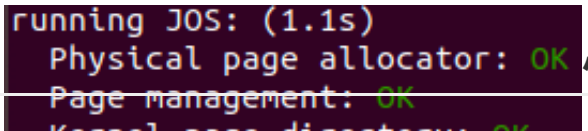
Lab2 Memory Management

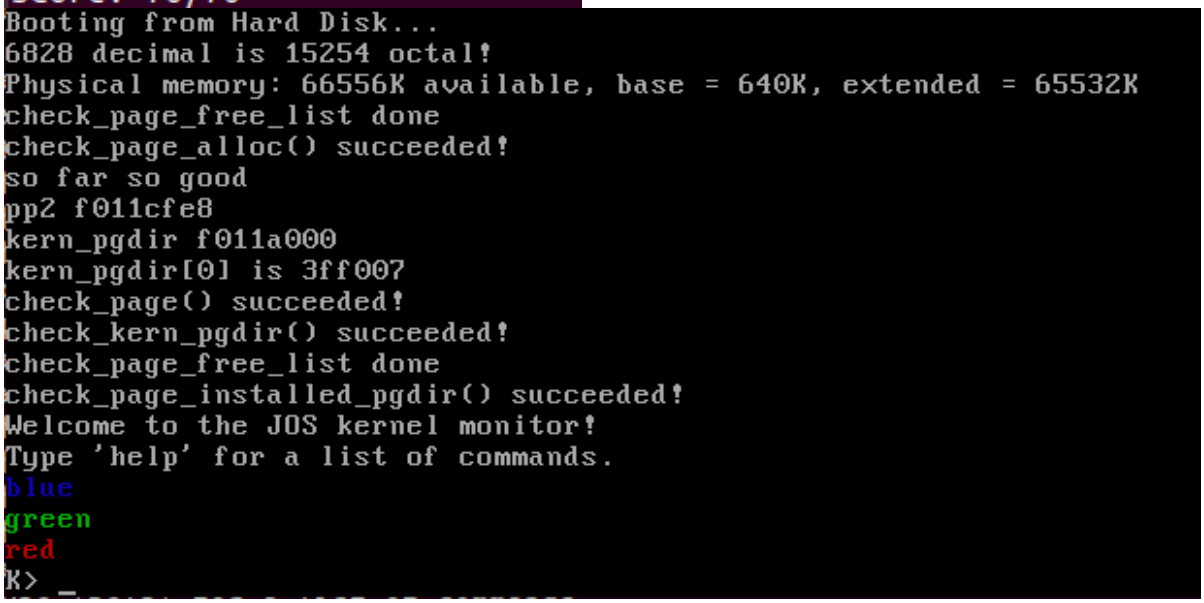
By 江晓湖 软件工程学院 同济大学

完成度

- 完成了所有的excercise。
- 完成了所有的question。
- 代码地址：https://github.com/PtNan/OSCD/tree/lab2
- 结果示例：



总览



Lab2介绍

通过该Lab，我们将为操作系统编写内存管理代码。内存管理包含两部分。

- 第一部分是为操作系统内核(kernel)进行物理内存分配。物理内存分配以页为单位，每页4096个字节。
- 第二部分是虚拟内存管理，虚拟内存将内核和用户使用的虚拟地址映射到物理地址。x86系统通过内存管理单元 MMU(memory management unit)实现映射。

Lab2实现目标

- 建立结构体PageInfo数组并一一对应物理地址
- 实现物理内存管理中页的分配与释放
- 建立页目录和页表(page tables)，填写项，一一对应虚拟地址
- 实现把输入的虚拟地址和物理地址做映射的函数
- 对内核部分的 虚拟地址完全映射到指定物理地址

Lab2可分为三部分

- ***Part1***

  Physical Page Management物理页管理

- ***Part2***

  Virtual Memory虚拟内存

- ***Part3***

  Kernel Address Space内核地址空间

Lab2新增文件

新增文件	描述
inc/memlayout.h	描述虚拟地址空间
kern/pmap.h	同memlayout.h共同定义了页结构，用来记录物理地址的页是否被使用
kern/pmap.c	实现虚拟地址空间
kern/kclock.h	
kern/kclock.c	

