Lab 2: Memory Management

介绍

在这个实验中，你将为你的操作系统写一些 *内存管理代码* 。内存管理由两部分组成。

第一部分是为了内核的物理内存分配器，因此内核可以分配内存并且释放。你的分配器将操作叫做 *pages* 的4096个bytes的单元。你的任务是维持数据结构，这个数据结构记录了哪些物理页是空闲和哪些是被分配的，以及有多少进程正在分享每一个被分配的页。你也会写一些常规例程来分配和释放内存页。

第二个部分是 *虚拟内存* ，它映射了内核和用户软件使用的虚拟地址到物理内存地址。x86的内存管理单元（MMU）在指令使用内存时，执行映射，通过查询一个页表集。你将根据我们提供的特例化修改 `JOS` 来设置 `MMU` 的页表。

Getting started

在这个和之后的lab你将一步一步搭建你的内核。我们也会给你提供一些额外的资源。获取那些资源，使用 `git` 提交自从上交lab1之后的修改（如果有的话），获取课程库的最新版本，然后创建一个本地分支lab2基于我们的lab2分支， `origin/lab2` ：

```
cd ~/6.828/lab
add git
git pull
# 如果发生错误 设置一下环境变量
export GIT_SSL_NO_VERIFY=1
Already up-to-date.
git checkout -b lab2 origin/lab2
Branch lab2 set up to track remote branch refs/remotes/origin/lab2.
Switched to a new branch "lab2"
```

你需要将你在 `lab1` 分支做的修改合并到 `lab2` 分支

```
git merge lab1
```

在一些情况下， `git` 可能不能弄明白如何将你的更改和新的实验室作业合并。这种情况， `git commit` 命令将告诉你那些文件是冲突的，你应该首先解决冲突（编辑相关文件）然后用命令 `git commit -a` 来提交最终文件。

Lab2 包含了以下新文件，你应该先浏览一下：

```
inc/memlayout.h
kern/pmap.c
kern/pmap.h
kern/kclock.h
kern/kclock.c
```

`memlayout.h` 描述了虚拟地址空间的布局，这些你必须通过修改 `pmap.c` 和 `memlayout.h` 来实现。 `pamp.h` 定义了 `PageInfo` 结构体，你将会用来跟踪哪些物理内存是空闲的。 `kclock.c` 和 `kclock.h` 操作PC的电池供电时钟和 CMOS RAM 硬件，这是BIOS记录PC所包含物理内存数量的地方。为了弄清楚有多少物理内存， `pmap.c` 中的代码需要去读取这个设备硬件，但是那部分已经为你做好了：你不需要知道 CMOS 硬件工作的细节。

尤其需要注意的是 `memlayout.h` 和 `pmap.h` ，因为这个lab需要你使用和理解很多它们包含的定义。你可能也会想去复习 `inc/mmu.h` ，因为它也包含了一些对这个lab有用的定义。

在开始这个lab之前，不要忘记 `add -f 6.828` 来获取6.828版本的QEMU。

Lab Requirements

在这个lab以及后续的lab，这些lab中有常规的练习和 *至少一个* 有挑战性的问题。额外的，写下 `lab` 中每个问题 ***简短的答案*** 以及 ***一段描述*** 来解释你做了什么来解决你选择的有挑战性的问题。将写下的内容放在一个叫做 `answers-lab2.txt` 的文件中，这个文件位于 `lab` 文件夹的第一层级。

Hand- In Procedure

当你准备提交你的实验代码和笔记，将 `answers-lab2.txt` 加入Git库，提交你的修改，然后运行 `make handin` 。

```
git add answers-lab2.txt
git commit -am "my answer to lab2"
make handin
```

Part 1: Physical Page Management

操作系统必须追踪物理内存中哪些是空闲的，哪些正在使用。 `JOS` 使用 `page granularity` 来管理PC的物理内存所以它可以使用 `MMU` 来映射和保护每一个分配的内存。

你现在将要写一个物理页分配器。他用一个 `struct PageInfo` 对象的链表来持续跟踪哪些页是空闲的，每个都对应一个物理页。

在你可以写剩下的虚拟内存实现之前，你需要写**物理页分配器**，因为你的页表管理代码将需要分配物理内存来存储页表。

Exercise 1.

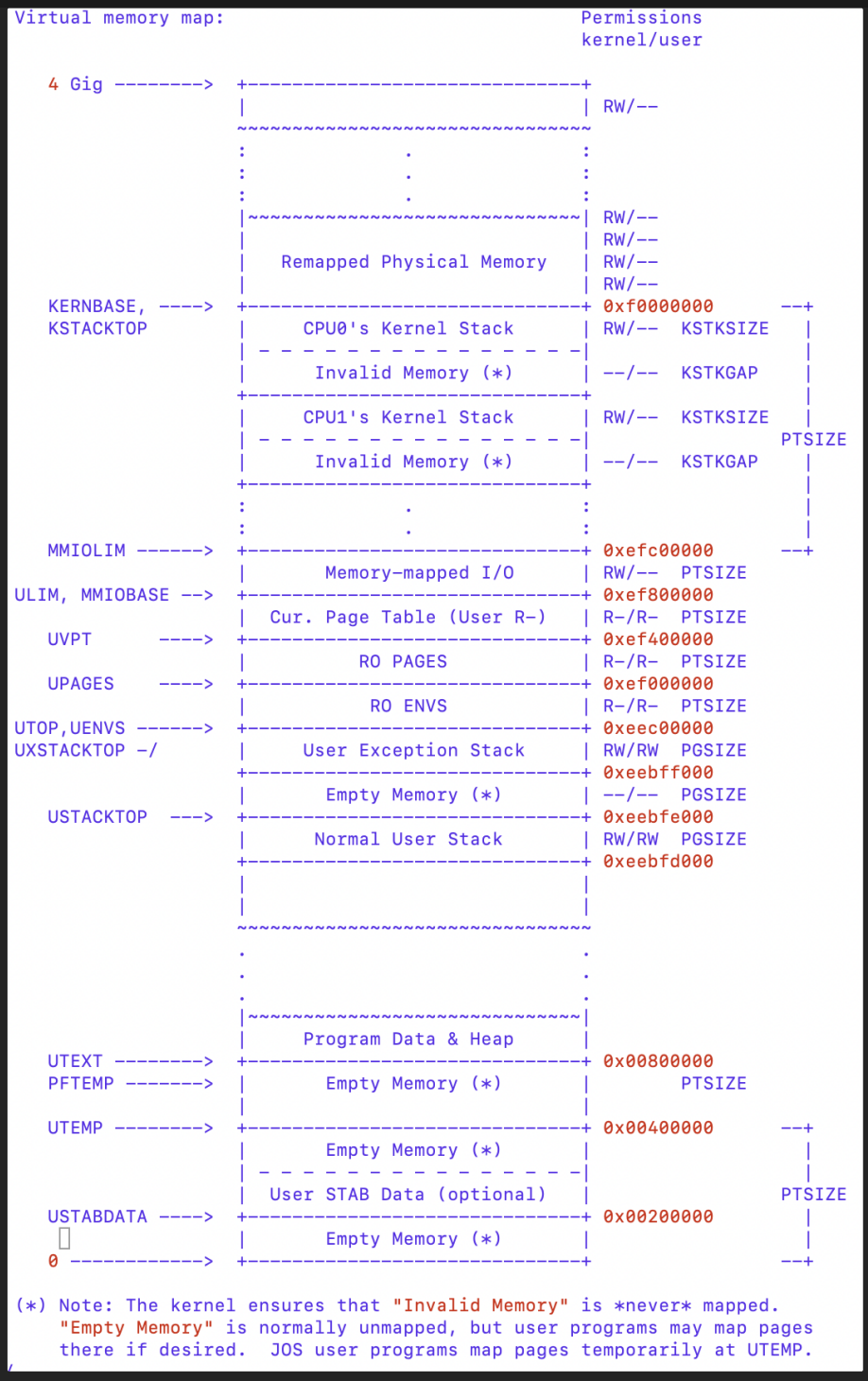
In the file `kern/pmap.c`, you must implement code for the following functions (probably in the order given).

`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.

实验理解

- 信息整理
    - 内存空间的分布情况


Virtual memory map:
Permissions
kernel/user
4 Gig
RW/--
Remapped Physical Memory
KERNBASE,
KSTACKTOP
CPU0's Kernel Stack
RW/--
KSTKSIZE
Invalid Memory (*)
--/--
KSTKGAP
CPU1's Kernel Stack
PTSIZE
MMIOLIM
0xefc00000
Memory-mapped I/O
ULIM, MMIOBASE
0xef800000
Cur. Page Table (User R-)
R-/R-
UVPT
0xef400000
RO PAGES
UPAGES
0xef000000
RO ENVS
UTOP,UENVS
0xeec00000
UXSTACKTOP -/
User Exception Stack
RW/RW
PGSIZE
0xeebff000
Empty Memory (*)
USTACKTOP
0xeebfe000
Normal User Stack
0xeebfd000
Program Data & Heap
UTEXT
0x00800000
PFTEMP
0x00400000
UTEMP
User STAB Data (optional)
USTABDATA
0x00200000
0
(*) Note: The kernel ensures that "Invalid Memory" is *never* mapped.
"Empty Memory" is normally unmapped, but user programs may map pages
there if desired. JOS user programs map pages temporarily at UTEMP.

