第二章 Memory Management

2.1 Physical Page Management

操作系统需要记录哪些内存区域是空闲的，哪些是正在被使用的。jOS 使用页为最小粒度来管理 PC 的物理内存区域，使其可以使用 MMU 对分配的内存进行映射和保护。

下面需要编写一个物理内存分配器：它使用一个 struct PageInfo 的链表来跟踪哪些内存是空闲的。每个结构对应一个物理页。此物理内存分配工具是其他的虚拟内存工具的基础。

> Exercise 1
> 在 kern/pmap.c 中实现以下的函数：
> • boot_alloc()
> • mem_init()
> • page_init()
> • page_alloc()
> • page_free()
> check_page_free_list() 和 check_page_alloc() 可以用于检查分配器实现是否正确。通过启动 jOS 来调用 check_page_alloc()

`Exercise 1 实验过程`

首先通过观察 kern/init.c，发现内核在初始化的时候会调用 meminit 函数，而 mem_init() 在调用 i386_detect_memory 函数检测系统中有多少可用的内存空间之后就会调用 boot_alloc 函数进行空间分配。通过 boot_alloc 函数的注释可知，这个函数只是暂时用于被当做页分配器，而使用的真实也分配器是 page_alloc 函数。boot_alloc 只维护一个局部静态变量 nextfree 用于存放下一个可以使用的空闲空间的虚拟地址。所以每次需要分配 n 个字节的内存的时只需要修改这个值即可。除了修改这个值之外还需要判断是否有足够的内存可以分配。因此 boot_alloc 添加的代码如下。在`kern/pmap.c` 的 `boot_alloc(PGSIZE)` 执行完成后，也就分配了一个页的内存。

```
result = nextfree;
nextfree = ROUNDUP(nextfree + n, PGSIZE);
if((uint32_t)nextfree - KERNBASE > (PGSIZE * npages))
  panic("boot alloc: out of memory.\n");
return result;
```

上述命令执行完成后，下一条更改 kern_pgdir 的指令为页目录添加一个目录表项。UVPT 是一段虚拟地址的起始地址，而 PADDR 取的是 kern_pgdir 的物理地址，而这一条指令也就是将虚拟地址映射到物理地址。根据注释可知，对于内核和用户而言这段内存都是只读的。

接下来就是需要补充完整的 mem_init 的部分。通过注释得知这一部分需要分配一块内存用于存放一个 PageInfo 数组，数组中的每一个 PageInfo 用于记录内存中的一页。同样，使用 boot_alloc 完成这一部分，这个数组的大小应该 *npages* × sizeof(*P ageInfo*)。`kern/pmap.c` 的 `mem_init` 函数补全的代码如下：

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

接下来 mem_init 将执行 page_init 函数，也是下一个需要补全的函数。同样，通过注释可以发现这个函数的功能有两个：初始化业庙的结构以及 pages_free_list，也就是空闲页的信息。在这个函数调用完成后 boot_alloc 将不再被使用。

根据注释中的信息，得知第 0 页、IO hole、以及 extended memory 中的内核部分和一些其它部分已经被占用，因此，最终 `kern/pmap.c` 的 `page_init` 函数修改得到的代码如下：

```
size_t i;
  const size_t pages_in_use_end = npages_basemem + 96 +
((uint32_t)boot_alloc(0) - KERNBASE) / PGSIZE;

  // 0 page is in use
  pages[0].pp_ref = 1;
  // low base memory is free
  for (i = 1; i < npages_basemem; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
  }
  // IO hole
  for(i = npages_basemem; i < pages_in_use_end; i++){
    pages[i].pp_ref = 1;
  }
  // extended memory
  for(int i = pages_in_use_end; i < npages; i++){
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
  }
```

处理与物理内存页有关的数据之后执行的 check_page_free_list(1) 以及 check_page_alloc() 是对于已经分配的页表的检查，查看页表以及空闲页是否为合法，以及检查 page_alloc 以及 page_free 是否能够正确运行。接下来就应该实现 page_alloc 以及 page_free 函数了。

通过 page_alloc 的注释可以知道这个函数的功能是分配一个物理页，并返回这个物理页所对应的 PageInfo 结构，其主要的工作是对于 free_page_list 进行更改。完成后的 `kern/pmap.c` 的 `page_alloc` 函数代码如下：

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

```
  temp = page_free_list;
  page_free_list =  temp -> pp_link;
  temp -> pp_link = NULL;
  if(alloc_flags & ALLOC_ZERO)
    memset(page2kva(temp), 0, PGSIZE);
  return temp;
}
```

