Lab2 设计描述

心得

这次 lab 虽然内容不如上一次丰富，但是虚拟地址的转换和空闲链表的管理都是非常繁琐的部分，自己多次不得不停下码代码的步伐去查各种宏或是在草稿纸上涂涂写写，虽然多花了很多时间但是增强了自己对内存管理的理解。这次的代码逻辑很清晰，注释基本上都在教逻辑，唉，但是总感觉自己阅读英文还是太吃力了。

Part1 physical page management

每个 lab 的入门都是最困难的，这个也不例外，好在这个系列的 lab 有详尽的说明与注释的指导，物理地址管理相对比较好懂，在理解了内存分布之后，只要熟悉一下几个重要的宏，在该进行转化的地方头脑保持清晰就可以了。

exercise1

首先是 boot_alloc 函数，这里已经把初始化的代码给出，因此自己只要模仿着写，每次将 nextfree 指针增长 n，另外 ROUNDUP 宏可以做到按 PageSize 对齐。

```
result = nextfree;
nextfree = ROUNDUP(nextfree+n, PGSIZE);
cprintf("boot alloc %x at %x\n",n,result);
return result;
```

接下来是 mem_init 函数，目前要求只做这一块，比较坑的是上面那个 panic 函数一开始没有去掉，直到使用 gdb 调试发现之后的代码根本没有转成汇编，才意识到这个错误。这块本身也不难，是将页表占用的空间分配出来，利用了之前实现的 boot_alloc 函数。

```
//////////////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct Page's and store it in 'pages'.
// The kernel uses this array to keep track of physical pages: for
// each physical page, there is a corresponding struct Page in this
// array.  'npages' is the number of physical pages in memory.
// Your code goes here:

pages = boot_alloc(npages*sizeof(struct Page));
```

然后是 page_init，这里遇到了一些小小的麻烦，经过一阵纠结后，我的思路确定为之前 mem_init 中所有分配出来的内存都应该跳过，刚好这一段也是连续的，用了一个简单的加法来确定上界。

```
        size_t i;
        extern char end[];
        uint32_t upp = (uint32_t)ROUNDUP((char *)end, PGSIZE) - KERNBASE + PGSIZE + ROUNDUP(npages * sizeof(struct
 Page), PGSIZE);
        cprintf("end: %x npages: %d sizeof struct Page: %x PGSIZE: %x IOPHYSMEM: %x\n",end,npages,sizeof(struct Pa
ge),PGSIZE,IOPHYSMEM);
        for (i = 0; i < npages; i++) {
                if (i == 0) continue;
                if (i >= PGNUM(IOPHYSMEM) &&
                        i <= PGNUM(upp))
                        continue;
                pages[i].pp_ref = 0;
                pages[i].pp_link = page_free_list;
                page_free_list = &pages[i];
        }
        chunk_list = NULL;
}
```

然而 asset(npage_extmem>0)一直报错，经分析我只能认为是上界的确定有问题，于是把值都打印出来看。

```
boot alloc 1000 at f0118000
boot alloc 207f8 at f0119000
end: f011798c npages: 16639 sizeof struct Page: 8 PGSIZE: 1000 IOPHYSMEM: a0000
```

小小的按了一下计算器，16639*8 确实是 0x207f8，问题出在后面，下界只有 a0000，确实可以看出上下界差太远了，这时我才意识到宏的使用方法不正确，上界减去 KERNBASE 后，成功通过测试。

```
struct Page *
page_alloc(int alloc_flags)
{
        if (page_free_list == NULL)
                return NULL;
        struct Page *ans = page_free_list;
        page_free_list = page_free_list->pp_link;
        if (alloc_flags & ALLOC_ZERO)
                memset(page2kva(ans),'\0',PGSIZE);
        return ans;
}
```

```
void
page_free(struct Page *pp)
{
        // Fill this function in
        pp->pp_link = page_free_list;
        page_free_list = pp;
}
```

page_alloc 和 page_free 函数让我想起了 ics 的 malloc lab……不过逻辑按注释提示的做就行了，alloc 部分直接从空闲链表中取出一块，free 则是直接将这块放入空闲链表。

