

实验二 物理内存管理

计31 张正 2013011418

一、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

二、实验内容

本次实验包含三个部分。首先了解如何发现系统中的物理内存；然后了解如何建立对物理内存的初步管理，即了解连续物理内存管理；最后了解页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，对段页式内存管理机制有一个比较全面的了解。

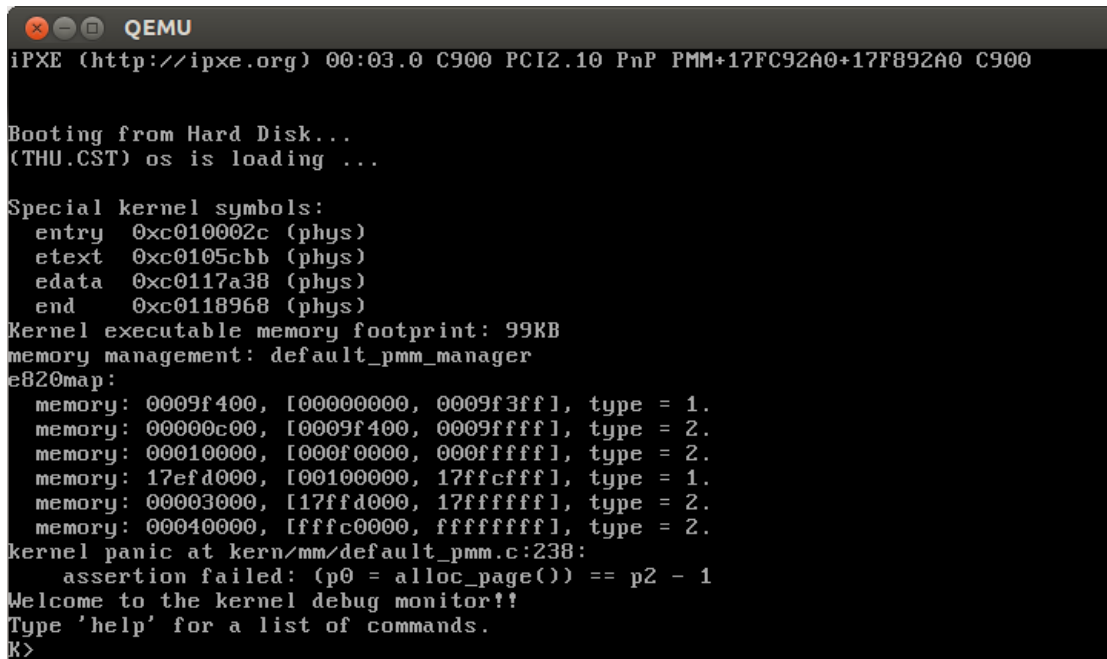
练习0：填写已有实验

使用meld手动完成合并

练习1：实现 first-fit 连续物理内存分配算法

在实现first fit 内存分配算法的回收函数时，要考虑地址连续的空闲块之间的合并操作。

首先对lab2的代码进行make，发现出现了如下的错误



```
QEMU
iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+17FC92A0+17F892A0 C900

Booting from Hard Disk...
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0xc010002c (phys)
  etext 0xc0105cbb (phys)
  edata 0xc0117a38 (phys)
  end   0xc0118968 (phys)
Kernel executable memory footprint: 99KB
memory management: default_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 17efd000, [00100000, 17ffcfff], type = 1.
  memory: 00003000, [17ffd000, 17ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
kernel panic at kern/mm/default_pmm.c:238:
  assertion failed: (p0 = alloc_page()) == p2 - 1
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

可以发现其错误出现在default_check(void)这个函数之中，该函数为检查firstfit算法的函数。

分析源码后可知，在其对内存进行一些列分配释放操作后，再次申请一页内存后出现错误，可知其在最后一次`p0 = alloc_page()`申请中得到内存页的位置与算法规则不相符，回到`default_alloc_pages(size_t n)`、

`default_free_pages(struct Page *base, size_t n)`函数中可以分析得到，在分配函数和释放函数中都出现错误：

分配函数中若分得的块大小大于申请页数，则需要将多余的页形成一个块，按照从低地址到高地址的顺序挂回`free_list`中，而不是直接挂到`free_list`的后面。

将释放页与空闲页合并操作之后，只是将新的空闲区域挂到了`free_list`的后面，并没有按照从低地址到高地址的顺序将其挂到`free_list`之中，导致后面`check`函数中出现错误。对源代码做如下修改：

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    list_entry_t *le, *len;
    le = &free_list;

    while((le=list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if(p->property >= n){
            int i;
            for(i=0;i<n;i++){
                len = list_next(le);
                struct Page *pp = le2page(le, page_link);
                SetPageReserved(pp);
                ClearPageProperty(pp);
                list_del(le);
                le = len;
            }
            if(p->property>n){
                (le2page(le,page_link))->property = p->property - n;
            }
        }
    }
```

```

        ClearPageProperty(p);
        SetPageReserved(p);
        nr_free -= n;
        return p;
    }
}
return NULL;
}

