XV6 虚拟存储 阅读报告

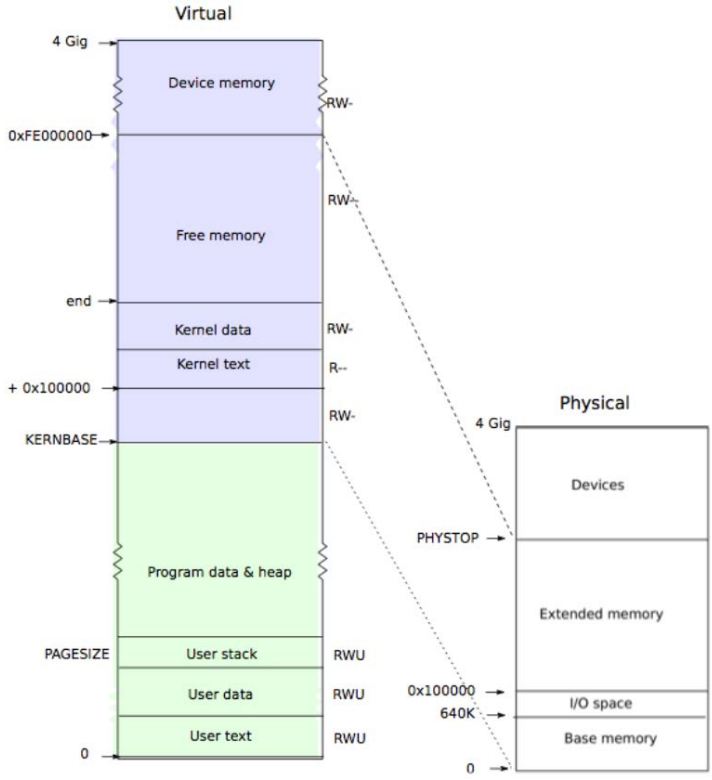
本次代码阅读主要对 XV6 中的进程调度机制进行调研。主要涉及的代码有：memlayout.h、mmu.h、kalloc.c、vm.c、bootasm.S、bootasm.c、entry.S、initecode.S、exec.c。下面我将逐个剖析，以阐释 XV6 的虚拟存储机制。

一、核心代码分析

首先让我们来看一下 XV6 中为了实现虚拟存储管理所设计的宏和功能函数

（一）内存空间布局

XV6 的虚拟内存布局大致如下图所示：



进程的用户内存从 0 开始，最多可以增长到 KERNBASE 处，这限制了用户态下可以使用的最大内存。物理内存中从 0 到 PHYSYTOP 的空间被映射到进程的内核虚拟内存空间中从 KERNBASE 到 KERNBASE+PHYSYTOP 的位置。这样既可以让内核使用自己的代码和数据，又可以让内核直接操作物理页。但由于 KERNBASE 的设定，导致 XV6 的进程可使用的最大物理内存空间被限制为 2GB。一些使用内存映射的 I/O 设备的物理内存位于 0xFE000000 之上，这一部分的地址空间被直接映射到 XV6 的虚拟地址空间的对应区域。此外，KERNBASE 之上为只有内核才能使用的地址空间。

上面叙述中提到的宏定义在 memlayout.h 中，具体内容如下：

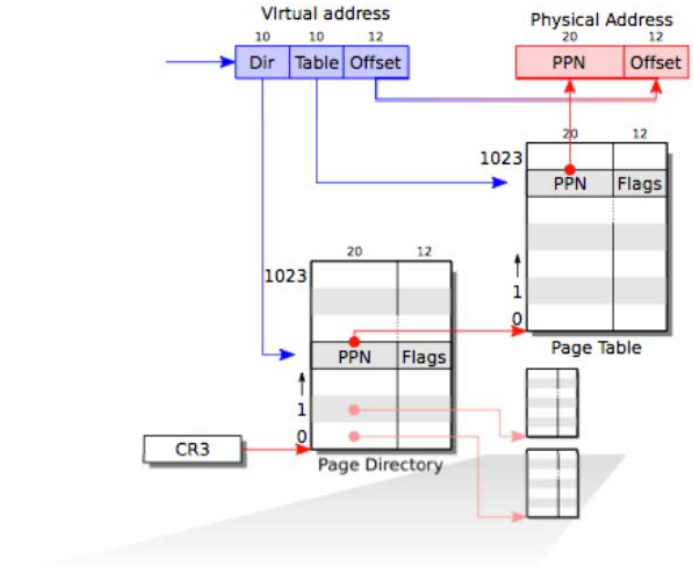
```
#define EXTMEM  0x100000
#define PHYSTOP 0xE000000
#define DEVSPACE 0xFE000000

#define KERNBASE 0x80000000
#define KERNLINK (KERNBASE+EXTMEM)
```

