XV6 虚存管理机制

by 1801210840 姜慧强

在 XV6 中有关内存分配的代码在 kalloc.c 和 vm.c 两个文件。

其中 kalloc.c 主要负责物理内存的分配工作，一次分配一个 4096B 的一个 Page。

分配内存给到用户进程，内核态栈，PageTable，缓存区块等。

首先定义了一个链表结构 run, 然后基于这个链表结构，定义了一个同步版链表结构 kmem。

在 XV6 关于 kalloc 的初始化分为两个阶段来做。

1. 首先 main() 调用 kinit1()，并且同时仍使用 entrypgdir 将 entrypgdir 映射的页面放在空闲列表中。
    - 在 kinit1()中主要做的是初始化 kmem 自旋锁，设定用户锁状态，分配一个空闲区域。
2. 然后 main() load 一个物理 PageTable 之后，将 调用其他物理页面及 kinit2（）。
    - 在 kinit2() 中同样分配一个空闲区域，然后还更新用户锁状态。

而分配空闲区域函数 freerange 则是通过循环调用 free 函数来实现的。

而 free 函数则是用来释放指定指针指向的物理内存页面，通常 free 应该通过调用 kalloc（）返回。

而 free 如果在 user 状态时需要先去获取 kmem 的 lock。

kalloc()函数则是去物理内存中申请一个 4096B 大小的 Page。整个过程与 free 较为类似，同样的是通过如果在用户状态则去申请 kmem 的 lock。然后才是申请分配物理内存。

而 vm.c 中定义了一些有关处理虚拟内存相关的函数。

其中 seginit()函数用来初始化 CPU 内核段描述符,每核 CPU 中都需要申请一次。

把逻辑地址映射到虚拟地址中。不能把代码描述符即共享给用户，又共享给系统。

还定义了 walkpgdir()函数，该函数会去 pgdir 对应的 PageTable 中查找虚拟地址相对应的 PTE，如果没找到则会新建一个 PageTable.

函数 mappages 用于对虚拟地址和物理地址，通过循环调用 walkpgdir()来实现。

在 XV6 中每个进程有独立的 PageTable，每个 CPU 中不在 running 状态的进程也会存放在一张表里。

用 kmap 实现内核地址与用户地址之间的映射。

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern
text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern
data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

正因为每个进程都有独立的 PageTable，所以 XV6 在切换进程的过程中需要频繁切换 PageTable。

而切换页表通过 switchkvm()来实现

```
// Switch h/w page table register to the kernel-only page table,
// for when no process is running.
void
switchkvm(void)
{
  lcr3(v2p(kpgdir));   // switch to the kernel page table
}

// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  pushcli();
  cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
  cpu->gdt[SEG_TSS].s = 0;
  cpu->ts.ss0 = SEG_KDATA << 3;
  cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
  ltr(SEG_TSS << 3);
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
  lcr3(v2p(p->pgdir));  // switch to new address space
  popcli();
}
```

switchkvm()通过设置 cr3 寄存器的值为 kpgdir 首地址来实现 PageTable 的保存。 swicthuvm()先关中断，然后修改全局表述表，修改状态中通用寄存器。在设置 cr3 寄存器为进程的 PageTable 地址。

同样的还有相应的 loaduvm, allocuvm, deallocuvm, freevm 等函数，分别用作

- loaduvm 将 I 节点内容读取载入到相应的地址上，通过调用 allocuvm 分配相应的内存空间。

- allocuvm 设置页表，分配虚拟地址进入内存。
- deallocuvm 释放页表
- freevm 释放一个 PageTable 和所有用户态物理内存 Page。首先将 0 到 KERNBASE 的虚拟地址空间回收，然后销毁整个进程的页表
- clearpteu 清除页面上的 PTE_U，用于在用户堆栈下创建无法访问的页面。
- setupkvm()通过 kalloc 分配一页内存作为页目录，然后将按照 kmap 数据结构映射内核虚拟地址空间到物理地址空间，期间调用了工具函数 mappages；
- copyuvm() 复制一个新的页表并分配新的内存，在 XV6 中使用这个函数作为 fork()底层实现

QA

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)

内核代码区位于物理地址 0x100000 处。

PageTable 则是 main.c 文件中定义的 entrypgdir 数组，其中虚拟地址低 4M 映射物理地址低 4M，虚拟地址 [KERNBASE, KERNBASE+4MB) 映射到 物理地址[0, 4MB)

初始化完 PageTable 之后初始化 kvm。

首先调用 kinit1 初始化内核末尾到物理内存 4M 的物理内存空间为未使用，然后调用 kinit2 初始化剩余内核空间到 PHYSTOP 为未使用。

kinit1 调用前使用的是最初的页表，只能初始化 4M，同时需要通过 switchkvm 等获得实际存放页表的物理空间。

XV6 中通过在 main 函数最开始处释放内核末尾到 4Mb 的空间来分配页表，由于在最开始时多核 CPU 还未启动，所以没有设置锁机制。

kinit2 在内核构建了新页表后，能够完全访问内核的虚拟地址空间，所以在这里初始化所有物理内存，并开始了锁机制保护空闲内存链表；

然后 main 函数通过调用 void kvmalloc(void)函数来实现内核新页表的初始化

最后内存布局和地址空间如下：内核末尾物理地址到物理地址 PHYSTOP 的内存空间未使用，虚拟地址空间 KERNBASE 以上部分映射到物理内存低地址相应位置。

XV6 的动态内存管理是如何完成的?

