MIT6.828 Lab2: Memory Management

Zhuofan Zhang

Jan 2022

Contents

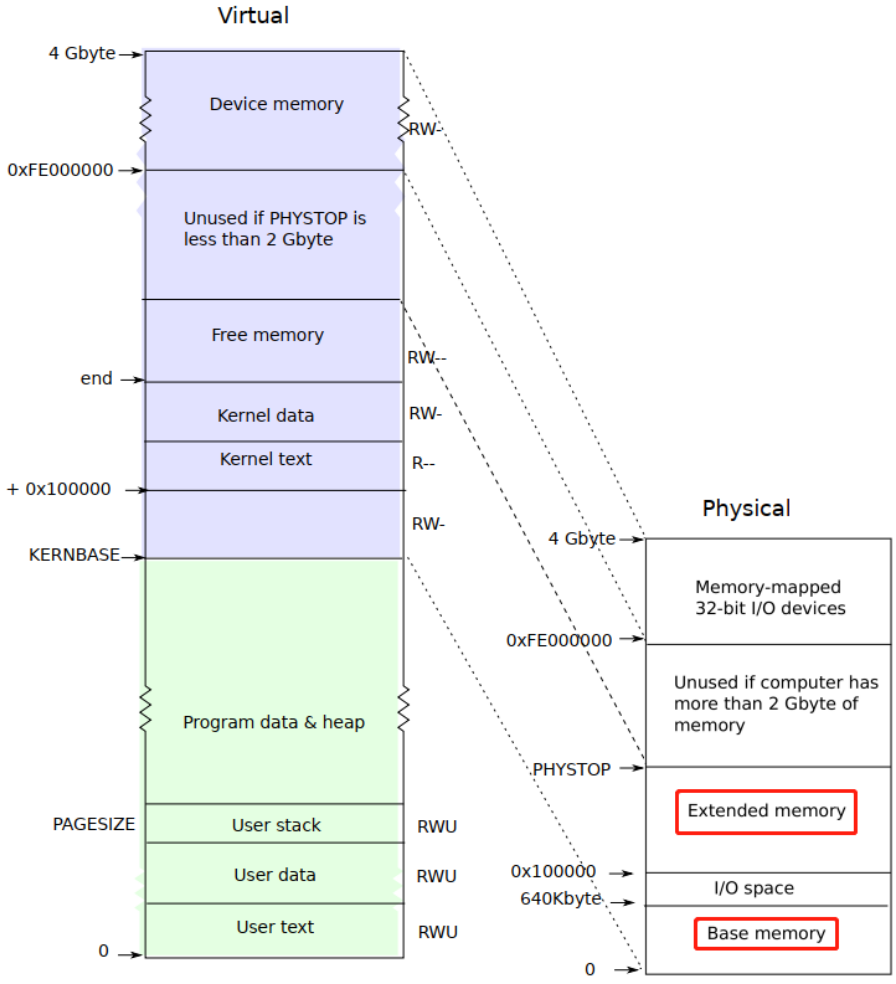
Chapter 1

Physical Page Management

本次 Lab 的内容主要分为两个部分，第一个部分是对可用物理内存的管理，第二个部分是虚拟内存的管理。

操作系统负责对可用物理内存的管理。JOS 的物理页使用 PageInfo 结构相关数据（数组/空闲链表）进行物理内存管理。PageInfo 数组（pages）是对整个物理地址空间中可用内存（base-mem + extend-mem）的映射：将整个物理地址空间按 4KB-page 划分，则每个单页可由 pages 对应下标的 PageInfo 结构唯一指代（原理可见 pmap.h/page2pa 的实现）。

下图为 xv6 的 VA-PA 映射关系图，在本次 Lab 中我们将在 JOS 上实现类似的映射。



Exercise 1

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

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.