Part 1: Physical Page Management

操作系统需要知道RAM上物理地址哪一块是 free的 哪一块正在被使用。操作系统通过把物理地址按照页为单位做最小划分,用上MMU(内存管理单元位于CPU上)进行管理

在这一部分，我们要实现物理页的分配(allocator),要使用`struct PageInfo`的链表结构，链表每一项一一对应一个物理地址。

Exercise 1

```
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() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()

check_page_free_list() and check_page_alloc() test your physical page
allocator. You should boot JOS and see whether check_page_alloc() reports
success. Fix your code so that it passes. You may find it helpful to add your
own assert()s to verify that your assumptions are correct.
```

Solution 1

我们需要实现`kern/pmap.c`中的下列函数

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

`check_page_free_list()`和`check_page_alloc()`用于测试正确性。

1 boot_alloc()

`boot_alloc()`为物理地址的allocator

函数`static void * boot_alloc(n)`接受参数n

- 如果n>0且能分配n bytes的连续空间,则分配,不需要初始化,返回一个内核虚拟地址
- 如果n==0 返回下一个空闲页的地址 但不进行allocte
- 如果越界 则panic
- 该函数在初始化时执行

整个逻辑实现为

```
static void *boot_alloc(uint32_t n) {
  static char *nextfree; // virtual address of next byte of free memory
  char *result;

  // Initialize nextfree if this is the first time.
  // 'end' is a magic symbol automatically generated by the linker,
  // which points to the end of the kernel's bss segment:
  // the first virtual address that the linker did *not* assign
  // to any kernel code or global variables.
  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.
  //
  // LAB 2: Your code here.
  if(n > 0) {
    result = nextfree;
    nextfree = ROUNDUP((char*)(nextfree+n), PGSIZE);
    if((uint32_t)nextfree - KERNBASE > (npages*PGSIZE))
      panic("Out Of Memory!\n");
    return result;
```

```
  }
  else if(n == 0)
    return nextfree;
  return NULL;
}
```

2 mem_init()

根据待完成部分的注释，完成如下：

```
  //////////////////////////////////////////////////////////////////////
  // Allocate an array of npages 'struct PageInfo'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 PageInfo in this
  // array.  'npages' is the number of physical pages in memory.
  // Your code goes here:
  pages = (struct PageInfo *)boot_alloc(sizeof(struct PageInfo) * npages);
```

3 page_init()

开始实现`page_init()`我们要用PageInfo链表pages来记录哪些物理地址是空闲的，我们需要按照注释中所说的按不同段进行初始化。

来看一下分层

- `0x000000~0x0A0000(npages_basemem*PGSIZE or IOPHYSMEM)`,basemem，是可用的。`npages_basemem`记录basemem的页数
- `0x0A0000(IOPHYSMEM)~0x100000(EXTPHYSMEM)`，这部分叫做IO hole，是不可用的，主要被用来分配给外部设备了。
- `0x100000(EXTPHYSMEM)~0x???`部分空闲，部分已被使用。`npages_extmem`记录extmem的页数。

整个逻辑实现为

```
void page_init(void) {
  // The example code here marks all physical pages as free.
  // However this is not truly the case.  What memory is free?
  //  1) Mark physical page 0 as in use.
  //     This way we preserve the real-mode IDT and BIOS structures
  //     in case we ever need them.  (Currently we don't, but...)
  //  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
  //     is free.
  //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
  //     never be allocated.
  //  4) Then extended memory [EXTPHYSMEM, ...).
  //     Some of it is in use, some is free. Where is the kernel
  //     in physical memory?  Which pages are already in use for
  //     page tables and other data structures?
  //
  // Change the code to reflect this.
```

```
  // NB: DO NOT actually touch the physical memory corresponding to
  // free pages!
  size_t i;
  for (i = 1; i < npages_basemem; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
  }
  //回看inc/memlayout.h中画的内存结构的设计,我们以一个页为单位进行分配，所以这里一个
PageInfo *对应一个页  是否使用，所以要除以1个PGSIZE
  int med = (int)ROUNDUP(((char *)pages) + (sizeof(struct PageInfo) * npages)
-
                             0xf0000000,
                         PGSIZE) /
           PGSIZE;
  cprintf("pageinfo size: %d\n", sizeof(struct PageInfo));
  cprintf("%x\n", ((char *)pages) + (sizeof(struct PageInfo) * npages));
  cprintf("med=%d\n", med);
  //回看函数i386_detect_memory()可知npages = (EXTPHYSMEM / PGSIZE) +
npages_extmem
  for (i = med; i < npages; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
  }
}
```

