OS Lab4 内存管理

2018级 信息安全 管箫 18307130012

代码实现

物理内存页管理

JOS内核首先调用i386_init()，此函数会调用mem_init()，而mem_init()通过调用我们将实现的函数来实现内核内存管理。

boot_alloc()

本函数维护了一个static指针nextfree，初始值指向符号值end，也即bss段末尾。
我们要实现的功能是：分配一块大小为传入参数n的物理内存空间，然后更新nextfree的值，使其指向下一处空闲内存地址。

```
if (n == 0) { return nextfree; }
result = nextfree;
nextfree = ROUNDUP((char*)result + n, PGSIZE);
//cprintf("boot_alloc memory allocate at %x, next memory allocate at %x\n", result, nextfree);
return result;
```

mem_init()

接着，mem_init()函数分配内存空间，设置一个追踪全部Page的信息数组。然后调用page_init()对所有页面进行初始化。

```
// Your code goes here:
pages = (struct PageInfo*)boot_alloc(sizeof(struct PageInfo)* npages);
memset(pages, 0, npages * sizeof(struct PageInfo));
```

page_init()

此函数对上一步生成的pages信息数组进行初始化，标记各页面属性，并且建立空闲页面链表。

```
size_t i;
size_t io_hole_start_p = (size_t)IOPHYSMEM / PGSIZE;
size_t kernel_end_p = PADDR(boot_alloc(0)) / PGSIZE;
for (i = 0; i < npages; i++) {
    if (i == 0) {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
    } else if (i >= io_hole_start_p && i < kernel_end_p ) {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
    } else{
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

其中，我们必须注意到，Page0被实模式IDT和BIOS占用，而IO_Hole也不可被用户使用，这两部分的页面必须被标记为已占用，并且不加入空闲页面链表中。

page_alloc()

此函数从空闲页面链表中取出一个PageInfo结构，然后根据传入参数决定是否初始化内存空间。

```
// Fill this function in
if (page_free_list == NULL) {
    //cprintf("page_alloc: out of free memory\n");
    return NULL;
}

struct PageInfo *allocated_page = page_free_list;
page_free_list = page_free_list->pp_link;
allocated_page->pp_link = NULL;

if(alloc_flags & ALLOC_ZERO) {
    memset(page2kva(allocated_page), 0, PGSIZE);
}

return allocated_page;
```

page_free()

此函数将传入的PageInfo结构重新放回空闲页面链表。

```
// pp->pp_link is not NULL.
if (pp->pp_ref != 0 || pp->pp_link != NULL) {
    panic("page_free: pp->pp_ref is non-zero or pp->pp_link is not NULL\n");
}
pp->pp_link = page_free_list;
page_free_list = pp;
```

虚拟内存管理

这部分，我们通过实现页目录和页表的操作函数，实现从虚拟地址到物理地址的内存地址转换。

pgdir_walk()

函数传入参数为页目录的虚拟地址、查询的虚拟地址、布尔值，效果在于查询该虚拟地址所对应的页表条目。

```
pde_t* pde_ptr = pgdir + PDX(va);
    if (!(*pde_ptr & PTE_P)) {
        if (!create) {
            return NULL;
        } else {
            struct PageInfo *pp = page_alloc(1);
            if (!pp) {
                return NULL;
            }
            (pp->pp_ref)++;
            *pde_ptr = (page2pa(pp)) | PTE_P | PTE_U | PTE_W;
        }
    }
return (pte_t *) KADDR(PTE_ADDR(*pde_ptr)) + PTX(va);
```

如果页表项还没有分配，将会分配一个新的页面。

boot_map_region()

此函数映射[va, va+size)的虚拟空间到[pa, pa+size)的物理空间，通过逐个修改PTE完成。

```
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    // Fill this function in
    size_t pg_num = (size % PGSIZE == 0)?(size / PGSIZE):(size / PGSIZE + 1);
    for (int i = 0 ; i < pg_num ; i++, pa += PGSIZE, va += PGSIZE) {
        pte_t *pte = pgdir_walk(pgdir, (void*)va, 1);
        if (pte == NULL) {
            panic("boot_map_region: out of memory\n");
        }
        *pte = pa | PTE_P | perm;
    }
}
```

同时设置映射后，必须更改权限位和标志位。

page_insert()