```

释放内存与申请内存思路大致相同：

```

static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(PageReserved(base));
    list_entry_t *le = &free_list;
    struct Page * p;
    while((le=list_next(le)) != &free_list) {
        p = le2page(le, page_link);
        if(p>base){
            break;
        }
    }
    for(p=base;p<base+n;p++){
        list_add_before(le, &(p->page_link));
    }
    base->flags = 0;
    set_page_ref(base, 0);
    ClearPageProperty(base);
    SetPageProperty(base);
    base->property = n;

    p = le2page(le,page_link) ;
    if( base+n == p ){
        base->property += p->property;
        p->property = 0;
    }
    le = list_prev(&(base->page_link));
}

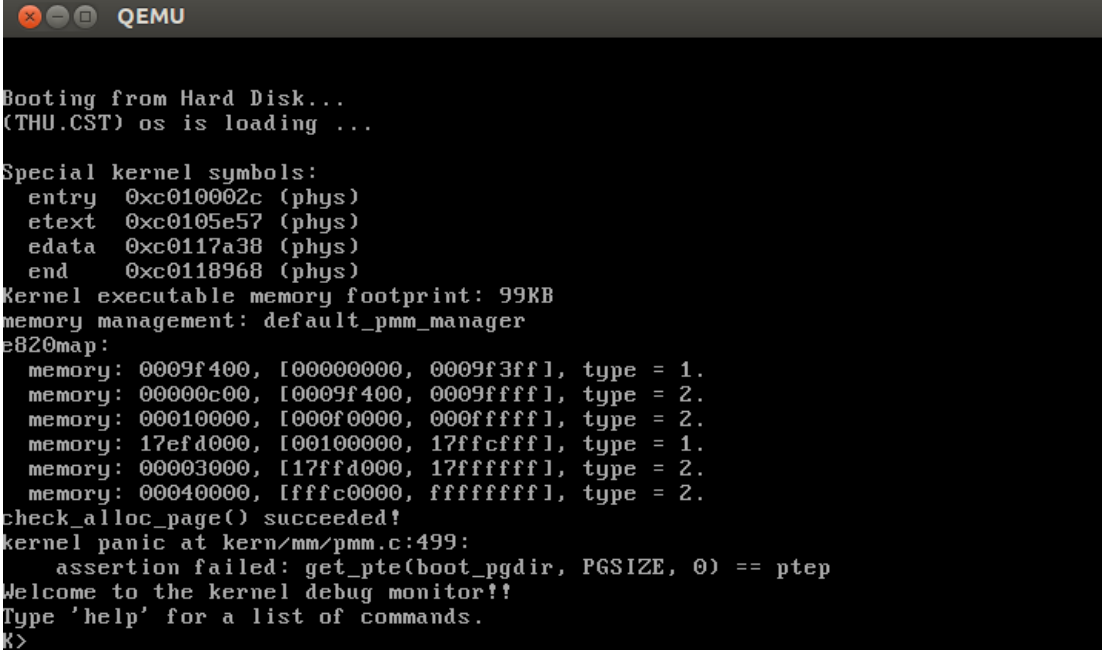
```

```

p = le2page(le, page_link);
if(le!=&free_list && p==base-1){
    while(le!=&free_list){
        if(p->property){
            p->property += base->property;
            base->property = 0;
            break;
        }
        le = list_prev(le);
        p = le2page(le,page_link);
    }
}
nr_free += n;
return ;
}

```

再次make，得到结果去下图，check_alloc_page() succeeded!



```

QEMU
Booting from Hard Disk...
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0xc010002c (phys)
  etext 0xc0105e57 (phys)
  edata 0xc0117a38 (phys)
  end    0xc0118968 (phys)
Kernel executable memory footprint: 99KB
memory management: default_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 17efd000, [00100000, 17ffcfff], type = 1.
  memory: 00003000, [17ffd000, 17ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
kernel panic at kern/mm/pmm.c:499:
  assertion failed: get_pte(boot_pgdir, PGSIZE, 0) == ptep
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
k>

```

练习2：实现寻找虚拟地址对应的页表项

通过设置页表和对应的页表项，可建立虚拟内存地址和物理内存地址的对应关系。本练习需要补全get_pte函数 in kern/mm/pmm.c，实现其功能。

从上次make的结果可以看出，在kern/mm/pmm.c中get_pte实现失败，所以对该函数进行了如下的补全。

```

pde_t *pdep = &pgdir[PDX(la)];
if (!(*pdep & PTE_P)) {

```

```

struct Page *page;
if (!create || (page = alloc_page()) == NULL) {
    return NULL;
}
set_page_ref(page, 1);
uintptr_t pa = page2pa(page);
memset(KADDR(pa), 0, PGSIZE);
*pdep = pa | PTE_U | PTE_W | PTE_P;
}
return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];

```

• 请描述页目录项（Pag Director Entry）和页表（Page Table Entry）中每个组成部分的含义和以及对ucore而言的潜在用处。

页目录项（Pag Director Entry）和页表（Page Table Entry）的前20位分别储存着页表的基址，物理页的基址，后十二位储存着属性值

R/W：页是否是只读的

U/S:是一般用户访问还是超级用户访问

A：是否访问过该页

P-驻留位

D-修改位

等

对于ucore，这种方式起到了一种安全保护机制，

• 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？

CPU会把产生异常的线性地址存储在CR2中，并且把表示页访问异常类型的值（简称页访问异常错误码，errorCode）保存在中断栈中。

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

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。请仔细查看和理解page_remove_pte函数中的注释。为此，需要补全在 kern/mm/pmm.c中的page_remove_pte函数。

函数补全为

```

if (*ptep & PTE_P) {
    struct Page *page = pte2page(*ptep);

```

```
if (page_ref_dec(page) == 0) {  
    free_page(page);  
}  
*ptep = 0;  
tlb_invalidate(pgdir, la);  
}
```

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？’

有，**ref**表示这样页被页表的引用记数。如果这个页被页表引用了，即在某页表中有一个页表项设置了一个虚拟页到这个**Page**管理的物理页的映射关系，就会把**Page**的**ref**加一；反之，若页表项取消，即映射关系解除，就会把**Page**的**ref**减一。**flags**表示此物理页的状态标记。