Exercise 1. In the file kern/pmap.c, you must implement code for the following functions (probably in the order given).
boot_alloc() mem_init() page_init() page_alloc() page_free()

```
static void * boot_alloc(uint32_t n)
{
	static char *nextfree;
	char *result;

	if (!nextfree) {
	    extern char end[];
	    nextfree = ROUNDUP((char *) end, PGSIZE);
	}

	// Allocate a chunk large enough to hold 'n' bytes, then update nextfree.  Make sure nextfree is kept aligned to a multiple of PGSIZE.
	result = nextfree;
	nextfree += ROUNDUP(n, PGSIZE);
	return result;
}
```

在 nextfree 不是 0 的时候，也就是已经被 alloc 之后，先把当前的 nextfree 位置保存在 result 中，作为之后分配的大小的 head。

由于需要与 PGSIZE 对齐，所以利用 ROUNDUP 函数将 n 上取 PGSIZE 整。

将 nextfree 加上该值，代表这些内容已经被分配出去了。

最后将 result（即原 nextfree）返回作为分配的空间的指针。

```
void mem_init(void)
{
    ......
        // 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.
    pages = boot_alloc(sizeof(struct Page) * npages);
    ......
}
```

用 boot_alloc 初始化页表项

需要 npages 个 Page 结构 ,所以需要分配的大小为 sizeof(struct Page)*npages)

将这个得到的空间的头指针赋给全局变量 pages

```
void page_init(void)
{
        int i;
        for (i = npages-1; i > -1; i--) {
            // I/O 空洞（640K 之后的 384K）与第 0 个 page 不能被分配出去
            if((PGSIZE * i >= IOPHYSMEM && PGSIZE * i < EXTPHYSMEM) || i == 0) continue;
            // base memory（0—640K，但不包括第一个 4K） && pages 之后 是 free 的
            if(i < npages_basemem || i > (int)ROUNDUP((char*)pages + sizeof(struct Page) * npages - KERNBASE, PGSIZE) / PGSIZE)
            {
                pages[i].pp_ref = 0;
                pages[i].pp_link = page_free_list;
                page_free_list = &pages[i];
                //cprintf("page_init:%x\n",page_free_list);
            }
        }
        chunk_list = NULL;
}
```

page_free_list 代表着可以分配的 page 的链表。

0—640K 的 base_memory(但是不包括第一个 4K)以及分配出的 pages 之后的位置是空闲的 page ,可以被加入 page_free_list 链表。

I/O 空洞(640K 之后的 384K)与第 0 个 page 以及分配给 page 结构的位置不能被分配出去

page_free_list 是链表头为 0 的正向链表。(即 4K 位置为第一个 page，pp_link 指向 8K 位置的 page)，与初始给出的不同。这个做法使得之后 page_alloc_npages 更加方便，如果是反向链表，则分配时给出的分配位置的最后，需要反向修改 pp_link，非常麻烦。

当然最初没有思考这个问题，在出错了很长时间之后，在 check_continues 中 cprintf 出值才发现是-4096。

最后 chunk_list 是缓存页表链，page_free_npages 时会将 free 的链表存在 chunk_list 中以便于复用。但是初始为空。

```
struct Page *page_alloc(int alloc_flags)
{
        if(page_free_list)
        {
            // 把当前 page_free_list 的第一个 Page 分配出去，再把 page_free_list 指向下一个
            struct Page* tmp = page_free_list;
            page_free_list = page_free_list -> pp_link;
            // 把该地址转换成 kernel virtiual address ，利用 memset 把这个 page 都设为 0
            if(alloc_flags & ALLOC_ZERO)
                memset(page2kva(tmp), '\0', PGSIZE);
            tmp->pp_link = NULL;
            return tmp;
        }
        return 0;
}
```

首先检查 page_free_list，看 page_free_list 中是否还有未分配的 page（即是否为空）

把当前 page_free_list 的第一个 Page 分配出去，再把 page_free_list 指向下一个，且把分配出的 Page 与之后的 Page 断开（pp_link 指向空）

