内存的初始化

操作系统实例分析第一部分

何广心、於修远、林钢亮

0| 内存初始化过程概述：

- 分页机制使用二级页表结构

```
int main(void)
{
  kinit1(end, P2V(4*1024*1024));                 // 初始化物理空间分配器
  kvmalloc();                                    // 建立和使用内核页表
  seginit();                                     // 初始化虚拟地址“分段”机制
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP));        // 第二次初始化物理空间分配器
  userinit();                                    // 初始化第一个用户进程
}
```

1| 初始化时给内核分配了多大的空间(几个页)？为什么？每个页都存放了哪些内容?

- kinit1() 初始化物理空间分配器

```
int main(void)
{
  kinit1(end, P2V(4*1024*1024));
}
```

- 调用kinit1()后，从 V2P(end) 到 4MB 的物理空间可以用于分配(kalloc)

- end：内核代码结束的虚地址

1| 初始化时给内核分配了多大的空间(几个页)？为什么？每个页都存放了哪些内容?

- kvmalloc() 建立和使用内核页表

```
int main(void)
{
  kinit1(end, P2V(4*1024*1024));
  kvmalloc();
}

void kvmalloc(void)
{
  kpgdir = setupkvm(); // 初始化内核页表：      分配页目录      4KB(1 page)
                       //                       分配二级页表    64KB(16 page)
  switchkvm();  // 设置硬件，开始使用内核页表
}
```

- kpgdir：内核页表，内核初始化线程以及内核调度器使用的页表

1| 初始化时给内核分配了多大的空间(几个页)？为什么？每个页都存放了哪些内容?

```
void userinit(void)
{
  if((p->pgdir = setupkvm()) == 0)                    // 初始化进程页表        17 page
    panic("userinit: out of memory?");
  ......
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
                                                      // 分配虚拟地址从0开始的一页内存：
                                                      //        页目录
                                                      //        页表     4KB(1 page)
                                                      //        页框     4KB(1 page)
}
```

- inituvm() 分配的内存(0x0..0x1000)：将initcode.S代码移至用户段空间
- kvmalloc() 与userinit() 分配了36页物理空间

2| 总共分配了几次？为什么？

1. 第一次调用函数kinit1()，使V2P(end)..4MB物理内存可分配，此时不使用内存锁
2. 第二次调用函数kinit2()，使4MB..PHYSTOP物理内存可分配，并开始使用内存锁

原因：

1. 直到执行startothers() 前，内核无法调用锁相关的函数，所以先初始化一部分内存供此时的内核线程使用。在startothers() 初始化锁机制后，再执行kinit2() 完成所有物理内存初始化并将内存锁投用，此后其它CPU核可以进行内存分配。
2. 在kinit1() 函数调用前使用的是bootloader装载的页表，只映射4MB内存，所以kinit1() 只能访问4MB内存。
3. kinit1() 和kinit2() 之间的函数不会使用大于4MB的内核内存。

- 内存锁：XV6多核系统中内存是共享资源，管理内存的数据结构kmem是共享变量，分配和释放内存的操作需要同步

3| setupkvm()函数的作用是什么?

1. 每个进程都有各自的页表，不仅映射了用户段，还映射了内核段

2. XV6中每个进程的内核段都使用相同的映射方式

- 调用setupkvm() 为进程分配页目录并建立内核段建立映射（分配页表、填写页表项）

3| setupkvm()函数的作用是什么?

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

  if((pgdir = (pde_t*)kalloc()) == 0)                    // 分配页目录内存
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)             // 为内核段建立映射
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

3| setupkvm()函数的作用是什么?