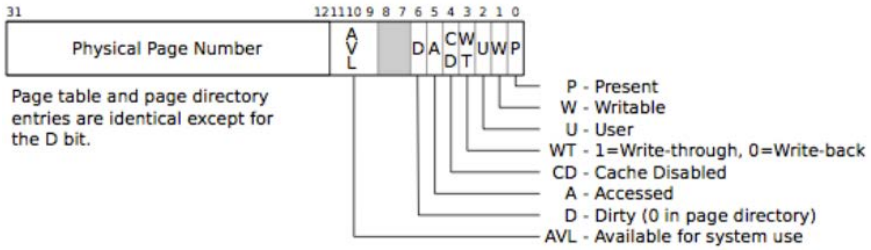
此外，memlayout.h 中还提供了内核的虚拟地址与对应的物理地址之间的转换，其实只需要加上或减去 KERNBASE 这一偏移：

```
#ifndef __ASSEMBLER__

static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }

#endif

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as V2P, but without casts
```

（二）虚拟页、页表、页表条目

XV6 采用 x86（32 位）模式的内存管理。相应的宏和数据结构在 mmu.h 中被定义。x86（32 位）的内存管理大致如下图所示：





XV6（32 位 x86）中采用二级页表，每级页表的 VPN 长度都是 10 位，VPO 为 12 位，因此页大小为 4096 字节。且第一级页表被称作页目录，表项被称作 PDE（page directory entry），第二级页表被称作页表，表项被称作 PTE。在 mmu,h 中，定义和说明如下：

```
// A virtual address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/

                #define PGSHIFT         12
                #define PTXSHIFT        12
                #define PDXSHIFT        22

               #define NPDENTRIES      1024
               #define NPTENTRIES      1024
               #define PGSIZE          4096
```

为了便于计算，mmu.h 中把虚拟地址中三部分的偏移（或者长度）用宏进行了声明。页大小 PGSIZE 确实为 4096 字节（4KB），且每页中 PDE 条目或 PTE 条目均为 1024 个，因此可以推断，PDE、PTE 大小均为 4 字节，这也与 x86 中的设计相吻合。

此外，为了简化后续的使用，mmu.h 还提供了一些用于计算的宏定义，包括：计算所需页面数的上下取整结果，提取页目录索引和页表索引，根据页目录索引、页表索引、页内偏移构建虚拟地址：

```
// page directory index
#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))

     #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
     #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

页表条目（PDE、PTE）除了提供物理地址的高位，还需要提供保护作用，因此在 mmu.h 中定义了如下宏，以便于设置对应的位和取出物理地址：

```
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PWT         0x008   // Write-Through
#define PTE_PCD         0x010   // Cache-Disable
#define PTE_A           0x020   // Accessed
#define PTE_D           0x040   // Dirty
#define PTE_PS          0x080   // Page Size
#define PTE_MBZ         0x180   // Bits must be zero

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
```

可以看到，PTE 的 1-8 位依次为：有效位（是否在内存中）、读写位、用户是否可访问、写回方式、是否缓存、存取、修改、页大小（大页或者普通页）。XV6 中不开启大页模式，因此第 8 位必须为 0。除此之外，第 9 位也要为 0。

PAGE_ADDR 则用于从 PTE 中获取地址的高 20 位。

（三）内存管理单元中的其他数据结构

除了上述内容以外，mmu.h 中还包含了 EFLAGS 寄存器的位索引、控制寄存器 CR0 和 CR4 的位索引、虚拟内存段分类和种类的位表示、段描述符、门描述符、CPU 的 task state 结构。其中的段描述符、门描述符、task state 结构在前面的中断异常处理和进程模型中已分析过，这里不再赘述。

（四）物理内存操作

XV6 对物理内存的操作主要在 kalloc.c 中实现，包含以下数据结构和函数：

1.内存管理数据结构（物理内存分配器）—kmem

kalloc.c 中包含了一个用于管理物理内存的数据结构 kmem，其中包含了一个互斥锁、是否使用锁的标记，和一个空闲链表。空闲链表中的每一项只包含一个指向下一链表位置的指针，用于指示空闲内存的位置。