4 page_alloc()

下面开始实现`page_alloc`注释中有清零条件`(alloc_flags & ALLOC_ZERO)`,以及超界返回NULL

思路为 从链表上取头部 如果非空且需要初始化，则通过辅助函数初始化为零。对`free_list`移动并返回申请到的PageInfo。

实现如下

```
// Allocates a physical page.  If (alloc_flags & ALLOC_ZERO), fills the entire
// returned physical page with '\0' bytes.  Does NOT increment the reference
// count of the page - the caller must do these if necessary (either
explicitly
// or via page_insert).
//
// Returns NULL if out of free memory.
//
// Hint: use page2kva and memset
struct PageInfo *page_alloc(int alloc_flags) {
  if (page_free_list) {
    struct PageInfo *ret = page_free_list;
    page_free_list = page_free_list->pp_link;
    if (alloc_flags & ALLOC_ZERO)
```

```
      memset(page2kva(ret), 0, PGSIZE);
    return ret;
  }
  return NULL;
}
```

5 page_free()

free 和alloc对应，需要把一页 重新加入`free_list`,实现如下

```
// Return a page to the free list.
// (This function should only be called when pp->pp_ref reaches 0.)
//
void page_free(struct PageInfo *pp) {
  pp->pp_link = page_free_list;
  page_free_list = pp;
}
```

测试`make grade` MIT可以看到

```
Physical page allocator: OK
```

Part 2: Virtual Memory

Exercise 2

```
Exercise 2. Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if
you haven't done so already. Read the sections about page translation and
page-based protection closely (5.2 and 6.4). We recommend that you also skim
the sections about segmentation; while JOS uses the paging hardware for
virtual memory and protection, segment translation and segment-based
protection cannot be disabled on the x86, so you will need a basic
understanding of it.
```

Thoughts After Reading

x86保护模式下的内存管理结构,有: 逻辑地址(虚拟地址)->线性地址->物理地址

通常，线性地址(linear)=逻辑地址(virtual)+段首地址()

在x86中，`线性地址=逻辑地址+ 0x00000000`,逻辑地址和线性地址相等,因此该部分的关注点在于利用页来做转换，看做虚拟地址=线性地址就好了。

接下来，是实现线性地址到物理地址的部分的转换：

- 首先CPU得到一个地址，看看页的开关开没有(该开关在CR0上，见 entry.S的设置) 没有的话就直接视为物理地址访问了
- 而打开了的话则 把这个地址给MMU

- MMU 去TLB(目前的lab还未有) 如果找到了那就好咯，访问对应物理地址
- 没有找到的话，告诉CPU，CPU去安排该虚拟地址对应的物理地址，安排好了,写入TLB,返回MMU

对于具体的一个地址32位 [31..22]为DIR，[21..12]为page，[11..0]为offset偏移量

- 第一步通过CR3存的地址(entry.S中设置)找到PAGE DIRECTORY页目录
- 通过DIR作为偏移量 定位到 PAGE DIRECTORY中的具体一项DIR ENTRY
- 以该项的值找到PAGE TABLE页表
- 通过page作为偏移量 定位到PAGE TABLE的具体一项 PAGE TABLE ENTRY
- 以该项的值定位到物理页
- 以offset 定位到该物理页的物理地址

页表的每一项是一个32位数，页表本身也是一个页，因此4KB页能存4KB÷32bit=1024项，一项对应一页，一共可以映射1024×4KB=4MB的虚拟内存

上面的DIR，page，offset分化则[31..12]就算分为两层 也是共同确定一页，可以确定2的20次方页，也就一共可以映射2^20×4KB=4GB的虚拟内存，也就是1M的页，和4GB虚拟内存相关。

Exercise 3

```
Exercise 3. While GDB can only access QEMU's memory by virtual address, it's
often useful to be able to inspect physical memory while setting up virtual
memory. Review the QEMU monitor commands from the lab tools guide, especially
the xp command, which lets you inspect physical memory. To access the QEMU
monitor, press Ctrl-a c in the terminal (the same binding returns to the
serial console).

Use the xp command in the QEMU monitor and the x command in GDB to inspect
memory at corresponding physical and virtual addresses and make sure you see
the same data.

Our patched version of QEMU provides an info pg command that may also prove
useful: it shows a compact but detailed representation of the current page
tables, including all mapped memory ranges, permissions, and flags. Stock QEMU
also provides an info mem command that shows an overview of which ranges of
virtual addresses are mapped and with what permissions.
```

