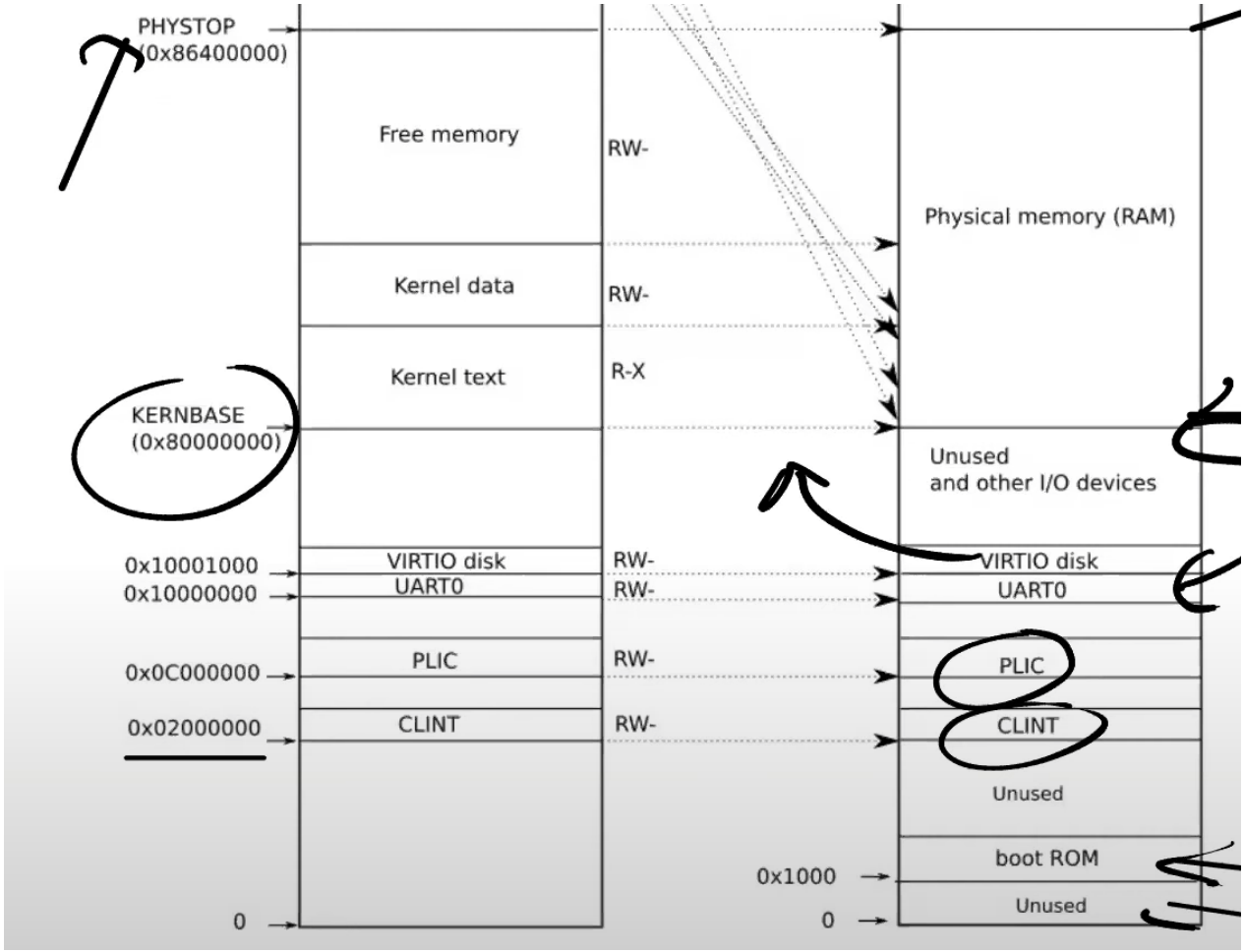
- CLINT（Core Local Interruptor）也是中断的一部分。所以多个设备都能产生中断，需要中断控制器来将这些中断路由到合适的处理函数。
- UART0（Universal Asynchronous Receiver/Transmitter）负责与Console和显示器交互。
- VIRTIO disk，与磁盘进行交互。