```
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, 1);
    if (pte == NULL) {
        return -E_NO_MEM;
    }
    if (*pte & PTE_P) {
        if (PTE_ADDR(*pte) == page2pa(pp)) {
            *pte = page2pa(pp) | perm | PTE_P;
            return 0;
        }
        page_remove(pgdir, va);
    }
    (pp->pp_ref)++;
    *pte = page2pa(pp) | perm | PTE_P;
    return 0;
```

“增”操作。将给定的va映射到pp指定的物理空间，通过操作PTE实现。

page_lookup()

```
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, 0);
    if (!pte || !(*pte & PTE_P)) {
        return NULL;
    }
    if (pte_store != 0 ){
        *pte_store = pte;
    }
    return pa2page(PTE_ADDR(*pte));
```

“查”操作，遍历页目录，返回传入va对应的页的PageInfo。

page_remove()

```
page_remove(pde_t *pgdir, void *va)
{
    // Fill this function in
    pte_t *pte;
    struct PageInfo *pp = page_lookup(pgdir, va, &pte);
    if (pp) {
        page_decref(pp);
        *pte = 0;
        tlb_invalidate(pgdir, va);
    }
    return;
}
```

“删”操作，先查找va对应的PageInfo信息，如果存在，那么将其ref减少，清空其PTE并使TLB缓存无效化。

内核线性空间初始化

```
// Your code goes here:
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);

//////////////////////////////////////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
// stack.  The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
//       the kernel overflows its stack, it will fault rather than
//       overwrite memory.  Known as a "guard page".
//     Permissions: kernel RW, user NONE
// Your code goes here:
boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W);

//////////////////////////////////////////////////////////////////////
// Map all of physical memory at KERNBASE.
// Ie.  the VA range [KERNBASE, 2^32) should map to
//      the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
boot_map_region(kern_pgdir, KERNBASE, 0xffffffff - KERNBASE, 0, PTE_W);
```

映射部分由内核所有的空间，包括pages数组、页目录和内核栈等。

地址映射展示

工具函数

```
void
showva2pa_info(uintptr_t va) {
    struct PageInfo *page = page_lookup(kern_pgdir, (void*)va, 0);
    if (page == NULL) { cprintf("VA %x does not have a mapped physical page!", va); }
    uintptr_t pa = page2pa(page);
    uint16_t ref = page->pp_ref;
    pte_t *pte = pgdir_walk(kern_pgdir, (void*)va, 1);
    int u = !!(*pte & PTE_U);
    int w = !!(*pte & PTE_W);
    cprintf("VA: 0x%x, PA: 0x%x, pp_ref: %d, PTE_W: %d, PTE_U: %d\n", va, pa, ref, w, u);
}
```

在kdebug.c中定义了一个工具函数，可以展示传入va对应的物理地址和页信息。页信息通过page_lookup()函数实现了查找，而物理地址则由page2pa完成从页信息到物理地址的转换。

主体函数

```
int
mon_showva2pa(int argc, char **argv, struct Trapframe *tf)
{
    if (argc == 2) {
        showva2pa_info(str2va(argv[1]));
    } else if (argc == 3) {
        int pg_num = (str2va(argv[2]) - str2va(argv[1])) / 0x1000 + 1;
        for (int i = 0 ; i < pg_num ; i++) {
            showva2pa_info(str2va(argv[1]) + 0x1000 * i);
        }
    }
    return 0;
}
```

函数的主体部分定义在monitor.c中，主体函数解析传入的参数，对单页面和页面范围分别进行处理。如果单页面则直接调用工具函数，多参数则作为页面范围，以0x1000作为默认页面大小，逐个调用工具函数。

辅助函数