根据题目要求，我们依次实现上述函数。

对于 **boot_alloc** 函数，根据注释提示可知其为系统启动初期的 physical-page allocator，协助完成初始化工作。函数中使用了一个变量 end，我们从内核的链接脚本（kernel.ld）中可以定位到它的位置，即.BSS 段的末尾；根据注释提示，end 表示第一处未被 kernel 使用的虚拟地址。由 Lab1 可知内核被链接至高地址位（0xf0100000 开始的 4MB 区域），因此也许保证在这个阶段分配的物理内存不超过 4MB，需要对分配是否越界进行检查。明确上述要求后，我们即可实现该函数：

```
static void *
boot_alloc(uint32_t n)
{
    ...

    // 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(n > 0)
        nextfree = ROUNDUP(nextfree + n, PGSIZE);
    else if(n < 0)
        // n < 0: should not come here
        panic(”boot_alloc: parameter n < 0.”);

    if(PADDR(nextfree) >= 0x400000)
        panic(”boot_alloc: run out of mem.”);
    return result;
}
```

mem_init 函数是被内核 i386_init() 调用的函数，用来完成内核内存管理的初始化；本节仅需完成物理内存管理相关的部分。

根据本节开头我们提到的物理内存管理方法，我们首先需要创建管理物理页的数组 pages。由于 pages 本身也需要占用内存，我们利用上一步实现的 boot_alloc 分配内存并将 pages 进行 0 初始化。需要注意的是数组大小（npages）由 i386_detect_memory 调用得到，它负责检查系统可用的物理内存大小。根据上述分析，我们实现 mem_init 中初始化 pages 部分的内容：

```
void
mem_init(void)
{
    ...

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

    ...

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

    ...
}
```

上一步实现的代码中，我们可以看到在为 pages 分配内存并进行 0 初始化后，mem_init 调用了真正对 pages 及空闲列表 page_free_list 进行初始化的 **page_init** 。根据 Lab1 中提到的 PC's Physical Address Space 的布局我们可知，物理空间中可用内存由 Base-memory 和 Extended-memory 两部分组成，我们需要将这些可用的内存放入空闲链表；由于实际上系统运行至这一步时已经使用了一部分物理页，这部分页不应再放入空闲列表；此外，物理空间中也存在一些不可使用的段（IO holes/BIOS/...），我们也许将其排除。

根据 page_init 中的提示，我们完成代码如下。其中比较需要关注的是 extended memory 空闲部分的初始化，第三步除了设置 IO hole 不空闲外，也需设置已经被使用的物理页，而确定当前已用物理页位置末端（以虚拟地址表示）的方法是对 boot_alloc 的零调用：

```
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!
    // 1) Mark physical page 0 as in use
    pages[0].pp_ref = 1;
    // 2) Base-memory
    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];
    }
    // 3) IO hole + pages that have been used
    int hole_and_used = npages_basemem +
                        ((EXTPHYSMEM - IOPHYSMEM) +
                        PADDR(boot_alloc(0)))/PGSIZE;
    for(; i < hole_and_used; i++)
        pages[i].pp_ref = 1;

    // 4) extended memory
    for(; i < npages; i++)
    {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

初始化 pages 及 page_free_list 后，就可以进一步实现分配和回收页的函数：**page_alloc** 与 **page_free** 。注意到实现物理页分配时，对页面进行初始化时 需要获得物理页的虚拟地址（C 语言中地址均匀虚拟地址），根据提示使用 page2kva 实现。page2kva 本身的实现是使用 page2pa 得到物理页的实际地址，再根据当前的映射使用 KADDR 转化为虚拟地址。

```
struct PageInfo *
page_alloc(int alloc_flags)
{
    // Fill this function in
    if(!page_free_list)
        return NULL;

    struct PageInfo *alloc_page = page_free_list;
    page_free_list = page_free_list->pp_link;
    alloc_page->pp_link = NULL;

    if(alloc_flags & ALLOC_ZERO)
        memset(page2kva(alloc_page), 0, PGSIZE);

    return alloc_page;
}

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: pp_ref != 0");

   pp->pp_link = page_free_list;
   page_free_list = pp;

