

Figure1:xv6中的虚拟内存布局(图中maxva应该使用大写）

作业逻辑：作业假设user会在访问某个文件前，先使用mmap（），然后之后在访问这个文件，产生page-fault（此时vma中已经有了对应的信息了），然后我们handle page-fault

作业逻辑：vma只是在虚拟内核内存中的一个表示用户虚拟内存地址[address,address+length]的信息的数据结构，并非是真的在在[address,address+length]上。

作业逻辑：mapfile，只能上传pagesize这么多数据量到物理内存mem上，所以如果只能发挥mappage的一个循环大的功能，不过即使mapfile一次能上传多倍pagesize（必须是pagessize的整数倍，这是使用mmap（）上传数据的要求！），mappage的逻辑是把地址x先找到他的下面最近的一个pagesize整数倍a，然后用这个数字作为pagetable的pte序号，写pte，挂钩的physical mem是mem。然后下一个写的是pte的虚拟地址是a+pagesize，对应的物理内存是mem+pagesize。也就是说，即使mapfile被改成允许一次能上传多倍pagesize，那么由于mappage函数的特性，pagetable里第一次是在4096t行上写，第二次在4096（t+1）行上写，而做不到第一次是在4096t行 上写，第二次是在4096（t+4）行上写，但是这样也有好处，就是够细致。我要是想要pagetable中4096（t+4）行也能够写东西，那么就用四次mappage。（现在假设是mapfile被改成允许一次能上传多倍pagesize，自然也能够一次上传4倍pagesize）。当然因为实际上mapfile一次只能上传一倍的pagesize也就是说，所以我要连续使用四次的mapfile+mappage。其实一开始早期用户使用mapp（）时其实是可以指定上传数据是多倍的pagesize的，这个特点也被继承到了vma中。用户可以如果想上传一个大文件里的很大部分，可以指定mapp的参数size是很大倍pagesize。但是之后用户触发pagefault的，也就是这个和形成pagetable有关的阶段，用户访问的不一定是pagesize整数倍，因为因为用户想要访问的有可能是虚拟地址中比如是pagesize整数倍多x的地址，这个和现实有关。我用户访问虚拟内存又不遵纪守法。那怎么搞才能让用户使用一个整数倍papgesize的虚拟内存的时候，映射过去的其实是整数倍pagesize+x对应的数据呢？  
是这样的，本来什么都不特别设计的情况是，整数倍pagesize地址对应的是物理内存中的地址mem，那么我用户访问整数倍pagesize+x，就是想访问mem+x上的数据，但是我的pagetable中的整数倍pagesize+x行中没有对应的pte，咋办？只有整数倍pagesize行有pte，那咋办，那就要利用lazy-loading特点，这时候其实电脑还没有上传数据到mem处，那么我干脆上传本来上传到mem+x地方的数据到mem，欸，这样用户用第整数倍pagesize行的pte去到mem，拿到的就是它想要的数据，这就是lazy-loading 的好处。普通的read不行啊，普通的read它如果用户访问的这个虚拟地址对应的pagetable中没有东西，那么你就没办法，除非正好对应的pagetable中有pte，欸那你可以，但是这个流程是得先有的pagetable用户再正好想要访问的地址是pagetable上对应的有pte的地址，这概率很小，但是mapp就不一样了，你用户可以先mapp一下，然后无论之后像访问的地址是不是正好对应的pagetable的行里面有pte，都可以被最近的整数倍的pagesize的虚拟地址的pte记录给见机行事。

作业疑问：只读file中的一页吗？那我想读很多页怎么办？

就只用offset行吗，不用(va-vma->addr)+offset？

mmap 和 munmap 确实可以用于 进程间通信，但它们并不是唯一的方法。进程间通信（IPC）在 UNIX 系统中有多种实现方式，包括：

管道（pipe）

消息队列（message queue）

共享内存（shared memory）使用共享内存（如 shmget、shmat）也是一种常见的方式，它会显式地将同一物理内存区域映射到多个进程的虚拟地址空间

信号（signals）

套接字（sockets)

**用户的虚拟地址空间与物理内存的映射 由用户进程的页表管理，那么这个页表所在是在虚拟用户空间中还是虚拟内核空间中，如果在虚拟内核中间中，是和内核页表放在一起的吗**

ChatGPT：

· 虚拟内核空间是内核在每个进程的虚拟地址空间中保留的一部分地址范围，用于内核代码和数据的访问。 这部分虚拟地址空间在所有进程中是相同的，但并不意味着页表本身存储在虚拟内核空间中。

**Physical Storage**: Page tables are data structures stored in **physical memory**, not in any virtual address space (neither user nor kernel virtual address space).