```
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
```

2.初始化函数

kalloc.c 中还包含了两个用于初始化的函数：kinit1 和 kinit2。这两个函数在 XV6 启动时依次被调用，用于初始化不同区域的内存空间。kinit1 在 main 的最开始被调用，所以它首先初始化 kmem 结构中的互斥锁，然后调用 freerange 函数清空一部分内存。kinit2 函数在 main 执行了其他初始化后进一步初始化内存空间，也是调用 freerange 函数来初始化内存。kinit1 分配时不需要启动锁，而 kinit2 分配完成后需要开启锁。

```
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}

void
kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);
  kmem.use_lock = 1;
}
```

freerange 函数是对 kfree 函数的一个封装，它循环遍历参数指定范围的地址空间，并

调用 kfree 函数把它们逐个清零。每次循环以页为单位。

```
void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

3.内存清空函数

kalloc.c 中提供了 kfree 函数，用于清空单个物理页。kfree 首先判断参数中的指针是否合法（是否指向了页的起始位置、是否小于合法的起始地址、是否大于最大可使用的地址），如果违反了上述要求，那么函数调用 panic 报错。如果地址合法，那么函数会把参数 v 指向的那一页用 memset 初始化。**这里初始化为全为 1 是为了在非法引用时可以尽早崩溃，从而尽早报错。**接下来。如果 kmem 的 use_lock 被标记为非 0，那么就需要获取空闲链表中的锁。获取成功后，函数会把前面清空了的页插到 kmem 中的空闲链表的表头。插入完成后，如果据 kmem 的 use_lock 非零，还要释放 kmem 的锁。

```
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

4.分配物理页

kalloc.c 中的 kalloc 函数提供了对物理页的分配。如果 kmem 的 use_lock 非 0，那么函数首先试图获取 kmem 的锁。然后函数检查表头地址。如果表头地址非空，那么选取表头地址来作为要分配出去的页的起始地址。同时，修改 freelist 的地址为原来链表中的下一项所在的地址。如果 use_lock 为 1，函数会释放锁。最后，函数返回上面分配的指针。如果不存在空闲页，那么 freelist 指针就为空，于是函数也会返回 0（空指针）。

```
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

（五）虚拟内存操作

XV6 对虚拟内存的操作主要在 vm.c 中实现，主要包含以下内容：

1.seginit

这个函数用于**初始化 cpu 的段描述符表**。它为虚拟地址空间中的内核代码段、内核数据段、用户代码段、用户数据段设置了使用权限和 CPU 等级。内核段要在 0 级（内核态）下才能访问，用户段在 DPL_USER 级别下（用户态，3）下即可访问。代码段可读可执行，数据段可读写。此外，函数还对对 cpu 进行映射，并设置段基址寄存器的值（lgdt 指令）。最后函数还会用上面初始化过的 cpu 来初始化全局的 cpu 变量。proc 变量则被设置为 0。

```
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpunum()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);

  // Map cpu, and curproc
  c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);

  lgdt(c->gdt, sizeof(c->gdt));
  loadgs(SEG_KCPU << 3);

  // Initialize cpu-local storage.
  cpu = c;
  proc = 0;
}
```