}
```

Chapter 2

Virtual Memory

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.

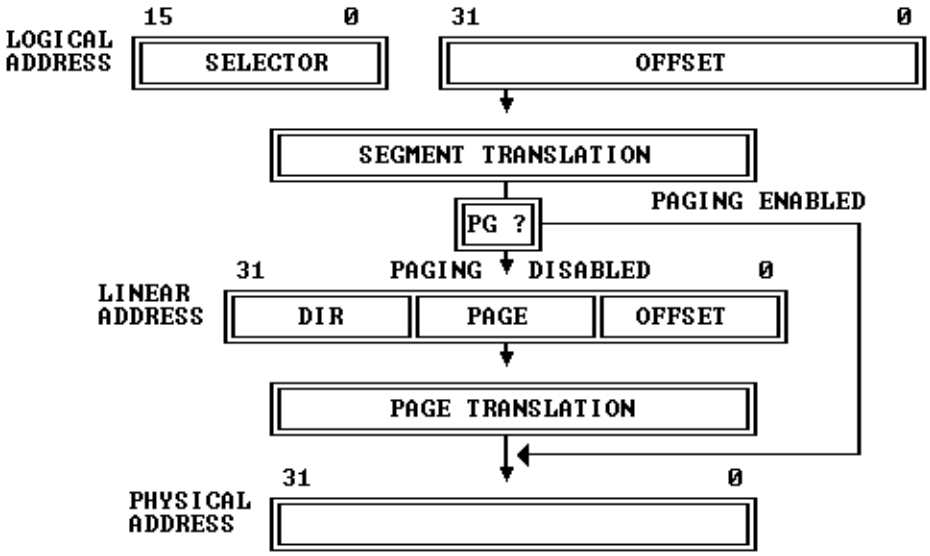
这一节内容需要我们构建虚拟内存管理的基本设施，Exercise2 为我们提供了 80386 处理器内存管理模式的相关材料。阅读相关材料后，我们可以整理出处理器为我们提供的内存管理机制：分段/分页。

2.1 Virtual, Linear, and Physical Addresses

在 x86 体系中，我们可以定义 3 种地址：虚拟地址（Virtual Address, VA）[也称 Logical Address]，线性地址（Linear Address, LA）与物理地址（Physical Address, PA）。

对于用户而言，可见的只有 VA，而数据在物理内存中的实际地址是 PA。x86 的典型地址翻译流程如下图所示，其中 VA $\rightarrow$ LA 的过程为 分段（*Segmentation*）机制，LA $\rightarrow$ PA 过程为 分页（*Paging*）机制。

关于分页与分段的详细内容不在此处展开，但需知道：JOS 并不使用分段机制，但 x86 系统并没有显式关闭分段的方法。因此，在设置 Segment Descript Table 时，JOS 将段基址设置为 0，并将 LIMIT 设置为 0xffffffff，即将整个物理地址空间视为一段，在这种情况下 VA = LA。设置的过程在 Lab1 中的 boot 阶段完成（boot.S）。



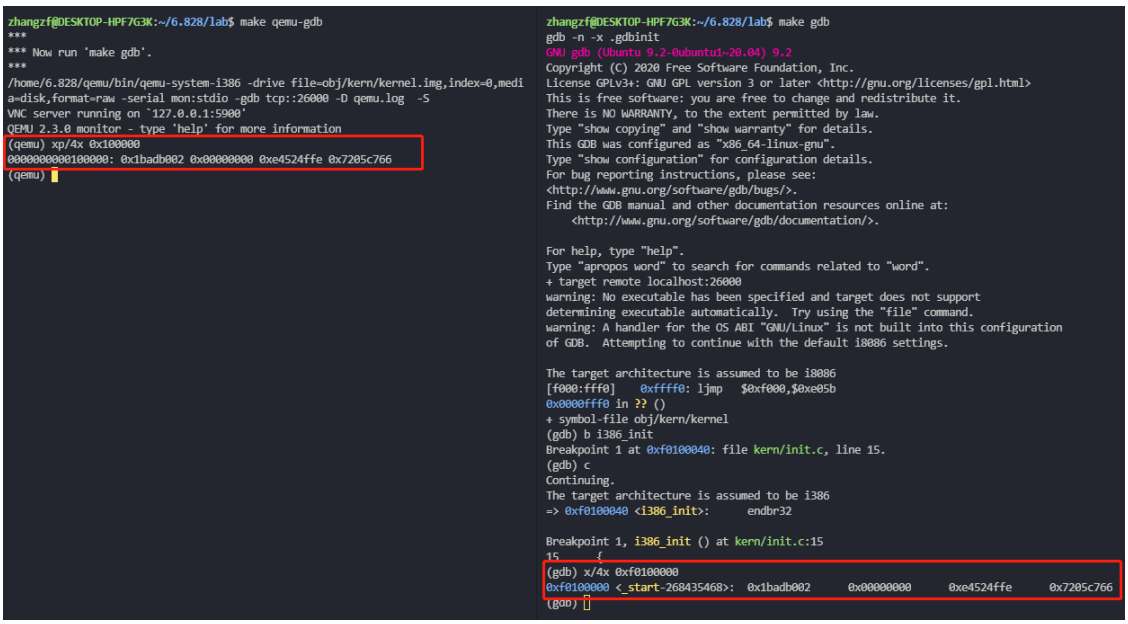
Exercise 3

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.

这一个练习的第一个要求与 Lab1 类似，要求我们观察虚拟地址与其实际映射到的物理地址的内容是否一致（即虚拟内存映射是否成功）。与 Lab1 不同，此处采用更泛用的方法：使用 GDB 观察 VA，使用 QEMU-monitor 观察 PA。根据提示，我们在 gdb 设置内存映射成功后的位置检查，可以看到 VA 及其对应的 PA 内容的一致性：



此外，我们还根据提示，使用 **info pg/mem** 命令查看页表情况与内存权限情况：

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[00000-003ff]  PDE[000]     ----A----P
    [00000-000ff]  PTE[000-0ff] --------WP 00000-000ff
    [00100-00100]  PTE[100]     ----A---WP 00100
    [00101-00111]  PTE[101-111] --------WP 00101-00111
    [00112-00112]  PTE[112]     ---DA---WP 00112
    [00113-003ff]  PTE[113-3ff] --------WP 00113-003ff
[f0000-f03ff]  PDE[3c0]     ----A---WP
    [f0000-f00ff]  PTE[000-0ff] --------WP 00000-000ff
    [f0100-f0100]  PTE[100]     ----A---WP 00100
    [f0101-f0111]  PTE[101-111] --------WP 00101-00111
    [f0112-f0112]  PTE[112]     ---DA---WP 00112
    [f0113-f03ff]  PTE[113-3ff] --------WP 00113-003ff
(qemu) info mem
0000000000000000-0000000000400000 0000000000400000 -r-
00000000f0000000-00000000f0400000 0000000000400000 -rw
```

