操作系统实习报告

曹胜操 - 1500012838

Lab 2: Memory Management

Part 1: Physical Page Management

> Exercise 1. 在`kern/pmap.c`中实现以下函数：`boot_alloc()` `mem_init()`（只到调用`check_page_free_list(1)`之前） `page_init()` `page_alloc()` `page_free()`。
>
> `check_page_free_list()`和`check_page_alloc()`会检查这一部分。

`boot_alloc()`这一函数只会在JOS对虚拟内存系统进行初始化时被使用。实际上，我们需要添加的代码并不多，只需要按照对齐要求和大小限制给出下一个可行的地址即可。我们添加以下代码：

```
if ((uint32_t)nextfree >= KERNBASE + npages * PGSIZE)
    panic("boot_alloc: Out of memory!\n" );
result = nextfree;
nextfree += ROUNDUP(n, PGSIZE);
return result;
```

`mem_init()`这一函数会开始设置两级页表，我们在这里需要完成的任务是初始化`pages`这个数组，其中将会存储每一个物理页面的信息。

```
uint32_t pages_size = npages * sizeof(struct PageInfo);
pages = (struct PageInfo*)boot_alloc(pages_size);
memset(pages, 0, pages_size);
```

`page_init()`这一函数会对物理页面的信息进行初始化，主要是是否可用。根据题目要求和参考Lab1的内容，我们可以推导：

- $[0, 1)$页面不可用，保留给实模式IDT和BIOS。
- $[1, \mathbf{npages_basemem})$页面可用，这些被称为Base memory。
- $[\mathbf{IOPHYSMEM/PGSIZE, EXTPHYSMEM/PGSIZE})$页面不可用，这些被保留给I/O。
- $[\mathbf{EXTPHYSMEM/PGSIZE}, \ldots)$页面部分不可用，因为已经被Kernel和页表等其它内容所占用（并且以后也不能作为它用），之后的页面可用。

由此，令`size_t t0 = 0, t1 = 1, t2 = npages_basemem, t3 = PGNUM(PADDR(boot_alloc(0))), t4 = npages;`，则这些下标可以作为页面是否可用的分界线。注意链表的连接等细节，我们可以添加以下代码：

```
page_free_list = NULL;
size_t t0 = 0, t1 = 1, t2 = npages_basemem,
       t3 = PGNUM(PADDR(boot_alloc(0))), t4 = npages;
size_t i;
for (i = 0; i < npages; ++i)
{
    if (t0 <= i && i < t1)
    {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
    }
    if (t1 <= i && i < t2)
    {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
    if (t2 <= i && i < t3)
    {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
    }
    if (t3 <= i && i < t4)
    {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

`page_alloc()`会通过空闲页面链表分配出一个可用的物理页面。需要特殊处理的情况包括：链表为空（没有更多空闲页面）、要求用零填充（通过`alloc_flags`），另外`pp_ref`不需修改。

```
struct PageInfo *
page_alloc(int alloc_flags)
{
    if (!page_free_list) return NULL;
    struct PageInfo* pp = page_free_list;
    page_free_list = pp->pp_link;
    pp->pp_link = NULL;
    if (alloc_flags & ALLOC_ZERO) memset(page2kva(pp), 0, PGSIZE);
    return pp;
}
```

`page_free()`会将一个物理页面重新设为空闲，加入链表。对于错误的调用（检查`pp_ref` `pp_link`是否非零）可以`panic()`。

```
void
page_free(struct PageInfo *pp)
{
    if (pp->pp_ref) panic("page_free: pp_ref is nonzero!\n" );
    if (pp->pp_link) panic("page_free: pp_link is not NULL!\n" );
    pp->pp_link = page_free_list;
    page_free_list = pp;
}
```

Part 2: Virtual Memory

> Exercise 2. 阅读https://pdos.csail.mit.edu/6.828/2016/readings/i386/toc.htm的第5、6章，特别是5.2和6.4。建议浏览关于Segmentation的部分。

这里主要是需要熟悉如何进行地址翻译和实现保护机制，接下来的内容需要我们对这些内容有所了解，否则难以理解题目的要求。

> Exercise 3. QEMU中的`xp`指令可以查看物理地址对应的内存，相对于GDB的`x`，这在刚开始设置虚拟内存时会十分有用。使用这两种指令，保证对应的物理地址中和虚拟地址中有着相同的内容。
>
> `info pg` `info mem`指令也可以给出一些有用的信息。

运行JOS后，我们可以通过这些指令查看内存信息。以Kernel开头的entry部分为例：

- 在QEMU中，输入`xp/16wx 0x00100000`，通过物理地址查看：