- 页信息结构体

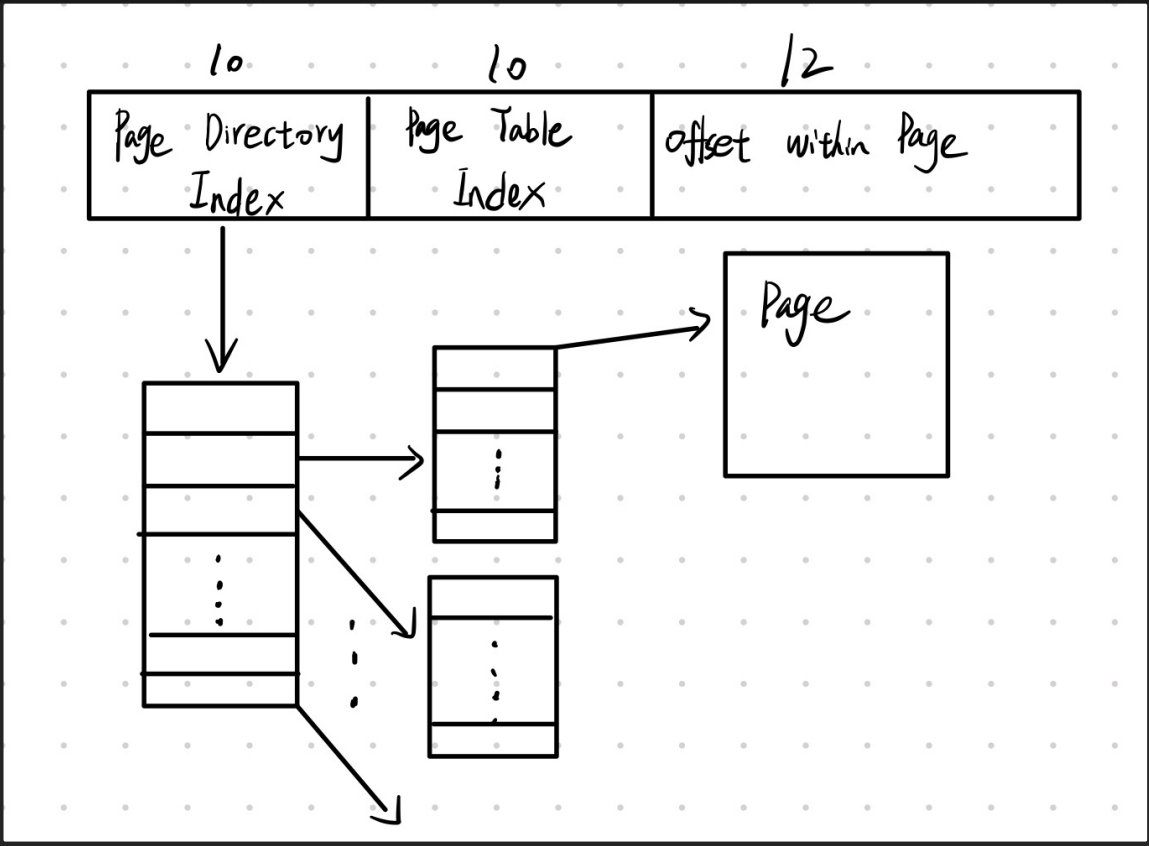
```
struct PageInfo {
        // Next page on the free list.
        struct PageInfo *pp_link;

        // pp_ref is the count of pointers (usually in page
table entries)
        // to this page, for pages allocated using
page_alloc.
        // Pages allocated at boot time using pmap.c's
        // boot_alloc do not have valid reference count
fields.

        uint16_t pp_ref; // 0 表示空闲，非 0 表示被多少人使用
};
```

- 虚拟地址的表示

```
// 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) ----------/
//
// The PDX, PTX, PGOFF, and PGNUM macros decompose linear addresses as shown.
// To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
// use PGADDR(PDX(la), PTX(la), PGOFF(la)).
```



- `boot_alloc()` 函数

  此函数在 *字节* 这个层级进行内存分配，所以在分配时需要计算分配字节的大小。

  通过移动 `nextfree` 的地址来实现内存的分配。

```
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
        /* 这里的 end 也就是 bss 段，是第一个没有被链接器分配的虚拟地
址。 */
        // 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.
        result = nextfree;
        if(0==n){
                return result;
        }
        if(0<n){
                if(npages * PGSIZE < n){
                        panic("boot_alloc out of memory");
                }
                nextfree = ROUNDUP(result + n, PGSIZE);
                cprintf("boot_alloc memory at %x, next memory
    allocate at %x\n", result, nextfree);
                return result;
        }
        return NULL;
    }
```

- `mem_init()` 函数

  这个函数仅仅设置大于 `UTOP` 的内存地址，也就是地址空间的 `kernel` 部分。用户部分会在后面再设置。

```
struct PageInfo *pages;         // Physical page state array
                                // 用于存放分配的页
static struct PageInfo *page_free_list; // Free list of
physical pages
                                        // 空闲物理页的链表


// Set up a two-level page table:
//    kern_pgdir is its linear (virtual) address of the root
//
```

```
// This function only sets up the kernel part of the address space
// (ie. addresses ≥ UTOP).  The user part of the address space
// will be set up later.
//
// From UTOP to ULIM, the user is allowed to read but not write.
// Above ULIM the user cannot read or write.
void
mem_init(void)
{
        uint32_t cr0;
        size_t n;

        // Find out how much memory the machine has (npages & npages_basemem).
        i386_detect_memory();

        // Remove this line when you're ready to test this function.
        panic("mem_init: This function is not finished\n");


 ////////////////////////////////////////////////////////////////////
        // create initial page directory.
        kern_pgdir = (pde_t *) boot_alloc(PGSIZE);  // 这里应该是上图中Page_Directory_Index
        memset(kern_pgdir, 0, PGSIZE);


 ////////////////////////////////////////////////////////////////////
      // Recursively insert PD in itself as a page table, to form
      // a virtual page table at virtual address UVPT.
      // (For now, you don't have understand the greater purpose of the
      // following line.)
```

```
	// Permissions: kernel R, user R
	kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;


 ////////////////////////////////////////////////////////////////////
/////
	// 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.  Use memset
	// to initialize all fields of each struct PageInfo to 0.
	// Your code goes here:

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

	/* struct PageInfo phy_pages[npages];
	memset(phy_pages, 0, npages);
	for(int i = 0; i < npages; ++i){
		pages[i] = phy_pages[i];
	} */
```

- `page_init(void)`函数

  注意：`memlayout.h`中的地址布局是**虚拟地址**的布局，真正的**物理内存地址**的布局还在`lab1`中。

  而**物理地址**在**虚拟地址**中的映射都是从`kernelbase`开始的，即将`kernelbase`视为0。

  ```c
  // Initialize page structure and memory free list.
```

```
  // After this is done, NEVER use boot_alloc again.  ONLY use the
page
  // allocator functions below to allocate and deallocate physical
  // memory via the page_free_list.
  //
  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!
          // 具体布局可以参看lab1的物理内存分布
          size_t i;
          i = 0;
          // 1)
          page[i]->pp_ref = 1;
```

```
        pages[i]->pp_link = NULL;

        // 2) [PGSIZE, npages_basemem * PGSIZE] is free
        // PGSIZE = 4k
        i = PGNUM(PGSIZE);
        for(; i < npage_basemem; ++i){
                pages[i]->pp_ref = 0;
                pages[i]->pp_link = page_free_list;
                page_free_list = &pages[i];
        }

        // 3) [IOPHYSMEM, EXTPHYSMEM] is allocated, so we set
the pp_ref = 1, pp_link = NULL
        // IOPHYSMEM / PGSIZE = 160, EXTPHYSMEM / PGSIZE = 256
        size_t extphysmem_page = PGNUM(EXTPHYSMEM);
        for(; i < extphysmem_page; ++i){
                pages[i]->pp_ref = 1;
                pages[i]->pp_link = NULL;
        }

        // 4) [EXTPHYSMEM, ...]
        size_t already_in_use_page =
PGNUM(PADDR(ROUNDUP(boot_alloc(0), PGSIZE)));
        for(; i < already_in_use_page; ++i){
                pages[i]->pp_ref = 1;
                pages[i]->pp_link = NULL;
        }
        for(; i < npages; ++i){
                pages[i]->pp_ref = 0;
                pages[i]->pp_link = page_free_list;
                page_free_list = &pages[i];
        }
  }
```

