《操作系统原理》lab3实验报告

专业：管理学院会计学（辅修）

姓名：莫书琪

学号：17318086

1.实验目的

- 理解基于段页式内存地址的转换机制
- 理解页表的建立和使用方法
- 理解物理内存的管理方法

2.实验过程

练习1：实现 first-fit 连续物理内存分配算法

`first-fit`算法的原理是按地址递增的次序链接空闲分区链。在分配内存时，从链首开始顺序查找，直至找到一个大小能满足要求的分区为止；然后再按照作业的大小，从该分取中划出一块内存空间分配给请求者，余下的空闲分区仍留在空闲链中。若从链首直到链尾都不能找到一个能满足要求的分区，则此次内存分配失败，返回。

实现`first-fit`连续物理内存分配算法需要对`default_pmm.c`中的`default_init_memmap`方法、`default_alloc_pages`方法和`default_free_pages`方法进行修改。

default_init_memap

这个函数的作用是初始化一个空闲块，传入参数为物理页基地址`base`和物理页个数`n`。初始化的思路是：先初始化块中的n个物理页，判断是否为保留页，如果不是，将标志位和连续空页个数清0，引用此页的虚拟页个数清0。n个物理页初始化完成后，还需要修改`base`的连续空页值和空闲页总数，将标志位设置为1，在双向链表头节点`free_list`的前面加入`base`。

原来的代码是在`free_list`的后面插入`base`，这样子顺序就乱了，如果想在双向循环链表的末尾加入新的结点，应该要在头节点的前面插入。

修改后的代码：

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    // 初始化n个物理页
    for (; p != base + n; p ++) {
        assert(PageReserved(p));    // 检查是否为保留页
        p->flags = p->property = 0; // 标志位清0
        set_page_ref(p, 0);         // 清除引用此页的虚拟页个数
    }
    base->property = n;                                 // 设置连续空页值
    SetPageProperty(base);                              // 设置标志位为1
    nr_free += n;                                       // 设置空闲页总数
    // 和原来的代码相比只修改了这一步，把list_add改成了list_add_before
    list_add_before(&free_list, &(base->page_link));    // 在空闲链表中插入base
}
```

其中，物理页的结构体定义为：

```
struct Page {
    int ref;                // 当前页被引用的次数
    uint32_t flags;         // 标志位
    unsigned int property;  // 空闲的连续页数
    list_entry_t page_link; // 两个分别指向上一个和下一个非连续空闲页的指针
};
```

`list_add`和`list_add_before`方法的定义：

```
// 在双向链表的listelm节点后插入elm
static inline void
list_add(list_entry_t *listelm, list_entry_t *elm) {
    list_add_after(listelm, elm);
}

// 在双向链表的listelm节点前插入elm
static inline void
list_add_before(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm->prev, listelm);
}

static inline void
list_add_after(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm, listelm->next);
}

static inline void
__list_add(list_entry_t *elm, list_entry_t *prev, list_entry_t *next) {
    prev->next = next->prev = elm;
    elm->next = next;
    elm->prev = prev;
}
```

`free_list`的含义：所有连续空闲pages中的第一个Page结构都会构成一个双向链表，该链表的头节点是`free_list`

```
/* free_area_t - maintains a doubly linked list to record free (unused) pages */
typedef struct {
  list_entry_t free_list;         // the list header
  unsigned int nr_free;           // # of free pages in this free list
} free_area_t;
free_area_t free_area;
```

default_alloc_pages

这个函数的作用是分配物理内存页，传入参数是分配`n`个物理页大小的连续物理空间。实现思路是：先遍历空闲链表，找到第一个满足要求的空闲页数大于等于n的空闲块，从该块中取出`n`个页，然后把剩余空间插入到头节点`free_list`后面。

原来的代码在找到大小大于等于`n`的空闲块后，程序会先将该页头从链表中断开与删除，然后将剩余空间插入到链表头节点的后面，这里没有按地址顺序安排剩余空间。修改的思路是：先更新剩余空间的属性，令剩余空间形成一个新的块，并把这个块插入到要断开的页头后面，最后再断开页头。

原来的代码：

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    // 遍历空闲链表
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        // 发现一个满足要求的，空闲页数大于等于n的空闲块
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    // 成功找到空闲块后，分配物理内存
    if (page != NULL) {
        list_del(&(page->page_link));                  // 断开page
        if (page->property > n) {
            struct Page *p = page + n;                 // 定位到当前块的第n个地址
            p->property = page->property - n;          // p空闲块大小 = 当前找到空闲块大
小 - n
            list_add(&free_list, &(p->page_link));     // 在free_list后面插入p
        }
        nr_free -= n;                                  // 空闲链表整体空闲页数量-n
        ClearPageProperty(page);                       // 清除page的property
    }
    return page;
}
```

修改后的部分代码：

```
    if (page != NULL) {
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));
        nr_free -= n;
        ClearPageProperty(page);
    }
```