```
(qemu) xp/16wx 0x00100000
0000000000100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0000000000100010: 0x34000004 0x5000b812 0x220f0011 0xc0200fd8
0000000000100020: 0x0100010d 0xc0220f80 0x10002fb8 0xbde0fff0
0000000000100030: 0x00000000 0x115000bc 0x0002e8f0 0xfeeb0000
```

- 在GDB中，输入`x/16wx 0xf0100000`，通过虚拟地址查看：

```
(gdb) x/16wx 0xf0100000
0xf0100000 <_start+4026531828>:  0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0xf0100010 <entry+4>:    0x34000004  0x5000b812  0x220f0011  0xc0200fd8
0xf0100020 <entry+20>:   0x0100010d  0xc0220f80  0x10002fb8  0xbde0fff0
0xf0100030 <relocated+1>:    0x00000000  0x115000bc  0x0002e8f0  0xfeeb0000
```

- 可见，这部分内存的确是被正确映射了。
- 在QEMU中，继续输入`info pg`，可以看到当前的页表信息：

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef03f]  PTE[000-03f] -------U-P 00119-00158
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 00118
  [ef7bf-ef7bf]  PTE[3bf]     -------UWP 003fe
  [ef7c0-ef7df]  PTE[3c0-3df] ----A--UWP 003ff 003fc 003fb 003fa 003f9 003f8 ..
  [ef7e0-ef7ff]  PTE[3e0-3ff] -------UWP 003dd 003dc 003db 003da 003d9 003d8 ..
[efc00-effff]  PDE[3bf]     -------UWP
  [efff8-effff]  PTE[3f8-3ff] --------WP 0010d-00114
[f0000-f03ff]  PDE[3c0]     ----A--UWP
  [f0000-f0000]  PTE[000]     -------UWP 00000
  [f0001-f009f]  PTE[001-09f] ---DA--UWP 00001-0009f
  [f00a0-f00b7]  PTE[0a0-0b7] -------UWP 000a0-000b7
  [f00b8-f00b8]  PTE[0b8]     ---DA--UWP 000b8
  [f00b9-f00ff]  PTE[0b9-0ff] -------UWP 000b9-000ff
  [f0100-f0104]  PTE[100-104] ----A--UWP 00100-00104
  [f0105-f0113]  PTE[105-113] -------UWP 00105-00113
  [f0114-f0114]  PTE[114]     ---DA--UWP 00114
  [f0115-f0116]  PTE[115-116] -------UWP 00115-00116
  [f0117-f0118]  PTE[117-118] ---DA--UWP 00117-00118
  [f0119-f0119]  PTE[119]     ----A--UWP 00119
  [f011a-f011a]  PTE[11a]     ---DA--UWP 0011a
  [f011b-f0158]  PTE[11b-158] ----A--UWP 0011b-00158
  [f0159-f03bd]  PTE[159-3bd] ---DA--UWP 00159-003bd
  [f03be-f03ff]  PTE[3be-3ff] -------UWP 003be-003ff
[f0400-f7fff]  PDE[3c1-3df] ----A--UWP
  [f0400-f7fff]  PTE[000-3ff] ---DA--UWP 00400-07fff
[f8000-fffff]  PDE[3e0-3ff] -------UWP
  [f8000-fffff]  PTE[000-3ff] -------UWP 08000-0ffff
```

- 输入`info mem`，则可以看到：

```
(qemu) info mem
00000000ef000000-00000000ef040000 0000000000040000 ur-
00000000ef7bc000-00000000ef7be000 0000000000002000 ur-
00000000ef7bf000-00000000ef800000 0000000000041000 ur-
00000000efff8000-00000000f0000000 0000000000008000 -rw
00000000f0000000-0000000100000000 0000000010000000 urw
```