如果需要清空(即 alloc_flags & ALLOC_ZERO),调用 memset 清空

```
void page_free(struct Page *pp)
{
        struct Page* tmp = page_free_list;
        if(page_free_list == 0)
        {
            page_free_list = pp;
            return;
        }
        if(page2pa(pp) < page2pa(page_free_list))
        {
            pp->pp_link = page_free_list;
            page_free_list = pp;
        }
        else
        {
            while(tmp->pp_link  &&  page2pa(tmp->pp_link)  <
page2pa(pp)) tmp = tmp->pp_link;
            pp->pp_link = tmp->pp_link;
            tmp->pp_link = pp;
        }
}
```

首先检查 page_free_list ，如果为空，那么直接将当前 page_free_list 置为 pp（也就是 pp 为链表头）

如果 pp 的 physical address 在 page_free_list 之前 ,那么将 pp 插入 page_free_list 的 head

否则，在 page_free_list 中找到 pp 的正确位置并插入

这个做法是为了保持 page_free_list 的顺序性，page_alloc_npages 需要连续页表，如果破坏了 page_free_list 的顺序性，那么分配连续页表将变得非常困难。

Exercise 2. In the file kern/pmap.c, you must implement code for the following functions.

page_alloc_npages(int alloc_flags, int n)

page_free_npages(struct Page *pp, int n)

```
struct Page *page_alloc_npages(int alloc_flags, int n)
{
    if(n <= 0) return NULL;
   struct Page* tmp = NULL,*now = NULL,*tmp_before = NULL;
    int i = 0;
    if(chunk_list) {
        tmp_before = tmp = now = chunk_list;
        while(now && i<=n)
        {
            i++;
            if(i == n) {
                if(tmp_before == chunk_list) chunk_list = now->pp_link;
                else tmp_before->pp_link = now->pp_link;
                now->pp_link = NULL;
                if(alloc_flags & ALLOC_ZERO) memset(page2kva(tmp), '\0', n * PGSIZE);
```

```
            return tmp;
        }
        if(now->pp_link == NULL) break;
        if(page2pa(now->pp_link)-page2pa(now) != PGSIZE)
        {
            tmp_before = now;
            tmp = now = now->pp_link;
            i = 0;
            continue;
        }
        now = now->pp_link;
    }
}
i = 0;
if(page_free_list)
{
    tmp_before = tmp = now = page_free_list;
    while(now && i<=n)
    {
        i++;
        if(i == n)
        {
```

```
                cprintf("page_free_list\n");
                if(tmp_before == page_free_list)
                    page_free_list = now->pp_link;
                else tmp_before->pp_link = now->pp_link;
                now->pp_link = NULL;
                if(alloc_flags & ALLOC_ZERO)
memset(page2kva(tmp), '\0', n * PGSIZE);
                return tmp;
            }
            if(now->pp_link == NULL) break;
            if(page2pa(now->pp_link)-page2pa(now) != PGSIZE)
            {
                tmp_before = now;
                tmp = now = now->pp_link;
                i = 0;
                continue;
            }
            now = now->pp_link;
        }
    }
    return NULL;
}
```

在 chunk_list 中查找有没有数量为 n 的连续 page，如果有，则分配出去，如果没有，在 page_free_list 中查找。

具体做法为：首先置寻找位置为 head，然后逐一检查其 pp_link 的 physical address 与其是否相差 PGSIZE（即连续），如果检查到一个不是，那么 head 就重新变为 pp_link，直到找到连续的 n 个 page。

