Lab6: RISC-V 动态内存分配与缺页异常处理

注：本次实验由于需要考试，个人精力不够的问题我只完成了第一个目标也就是buddy system和slub的实现，最终实现了kmalloc()的完整接口，并对lab5中使用的kalloc()进行了替换，验证了当前实现的内存分配系统的可用性，最终的成果是讲lab5中内存分配全部替换掉了，同时能够争产不过跑lab5的用户代码，实现一样的结果。由于本次没有完成全部实验，但我认为在我完成的部分都实现的非常好，希望老师和助教能够酌情给分，谢谢老师了！

1 实验简介

在充分理解前序实验中RISC-V地址空间映射机制与任务调度管理的前提下，进一步了解与**动态**内存管理相关的重要结构，实现基本的内存管理算法，并最终能够较为**综合**地实现对进程的内存管理与调度管理。

2 实验目标

- 目标一：了解 **Buddy System** 和 **Slub Allocator** 物理内存管理机制的实现原理，并用 **Buddy System** 配合 **Slub Allocator** 管理和分配物理内存，最终实现统一的内存分配/释放接口:**kmalloc/kfree**。
- 目标二：在**mm_struct**中补充实现**vm_area_struct**数据结构的支持，并结合后续**mmap**等系统调用实现对进程**多区域**虚拟内存的管理。
- 目标三：实现**222: mmap, 215: munmap, 226: mprotect**系统调用。
- 目标四：在 **Lab5** 实现用户态程序的基础上，添加缺页异常处理**Page Fault Handler**，并在分配物理页或物理内存区域的同时更新页表。
- 目标五：综合上述实现，为进程加入 **fork** 机制，能够支持创建新的用户态进程，并测试运行。