首先检查 page_free_list 中是否还有空闲的节点，如果有则将其取出进行分配，否则返回 NULL。然后根据注释的要求，如果 alloc_flags & ALLOC_ZERO 为真，则将这一页内容全部置零。

接下来实现 `kern/pmap.c` 的 `page_free` 函数。同样，根据注释，要实现这个函数就是要将一个 PafeInfo 重新连接到 page_free_list 之中，代表对于这个页的回收。因此实现的代码如下。根据注释提示，当 pp_ref 或者 pp_link 不为 0 时应该发出 panic。

```
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 || pp -> pp_link)
    panic("page_free: illegal PageInfo.");
  pp -> pp_link = page_free_list;
  page_free_list = pp;
}
```

至此对于物理内存管理的代码已经补充完整。**注释掉 mem_init 中的第一个 panic**（不注释的话这个 panic 会导致 make grade 拿不到分） 并重新编译、运行 qemu，得到的输出如图 2.1 所示。可以看到，check_page_free_list 和 check_page_alloc 的检查已经通过。而最后的 check_page 检查需要在下一个实验中完成。

```
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,medi
a=disk,format=raw -serial mon:stdio -gdb tcp::25000 -D qemu.log
6828 decimal is XXX octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
kernel panic at kern/pmap.c:755: assertion failed: page_insert(kern_pgdi
r, pp1, 0x0, PTE_W) < 0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

图 2.1: 重新编译后运行 qemu 的输出

2.2 Virtual Memory

> Exercise 2
>
> 阅读 Intel 80386 Reference Manua的第 5 章和第 6 章，并仔细阅读有关分页转换以及基于分页的保护。

`Exercise 2 实验过程`

对于分页转换而言，需要注意的是线性地址的格式、页表的格式以及线性地址如何转换为物理地址。线性地址的 31~22bit 为页目录地址，21~12 为页地址，而 11~0 位为偏移量。线性地址到物理地址的转换过程如图2.2所示。

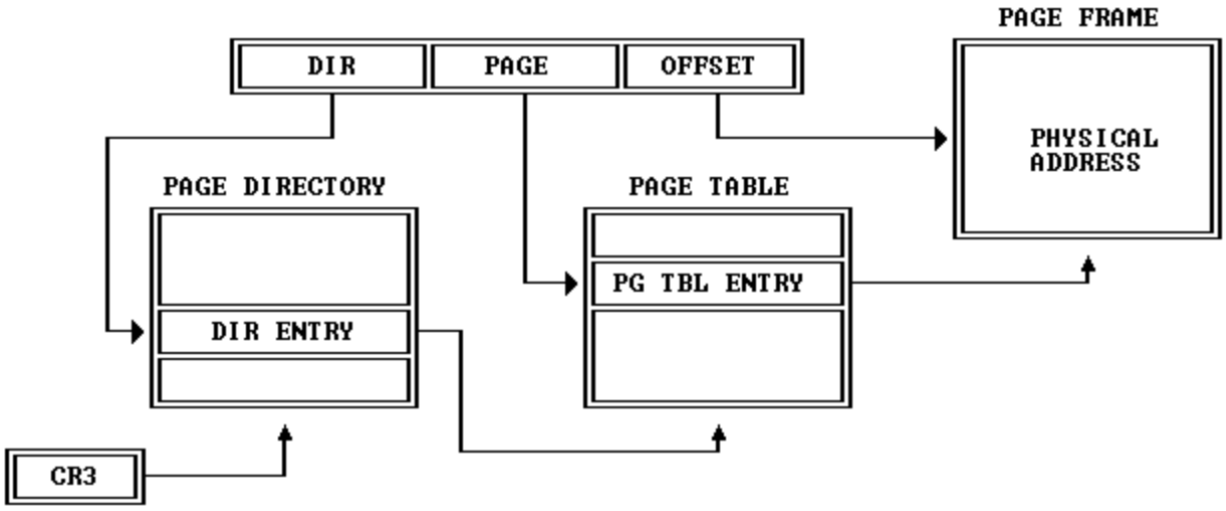

图 2.2: 线性地址到物理地址的转换

