lab2 实验报告

练习1：实现 first-fit 连续物理内存分配算法（需要编程）
需要实现的方法

default_init（initialize internal description&management data structure free block list, number of free block）

default_init_memmap（setup description&management data structcure according to the initial free physical memory space）

default_alloc_pages（allocate >=n pages, depend on the allocation algorithm）

default_free_pages（free >=n pages with “base” addr of Page descriptor structures(memlayout.h)）

工程里面代码的注释介绍的很详细，且有一些提示类的代码，我们需要完善。

default_init

static void
default_init(void) {
  list_init(&free_list);
  nr_free = 0;

default_init_memmap

static void
default_init_memmap(struct Page *base, size_t n) {
  // 需要循环设置每一页的flags, property, ref, 且要在链表中连接起来(1->2->3)
  // 然后把这一段内存区域加入到free_list中，设置nr_free

```
assert(n > 0);
struct Page *p = base;
for (; p != base + n; p++) {
    assert(PageReserved(p));
    p->flags = p->property = 0;
    set_page_ref(p, 0);
}

// 设置这段内存区域第一页的属性
SetPageProperty(base);
base->property = n;
nr_free += n;
// 加入到free_list中
// 双向循环链表
// free_list<->add1_pages<->add2_pages<->add3_pages<->free_list
list_add(&free_list, &base->page_link);
```

}

default_alloc_pages

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;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        // 找到符合条件的
        if (p->property >= n) {
            page = p;
            break;
        }
    }

    if (page != NULL) {
        // 如果分配n个后还有page页，则后面的补上
        if (page->property > n) {
            struct Page *pn = page + n;
            pn->property = page->property - n;
            SetPageProperty(pn);
            list_add_after(&page->page_link, &pn->page_link);
        }

        ClearPageProperty(page);
        nr_free -= n;
        list_del(&page->page_link);
    }

    return page;
```

}

default_free_pages

static void
default_free_pages(struct Page *base, size_t n) {
  // 逻辑如下：
  // 1 先把base的属性更新
  // 2 循环在free_list中查找
  // 2.1 *base可能刚好在某个空闲块的后面
  // 2.2 *base可能刚好在某个空闲块的前面
  // 2.3 可能在尾部或者某两个之间
  // 2.4 对应的物理页地址没有问题的情况下插入到空闲链表中
  // 3、可能刚好应该插入的头的情况

```
    // 先做一些处理（设置flags, ref, 页链表头的property等）
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
```

```
    set_page_ref(p, 0);
}
base->property = n;
SetPageProperty(base);

struct Page *t = base + base->property;
// 再往合适的位置（free_list）插
int i = 0;
list_entry_t *le = &free_list;
while ((le = list_next(le)) != &free_list) {
    p = le2page(le, page_link);
    i++;
    // 刚好在前面
    // 两个if不能直接break，因为可能存在
    // p + p->property = base， 下一个链表
    // base + base->property = p(next le)
    if (base + base->property == p) {
        base->property += p->property;
        // tempP不是头
        ClearPageProperty(p);
        list_del(&(p->page_link));
    }
    // 刚好在后面
    if (p + p->property == base) {
        p->property += base->property;
        ClearPageProperty(base);
        base = p;
        list_del(&(p->page_link));
    }

}
le = &free_list;
while ((le = list_next(le)) != &free_list) {
    // 在中间(前插，即找到比base地址大的，插入前面)
    p = le2page(le, page_link);
    if (base + base->property <= p) {
        assert(base + base->property != p);
        break;
    }
}

list_add_before(le, &(base->page_link));

nr_free += n;
```

}

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

你的first fit算法是否有进一步的改进空间

暂时没想到，可能要用线段树来实现

练习2：实现寻找虚拟地址对应的页表项（需要编程）
get_pte也就是给定线性地址，得到线性地址对应的二级页表项（page table entry）对应的内核虚地址（kernel virtual address），二级页表项不存在，分配一个包含此项的二级页表。

根据理论课，及Intel的参考手册大概可以知道地址的映射关系。

此函数入参为pgdir指针，线性地址，及是否给PT分配一个页，出参为pte的虚地址。从线性地址的转换可以想到大概步骤如下：

线性地址（la）前10位，在pgdir（页目录表中查找）对应的页表项，得到指向页表项的指针pdep。
检查pdep指向的页表向是否有效（即指向的二级页表向是否存在）。
如果无效（不存在），分配一个新的页（Page）为一个新的二级页表（一页为4KB，即分配的一个page，刚好可以存1024个page table entry）。把这个页的物理转换为内核虚拟地址，用memset方法，将内容清零。
将上面得到的地址填入到PDE中，设置为可读可写。
获取PDE中的物理地址，转换为内核虚拟地址，即得到二级页表的起始地址。再加上PTE对应的偏移量，就得到真正PTE的地址。
代码如下：

