XV6 内存管理阅读报告

组长：陈肯
小组成员：郭一臣、张颜、王旭升、袁子栩

一、源代码阅读

kalloc.c：物理内存分配器，主要管理物理内存页面如何组织，如何分配，分配和回收物理内存空间的具体操作。

首先定义了结构体 kmem，XV6 系统的空闲物理内存按页的形式形成一个空闲链表，每页大小定义为 4KB，kmem 结构体就是用来管理链表和访问锁的数据结构，具体如下：

```
14
15  struct run {
16      struct run *next;
17  };
18
19  struct {
20      struct spinlock lock;
21      int use_lock;
22      struct run *freelist;
23  } kmem;
24
```

空闲页面每页大小为 4KB，在每页中用 4 个字节的指针指向下一个页面，形成链表，*freelist 指向该链表的第一个结点，这样整个空闲物理内存都可以通过该链表来进行分配和回收。

```
30  void
31  kinit1(void *vstart, void *vend)
32  {
33    initlock(&kmem.lock, "kmem");
34    kmem.use_lock = 0;
35    freerange(vstart, vend);
36  }
37
38  void
39  kinit2(void *vstart, void *vend)
40  {
41    freerange(vstart, vend);
42    kmem.use_lock = 1;
43  }
44
45  void
46  freerange(void *vstart, void *vend)
47  {
48    char *p;
49    p = (char*)PGROUNDUP((uint)vstart);
50    for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
51      kfree(p);
52  }
53
```

kinit1(void *vstart, void *vend)和 kinit2(void *vstart, void *vend)用来对整个物理空间进行初始化，在 main.c 中首先调用 kinit1(void *vstart, void *vend)初始化 4MB 大小的空间，在之后建立完整页表之后再调用 kinit2(void *vstart, void *vend)把剩余空间初始化。

freerange(void *vstart, void *vend)这个函数用来把[vstart，vend]这个区间的空间按页大小链接到管理物理空间的链表上。

```
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);
}
```

kfree(char *v)用来对物理内存进行回收，进行上下界判定之后把要进行回收的页面刷 1，然后再把这个页面插入到 kmem 的链表里，在这过程中需要申请锁来进行互斥。

```
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;
}
```

kalloc(void)和上述的 kfree(char *v)正好相反，它用来对物理内存进行分配，当需要物理内存时调用 kalloc 函数就可以从 kmem 的链表里取下一个页面分配给需要的用户，在此过

程中同样需要申请锁进行互斥。

vm.c：里面定义了许多函数，具体实现了 XV6 的内存管理的工作，比如页表如何建立，虚拟地址到物理地址如何映射，用户怎么申请和释放空间等。

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

内核调用 kvmalloc(void)建立内核页表，kvmalloc(void)实际上是调用了 setupkvm()按照 kmap 表的映射关系创建了一个线性映射的页表。

```
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;
}
```

setupkvm()会调用 kalloc()来申请物理页面，首先判断 kmem 链表中还有没有空闲的页面，如果有就把这个页面作为页表目录，然后再遍历 kmap 表，调用 mappages 函数来建立虚拟地址到物理地址的映射关系。

```
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
};
```

kmap 表如上图所示，它指定了对应虚拟地址到物理地址的映射关系，对内核的 IO 空间，text 和 data 的空间，设备空间都进行了划分。

```
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;
}
```

mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)函数用来建立虚拟地址和物理地址的映射关系，PGROUNDDOWN 宏用来对地址对齐，按页面大小来递增 a 和 pa，这样每循环一次就创建了一个页表项，直到到达指定的 last 位置为止。

```
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)];
}
```

walkpgdir(pde_t *pgdir, const void *va, int alloc)函数作用是根据给定的虚拟地址去在页目录里找到对应的二级页表的地址，alloc 参数设置为 1 时如果没有找到就会重新分配一页，