使用GDB的xp命令来查看物理地址数据的

```
xp/Nx paddr
  Display a hex dump of N words starting at physical address paddr. If N is
omitted, it defaults to 1. This is the physical memory analogue of GDB's x
command.
```

尝试指令info pg，info mem

```
(qemu) info mem
00000000-00400000 00400000 -rw
f0000000-f0400000 00400000 -rw
```

```
(qemu) info pg
 |-- PTE(000400) 00000000-00400000 00400000 -rw
 |-- PTE(000400) f0000000-f0400000 00400000 -rw
(qemu) info registers
EAX=ffffffff EBX=f0117544 ECX=ffffffff EDX=f0100260
ESI=f010023e EDI=f0117340 EBP=f0114dd8 ESP=f0114dc0
EIP=f0100295 EFL=00000046 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300
CS =0008 00000000 ffffffff 00cf9a00
SS =0010 00000000 ffffffff 00cf9300
DS =0010 00000000 ffffffff 00cf9300
FS =0010 00000000 ffffffff 00cf9300
GS =0010 00000000 ffffffff 00cf9300
LDT=0000 00000000 0000ffff 00008200
TR =0000 00000000 0000ffff 00008b00
GDT=     00007c4c 00000017
IDT=     00000000 000003ff
CR0=e0010011 CR2=00000000 CR3=00115000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
```

本项目中`uintptr_t`表示虚拟地址，而`physaddr_t`表示物理地址(这两个实际都是`uint32_t`)

但是kernel只应当把`uintptr_t`转换为指针,也就是虚拟地址的指针，而物理地址要通过MMU和配置的表等去转换，而不能让kernel直接操作。

C type	Address type
`T*`	Virtual
`uintptr_t`	Virtual
`physaddr_t`	Physical

Question 1 & Answer

Assuming that the following JOS kernel code is correct, what type should variable x have, `uintptr_t or physaddr_t`?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

应为`uintptr_t`，因为对于程序来说只有虚拟地址

Excercise 4

```
Exercise 4. 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()

check_page(), called from mem_init(), tests your page table management
routines. You should make sure it reports success before proceeding.
```

Excercise 4通过实现下列函数，来进行页表管理`check_page()`用来测试正确性。

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

1 pgdir_walk()

`pgdir_walk()`需要做一个二级页表，该函数需要返回一个PTE指针(linear address)

```
// Given 'pgdir', a pointer to a page directory, pgdir_walk returns
// a pointer to the page table entry (PTE) for linear address 'va'.
// This requires walking the two-level page table structure.
//
// The relevant page table page might not exist yet.
// If this is true, and create == false, then pgdir_walk returns NULL.
// Otherwise, pgdir_walk allocates a new page table page with page_alloc.
//    - If the allocation fails, pgdir_walk returns NULL.
//    - Otherwise, the new page's reference count is incremented,
//  the page is cleared,
//  and pgdir_walk returns a pointer into the new page table page.
//
// Hint 1: you can turn a Page * into the physical address of the
// page it refers to with page2pa() from kern/pmap.h.
```

```
//
// Hint 2: the x86 MMU checks permission bits in both the page directory
// and the page table, so it's safe to leave permissions in the page
// more permissive than strictly necessary.
//
// Hint 3: look at inc/mmu.h for useful macros that mainipulate page
// table and page directory entries.
```

函数拿到一个虚拟地址va和一个页目录，也就是最外层页表，需要返回一个指向下一层页表的指针,也就是下一层页表的地址。

那么函数的实现可以分为以下步骤

- 把va分段提取 DIR
- 根据DIR 的到 一个具体的ENTRY
- 如果需要分配且没分配则分配
- 否则返回地址ENTRY中记录的地址，或者没分配返回NULL