这个做法是由 page_free_list, chunk_list 的顺序性所保证的。

```
int page_free_npages(struct Page *pp, int n)
{
    int i = 0;
    struct Page* tmp = pp;
    for(i = 0;i < n-1; i++)
    {
        if(tmp == NULL || tmp->pp_link == NULL ||
page2pa(tmp->pp_link)-page2pa(tmp) != PGSIZE)
            return -1;
        tmp = tmp->pp_link;
    }

    tmp->pp_link = NULL;

    if(chunk_list == NULL)
    {
        chunk_list = pp;
        return 0;
    }
    if(page2pa(pp) < page2pa(chunk_list))
    {
        tmp->pp_link = chunk_list;
```

```
        chunk_list = pp;
    }
    else
    {
        struct Page* chunk_now = chunk_list;
        while(chunk_now->pp_link                    &&
page2pa(chunk_now->pp_link) < page2pa(pp))
            chunk_now = chunk_now->pp_link;
        tmp->pp_link = chunk_now->pp_link;
        chunk_now->pp_link = pp;
    }
    return 0;
}
```

首先检查pp的连续性，如果有一个不连续（其pp_link的physical address 与其相差不为 PGSIZE）或者链表的长度不足 n，则直接返回-1。

在 chunk_list 中找到 pp—pp+n 的位置（方法与 free_page 相同），并插入 chunk_list 中。

同样，这个做法是为了保证 chunk_list 的顺序性。

Exercise 2(2). You may know realloc() in C program. Sometimes, applications want to shrink and enlarge the memory they alrealy allocated. A simple way is to allocate a new one and use memcpy() to fill it with data, then free the old one. However, allocator can provide a fast way to support this function. Please implement the following function which changes the allocated size from old_n pages to new_n pages.

page_realloc_npages(struct Page *pp, int old_n, int new_n);

```
struct Page *page_realloc_npages(struct Page *pp, int old_n, int new_n)
{
    if(new_n == 0)
    {
        page_free_npages(pp,old_n);
        return NULL;
    }
    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;
}
struct Page* new = pp+old_n;
struct Page* now = chunk_list;
struct Page* tmp_before = chunk_list;
while(new>now)
{
    tmp_before = now;
    now = now->pp_link;
}
if(page2pa(now) == page2pa(new))
{
    int i = 0;
    while(now && i<=new_n-old_n)
    {
        i++;
        if(i == new_n-old_n)
        {
            if(tmp_before == chunk_list)
                chunk_list = now->pp_link;
            else tmp_before->pp_link = now->pp_link;
            now->pp_link = NULL;
```

```
            (pp+old_n-1)->pp_link = new;
            return pp;
        }
        if(now->pp_link == NULL) break;
        if(page2pa(now->pp_link)-page2pa(now) != PGSIZE)
            break;
        now = now->pp_link;
    }
}

now = page_free_list;
tmp_before = now;
while(new>now)
{
    tmp_before = now;
    now = now->pp_link;
}
if(page2pa(now) == page2pa(new))
{
    int i = 0;
    while(now && i<=new_n-old_n)
    {
```

```
            i++;
            if(i == new_n-old_n)
            {
                if(tmp_before == page_free_list)
                    page_free_list = now->pp_link;
                else tmp_before->pp_link = now->pp_link;
                now->pp_link = NULL;
                (pp+old_n-1)->pp_link = new;
                return pp;
            }
            if(now->pp_link == NULL)
                break;
            if(page2pa(now->pp_link)-page2pa(now) != PGSIZE)
                break;
            now = now->pp_link;
        }
    }

    page_free_npages(pp,old_n);
    return page_alloc_npages(1,new_n);
}
```

最简单的实现方法即为先 page_free_npages，再page_alloc_npages(由于是正向链表,实际也能通过测试),但是这么做很没有效率。

考虑到优化,那么如果 new_n 比 old_n 要小的话,那么只要 free 之后的 old_n-new_n 个 page 就好。

如果 new_n 与 old_n 相等的话，那么直接不做改动。

如果 new_n 比 old_n 大的话，分别从 chunk_list 与 page_free_list 中找有没有 pp+old_n 以及之后连续的 new_n-old_n 个 page(正向顺序链表使之成为可能),如果都没有，再用最原始的方法。

