北京大学

虚存管理

XV6 源码阅读报告

组长： 罗 登 2001210364

组员1： 毕廷竹 2001210186

组员2： 周旭敏 2001210723

组员3： 罗旭坤 2001210368

目录

1. 初始化内存布局

> XV6 初始化之后到执行 main.c 时，内存布局是怎样的(其中已有哪些内容)?

下面先分析XV6的初始化流程，之后再据此给出初始化结束时的内存布局。

1. 要计算机启动进入XV6，需将其引导程序存放到硬盘的第一个扇区，也就是说，需要首先将 `bootasm.S` 的二进制代码存入0号扇区。
2. 机器启动时，读取并执行 `bootasm` 。这段汇编代码主要加载了临时的GDT、从实模式切换到保护模式、初始化部分寄存器，之后调用 `bootmain` 。
3. `bootmain` 负责将一页 `4KB` 的内容（从1号扇区开始，每个扇区容量 `512字节` ）读入内存。之后按照ELF文件的格式规定找到程序段的偏移 `e_phoff` ，再跟据程序段中规定的程序块偏移 `p_offset` 、文件占用空间 `p_filesz` 、内存占用空间 `p_memsz` 、物理内存装载地址 `p_paddr` 将内核程序装入内存。之后调用执行内核文件的入口代码（ `entry` ）。
4. 执行 `entry.S` 。
    1. 打开页大小扩展（设 `CR4` 寄存器的 `PSE` 位为 `1` ）。
    2. 将启动临时页目录（ `entrypgdir` ）设置到 `CR3` ，作为系统当前的分页系统使用的页目录。该页目录里只有两条预设的记录： （注，本文表格中的物理地址结束地址是实际精确物理地址-1，不包括该字节，即物理地址范围是[起始地址, 结束地址)）

编号	虚拟地址起始地址	物理地址起始地址	物理地址结束地址	权限	含义
1	0	0	0x400000	可写、内核访问	高位地址空间的内核页面
2	`KERNBASE` 0x80000000	0	0x400000	可写、内核访问	低位地址空间的内核页面

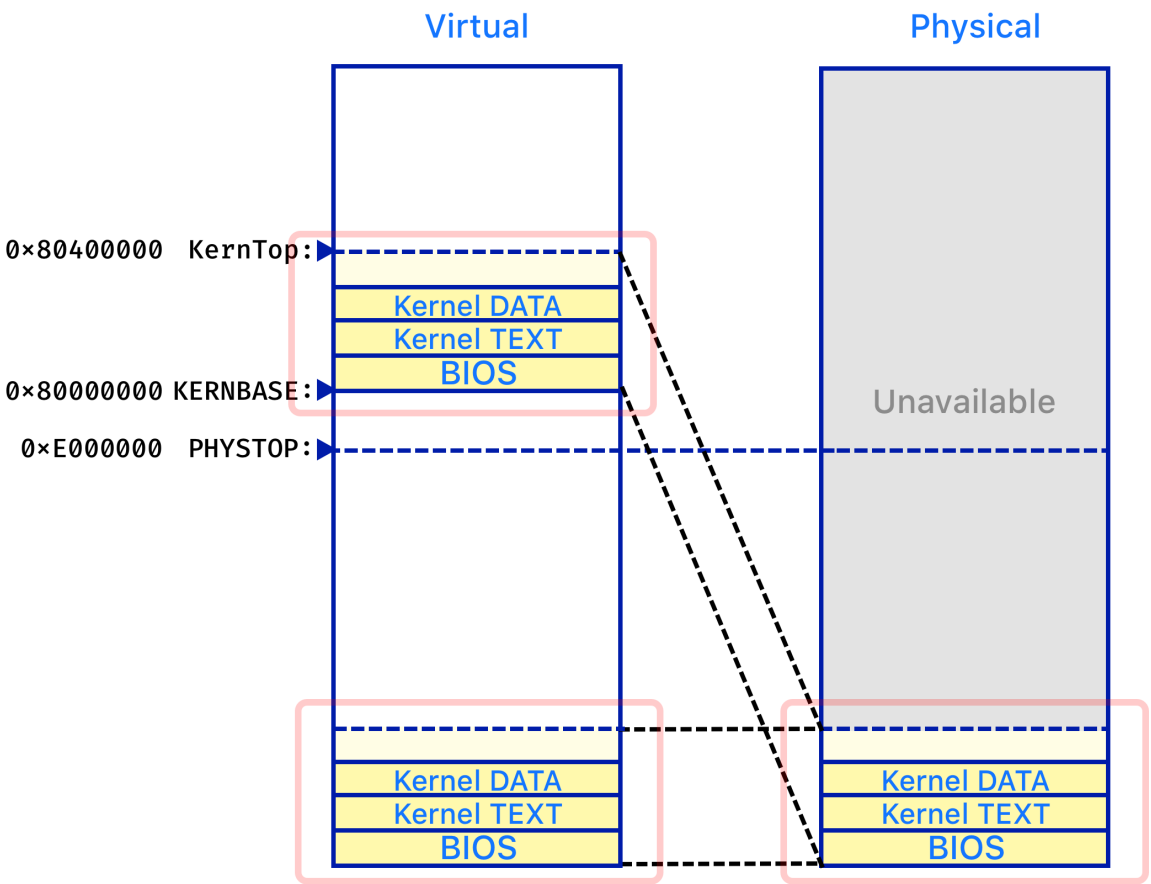
需要注意的是，这两条页目录记录都直接指向大小为4MB的内核页框；这两条页目录记录也都打开了 `PS` （页大小）开关，而无需再指向另一个页表。