分配的过程

- 申请一个页用Page结构,这个页用于作为页表
- 获取该空闲页的真实物理地址，用或操作对权限位等进行设置
- 修改好页目录项
- 最后在页目录项写好以后，返回指向的页表中的根据va分段算出page得到的具体的一个pte
- 这样就有一个空的页表，和指向它的新的页目录项，

实现代码如下

```
pte_t *pgdir_walk(pde_t *pgdir, const void *va, int create) {
  int dindex = PDX(va), tindex = PTX(va);
  // dir index, table index
  if (!(pgdir[dindex] & PTE_P)) { // if pde not exist
    if (create) {
      struct PageInfo *pg = page_alloc(ALLOC_ZERO); // alloc a zero page
      if (!pg)
        return NULL; // allocation fails
      pg->pp_ref++;
      pgdir[dindex] = page2pa(pg) | PTE_P | PTE_U | PTE_W;
    } else
      return NULL;
  }
  pte_t *p = KADDR(PTE_ADDR(pgdir[dindex]));
  return p + tindex;
}
```

2 boot_map_region()

`boot_map_region` 把`[va, va+size)`的虚拟地址映射到`[pa, pa+size)`的物理地址,要设置`perm|PTE_P`位,这里的参数size是PGSIZE的倍数,这个是为了设置静态的在UTOP之上的映射,它不应该修改`pp_ref`

那用上函数`pgdir_walk()`,可以得到一个 page table entry，我们只要把对应page table entry中记录的地址写为pa即可

```
// Map [va, va+size) of virtual address space to physical [pa, pa+size)
// in the page table rooted at pgdir.  Size is a multiple of PGSIZE.
// Use permission bits perm|PTE_P for the entries.
//
// This function is only intended to set up the ``static'' mappings
// above UTOP. As such, it should *not* change the pp_ref field on the
// mapped pages.
//
// Hint: the TA solution uses pgdir_walk
static void boot_map_region(pde_t *pgdir, uintptr_t va, size_t size,
                            physaddr_t pa, int perm) {
  int i;
  //cprintf("Virtual Address %x mapped to Physical Address %x\n", va, pa);
  for (i = 0; i < size / PGSIZE; ++i, va += PGSIZE, pa += PGSIZE) {
    pte_t *pte = pgdir_walk(pgdir, (void *)va, 1); // create
    if (!pte)
      panic("boot_map_region panic, out of memory");
    *pte = pa | perm | PTE_P;
  }
  //cprintf("Virtual Address %x mapped to Physical Address %x\n", va, pa);
}
```

这样 我们就完全 把虚拟地址和物理地址连接上了

3 page_lookup()

`page_lookup`返回`PageInfo *`,在参数中实际还返回了虚拟地址va对应的页表项pte的地址,注释提示我们使用函数`pa2page`,功能是通过va获取`PageInfo *`和`pte的地址`。

实现如下

```
// Return the page mapped at virtual address 'va'.
// If pte_store is not zero, then we store in it the address
// of the pte for this page.  This is used by page_remove and
// can be used to verify page permissions for syscall arguments,
// but should not be used by most callers.
//
// Return NULL if there is no page mapped at va.
//
// Hint: the TA solution uses pgdir_walk and pa2page.
//
```

```
struct PageInfo *page_lookup(pde_t *pgdir, void *va, pte_t **pte_store) {
  pte_t *pte = pgdir_walk(pgdir, va, 0); // not create
  if (!pte || !(*pte & PTE_P))
    return NULL; // page not found
  if (pte_store)
    *pte_store = pte; // found and set
  return pa2page(PTE_ADDR(*pte));
}
```

4 page_remove()

`page_remove` 把va对应的物理地址解绑，如果对应的没有分配就什么也不做，注释中提示我们使用函数`page_lookup,tlb_invalidate, page_decref`

细节

- 此处`pp_ref`需要减1
- 如果`pp_ref==0`需要被free掉 用part1中的函数
- 如果有对应的 pte也需要设为0。
- 如果从页表中去掉一个页表项，则相应的TLB表也需要进行修改。