- `page_allco(int alloc_flags)`

```
  //
  // 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).
//
// Be sure to set the pp_link field of the allocated page to
NULL so
// page_free can check for double-free bugs.
//
// Returns NULL if out of free memory.
//
// Hint: use page2kva and memset
struct PageInfo *
page_alloc(int alloc_flags)
{
        // Fill this function in
        if(NULL == page_free_list[0]->pp_link){
                return NULL;
        }
struct PageInfo* alloc_page = (struct
PageInfo*)page_free_list[0];
        if(alloc_flags & ALLOC_ZERO){
                memset(page2kva(alloc_page), '\0', PGSIZE);
        }
        page_free_list[0]->pp_link = page_free_list[0]-
>pp_link->pp_link;
        page_free_list[0]->pp_link = NULL;
        return alloc_page;
}
```

- `page_free(struct PageInfo*)`

```
//
// 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)
{
        // Fill this function in
        // Hint: You may want to panic if pp->pp_ref is nonzero
or
        // pp->pp_link is not NULL.
        if( 0 != pp->pp_ref | NULL != pp->pp_link){
                panic("The page may not be free.");
        }
        pp->pp_link = page_free_list;
        page_free_list = pp;
}
```

```
ubuntu@VM-4-14-ubuntu:~/documents/projects/mit6828/lab$ make qemu-nox
+ cc kern/pmap.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::25500 -D qemu.log
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
boot_alloc memory at f0117000, next memory allocate at f0118000
boot_alloc memory at f0118000, next memory allocate at f0158000
PGSIZE / 1024 is 4
PGNUM(PGSIZE) is 1
i is 160 and npages is 32768
npages_basemem is 160
extphysmem_page is 256
iophysmem_page is 160
already_in_use_page is 344
if(only_low_memory) finish
first_free_page finish.
check_page_free_list() succeeded!
check_page_alloc() succeeded!
kernel panic at kern/pmap.c:753: assertion failed: page_insert(kern_pgdir, pp1, 0x0, PTE_W) < 0
```

这个实验以及所有的 `6.828` 实验，需要你做一些侦查性的工作来搞清楚你需要做什么。这个任务不会描述出你将不得不加入 `JOS` 代码的细节。找出你必须更改的 `JOS` 源代码部分的批注；那些批注经常包含一些明确说明和暗示。

你也将需要查看 `JOS` 的相关部分、 `Intel` 使用手册或许是你 `6.004` 或 `6.003` 的笔记。

Part 2: Virtual Memory

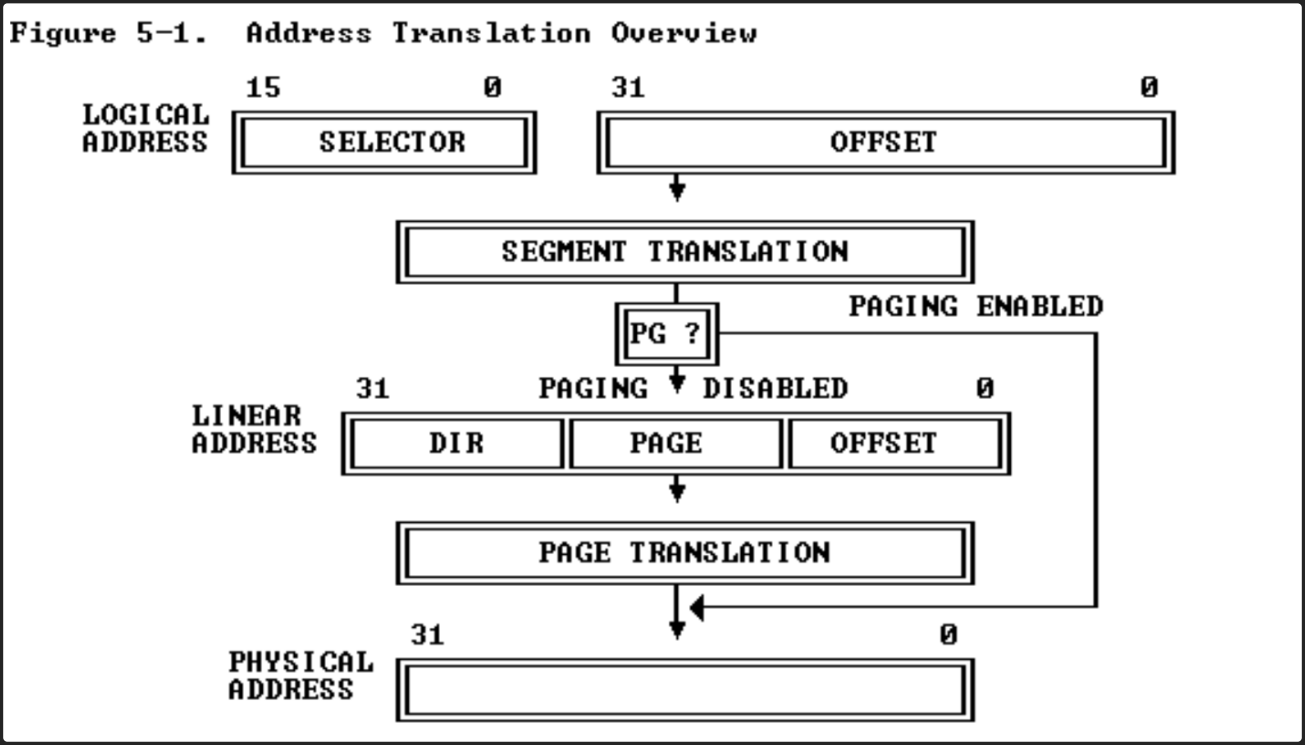
在开始之前，了解 `x86` 架构的保护模式内存管理结构，又叫做*segmentation* and *page* translation。

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.

`80386` 将逻辑地址转化为物理地址分为两步：

- *Segment translation*：将逻辑地址（由段选择器和段位移组成）转化为线性地址
- *Page translation*：将线性地址转化为物理地址（这一步是可选择的，由系统软件设计者执行决定）



