Lab 5

宁晨然 17307130178

1.实验目的

- 了解操作系统对物理内存和虚拟内存的管理
- 了解内核地址空间

2.实验过程

2.1 TODO 1 Physical Page Management

操作系统在实际操作虚拟内存之前，首先需要建立物理内存的管理方式，即物理页面的分配问题。操作系统需要了解RAM中的页面信息，比如哪些在使用中，组织方式如何，所以需要构建一个页面的数据结构。这里利用`PageInfo`这个数据结构，一一对应一个`page`，其中记录了是否被使用的信息，整体构成链表结构。

2.1.1 `boot_alloc()`

作用：物理内存分配

使用时机：`JOS`正在设置虚拟内存系统，`page_alloc()`才是真正的分配器。

分配方式：

- `n>0`：分配连续物理内存，足以装下`n`字节。并不初始化，返回内核虚拟地址。
- `n==0`：不分配任何内存，返回下一个自由页地址。
- 内存不足：`panic`

注意：这个函数只在初始化时使用，并且在`page_free_list`设置好之前使用。`page_free_list`出现之后，可以直接在`free_list`中找到未被使用的内存空间。

函数解释：

- `end`：`end`指向`kern`的`bss`段的末尾，即第一个未使用的虚拟内存地址。
- 当调用`boot_alloc(0)`时，可以用作调用得到下一个自由页地址。

```
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree; // virtual address of next byte of free memory
    char *result;
    if (!nextfree)
    {
        extern char end[];
        nextfree = ROUNDUP((char *)end, PGSIZE);
    }
    result = nextfree;
    nextfree = ROUNDUP(nextfree + n, PGSIZE);
    if ((uint32_t)nextfree - KERNBASE > (npages * PGSIZE))
        panic("Out of memory!\n");
    return result;
```

```
}
```

2.1.2 `page_init()`

作用：初始化页结构和内存自由链表

使用时机：已经分配好了页目录和页信息的物理内存后；使用页初始后，之后分配内存均使用`page_*`。

注意：使用`page_*`从`page_free_list`中分配、重分配物理内存。

分配方式：

- `page0`：物理页在使用中，万一之后`IDT`或者`BIOS`用到它。
- `base memory`：自由可分配区域，`[PGSIZE, npages_basemem*PGSIZE)`，之前探测出来的值。
- `IO hole`：永不能分配区域，`[IOPHYSMEM, EXTPHYSMEM)`
- `extended memory`：部分在使用中，部分自由可分配。`[EXTPHYSMEM, ...)`

```
void page_init(void)
{
    size_t i;
    page_free_list = NULL;
    //num_alloc：在extmem区域已经被占用的页的个数
    int num_alloc = ((uint32_t)boot_alloc(0) - KERNBASE) / PGSIZE;
    //num_iohole：在io hole区域占用的页数
    int num_iohole = 96;
    for (i = 0; i < npages; i++)
    {
        if (i == 0)
        {
            pages[i].pp_ref = 1;    //page 0
        }
        else if (i >= npages_basemem && i < npages_basemem + num_iohole +
num_alloc)
        {
            pages[i].pp_ref = 1;    //可分配的部分
        }
        else
        {
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;  //更新未分配的page，加入
page_free_list中
            page_free_list = &pages[i];
        }
    }
}
```

其中`PageInfo`是很重要的页表信息结构体贯穿始终，很重要：

- 链表结构：组织性、一一对应的将RAM成功记录为物理内存页面。
- `pp_ref`：有多少个指针指向该页
- `pp_link`：空闲页面链表中下一个空闲页面；非空闲页该值为`NULL`。

```
struct PageInfo {
    struct PageInfo *pp_link;
    uint16_t pp_ref;
};
```

2.1.3 page_alloc()