本节的最后，Lab 还简要介绍了 JOS 中地址类型的 typedef。JOS 将地址类型区分为虚拟地址（uintptr_t）与物理地址（physaddr_t），它们均为 uint32_t 的 typedef。由于 C 中所有出现的“地址”均为虚拟地址，故其中 uintptr_t 可以被解引用使用，而 physaddr_t 不应被解引用，它仅作为物理地址字面值被使用，对其解引用将引发 UB。

2.2 Page Table Management

本节的内容是构建页表管理的 API，为下一节构建内核 VA 映射提供支持。

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

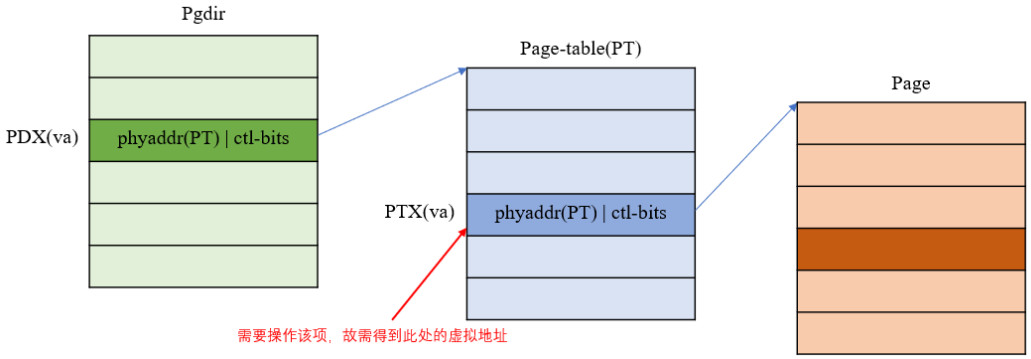
pgdir_walk 函数的功能是在页表目录（pgdir）查询目标虚拟地址 va，返回指向查询的虚拟地址对应页表项（PTE）的指针；若页表项不存在（即 PDE 中 PTE_P 非置位，对应页表不存在），可以设置 create 选项决定是否创建页表。理解这个函数的实现有两个比较重要的点：