3 实验环境

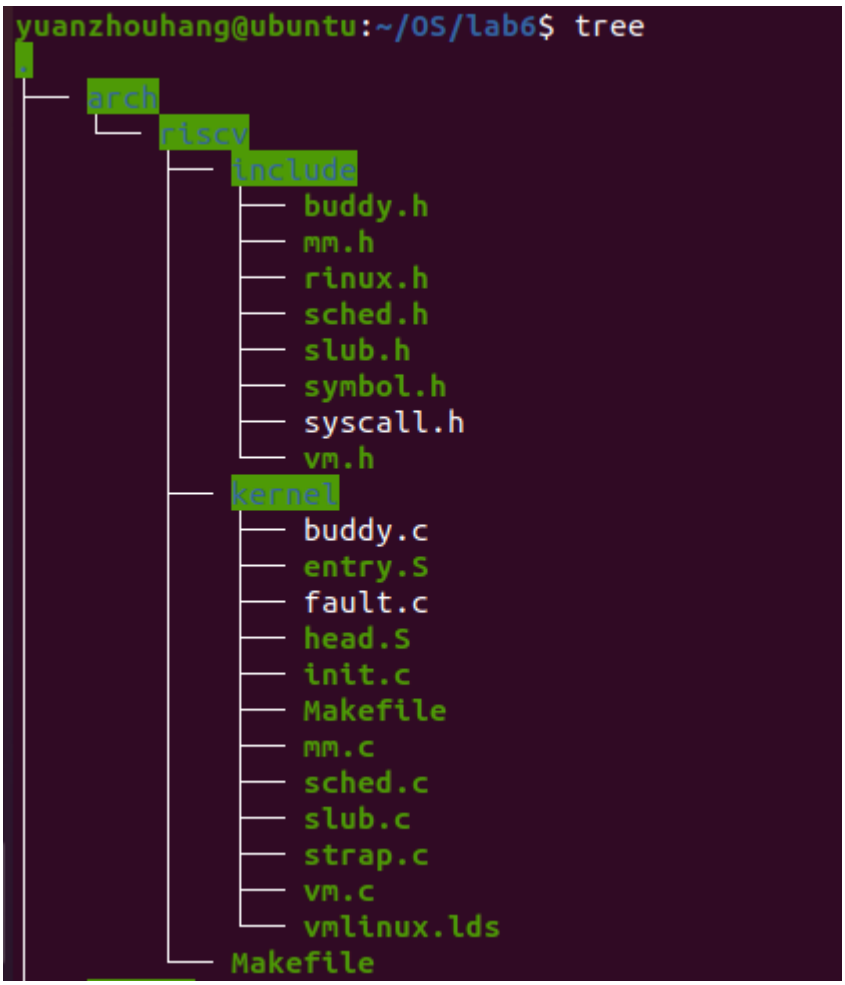
Docker Image

4 实验背景知识

略

5 实验步骤

5.1 环境搭建



5.2 管理空闲内存空间，在buddy.c中实现buddy system

在这部分我将分函数讲解buddy system的构建，具体的抽象原理报告上给出的已经很详细了我这里就不再重复了，我这里主要讲我的实现方法。

init_buddy_system

```
```

init_buddy_system的作用是创建一个buddy system实体，作为全局物理内存的管理系统，之后程序中所有申请空余物理内存都将从这块我们划定好的归属于buddy system的物理内存中分配，且都要通过buddy system的分配方法来分配，这里后面会演示到。

buddy system实际上就是由一个整数size和一个数组bitmap构成的，这个bitmap可以描述哪些页分配掉了，哪些页可以被分配这样的信息。我们初始化buddy system实际上做的就是分配这样一个buddy system。我为了方便，就讲buddy system设置为了一个全局变量，数组也放置在了全局变量域。

我的物理地址是从PHY_START(0x8000,0000)~PHY_END(0x8100,0000)，kernel位于PHY_START~_end

之后初始化要做的便是给bitmap各个位赋值。复制完之后需要将kernel代码所处的部分分配出来，这样做是为了保护kernel位置的内存，因为kernel部分已经由kernel的代码和栈等内容，并不是空闲的，如果不把这部分空间设置为已分配则后面申请空余空间的时候可能将这部分地址返回，这样便会破坏当前的kernel代码造成错误。而实际上我们只需要利用alloc_pages()函数分配一个kernel长度的空间出来即可完成对kernel的保护，因为当前bitmap是完整的，因此此时分配空间，不管分配多大，根据我的分配算法，其开始地址一定是从0x8000,0000开始的。

alloc_pages

```
void *alloc_pages(int npages){
    int page_size = 1;
    while(page_size < npages){
        page_size *= 2;
    }

    void* addr;
    int index = 0;

//    printf("page_size:%d\n", page_size);
    for(index=0; index < buddy_system.size*2 - 1; index++){
       if((buddy_system.bitmap[index] == page_size) &&
((index+1)*buddy_system.bitmap[index] >= buddy_system.size)){
//     printf("index: %d\n", index);
           buddy_system.bitmap[index] = 0;
           int i = (index+1)/2 - 1;
           int pre_i  = index;
           int pre_i_sibling = (index%2 == 0)?(index-1):(index+1);
           int son1, son2;
           while(i >= 0){
            if(buddy_system.bitmap[i] == 0) break;
            son1 = buddy_system.bitmap[pre_i];
            son2 = buddy_system.bitmap[pre_i_sibling];
            buddy_system.bitmap[i] = (son1 < son2)?son2:son1;
            pre_i = i;
            pre_i_sibling = (i%2 == 0)?(i-1):(i+1);
            i = (i+1)/2 - 1;
           }
           if(i >= 0)   continue;
           break;
       }
    }


    if(index >= buddy_system.size*2 - 1 || index < 0){
        panic("Run out of free physic pages");
    }

    //根据index和page_size计算出地址
    int same_level_start_index = buddy_system.size/page_size - 1;
    addr = PHY_START + PAGE_SIZE * (index - same_level_start_index) * page_size;

    printf("[S] Buddy allocate addr: 0x%lx, page_num: %d, size: %d\n", addr, (long int)
(addr-PHY_START)/PAGE_SIZE, page_size);
    return (void*)PA2VA(addr);
}
```

alloc_pages的功能是，根据所给出的所要分配的大小，在bitmap中找到能够容纳这个大小的连续的页，将其首地址传给调用者。由于要查询bitmap，而bitmap是一个二叉树，因此这里原则上由多种方式，最快速也最好写的方式肯定是使用递归函数查询bitmap的每一层，但这会产生一个问题，就是使用递归函数会对stack造成很大的负担，因为递归的每次调用都会往栈中保存大量的数据，我认为在底层使用递归是非常不安全的，因此我没有使用递归，而是使用了大循环遍历bitmap，下面我将详细讲一下这个流程。