作用：分配一个物理页

使用时机：调用者需要分配一个物理页的时候

分配步骤：

- 如果 alloc_flags & ALLOC_ZERO：将返回的物理页置'\0'。
- 不能增加 pageinfo 中引用指针计数大小，由调用者自己改变。
- 从 page_free_list 中调出一个空页，并返回该页；需要改变 page_free_list 的头和调出页的 pp_link。
- 使用 page2kva(result) 获取该页的物理地址，并将该页置零。

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

2.1.4 page_free()

作用：释放一个页

使用时机：当没有引用该页指针时，且要释放页。

步骤：

- 检查 pp_ref 为0，且 pp_link 不为 NULL。
- 将该页释放入 page_free_list 中。

```
void page_free(struct PageInfo *pp)
{
    assert(pp->pp_ref == 0);
    assert(pp->pp_link == NULL);
    pp->pp_link = page_free_list;
    page_free_list = pp;
}
```

2.1.5 mem_init()

作用：设置一个二级页表结构

使用时机：需要设置一个内核部分的内存空间的时候。（用户部分的内存空间之后设置）

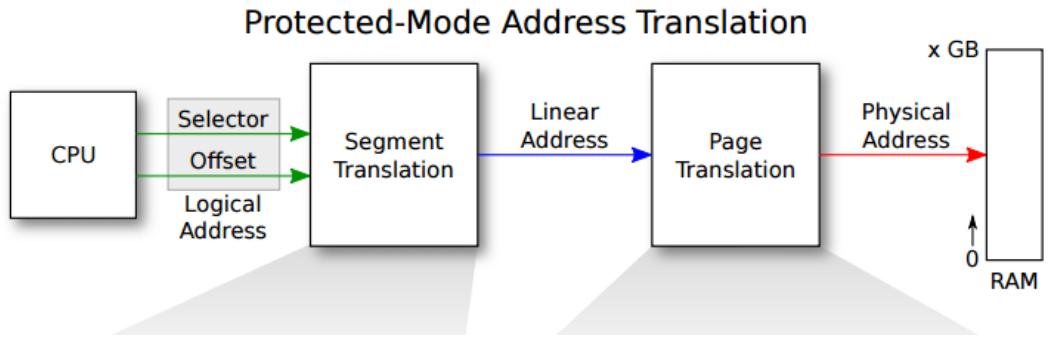
层级关系：

用户权限可以分为三层结构，`-UTOP`、`UTOP-ULIM`、`ULIM-`

- `-UTOP`：用户可读可写
- `UTOP-ULIM`：用户可读不可写
- `ULIM-`：用户不可读不可写

页表结构可以分为二层结构：

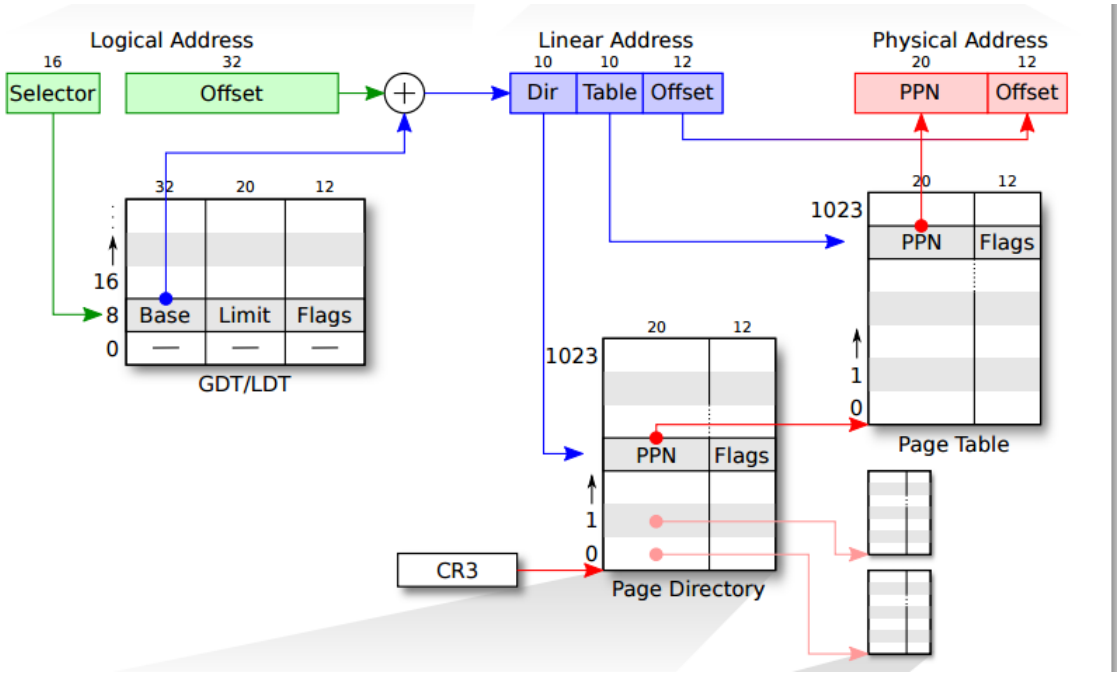
访问物理地址的时候，`CPU`需要简单的操作将虚拟地址向内存地址转换。 而二级页表，在`mem_init`体现为`page_directory`和`page_table`两级页表。



