Lab2 实验

练习 0

1、使用 meld 图形化对将 lab1 改过的代码合并到 lab2 上，想想还有点小激动。

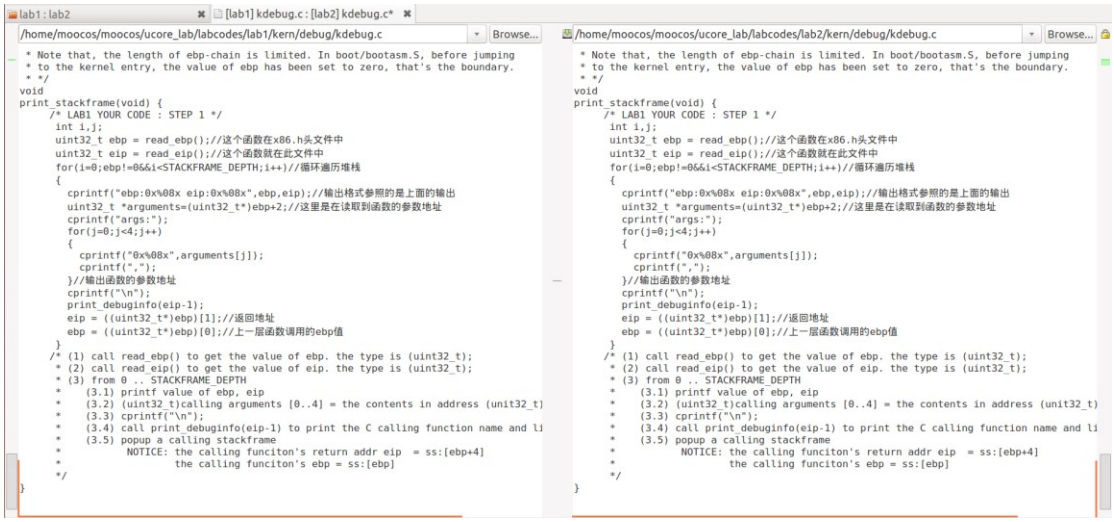

图 1 正在操作 meld 合并代码

meld 能够自动检测两个类似代码段的差别以及代码段增删改的内容，在实际操作过程中只需要手动导入文件夹（文件）即可。

2、 正式做 lab2 之前我又回去看了 lab1，因为感觉时间有点久了，很多地方都比较模糊。Lab1 只是简单的操作系统启动过程，而 Lab2 则涉及到物理内存分配、页目录表页表等内容。

3、从 lab2 开始用 understand 进行代码的查看和修改，understand 的主要优点有两个：一是能提供清晰的结构目录，二是能够实现引用函数的原位置追溯，只需轻轻一点，就能看到引用函数的函数定义。

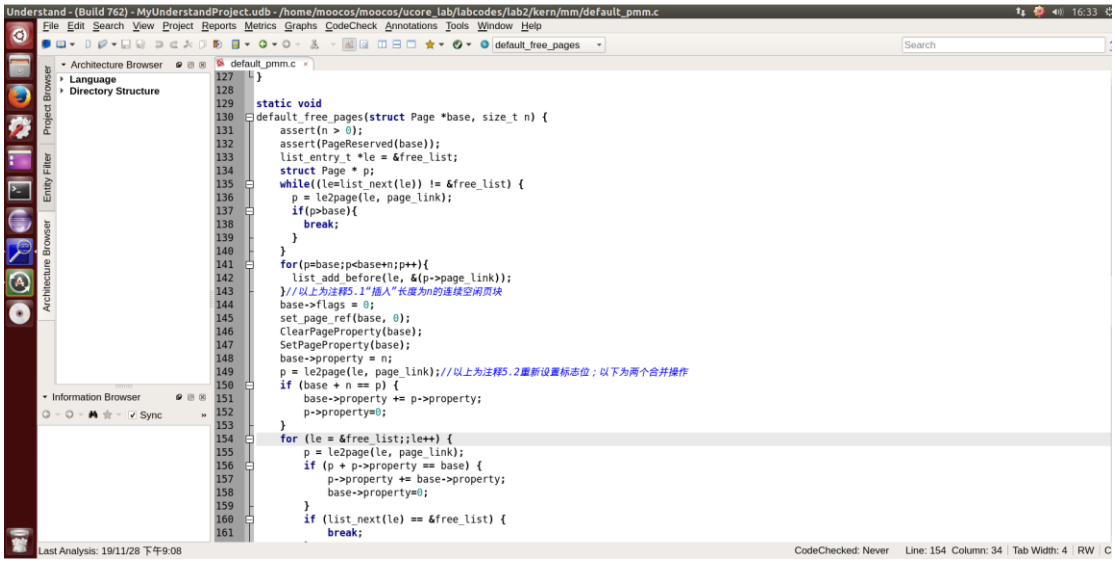

图 2 正在用 understand 进行编程

练习 1

1、实现 first-fit 算法过程，主要涉及到的是 default_alloc_pages 和 default_free_pages 两个函数，第一个函数实现的是分配连续的空闲物理内存页，第二个函数则与之相反，释放物理内存页使之空闲。在实现物理内存分配算法之前，需要对物理内存进行初始化，即将每页用一个双向链表链接起来，即 default_init_memmap 函数实现的内容。