这里用到了`le2page`宏，通过`le2page`可以由`page_link`反向得到节点所属的`Page`结构：

```
// convert list entry to page
#define le2page(le, member)                 \
    to_struct((le), struct Page, member)
// to_struct - get the struct from a ptr
#define to_struct(ptr, type, member)                               \
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

default_free_pages

这个函数的作用是将释放的页重新加入到页链表中，思路是遍历链表，找到第一个页地址大于所释放的第一页的地址，首先判断所释放的页的基地址加上所释放的页的数目是否刚好等于找到的地址，如果是则进行合并，同理找到这个页之前的第一个property不为0的页，判断所释放的页和上一个页是否是连续的，如果连续则进行合并操作。

原来的代码直接把待释放的页头节点`base`插入到头节点`free_list`的后面，没有按地址顺序插入。修改思路是遍历空闲链表，按地址大小找到合适的位置插入。

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    // 遍历这n个连续的Page页，将其相关属性设置为空闲
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    // 由于被释放了n个空闲物理页，base头Page的property设置为n
    base->property = n;
    SetPageProperty(base);
    list_entry_t *le = list_next(&free_list);
    // 遍历空闲链表
    while (le != &free_list) {
        p = le2page(le, page_link);
        le = list_next(le);
        if (base + base->property == p) {
            // 如果当前base释放了n个物理页后，尾部正好能和Page p连上，则进行两个空闲块的合并
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            // 如果当前Page p能和base头连上，则进行两个空闲块的合并
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;    // 空闲链表整体空闲页数量自增n
    // 原来的代码：list_add(&free_list, &(base->page_link));
    // 修改：遍历空闲链表
    for(le = list_next(&free_list); le != &free_list; le = list_next(le)) {
        p = le2page(le, page_link);
        if(base + base->property <= p) {
```

```
            assert(base + base->property != p); // 进行空闲链表结构的校验，不能存在交叉覆盖的地方
            break;
        }
    }
    list_add_before(le, &(base->page_link));    // 将base插入到le前面
}
```

问题1：你的first fit算法是否有进一步的改进空间?

实验中实现的`first fit`算法目前是采用了链表结构和使用顺序查找，可以通过采用其他数据结构（比如数组）并使用二分查找进一步改进。

练习2：实现寻找虚拟地址对应的页表项

get_pte

这个函数的作用是给定虚拟地址`la`和一级页表项`pgdir`，找到这个虚拟地址在二级目录中对应的项，即给出**线性地址**。在保护模式中，x86 体系结构将内存地址分成三种：逻辑地址（虚地址）、线性地址和物理地址。逻辑地址是程序指令中使用的地址，物理地址是实际访问内存的地址，线性地址是逻辑地址到物理地址变换之间的中间层。在段式管理中，程序代码会产生逻辑地址，或说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址；如果启用了分页机制，那么线性地址能再经变换以产生一个物理地址。

对于32位的线性地址，可以拆分成：一级页表项（directory，高10位）、二级页表项（table，中间10位）和偏移（offset，低12位）三部分。给出线性地址的思路：首先获取一级页表项，然后获取二级页表项。如果在查找二级页表项时，发现对应的二级页表不存在，则需要根据`create`参数的值来处理是否创建新的二级页表。如果`create`参数为`0`，则`get_pte`返回`NULL`；如果`create`参数不为`0`，则`get_pte`需要申请一个新的物理页，再在一级页表中添加页目录项指向表示二级页表的新物理页。新申请的页必须全部设定为零，因为这个页所代表的虚拟地址都没有被映射。最后还需要设置一级页表项的控制位，表明新建了二级页表。

```
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    pde_t *pdep = &pgdir[PDX(la)];  // 获取一级页表项
    if (!(*pdep & PTE_P)) {         // 如果二级页表项不存在
        struct Page* page;
        if(!create || (page = alloc_page()) == NULL) {
            return NULL;            // 如果分配页面失败或者不允许分配返回NULL，否则创建新的二级页表
        }
        set_page_ref(page, 1);          // 设置该物理页引用次数为1
        uintptr_t pa = page2pa(page);   // 获取当前物理页面所管理的物理地址
        memset(KADDR(pa), 0, PGSIZE);   // 根据虚拟地址清空该物理页面的数据
        *pdep = pa | PTE_U | PTE_W | PTE_P;  // 设置控制位
    }
    return &((pte_t*)KADDR(PDE_ADDR(*pdep)))[PTX(la)];  // 返回二级页表项
}
```