3. 将 `CR0` 的 `PG` 开关打开，启用分页机制；将 `CR0` 的 `WP` 开关打开，启用写保护机制。
4. 设置大小为 `4K` 的栈的栈指针。
5. 跳转到 `main` 。

5. 执行 `main` 。之后的虚拟内存初始化情况在问题3中继续讨论。

综上分析，第3步中内核程序被读入内存放置到预定的物理地址，读取并提交了一个过渡性的页目录。不妨将内核页框的顶部地址 `0x80400000` 称为 `KernTop` ，将物理地址中无法访问的区域称为不可访问区（ `unavailable` ），则内存布局如图1所示（图中标红的区域为本阶段发生重要变化的内存空间）：

图1 初始化初期阶段 XV6内存布局



2. 动态内存管理

> XV6 的动态内存管理是如何完成的? 有一个 kmem(链表)，用于管理可分配的物理内存页。(vend=0x00400000，也就是可分 配的内存页最大为 4Mb)

XV6的动态内存管理逻辑主要在 `kalloc.c` 中实现。

从大的方面讲，操作系统的动态内存管理的主要工作包括维护一个只是当前空闲内存空间的数据结构、按需分配空闲空间里内存、按需释放指定内存空间三种主要操作。下面，我们依次分析XV6的具体实现方式。

- 首先，XV6定义了一个单链表的将空闲内存空间按页（一页大小为4KB）链接起来。通过递归声明的结构体 `run`，即实现了前后的链接，也巧妙地原地存储了下一个空闲页的地址。因为空闲页本身没有有效的内容，所以借用空页面中的4个字节来存储下一空闲页的地址是完全可行的。为了逻辑清晰，链表和锁放在一起构成了线程安全的空闲内存管理结构 `kmem`。
- 之后，系统初始化时以及相关资源释放时，会将新的空页面插入空闲链表之后，即添加进可用的空闲页面集合。具体的，系统初始化时，先执行 `kinit1` 将内核页框的剩余空闲空间切分为对其到4KB的小页框依次插入链表；之后系统再执行 `kinit2` 将可用物理空间中除内核页之外的剩余空间按页依次插入链表。经过两轮初始化，系统所有可用的内存空间都被以页为单位管理了起来。另外，XV6也支持将不再被使用的内存页释放，通过调用 `kfree`，将页面内容抹除，重新插入表头。
- 最后，XV6的内存动态分配通过 `kalloc` 实现。每当被要求分配一个空闲页面时，系统从空闲链表中取出表头返回作为分配的空闲页面，同时将该页面指向的下一页面作为新的表头。

3. 虚拟内存初始化 & 虚拟内存布局

> XV6 的虚拟内存是如何初始化的?画出 XV6 的虚拟内存布局图，请说出每一部分对应的内容是什么。 见 memlayout.h 和 vm.c 的 kmap 上的注释

