2020 年秋操作系统 xv6 源码阅读报告 5

虚拟存储

黎善达

1800012961@pku.edu.cn

2020 年 12 月 13 日

1 xv6 的虚拟存储机制概述

1.1 地址翻译的机制

xv6 采用了**二级页表**的结构，每个页大小为 4KB。一个 32 位的地址中，高 10 位为它在页目录的索引（下标），中间 10 位为它在页表项的索引（下标），低 12 位为页内偏移；每个页目录项和页表项的大小都是 **4 字节**。简单计算可得：在 xv6 中，共有 1 个页目录页，1024 个页表页，页目录的基地址由寄存器 `cr3` 保存。地址翻译的过程如图 1所示：

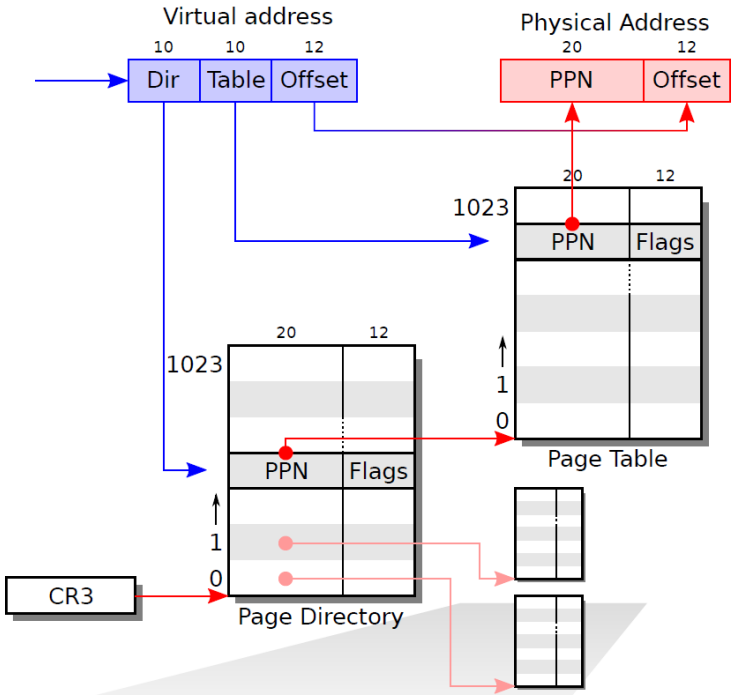

图 1: 二级页表

如前所述，每个页目录项和页表项的大小都是 **4 字节**，即 32 位。这 32 位的高 20 位用来存储页表页号/物理页号，而低 12 位是一些标志位，记录了对应地址是否有效、是否可读可写可执行、是否被修改过等信息，如图 2：