最后返回二级页表的地址。

```
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(void)和 switchuvm(struct proc *p)都是将页目录的地址存到 cr3 寄存器中，不同之处在用户切换时会对 cpu 的全局描述符表进行设置。

```
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;
}
```

allocuvm(pde_t *pgdir, uint oldsz, uint newsz)函数用来给用户进程分配从 oldsz 到 newsz 的以页为单位的内存空间，使用 kalloc()函数找到 kmem 链表中空闲的页，然后调用 mappages 函数完成虚拟地址到物理地址的映射，同时在用户标志位和可写入标志位置位。

```
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;
}
```

deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)和上面的正好相反，它是把用户进程的从 newsz 到 oldsz 的按页为单位的内存空间回收，调用 walkpgdir 函数找到页目录对应的二级页表的地址，如果不存在那么就会去查找下一个页目录，如果存在那么就会调用 kfree 函数把内存进行回收。

```
277  void
278  freevm(pde_t *pgdir)
279  {
280    uint i;
281
282    if(pgdir == 0)
283      panic("freevm: no pgdir");
284    deallocuvm(pgdir, KERNBASE, 0);
285    for(i = 0; i < NPDENTRIES; i++){
286      if(pgdir[i] & PTE_P){
287        char * v = p2v(PTE_ADDR(pgdir[i]));
288        kfree(v);
289      }
290    }
291    kfree((char*)pgdir);
292  }
```

freevm(pde_t *pgdir)会把用户的全部空间进行回收操作，是通过 deallocuvm 函数来实现的。

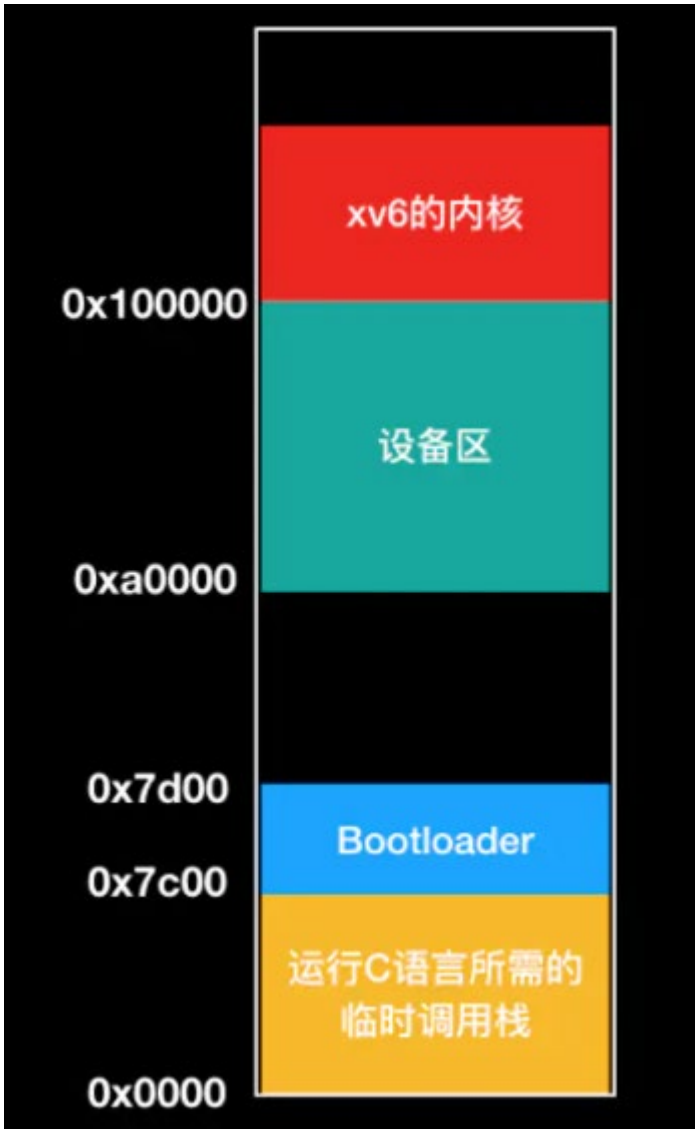
```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  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);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)p2v(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
      goto bad;
  }
  return d;