地址0x02000000对应CLINT，当你向这个地址执行读写指令，你是向实现了CLINT的芯片执行读写。这里你可以认为你直接在与设备交互，而不是读写物理内存。

> 学生提问：确认一下，低于0x80000000的物理地址，不存在于DRAM中，当我们在使用这些地址的时候，指令会直接走向其他的硬件，对吗?
>
> Frans教授：是的。高于0x80000000的物理地址对应DRAM芯片，但是对于例如以太网接口，也有一个特定的低于0x80000000的物理地址，我们可以对这个叫做内存映射I/O（Memory-mapped I/O）的地址执行读写指令，来完成设备的操作。
>
> 学生提问：为什么物理地址最上面一大块标为未被使用?
>
> Frans教授：物理地址总共有2^56那么多，但是你不用在主板上接入那么多的内存。所以不论主板上有多少DRAM芯片，总是会有一部分物理地址没有被用到。实际上在XV6中，我们限制了内存的大小是128MB。
>
> 学生提问：当读指令从CPU发出后，它是怎么路由到正确的I/O设备的？比如说，当CPU要发出指令时，它可以发现现在地址是低于0x80000000，但是它怎么将指令送到正确的I/O设备?
>
> Frans教授：你可以认为在RISC-V中有一个多路输出选择器（demultiplexer）。

接下来我会切换到第一张图的左边，这就是XV6的虚拟内存地址空间。当机器刚刚启动时，还没有可用的page，XV6操作系统会设置好内核使用的虚拟地址空间，也就是这张图左边的地址分布。

因为我们想让XV6尽可能的简单易懂，所以这里的虚拟地址到物理地址的映射，大部分是相等的关系。比如说内核会按照这种方式设置page table，虚拟地址0x02000000对应物理地址0x02000000。这意味着左侧低于PHYSTOP的虚拟地址，与右侧使用的物理地址是一样的。