Exercise2

```
struct Page *
page_alloc_npages(int alloc_flags, int n)
{
        // Fill this function
        if (n <= 0) return NULL;
        struct Page *ans = page_free_list;
        struct Page *prev = NULL;
        cprintf("ans:\n");
        while (ans != NULL)
        {
//              cprintf("%d\n",page2pa(ans));
                if (my_check_continuous(ans,n)) break;
                prev = ans;
                ans = ans->pp_link;
        }
        if (ans == NULL)
        {
                cprintf("haven't find continuous %d pages!\n",n);
                return NULL;
        }
        struct Page *iter;
        int i;
        cprintf("find these %d pages: ",n);
        for (i = 0, iter = ans; i < n; i++, iter = iter->pp_link)
                cprintf("%d ",page2pa(iter));
        cprintf("\n");
```

```
        struct Page *ans2 = ans;
        for (i = 0, iter = ans; i < n; i++)
        {
                if (iter == NULL)
                        return NULL;

                if (alloc_flags & ALLOC_ZERO)
                        memset(page2kva(iter),'\0',PGSIZE);
                struct Page *lst = iter;
                iter = iter->pp_link;
                if (i == 0){
                        ans2 = ans;
                        ans2->pp_link = NULL;
                }else{
                        lst->pp_link = ans2;
                        ans2 = lst;
                }
        }
        prev->pp_link = iter;
        cprintf("answer is these %d pages: ",n);
        for (i = 0, iter = ans2; i < n; i++, iter = iter->pp_link)
                cprintf("%d ",page2pa(iter));
        cprintf("\n");

//      if (check_continuous(ans,n) == 0)
//              return NULL;
        return ans2;
}
```

这一部分个人感觉很麻烦……所以花了大量的输出来调试，刚开始的时候以为很简单，就是多次调用 alloc 函数的感觉，然后才发现，首当其冲的一个难点是“连续的物理地址”，本以为调用自带的 check_continuous 函数可以轻松解决，这时又发现它对于降序不能判断……光加一个降序版本也不能解决问题，还得将答案也翻转一遍，不然也无法通过测试，这里我有点被指针操作恶心到了，只能打印翻转前后的值对着调，艰难通过。

```
find these 4 pages: 4182016 4177920 4173824 4169728
answer is these 4 pages: 4177920 4182016 -589299712 -555835392
```

这里的一个典型错误是调整新链表的值的时候修改了迭代器的值。另外，如果链表增加 previous 指针，应该会好调的多……

```
int
page_free_npages(struct Page *pp, int n)
{
        if (check_continuous(pp,n) == 0)
                return -1;
        int i;
        for (i = 0; i < n; i++)
        {
                if (pp == NULL)
                        return -1;
                struct Page *next = pp->pp_link;
                pp->pp_link = chunk_list;
                chunk_list = pp;
                pp = next;
        }
        return 0;
}
```

紧接着的 free 函数就简单了许多，一遍循环将所有页指针加入 chunk_list 即可。

Exercise2(2)

```
struct Page *
page_realloc_npages(struct Page *pp, int old_n, int new_n)
{
        if (old_n == new_n)
                return pp;
        if (old_n > new_n)
        {
                page_free_npages(pp+new_n,old_n-new_n);
                pp[new_n-1].pp_link = NULL;
                return pp;
        }
        int i;
        bool p = 1;
        for (i = old_n; i < new_n; i++)
        {
                if (pp[i].pp_ref != 0)
                {
                        p = 0;
                        break;
                }
        }
        if (p == 0)
        {
                struct Page *newpp = page_alloc_npages(ALLOC_ZERO,new_n);
                memmove(page2kva(newpp),page2kva(pp),old_n*PGSIZE);
                page_free_npages(pp,old_n);
                pp = newpp;
        }else
        {
                pp[old_n-1].pp_link = pp+old_n;
                struct Page *iter = page_free_list;
                struct Page *lst = iter;
                while (iter != NULL)
                {
                        if (iter >= pp+old_n && iter <= pp+new_n)
                        {
                                if (iter == page_free_list)
                                        page_free_list = iter->pp_link;
                                else{
                                        lst->pp_link = iter->pp_link;
                                }
                        }else{
                                lst = iter;
                        }
                        iter = iter->pp_link;
                }
                for (i = old_n; i < new_n; i++)
                {
                        pp[i].pp_link = pp+i+1;
                }
                memset(page2kva(pp+old_n),0,(new_n-old_n)*PGSIZE);
        }
        return pp;
}
```