虚拟内存初始化

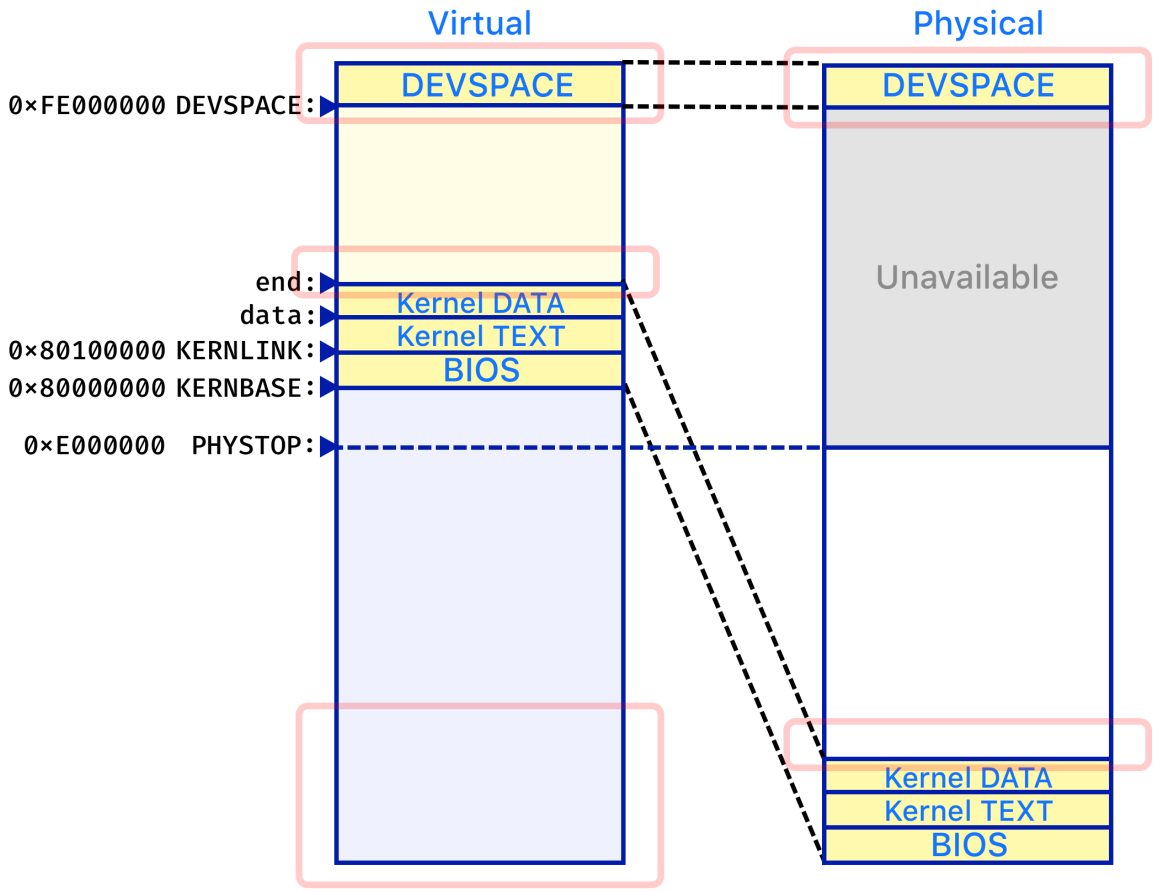
XV6的虚拟内存经历三个阶段完成初始化。

1. 第一阶段是在初始化初期阶段（ `entry` ），首次加载内核程序。内核被加载的到内存底部，通过设置 `entrypgdir` 为页目录，物理内存底部存有内核的空间整体作为一个大小为4MB的大页框被映射到虚拟内存中的底部和中部两处，此时内存布局情况见上文图1。
2. 第二阶段是在初始化中期阶段（ `main` ），启动运行内核程序。通过调用 `kvmalloc`，其中的 `setupkvm` 函数会创建一个新的页目录，以静态常量 `kmap` 中规定的映射规则将整个内存空间依次映射到高位虚拟地址空间（ `KERNBASE` 之上）。 `kmap` 中规定的映射关系如下表所示：

编号	虚拟地址起始地址	物理地址起始地址	物理地址结束地址	权限	含义
1	`KERNBASE` 0x80000000	0	`EXTMEM` 0x100000	可写、内核访问	IO空间
2	`KERNLINK` 0x80100000	`EXTMEM` 0x100000	`data` - `KERNBASE`	只读、内核访问	内核代码加只读数据
3	`data`	`data` - `KERNBASE`	`PHYSTOP` 0xE000000	可写、内核访问	内核数据+扩展可用内存
4	`DEVSPACE` 0xFE000000	`DEVSPACE` 0xFE000000	0 (即0xffffffff)	可写、内核访问	设备占用内存