实现如下

```
// Unmaps the physical page at virtual address 'va'.
// If there is no physical page at that address, silently does nothing.
//
// Details:
//   - The ref count on the physical page should decrement.
//   - The physical page should be freed if the refcount reaches 0.
//   - The pg table entry corresponding to 'va' should be set to 0.
//     (if such a PTE exists)
//   - The TLB must be invalidated if you remove an entry from
//     the page table.
//
// Hint: The TA solution is implemented using page_lookup,
//  tlb_invalidate, and page_decref.
//
void page_remove(pde_t *pgdir, void *va) {
  pte_t *pte;
  struct PageInfo *pg = page_lookup(pgdir, va, &pte);
  if (!pg || !(*pte & PTE_P))
    return; // page not exist
  //   - The ref count on the physical page should decrement.
  //   - The physical page should be freed if the refcount reaches 0.
  page_decref(pg);
  //   - The pg table entry corresponding to 'va' should be set to 0.
  *pte = 0;
  //   - The TLB must be invalidated if you remove an entry from
  //     the page table.
  tlb_invalidate(pgdir, va);
```

```
}
```

5 page_insert()

`page_insert`,把物理地址和虚拟地址做映射

实现步骤如下：

- 如果va已经映射了就解绑。
- 如果对应的pgdir之类的里面都没有的话，就分配被插入
- 如果成功映射就`pp_ref`+1

注释提示我们使用函数`pgdir_walk, page_remove, page2pa`

返回值：成功0，失败`-E_NO_MEM`

具体步骤如下：

- 首先 `pgdir_walk`获得pte 失败就返回`-E_NO_NEM`
- 先对`pp->pp_ref`增加
- 如果有，再解除原来的va关系
- 最后建立新的映射

实现如下

```
// Map the physical page 'pp' at virtual address 'va'.
// The permissions (the low 12 bits) of the page table entry
// should be set to 'perm|PTE_P'.
//
// Requirements
//   - If there is already a page mapped at 'va', it should be page_remove()d.
//   - If necessary, on demand, a page table should be allocated and inserted
//     into 'pgdir'.
//   - pp->pp_ref should be incremented if the insertion succeeds.
//   - The TLB must be invalidated if a page was formerly present at 'va'.
//
// Corner-case hint: Make sure to consider what happens when the same
// pp is re-inserted at the same virtual address in the same pgdir.
// However, try not to distinguish this case in your code, as this
// frequently leads to subtle bugs; there's an elegant way to handle
// everything in one code path.
//
// RETURNS:
//   0 on success
//   -E_NO_MEM, if page table couldn't be allocated
//
// Hint: The TA solution is implemented using pgdir_walk, page_remove,
// and page2pa.
//
int page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm) {
  pte_t *pte = pgdir_walk(pgdir, va, 1); // create on demand
```

```
  if (!pte)                                // page table not allocated
    return -E_NO_MEM;
  // increase ref count to avoid the corner case that pp is freed before it is
  // inserted.
  pp->pp_ref++;
  if (*pte & PTE_P) // page colides, tle is invalidated in page_remove
    page_remove(pgdir, va);
  *pte = page2pa(pp) | perm | PTE_P;
  return 0;
}
```

运行`make qemu-nox`可以看到

```
check_page_alloc() succeeded!
check_page() succeeded!
```

执行`make grade`可以看到`Page management: OK`

Part 3: Kernel Address Space

JOS把32位线性地址分为两部分，用户地址(物理地址高 虚拟地址低)，内核地址(物理地址低 虚拟地址高),具体分割线为`inc/memlayout.h`中的`ULIM`：

```
 *    ULIM     ------> +------------------------------+ 0xef800000      --+
```

内核地址预留了约256MB，内核地址对用户地址是完全控制.

Permissions and Fault Isolation

在页表上通过设置权限位来保证用户态的错误不会操作到内核态的数据，从而引起kernel崩溃

用户态对ULIM以上的部分无权限

对于`[UTOP,ULIM)`之间的内核和用户都有权限读，但都无权限写，其实在kernel初始化的时候会写的，这部分用来表示内核数据结构的一些信息

低于UTOP的就算是用户可以进行读和写的。

Excercise 5

```
Exercise 5. Fill in the missing code in mem_init() after the call to
check_page().

Your code should now pass the check_kern_pgdir() and
check_page_installed_pgdir() checks.
```