二级页表，此处分为`page directory`（`PDE`）和`page table`（`PTE`）两个部分，比如32位的系统中，一个32位的地址，低位12位作为页内偏移量，高位的10位作为`page directory`的索引，中间的10位作为`page table`的索引，这样如果映射的时候可以节省大量的内存空间。因为此时可以以很少的一级页表`page directory`就可以访问到大量连续局部的页面数据，相对节省内存空间。

下图中分为了三种地址：逻辑地址、线性地址、物理地址。`CPU`访问一个地址时，使用的是虚拟地址，即逻辑地址，前面16位是段选择器，分段的目的就是为了把代码段、数据段和堆栈段等分开，后32位是偏移量。通过虚拟地址获得线性地址，即之前的二级页表结构。再通过页表映射到物理地址。

由于`JOS`中只有一个段，所以虚拟地址的`Offset`就是线性地址，之后的实验也可以直接把虚拟地址当作线性地址使用。



并且注意到分配内存时使用的是`boot_alloc`，这是`JOS`设置自身虚拟内存系统时使用的物理内存分配器，只在`mem_init`中使用，之后空闲页面列表`page_free_list`设置好之后，就使用`page_*`的方式管理页面。

步骤：

- 设置物理内存
  - `i386_detect_memory`：探测物理内存大小，主要设置好变量`npages`和`npages_basemem`。
  - `kern_pgdir`指向线性虚拟地址的根部。
  - `PageInfo`：分配`npages`个的页信息，并初始化为0。
  - `page_init`：初始化页结构和内存自由链表。
  - `check`各种信息
    - `check_page_free_list`：保证`page_free_list`的页表合理。
    - `check_page_alloc`：保证页分配功能正确，主要是`page_alloc()`,`page_free()`,`page_init()`三个功能需要正确。
    - `check_page`：保证页操作功能正确，主要是`page_insert`,`page_remove`功能。
- 设置虚拟内存（这里暂时忽略）

```
void mem_init(void)
{
    uint32_t cr0;
    size_t n;
    i386_detect_memory();
    kern_pgdir = (pde_t *)boot_alloc(PGSIZE);
    memset(kern_pgdir, 0, PGSIZE);
    kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
    pages = (struct PageInfo *)boot_alloc(npages * sizeof(struct PageInfo));
    memset(pages, 0, npages * sizeof(struct PageInfo));
    page_init();
    check_page_free_list(1);
    check_page_alloc();
    check_page();
    ...
}
```

2.2 TODO 2 Virtual Memory

在填充代码之前，需要先理解几个比较重要的函数和宏定义。页面的结构主要是线性地址（`JOS`中就是虚拟地址）到物理地址的转换过程需要留意，其中的宏定义都在`inc/mmu.h`中。