其中， `data` 指内核镜像中代码和常量在虚拟内存高地址空间中的结束地址；图中 `end` 指内核镜像中变量在虚拟内存高地址空间中的结束地址。此时内存布局见图2。

图2 初始化中期阶段 XV6内存布局

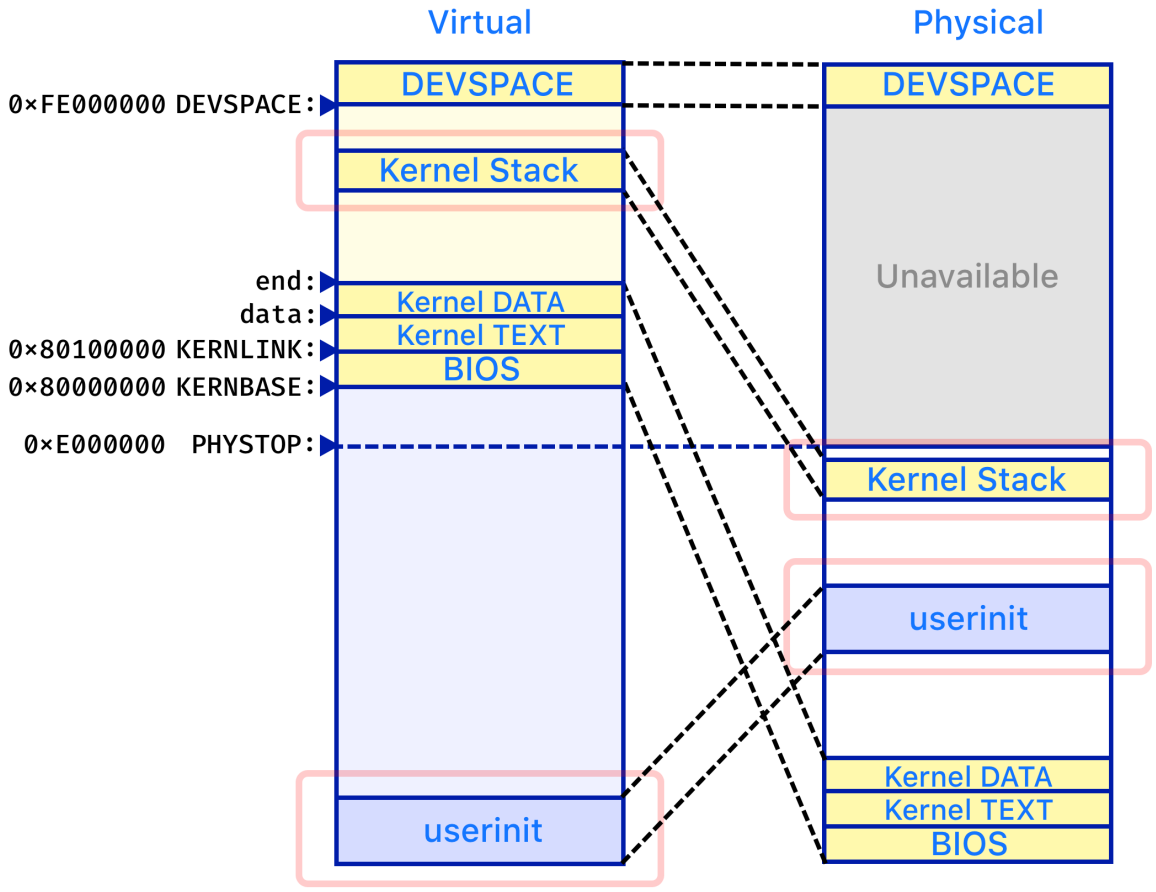


3. 第三阶段是初始化后期阶段（ `userinit` ），首次启动用户进程 `initcode` 。系统先通过调用 `setupkvm` 创建一个和上表完全一致的包含内核页目录，之后再通过 `inituvm` 建立程序ELF镜像中的代码与数据地址的虚拟地址映射。值得注意的是， `initcode` 在内存中位于内核代码区域，用户态下没有权限访问。故系统先分配了一个新页面，并设置其权限为 `PTE_W` 、 `PTE_U` （可写、用户可访问），再将 `initcode` 通过 `memmove` 拷贝一份到此页面中。最后， `inituvm` 中将虚拟内存最底部一页（0-0xfff）映射到新页面的物理地址：