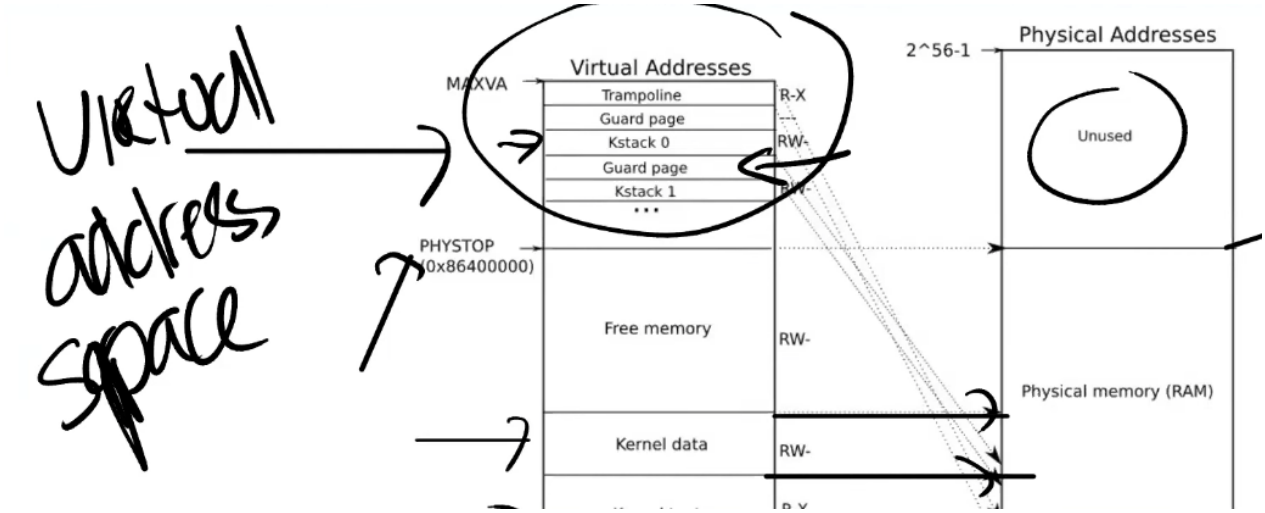


所以，这里的箭头都是水平的，因为这里是完全相等的映射。

除此之外，这里还有两件重要的事情：

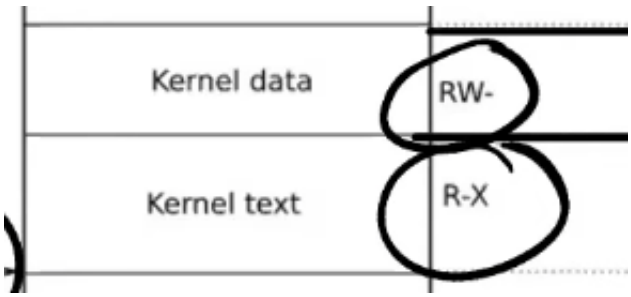
第一件事情是，有一些page在虚拟内存中的地址很靠后，比如kernel stack在虚拟内存中的地址就很靠后。这是因为在它之下有一个未被映射的Guard page，这个Guard page对应的PTE的Valid 标志位没有设置，这样，如果kernel stack耗尽了，它会溢出到Guard page，但是因为Guard page的PTE中Valid标志位未设置，会导致立即触发page fault，这样的结果好过内存越界之后造成的数据混乱。立即触发一个panic（也就是page fault），你就知道kernel stack出错了。同时我们也又不想浪费物理内存给Guard page，所以Guard page不会映射到任何物理内存，它只是占据了虚拟地址空间的一段靠后的地址。

同时，kernel stack被映射了两次，在靠后的虚拟地址映射了一次，在PHYSTOP下的Kernel data中又映射了一次，但是实际使用的时候用的是上面的部分，因为有Guard page会更加安全。

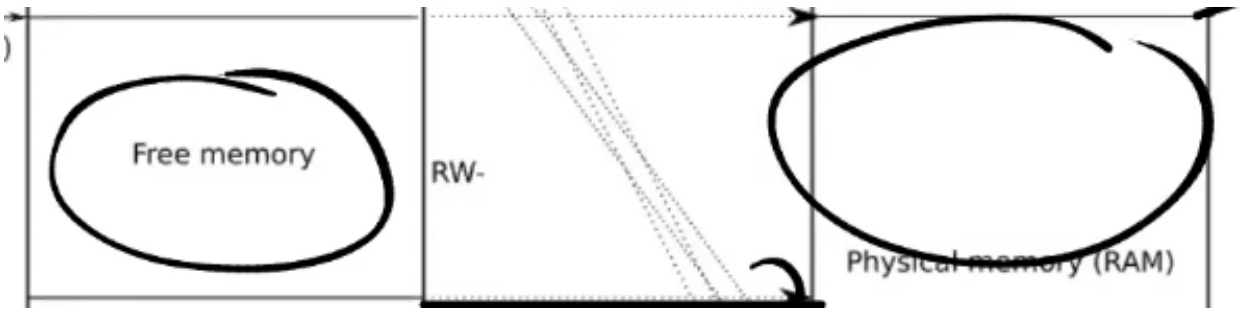


这是众多你可以通过page table实现的有意思的事情之一。你可以向同一个物理地址映射两个虚拟地址，你可以不将一个虚拟地址映射到物理地址。可以是一对一的映射，一对多映射，多对一映射。XV6至少在1-2个地方用到类似的技巧。这的kernel stack和Guard page就是XV6基于page table使用的有趣技巧的一个例子。

第二件事情是权限。例如Kernel text page被标位R-X，意味着你可以读它，也可以在这个地址段执行指令，但是你不能向Kernel text写数据。通过设置权限我们可以尽早的发现Bug从而避免Bug。对于Kernel data需要能被写入，所以它的标志位是RW-，但是你不能在这个地址段运行指令，所以它的X标志位未被设置。（注，所以，kernel text用来存代码，代码可以读，可以运行，但是不能篡改，kernel data用来存数据，数据可以读写，但是不能通过数据伪装代码在kernel中运行）