2.walkpgdir

这个函数根据传入的页目录基址和虚拟地址，**查找页表，寻找对应的页表项 PTE 并返回**。函数首先获取虚拟地址所对应的 PDE 表项。

如果 PDE 存在，函数获取 PDE 指向的页表后，根据虚拟地址中的 VPN2，返回对应的 PTE 条目。

如果 PDE 不存在，且未设置 alloc 选项，认为不分配页表，那么返回空指针。如果 alloc 非 0，且为页表分配物理页失败了（kalloc 返回了空指针），也要返回空指针。如果页表分配成功，那么首先把这一页初始化，然后为对应 PDE 设置地址偏移和权限位。最后在刚才分配的页表中返回对应位置的 PTE 地址。

```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

3.mappages

这个函数为从虚拟地址 va 开始的 size 大小的虚拟地址空间和从物理地址 pa 开始的 size 大小的物理地址空间建立 PTE 条目，从而**建立从虚拟内存到物理内存的映射**。

函数首先把虚拟地址转换为它所在的虚拟页的起始位置和加上 size 后的结束位置（所在页号）。然后函数从第一个虚拟页开始，循环调用 walkpgdir 新建 PTE 条目。如果新建的这个 PTE 条目已经被占用了，那么调用 panic 函数报错并退出；如果 PTE 之阵位空，返回-1，表示空间不足。否则，PTE 会利用物理地址、参数中的权限初始化对应的 PTE 表项，并把表项标记为存在。函数每次循环为一个页分配 PTE，直到最后一页，结束循环，返回 0。

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

4.kmap（数据结构）

kmap 是一个数组，里面的**每个条目指明了内核不同的内存段的虚拟内存中的起始地**

址、物理内存中的起始地址和终止地址，以及段的权限。kmap 中包含了 I/O 空间、代码和只读数据段、数据段和内存、其他设备空间。这些映射关系与前面的图示相吻合。

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,              EXTMEM,    PTE_W},
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W},
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W},
};
```

5.setupkvm

这个函数用于**初始化页表中内核使用的部分**。函数首先分配一个物理页，作为内核页表的页目录。然后函数检查物理地址顶部是否超出了限制（再往上要留给设备使用），如果超出限制则报错。没超出限制的话，函数会根据 kmap 数组中的表项，调用 mappages 函数，为内核的四个段进行虚拟地址和物理地址的映射，并设置相应的权限。如果都成功了，函数会返回页目录的地址，否则会返回空指针。

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (p2v(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

6.kvmalloc

这个函数在调用上面的 setupkvm，建立内核页表之后，调用 switchkvm 函数，把系统的页表切换为内核页表（修改 CR3 寄存器）。

```
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

7.switchkvm

这个函数通过执行 lcr3 指令，**把 CR3 寄存器的值修改为内核页目录的起始地址，从而**

实现页表的切换。

```
void
switchkvm(void)
{
  lcr3(v2p(kpgdir));
}
```

8.switchuvm

这个函数通过设置 cpu 的 task state 上下文结构和栈指针，以及设置 CR3 寄存器的值，来**实现进程切换时的页表（虚拟地址空间）切换**。如果进程的页目录为空指针，会在运行时调用 panic 报错。

```
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();
}
```

9.inituvm

这个函数**把初始化代码（initcode.S）装载到虚拟地址为 0 的位置，并设置对应的页目录**。这里要求占用的地址空间小于一页，否则函数会调用 panic 报错。

```
void
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```

10.loaduvm

这个函数**把程序的各个段加载到页表中**。函数的开始，便对地址进行了检查，要求参数中的地址要与页对齐，否则函数会调用 panic 报错。接下来，函数循环地从文件中读取数据，直到达到了 sz 大小。这里，函数要求虚拟地址所对应的 PTE 必须有效，否则也会调用 panic 报错（因为这意味着某一页还不存在）。函数在读取之前判断了是否会出现不足值，并对读取尺寸进行了处理。如果读取失败，那么返回-1，否则返回 0。

```
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;

  if((uint) addr % PGSIZE != 0)
    panic("loaduvm: addr must be page aligned");
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, p2v(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

11.allocuvm

这个函数为**进程分配新的内存空间，并设置对应的物理内存和虚拟内存之间的映射**。首先函数需要检查 oldsize 和 newsize。如果 newsize 大于可分配给用户进程的最大空间（KERNSIZE），那么返回 0。如果 newsize 小于 oldsize，那么函数返回 oldsize，表示不用进行额外的分配操作。否则，函数会从原有的页数量考试，逐页寻找物理内存，初始化，然后调用 mappages 进行内存映射。如果中途发现物理内存不够用了，那么函数会释放先前已经分配过的空间，打印提示信息，并返回 0，否则函数会返回 newsize，表示分配成功。

```
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
  }
  return newsz;
}
```

12.deallocuvm

这个函数用于**释放一部分用户空间占用的内存**。如果 newsize 不小于 oldsize，那么返回 oldsize，表示不需要释放空间。否则，函数从 newsize 的下一页开始，寻找每页对应的 PTE 表项。如果 PTE 为空，函数会跳过一页 PTE 对应的地址空间大小（因为这说明这页二级页表是在调用 walkpgdir 时新建的）。如果 PTE 不为空，且 PTE 有效，那么函数首先检查物理地址。如果物理地址为 0，说明先前的运行出了问题，函数会调用 panic 报错。否则，函数调用 kfree 来释放这片地址空间，最后清空 PTE 表项。上述循环结束后，函数返回 newsize。