- 内核段映射

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {       // 内核段映射
 { (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
};
```

3| setupkvm()函数的作用是什么?

- 内核段映射

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {        // 内核段映射
}
```

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, 0x8E000000]	[data, 0xE000000]	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

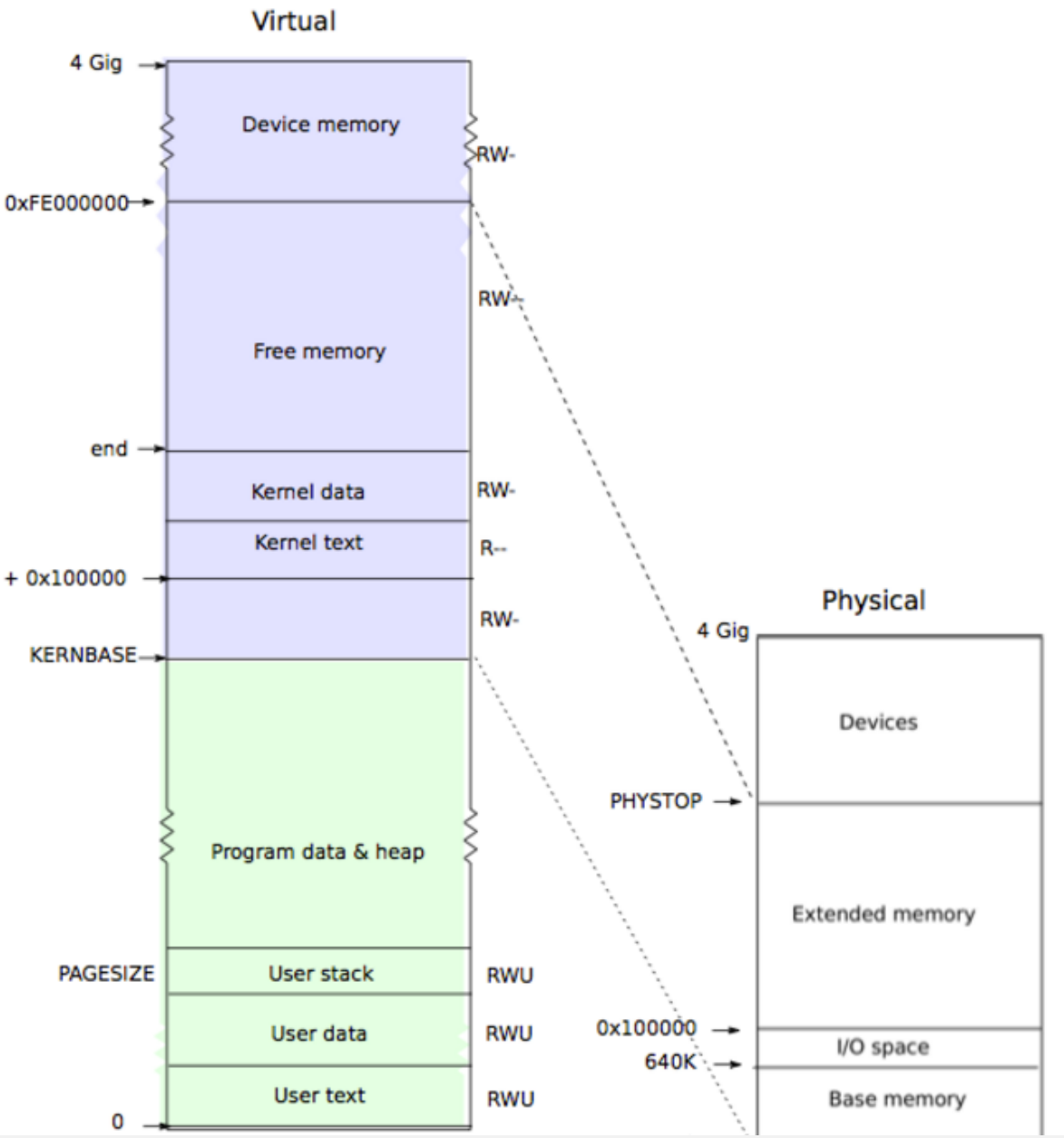
3| setupkvm()函数的作用是什么?