在JOS中，我们用`uintptr_t`来代表一个虚拟地址，而用`physaddr_t`来代表一个物理地址。尽管它们实际上都是`uint32_t`的别名，但我们还是作此区分。

> Question 1. 假设下面的代码是正确的，`x`应该是什么类型？
>
> ```
> mystery_t x;
> char* value = return_a_pointer();
> *value = 10;
> x = (mystery_t) value;
> ```

这里的`value`指针中存储着一个虚拟地址，因此`x`的类型应是`uintptr_t`。

在JOS中，有时候我们也需要在虚拟地址和物理地址之间做转换。在Remapped region中，我们使用函数`KADDR(pa)`和`PADDR(va)`来完成这种转换。本质上它们只是加上或者减去了`0xf0000000`。

> Exercise 4. 在`kern/pmap.c`中，实现以下函数：`pgdir_walk()` `boot_map_region()` `page_lookup()` `page_remove()` `page_insert()`。
>
> `check_page()`会检查这一部分。

`pgdir_walk()`这一函数通过页目录地址`pgdir`和给定虚拟地址`va`找到对应的PTE。当这个PTE不存在时，根据第三个参数`create`的真假来决定是否创建。

只需要模拟虚拟地址通过两级页表翻译的过程就可以了，注意虚拟地址和物理地址之间的转换。

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    uint32_t pdx = PDX(va), ptx = PTX(va);
    if (pgdir[pdx])
        return (pte_t*)KADDR(PTE_ADDR(pgdir[pdx])) + ptx;
    else
        if (create)
        {
            struct PageInfo* pp = page_alloc(ALLOC_ZERO);
            if (pp)
            {
                ++pp->pp_ref;
                pgdir[pdx] = (pde_t)page2pa(pp) | PTE_P | PTE_W | PTE_U;
                return (pte_t*)KADDR(PTE_ADDR(pgdir[pdx])) + ptx;
            }
            else return NULL;
        }
        else return NULL;
}
```

`boot_map_region()`这一函数将一段虚拟地址映射到物理地址上，我们可以利用`pgdir_walk()`来找到对应的PTE。

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    size_t i;
    for (i = 0; i < size; i += PGSIZE, va += PGSIZE, pa += PGSIZE)
    {
        pte_t* ptep = pgdir_walk(pgdir, (void*)va, 1);
        if (ptep)
            *ptep = pa | perm | PTE_P;
        else
            panic("boot_map_region: invalid PTE!\n" );
    }
}
```

`page_lookup()`会找到某个虚拟地址对应页的`PageInfo`。需要注意的是`PTE_P`这一项。

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    pte_t* ptep = pgdir_walk(pgdir, va, 0);
    if (ptep && (*ptep & PTE_P))
    {
        if (pte_store) *pte_store = ptep;
        return pa2page(PTE_ADDR(*ptep));
    }
    else return NULL;
}
```

`page_remove()`会取消某个页的映射关系。先找到它，再通过`page_decref()`去减小`ref`。

```
void
page_remove(pde_t *pgdir, void *va)
{
    pte_t* ptep = NULL;
    struct PageInfo* pp = page_lookup(pgdir, va, &ptep);
    if (pp)
    {
        page_decref(pp);
        *ptep = 0;
        tlb_invalidate(pgdir, va);
    }
}
```

`page_insert()`会添加某个页的映射关系。如果之前已有映射关系，我们需要先用`page_remove()`来将其取消。在下面的写法中，我们先增加`ref`，再检查是否已有映射关系，这样在重复添加映射关系的时候也不会导致错误，因为避免了不应该有的`page_free()`。

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    pte_t* ptep = pgdir_walk(pgdir, va, 1);
    if (ptep)
    {
        ++pp->pp_ref;
        if (*ptep & PTE_P) page_remove(pgdir, va);
        *ptep = page2pa(pp) | perm | PTE_P;
        pgdir[PDX(va)] |= perm;
        return 0;
    }
    else return -E_NO_MEM;
}
```