```
uintptr_t
str2va(char *str)
{
    uintptr_t ptr;
    int temp, i;
    size_t length;

    length = strlen(str) ;
    ptr = 0;
    for (i = 2; str[i] != '0'; i++) {

        if(str[i] <= '9')
            temp = str[i] - '0';
        else
            temp = str[i] - 'a' + 10;
        ptr += temp * ( 1 << (length - i - 1) * 4 );
    }
    return ptr;
}
```

由于输入参数是十六进制格式的字符串化数字，而实际查询地址需要无符号整型格式的虚拟地址，此处编写了自定义函数对两者进行转换。

运行结果展示

make grade

```
make[1]: Leaving directory '/home/overseercouncil/Desktop/lab4'
running JOS: (1.3s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

showva2pa

single page

```
Type 'help' for a list of commands.
K> showva2pa 0xef7f0000
VA: 0xef7f0000, PA: 0x3cd000, pp_ref: 1, PTE_W: 1, PTE_U: 1
```

multi page

```
K> showva2pa 0xef7f0000 0xef7f9000
VA: 0xef7f0000, PA: 0x3cd000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f1000, PA: 0x3cc000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f2000, PA: 0x3cb000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f3000, PA: 0x3ca000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f4000, PA: 0x3c9000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f5000, PA: 0x3c8000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f6000, PA: 0x3c7000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f7000, PA: 0x3c6000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f8000, PA: 0x3c5000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f9000, PA: 0x3c4000, pp_ref: 1, PTE_W: 1, PTE_U: 1
```

问题回答

程序中的地址从什么时候开始都是虚拟地址了，请找到那几行代码。

```
# Load the physical address of entry_pgdir into cr3.  entry_pgdir
# is defined in entrypgdir.c.
movl    $(RELOC(entry_pgdir)), %eax
movl    %eax, %cr3
# Turn on paging.
movl    %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0
```

在entry.S中开启分页机制后，JOS代码将运行在虚拟地址下。

mem_init()函数中kern_pgdir 的虚拟地址是多少？物理地址呢？在我们还未完成本次lab 之前，为什么我们已经可以使用虚拟地址了？

kern_pgdir的虚拟地址是0xef400000，物理地址是0x119000。因为JOS在开始执行时使用了“人工手写”的方式，将部分虚拟地址和物理地址使用硬编码的方式建立了映射。

哪一行代码使得本次lab 所构建的虚拟内存系统真正被使用？请指出它的位置。

```
//////////////////////////////////////////////////////////////////////
// Now that we've allocated the initial kernel data structures, we set
// up the list of free physical pages. Once we've done so, all further
// memory management will go through the page_* functions. In
// particular, we can now map memory using boot_map_region
// or page_insert
page_init();
```

从完成此处的init后，本次lab的分页机制建立，虚拟内存系统投入使用。

此操作系统可支持的最大物理内存是多少？为什么？

[UPAGES, UVPT)这段存储的是所有页的PageInfo结构，每一个物理页都要在其中有一个这样的结构，因此这段的大小实际上就决定了JOS操作系统最多能支持的物理内存大小。

inc/memlayout.h中指出这段的大小是PTSIZE，inc/mmu.h定义了它为PGSIZE * NPTENTRIES，再继续探查可知，这个值的大小为2^22 byte。另外，一个PageInfo结构的大小是8 byte（一个指针和一个32位整数），每个PageInfo结构对应着一个4 KB = 4096 byte的页面。

综上，最多能支持的物理内存为2^22 / 8 * 4096 = 2^31 byte = 2 GB。

请详细描述在JOS 中虚拟地址到物理地址的转换过程。

虚拟地址通过分段机制转换为线性地址，因为JOS中设置所有的段基址均为0，所以线性地址就等于虚拟地址。当处理器碰到一个线性地址后，它的MMU部件会把这个地址分成 3 部分，分别是页目录索引(Directory)、页表索引(Table)和页内偏移(Offset)， 这 3 个部分把原本 32 位的线性地址分成了 10+10+12 的 3 个片段。每个页表的大小为4KB（因为页内偏移为12位）。