```
int
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  pte_t *pte;
  uint a, pa;

  if(newsz >= oldsz)
    return oldsz;

  a = PGROUNDUP(newsz);
  for(; a  < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
      a += (NPTENTRIES - 1) * PGSIZE;
    else if((*pte & PTE_P) != 0){
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = p2v(pa);
      kfree(v);
      *pte = 0;
    }
  }
  return newsz;
}
```

13.freevm

这个函数用于**释放进程的整个地址空间**。函数首先检查页目录基址是否为空，如果为空，函数调用 panic 报错。否则，函数首先调用 deallocuvm 来释放内核地址空间，接下来函数对每个存在的（PTE_P 为 1）页目录条目所对应的页表虚拟地址调用 kfree 函数，释放所有页表页。最后，函数释放页目录所在的那页。这样便把整个进程的地址空间全部释放。

```
void
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, KERNBASE, 0);
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      char * v = p2v(PTE_ADDR(pgdir[i]));
      kfree(v);
    }
  }
  kfree((char*)pgdir);
}
```

14.clearpteu

这个函数用于**清除 PTE 中的 USER 位**，让对应的 PTE 和物理页在用户态下无法访问。在初始化内核页表时可能会使用到。

```
void
clearpteu(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if(pte == 0)
    panic("clearpteu");
  *pte &= ~PTE_U;
}
```

15.copyuvm

这个函数在 fork 时使用，**用于为子进程提供一份父进程的地址空间副本**。函数首先为子进程设置内核地址空间，如果设置失败，返回空指针。接下来，函数逐页设置子进程的虚拟地址空间。如果在执行中发现 PTE 为空，这说明这个虚拟地址所在的页目录项为空，函数会调用 panic 报错。如果发现 PTE 不存在（PTE_P 为 0），这说明这个页不存在，函数也会调用 panic 报错。如果在 kalloc 分配空间或者 mappages 映射空间时发生错误，那么函数会跳转到 bad 处，释放所有已分配的空间，并返回空指针。否则，函数会返回页目录的基地址。

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)p2v(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, v2p(mem), PTE_W|PTE_U) < 0)
      goto bad;
  }
  return d;

bad:
  freevm(d);
  return 0;
}
```

16.uvaka

这个函数**根据用户的虚拟地址来生成内核对应位置的地址**。两个虚拟地址指向相同的物理内存。但转换之前，需要检查 PTE 是否有效，以及这个地址是否为用户地址。如果上述两条中存在不满足的情况，函数返回空指针。

```
char*
uva2ka(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if((*pte & PTE_P) == 0)
    return 0;
  if((*pte & PTE_U) == 0)
    return 0;
  return (char*)p2v(PTE_ADDR(*pte));
}
```

17.copyout

这个函数用于**把用户地址空间中的部分内容拷贝到内核地址中的对应位置**。在调用 uva2ka 转换地址时，如果出现了权限问题（PTE_U 为 0）会返回空地址，此时 copyput 会停止复制并返回-1。这样便保证了函数只会拷贝用户地址空间。

```
int
copyout(pde_t *pgdir, uint va, void *p, uint len)
{
  char *buf, *pa0;
  uint n, va0;

  buf = (char*)p;
  while(len > 0){
    va0 = (uint)PGROUNDDOWN(va);
    pa0 = uva2ka(pgdir, (char*)va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (va - va0);
    if(n > len)
      n = len;
    memmove(pa0 + (va - va0), buf, n);
    len -= n;
    buf += n;
    va = va0 + PGSIZE;
  }
  return 0;
}
```