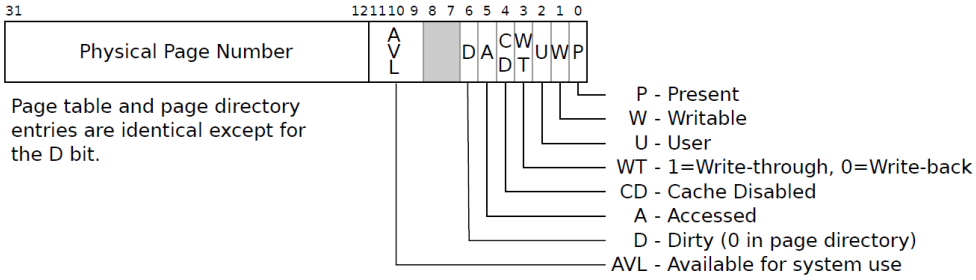

图 2: 页目录项和页表项的结构

1.2 虚拟地址空间的格局

图 3描述了 xv6 虚拟地址空间格局及与物理地址空间的对应：

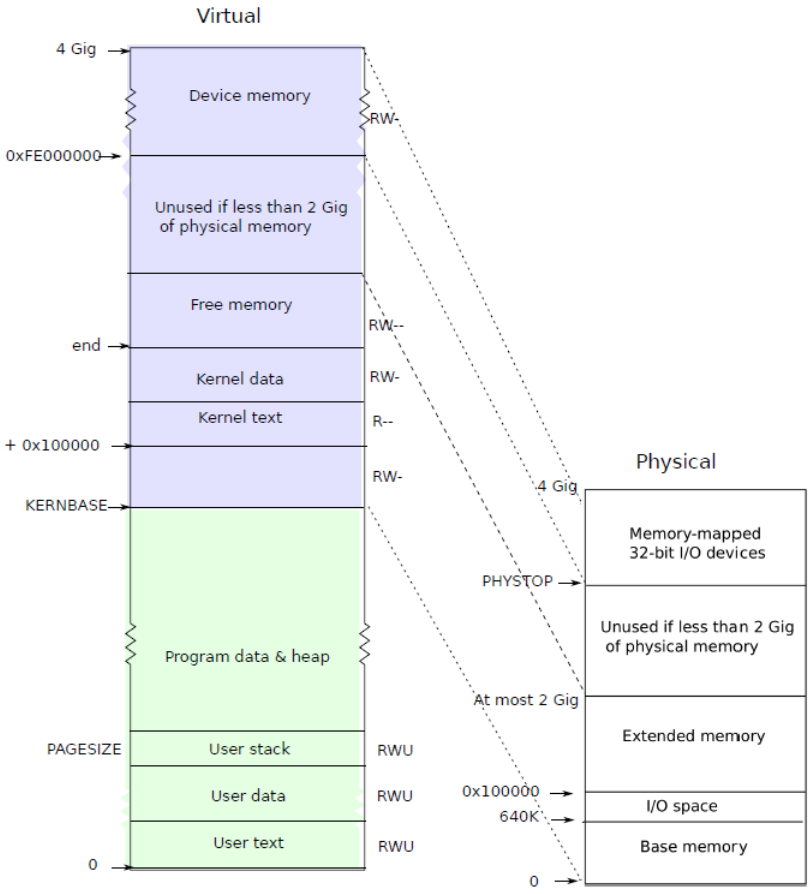

图 3: 虚拟地址空间的格局及与物理地址空间的对应

在 xv6 中，因为虚拟地址是 32 位的，所以虚拟地址空间大小为 4GB。从图 3可以看出，内核的代码和数据存放在地址空间的高地址处，具体来说，以虚拟地址 KERNBASE(0x80000000) 为界，以上为内核空间，以下为用户空间，因此用户、内核各占据 2GB 的虚拟地址空间。物理地址的最低地址 0 对应内核空间的起始地址 KERNBASE，由此向上，BIOS 和内核的代码、数据共占据 4MB 的空间。从虚拟地址空间的最高地址（4G）向下的一段空间为内存映射的 I/O 设备对应的地址空间，这一部分与物理地址直接映射。在这样的设计之下，xv6 不能利用多于 4GB 的物理内存；换言之，xv6 总是假定物理内存大小不足 4GB。

2 关键代码阅读与分析：虚拟存储机制的实现

xv6 源码中，涉及虚拟存储的宏、数据结构和主要功能接口在文件 mmu.h、memlayout.h、kalloc.c 和 vm.c 中定义，这些宏和接口在其它的功能模块中有着广泛的应用。该部分将重点阅读、分析上述四个文件。

2.1 mmu.h

该文件很长，包含了与内存管理单元相关的主要数据结构和宏。概括地说，该文件包含了以下宏定义：

- Eflags 寄存器中的各个标志位，如进位标记、溢出标记、中断使能等。
- 控制寄存器中的各个标志位。
- 各种段选择符，如内核代码段 SEG_KCODE、用户数据段 SEG_UDATA 和程序任务状态段 SEG_TSS 等等。
- 应用断类型 (segment type) 和系统断类型中的各个位，具体包括读、写、执行的权限等。
- 与地址翻译相关的宏。

下面重点说明与地址翻译相关的宏：

如前所述，xv6 采用**二级页表**，确定一个虚拟地址对应的物理地址需要先查找**页目录**，其次查找对应的**页表**，最后结合页内偏移计算出物理地址。该文件提供了获取页目录索引、获取页表索引的宏 PDX 和 PTX。