而对于每一个页表项，其不仅包含物理页的地址，还包含一系列的表示位用于表示页的属性，如权限、使用标志等。基于分页的保护就是建立在这种页表项的基础之上的。基于分页的保护参数包括 2 个标志位，一个是页表项中的 U/S 位，当其为 0 时表示监督模式，此时这个分页用于操作系统以及系统软件，当其为 1 时表示用户模式，为用户态软件提供服务。另一个标志位则是 R/W 位，当其为 0 时表示只读模式，为 1 时则表示读写模式。

2.2.1 Virtual, Linear, and Physical Addresses

在 x86 系统中，一个虚拟地址由两部分组成：段选择器和段内偏移。线性地址值得是通过段地址转换机构把虚拟地址进行转换之后得到的地址，而物理地址则是分页地址转换机构将线性地址转换之后得到的真实内存地址。

C 语言中的指针是虚拟地址中的段内偏移部分。boot/boot.S 中的的全局描述符表将所有段的基址设置为 0，上限设置为 0xffffffff，因而关闭了分段管理的功能。此时虚拟地址中段选择器字段的内容失去了意义，因为线性地址的值总是等于虚拟地址中段内偏移的值。lab1 中 part3 引入了一个简单的页表，这个页表只映射了 4MB 的空间。在 jOS 中需要将这种映射扩展到 256MB 的空间上。

> Exercise 3

> Gdb 只能够通过虚拟地址访问 qemu 的内存，但如果能够访问到实际内存地址将会是十分有用的。在运行 qemu 的终端中按下 Ctrl-a c 即可进入 qemu 的监视器在 qemu 监视器中使用 xp 命令以及 gdb 中的 x 命令来查看虚拟地址和对应物理地址的内存内容。对于 6.828 patch 版本的 qemu 而言，info pg 能够显示页表的内容，而 info mem 能够显示所有已经被页表映射的虚拟地址的空间以及它们的访问优先级。

`Exercise 3 实验过程`

编译并运行 qemu，在进入内核之后，在 GDB 中使用 x 命令检查 0xf0100000 处的部分内容，并使用 qemu 的 xp 命令检查 0x00100000 处的部分内容，其结果如图2.3所示。

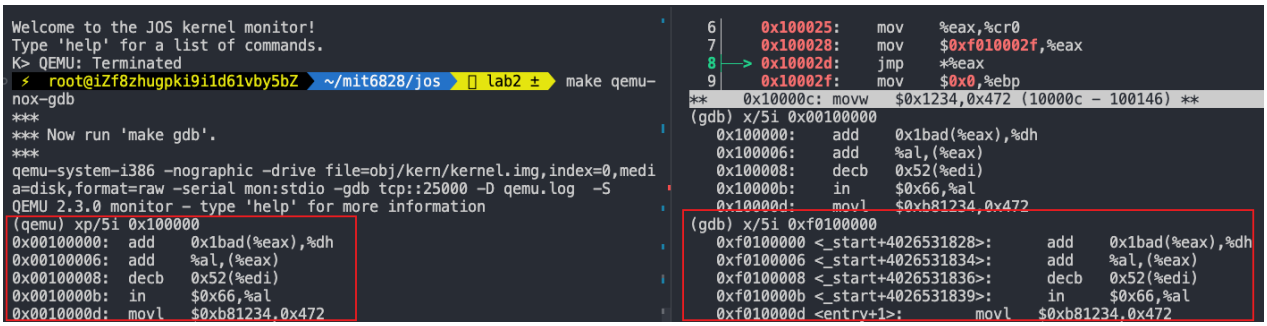

图 2.3: qemu 以及 gdb 对于映射地址内容的显示

从图中可以看出，映射后的虚拟地址的内容和物理地址的内容是一致的。在 qemu 中输入 info pg 以及 info mem 后其输出如图2.4所示。从图中可以看出，qmeu 打印出了页表的内容以及映射的虚拟地址空间。

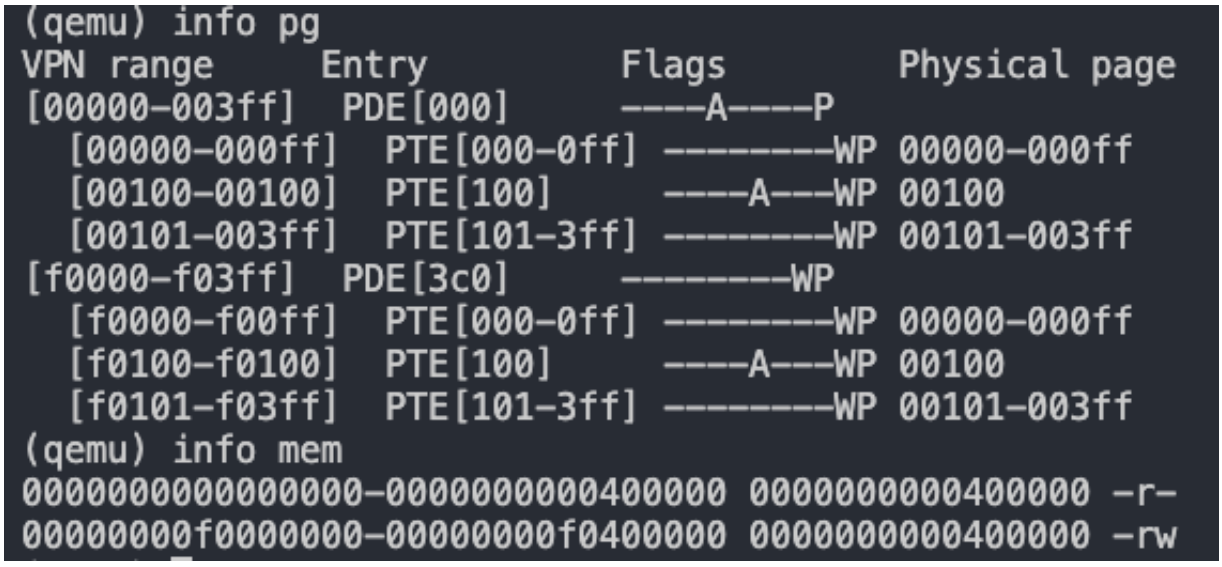

图 2.4: qemu 中 info pg 以及 info mem 的输出

进入保护模式之后，就不能够直接使用物理地址和线性地址了，所有代码中的地址都是虚拟地址的形式，并被 MMU 转换。jOS 内核需要同时操作虚拟地址和物理地址但不对其解引用。为了帮助记录代码，jos 中有两个类型的指针：unitptr_t 表示虚拟地址，而 physaddr_t 表示物理地址。实际上，它们都是 32 位整型值 (uint32_t)。正因为如此编译器不会阻止将一个类型赋值给另一个类型，但是会阻止对这些值的解引用。要对其解引用，需要首先将其强制转换为一个指针类型。