编号	虚拟地址起始地址	物理地址起始地址	物理地址结束地址	权限	含义
1	0	`mem`	`mem` + 0x1000	可写、用户访问	用户初始化进程的代码

其中，`mem`指为代码分配的新页面的页框物理起始地址。此时内存布局见图3。

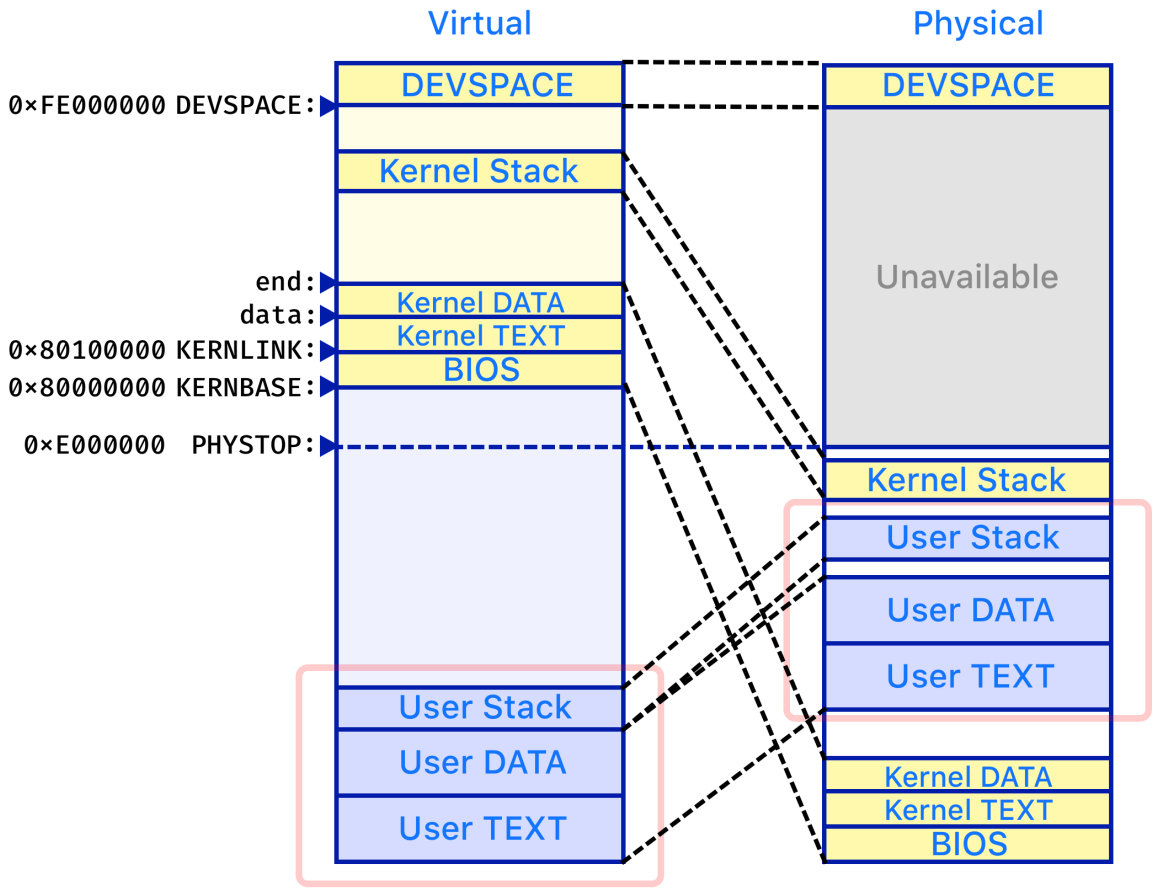
图3 初始化后期阶段 XV6内存布局



虚拟内存布局

当系统启动完成后，某用户进程通过`exec`载入运行时，系统的内存布局与初始化后期阶段基本一致，这时用户进程还拥有总大小两个页面的`用户栈`空间。相比如图4所示。

图4 运行用户进程时 XV6内存布局



4. 内存页式管理&映射

> 关于 XV6 的内存页式管理。发生中断时，用哪个页表?一个内页是多大?页目录有多少 项?页表有多少项?最大支持多大的内存?画出从虚拟地址到物理地址的转换图。在 XV6 中，是如何将虚拟地址与物理地址映射的(调用了哪些函数实现了哪些功能)?