此外，前文还曾对页目录项、页表项的结构进行了说明：其中不同的位有不同的含义。该文件中提供了对应各个标志位的宏。

此处宏定义还包括一些常量（如页大小、一个页目录中的页目录项数、一个页表页中的页表项数）和一些进行简单计算的宏（如按页大小向上或向下取整）。

除了以上所述的宏定义，该文件还定义了若干结构体，如段描述符和状态段。状态段就是熟知的 **tss**，其中存储的 cr3 寄存器的用于记录**页目录的基地址**。

2.2 memlayout.h

该文件定义了与虚拟内存的布局相关的宏。具体来说，可以分为：

（1）与虚拟内存的布局相关的重要的常量，如 KERNBASE、PHYSTOP 等（参见图 3）。

（2）虚拟地址与物理地址的转换。从图 3可以看出，内核的数据和代码在物理内存存放的时候，其虚拟地址总是恰好比物理地址多 KERNBASE。根据这一规律，xv6 定义了 V2P(a)、V2P(a) 等宏。

2.3 kalloc.c

该文件是**物理内存分配**的功能模块。

在 xv6 中，可分配的物理页通过空闲链表的数据结构进行组织；内核仅保存链表的头指针，链表的各个元素则由空闲的页面本身保存。在初始化时，内核构建这一链表。在系统运行时，如果内核接到了分配页面的请求，就从空闲链表中删去一个元素作为分配的页面；如果内核接受到了释放页面的请求，就将释放的页面加入空闲链表。以上机制就是 xv6 的**动态内存管理机制**，相关的函数在本文件中定义。

2.3.1 kmem 变量

kmem 变量即对应内核维护的可分配物理页构成的链表，相关声明如下：

```
1  struct run {
2    struct run *next;
3  };
4
5  struct {
6    struct spinlock lock;
7    int use_lock;
8    struct run *freelist;
9  } kmem;
```

可以看出，kmem.freelist 即为空闲链表头指针。这一变量是一个**临界资源**，在并发环境下可能出现竞争，因此以上结构中还需要**锁**来确保对空闲链表操作的互斥性。

2.3.2 void kinit1(void *vstart, void *vend) 和 void kinit2(void *vstart, void *vend)

这两个函数初始化空闲链表。它们在 main.c 的 main() 函数中先后被调用。它们的差别在于：

（1）kinit1() 首先被调用，因此其中包含对 kmem.lock 的初始化。

（2）kinit1() 初始化空闲链表时不使用锁，而 kinit2() 会使用。

（3）两个函数初始化的物理内存的范围不同。

在第3节中，这两个函数将被更详细地讨论。

2.3.3 void freerange(void *vstart, void *vend)

该函数将由 vstart 开始到 vend 结束的连续的一段虚拟地址空间释放；其中，vstart 和 vend 未必是页对齐的。实现上，该函数将 vstart 向上取整到页面的整数倍，然后调用 kfree() 函数逐页释放内存。

2.3.4 void kfree(char *v) 和 char* kalloc(void)

kfree() 函数释放指针 v 所指向的内存，释放后的物理页被加入空闲链表。当指针 v 不指向一个页面的基地址，或是指向了非法的空间，系统都调用 panic() 报错崩溃。

kfree() 函数中有一个有趣的细节：它**将被释放的物理页的各个字节上都填入 1**。这些数据被称作*垃圾数据*，用处在于：如果有*悬挂指针*指向这一部分（已被释放的）内存，那么通过指针做间接访问运算将得到错误数据*而非原先的正确数据*，借此使错误的代码能够更快地崩溃。

kalloc() 函数分配一个 4KB 大小的物理页，若分配成功，返回物理页的虚拟地址；否则返回 0。

kfree() 和 kalloc() 函数在维护空闲链表时都运用锁确保互斥。

2.4 vm.c

该文件是虚拟内存机制最核心的文件，其中各个函数的实现离不开文件 mmu.h、memlayout.h 和 kalloc.c 中的宏和接口。下面对其中的重要函数和数据结构进行说明。

2.4.1 static pte_t* walkpgdir(pde_t *pgdir, const void *va, int alloc)