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

copyuvm(pde_t *pgdir, uint sz)用来为子进程复制一份父进程的页表，在进程部分的 Fork 函数中使用。

二、对问题的思考与讨论

1. 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）？

在查询了网络上的资料后得知，当电脑通电之后，首先会加载引导程序 bootloader，这时 pc 启动在实模式下，但引入 xv6 的内核需要从实模式切换到保护模式，bootasm.s 的工作主要是为实模式做一些收尾工作和切换到保护模式，在保护模式下，bootloader 初始化段寄存器，分配了基本的栈空间和设备区。bootmain.c 作用是在磁盘扇区找到内核程序，内核是 ELF 格式的二进制文件，装载到 0x100000 处。此时的内存布局如下图所示



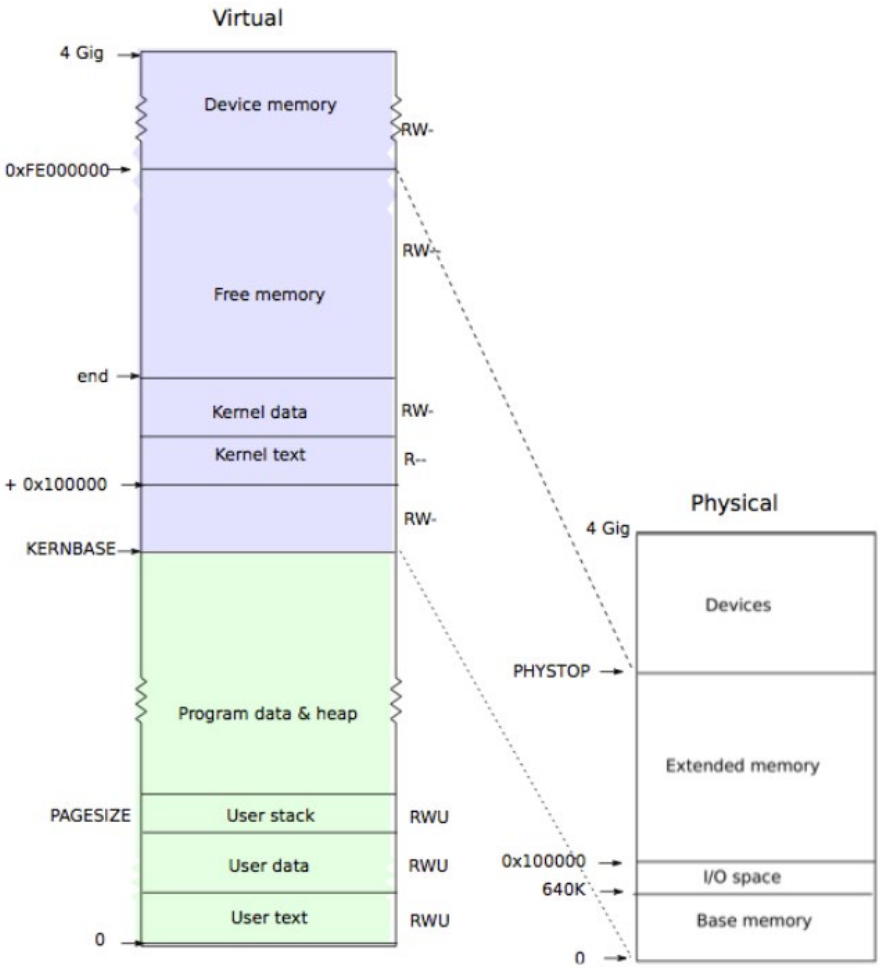
内核代码存在于物理地址低地址的 0x100000 处，页表为 main.c 文件中的 entrypgdir 数组，其中虚拟地址低 4M 映射物理地址低 4M，虚拟地址[KERNBASE, KERNBASE+4MB)映射到物理地址[0, 4MB)。