由于有了前两次辛苦调试的经验，测试数据也不强，这次过的出奇的顺利，思路就如题面描述一般，如果新值更小，把多余的 page free 掉就可以，如果新值更大，那么就要考察

多出的部分，如果这部分可以直接被利用，那么手动对其进行设置（pp[i].pp_link = pp+i+1），否则整个把之前的 free 掉，alloc 新的长度。

这里要吐槽的部分就是为了尽量减少暴力的 alloc/free，题目建议我们能“增长”就“增长”，然而为了在空闲链表中去除这些新增长的区域，不得不遍历整个空闲链表，效率无疑是很低下的。

Part2

Exercise3

这里提到详细阅读 80386 reference manual 的 5、6 两章节，于是便去读了。

虚拟地址的翻译要经过段地址再到物理地址的两重翻译，由于下面还提到了在这个 lab 里只需要关注页变换，这里主要详细阅读了 5.2 节。

一个页包含 4K 内容，段地址结构由于图刷不出，我自己去网上查了一下，结合文章中的文字说明，它分为 3 块：前 10 位（dir）用来从页表目录中找到对应页表的索引，，中间 10 位（page）用来在页表中找到目标地址的索引，加上后 12 位（offset）就得到了目标物理地址。接着是 6.4 中描述的页级别的 protection，主要提出了两点：可分配域的限制以及类型检查，大意是说给每个页设置一位用来标识是否需要管理员权限运行，用户权限下可运行的页具体分为可读写和不可写两类。

下文还提到 JOS 中你可以直接通过 KADDR(pa)，即加上 0xf0000000 来实现物理地址到虚拟地址的转换，反之，内核全局变量和通过 boot_alloc()出的虚拟地址也可以直接这样转换为物理地址。

Exercise4

接下来进入代码部分，个人感觉这块逻辑虽然不难，根据注释一步步来就行了，就是实现起来还是有点纠结，常用的几个宏要熟练运用。值得一提的是之前提到的地址翻译和保护机制，后者用了几个位来 check 页表项的类型，如 PTE_P 表示是否存在，PTE_U 表示是否为用户模式等。

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        cprintf("pgdir_walk: va:%x pgdir[PDX(va)]:%x\n",va,pgdir[PDX(va)]);
        if (pgdir[PDX(va)] & PTE_P)
        {
                return PTX(va)+(pte_t*)(KADDR(PTE_ADDR(pgdir[PDX(va)])));
        }else{
                if (create == 0) return NULL;
                struct Page *newp = page_alloc(ALLOC_ZERO);
                if (newp == NULL) return NULL;
                newp->pp_ref++;
                pgdir[PDX(va)] = page2pa(newp) | PTE_P | PTE_W | PTE_U;
                return PTX(va)+(pte_t*)page2kva(newp);
        }
        return NULL;
}
```

首先是 pgdir_walk，一开始犯了个错误觉得返回的应该是物理地址，后来才意识到对于程序员而言代码里几乎用不到物理地址，Question1 让我意识到运行时程序里的取址都会经过翻译，因此这里的返回值自己修改了好几次。最后确定思路为，取页表目录中的对应项，由于直接存储的是物理地址，翻译成虚拟地址后再加上原地址的 page 部分，这里非常绕，老实说我没有全部理顺畅。

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
        pte_t *pte;
        int i;
        for (i = 0; i < size/PGSIZE; i++)
        {
                pte = pgdir_walk(pgdir,(void *)va+i*PGSIZE,1);
                *pte = pa | perm | PTE_P;
                pa += PGSIZE;
        }
}
```