页表大小与结构

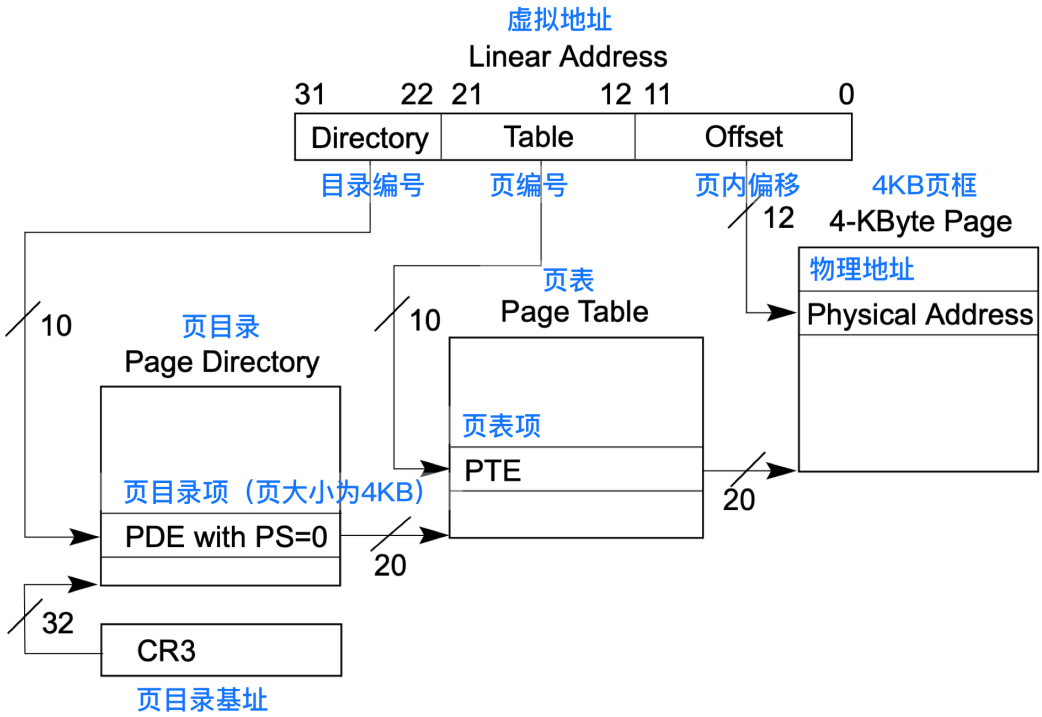
简要回答前3个问题：

- XV6在发生中断后切换到内核态之后，内核会借用页目录，即继续使用该进程在用户态时所设置的页目录。
- 一个内页大小为 `4KB` 。页目录含有2^{10}= `1024` 页目录项。仅当系统初始化初期时，在开启CPU页大小扩展的情况下（ `CR4_PSE` ），页目录里记录了两项4MB大空间页面；在系统启动完毕后，不再使用大页框，一律使用标准 `4KB` 页框。
- 最大支持2GB-16MB= `2032MB` 内存。 `DEVSPACE` （0xFE000000）之上会一直占用16MB的物理内存空间，XV6在 `setupkvm` 中设置页目录时会做判断，高位虚拟地址空间若高于 `DEVSPACE` ，系统会因为内存配置过大而中止。

地址转换

XV6的虚拟地址和物理地址都是 `32` 位的宽度，采用了页目录-页表的 `二级索引` 结构。虚拟地址到物理地址的转换图如图5所示：

图5 XV6虚拟地址到物理地址的转换图



地址映射

为了将虚拟地址和物理地址之间建立映射，XV6对于内核进程、用户进程分别以软件方式构建各个相关页表项。在配置好页表之后，提交给系统，CPU中内置的硬件机制便可自动执行地址转译的工作。

首先我们讨论XV6是如何构建页表项的，之后再分析内核进程、用户进程分别是在何种时机下执行构建和提交的。

XV6中虚拟内存相关逻辑的代码主要存在于 `vm.c` 中。该文件内部的 `mappages` 和 `walkpgdir` 是两个主要页表构建函数， `vm.c` 对外导出的函数就是对其的一层封装。