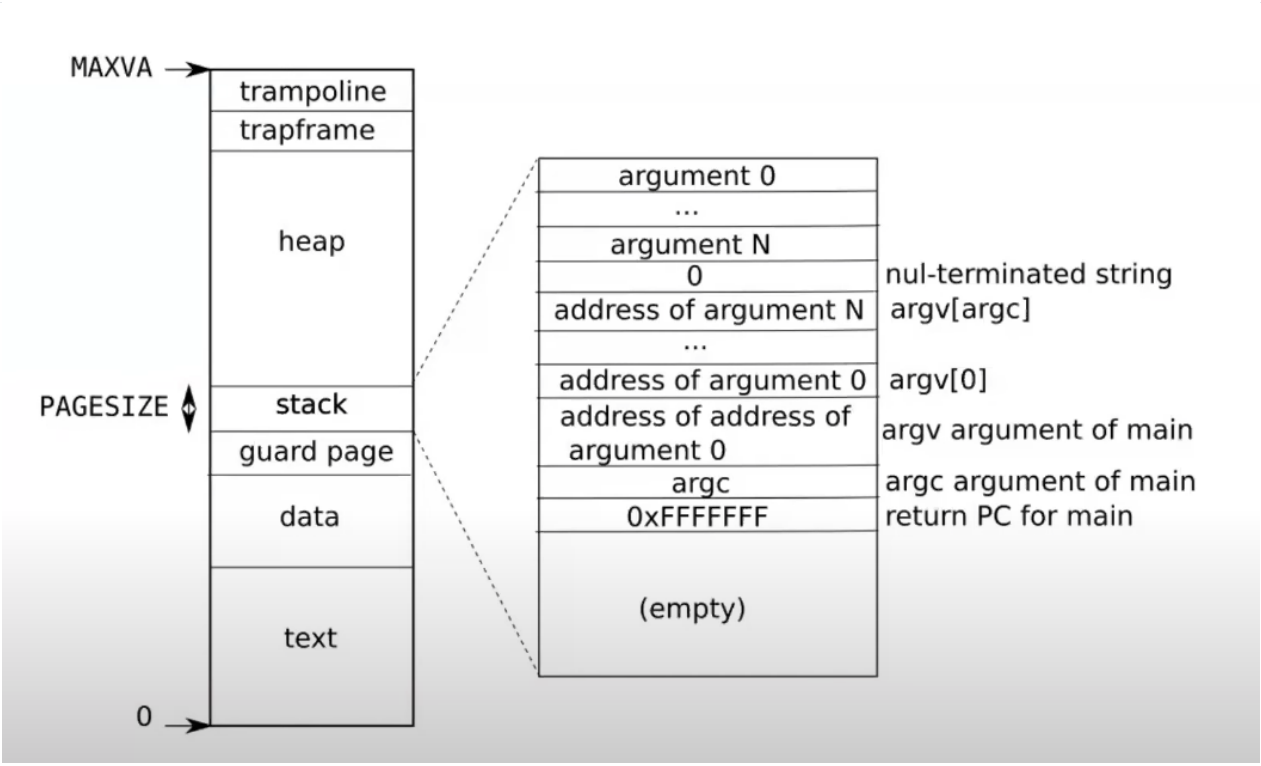
> 学生提问：对于不同的进程会有不同的kernel stack吗?
>
> Frans：答案是的。每一个用户进程都有一个对应的kernel stack
>
> 学生提问：用户程序的虚拟内存会映射到未使用的物理地址空间吗?
>
> Frans教授：在kernel page table中，有一段Free Memory，它对应了物理内存中的一段地址。



> XV6使用这段free memory来存放用户进程的page table，text和data。如果我们运行了非常多的用户进程，某个时间点我们会耗尽这段内存，这个时候fork或者exec会返回错误。
>
> 同一个学生提问：这就意味着，用户进程的虚拟地址空间会比内核的虚拟地址空间小的多，是吗?
>
> Frans教授：本质上来说，两边的虚拟地址空间大小是一样的。但是用户进程的虚拟地址空间使用率会更低。
>
> 学生提问：如果多个进程都将内存映射到了同一个物理位置，这里会优化合并到同一个地址吗?
>
> Frans教授：XV6不会做这样的事情，但是page table实验中有一部分就是做这个事情。真正的操作系统会做这样的工作。当你们完成了page table实验，你们就会对这些内容更加了解。

（以下问答来自课程结束部分，以为内容相关就移过来了）

> 学生提问：每个进程都会有自己的3级树状page table，通过这个page table将虚拟地址翻译成物理地址。所以看起来当我们将内核虚拟地址翻译成物理地址时，我们并不需要kernel的page table，因为进程会使用自己的树状page table并完成地址翻译（注，不太理解这个问题点在哪）。
>
> Frans教授：当kernel创建了一个进程，针对这个进程的page table也会从Free memory中分配出来。内核会为用户进程的page table分配几个page，并填入PTE。在某个时间点，当内核运行了这个进程，内核会将进程的根page table的地址加载到SATP中。从那个时间点开始，处理器会使用内核为那个进程构建的虚拟地址空间。
>
> 同一个学生提问：所以内核为进程放弃了一些自己的内存，但是进程的虚拟地址空间理论上与内核的虚拟地址空间一样大，虽然实际中肯定不会这么大。
>
> Frans教授：是的，下图是用户进程的虚拟地址空间分布，与内核地址空间一样，它也是从0到MAXVA。



> 它有由内核设置好的，专属于进程的page table来完成地址翻译。

> 学生提问：但是我们不能将所有的MAXVA地址都使用吧?
>
> Frans教授：是的我们不能，这样我们会耗尽内存。大多数的进程使用的内存都远远小于虚拟地址空间。