> Question
>
> 1. 假设以下 jOS 内核代码是正确的，那么 x 应该是 unitptr_t 类型还是 physaddr_t 类型?

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

Answer

由于使用了 * 操作符进行了解引用，因此变量 x 应该为虚拟地址，是 uintptr_t 类型。

2.2.2 Reference Counting

在之后的实验中将会遇到多个不同的虚拟地址被同时映射到相同的物理页面上的情况。在这种情况下我们需要记录每一个物理页面上存在多少不同的虚拟地址来引用它。这个值存放在 PageInfo 的 pp_ref 中。当这个值变为 0 时物理页才可以被释放。通常情况下任意一个物理页 p 的 pp_ref 的值等于它所在的所有页表中被虚拟地址 UTOP 下方的虚拟页映射的次数。

2.2.3 Page Table Management

这一部分需要创建一个页表管理程序，包括插入和删除线性地址到物理地址的映射，以及页表的创建等。

Exercise 4

完成 kern/pmap.c 中以下几个函数的实现。

- pgdir_walk()
- boot_map_region()
- page_lookup()
- page_remove()
- page_insert()

mem_init 中调用的 check_page 可以用于检查这些函数的实现是否正确。

`Exercise 4 实验过程`

首先实现 `kern/pmap.c` 的函数 `pgdir_walk`。根据注释，可以得知该函数所需要实现的功能为：给定一个页表目录指针 pgdir，该函数应该返回线性地址 va 对应的页表项的指针。此时可能这个页不存在，当这个页不存在时，如果 create==true 则分配新的页，否则返回 NULL，也就是说实现函数时需要注意页目录项对应的页表是否存在于内存中。最终，函数的实现如下：

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
  // Fill this function in
  pde_t * pgdir_entry = pgdir + PDX(va);
  if(!(*pgdir_entry & PTE_P)){
    if(!create)
      return NULL;
```

```
    else{
      struct PageInfo* new_page = page_alloc(1);
      if(!new_page)
        return NULL;
      *pgdir_entry = (page2pa(new_page) | PTE_P | PTE_W | PTE_U);
      ++new_page->pp_ref;
    }
  }

  return (pte_t *)(KADDR(PTE_ADDR(*pgdir_entry))) + PTX(va);
}
```