`walkpgdir` 用于在指定页目录中查找指定虚拟地址的页表项指针，并返回指向该页表项的指针。需要注意的是，该函数在alloc参数开启时，还会在页表不存在导致搜索没有命中情况下创建涉及到的页表。walkpgdir首先将虚拟地址右移 `22` 位再与 `0x3ff` 按位与（取出 `高10位` ），得到 `PDX` （page directory index，页目录下标），之后根据下标作为偏移找到页目录项（ `PDE` ）。通过判断 `PTE_P` 位（页目录项/页表项是否有效）是否为1确认页表是否存在。

- 若存在，则取 `PDE` 的前 `20` 位，再在之后拼上 `12` 位的0得到页表的基地址。

- 若不存在但允许创建（`alloc`等于1），则直接调用`kalloc`分配一个新页面作为页表，同时在页目录中把页表的起始地址前`20`位再加上`PTE_P`、`PTE_W`、`PTE_U`（有效、可写、用户态可访问）的标志位，新页面的地址也就是页表的基地址。
    - 若不存在且不允许创建，则返回`0`，表示出现错误。

有了页表的基地址，再将虚拟地址右移`12`位之后和`0x3ff`做与运算（取出`中间10位`），得到`PTX`（page table index，页表下标），之后根据下标作为偏移找到`PTE`（page table entry，页表项）的地址作为指针返回。

`mappages`负责将指定虚拟地址作为起始地址，将其之上虚拟地址批量地映射到指定地址开始的、指定总空间大小的物理地址空间。该函数主要是对`walkpgdir`做了封装，首先将起始虚拟地址下取整对齐到最近的页框，再以每次映射一页的大小做迭代。每次迭代中都会调用`walkpgdir`获取相关页表项的指针：

- 如果该页表项已经存在映射关系（`PTE_P`为1），则因为重复映射的错误中止系统。
- 如果该页表项尚未映射，将其地址设置为物理页框地址首地址，并设置对应的权限标志位、开启PTE_P有效标志位。

XV6中`kvm`（kernel virtual memory，内核虚拟内存，指高地址空间）和`uvm`（user virtual memory，用户虚拟内存，指低地址空间）相关的初始化和配置函数（如`inituvm`、`setupkvm`、`allocuvm`、`loaduvm`，见第3问中的分析）都通过调用`mappages`，创建所需的映射关系，实现了虚拟内存的初始化。

5. 相关问题补充与探究

1. 简要描述ELF文件格式，以及如何中读取镜像?

- `ELF`全称可执行和可链接格式（Executable and Linkable Format）。
- 32位二进制下头部（`ELF header`）大小为`52`字节，64位二进制下头部大小为64字节。
- 头部中的`e_phoff`指示程序头（`program header`）的起始偏移、`e_phnum`指示程序头的总个数。
- 程序头中：
    - `p_type`指示程序段的类型，`PT_LOAD`（即1）说明该段需要被载入内存。
    - `p_vaddr`/`p_paddr`指示程序段应被载入到的目标虚拟/物理内存起始地址。多数情况以虚拟地址为准，但XV6的内核初始化是直接加载到目标物理地址的。
    - `p_filesz`/`p_memsz`指示程序段占用的硬盘/内存空间。由于可能存在`BSS`（`Block Started by Symbol`，不占硬盘空间但仍需分配内存空间的程序变量数据段），内存占用空间总是大于等于硬盘占用空间。
- 遍历每个程序头，依次将需要载入内存的程序段载入程序头中预先规定的位置即完成了ELF的镜像读取。

2. 在内核状态下，可以不维护页表，直接通过运算操纵内存吗?

不可以。

虽然XV6内置了 `V2P` 和 `P2V` 宏定义，极大简化了虚拟内存中供内核使用的高地址空间与物理内存中低地址空间的转化流程，看似无需再调用 `kvmalloc` 为每个进程专门创建供内核使用的高地址空间的页表项映射。但同时需要注意的是，在IA32 CPU开启了分页之后，所有送入CPU供其执行的指令中，涉及地址的数据值，无论处在内核态还是用户态，都涉及到了硬件地址转译，视为虚拟地址。一旦缺失了映射关系，物理内存将 `不可达` 、 `无法访问` 。

除非在切换到内核态后， `禁用` 分页；在切换到用户态前，再 `启用` 分页。但如此做来相当复杂，不如用户态和内核态公用一张页目录，省去CPU状态配置的麻烦。

3. __attribute__ (__aligned__(PGSIZE))的作用是什么?

