Lab #2

陈威宇

Exercise 1

boot_alloc 分配 n 字节物理内存，只在 JOS 创建虚拟内存系统的时候调用，之后就不用了. 代码如下，注意判断 out of memory.

```
  if (n>0){
    uint32_t sz = ROUNDUP(n, PGSIZE);
    nextfree += sz;
    if (((uintptr_t)nextfree)-KERNBASE > 0x00400000)
      panic("out of memory");
    return (nextfree-sz);
  }
  else{
    return nextfree;
  }
```

mem_init 里的 npages 是物理页的总数, pages 是维护这些物理页的数组, 每个元素的类型都是 struct PageInfo.

```
  pages=(struct PageInfo *)boot_alloc(npages*sizeof(struct PageInfo));
  memset(pages,0,npages*sizeof(struct PageInfo));
```

page_init 的作用是把 pages 数组初始化，并初始化维护 free physical page 的 page_free_list 链表.

```
void
page_init(void)
{
  page_free_list=NULL;
  size_t i;
  for (i = 0; i < npages; i++) {
    pages[i].pp_ref = 0;
    if (i==0 || (i*PGSIZE >= IOPHYSMEM && i*PGSIZE < EXTPHYSMEM) || (i*PGSIZE>=0x100000 && i*P
      pages[i].pp_link = NULL;
    }
    else{
      pages[i].pp_link = page_free_list;
      page_free_list = &pages[i];
    }
```

```
  }
}
```

page_alloc 的作用是分配一个 free physical page, 直接从 free list 取即可.

```
struct PageInfo *
page_alloc(int alloc_flags)
{
  // Fill this function in
  struct PageInfo * t;
  if (page_free_list==NULL) return NULL;
  t = page_free_list;
  page_free_list = (t -> pp_link);
  if ((t->pp_ref)!=0){
    panic("PANIC!!!");
  }
  (t->pp_link)=NULL;
  if (alloc_flags & ALLOC_ZERO)
    memset(page2kva(t),0,PGSIZE);
  return t;
}
```

page_free 的用处是将一个 page 放回 free list，当然，要确保其被 reference 的次数已经是 0 了.

```
void
page_free(struct PageInfo *pp)
{
  // Fill this function in
  // Hint: You may want to panic if pp->pp_ref is nonzero or
  // pp->pp_link is not NULL.
  if ((pp->pp_ref)!=0 || (pp->pp_link)!=NULL){
    panic("page_free PANIC!!!");
  }
  (pp->pp_link) = page_free_list;
  page_free_list = pp;
}
```

Question after Exercise 3

1

变量 x 的类型应当是 uintptr_t.
因为 *value 可以直接修改，所以 value 是虚拟地址.

Exercise 4

`pgdir_walk` 的作用是定位到一个 virtual address 对应的 PTE，直接查 page directory 即可. 如果其对应的 page table 不存在的话，根据 create 的值可能会新建一个.

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
  // Fill this function in
  uintptr_t la = (uintptr_t)va;
  pde_t t = (*(pgdir+PDX(la)));
  if (t&PTE_P){
    physaddr_t p1 = PTE_ADDR(t);
    uintptr_t p2 = (uintptr_t)KADDR(p1);
    return ((pte_t *)p2)+PTX(la);
  }
  if (create==0)
    return NULL;
  struct PageInfo * tmp = page_alloc(1);
  if (tmp==NULL)
    return NULL;
  memset(page2kva(tmp),0,PGSIZE);
  ++(tmp->pp_ref);
  (*(pgdir+PDX(la))) = (page2pa(tmp) | 0x007);
  return ((pte_t *)page2kva(tmp)) + PTX(la) ;
}
```

`boot_map_region` 的作用是，将连续的一些 virtual page 映射到连续的一些 physical page. 做法是，对每个 virtual page，通过 `pgdir_walk` 找到 PTE，然后直接修改即可.

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
  // Fill this function in
  if (size==0) return;
  for (int i=0;i<size/PGSIZE;++i){
    uintptr_t VA=va+i*PGSIZE;
    physaddr_t PA=pa+(VA-va);
    pte_t * tmp = pgdir_walk(pgdir, (void *)VA, 1);
    if (tmp==NULL)
      panic("PANIC!!!");
    (*tmp) = PA | (perm|PTE_P);
```

```
  }
}
```

`page_lookup` 的作用是对一个 virtual address 求其对应的物理页. 调用 `pgdir_walk` 后直接根据 PTE 上的 physical address 转换为物理页即可 (利用 `pa2page`).