**Managed by the OS**: The operating system's kernel is responsible for creating, managing, and modifying page tables.

**Processes Do Not Access Page Tables Directly:** When a process accesses memory, it uses **virtual addresses**. The translation from virtual addresses to physical addresses is handled automatically by the CPU's **Memory Management Unit (MMU)** using the page tables. User processes do **not** access or modify the page tables directly. They are unaware of the page tables' existence and functionality.

**The Operating System Accesses Page Tables:** The OS kernel operates with higher privileges, allowing it to access and manipulate hardware resources and **physical memory directly**, including page tables. The kernel can access the page tables using **physical addresses**. It does not need the page tables to be mapped into any virtual address space to access them.

**Process Execution**: A process issues a memory access using a virtual address.

**MMU Translation**: The MMU uses the page table entries to translate the virtual address to the corresponding physical address. This translation is done by hardware and is transparent to the process.

**Role of the Kernel**: **Managing Mappings**: The kernel sets up and modifies the page tables when necessary (e.g., when loading a program, allocating memory, handling page faults).**No Virtual Dependency**: Since the kernel can access physical memory directly, it doesn't require the page tables to be in virtual memory.

**Why Page Tables Are Not in Virtual Address Space:** If page tables were in virtual memory, accessing them would require a translation from virtual to physical addresses.

**用户虚拟空间是某个特定进程相关的空间，不能独立于进程存在，但是内核虚拟空间被所有进程共享，并且独立于进程存在，我说的对吗**

ChatGPT

* ChatGPT 认可了用户的观点，并详细阐述了用户虚拟地址空间和内核虚拟地址空间的区别。
* 解释了用户虚拟地址空间是每个进程独立的，内核虚拟地址空间是所有进程共享的。

**分析：**

* **正确性：** ChatGPT 的回答是正确的。用户虚拟地址空间是进程特有的，每个进程有自己的虚拟地址空间，彼此隔离。内核虚拟地址空间是全局共享的，所有进程在内核态下共享同一套内核代码和数据。

**虚拟内核空间里的每个地址是否都有物理内存的对应地址？所有进程共享相同的虚拟内核空间，那是不是说明任意进程的虚拟内核空间都是由物理内核空间为模板，操作系统拷贝了物理内核空间，派发给这个进程当作这个进程的虚拟内核空间？**

虚拟内核地址空间 的每个地址并不一定都有物理内存的对应地址，只有通过页表的映射，虚拟内核地址才会映射到物理内存中的一个页面。

**内核地址空间共享：** 在 xv6 中，每个进程都有自己的页表，但内核部分的映射是相同的。这意味着每个进程的页表中，内核空间的映射部分是相同的，指向相同的物理内存。

**关于“拷贝”内核空间：** 当创建新的进程页表时，操作系统会将内核空间的映射添加到进程的页表中。这并不是“拷贝”物理内存，而是复制了页表中的内核映射部分，使得所有进程的内核空间映射一致。

· **纠正：**

**页表的组织：** 在 xv6 中，进程的页表包括用户空间和内核空间的映射。虽然每个进程的页表是独立的，但内核空间的部分在所有进程中是相同的映射。

**内核页表：** 并不存在一个全局的“内核页表”供所有进程使用，而是每个进程的页表中都包含了相同的内核空间映射。

1. **那xv6中，某个进程的struct proc结构体是存放在该进程专属的虚拟用户空间，还是存放在所有进程共享的虚拟内核空间？**
2. **mmap能够直接把文件从磁盘加载到进程的用户物理地址上，然后映射到进程的用户虚拟地址上，而不用经过进程的内核物理地址这一中间站点，这就是mmap比read优越的地方，我理解的对吗**
3. **这段代码描述的是xv6中的物理内存的layout对吗？**