二、XV6 如何建立一个地址空间

在分析完上述核心代码后，我们不妨来以 main 程序启动系统是创建地址空间的过程为例，来分析一下 XV6 如何建立一个地址空间。

（一）准备工作：系统引导程序

在运行 main 函数之前，XV6 会先运行引导程序。引导程序分为汇编代码（bootasm.S）

和 C 代码（bootmain.c）两段。

1.bootasm.S

在汇编引导程序中，由于系统在刚启动时位于实模式，这段代码的一个工作便是把实模式转换为保护模式。程序把内嵌的 bootstrap GDT 表加载到 GDT（全局描述符表）寄存器中。加载完毕后，引导加载器将%cr0 中的 CR_PE 置为 1，这样便开启了保护模式。

```
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0
```

```
# Bootstrap GDT
.p2align 2
gdt:
  SEG_NULLASM
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)
  SEG_ASM(STA_W, 0x0, 0xffffffff)

gdtdesc:
  .word   (gdtdesc - gdt - 1)
  .long   gdt
```

但允许保护模式并不会直接改变地址翻译的过程。因此，代码使用了一个 ljmp 指令，跳转到 start32 处，从而真正切换到 32 位的保护模式下。

```
ljmp    $(SEG_KCODE<<3), $start32

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call    bootmain
```

在此时的 32 位模式下，引导加载器首先用 SEG_DATA 初始化数据段寄存器。在进行段初始化工作后，汇编代码调用 bootmain 这一 c 代码。

2.bootmain.c

bootmain 要做的事情是，在磁盘中找到内核程序。内核程序是 ELF 格式的二进制文件。bootmain 首先读入 ELF 头，检查这是否为一个 ELF 文件。检查完毕后，函数会加载文件的代码段，并跳转到 entry 的入口点。

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

3.entry.S

执行完前两步的引导后，系统会跳转到 entry.S，来设置内存管理机制。

```
# By convention, the _start symbol specifies the ELF entry point.
# Since we haven't set up virtual memory yet, our entry point is
# the physical address of 'entry'.
.globl _start
_start = V2P_WO(entry)

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer.
  movl $(stack + KSTACKSIZE), %esp

  # Jump to main(), and switch to executing at
  # high addresses. The indirect call is needed because
  # the assembler produces a PC-relative instruction
  # for a direct jump.
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```

这段代码的工作流程如下：

（1）设置大页模式（4MB）

（2）修改 CR3 寄存器的值，设置页目录为 entrypgdir，内容如下：

```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

（3）开启分页模式，设置 CR0 寄存器中的相应位

（4）设置%esp 寄存器，修改栈指针

（5）跳转到 main 函数，准备运行操作系统的主程序。

（二）系统初始化时对内核地址空间的进一步建立。

执行完上述引导流程后，系统便会真正开始运行。main 函数的执行流程如下：

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  // Finish setting up this processor in mpmain.
  mpmain();
}
```

可以看到，**虽然 entry.S 已经提供了一个能够产生足够多的、可供内核运行的页表映射，但是 main 函数还是选择调用 kvmalloc，重新建立内存映射**。在调用 kvmalloc 之前，调用 kinit1 是为了把先前 entrypgdir 中分配的页释放掉。

执行完前两个函数后，内核创建并且切换到了映射到内核地址空间（KERNBASE 以上）的页表（实际上**对页表和地址空间映射的操作主要由 kvmalloc 中的 setupkvm 来完成**）。

在进行完其他初始化后，main 调用了 kinit2，分配了更多的物理页到空闲链表中。

（三）用户地址空间的建立

上述初始化完成后，main 调用 **userinit 函数，其中调用了前面分析过的 inituvm 函数**，

来初始化用户地址空间。

```
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;
}
```

inituvm 会将控制转移到 initcode.S 这一进程入口点。

```
# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL
```

这段汇编代码会调用 exec 函数来真正进入待执行程序的控制流。exec 时创建地址空间中用户部分的系统调用，它使用某个文件来初始化这一部分的虚拟内存空间。exec 执行流程如下：

（1）调用 namei 打开二进制文件，并读取 ELF 头来做格式检查，并创建一个没有用户部分映射的页表。

```
char *s, *last;
int i, off;
uint argc, sz, sp, ustack[3+MAXARG+1];
struct elfhdr elf;
struct inode *ip;
struct proghdr ph;
pde_t *pgdir, *oldpgdir;

if((ip = namei(path)) == 0)
  return -1;
ilock(ip);
pgdir = 0;

// Check ELF header
if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
  goto bad;
if(elf.magic != ELF_MAGIC)
  goto bad;

if((pgdir = setupkvm()) == 0)
  goto bad;
```

如果读文件失败、格式错误，或者无法创建页目录，那么函数会跳转到 bad 标号下，并释放已分配的内存并返回-1。