5.1

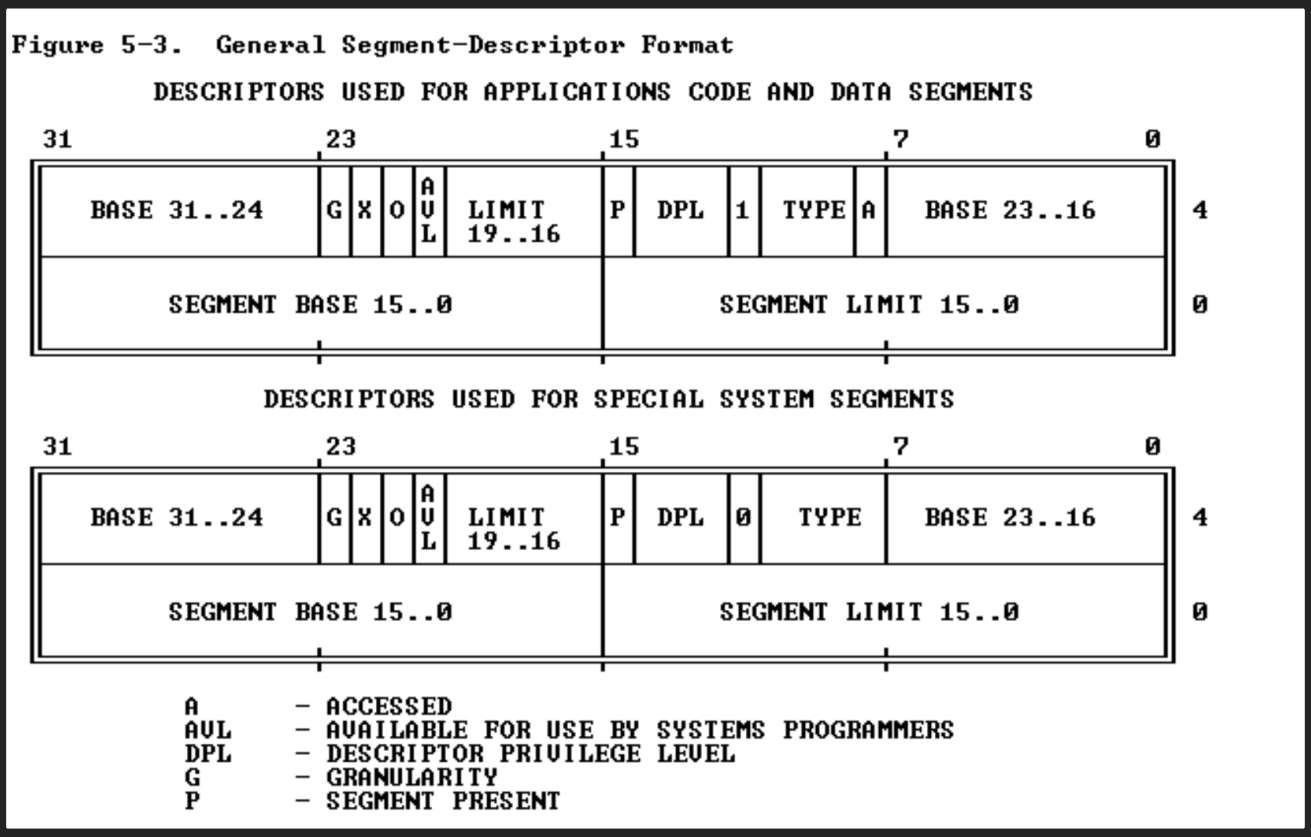
处理器用如下数据结构来将逻辑地址转化为线性地址。

- Descriptors
- Descriptor tables
- Selectors
- Segment Registers

5.1.1 Descriptors

段描述符向处理器提供从 *逻辑地址* 映射到 *线性地址* 所需要的数据。

描述符由编译器，链接器，加载器或者操作系统产生，而不是应用程序。

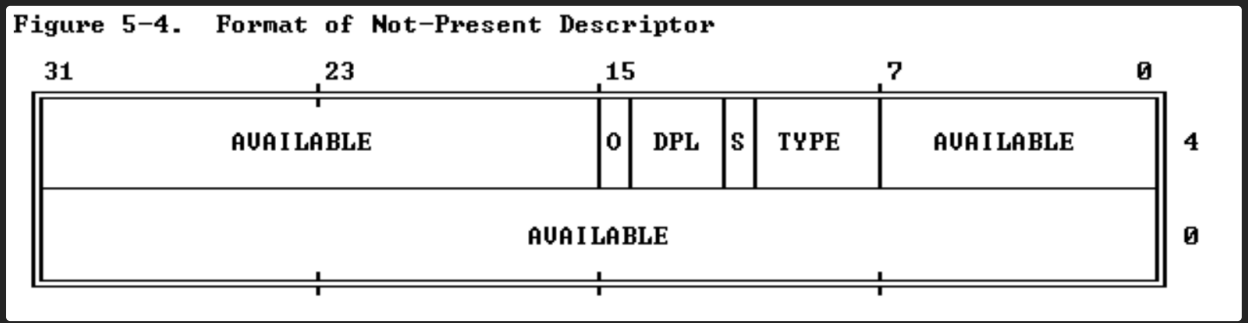


上图是描述符的两种结构。

段描述符各位含义：

- BASE：定义段的位置，在4GB的线性地址空间
- LIMIT：定义段的大小
- TYPE：区分不同类型的描述符
- DPL(Descriptor Privilege Level)：保护机制所使用
- Segment-Present bit：标记描述符对地址转换是否有效

  操作系统会在以下情况*clear* `Segment-Present bit`
  - 被段扩展的线性空间没有被 *页机制* 映射
  - 段不在内存中



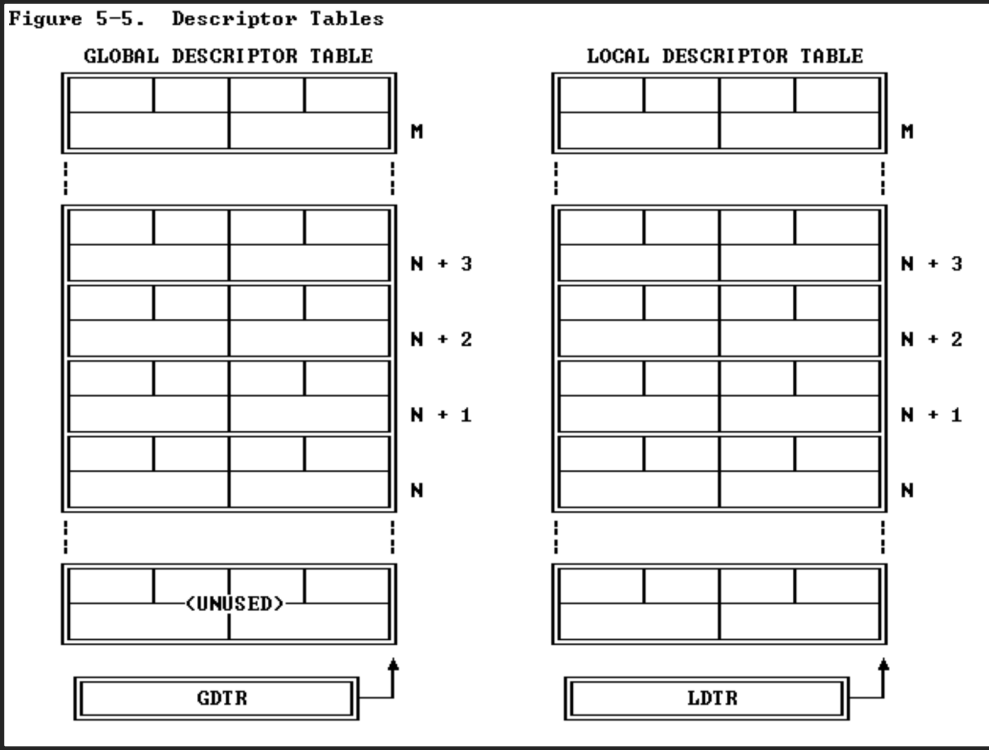
上图是 `Segment-Present bit = 0` 的情况。

- Accessed bit：当段被访问的时候，处理器设置这个位。

5.1.2 Descriptor Tables

段描述符存储在以下两种描述符表之一：

- The global descriptor table(GDT)
- A local descriptor table(LDT)

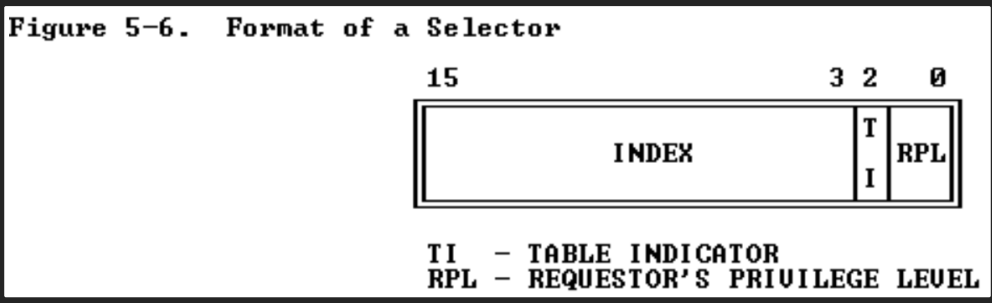


描述符表就是一个包含描述符的8字节条目的内存数组，如上图所示。

处理器通过 `GDTR` 和 `LDTR` 寄存器来定位在内存中的 `GDT` 和当前 `LDT` 。

5.1.3 Selectors

逻辑地址的段选择部分通过指定 *描述符表* 和在表中索引 *描述符* 来标识一个描述符。



上图是一个 *选择器* 的格式。

各个标志的含义如下：

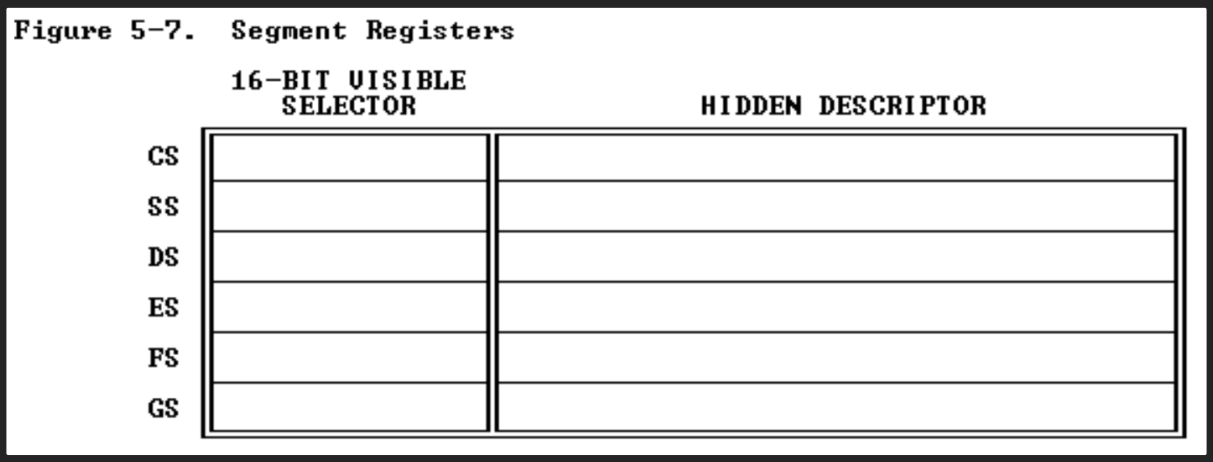
- Index：从描述符表中选择8192个描述符中的一个。

- Table Indicator: 指定选择器指向的描述符表。（0表示 `GDT` ，1表示 `current LDT` ）
- Requested Privilege Level: 保护机制所使用。