首先我将要分配的大小扩展乘2的幂次方形式，之后遍历bitmap在该大小的层找到一个满的块返回，在找到该块之后要按路径返回其父节点，将父节点的值按照(如果两个节点都是满的，则父是其和，否则父为其中更大的那个)的逻辑修改好。注意，这里不考虑不同块但连续的地址，即如果有两个块，他们包含连续的地址，但他们处于bitmao的不同分支中，那么算法也是不考虑的。这样做的缺点是可能产生一些内部碎片，但优点是算法简单且不容易错，并且在我们的实验中大量的使用了1个page的小内存区域，内部碎片大多都可以被填掉。这个算法只在本实验中使用。

如果在bitmap中没能找到我们想要的大小的内存区域，就使用panic报run of的错，之后算出地址返回即可。

free_pages

```
void free_pages(void* addr){
    int index = ((long int)addr - PHY_START)/PAGE_SIZE + buddy_system.size - 1;

    while(index >= 0){
        if(buddy_system.bitmap[index] == 0){
            int value = 1;
        while(value * (index+1) < buddy_system.size){
            value *= 2;
        }
        buddy_system.bitmap[index] = value;
        int i = (index+1)/2 - 1;
        int pre_i  = index;
        int pre_i_sibling = (index%2 == 0)?(index-1):(index+1);
        int son1, son2;
        while(i >= 0){
            son1 = buddy_system.bitmap[pre_i];
            son2 = buddy_system.bitmap[pre_i_sibling];
            buddy_system.bitmap[i] = (son1 < son2)?son2:son1;
            pre_i = i;
            pre_i_sibling = (i%2 == 0)?(i-1):(i+1);
            i = (i+1)/2 - 1;
           }
           break;
        }
        if(index%2 != 0)
            panic("Invalid physic page address to free");
        index = (index+1)/2 -1;
    }

    if(index < 0){
        panic("Invalid physic page address to free");
    }
}
```

free_pages跟alloc_pages有很多相似之处，也是要遍历其父节点去修改父节点的值，同时由于给出的是地址而不是页号，因此概要根据这个地址去找这段分配的空间的大小(即找bitmap中的0)，这个算法逻辑在这里就不详细讲了，因为实验中也没怎么用到。

5.3 在 slub.c 中实现 slub 内存动态分配算法

这一小节我将讲一讲slub.c的逻辑。

简单的来说，slub可以分配比一个page更小的内存，因此它可以用来解决使用buddy带来的内部碎片的问题。本实验的slub的流程是这样的：首先通过slub_init()函数初始化slub系统，slub系统中给每种大小(8-2048)都分配了一个kem_cache结构，这个结构实际上是通过buddy系统分配出的4个page大小的空间，在slub系统初始化好之后，这些空间将是空的，之后每一次使用slub分配空间，例如想用slub分配一个256bytes的空间，slub将会从256对应的那4个page的空间中取出256bytes，原理其实非常简单。

5.4 实现统一内存分配接口

由于其他内容slub.c都已经实现好，我们只需要看懂然后使用即可。唯二需要我们写的函数为kmalloc和kfree（因为重名我在这里改为了kkfree）

kmalloc

```
void *kmalloc(size_t size)
{
    int objindex;
    void *p;

    if(size == 0)
            return NULL;

        // size 若在 kmem_cache_objsize 所提供的范围之内，则使用 slub allocator 来分配内存
        for(objindex = 0; objindex < NR_PARTIAL; objindex ++){
        // YOUR CODE HERE
        if(size <= kmem_cache_objsize[objindex]){
        //    struct kmem_cache* KC = kmem_cache_create(kmem_cache_name[objindex],
kmem_cache_objsize[objindex], 8, 0, NULL);
            p = kmem_cache_alloc(slub_allocator[objindex]);
            memset((char*)(p), 0xf, size);
            break;
        }
    }

        // size 若不在 kmem_cache_objsize 范围之内，则使用 buddy system 来分配内存
    if(objindex >= NR_PARTIAL){
            // YOUR CODE HERE
            p = VA2PA(alloc_pages(size/PAGE_SIZE));
            set_page_attr(PA2VA(p), (size-1) / PAGE_SIZE, PAGE_BUDDY);
            memset((char*)(PA2VA(p)), 0xf, size);
    }

        return p;

}
```