代码里面用到一些宏和定义：

```
// A linear address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table    | Offset within Page  |
```

```
// |      Index     |     Index      |                    |
// +----------------+----------------+--------------------+
//  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \----------- PPN(la) -----------/
PDX(la)                                 // 返回虚拟地址la的页目录索引
KADDR(pa)                               // 返回物理地址pa对应的虚拟地址
set_page_ref(page,1)                    // 设置此页被引用一次
page2pa(page)                           // 得到page管理的那一页的物理地址
struct Page * alloc_page()              // 分配一页出来
memset(void * s, char c, size_t n)      // 设置s指向地址的前面n个字节为'c'
PTE_P 0x001                             // 表示物理内存页存在
PTE_W 0x002                             // 表示物理内存页内容可写
PTE_U 0x004                             // 表示可以读取对应地址的物理内存页内容

typedef uintptr_t pde_t;                // 一级页表项 page directory entry
typedef uintptr_t pte_t;                // 二级页表项 page table entry
```

问题1：请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中每个组成部分的含义以及对ucore而言的潜在用处。

每个页目录项/页表项都由一个32位整数来存储数据，其结构如下：

```
      31-12      9-11     8    7    6   5   4      3    2   1   0
+--------------+-------+-----+----+---+---+-----+-----+---+---+---+
|     Offset   | Avail | MBZ | PS | D | A | PCD | PWT | U | W | P |
+--------------+-------+-----+----+---+---+-----+-----+---+---+---+
// address in page table or page directory entry
#define PTE_ADDR(pte)   ((uintptr_t)(pte) & ~0xFFF) // PTE的高二十位表示该PTE对应的页表物理地址
#define PDE_ADDR(pde)   PTE_ADDR(pde)               // PDE的高二十位表示该PDE对应的页表物理地址

/* page table/directory entry flags */
#define PTE_P           0x001                   // 存在位
#define PTE_W           0x002                   // 可写位
#define PTE_U           0x004                   // 该页访问需要的特权级
#define PTE_PWT         0x008                   // 是否使用直写
#define PTE_PCD         0x010                   // 若为1则不对该页进行缓存
#define PTE_A           0x020                   // 该页是否被使用过
#define PTE_D           0x040                   // 脏位
#define PTE_PS          0x080                   // 设置page大小
#define PTE_MBZ         0x180                   // 恒为0
#define PTE_AVAIL       0xE00                   // 如果没有被CPU使用可以保留给操作系统
```

问题2：如果ucore执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情?

1. 将引发页访问异常的地址保存到`cr2`寄存器中
2. 在中断栈中依次压入`EFLAGS`，`CS`，`EIP`保护现场，以及页访问异常码`error code`将外存的数据换到内存中。如果页访问异常是发生在用户态，则还需要先压入`ss`和`esp`，并且切换到内核栈
3. 引发`Page Fault`，根据中断描述符表查询到对应`Page Fault`的`ISR`，跳转到对应的`ISR`处执行，接下来将由软件进行`Page Fault`处理

练习3：释放某虚地址所在的页并取消对应二级页表项的映射

page_remove_pte

当释放一个包含某虚地址的物理内存页时，需要让对应此物理内存页的管理数据结构Page做相关的清除处理，使得此物理内存页成为空闲；另外还需把表示虚地址与物理地址对应关系的二级页表项清除。这个函数的作用是判断此页被引用的次数，如果仅仅被引用一次，则这个页也可以被释放，否则只能释放页表入口。

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if (*ptep & PTE_P) {                      // 如果传入的二级页表项可用
        struct Page *page = pte2page(*ptep);  // 获取该页表项对应的物理地址
        if (page_ref_dec(page) == 0)          // 如果该页的引用次数在减1后为0
            free_page(page);                  // 释放当前页
        *ptep = 0;                            // 清空PTE
        tlb_invalidate(pgdir, la);            // 刷新TLB内的数据
    }
}
```

这里用到了`pte2page`宏，`page_ref_dec`函数和`tlb_invalidate`函数：

```
// 根据pte值获取相应页面
static inline struct Page *
pte2page(pte_t pte) {
    if (!(pte & PTE_P)) {
        panic("pte2page called with invalid pte");
    }
    return pa2page(PTE_ADDR(pte));
}

// 返回减1的引用数
static inline int
page_ref_dec(struct Page *page) {
    page->ref -= 1;
    return page->ref;
}

// invalidate a TLB entry, but only if the page tables being
// edited are the ones currently in use by the processor.
void
tlb_invalidate(pde_t *pgdir, uintptr_t la) {
    if (rcr3() == PADDR(pgdir)) {
        invlpg((void *)la);
    }
}
```

运行结果截图

```
    kern/init/init.c:64: grade_backtrace+34
ebp:0xc0117ff8 eip:0xc010008b args:0xc010621c 0xc0106224 0xc0100d0b 0xc0106243
    kern/init/init.c:29: kern_init+84
memory management: default_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07ee0000, [00100000, 07fdffff], type = 1.
  memory: 00020000, [07fe0000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

3.实验感想

做这个实验的时候花了很多时间理解ucore中的双向循环链表数据结构，完成实验后感觉对段页式管理有了更深刻的认识。