5.1.4 Segment Registers

`80386` 将来自 *描述符* 的信息存储于 *段寄存器* 中，从而避免每次访问内存时需要查阅 *描述符表* 。

Figure 5-7. Segment Registers



如上图所示，每个段寄存器都有一个 *可见* 部分和 *不可见* 部分。

可见 部分可被程序当作16位 *寄存器* 操作，而 *不可见* 部分由处理器操作。

5.2 Page Translation

这个章节实现了 *面向页* 的虚拟内存系统和 *页级别* 的保护。

只有 `CR0` 寄存器中的 `PG` 位被设置， *页翻译* 才有效。

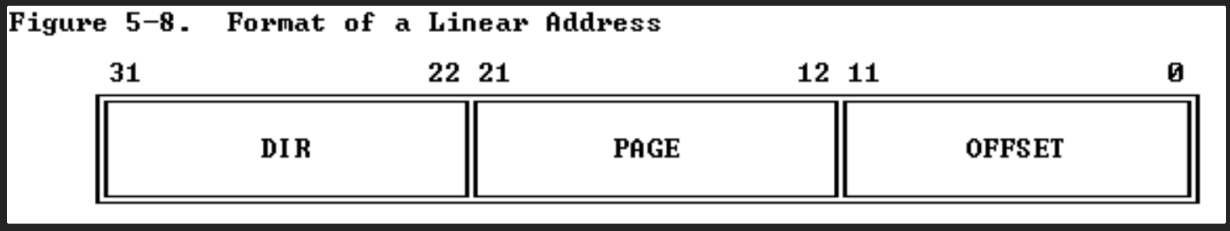
5.2.1 Page Frame

一个 *页帧* 是物理内存中连续的4K单元。页面从字节边界开始，大小固定。

5.2.2 Linear Address

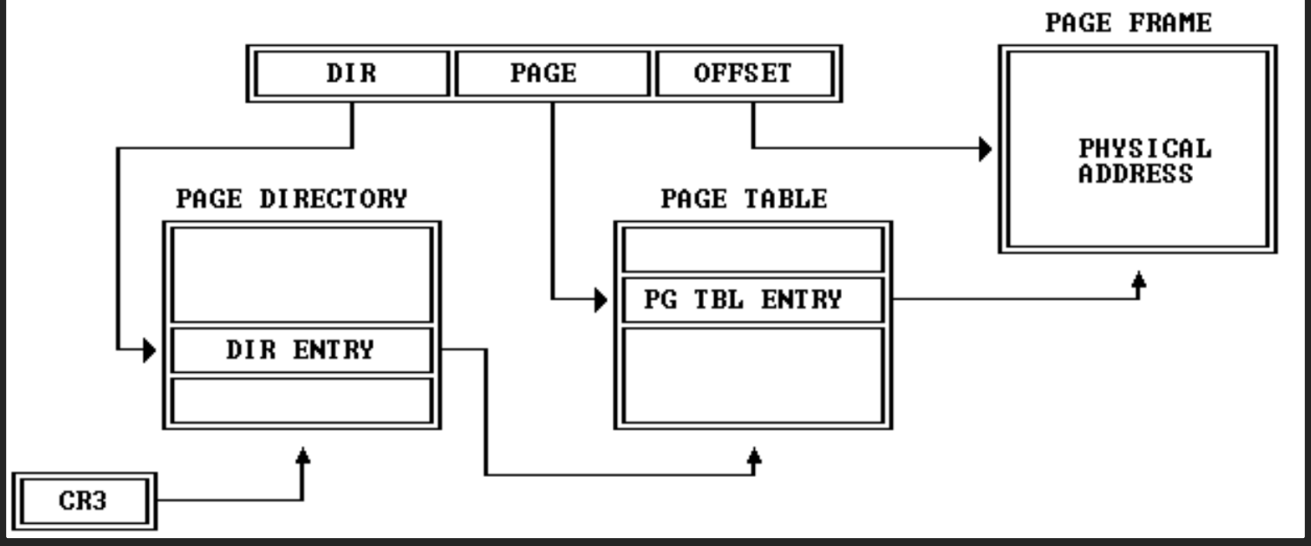
一个线性地址通过指定 *页表* ， *页表中的页* ， *页中的位移* 来直接指向一个物理地址。

Figure 5-8. Format of a Linear Address



线性地址格式如上图所示。

Figure 5-9. Page Translation



上图展示了一个处理器如何通过查阅 *两级页表* 将线性地址转换为物理地址。

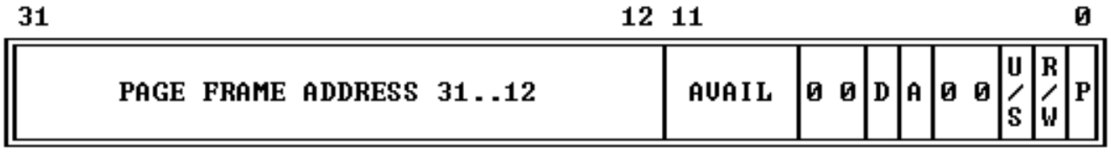
5.2.3 Page Tables

页表只是一个32位 *页说明符* 的数组。

当前 *页目录* 的 *物理地址* 存储于CPU寄存器 `CR3` ，也叫做 *页目录基寄存器（PDBR）* 。

5.2.4 Page-Table Entries

Figure 5-10. Format of a Page Table Entry



P - PRESENT
R/W - READ/WRITE
U/S - USER/SUPERVISOR
D - DIRTY
AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

各级别的 *页表条目* 格式如上图。

5.2.4.1 Page Frame Address

由于页大小为4KB，所以页表条目格式低12位为0。当该条目位于页目录中，`Page Frame Address` 表示 *页表* 的地址；当该条目位于二级页表中，`Page Frame Address` 表示 *页* 地址。