Exercise 5. In the file kern/pmap.c, you must implement code for the following functions.

pgdir_walk() boot_map_region() page_lookup() page_remove() page_insert()

```
pte_t *pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    if(pgdir[PDX(va)] == 0)
    {
        if(!create) return NULL;
        struct Page* new_page = page_alloc(1);
        if(!new_page) return NULL;
        new_page->pp_ref++;
        pgdir[PDX(va)]=page2pa(new_page)|PTE_P|PTE_W|PTE_
    U;
        return (((pte_t*)page2kva(new_page))+PTX(va));
    }
    else
        return
(((pte_t*)page2kva(pa2page(PTE_ADDR(pgdir[PDX(va)]))))+PT
X(va));
}
```

检查一级页目录里有没有分配二级页目录的 page ,如果分配了那么找到相应的二级页目录中 page 指针并把它转换回 virtual address 并返回。

如果没有分配且允许创建（create），那么 page_alloc，分配一个新的 page，并将一级页目录中该值置为这个二级页目录的 physcial address。并将二级页目录中 page 指针转换回 virtual address 并返回

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

如果 virtual address 是在 User Top 之上，这一块内存是不能被映射的，直接返回。

否则从 virtual address 开始，以 PGSIZE 为一个 page 映射 physical address 到 virtual address（即利用 pgdir_walk 在页目录中分配页表的 physical address），且设置权限为 perm | PTE_P

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

利用 pgdir_walk 检查当前 virtual address 所在的页表有没有被分配（在未分配的情况下不新创建）

如果没有分配，那么返回 NULL

如果分配了，那么返回改分配 physical address 指向的 page

在 pte_store 不为空的情况下 ,把其内容置为该 physical address

```
void page_remove(pde_t *pgdir, void *va)
{
    pte_t* tmp=0;
    struct Page* page=page_lookup(pgdir,va,&tmp);
    if(!page) return;
    page_decref(page);
    *tmp = 0;
    tlb_invalidate(pgdir,va);
}
```

检查当前 virtual address 在页目录中是否已经分配了 page

如果没有分配，直接返回

如果分配了，则首先把页表的 ref 减为 0，再把页目录中的值置为 0，再使 tlb 变成 invlidate

```
int page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
    pte_t* pte;
    struct Page* pg=page_lookup(pgdir,va,NULL);
    if(pg==pp)
    {
        pte=pgdir_walk(pgdir,va,1);
        *pte = page2pa(pp)|perm|PTE_P;
        return 0;
    }
    else if(pg) page_remove(pgdir,va);
    pte=pgdir_walk(pgdir,va,1);

    if(!pte) return -E_NO_MEM;

    *pte =page2pa(pp)|perm|PTE_P;
    pp->pp_ref++;
    return 0;
}
```

首先检查 virtual address 在页目录中是否已经被分配了 page

如果被分配了且该 page 与目标 pp 相同的话，那么把其权限修改即可

如果不同的话，把该 page 先 remove，变为未分配状态

如果未分配，则使用 pgdir_walk 找到该 virtual address 在页目录中的 entry（失败返回-E_NO_MEM），并把其值置为 pp 的物理地址，且设置权限。

Exercise 6. Fill in the missing code in mem_init() after the call to check_page().

```
void mem_init(void)
{
    ……
    boot_map_region(kern_pgdir, UPAGES, ROUNDUP(npages*sizeof(struct Page), PGSIZE), (physaddr_t)(PADDR(pages)), PTE_U | PTE_P);
    boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, (physaddr_t)(PADDR(bootstack)), PTE_W | PTE_P);
    boot_map_region(kern_pgdir, KERNBASE, ROUNDUP((uint32_t)0xFFFFFFFF-KERNBASE, PGSIZE), 0, PTE_W | PTE_P);
    ……
}
```

利用 boot_map_region 对 virtual address 相应的 physical address 进行映射。权限等按照说明进行。