通常情况下，该函数返回指向虚拟地址 va 所对应的*页表项*的指针。

该函数首先根据页目录的基地址 pgdir 和需要翻译的虚拟地址 va 的高 10 位找到对应的页目录项。

如果找到的*页目录项有效*，那么该函数可以直接通过页目录项找到对应的页表（基址），其次通过 va 的中间 10 位（偏移）找到对应的页表项，最后返回该页表项的地址。

如果找到的*页目录项无效*，则分以下两种情况：

- 若参数 alloc 非零，则表明*调用者希望在页目录项无效时创建对应的页表*。为此，该函数通过 kalloc() 函数申请一个物理页作为页表页（若申请失败，则该函数返回 0），其次将申请到的页表页的所有位都填上 0 以确保所有页表项的有效位都是 0，然后更新页目录中的对应项，最后返回新建立的页表中 va 对应的页表项的地址。
- 若参数 alloc 为 0，则直接返回 0。

可以看出，该函数的行为本质上是**对硬件地址翻译流程的模仿**。

2.4.2 static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)

该函数为从虚拟地址 va 开始的，大小为 size 的一块虚拟内存创建映射，使之映射到从物理地址 pa 开始的一块连续的物理内存。

具体实现上，该函数逐页调用 walkpgdir() 函数，其中参数 alloc 的值为 1。

2.4.3 static struct kmap

该文件中定义了 kmap 结构体，然后声明了 kmap 类型的数组。这一数组记录的是内核的虚拟内存的各个片段与物理内存的映射关系和访问权限。在创建新的进程时，这里的数据将用来建立内核部分的页目录项、页表项。

具体来说，此处的定义将内核的虚拟内存划分成了 4 部分，分别是用于 I/O 的空间、代码和只读数据、数据和空闲内存、其它设备（如 I/O 中断控制器）。图 3可作为此部分的参照。

2.4.4 pde_t* setupkvm(void)

该函数根据 kmap[] 提供的信息建立内核部分的页目录项、页表项。具体实现上，该函数对 kmap[] 中的各个部分逐个调用 mappages() 函数完成这项工作。

2.4.5 void switchkvm(void)

该函数使用汇编指令，将硬件中存放页目录地址的寄存器 cr3 中的内容修改为内核线程的页目录地址。

2.4.6 void kvmalloc(void)

该函数完成内核的虚拟内存空间的创建工作。它首先调用 setupkvm() 创建页目录和页表，其次调用 switchkvm() 将创建的页目录的地址存入 cr3 寄存器。在第3节的分析中我们将会看到，在此之前 cr3 寄存器存放的是临时页目录 entrypgdir 的地址。

2.4.7 int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)

实现用户内存空间内存分配和释放：将用户内存从 oldsz 增长至 newsz。

熟知的系统调用 sys_sbrk() 可以给用户进程分配内存，本质上就依赖于这一接口。

2.4.8 int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)

实现用户内存空间内存分配和释放：将用户内存从 oldsz 减少至 newsz。其中，newsz 未必真的要比 oldsz 小；oldsz 可以比真实的进程大小大。这里，对参数接受的灵活性事实上是为了使 allocuvm()

等函数中对它的调用更方便。

3 讨论：第一个地址空间的创建

涉及虚拟存储的源代码中，最令人费解的问题莫过于**第一个地址空间的创建**。我们熟知，在开机之后的较短一段时间内，操作系统还只能通过物理地址进行访存。操作系统如何实现物理地址访存到虚拟地址访存的过渡？在这一过渡阶段中，页目录、页表等数据结构是如何构造的？

为了回答以上问题，笔者花费了很多时间阅读和理解 xv6 中和**启动**相关的代码。事实上，这一过程是一个软硬件结合的过程，本节将对这一过程进行梳理。

本节涉及的文件包括 `main.c`、`entry.S` 等；当然也用到了第2节中的许多内容。

3.1 启动初期：临时使用 `entrypgdir`

当计算机启动时，硬件将 xv6 内核从磁盘中载入内存物理地址 `0x100000` 处，并从 `entry.S` 文件中对应的指令开始执行。此时，x86 的分页硬件在此时还没有开始工作，系统直接使用物理地址访存。