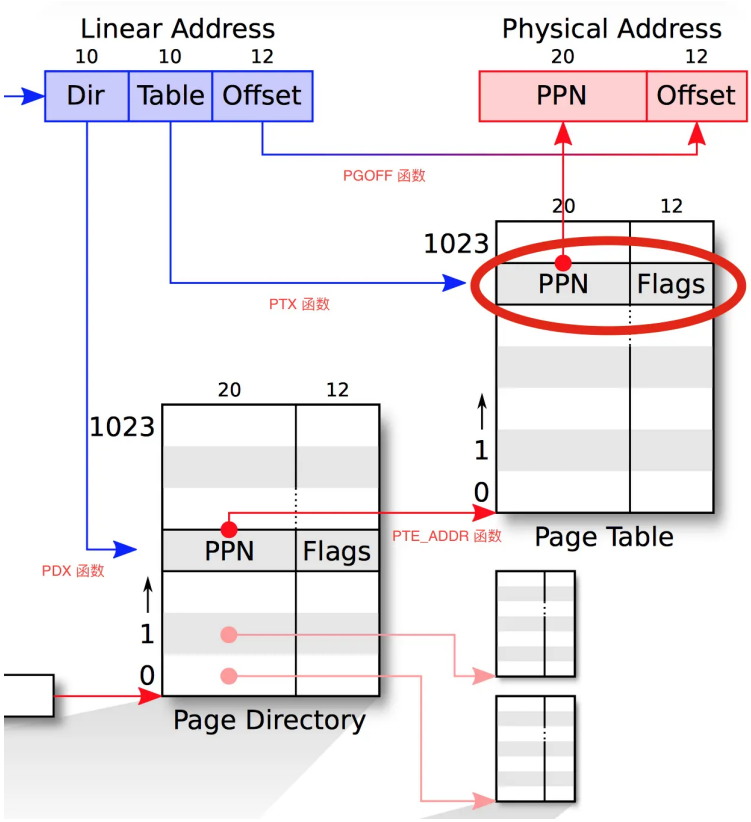
```
// A linear address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \---------- PGNUM(la) ----------/
```

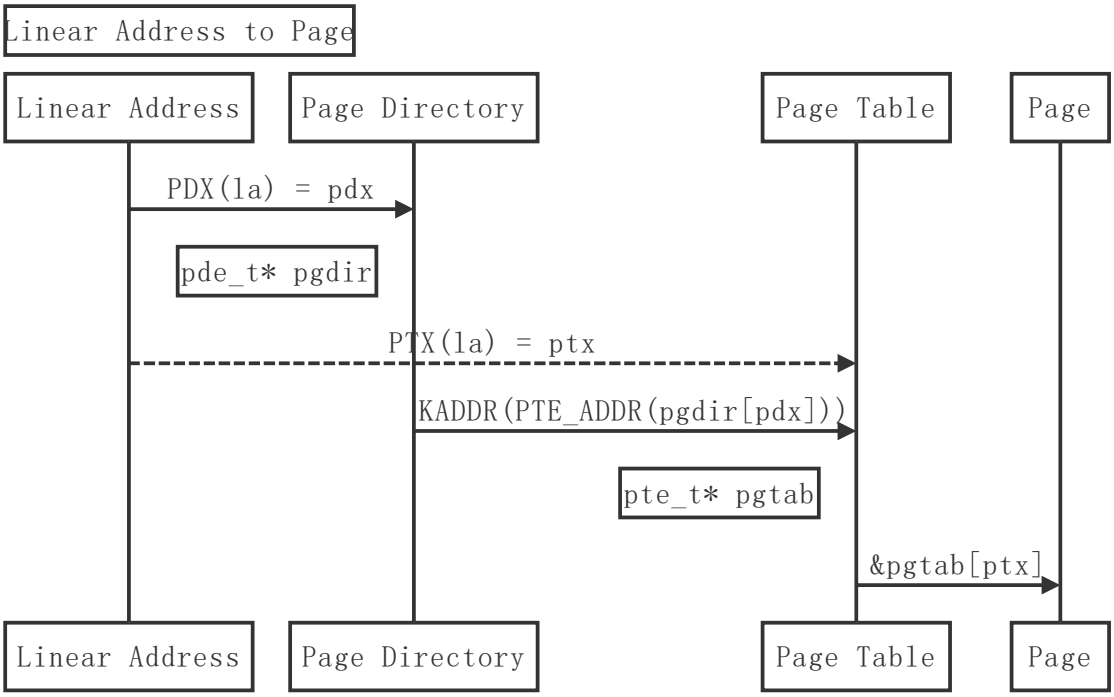
`pte_t`：`uint32_t`类型，表示`page_table_index`的变量类型。