其实这两个函数都类似handler，我们只需要判断需要分配的大小在2048的哪边即可，如果在2048以下，则适合使用slub来分配，于是调用slub的alloc函数。否则便只能用buddy分配，于是调用buddy的alloc函数，同时还要对其做page的attribute的修改(slub的page attribute在前面函数里就有修改，这里不需要重复做)。在拿到地址之后将地址位置清空就可以用了。

kfree

```
void kkfree(const void *addr)
{
    struct page *page;

    if(addr == NULL)
        return;

    // 获得地址所在页的属性
    // YOUR CODE HERE
    page->flags = ADDR_TO_PAGE(addr)->flags;

    // 判断当前页面属性
    if(page->flags == PAGE_BUDDY){
        // YOUR CODE HERE
        free_pages(addr);
        clear_page_attr(ADDR_TO_PAGE(addr)->header);

    } else if(page->flags == PAGE_SLUB){
        // YOUR CODE HERE
        kmem_cache_free(addr);
    }

    return;

}
```

kfree跟kmalloc非常类似，区别就在于判断使用slub还是buddy是使用page的attribute，这里就不详细解释了。

做到这里目标一就结束了，如下是目标一的一些结果

这个实验我是分目标写的，而且目标一可以单独测试，因此我在这里进行了测试。我们首先要把之前使用mm.c里面连续分配的内存改成现在使用buddy system和slub分配的内存。具体做法是将之前所有使用kalloc和alloc_page(mm.c中定义的方法)改成kmalloc和alloc_pages。需要注意的是kalloc每次都是一页，而kmalloc可以调整页数。然而，在页表分配的时候绝不能修改分配大小，我在调试的时候就卡在这里了很久，因为页表都是一页一页的，页表所需页的大小与创建映射的大小没有关系。