1. 返回的指向对应 PTE 的指针是什么。我们需要取的是 va 对应页表的虚拟地址。根据 va 对应的 PDE，我们可以得到页表的物理地址（PTE_ADDR(pde)）；而根据 Lab1 及前文所述，JOS/xv6 都将物理地址空间映射到了高位虚拟地址空间（即 KERNBASE 上方，见前文 xv6: VA-PA 映射图），因此我们可以通过直接将物理地址加上 KERNBASE（JOS 提供了宏：KADDR）得到 pte 的虚拟地址。

（注：可能存在疑问——当前程序状态仅在虚拟内存空间中映射 [0, 4MB) 的物理地址，但实际此处分配得到的页表可能在更高的地址中，是否存在问题？答案是否定的，因为此时我们所有的操作仍未加载到处理器上，实际该页表目录是等到完整映射构建后才被加载的，当时物理地址空间已被完整映射到 JOS-VA 的 remappad 区域。见 Kernel Address Space 一节）

2. 若需 create，实际创建了什么。根据 Lab-Hints，pgdir_walk 发生分配时实际上是分配 va 所在的页表，只需 PTE 被建立即可，而 va 实际对应的物理空间仍未被分配。

根据上述分析，我们可以实现函数如下：




```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // ppde/ppte: point-to-pde/pte
    pde_t *ppde = pgdir + PDX(va); // pde is the VA of pgdir[PDX(va)]
    pte_t *ppte = NULL;         // default result: NULL
    if(!(*ppde & PTE_P))
    {
        if(create)
        {
            struct PageInfo *new_pt = page_alloc(ALLOC_ZERO);
            if(!new_pt)
                // alloc failed
                return NULL;
            (new_pt->pp_ref)++;
            *ppde = (page2pa(new_pt) | PTE_P | PTE_W | PTE_U);
        }
        else
            return NULL;
    }

    // Don’t forget the (pte_t*) cast
    ppte = (pte_t*)KADDR(PTE_ADDR(*ppde)) + PTX(va);  // key: the use of KADDR
    return ppte;
}
```

boot_map_region

这个函数是用于构建 PA-VA 的核心函数，用于将 [pa, pa+size) 映射到 [va, va+size)。Lab-Hints 提示使用 pgdir_walk 实现，实现结果如下：

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    if((size % PGSIZE))
    {
        cprintf("HERE:%#x, SIZE:%#x\n", va, size);
        panic("boot_map_region: size is not a multiple of PGSIZE.");
    }

    for(int sz = 0; sz < size; sz += PGSIZE)
    {
        pte_t *pte = pgdir_walk(pgdir, (void *)(va + sz), 1);
        *pte = (pa + sz) | perm | PTE_P;
    }

}
```

page_lookup

这个函数用于查找 va 对应的实际物理页（注意与 pgdir_walk 功能区分），返回物理页对应的 PageInfo 对象，并可选地记录 PTE 的虚拟地址。若 va 对应的物理页尚未分配则返回空指针。函数实现如下：

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // ppte: point-to-pte
    pte_t *ppte = pgdir_walk(pgdir, va, 0);
    if(!ppte)
        return NULL;

    if(!(*ppte & PTE_P))
        return NULL;

    if(pte_store != NULL)
        *pte_store = ppte;

    return pa2page(PTE_ADDR(*ppte));
}
```

page_remove & page_insert

这两个函数用于映射的删除与分配。其中有以下几点需要注意：

1. page_remove 在删除映射时需同时刷新 TLB，并在 pp_ref 引用为 0（即页已结束生命期）时将其重新维护入空闲列表。

2. page_insert 的 corner_case。由于实际插入页可能涉及自身替换问题，需先增加 pp_ref 再进行旧页 remove 操作，防止错误释放目标页。