页帧地址 代表页的物理起始地址

5.2.4.2 Present Bit

表示 *页表条目* 是否可用于地址转换。1表示可以使用。

`P=0` 时，*页表条目* 格式如上图。

5.2.4.3 Accessed and Dirty Bits

5.2.4.4 Read/Write/ and User/Supervisor Bits

不用于地址翻译，而是用于 *页级* 保护。处理器在进行 *地址翻译* 的同一时间执行 *页级* 保护。

5.2.5 Page Translation Cache

5.3 Combining Segment and Page Translation

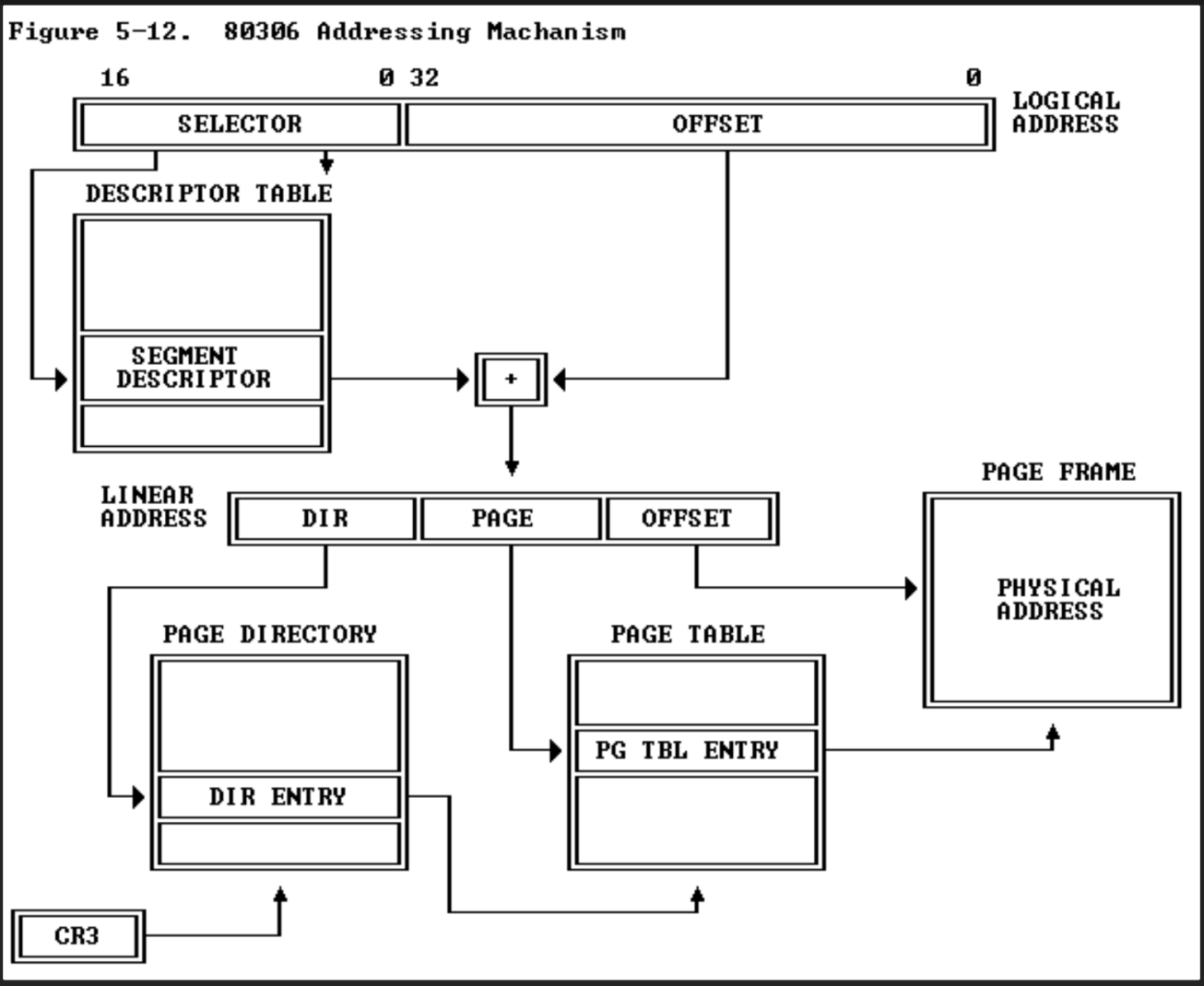
5.3.1 “Flat” Architecture

当 `80386` 用来执行没有段结构的软件，关闭 `80386` 的段特性可能会比较高效。

尽管 `80386` 没有禁用段功能的模式，但可以通过将包含整个32位线性地址空间的 *描述符的选择器* 加载到 *段寄存器* 来达到同样的效果。

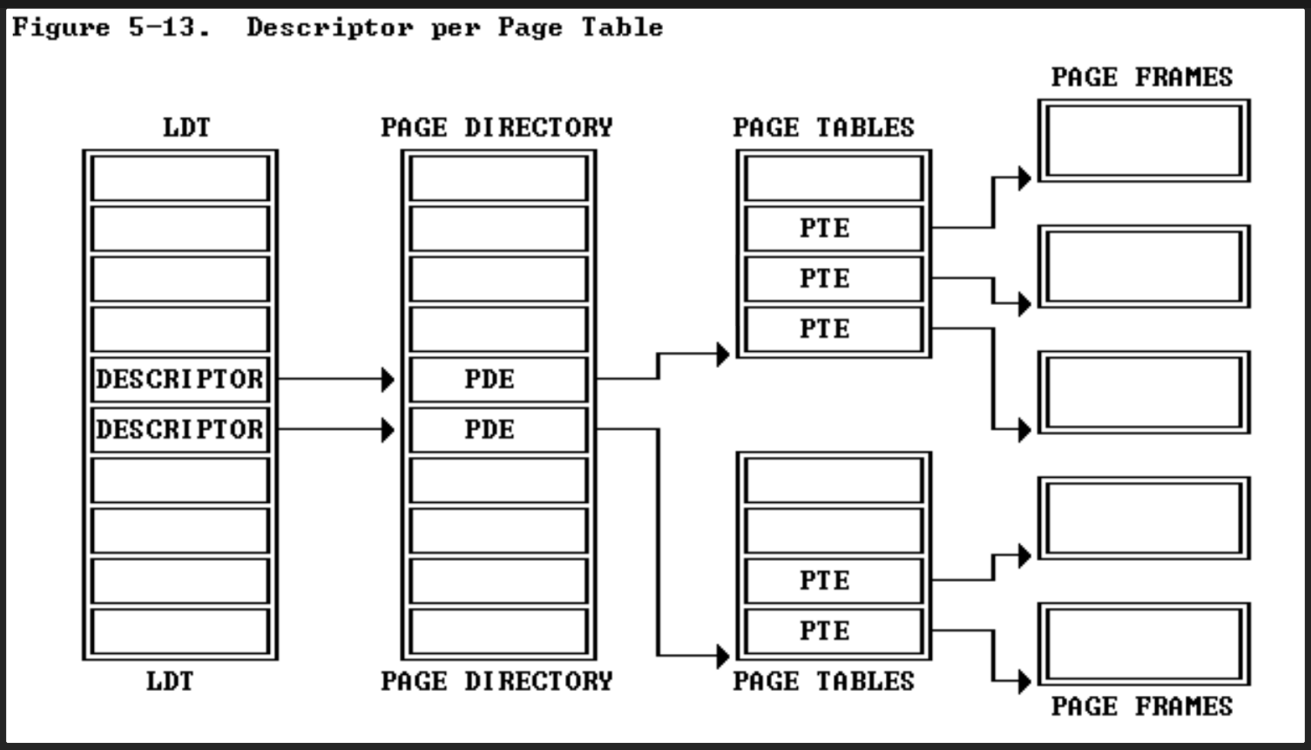
5.3.2 Segments Spanning Several Pages

Figure 5-12. 80306 Addressing Machanism



5.3.6 Page-Table per Segment

进一步简化空间管理软件的空间管理方法就是：维持一个段描述符和页目录条目 *一对一* 对应关系。



一对一关系如上图所示。

6.1 Why Protection?

`80386`：帮助检测和识别`bug`。

6.2 Overview of 80386 Protection Mechanisms

`80386`中的保护有五个方面：

- Type checking
- Limit checking
- Restriction of addressable domain（可寻址域的限制）
- Restriction of procedure entry points（程序入口点的限制）
- Restriction of instruction set（指令集的限制）

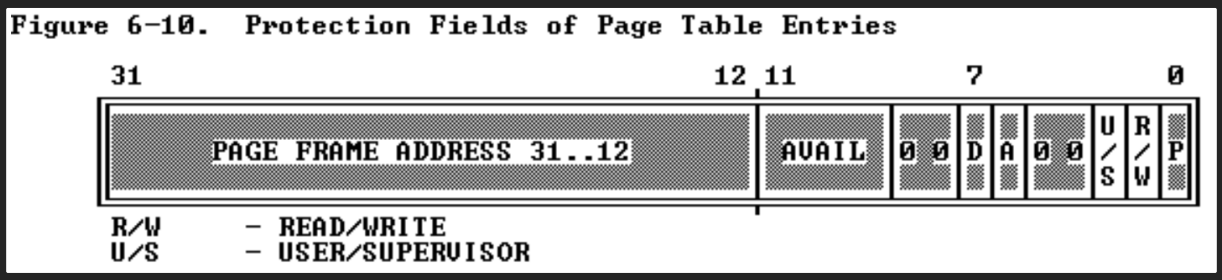
d

6.4 Page-Level Protection

和 `page` 相关的保护有两个：

- Restriction of addressable domain
- Type checking

6.4.1 Page-Table Entries Hold Protection Parameters



6.4.1.1 Restricting Addressable Domain

通过给每个 *页* 分配两个级别中的一个来实现 *页* 的 *特权* 概念：

- Supervisor level (U/S=0) 用于操作系统和其他系统软件和相关数据
- User level (U/S=1) 用于应用进程和数据

当前 *level* 和 `CPL` 的值有关。0，1，2为 `Supervisor` ；3为 `User` 。

6.4.1.2 Type Checking

页寻址 有两种类型级别：

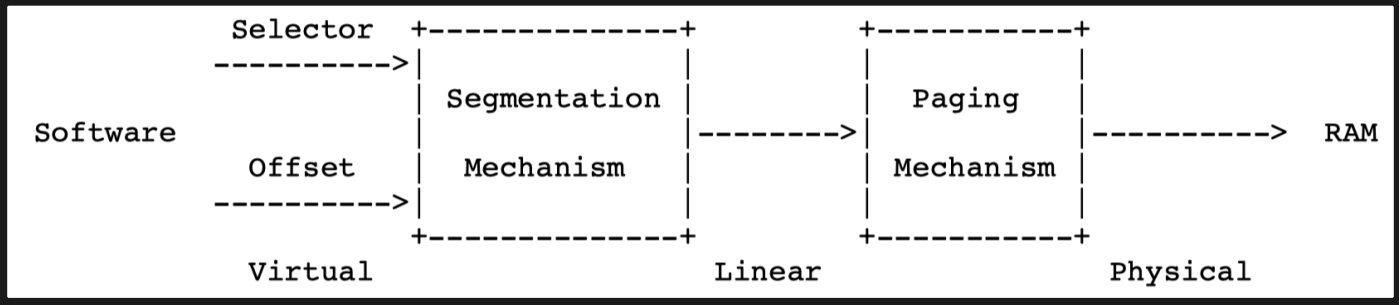
- Read-only access (R/W=0)
- Read/write access (R/W=1)