举例：现在要将线性地址 0xf011294c 转换成物理地址。首先取高 10 位(页目录项偏移)即960(0x3c0)，中间 10 位(页表项偏移)为274(0x112)，偏移地址为1942(0x796)。首先，处理器通过 CR3 取得页目录，并取得其中的第 960 项页目录项，取得该页目录项的高 20 位地址，从而得到对应的页表物理页的首地址，再次取得页表中的第274项页表项，并进而取得该页表项的首地址，加上线性地址的低12位偏移地址1942，从而得到物理地址。

在函数pgdir_walk()的上下文中，请说明以下地址的含义，并指出他们是虚拟地址还是物理地址：

pgdir

pgdir的含义为页目录的虚拟地址

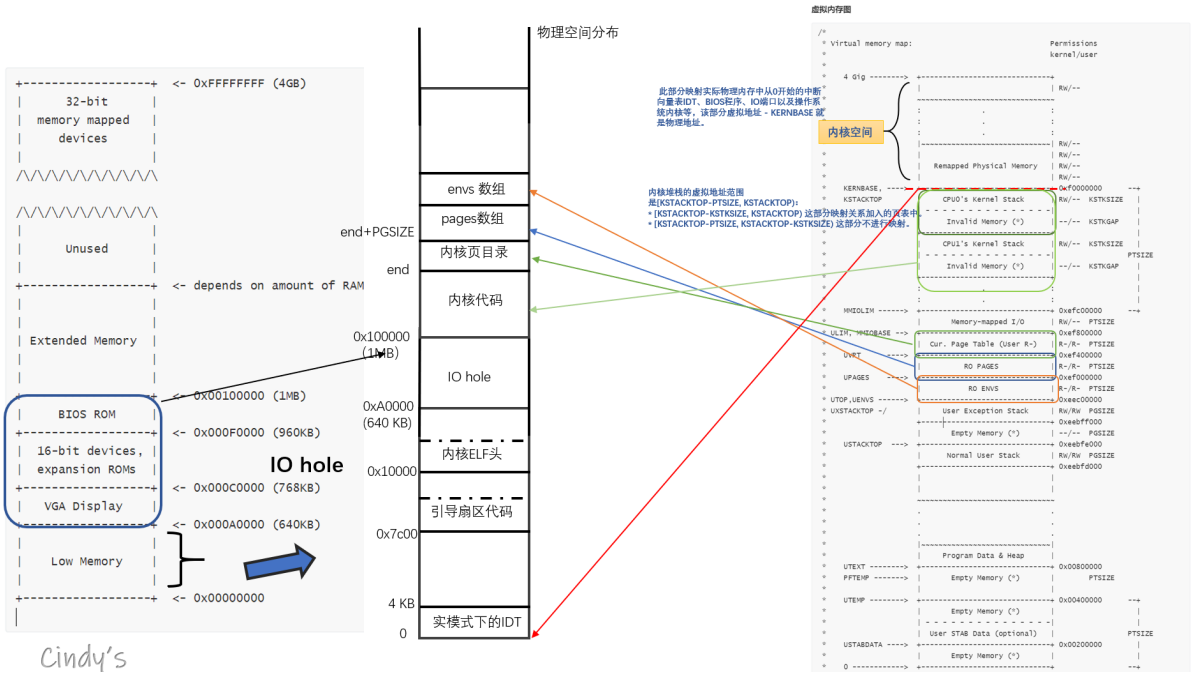
pgtab = PTE_ADDR(pgdir[PDX(va)])

va对应的页目录的物理地址

pg = PTE_ADDR(KADDR(pgtab)[PTX(va)])

va的页表项的物理地址

画出本次Lab 结束后虚拟地址空间与物理地址空间的映射关系，地址空间表示图中应至少包含kern_pgdir 与pages，展示越多的细节越好。（提示：地址空间的表示方式可以参考Lab 1-“The PC's Physical Address Space”小节）



(图片参考https://www.cnblogs.com/cindycindy/p/13524709.html)