`pde_t`：`uint32_t`类型，表示`page_directory_index`的变量类型。

`pte_t *pgdir_walk(pde_t *pgdir, const void *va, int create)`:

- 给定`page directory`入口`pgdir`，和虚拟地址`va`，是否新建变量`create`
- 查找`page directory`，找到`va`对应的`page directory`中的`page table entry`指针并返回。
- 如果没找到，根据`create`值判断是否新建。
- 返回值就是下面红圈部分
- 注意`page directory`和`page table`中的地址都是前20位信息和后12位的`flags`





上图展示了从线性地址找到页面信息的流程。

2.2.1 `page_lookup()`

作用：查找某页

使用时机：一般被`page_remove()`调用，大多数情况其他调用者不能调用该函数

步骤：

- 输入：页目录入口指针，虚拟地址，（可选）存储页表地址的地方。
- 输出：返回查找到的页表入口，如果没有返回NULL
- 按需把查找到的页表入口存储

注意：

- `entry`是一个指向`page_table`其中一项的指针，这个地址前20位即物理地址的前20位(`PNN`)。
- `pgdir_walk()`可以返回一个这样的`entry`
- `pte_store`是可选项，是否需要存储该`entry`
- 返回的类型是该物理地址的页面信息，所以需要先将`page table`中的地址通过`PTE_ADDR`转换为物理地址，再用`pa2page`将物理地址转换为页面信息指针。

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    pte_t *entry = NULL;              //entry points to a page table address.
    entry = pgdir_walk(pgdir, va, 0); // get the mapping page of this address va
but don't creat.
    if (entry == NULL)
        return NULL; // didn't find anything

    // found one
    if (pte_store)
        *pte_store = entry;
    return pa2page(PTE_ADDR(*entry));
}
```

2.2.2 `page_remove()`

作用：移除某页

步骤：

- 输入：页目录入口指针，虚拟地址。
- 查找`va`对应的页表地址，移除该页并自动减少该页的引用。
- 清空页表中的该地址，防止可以重新获得该页物理地址。
- 当页面信息被移除时，需要使对应`TLB`失效。
- 如果没有找到，则不做任何操作。

注意：

- 调用`page_lookup`查找页面信息
- `page_decref()`取消引用；`tlb_invalidate`使`TLB`失效。