如下是实验五修改为buddy system之后的实验结果：

```
[init.c:31 start_kernel] Finish device_init
[:34 start_kernel] Finish kinit
[S] Buddy allocate addr: 0x0000000080000000, page_num: 0, size: 32
[:37 start_kernel] Finish init_buddy_system
[S] Buddy allocate addr: 0x0000000080040000, page_num: 64, size: 64
[S] Buddy allocate addr: 0x0000000080020000, page_num: 32, size: 16
[S] Create New cache: name:slub-objectsize-8            size: 8, align: 8
[S] Buddy allocate addr: 0x0000000080030000, page_num: 48, size: 4
[S] Create New cache: name:slub-objectsize-16           size: 16, align: 8
[S] Buddy allocate addr: 0x0000000080034000, page_num: 52, size: 4
[S] Create New cache: name:slub-objectsize-32           size: 32, align: 8
[S] Buddy allocate addr: 0x0000000080038000, page_num: 56, size: 4
[S] Create New cache: name:slub-objectsize-64           size: 64, align: 8
[S] Buddy allocate addr: 0x000000008003c000, page_num: 60, size: 4
[S] Create New cache: name:slub-objectsize-128          size: 128, align: 8
[S] Buddy allocate addr: 0x0000000080080000, page_num: 128, size: 4
[S] Create New cache: name:slub-objectsize-256          size: 256, align: 8
[S] Buddy allocate addr: 0x0000000080084000, page_num: 132, size: 4
[S] Create New cache: name:slub-objectsize-512          size: 512, align: 8
[S] Buddy allocate addr: 0x0000000080088000, page_num: 136, size: 4
[S] Create New cache: name:slub-objectsize-1024         size: 1024, align: 8
[S] Buddy allocate addr: 0x000000008008c000, page_num: 140, size: 4
[S] Create New cache: name:slub-objectsize-2048         size: 2048, align: 8
[S] Buddy allocate addr: 0x0000000080090000, page_num: 144, size: 4
[:40 start_kernel] Finish slub_init
[S] Buddy allocate addr: 0x0000000080094000, page_num: 148, size: 1
[S] Buddy allocate addr: 0x0000000080095000, page_num: 149, size: 1
[S] Buddy allocate addr: 0x0000000080096000, page_num: 150, size: 1
[S] Buddy allocate addr: 0x0000000080097000, page_num: 151, size: 1
[S] Buddy allocate addr: 0x0000000080098000, page_num: 152, size: 1
[S] Buddy allocate addr: 0x0000000080099000, page_num: 153, size: 1
[S] Buddy allocate addr: 0x000000008009a000, page_num: 154, size: 1
[S] Buddy allocate addr: 0x000000008009b000, page_num: 155, size: 1
[S] Buddy allocate addr: 0x000000008009c000, page_num: 156, size: 1
[S] Buddy allocate addr: 0x000000008009d000, page_num: 157, size: 1
[S] Buddy allocate addr: 0x000000008009e000, page_num: 158, size: 1
[S] Buddy allocate addr: 0x000000008009f000, page_num: 159, size: 1
[S] Buddy allocate addr: 0x00000000800a0000, page_num: 160, size: 1
[:43 start_kernel] Finish kvminit
rinux start...
task init...
```

```
task init...
[S] Buddy allocate addr: 0x00000000800a1000, page_num: 161, size: 1
[S] Buddy allocate addr: 0x00000000800a2000, page_num: 162, size: 1
[S] Buddy allocate addr: 0x00000000800a3000, page_num: 163, size: 1
[S] Buddy allocate addr: 0x00000000800a4000, page_num: 164, size: 1
[S] Buddy allocate addr: 0x00000000800a5000, page_num: 165, size: 1
[S] Buddy allocate addr: 0x00000000800a6000, page_num: 166, size: 1
[S] Buddy allocate addr: 0x00000000800a7000, page_num: 167, size: 1
[S] Buddy allocate addr: 0x00000000800a8000, page_num: 168, size: 1
[S] Buddy allocate addr: 0x00000000800a9000, page_num: 169, size: 1
[S] Buddy allocate addr: 0x00000000800aa000, page_num: 170, size: 1
[S] Buddy allocate addr: 0x00000000800ab000, page_num: 171, size: 1
[S] Buddy allocate addr: 0x00000000800ac000, page_num: 172, size: 1
[S] Buddy allocate addr: 0x00000000800ad000, page_num: 173, size: 1
[S] Buddy allocate addr: 0x00000000800ae000, page_num: 174, size: 1
[S] Buddy allocate addr: 0x00000000800af000, page_num: 175, size: 1
[S] Buddy allocate addr: 0x00000000800b0000, page_num: 176, size: 1
[S] Buddy allocate addr: 0x00000000800b1000, page_num: 177, size: 1
[S] Buddy allocate addr: 0x00000000800b2000, page_num: 178, size: 1
[S] Buddy allocate addr: 0x00000000800b3000, page_num: 179, size: 1
[S] Buddy allocate addr: 0x00000000800b4000, page_num: 180, size: 1
[S] Buddy allocate addr: 0x00000000800b5000, page_num: 181, size: 1
[S] Buddy allocate addr: 0x00000000800b6000, page_num: 182, size: 1
[S] Buddy allocate addr: 0x00000000800b7000, page_num: 183, size: 1
[S] Buddy allocate addr: 0x00000000800b8000, page_num: 184, size: 1
[S] Buddy allocate addr: 0x00000000800b9000, page_num: 185, size: 1
[S] Buddy allocate addr: 0x00000000800ba000, page_num: 186, size: 1
[S] Buddy allocate addr: 0x00000000800bb000, page_num: 187, size: 1
[S] Buddy allocate addr: 0x00000000800bc000, page_num: 188, size: 1
[S] Buddy allocate addr: 0x00000000800bd000, page_num: 189, size: 1
[S] Buddy allocate addr: 0x00000000800be000, page_num: 190, size: 1
[S] Buddy allocate addr: 0x00000000800bf000, page_num: 191, size: 1
[S] Buddy allocate addr: 0x00000000800c0000, page_num: 192, size: 1
[S] Buddy allocate addr: 0x00000000800c1000, page_num: 193, size: 1
[S] Buddy allocate addr: 0x00000000800c2000, page_num: 194, size: 1
[S] Buddy allocate addr: 0x00000000800c3000, page_num: 195, size: 1
[S] Buddy allocate addr: 0x00000000800c4000, page_num: 196, size: 1
[S] Buddy allocate addr: 0x00000000800c5000, page_num: 197, size: 1
[S] Buddy allocate addr: 0x00000000800c6000, page_num: 198, size: 1
[S] Buddy allocate addr: 0x00000000800c7000, page_num: 199, size: 1
[S] Buddy allocate addr: 0x00000000800c8000, page_num: 200, size: 1
[S] Buddy allocate addr: 0x00000000800c9000, page_num: 201, size: 1
[PID = 1] Process Create Successfully! counter = 1
```

```
[S] Buddy allocate addr: 0x0000000080104000, page_num: 260, size: 1
[S] Buddy allocate addr: 0x0000000080105000, page_num: 261, size: 1
[PID = 4] Process Create Successfully! counter = 4
[:52 start_kernel] Finish task_init
[PID = 0] Context Calculation: counter = 0
[!] Switch from task 0 [task struct: 0xffffffe0000a1000, sp: 0xffffffe0000a2000] to task 1 [task struct: 0xffffffe0000b6000, sp: 0xffffffe0000b7000], prio: 5, counter: 1
[User] pid: 1, sp is ffffffdf7ffffd0
[PID = 1] Context Calculation: counter = 1
[!] Switch from task 1 [task struct: 0xffffffe0000b6000, sp: 0xffffffe0000b7000] to task 4 [task struct: 0xffffffe0000f2000, sp: 0xffffffe0000f3000], prio: 5, counter: 4
[User] pid: 4, sp is ffffffdf7ffffd0
[PID = 4] Context Calculation: counter = 4
[PID = 4] Context Calculation: counter = 3
[PID = 4] Context Calculation: counter = 2
[PID = 4] Context Calculation: counter = 1
[!] Switch from task 4 [task struct: 0xffffffe0000f2000, sp: 0xffffffe0000f3000] to task 2 [task struct: 0xffffffe0000ca000, sp: 0xffffffe0000cb000], prio: 5, counter: 4
[User] pid: 2, sp is ffffffdf7ffffd0
[PID = 2] Context Calculation: counter = 4
[PID = 2] Context Calculation: counter = 3
[PID = 2] Context Calculation: counter = 2
[PID = 2] Context Calculation: counter = 1
[!] Switch from task 2 [task struct: 0xffffffe0000ca000, sp: 0xffffffe0000cb000] to task 3 [task struct: 0xffffffe0000de000, sp: 0xffffffe0000df000], prio: 5, counter: 5
[User] pid: 3, sp is ffffffdf7ffffd0
[PID = 3] Context Calculation: counter = 5
[PID = 3] Context Calculation: counter = 4
[PID = 3] Context Calculation: counter = 3
[PID = 3] Context Calculation: counter = 2
[User] pid: 3, sp is ffffffdf7ffffd0
[PID = 3] Context Calculation: counter = 1
[PID = 1] Reset counter = 5
[PID = 2] Reset counter = 5
[PID = 3] Reset counter = 5
[PID = 4] Reset counter = 2
[!] Switch from task 3 [task struct: 0xffffffe0000de000, sp: 0xffffffe0000df000] to task 4 [task struct: 0xffffffe0000f2000, sp: 0xffffffe0000f3000], prio: 5, counter: 2
[User] pid: 4, sp is ffffffdf7ffffd0
[PID = 4] Context Calculation: counter = 2
[PID = 4] Context Calculation: counter = 1
[!] Switch from task 4 [task struct: 0xffffffe0000f2000, sp: 0xffffffe0000f3000] to task 3 [task struct: 0xffffffe0000de000, sp: 0xffffffe0000dee18], prio: 5, counter: 5
[User] pid: 3, sp is ffffffdf7ffffa0
[PID = 3] Context Calculation: counter = 5
```