- 内核段映射

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {       // 内核段映射
```

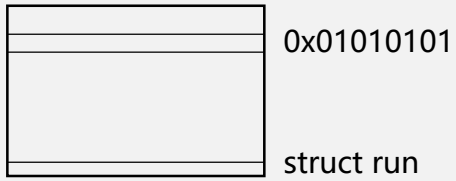
虚拟地址	映射到物理地
[0x80000000, 0x80100000]	[0, 0x1000(
[0x80100000, 0x80000000+data]	[0x100000,
[0x80000000+data, 0x80E00000]	[data, 0xE0
[0xFE000000, 0]	[0xFE0000(

}



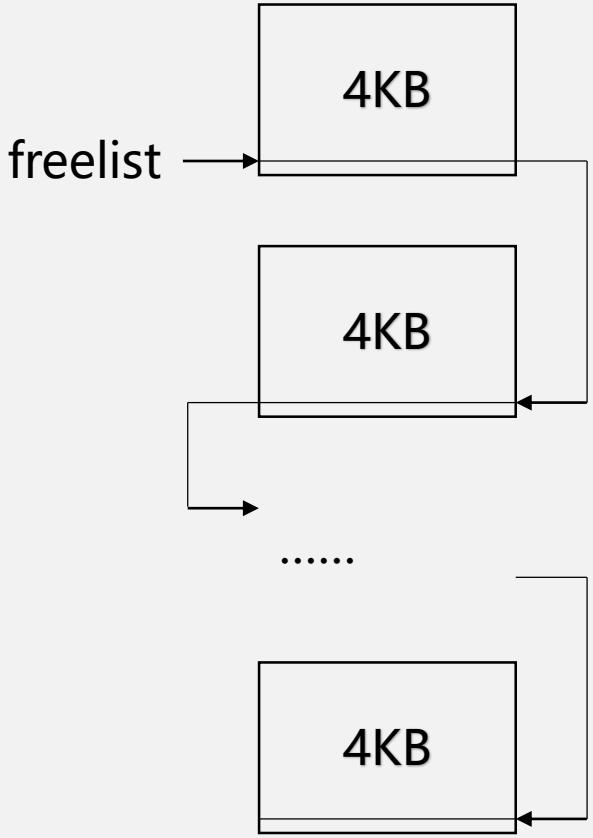
4| xv6中可用物理内存是用什么数据结构进行组织管理的?

- 链表
- kmem.freelist 指向最新释放的页
- 可用物理页有一个next指针
- 指向下一个可用物理页
- 最后一个next=NULL
- 可用物理页数据结构




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

  struct run *freelist;
```



4| xv6中可用物理内存是用什么数据结构进行组织管理的?

- 系统初始化前：freelist=NULL
- 系统初始化时：调用kinit1() 与kinit2() 将空闲物理页框加入freelist

```
void kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");  // 初始化内存锁
  kmem.use_lock = 0;             // 不使用锁
  freerange(vstart, vend);       // 将0..4MB物理空间中空闲页框加入freelist
}
void kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);       // 将剩余空闲物理页框加入freelist
  kmem.use_lock = 1;             // 开始使用锁
}
```

4| xv6中可用物理内存是用什么数据结构进行组织管理的?

```
void kfree(char *v)                    // 释放一页
{
  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);                // 初始化填 1

  if(kmem.use_lock)
    acquire(&kmem.lock);               // 内存锁
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;                   // 将空闲页插入freelist链表头
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

4| xv6中可用物理内存是用什么数据结构进行组织管理的?

```
char* kalloc(void)                    // 分配一页，从表头分配，返回页地址
{
  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;
}
```

进阶| linux内核内存初始化时都做了哪些工作?

```
asmlinkage __visible void __init start_kernel(void)
{
    page_address_init();     // 初始化高端内存(内核高地址的一段内存区)

    setup_arch(&command_line);
            // 1) 初始化系统运行早期内存分配器(memblock、bootmem)
            // 2) 初始化地址空间内核段分页机制
            // 3) 初始化内存管理的基础数据结构(结点pg_data, 内存域zone, 页面page)

    setup_per_cpu_areas();              // 为每个CPU分配内存
                                        // 初始化Per-CPU变量(用于内核同步)

    build_all_zonelists(NULL, NULL);            // 初始化zonelist(内存域链表)

    mm_init();       // 建立内存内核管理器(buddy)，停用bootmem
}
```

谢谢