```
void page_remove(pde_t *pgdir, void *va)
{
    pte_t *entry; //entry points to a page table address
    struct PageInfo *pginfo = page_lookup(pgdir, va, &entry);
    if (pginfo == NULL) //didn't find anything
        return;
    page_decref(pginfo);
    *entry = 0; //clear the address in page table like never used before
    tlb_invalidate(pgdir, va);
    return;
}
```

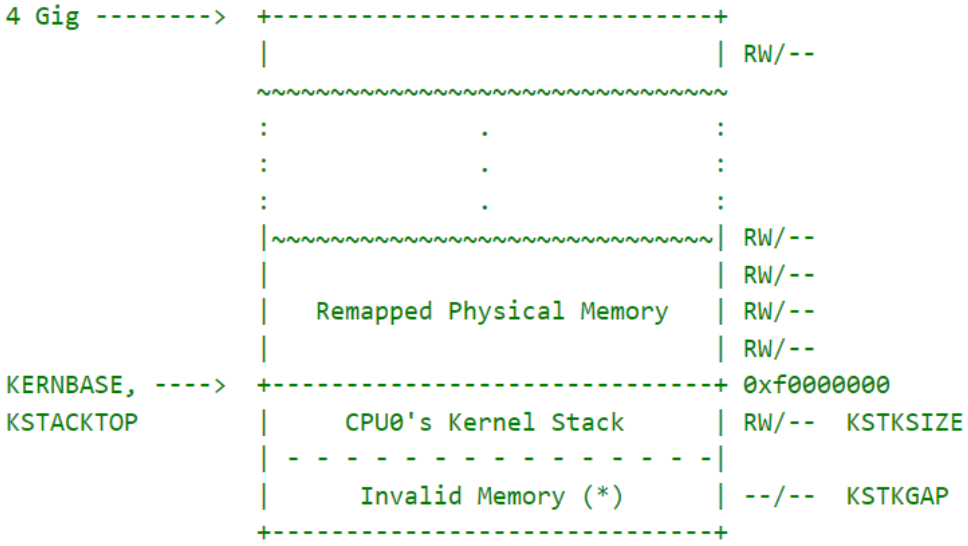
2.3 TODO 3 Kernel Address Space

为了补全`mem_init()`的代码段，实际上就是补全从**虚拟地址**映射到对应物理地址的部分。可以参考`UPAGES`的映射方法：

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int
perm)
```

这个函数的目的就是**将`[va, va+size)`的虚拟地址空间映射到`[pa, pa+size)`物理地址空间。**

- `page table`在`pgdir`中查找。
- 总大小是`size`的整数倍，且`va`和`pa`已经对齐。
- 仅用于设置`UTOP`上面的静态区域的区域；不能修改已经映射好的页面的`pp_ref`。



```
* ULIM, MMIOBASE -->  +------------------------------+ 0xef800000
*                     |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
*    UVPT      ---->  +------------------------------+ 0xef400000
*                     |          RO PAGES            | R-/R-  PTSIZE
*    UPAGES    ---->  +------------------------------+ 0xef000000
*                     |           RO ENVS            | R-/R-  PTSIZE
* UTOP,UENVS ------>  +------------------------------+ 0xeec00000
* UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
*                     +------------------------------+ 0xeebff000
*                     |       Empty Memory (*)       | --/--  PGSIZE
*    USTACKTOP  --->  +------------------------------+ 0xeebfe000
```

上图给出了部分虚拟内存的结构，栈空间向上生长，右侧为权限（内核/用户）。其中我们要实现的三个虚拟空间的映射就是`UPAGES(0xef000000 ~ 0xef400000)`，`KSTACK(0xefff8000 ~ 0xf0000000)`，`Kernel(0xf0000000 ~ 0xffffffff)`。

首先借鉴一下`UPAGES`的映射方法：

```
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
```