这个编译属性规定了他所修饰变量的 `最小对齐` 字节大小。例如，当 `PGSIZE` 为 `4096` 时，该结构体一定会被以最少 `4KB` 的空间大小进行对齐。

由于 `entrypgdir` 需要作为单独的一页传给 `CR3` 作为页目录使用，使用编译属性对齐后，便可保证该变量被刚好存放在一个页框内。

4. kinit1和kinit2有何区别，为何依次执行kinit1、kvmalloc、kinit2初始化?

`kinit1` 和 `kinit2` 分别负责将 `0` - `0x400000` 、 `0x400000` - `PHYSTOP` 的的内存区域以按页框大小划分，遍历链入空闲空间链表。 `kvmalloc` 负责创建虚拟内存中 `KERNBASE` - `KERNBASE + PHYSTOP` 到物理内存 `0` - `PHYSTOP` 专供内核访问的页目录和相关页表项。

值得注意的是， `kinit` 和 `kvmalloc` 之间有一个 `引导` （ `bootstrap` ）的问题：没有页表，内核无法访问相关地址空间，也就无法将空间按页链接；没有空闲链表，内核无法分配出空闲页，没有空闲页来放置页目录和页表，也就无法完成分页地址映射。

因此XV6将 `kinit` 拆解成作用局部空间和其他全局空间两步，将 `kvmalloc` 穿插于两者之间：

- 执行局部空间空闲页链接（ `kinit1` ）：由于过渡性页目录 `entrypgdir` 存在并起效，内核可以访问 `end` - `0x400000` 之间的物理地址空间。
- 执行全局分页地址映射（ `kvmalloc` ）：有限的空闲空间链表可以满足分配空页作为页目录和页表的需求。
- 执行全局空间空闲页链接（ `kinit2` ）：在前一步的全部页表映射创建和提交后，内核已经拥有了寻址访问物理内存 `0` - `PHYSTOP` 全局空间的能力。

此外， `kinit1` 初始化了空闲空间结构体的 `自旋锁` 但并未启用； `kinit2` 在多处理器初始化即将结束、调度器即将运行时启用了空闲空间结构体的锁，保证了 `并发` 环境下空闲空间链表的数据一致性，消除了数据竞争的风险。

5. 每个进程都有一个内核栈，还是公用一个内核栈?

每个用户进程在 `allocproc` 初始化进程结构体（ `PCB` ）时都会被分配一个空闲页面作为内核栈存储区域。内核栈的页面地址会被保存到进程结构体中并在进程切换时也随之切换。故每个进程都有一个独立的内核栈。

6. 参考链接

1. Executable and Linkable Format (ELF) http://www.skyfree.org/linux/references/ELF_Format.pdf
2. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1 https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html
3. Bach, Maurice J. *The design of the UNIX operating system*. Vol. 5. Englewood Cliffs: Prentice-Hall, 1986. http://160592857366.free.fr/joe/ebooks/ShareData/Design%20of%20the%20Unix%20Operating%20System%20By%20Maurice%20Bach.pdf
4. Type Attributes - Using the GNU Compiler Collection (GCC) https://gcc.gnu.org/onlinedocs/gcc-4.7.0/gcc/Type-Attributes.html
5. Global Descriptor Table - OSDev Wiki https://wiki.osdev.org/Global_Descriptor_Table
6. GDT Tutorial - OSDev Wiki https://wiki.osdev.org/GDT_Tutorial
7. ELF - OSDev Wiki https://wiki.osdev.org/ELF
8. Understanding Memory Layout. The memory refers to the computer... | by Shohei Yokoyama | Medium https://medium.com/@shoheiyokoyama/understanding-memory-layout-4ef452c2e709
9. 深入理解BSS(Block Started by Symbol) - veli - 博客园 https://www.cnblogs.com/idorax/p/6400210.html
10. memory - Do modern OS's use paging and segmentation? - Stack Overflow https://stackoverflow.com/questions/24358105/do-modern-oss-use-paging-and-segmentation
11. Memory Layout of C Programs - GeeksforGeeks https://www.geeksforgeeks.org/memory-layout-of-c-program/