```
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    // typedef uintptr_t pde_t
    // PDX 左边10位(PDE)
    // PTX 中间10位(PTE)
    // KADDR - takes a physical address and returns the corresponding kernel virtual address
    // #define PTE_ADDR(pte)   ((uintptr_t)(pte) & ~0xFFF) address in page table or page directory entry
    // #define PDE_ADDR(pde)   PTE_ADDR(pde) address in page table or page directory entry
    // pdep: page dirtory
    pde_t pdep = NULL;
    uintptr_t pde = PDX(la);
    pdep = &pgdir[pde];
    // 非present也就是不存在这样的page（缺页），需要分配页
    if (!(pdep & PTE_P)) {
        struct Page *p;
        // 如果不需要分配或者分配的页为NULL
        if (!create || (p = alloc_page()) == NULL) {
            return NULL;
        }
        set_page_ref(p, 1);
        // page table的索引值（PTE)
        uintptr_t pti = page2pa(p);
```

```
        // KADDR: takes a physical address and returns the corresponding kernel virtual address.
        memset(KADDR(pti), 0, sizeof(struct Page));

        // 相当于把物理地址给了pdep
        // pdep: page directory entry point
        *pdep = pti | PTE_P | PTE_W | PTE_U;
    }

    // 先找到pde address
    // address in page table or page directory entry
    // 0xFFF = 111111111111
```

```
// ~0xFFF = 1111111111 1111111111 000000000000
// #define PTE_ADDR(pte)   ((uintptr_t)(pte) & ~0xFFF)
// #define PDE_ADDR(pde)   PTE_ADDR(pde)
uintptr_t pa = PDE_ADDR(*pdep);
// 再转换为虚拟地址（线性地址）
// KADDR = pa >> 12 + 0xC0000000
// 0xC0000000 = 11000000 00000000 00000000 00000000
pte_t *pde_kva = KADDR(pa);

// 需要映射的线性地址
// 中间10位(PTE)
uintptr_t need_to_map_ptx = PTX(la);
return &pde_kva[need_to_map_ptx];
```

}

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

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。

参考Intel手册：

从结构上可以看出来，页目录项（Page Directory Entry）和页表项（Page Table Entry）差不多，只是12-31位代表的含义不相同。

P 存在位（present）

R/W 读/写位（Read/Write）

U/S 访问该页需要的特权级

PWT write through（为1表示此项采用通写方式，表示该页不仅是普通内存，还是高速缓存）

PCD 若为1表示该页启用高速缓存，为0表示禁止将该页缓存

A 访问位，若为1表示该页被 CPU 访问过啦，所以该位是由 CPU 设置的。

D 脏页位，当 CPU 个页面执行写操作时，就会设置对应页表项的D位为 1，此项仅针对页表项有效，并不会修改页目录项中的位。

PAT 页属性位，在页一级的粒度上设置内存属性。

G 全局位，与TLB有关，为1表示该页是全局页，该页在高速缓存TLB中一直保存。

Avail 可用位，为1表示用户进程可用该页，为0则不可用。对操作系统无效。

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？

出现页访问异常

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（需要编程）
这个比较简单，大概步骤如下：

检查是否ptep是否存在
如果存在，则根据ptep得到对应的page，pqge的ref减一，如果减一后为0，则释放page（free_page）
清除ptep，刷新tlb
代码如下：

static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if ((ptep & PTE_P)) {
      struct Page *page = pte2page(ptep);
      if (page_ref_dec(page) == 0) {
         free_page(page);
      }

```
     // clear second page table entry
     *ptep = 0;

     // flush tlb
     tlb_invalidate(pgdir, la);
 }
```

}

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是啥？

pages的每一项与页表中的页目录项和页表项有对应，pages每一项对应一个物理页的信息。一个页目录项对应一个页表，一个页表项对应一个物理页。假设有N个物理页，pages的长度为N，而页目录项、页表项的前20位对应物理页编号。

如下图所示：

一个PDE对应1024个PTE，一个PTE对应1024个page。

如果希望虚拟地址与物理地址相等，则需要如何修改lab2，完成此事？ 鼓励通过编程来具体完成这个问题

只需要把KERNBASE从0xC0000000改成0x00000000，且把kernel.ld里面的0xC0100000改为0xC0000000