参数分别对应的含义是：

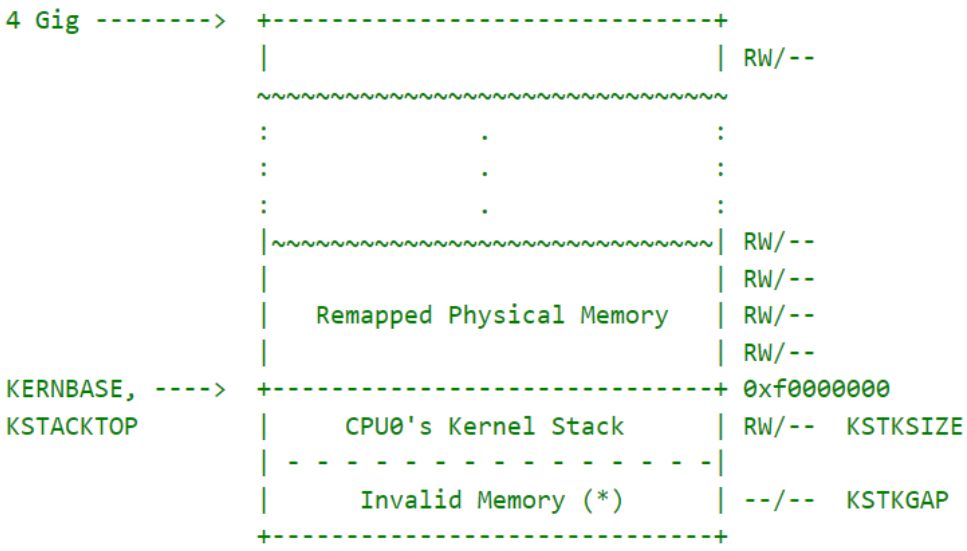
- `kern_pgdir`：新建的页目录。
- `UPAGES`：当前区域，`UPAGES`自身就是虚拟地址。
- `PTSIZE`：`PTSIZE = PGSIZE*NPTENTRIES`，其中`PGSIZE = 4096`，即有4096个`entry`；`NPTENTRIES=1024`，页表中一个`entry`大小为`1k`；所以页表大小`PTSIZE`就是`4MB`。对应了`UPAGES`的大小`(0xef000000 ~ 0xef400000)`。
- `PADDR(pages)`：将`UPAGES`虚拟地址映射到实际的`pages`物理地址上。
- `PTE_U`：表示用户权限。

`UPAGES`权限如下：

状态	内核	用户
新页面	R	R
页面本身	RW	None

2.3.1 Kernel Stack

物理空间中的`bootstack`意味着`kernel stack`，并且这个内核栈从上往下生长，初始地址为虚拟地址`KSTACKTOP`，所以内核的虚拟栈空间即`[KSTACKTOP-PTSIZE, KSTACKTOP)`。但是其中只有`[KSTACKTOP-KSTKSIZE, KSTACKTOP)`分配了物理地址，`[KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE)`并没有分配，所以如果栈溢出后，只会出错并不会覆盖物理地址空间。这里的权限只有内核`RW`。



可知现在的内核栈需要的几个参数就是

- `kern_pgdir`：页目录入口。
- `KSTAKTOP - KSTKSIZE`：虚拟地址，如图所示。一共32`KB`，对应`(0xefff8000 ~ 0xf0000000)`。
- `KSTKSIZE`：需要分配内核栈的空间大小。
- `PADDR(bootstack)`：`boostack`是物理地址中对应的内核栈。
- `PTE_W | PTW_P`：`kernel`可读可写，用户无权限。

代码如下：

```
boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack),
PTE_W | PTE_P);
```

2.3.2 Kernel

整个内核有256`mb`，是`(0xf0000000 ~ 0xffffffff)`，即虚拟地址`[KERNBASE, 2^32)`映射到物理地址`[0, 2^32 - KERNBASE)`，且权限是仅内核可读可写，与上一个类似。

代码如下：

```
boot_map_region(kern_pgdir, KERNBASE, ROUNDUP(0xffffffff - KERNBASE, PGSIZE), 0,
PTE_W | PTE_P);
```

3.实验结果

```
Booting from Hard Disk...
6828 decimal is XXX octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

非常成功！

4.实验感想

- 本次实验学会了虚拟地址和物理地址的分配和转换问题，理解了用户和内核的区别；了解了内核栈的分配，物理空间的分配方式等。
- 相对而言比较简单，这次实验的结果没有这么多debug让人欣慰。

5.参考文献

[1] https://blog.csdn.net/fang92/article/details/47320747

[2] https://www.jianshu.com/p/3be92c8228b6