Virtual, Linear, and Physical Addresses

`virtual address` *段选择器* 和 *段内偏移* 组成。

`linear address` 在 *段翻译* 之后， *页翻译* 之前得到。

`physical address` 是最后 *段翻译* 和 *页翻译* 之后得到，并且最终通过硬件总线传输到 `RAM` 。



在 `boot/boot.S` 中，我们安装了一个 `Global Descriptor Table(GDT)` ，通过将所有 *段基地址* 设为0， *limit* 设为0xffffffff来禁用 *段翻译* 。

这样一来，“*选择器*”就没有影响了。

实验3中，我们将会和 *段* 有更多的交互来设置 *特权级别* 。

在这个实验中，你将会设置 `JOS` ，我们将扩展这个来映射开始于 *虚拟地址* `0xf0000000` 的前256MB的 *物理地址* 和 *虚拟地址空间* 的一些其他领域。

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.

```
(gdb) si
=> 0xf010
400
(gdb) si
=> 0xf010
kbd_proc_data () at kern/console.c:317
317     {
(gdb) x/16wx 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x5000b812      0x220f0011      0xc0200fd8
0x100020:       0x0100010d      0xc0220f80      0x10002fb8      0xbde0fff0
0x100030:       0x00000000      0x113000bc      0x0068e8f0      0xfeeb0000
(gdb)
```

```
(qemu) xp/16x 0x00100000
0000000000100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0000000000100010: 0x34000004 0x5000b812 0x220f0011 0xc0200fd8
0000000000100020: 0x0100010d 0xc0220f80 0x10002fb8 0xbde0fff0
0000000000100030: 0x00000000 0x113000bc 0x0068e8f0 0xfeeb0000
(qemu)
```

```
VPN range     Entry         Flags        Physical page
[00000-003ff]  PDE[000]     ----A----P
[00000-00000]  PTE[000]     --------WP 00000
[00001-0009f]  PTE[001-09f] ---DA---WP 00001-0009f
[000a0-000b7]  PTE[0a0-0b7] --------WP 000a0-000b7
[000b8-000b8]  PTE[0b8]     ---DA---WP 000b8
[000b9-000ff]  PTE[0b9-0ff] --------WP 000b9-000ff
[00100-00103]  PTE[100-103] ----A---WP 00100-00103
[00104-00111]  PTE[104-111] --------WP 00104-00111
[00112-00112]  PTE[112]     ---DA---WP 00112
[00113-00115]  PTE[113-115] --------WP 00113-00115
[00116-003ff]  PTE[116-3ff] ---DA---WP 00116-003ff
[f0000-f03ff]  PDE[3c0]     ----A---WP
[f0000-f0000]  PTE[000]     --------WP 00000
[f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
[f00a0-f00b7]  PTE[0a0-0b7] --------WP 000a0-000b7
[f00b8-f00b8]  PTE[0b8]     ---DA---WP 000b8
[f00b9-f00ff]  PTE[0b9-0ff] --------WP 000b9-000ff
[f0100-f0103]  PTE[100-103] ----A---WP 00100-00103
[f0104-f0111]  PTE[104-111] --------WP 00104-00111
[f0112-f0112]  PTE[112]     ---DA---WP 00112
[f0113-f0115]  PTE[113-115] --------WP 00113-00115
[f0116-f03ff]  PTE[116-3ff] ---DA---WP 00116-003ff

(qemu) info mem
0000000000000000-0000000000400000 0000000000400000 -r-
00000000f0000000-00000000f0400000 0000000000400000 -rw
```

从在CPU执行的代码可以看到，一旦我们进入 *保护模式* （在 `boot/boot.S` 干的第一件事），就不可能直接使用 *线性地址* 或者 *物理地址* 。

所有的内存引用都被解释为虚拟地址并被 `MMU` 翻译，也意味着所有的*C指针* 都是虚拟地址。

`JOS` 经常把地址当作一个 *opaque value* 或者一个 *整数*，而不去解引用。

`JOS` 可以解引用一个 `uintptr_t`，通过先把它转换成一个指针类型。*但是*，`kernel` 不能 *解引用* 一个 *物理地址*，因为 `MMU` 翻译所有的内存引用。

`JOS kernel` 有时需要读取或修改它仅知道 *物理地址* 的内存。比如，向 *页表* 中加入一个映射可能需要分配物理内存来存储页目录然后初始化那个内存。**然而**，`kernel` 不能绕过 *虚拟地址* 翻译，也因此不能直接加载和存储物理地址。

`JOS` 重新映射从 *物理地址* 0开始的所有 *物理地址* 到 *虚拟地址* `0xf0000000` 的一个原因是帮助 `kernel` 读取和写入它只知道 *物理地址* 的内存。

为了将一个 *物理地址* 翻译成一个 *虚拟地址* 来让 `kernel` 能真正读取和写入，`kernel` 必须将 `0xf0000000` 和 *物理地址* 相加来找到它在重映射区域对应的虚拟地址。可以用**函数** `KADDR(pa)` 来完成。

有时候 `JOS kernel` 也需要能找到给定 *虚拟地址* 的 *物理地址*，这是 `kernel` 数据结构存储的地方。`kernel` 的全局变量和通过 `boot_alloc()` 分配的内存都在那个区域，那个区域是 `kernel` 被加载的地方，开始于 `0xf0000000`，也是我们映射所有 *物理内存* 的区域。因此，将一个此区域的 *虚拟地址* 转换成一个 *物理地址*，`kernel` 可以简单地减去 `0xf0000000`。可以用**函数** `PADDR(a)` 来完成。

Reference counting

之后的实验中，会经常遇到相同的 *物理地址* 被同时映射到多个 *虚拟地址*（或者是多个环境的地址空间）。

你将会维持一个每个物理页 `references` 的计数，这个计数用 *物理页* 所对应的 `struct PageInfo` 中的 `pp_ref` 来记录。当该计数为0时，说明这个 *页* 可以被释放，因为它不再被使用。

换句话来说，这个计数应该和 *物理页* 出现在所有 *页表* 中 `UTOP` 下方的次数相同。（在 `UTOP` 上方的映射在启动时被 `kernel` 所设置，不应该被释放，所以没有必要去考虑它们。）

我们也用它来跟踪 *指针数目*，这个 *指针* 指向 *页目录页*；还跟踪 *引用数目*，这个 *引用* 指的是 *页表页*。

小心使用函数 *`page_alloc()`*。这个函数返回的 *页* 引用计数为0，所以你一旦用返回的页做点什么，就要**增加** `pp_ref`，比如将其插入一个 *页表*。当然，有时这事已经被别的函数（比如 `page_insert`）处理，有时调用 `page_alloc()` 的函数必须直接处理。

Page Table Management

现在你需要写一些例程来管理 *页表*：

- 插入和移除 *线性地址-物理地址* 映射
- 需要时创建 *页表页*

> **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.
>
> `pgdir_walk()`
>
> ```
>   // 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 PageInfo * 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
  // directory more permissive than strictly necessary.
  //
  // Hint 3: look at inc/mmu.h for useful macros that manipulate
page
  // table and page directory entries.
  //
pte_t *
  pgdir_walk(pde_t *pgdir, const void *va, int create)
  {
      // Fill this function in
      // judge if the present-bit of PTE is 1(for convertable)
      /*
      pte_t pte_content = pgdir[PDX(va)][PTX(va)];
      if(PTE_P & pte_content){
          return pte_content;
      }else if(!(PTE_P & pte_content)){              // page
table page is not exist and create == false
          if(false == create){
              return NULL;
          }
      }
      */
      // maybe we just need to judge the address of page table
page rather than if pte is convertable
      // the consumption is wrong, we need to judge if the pde is
PTE_P
  uintptr_t pdx = pgdir[PDX(va)];
  pte_t* result;
      if(!(PTE_P & pdx)){
          if(false == create){
```

```
                return NULL;
            }else{
  // allocate a page with 'page_alloc'
  struct PageInfo* new_ptp = page_alloc(ALLOC_ZERO);
                cprintf("the new page is %x\n", new_ptp);
                if(NULL == new_ptp){
                    return NULL;
                }
                new_ptp->pp_ref += 1;
                cprintf("new_ptp->pp_ref is %x\n", new_ptp->pp_ref);
                // put the address of new page into pgdir[]
               // 注意：这里pgdir[PDX(va)]存放的还是pa，物理地址
                pgdir[PDX(va)] = page2pa(new_ptp) | PTE_P | PTE_U |
PTE_W;
            }
        }
        // what we return is a point that is virtual address
        // and we use 'PTE_ADDR' to get pte address
        result = (pte_t*)KADDR(PTE_ADDR(pgdir[PDX(va)])) + PTX(va);
        return result;
  }
```

boot_map_region