2. XV6 的动态内存管理是如何完成的？

由第一部分阅读代码可知，在 kalloc.c 中定义了 kmem 这个数据结构，用链表来对 XV6 的内存进行统一管理，用一个锁来进行互斥访问，具体分析可以看源代码阅读部分。

3. 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。

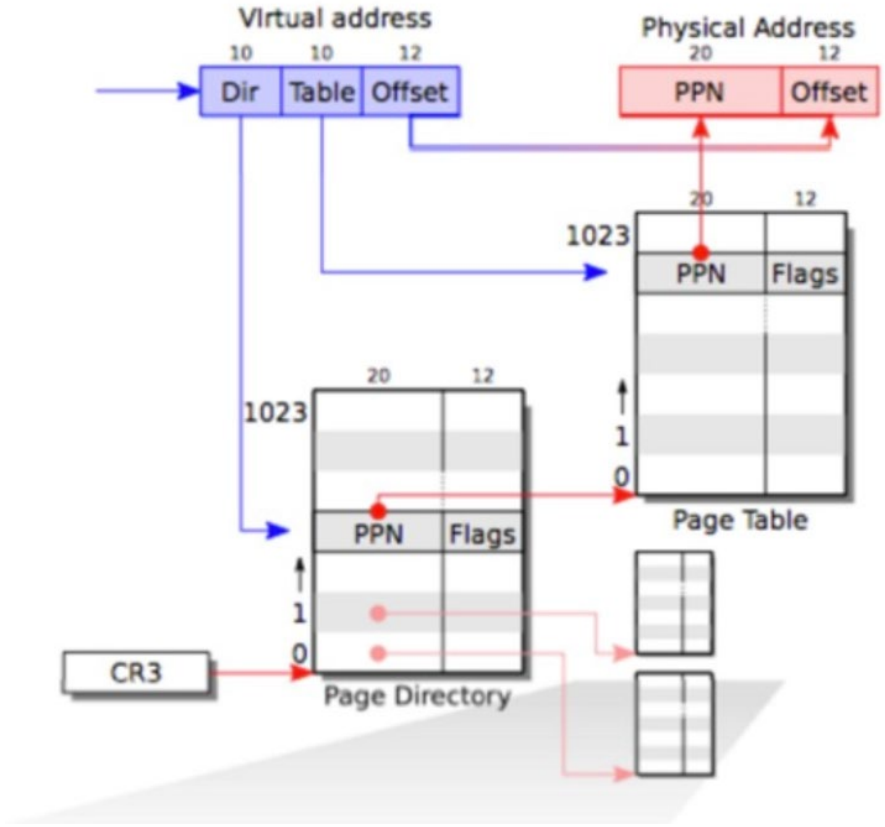
同样在源代码阅读 vm.c 这部分可知，XV6 的虚拟内存是在 main.c 通过调用 kinit1、kinit2、kvmalloc()这三个函数初始化的，虚拟内存布局图如下所示。



在 memlayout.h 中详细的定义了 KERNBASE、PHYSTOP、DEVSPACE 等的地址。

4. 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）？

XV6 的页表结构可参考下图，发生中断时，将换入 CPU 的进程的页表首地址存入 CR3 寄存器，一个内存页大小为 4096 字节，XV6 页表采用的二级目录，一级目录有 1024 条，二级目录有 1024*1024 条，页表项大小为 4 字节，故页表有 4096/4=1024 项，最大支持 4G 内存，虚实地址映射主要是通过 mappages 函数来实现的，具体分析可以参考源代码阅读部分。


Virtual address
10
10
12
Dir
Table
Offset
Physical Address
20
12
PPN
Offset
20
12
1023
PPN
Flags
1
0
Page Table
20
12
1023
PPN
Flags
1
0
CR3
Page Directory