每一个进程都有一张独立的 PageTable，当切换进程时，通过切换 PageTable 来完成内存空间的切换。

Kmem 是一个同步 Memory 链表，在其中定义了一个链表，和一个自旋锁。通过 KMem 来动态分配内存空间。

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

XV6 系统使用 end 指针来标记 XV6 的 ELF 文件所标记的结尾位置，于是[PGROUNDUP(end), 0x400000]范围内的物理内存页将被用作内存分配。

XV6 首先调用 kinit1(end, P2V(0x400000))来将这部分内存纳入虚拟内存页管理。在此之前这部分已经被初始化为 4MB 大小。

XV6 的内存分配器必须知道它要负责管理的内存范围。由于此时虚拟内存已经开启，且页表表项只有两条，因此 XV6 必须利用已有的虚拟地址空间，在其中创建新的页表。这就是 main()函数中 kinit1()和 kvmalloc()所做的事情。

kinit1()函数会调用 freerange()函数，建立从 PGROUNDUP(end)地址开始直到 0x400000 为止的全部内存页的链表。

这样，我们得到了第一组可以使用的虚拟内存页，然后内核就可以运行 kvmalloc()使用这些内存页了。 kvmalloc()函数获得一个虚拟内存页并将其初始化一级页表。这个一级页表的内容在 vm.c 中的 kmap 处被定义

虚拟地址	映射到物理地址	内容
[0x80000000, 0x80100000]	[0, 0x100000]	I/O 设备
[0x80100000, 0x80000000+data]	[0x100000, data]	内核代码和只读数据
[0x80000000+data, 0x80E00000]	[data, 0xE00000]	内核数据+可用物理内存
[0xFE000000, 0]	[0xFE000000, 0]	其他通过内存映射的 I/O 设备

然后，main()函数会调用 seginit()函数重新设置 GDT。

新的 GDT 与之前的 GDT 的主要区别在于新的 GDT 设置了用户数据段和用户代码段。

最后，main()函数会调用 kinit2()将[0x400000, 0xE00000]范围内的物理地址纳入到内存页管理之中。 至此，XV6 的内存页管理系统和内核页表已经全部建立完毕。

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）？

当中断发生时，使用的 PageTable 依然是其对应的用户进程的页表。

每个用户进程都有自己独立的页表，当中断处理程序决定退出当前进程或切换到其他进程时，当前页表才会被切换为调度器的页表（kpgdir）。

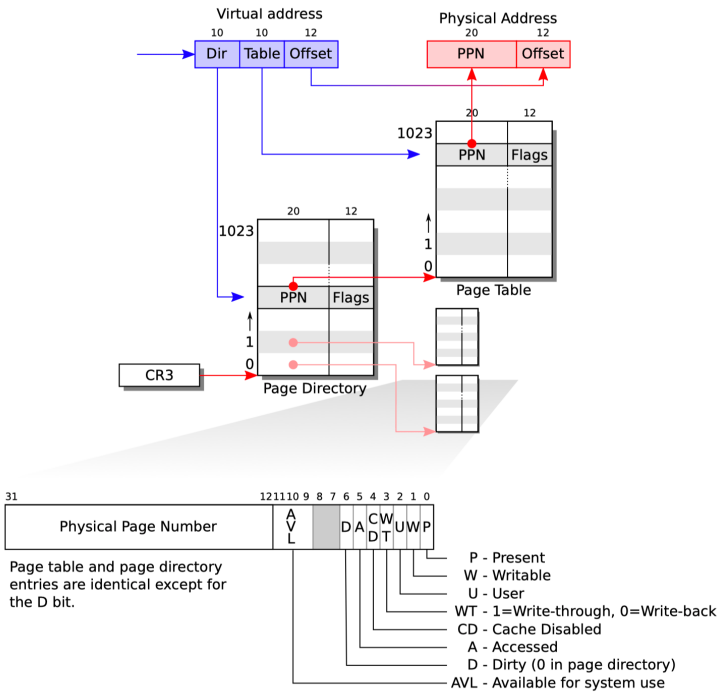

Figure 2-1. x86 page table hardware.

一个内存页 4098B=4KB.

XV6 页表采用的二级目录，一级目录有2^{10}项目，二级目录有2^{20}
页表项 4B，故页表最多 1024 项。最多可存储 4GB