接下来是 `kern/pmap.c` 的 `boot_map_region` 函数。这个函数要求将虚拟空间 [*va, va + size*) 映射到物理空间 [*pa, pa + size*) 的这种映射关系加入到 pgdir 中，其中 size 是 PGSIZE 的整数倍大小。此函数主是为了设置 UTOP 之上的静态映射地址范围。最终，函数的实现如下，函数中，每一轮循环将一个物理页和虚拟也的映射关系加入到页表中，直到将 size 个字节分配完。

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa,
int perm)
{
  // Fill this function in
  int offset;
  pte_t *pgtable_entry;
  for(offset = 0; offset < size; offset += PGSIZE, va+= PGSIZE, pa +=
PGSIZE){
    pgtable_entry = pgdir_walk(pgdir, (void *)va, 1);
    *pgtable_entry = (pa | perm | PTE_P);
  }
}
```

对于 page_lookup 函数而言，它返回 va 所映射的物理页的 PageInfo 指针，且若 page_store 不为 0 则将物理页的页表地址存入 pte_store 中，如果 va 没有被映射则返回 NULL。其实现如下：

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

下面是 page_remove 函数，其作用是将虚拟地址 va 的映射关系删除。其中，pp_ref 的值要减 1，且如果 pp_ref 减为 0 则需要将这个页回收，最后，这个页的页表项需要被置为 0，且如果移走了这个页，TLB 需要被无效化。最终的 page_remove 实现如下：

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

最后，是对于 page_insert 的实现。其功能与 page_remove 相对，也就是将映射到虚拟内存中的 va 映射到物理内存页 pp 上。如果 va 已经映射，则应该使用 page_remove 将其移除。并且页表应该按需分配并插入到 pgdir 中，且 pp_ref 需要加一，最后的函数实现如下：

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
  // Fill this function in
  pte_t *pgtable_entry = pgdir_walk(pgdir, va, 1);
  if(!pgtable_entry)
    return -E_NO_MEM;
  ++pp -> pp_ref;
  if((*pgtable_entry) & PTE_P){
    page_remove(pgdir, va);
  }
  *pgtable_entry = (page2pa(pp) | perm | PTE_P);
  *(pgdir + PDX(va)) |= perm;
  return 0;
}
```

将这些函数补充完整后，重新编译并启动 qemu。qemu 的输出结果如图2.5所示。

```
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,medi
a=disk,format=raw -serial mon:stdio -gdb tcp::25000 -D qemu.log
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!
kernel panic at kern/pmap.c:670: assertion failed: check_va2pa(pgdir, UP
AGES + i) == PADDR(pages) + i
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

图 2.5: 重新编译运行后 qemu 的输出

从图中可以看到，check_page 的检查已经通过，说明上述 5 个函数的实现基本正确，接下来的 kernel panic 需要在下一个实验中解决。

2.3 Kernel Address Space

jOS 将处理器的线性地址划分为占用低地址的用户环境和占用高地址的内核，其界限是 `inc/memlayout.h` 中的变量 ULIM。jOS 为内核保留了 256M 的地址空间，这也就是为什么在 Lab 1 中要给操作系统设计一个非常高的地址空间。

Memlayout.h 内存布局：

```
/*
 * Virtual memory map:                                Permissions
 *                                                    kernel/user
 *
 *    4 Gig --------->  +------------------------------+
 *                      |                              | RW/--
 *                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                      :              .               :
 *                      :              .               :
 *                      :              .               :
 *                      |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
 *                      |                              | RW/--
 *                      |   Remapped Physical Memory   | RW/--
 *                      |                              | RW/--
 *    KERNBASE, ---->   +------------------------------+ 0xf0000000      --
+
 *    KSTACKTOP         |     CPU0's Kernel Stack      | RW/--  KSTKSIZE
|
 *                      | - - - - - - - - - - - - - - -|
|
 *                      |      Invalid Memory (*)      | --/--  KSTKGAP
|
```