2、在实现 default_alloc_pages 的时候，我错误地认为只需要将分配出去的 n 个页中起始地址最小的那一页从空闲链表中删除即可，结果出现如下错误提示：

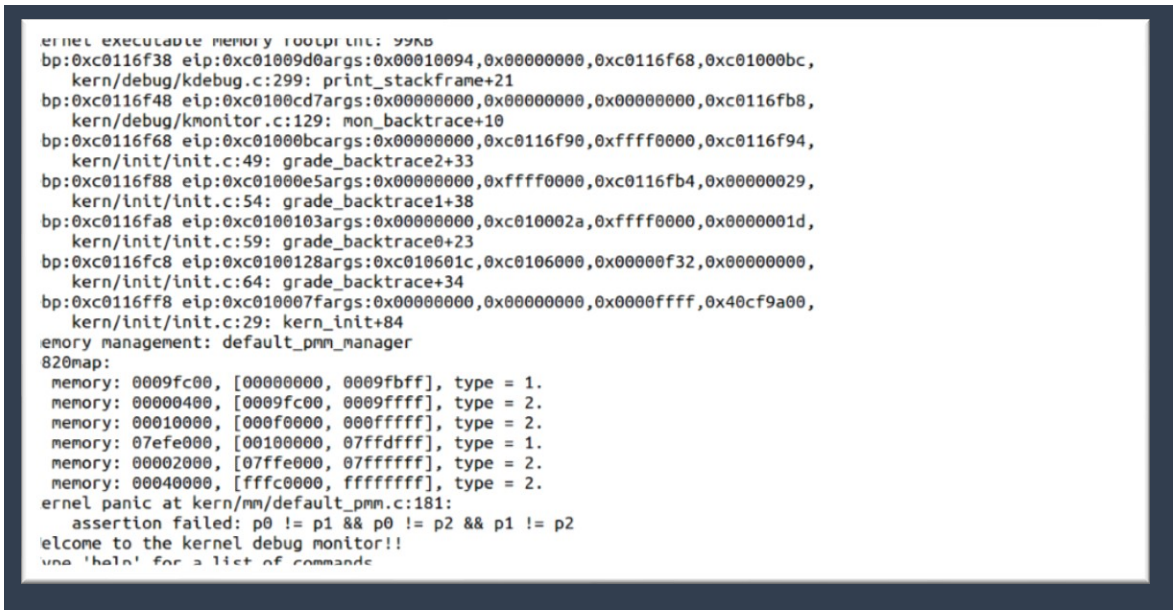

图 3 一个错误提示

事实上，在实现初始化物理内存的过程当中，是将每一页都用链表链接了起来，链接连续空闲物理页块的 free_area_t 数据结构链接最小单元也是每个物理页，因此 free_list 链表当中删除 n 个空闲物理页（这 n 个页被分配出去）需要执行 n 次 list_del 删除操作，否则便会出现已经被分配的页被再次分配，导致内存崩溃。

3、个人认为 default_free_pages 的实现是练习一最难的部分，难点主要在于地址连续的空闲块之间的合并操作。注意，需要合并的情况有两种：第一种是当前正在释放的连续物理内存页块（以 base 为头，页数量为 n）后面不跳页地链接一个空闲物理内存页块，即 base+n 为另一个连续空闲物理内存页块地址最小的一页的起始地址，这个时候的处理方法比较简单，合并的时候只需额外将 base 代表的那一页的 property 置成 n+(base+n->property)即可；第二种情况是 base 是被分配出去的连续物理内存页块地址最小的一页的起始地址，这个时候首先需要遍历双向循环链表 free_list，找到 base 前面那个空闲物理页块的起始地址再进行相应合并操作。下图所示蓝色代表已经分配的连续物理页块，浅蓝色代表当前准备释放的连续物理页块，空白代表连续空闲空闲物理页块。