要求把UTOP以上的虚拟地址进行适当的映射，把`mem_init()`中`check_page()`调用以后的代码补全。

- 第一个补全映射，UPAGES是一个分界线。那么要映射这一块的地址，通过看`inc/memlayout.h`的图知道了这一块大小为PTSIZE,再利用刚刚实现的`boot_map_region`函数，实现如下

```
  // Map 'pages' read-only by the user at linear address UPAGES
  // Permissions:
  //    - the new image at UPAGES -- kernel R, user R
  //      (ie. perm = PTE_U | PTE_P)
  //    - pages itself -- kernel RW, user NONE
  // Your code goes here:
  boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
  cprintf("PADDR(pages) %x\n", PADDR(pages));
```

- 下面两个映射同理，对照`inc/memlayout.h`以及注释确定每一个变量，和上面一样的映射方法，实现分别如下：

```
  // 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_W);
  cprintf("PADDR(bootstack) %x\n", PADDR(bootstack));
```

```
  // 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:

  boot_map_region(kern_pgdir, KERNBASE, -KERNBASE, 0, PTE_W);
```

至此 运行 `make grade` 已经满分

```
>make grade
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

Question 2

```
What entries (rows) in the page directory have been filled in at this point?
What addresses do they map and where do they point? In other words, fill out
this table as much as possible:
```

Entry	Base Virtual Address	Points to (logically):
1023	0xffc00000	Page table for top 4MB of phys memory
1022	0xff800000	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

现在page directory里已经有哪些了? 映射了哪些地址 指向了哪些地方？请完成填表。

Answer 2

Until now, we have just filled some entries in page directory.

First, we recursively insert PD in itself as a page table, to form a virtual page table at virtual address UVPT.

```
kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

This means we fill entry(0x3BD), its base virtual address is UVPT(0xef400000), points to `kern_pgdir`

Then we map 'pages' read-only by the user at linear address UPAGES This means we fill entry(0x3BC), its base virtual address is UPages(0xef000000), points to `pages`

Next we use the physical memory that 'bootstack' refers to as the kernel stack. This means we fill entry(0x3BF), its base virtual address is MMIOLIM(0xefc00000), points to `bootstack`

Finally we map all of physical memory at KERNBASE This means we fill entry(0x3C0-0x3FF), its base virtual address is MMIOLIM(0xefc00000), points to `kernel`

Question 3 & Answer 3

```
We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?
```

通常来说，操作系统通过两种方式实现对内核空间的保护，一种是分段式，一种是分页式，在JOS中，我们使用分页式，当PTE_U不被允许时，用户不的方位内核的内存空间。

Question 4 & Answer 4

```
What is the maximum amount of physical memory that this operating system can support? Why?
```

Since JOS use 4MB UPAGES space to store all the PageInfo struct information, each struct is 8B, so we can store 512K PageInfo structs. The size of a page is 4KB, so we can have most `512K * 4KB = 2GB` physical memory.

JOS使用 4MB UPAGES的空间去存储所有的PageInfo结构，存储单个PageInfo结构需要8字节，所以我们能存储512个PageInfo结构，每个PageInfo对应一页，每个页的大小为4KB，所以我们最多能表示 `512K * 4KB = 2GB` 的物理内存。

Question 5 & Answer 5

```
How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?
```

如果我们有2GB的物理内存，则我们需要 4MB的PageInfo去管理内存，2MB用于page table， 4KB用于page directory。

Question 6 & Answer 6

```
Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?
```

在 `jmp *%eax` 结束后，仍是可行的，因为entry_pgdir也从va[0-4M)映射到了pa[0-4M)。这样做有必要，因为不久kern_pgdir将被加载，va[0-4M)也会被抛弃。

总结

- `Part I` 是实现了 让我们不再看到硬件 对齐上面封装，用Page 和与page相关的宏来管理
- `Part II` 则实现了 页表相关，指定虚拟地址和物理地址的映射接口，还提供指定Page和va 创建/查看/删除页的接口
- `Part III`把非用户的虚拟地址 完全映射掉,实际的pages等，在物理地址中还是只有一份，但现在应该是有2~3个虚拟地址都指向它[和它物理地址相等的，KERNBASE以上的，最后新建的映射的]

参考

- Translation lookaside buffer
- Page Translation
- Chapter 5 Memory Management
- Memory management unit
- Translating a Virtual Address to a Physical Address
- Page_table
- How are same virtual address for different processes mapped to different physical addresses
- In virtual memory, can two different processes have the same address?
- Page Size Extension
- 逻辑地址、虚拟地址、物理地址以及内存管理