```
 *                    +------------------------------+
|
 *                    |     CPU1's Kernel Stack      | RW/--  KSTKSIZE
|
 *                    | - - - - - - - - - - - - - - -|
PTSIZE
 *                    |      Invalid Memory (*)      | --/--  KSTKGAP
|
 *                    +------------------------------+
|
 *                    :              .               :
|
 *                    :              .               :
|
 *    MMIOLIM ------>  +------------------------------+ 0xefc00000      --
+
 *                    |       Memory-mapped I/O      | RW/--  PTSIZE
 * 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
 *                    |      Normal User Stack       | RW/RW  PGSIZE
 *                    +------------------------------+ 0xeebfd000
 *                    |                              |
 *                    |                              |
 *                    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                    .                              .
 *                    .                              .
 *                    .                              .
 *                    |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
 *                    |     Program Data & Heap      |
 *    UTEXT -------->  +------------------------------+ 0x00800000
 *    PFTEMP ------->  |       Empty Memory (*)       |        PTSIZE
 *                    |                              |
 *    UTEMP -------->  +------------------------------+ 0x00400000      --
+
 *                    |       Empty Memory (*)       |
|
 *                    | - - - - - - - - - - - - - - -|
|
 *                    |  User STAB Data (optional)   |
PTSIZE
```

```
56    *    USTABDATA ----->  +------------------------------+ 0x00200000
     |
57    *                      |       Empty Memory (*)       |
     |
58    *    0 ------------>   +------------------------------+                --
     +
```

2.3.1 Permissions and Fault Isolation

由于内核和用户进程只能访问各自的地址空间，所以我们不许在 x86 中使用权限位来将用户的代码限制在只能访问在用户空间可写权限位 PTE_W 可以同时影响内核和用户代码。高于 ULIM 的内存内核可读写但用户没有权限。用户和内核在 [*UT OP, ULIM*) 有同样的可读不可写权限。低于 UTOP 的地址是用户空间。

2.3.2 Initializing the Kernel Address Space

在这一部分，需要设置 UTOP 之上的，给内核使用的空间。inc/memlayout.h 展示了应该使用的内存布局。

> Exercise 5
>
> 将 mem_init() 内的 check_page() 之后的代码补充完整。可以使用 check_kern_pgdir() 以及 check_page_installed_pgdir() 来检查实现的代码。

`Exercise 5 实验过程`

根据 `kern/pmap.c` 的 函数 `mem_init` 中调用 `check_page` 之后的注释。首先，我们需要将 pages 数组映射到线性地址 UPAGES 上，此处使用 `boot_map_region` 函数进行映射，添加的代码为：

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

然后映射物理地址到内核的栈区域，将 bootstack 标记的物理地址映射到内核的堆栈。但是只需要映射 [*KST ACKTOP – KSTKSIZE, KSTACKTOP*) 这部分区域。因此此处补充的代码为：

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

最后，

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

最后，重新编译 jOS 并启动 qemu，其输出如图2.6所示。从图中可以看出，所有的测试已经通过。

```
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,medi
a=disk,format=raw -serial mon:stdio -gdb tcp::25000 -D qemu.log
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>
```

图 2.6: 重新编译并启动 qemu 后的输出结果

Question

2. 至此 page directory 被填入了哪些行？它们又被映射到了哪里？填写下面这张表。

Entry	Base Virtual Address	Points to(logically)
1023	?	Page table for top 4MB of physical memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	?

回答:

Entry	Base Virtual Address	Points to(logically)
1023	0xffc00000	Page table for top 4MB of physical memory
..s.	...	...
960	0xf0000000	KERNBASE
959	0xefc00000	Kernel Stack
957	0xef400000	Page Directory
956	0xef000000	PageInfo array
...	...	...
0	0x00000000	Empty

3. 如果我们把 kernel 和 user environment 放在同一个地址空间中，为什么用户程序不能读写内核的内容？什么机制防保护了内核空间?

   回答：因为用户不能访问没有 PTE_U 权限的页，叶保护机制保护了内核空间。

4. 这个操作系统最大可以支持多少内存？为什么?

   回答：最大可以支持 4G 的空间。由 32 位地址控制

6. 回顾 kern/entry.S 以及 kern/entrypgdir.c。在分页刚被打开时，EIP 仍然是一个很低的数（1MB 多一点）。在哪里 EIP 变得比 KERNBASE 高的？在刚开启分页到 EIP 比 KERNBASE 高的这段时间内，是什么让 EIP 保持一个很小的值但是程序继续运行的？为什么这种转换是必要的?

回答：执行了以下代码以后 EIP 变得比 KERNBASE 高。

```
mov $relocated, %eax
jmp *%eax
```

之所以代码能够继续执行，是因为 0~4MB 的地址空间映射到了 0~4MB 的物理地址 (kern/entrypgdir.c) 中。转换的必要性在于需要在高地址处执行代码。