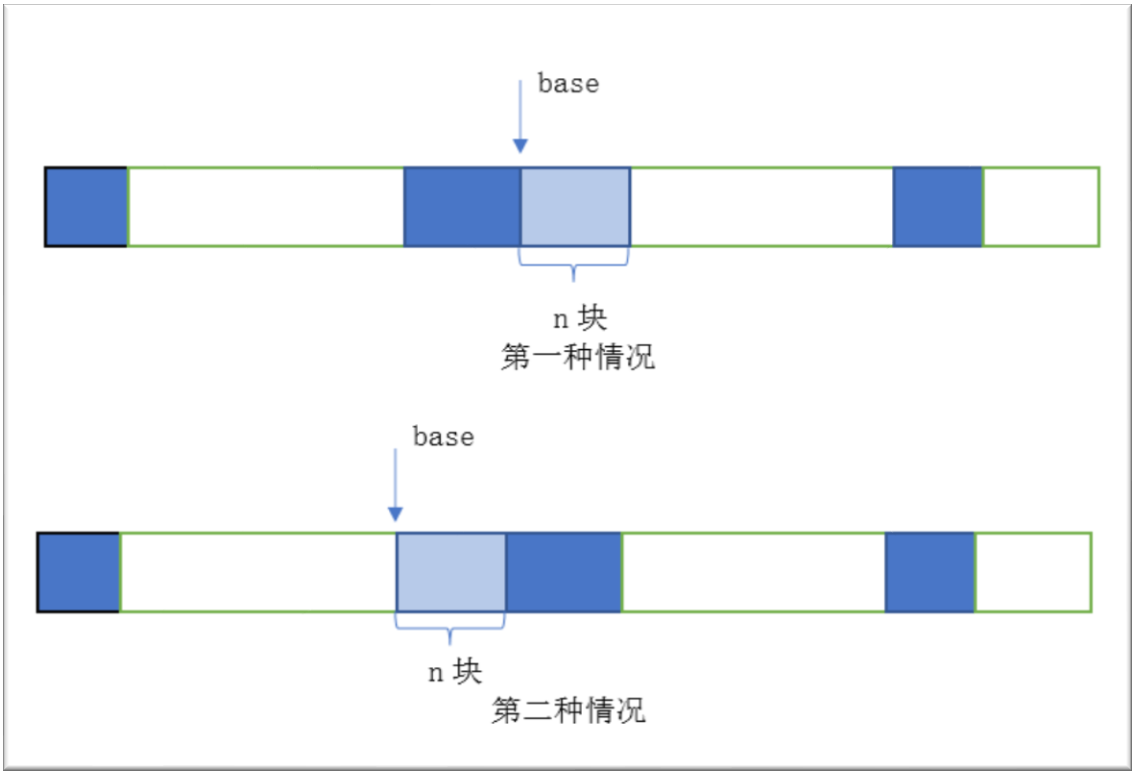

图 4 两种需要合并的情况

4、first-fit 物理内存分配算法相对来说比较简单，能改进的空间相对有限。

练习 2

1、做练习 2 和练习 3 都需要对页目录表页表以及虚拟地址之间的关系有清晰的认识。首先假设有一个 32 位虚拟地址 la，la 可以直观的分成三个部分，前 10 位是用于查找页目录表的索引，中间 10 位是用于查找页表的索引，最后 12 位用于查找物理页内的偏移量。页目录表的起始物理地址放在 CR3 寄存器中，首先通过 la 的前 10 位索引找到对应的 32 位的页目录表项，该表项前 20 位是对应的页表的起始物理地址，后 12 位是状态位。有了对应页表的起始物理地址，再根据 la 的中间 10 位找到对应的 32 位页表项（这是 gte_pte 函数的功能），该表项前 20 位是对应的物理页的起始地址，后 12 位是状态位。最后根据 la 的后 12 位找到物理页内的 offset（真实物理页内是按照字节寻址的方式，所以需要 12 位偏移量），即完成一次寻址工作。

2、get_pte 是实现寻找对应的页表项的函数。这里需要注意的是我们并不是一开始就为页表分配物理页，而是等到需要的时候再添加对应的页表，与这些操作相关的量是 PTE_P 和 create。get_pte 函数实现如下。

```
    pde_t *pdep = &pgdir[PDX(la)]; //find pde
    if (!(*pdep & PTE_P))//pde is not existence
    {
        struct Page* page = alloc_page();
        if (!create || page == NULL) //if create is not 0 or page is 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)];//return pte,PDE_ADDR(*pdep) is physical address
}
```

图 5 get_pte 函数实现

3、页访问异常，CPU 在当前内核栈保存当前被打断的程序现场，即依次压入当前被打断程序使用的 EFLAGS，CS，EIP，errorCode，并将引起页访问异常的线性地址装到 CR2 寄存器中；由于页访问异常的中断号是 0xE，CPU 把异常中断号 0xE 对应的中断服务例程的地址(vectors.S 中的标号 vector14 处)加载到 CS 和 EIP 寄存器中，开始执行中断服务例程。

练习 3

1、 练习 3 要做的就是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。这个练习没有什么难点，要注意的就是如果该页表项对应的页只被引用了一次，可以直接释放此页。page_remove_pte 函数实现如下。

```
    if (*ptep & PTE_P)  //判断页表中该表项是否存在
    {
        struct Page *page = pte2page(*ptep);
        if (page_ref_dec(page) == 0) //判断是否只被引用了一次
        {
            free_page(page);//如果只被引用了一次，那么可以释放掉此页
        }
        *ptep = 0;//如果被多次引用，则不能释放此页，只用释放二级页表的表项
        tlb_invalidate(pgdir, la);//更新页表
    }
}
```

图 6 page_remove_pte 函数

2、页的分配是以物理页为单位的，其地址要求按照页大小，即 4096 字节对齐。PDE 和 PTE 的高 20 位用于保存对应页表和页的基地址，低 12 位可以用作表项的标志位，对表项的属性进行说明，如是否存在，是否可写以及访问权限等。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状态；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。