**// Physical memory layout**

**// qemu -machine virt is set up like this,// based on qemu's hw/riscv/virt.c:**

**// 00001000 -- boot ROM, provided by qemu**

**// 02000000 -- CLINT**

**// 0C000000 -- PLIC**

**// 10000000 -- uart0**

**// 10001000 -- virtio disk**

**// 80000000 -- boot ROM jumps here in machine mode//-kernel loads the kernel here**

**// unused RAM after 80000000.**

**// the kernel uses physical memory thus:**

**// 80000000 -- entry.S, then kernel text and data**

**// end -- start of kernel page allocation area**

**// PHYSTOP -- end RAM used by the kernel**

1. **在这个layout中，特定进程的用户虚拟空间所对应的物理内存在哪里？**

ChatGPT

您提出的问题涉及到 xv6 中进程、内存映射以及物理内存布局的多个方面。我将逐个问题进行解答，并帮助您澄清相关的概念。

1. struct proc 结构体存放在哪里？

在 xv6 中，struct proc 结构体包含了一个进程的元数据，如进程的状态、内存信息、页表指针等。对于一个特定进程而言，struct proc 是存放在 内核虚拟空间 中，而不是进程的用户虚拟空间。这是因为 struct proc 是内核用来管理进程的结构，进程调度和管理等操作都由内核进行。

具体来说，进程的 struct proc 通常是存在内核内存区域的一部分，每个进程会有一个 struct proc 对象，内核通过调度器来访问和操作这些对象。虽然每个进程有自己的 用户虚拟空间，但 struct proc 结构体是在内核的地址空间中存储的，内核通过对这些结构的访问来管理进程状态。

总结：struct proc 结构体存储在 内核空间，而不是进程的用户虚拟地址空间。

2. mmap 和 read 的区别

· **共同点**：

· **mmap 和 read 都使用了内核缓冲区（buffer cache）**。文件的数据都会先被加载到 buffer cache 中。

· **区别**：

· **read**：

* + 数据从 buffer cache **复制**到用户态缓冲区。
  + 这涉及一次从内核空间到用户空间的内存拷贝。
* **mmap**：
  + 通过页表机制，直接将 buffer cache 中的物理页映射到用户虚拟地址空间。
  + **避免了数据从内核空间到用户空间的显式拷贝**。

· **mmap 的优势**：

· 省略了数据拷贝的步骤，减少了内存占用和 CPU 开销。

* 提高了 I/O 操作的效率，特别是对于大文件或需要随机访问文件的情况。

· **关键澄清**：

· **mmap 并未绕过内核缓冲区**。相反，它依赖于 buffer cache，将其内容映射到用户空间。

* 之前的误解在于认为 mmap 直接从磁盘映射数据，而不经过内核空间。

关于 read 系统调用：

用户态缓冲区的分配和映射：

当一个进程在用户空间分配缓冲区（例如使用 malloc 或在栈上分配），操作系统会为该缓冲区分配虚拟地址空间。

这些虚拟地址在分配时并不一定立即映射到物理内存。只有当这些地址被实际访问时，才可能触发缺页异常，进而由操作系统分配物理内存并更新页表。

页表的建立：

在缓冲区被分配时，页表条目可能尚未完全建立，但地址空间已经被预留。

当 read 调用尝试将数据写入用户缓冲区时，如果对应的物理内存尚未分配，会触发缺页异常，操作系统这时会为这些页分配物理内存并更新页表。

数据拷贝：

read 系统调用从内核的缓冲区（buffer cache）将数据复制到用户缓冲区。

这个过程使用用户提供的虚拟地址，操作系统通过页表将这些虚拟地址翻译为物理地址，以完成数据复制。

关键点：

read 不会在数据复制后再去修改页表或添加新的页表条目。页表的建立和更新是在内存分配和页面访问时完成的，而不是在 read 的数据复制过程中。

关于 mmap 系统调用：

映射过程：

mmap 直接在进程的虚拟地址空间中映射文件，建立虚拟地址到物理内存（实际上是内核缓冲区中的页）的映射。

这需要操作系统在页表中为相应的虚拟地址范围创建新的页表条目。

数据访问：

进程可以直接通过这些虚拟地址访问文件内容，无需额外的数据拷贝。

对您的观点的澄清：

您提到在 read 的过程中，需要在数据复制后修改页表，这是一个误解。

页表的建立和更新与内存的分配和访问有关，而不是与数据的复制过程直接关联。

在 read 过程中，数据复制是通过已有的（或在需要时由缺页异常处理程序创建的）页表映射完成的。

3. 物理内存的布局

这段代码描述的是 xv6 中的物理内存布局，以及在 QEMU 模拟环境 中如何初始化物理内存。以下是对这段布局的解释：

// Physical memory layout

// qemu -machine virt is set up like this,// based on qemu's hw/riscv/virt.c:

// 00001000 -- boot ROM, provided by qemu

// 02000000 -- CLINT

// 0C000000 -- PLIC

// 10000000 -- uart0

// 10001000 -- virtio disk

// 80000000 -- boot ROM jumps here in machine mode//-kernel loads the kernel here

// unused RAM after 80000000.