```
bad:
  if(pgdir)
    freevm(pgdir);
  if(ip)
    iunlockput(ip);
  return -1;
```

(2)此时，函数已经创建了一个没有用户映射的页表，接下来 exec 函数循环调用 readi、allocvm 和 loaduvm，按段读取文件，并为每个 ELF 段分配内存。如果过程中读取失败，或者内存不足，或者分配内存失败，或者加载 ELF 段进入内存失败，那么还会像上一步那样跳转到 bad 再返回-1。

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
iunlockput(ip);
ip = 0;
```

(3) 接下来函数为进程分配用户栈。虽然函数调用 allocuvm 时占用了两页内存，但只有分配的第二页能被用作栈。这样当用户访问超出一页的栈位置时便可以报错，同时还可以帮助 exec 处理过大的参数。

```
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;
```

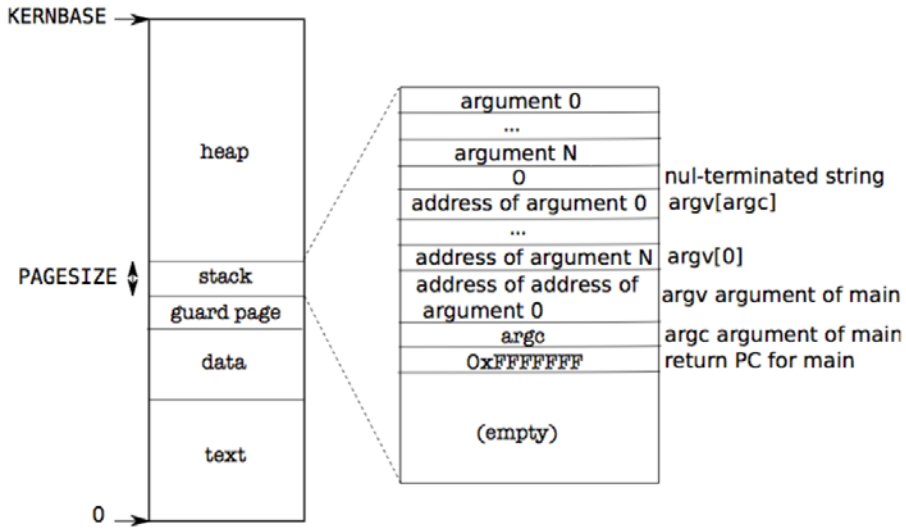
(4) 接下来 exec 把参数拷贝到栈顶，并把指向它们的指针保存到 ustack 中。此外，exec 函数还会在 main 参数列表和 argv 后放一个空指针，充当返回 PC。

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff;  // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4;  // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
  goto bad;
```

（5）最后，为了便于调试，exec 保存了程序名。此外，exec 还设置了全局变量 proc（指代当前进程）的一些关键值，并切换地址空间到当前进程。上述操作全部完成后，函数退出，并返回 0。

```
// Save program name for debugging.
for(last=s=path; *s; s++)
  if(*s == '/')
    last = s+1;
safestrcpy(proc->name, last, sizeof(proc->name));

// Commit to the user image.
oldpgdir = proc->pgdir;
proc->pgdir = pgdir;
proc->sz = sz;
proc->tf->eip = elf.entry;  // main
proc->tf->esp = sp;
switchuvm(proc);
freevm(oldpgdir);
return 0;
```

以上便是从操作系统启动到创建内核地址空间和用户地址空间的全过程。初始化完成后，用户地址空间的最终布局如下：



三、小结

本次代码阅读中，我调研了 XV6 的虚拟存储机制。XV6 使用空闲链表进行物理内存的管理。对于虚拟内存的布局，XV6 采用了 X86 的二级页表架构。第一级为页目录，第二级为页表。每个页的大小为 4KB，PDE 和 PTE 大小均为 4 字节。

在启动系统时，系统首先运行在实模式下，但由于实模式下的可使用空间过小，因此在执行完必要的初始化后，系统很快便会转到 32 位保护模式下。然后引导程序会设置必要的段寄存器值，加载系统入口程序，进一步初始化。当执行完所有的引导后，系统开始运行 main 函数，重新建立地址空间布局。main 会创建内核地址空间的映射，而用户地址空间的映射则由可执行文件内容来确定，由 exec 函数来创建。