6 心得体会

以上就是这次实验我实现的内容了，可以看到其实还是比较完整的，但由于时间的关系没办法做更多了(再做就没时间复习了)。其实我觉得这个实验设计的还是蛮好的，包括我没有做的部分我也看了并且理解了，后面的部分主要是让我们在buddy的基础上实现一个简单的demand paging的功能。

本次实验设计的比较好的地方在于kmalloc()这种接口性函数，在文档中给出了模板，在这个模板之上加上我对slub.c中其他部分代码的阅读让我很快的理解了kmalloc()这个函数它的作用是什么，以及它需要我干什么。这个方法就可以很好的避免一个问题：在开始做实验的时候读文档理解实验目的非常缓慢，并且如果理解错很容易走上歪路。提供代码框架和模板就可以让学生快速上手，这样也能加快一下实验的进度，让同学们都能把lab6做完。我觉得这种形式的提示可以更大幅度用在之前的实验中。

另外，虽然我们的实验真的很难，但我也确实在做完这些实验之后觉得自己对操作系统更加理解了，这也确实是高难度实验带来的一大好处，我觉得今后的实验可以在降低难度的基础上扩大范围一些，因为我们这次实验毕竟还是有很多课堂上的知识没有涉及，可以更多元化一点(只要难度降下来就好，可以利用代码框架等一些方法来降难度)。

总之，很感谢老师设计的实验，也很感谢助教辛苦的批改，尽管过程非常曲折，但我也算完成了这门课的实验部分学习，也在这门课上收获多多，非常感谢老师们！