// the kernel uses physical memory thus:

// 80000000 -- entry.S, then kernel text and data

// end -- start of kernel page allocation area

// PHYSTOP -- end RAM used by the kernel

解释:

Boot ROM: 地址 0x00001000 起始的是 QEMU 提供的启动 ROM，这是 QEMU 模拟的固件，负责加载操作系统。

CLINT (0x02000000): 这是 Clint（Core Local Interruptor）区域，用于中断管理。

PLIC (0x0C000000): 这是 PLIC（Platform-Level Interrupt Controller）区域，用于处理平台级中断。

UART (0x10000000): 这是 UART（通用异步收发传输器），用于串行通信。

Virtio Disk (0x10001000): 这是 Virtio 磁盘 设备区域，提供虚拟磁盘设备的访问。

内核加载区域 (0x80000000): 从 0x80000000 开始，内核被加载到这个区域，内核的代码和数据段会存放在这里。

内核的物理内存使用:

从 0x80000000 开始，内核的 entry.S、内核文本段、数据段 会被加载到物理内存中。

内核的 页表区域 和其他资源会从 end 开始分配。

PHYSTOP 是内核使用的物理内存的结束位置，表示内核已经用完的物理内存的边界。

4. 特定进程的用户虚拟空间所对应的物理内存位置？

在 xv6 中，特定进程的 用户虚拟空间 会通过 页表映射 与物理内存进行关联。每个进程的 用户空间 是独立的，但是这些虚拟地址通过页表映射到物理内存中的实际页面。

特定进程的 用户虚拟空间 的物理内存位置并没有直接固定的映射，它取决于 进程的页表 如何设置。不同进程的 用户虚拟空间 在虚拟地址上是隔离的，但它们的 物理内存位置 可以是不同的，因为操作系统会根据内存的空闲情况和进程的需求动态地映射物理内存。

**vma.addr和vma ->addr都是合理的对吗**

ChatGPT

Yes, both vma.addr and vma->addr can be correct, but they are used in different contexts depending on whether vma is a structure or a pointer to a structure:

vma.addr:

This syntax is used when vma is a variable of type struct vma, meaning it’s a structure itself (not a pointer)

struct vma vma\_instance;

vma\_instance.addr = some\_address;

vma->addr:

This syntax is used when vma is a pointer to a struct vma, typically allocated dynamically or referencing an element in an array of struct vma.

Example:

struct vma \*vma\_ptr = &vma\_array[i]; // Pointing to an element in an array of vma

vma\_ptr->addr = some\_address;

**page-aligned的意思是每块数据的起始点是pagesize的整数倍，还是说数据的大小是pagesize的整数倍，还是都要**

page-aligned 的意思通常是指数据的起始地址是页面大小的整数倍。但是在mmap（）中，除了要求内存的起始地址是 pagesize 的整数倍，还可能要求数据的大小也必须是 pagesize 的整数倍

假设页面大小为 4096 字节（4 KB），并且需要将一个文件的一部分映射到内存中：

起始地址对齐：如果映射区域的起始地址

（映射区域的起始地址指的是进程的虚拟内存地址，而不是文件在磁盘上的物理地址，如果我们希望将文件内容从偏移量 1000 处开始映射到进程的虚拟地址空间中，那么我们可以调用 mmap，请求从文件偏移量 1000 处开始映射。但是，映射到的虚拟内存地址需要是 4096 字节对齐的。操作系统可能会将映射区域的虚拟内存起始地址对齐到最近的 4096 字节边界（例如 0x400000），并将文件从 1000 处的内容加载到这个虚拟内存地址中。这样，进程的虚拟内存起始地址是页面对齐的，而文件的实际数据位置则可以是非页面对齐的偏移（例如 1000 字节））

不对齐到 4096 字节，操作系统可能会拒绝映射请求，因为它无法在非对齐的地址上直接映射页面。

大小为 pagesize 的整数倍：如果映射区域的大小不是 pagesize 的整数倍，系统通常会将最后一页填充到页面边界。例如，若文件的大小是 5000 字节，则可能需要扩展到 8192 字节（2 个页面），操作系统会在最后一页的剩余部分填充零值。这样处理可以简化内存管理，使得每次映射都能完整地以页面为单位进行管理。