```
  // Map [va, va+size) of virtual address space to physical [pa,
pa+size)
  // va and pa are both page-aligned.
  // 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)
```

```
  {
      // Fill this function in
  size_t pgs = size / PGSIZE;
      if(0 != size % PGSIZE){
          ++pgs;
      }
      for(unsigned i = 0;i < pgs;++i){
          // get the pa of the page table page for va
  pte_t* pte = pgdir_walk(pgdir, (void *)va, 1);
          if(NULL == pte){
              panic("The fuction pgdir_walk() return NULL.");
          }
          // when dereference an address, the address must be
virtual address.
          *pte = pa | perm | PTE_P;
          va += PGSIZE;
          pa += PGSIZE;
      }
  }
```

page_lookup()

```
// 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)
  {
      // Fill this function in
      // there is no page mapped at va, return NULL
      cprintf("pgdir[0] is %x\n", pgdir[0]);
  pte_t* pa = pgdir_walk(pgdir, va, false);
      // cprintf("*pa is %x\n", *pa);
```

```
      // cprintf("here4, pa is %x, *pa is %x, pte_store is %x,
*pte_store is %x\n", pa, pte_store, *pte_store);
      // cprintf("here5, pa2page(*pa) is %x\n", pa2page(*pa));
      if(NULL == pa || 0 == *pa){
          cprintf("here here\n");
          return NULL;
      }
      cprintf("(pte_t)pa is %x\n", (pte_t)pa);
      cprintf("here6, pa2page(pa) is %x\n", pa2page(PADDR(pa)));
      if(0 != pte_store){
          *pte_store = pa;
      }
      cprintf("here5, pa2page(*pa) is %x\n", pa2page(*pa));
      // !!REMIND: the parameter here is not '*pa', but
PADDR(pa).Since 'pgdir_walk()' returns a virtual address.
      // !!REMIND: pa is just va, and this function just returns
'PageInfo*'
      // Firstly, deference va 'pa'. Then convert it to
'PageInfo*'
      return pa2page(*pa);
  }
```

`page_remove()`

```
// 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)
  {
      // Fill this function in
  struct PageInfo* pp = page_lookup(pgdir, va, 0);
      cprintf("here3, pp is %x\n", pp);
      if(NULL == pp){
          return;
      }
      // !!REMIND: we also need to remove the corresponding pa by
setting it 0
  pte_t *pte = pgdir_walk(pgdir, va, 0);
      *pte = 0;
      page_decref(pp);
      tlb_invalidate(pgdir, va);
      // pgdir[PDX(va)][PTX(va)] = 0;

  }
```

page_insert()

```
// 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)
  {
      // Fill this function in
  // get pa of pp with 'page2pa'
  physaddr_t pa = page2pa(pp) | perm | PTE_P;
  // get address of pte for 'va'
  pte_t* pte = pgdir_walk(pgdir, va, true);
  cprintf("page_free_list is %x, pte is %x\n", page_free_list,
pte);
      if(NULL == pte){
          cprintf("here\n");
          return -E_NO_MEM;
      }
      // judge if the pa of *pte is same as 'pa'
      cprintf("pa is %x, *pte is %x\n", pa, *pte);
      if(PTE_ADDR(pa) == PTE_ADDR(*pte)){
          cprintf("here\n");
          // since the permission may be changed, we need to
assign '*pte' the new pa
          *pte = pa;
          // maybe the pp_ref needn't increment
```

```
            // ++pp->pp_ref;
            return 0;
        }else{
            page_remove(pgdir, va);
            *pte = pa;
            ++pp->pp_ref;
            return 0;
         }
    }
```

Part 3: Kernel Address Space

`JOS` 将处理器的32位线性地址空间分为两部分。

其中一个部分就是 *用户环境（进程）*，也是 `lab3` 中开始加载和运行的地方。这部分拥有对较低内存部分 *布局* 和 *内容* 的控制，而较高内存部分的控制权始终由 `kernel` 掌控。

两个部分的分割线定义于 `inc/memlayout.h` 中的 `ULIM` 标志，为 `kernel` 保留了大约 `256MB` 的虚拟地址空间。这个 *解释* 了为什么在 `lab1` 需要给 `kernel` 那么高的链接地址：

> 否则，`kernel` 虚拟地址空间就没有足够的空间来 *同时* 映射在它之下的用户环境。

Permissions and Fault Isolation

因为 `kernel` 和 `user memory` 在每个环境地址空间都存在，我们不得不使用 `x86` 页表中的 *权限位* 来允许 *用户代码* 仅访问地址空间中的用户部分。

> 否则，用户代码中的漏洞可能会重写内核数据，导致**冲突**或更多微小的 *故障*；用户代码也可能会偷取其他环境中的私密数据。
>
> 注意：可写权限 `PTE_W` 对用户和内核代码都有效。

用户环境对 `ULIM` 之上的任何内存都没有权限，而 `kernel` 可以读写这个内存。对于地址范围为 `[UTOP,ULIM)` 的地址，`kernel` 和用户环境有相同的权限：都**可以读**但**不能写**。这个范围的地址被用来展示核心 *内核数据结构*，对用户环境只读。

最后，低于 `UTOP` 的地址空间供用户环境使用；用户环境将设置 *权限* 来访问这个内存。

Initializing the Kernel Address Space

现在你将设置高于 `UTOP` 的地址空间：地址空间的*内核部分*。`inc/memlayout.h` 给出了你应该使用的布局。你将会使用刚刚用于设置合适的线性地址和物理地址映射的函数。

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.

First part

```
////////////////////////////////////////////////////////////////////
////
// Now we set up virtual memory

////////////////////////////////////////////////////////////////////
////
// Map 'pages' read-only by the user at linear address UPAGES
// 把 'pages' 的物理地址映射到虚拟地址UPAGES
// 也就是将所有页表项映射到虚拟地址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:
// struct PageInfo* pp = page_alloc(ALLOC_ZERO);
assert(NULL ≠ pages);
// page_insert(kern_pgdir, pages, (void*)UPAGES, PTE_W | PTE_P);
// change the permission of the new image at UPAGES
// REMIND: pages is also VIRTUAL ADDRESS, need to use 'PADDR()'
// WARNING: the permission here may be wrong
    // ⚠️这里的permission确实写错了 应该是 `PTE_P | PTE_U`
boot_map_region(kern_pgdir, UPAGES, ROUNDUP(npages * sizeof(struct
PageInfo), PGSIZE), PADDR(pages), PTE_P | PTE_W);
// WARNING: the permission here may be wrong
```

```
kern_pgdir[PDX(UPAGES)] = PTE_ADDR(kern_pgdir[PDX(UPAGES)]) |
PTE_P | PTE_U;
```

Second part

```
////////////////////////////////////////////////////////////////////
////
// Use the physical memory that 'bootstack' refers to as the
kernel
// stack.  The kernel stack grows down from virtual address
KSTACKTOP.              // which means map virtual address
KSTACKTOP to 'bootstack'
// 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
  uintptr_t va1 = KSTACKTOP-KSTKSIZE;
size_t size1 = KSTKSIZE;
// cprintf("bootstack is: %x\n", bootstack);
// cprintf("(physaddr_t)bootstack is: %x\n",
(physaddr_t)bootstack);
// cprintf("PADDR(bootstack) is: %x\n", PADDR(bootstack));
// cprintf("PTSIZE is: %x\n", PTSIZE);
// cprintf("bootstack - PTSIZE is: %x\n", bootstack - PTSIZE);
// cprintf("check_va2pa(kern_pgdir, KSTACKTOP - KSTKSIZE) is:
%x\n", check_va2pa(kern_pgdir, KSTACKTOP - KSTKSIZE));
// according to test function, here is just 'bootstack' without
subtract pa1 = bootstack
// physaddr_t pa1 = (unsigned)bootstack - PTSIZE;
boot_map_region(kern_pgdir, va1, size1, PADDR(bootstack), PTE_P |
PTE_W);
```

Third part

```
//////////////////////////////////////////////////////////////////////
////
// 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:
uintptr_t va3 = KERNBASE;
unsigned long long x = 1;
size_t size3 = (x << 32) - KERNBASE;
physaddr_t pa3 = 0;
boot_map_region(kern_pgdir, va3, size3, pa3, PTE_P | PTE_W);
```

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		Page table for top 4MB of phy memory

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?

   Answer: `Permission bits` .

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

物理内存的最大容量是