根据分析后实现代码如下：

```
void
page_remove(pde_t *pgdir, void *va)
{
    pte_t *ppte;
    struct PageInfo* unmap_pg = page_lookup(pgdir, va, &ppte);

    // such physic-page exist
    if(unmap_pg)
    {
        *ppte = 0;
        page_decref(unmap_pg);
        tlb_invalidate(pgdir, va);
    }
}

int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
   pte_t *ppte = pgdir_walk(pgdir, va, 1);
   if(!ppte)
      return -E_NO_MEM;
   // Consider the corner-case
   pp->pp_ref++;
   page_remove(pgdir, va); // If page-map exists, remove it
   *ppte = page2pa(pp) | perm | PTE_P;
   return 0;
}
```

Chapter 3

Kernel Address Space

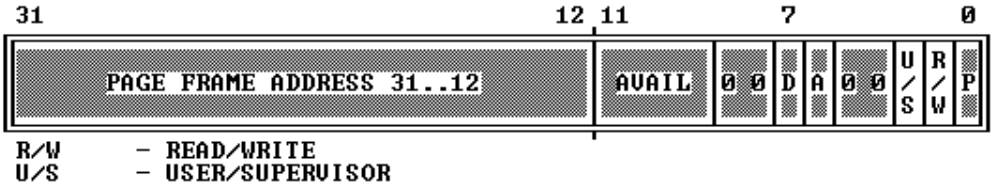
利用上一章节构建的 API，我们就可以实现 JOS 的内核部分地址空间的映射了。

JOS 的 VAS 布局可以参考 memlayout.h 文件。JOS 将整个地址空间划分为两部分，以 ULIM 为界下方为用户地址空间，上方为内核地址空间，且 [UTOP, ULIM) 段一经初始化即为只读（内核/用户态均仅有此权限），是内核为用户态提供的信息。

> **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.

查看 mem_init 的 Lab-hints，可以发现我们需要映射内核地址空间的三个部分：RO PAGES，KERNEL STACK，remapped physical address。根据提示我们即可补全上述代码完成映射。需要注意的是其中内核栈的顶部设置了一个无效区段，用于限制栈大小并使栈溢出时产生异常。（注：无效 = 此处的映射无需构建）。

此外，映射时需注意 PTE 的控制位设置，详情可见 Exercise 2 中提供的参考材料。



根据分析后实现代码如下：

```
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
boot_map_region(
    kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE,
    PADDR(bootstacktop)-KSTKSIZE, PTE_W
);
boot_map_region(kern_pgdir, KERNBASE, (size_t)(0x40000000000 - KERNBASE), 0, 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 phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	See Next Question

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?

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

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?

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?

尝试回答上述问题。

2. 要填写这部分内容，我们可以根据 va 地址的结构与 memlayout.h 的内容尝试填写：（注：仅标注代表性的 KERNBASE，其他部分均可由 memlayout.h 导出）：

Entry	Base Virtual Address	Points to(logically):
1023(0x3ff)	0xffc00000	Page table for top 4MB of phys memory
1022(0x3fe)	0xff800000	?
960(0x3c0)	0xf0000000	KERNBASE
...	...	...
2(0x2)	0x00800000	Program Data Heap
1(0x1)	0x00400000	Empty Memory
0	0x00000000	See Next Question

3. 位于内核态的映射均没有将 PTE_U 置位，通过 CPU 的页访问权限机制即可限制用户态代码对内核态内存的访问。

4. 系统最大支持 4GB 内存，这是由 32 位地址空间决定的。

5. 这个问题要求我们指出如果我们确实拥有 4GB 物理内存，管理内存需要多少空间开销。根据 Lab 的内容我们可知，管理内存需要三部分设施：

(1) 一个 PageInfo 数组 npages，所需的空间开销为 sizeof(PageInfo) * (4G/4K)；

(2) 一个 PageDirectory，需要 4KB；

(3) 一组页表，假定 PageDirectory 所有 ENTRY 都存在，需要 1024 * 4KB = 4MB。

6. 当 entrypgdir.c 中我们同样也将 VAS 中低位的 4MB 映射到 PAS 的低位 4MB 中，故可以进行正常访问。这个设置是必须执行的，因为打开分页后所有的地址均为虚拟地址，若不设置该映射则访问出错；栈的切换是在 entry 中执行的（见 entry.S）：

```
.globl entry
entry:

    ...

    # Now paging is enabled, but we're still running at a low EIP
    # (why is this okay?).  Jump up above KERNBASE before entering
    # C code.
    mov  $relocated, %eax
    jmp  *%eax
```