这个是整段的映射操作，逻辑较为清晰，以页为单位将页表项赋值为对应物理地址就行了，这里更加明确所谓“物理地址”与程序本身逻辑几乎无关，可以视为程序的一种产出。

```
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
        cprintf("page_insert: pp:0x%x va:0x%x perm:0x%x\n",pp,va,perm);
        bool exist = 0;
        pte_t *pte = pgdir_walk(pgdir,va,1);
        cprintf("page_insert: pte:0x%x *pte:0x%x\n",pte,*pte);
        if (pte == NULL)
                return -E_NO_MEM;
        if (*pte & PTE_P)
        {
                struct Page *nowpp = page_lookup(pgdir,va,0);
                if (nowpp != pp)
                        page_remove(pgdir,va);
                else{
                        exist = 1;
                }
        }
        cprintf("page_insert: do not exist\n");
        *pte = page2pa(pp) | perm | PTE_P;
        if (exist == 0)
        {
                pp->pp_ref++;
        }
        return 0;
}
```

这个函数也相对麻烦，主要是容易使人犯错，过程中遇到这样的麻烦：

```
kernel panic at kern/pmap.c:974: assertion failed: *pgdir_walk(kern_pgdir, (void*) PGSIZE, 0) & PTE_U
```

经过 gdb 的调试，大概掌握了测试流程，即以相同的页、不同的 perm 值对一个页表项进行赋值，之前自己将判断写为如果出现重复就什么也不做，这样自然无法更新 perm 值，现在改成无论如何都更新值，问题解决。

```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
        pte_t *pte = pgdir_walk(pgdir,va,0);
        if (pte == NULL)
                return NULL;
        if (pte_store != 0)
                *pte_store = pte;
        if (*pte & PTE_P)
                return (struct Page*)pa2page(PTE_ADDR(*pte));
        else
                return NULL;
}
```

这个函数的逻辑是返回页表项对应的页，判断页表项有效则返回目标页。

```
void
page_remove(pde_t *pgdir, void *va)
{
        pte_t *pte = pgdir_walk(pgdir,va,0);
        if (pte == NULL || !(*pte & PTE_P))
        {
                cprintf("remove an empty pte!\n");
                return ;
        }
        struct Page *pp = page_lookup(pgdir,va,0);
        *pte = 0;
        page_decref(pp);
        tlb_invalidate(pgdir,va);
}
```

删除页表项还包括对应的页，使用已经提供的 page_decref 和 tlb_invalidate 函数可以减少工作量使逻辑更加清晰，注意判断删除空页的情况，至此第二部分代码写完。

Part3

Exercise6

```
size_t pgs = ROUNDUP(npages*sizeof(struct Page),PGSIZE);
boot_map_region(kern_pgdir,UPAGES,pgs,PADDR(pages),PTE_P | PTE_W);
//////////////////////////////////////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
// stack.  The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
//       the kernel overflows its stack, it will fault rather than
//       overwrite memory.  Known as a "guard page".
//     Permissions: kernel RW, user NONE
// Your code goes here:

boot_map_region(kern_pgdir,KSTACKTOP-KSTKSIZE,KSTKSIZE,PADDR(bootstack),PTE_P | PTE_W);
//////////////////////////////////////////////////////////////////////
// Map all of physical memory at KERNBASE.
// Ie.  the VA range [KERNBASE, 2^32) should map to
//      the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:

uint64_t range = (1ll<<32);
range = range - KERNBASE;
boot_map_region(kern_pgdir,KERNBASE,(uint32_t)range,0,PTE_P | PTE_W);

// Check that the initial page directory has been set up correctly.
check_kern_pgdir();
```

为了方便将 3 段合在了一起，文章中提到 UTOP 以下的地址是用户空间,ULIM 以上是内核空间，这之间还有一段缓冲区，根据注释首先将页面全部映射到 UPAGES 处，其次是内核栈的初始化，最后将 KERNBASE 处的一整块内存完成映射，这部分由于有详尽的注释，完成的较快，唯一有点坑的是自己错把 sizeof(Page)和 PGSIZE 当成等价的而忘了对齐。

自此通过了全部测试，Question 部分见 txt 文件。