`entry.S` 所做的工作是：建立一个简易的页目录并启用虚拟存储机制。在内核的数据中，已经包含了用于充当临时页目录的数组 `entrypgdir[]`，其定义为：

```
1  // PTE_PS in a page directory entry enables 4Mbyte pages.
2
3  __attribute__((__aligned__(PGSIZE)))
4  pde_t entrypgdir[NPDENTRIES] = {
5    // Map VA's [0, 4MB) to PA's [0, 4MB)
6    [0] = (0) | PTE_P | PTE_W | PTE_PS,
7    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
8    [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
9  };
```

该定义中有以下要点值得注意：

- 第 3 行 `__attribute__((__aligned__(PGSIZE)))` 表明 `entrypgdir[]` 是页对齐的；换言之，该数组恰好占据一整个物理页。
- 定义的两个页目录项中，都将 `PTE_PS` 位置 1。其中注释已经指出：该标记使得页目录项对应的是大小为 4MB 的超级页。因此，在以 `entrypgdir` 作为临时页目录的时间里，地址翻译并不需要经过多级页表分步完成，而是直接一步完成的。
- 定义的两个页目录项将虚拟地址 `[0, 4MB)` 和 `[KERNBASE, KERNBASE+4MB)` 都映射到了物理地址 `[0, 4MB)`。后者是易于理解的：这与图 3中体现的映射关系对应。前者则是因为，在虚拟存储

机制启用后，系统仍需要执行完 entry.S 中的若干条指令，这些指令依然使用物理地址访存，所以虚拟地址 [0, 4MB) 到物理地址 [0, 4MB) 的直接映射也是需要的。

下面分析以下 entry.S 中系统启动时执行的前若干条指令：

```
1  .globl entry
2  entry:
3    # Turn on page size extension for 4Mbyte pages
4    movl %cr4, %eax
5    orl $(CR4_PSE), %eax
6    movl %eax, %cr4
7    # Set page directory
8    movl $(V2P_WO(entrypgdir)), %eax
9    movl %eax, %cr3
10   # Turn on paging.
11   movl %cr0, %eax
12   orl $(CR0_PG|CR0_WP), %eax
13   movl %eax, %cr0
```

通过注释可以看出，这段代码完成了三项工作：（1）设置 cr4 寄存器，启动硬件对 4MB 大小的超级页的支持（与 entrypgdir 的设计对应）；（2）将 entrypgdir 的**物理地址**装载到 cr3 寄存器中；（3）设置 cr0 寄存器，启动页式存储管理。

以上工作完成后，xv6 内核就建立起了以 entrypgdir 为单级页表的虚拟存储机制。此后系统将跳转执行 main.c 中的 main() 函数的代码。

3.2 建立正式的二级页表

尽管 entrypgdir 为页目录的一级页表也能够满足内核正常运转的需求，但 xv6 内核仍将建立二级页表，通过更精细、统一的方式管理内核空间。这部分工作在 main.c 中的 main() 函数完成。以下为 main() 函数中的相关代码：

```
1  int
2  main(void){
3    kinit1(end, P2V(4*1024*1024)); // physical page allocator
4    kvmalloc();                    // kernel page table
5
6    // ...
```

```
7     // Some code omitted.
8
9     kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
10
11    // ...
12    // Some code omitted.
13
14  }
```

首先，main() 调用了 kinit1()，让物理内存分配器初始化物理内存 [end-KERNBASE, 4MB) 部分的页面（即将其加入空闲链表），其中，end 指内核代码的结束位置，因此物理内存 [end-KERNBASE, 4MB) 都确实是空闲的。

其次，main() 调用了 kvmalloc()，建立内核页目录、页表。kvmalloc() 的具体行为见第2.4.6节，在这一步 cr3 寄存器中的内容由 entrypgdir 改为了二级页表的页目录 pgdir 的地址。

最后，main() 调用了 kinit2()，让物理内存分配器初始化物理内存 [4MB, PHYSTOP) 部分的页面。由此，实现了第一个地址空间的创建和动态内存管理机制。