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
  // Fill this function in
  pte_t * t = pgdir_walk(pgdir,va,0);
  if (t==NULL)
    return NULL;
  if (!((*t)&PTE_P))
    return NULL;
  if (pte_store!=0){
    (*pte_store) = t;
  }
  return pa2page(PTE_ADDR(*t));
}
```

`page_remove` 的作用是，将 va 这个 virtual address 对应的 virtual page 取消映射. 直接修改 PTE 即可，注意还要将映射到的 physical page 的 ref 减 1，即 decref. 另外还要 invalidate TLB.

```
void
page_remove(pde_t *pgdir, void *va)
{
  // Fill this function in
  pte_t * tmp;
  struct PageInfo * pg = page_lookup(pgdir, va, &tmp);
  if (pg==NULL)
    return;
  page_decref(pg);
  if (tmp!=NULL)
    (*tmp)=0;
  tlb_invalidate(pgdir,va);
}
```

`page_insert` 的作用是，将一个 virtual address 对应的 virtual page 映射到物理页 `pp`. 如果原先就有映射，要先 remove 掉. 需要注意的一点是，如果原先映射到的就是 `pp`, 不要在 remove 的时候将其加到 free list 上，我处理这个细节的办法是，在调用 `page_remove` 前给 `pp->ref` 加 1，调用后再减 1.

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
```

```
{
  // Fill this function in
  pte_t * t = pgdir_walk(pgdir,va,1);
  if (t==NULL)
    return -E_NO_MEM;
  if ((*t)&PTE_P){
    ++(pp->pp_ref);
    page_remove(pgdir, va);
    --(pp->pp_ref);
  }
  (*t) = page2pa(pp) | (perm|PTE_P);
  ++(pp->pp_ref);
  return 0;
}
```

Exercise 5

直接调用 boot_map_region 来建立内存映射.

```
  boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U | PTE_P);
  boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTACKTOP-(KSTACKTOP-KSTKSIZE), PADDR(bootst
  boot_map_region(kern_pgdir, KERNBASE, 0xffffffff-KERNBASE+1, 0, PTE_W | PTE_P);
```

Questions after Exercise 5

2

Entry	Base Virtual Address	Points to (logically):
1023	0xffc00000	Page table for top 4MB of phys memory
.....	.....	.....
960	0xf0000000	Page table for bottom 4MB of phys memory
959	0xefc00000	Page table for kernel stack
958	0xef800000	NULL
957	0xef400000	Page table for Page Directory
956	0xef000000	Page table for RO PAGES
955	0xeec00000	NULL
.....	.....	.....
1	0x00400000	NULL
0	0x00000000	NULL

3

对于 kernel momoey, 将 Page table/directory entry flags 的 PTE_U 位设为 0，用户就不能访问内核内存了。

4

256M. 这是因为, JOS 把所有物理内存都 map 到了虚拟地址 KERNBASE. 而 `KERNBASE=0xf0000000`, 于是物理内存最多为 `0x100000000` 字节，即 256M。

5

此时 overhead 分为 3 部分.
1. 存放 page directory 的空间, 即 4KB. 2. 存放所有 page table 的空间, 即 1024*4KB = 4MB 3. 维护 physical page 的数组 `pages`, 占用空间 `sizeof(struct PageInfo) * npages`, 即 `8B * (256MB/4K)) = 0.5MB`, 所以总的 overhead 有 `4.5MB + 4KB`.

6

执行完 entry.S 里的 `jmp *%eax` 语句后 EIP 跳转到比 KERNBASE 大的地方. 在 turn on paging 与 `jmp *%eax` 之间，能正常执行是因为 `entry_pgdir` 建立了虚拟地址 [0,4M) 到物理地址 [0,4M) 的映射. 这个转变是必要的，因为 kernel 要在虚拟地址的高地址执行，把低地址留给用户, 起到对 kernel memory 的保护作用.