Part 3: Kernel Addr ess Space

> Exercise 5. 填写`mem_init()`剩余的部分。

> `check_kern_pgdir()`和`check_page_installed_pgdir()`会检查这一部分。

我们需要对地址空间中`UTOP`以上的，即Kernel中的部分进行初始化，包括地址映射和权限设置。只需要使用刚才编写的`boot_map_region()`函数，根据注释中的提示来实现就可以了。

```
boot_map_region(kern_pgdir, UPAGES, ROUNDUP(pages_size, PGSIZE), PADDR(pages), PTE_U);
boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W);
boot_map_region(kern_pgdir, KERNBASE, ~KERNBASE + 1, 0, PTE_W);
```

> Question 2. 此时，页目录中哪些项已有内容？它们映射到的虚拟地址是什么？对应的内容又是什么？

根据上面填写的`boot_map_region()`的调用，结合QEMU控制台中`info pg`得到的结果和`memlayout.h`中的定义，我们可以填写：

Entry	Base Virtual Address	Points to
0x3C0~0x3FF	0xF0000000	`Remapped Physical Memory`
0x3BF	0xEFC00000	`Kernel Stack`
0x3BD	0xEF400000	`Cur. Page Table`
0x3BC	0xEF000000	`RO PAGES`

> Question 3. 我们把Kernel和用户环境放在同一个地址空间中。为什么程序不能读写属于Kernel的内存？是什么机制保护了Kernel内存？

页表机制保护了Kernel内存。用户态的代码即使能够得到Kernel内存中的虚拟地址，也没有对其进行读写的权限，这就起到了保护的作用。

> Question 4. 这个操作系统最大能支持多大的物理内存？为什么？

256MB。在Lab1中提到过，由于设计上的限制，JOS只会使用机器的前256MB内存，这实际上是因为它将所有虚拟地址都映射到了$[0, 2^{32} - \mathbf{KERNBASE})$这部分物理地址，而$2^{32} - \mathbf{KERNBASE} = 2^{32} - 15 \cdot 2^{28} = 256\mathbf{M}$，即只能利用256MB的内存。

通过实际运行JOS，可以观察到`npages`的值为32768，即只能支持32K个物理页。每个物理页的大小又是4KB，故总共能利用128MB的物理内存。但是这个限制是QEMU带来的，不是JOS本身支持的上限。

> Question 5. 如果物理内存最大化，管理内存所需的额外开销是多大？这是怎么划分的？

256MB对应着64K个物理页，存储页表需要256KB（页目录也在其中）。如果再加上存储`pages`需要512KB，则总共需要的额外开销为768KB。

> Question 6. 回顾`kern/entry.S`和`kern/entrypgdir.c`中的页表设置。开启页表机制时，`EIP`还是一个很小的数。什么时候我们转换到了`KERNBASE`之上的`EIP`？什么使得这个转换变得可能？这个转换为什么是必须的？

在`kern/entry.S`中，`jmp *%eax`使得`EIP`变为高地址，尽管从程序代码上是连续的。在`entrypgdir.c`中的初始页表里，低地址的映射是不变的，即使启用页表也仍然可以使用这些低地址。但是之后我们需要使用新的页表，并且Kernel也应该运行在高地址上，所以需要进行这一转换。

> Challenge! 扩展JOS的Kernel monitor，增添一些命令来显示或者修改内存的相关信息，以便于对Kernel进行调试。

在这个Challenge中，我们实现以下命令：

- `showmappings start_addr [end_addr]`：显示从`start_addr`到`end_addr`（或者仅仅`start_addr`）这段虚拟地址所在的页面映射到的物理地址及其权限。
- `setpermission perm start_addr [end_addr]`：修改从`start_addr`到`end_addr`（或者仅仅`start_addr`）这段虚拟地址所在的页面的权限位为`perm`。仅包括最低三位：`User` `Writeable` `Present`。
- `dumpmemory addr_type start_addr num`：显示从`start_addr`开始的`num`个字的内容，如果`addr_type`的第一个字符为`p`或`P`则代表物理地址，否则为虚拟地址。
- `pageinfo page_idx num`：显示编号从`page_idx`开始的`num`个物理页面的信息，包括对应的物理地址和被引用的次数。

具体实现可见`kern/monitor.c`，主要是利用`kern/pmap.c`中的一些函数。以Kernel被加载到内存中的最开始一小段内容为例，下面是运行的效果：

```
% 显示几个页的映射关系
K> showmappings 0xf0100000 0xf0101000
Show mappings:
VA               PA              PERM (User Writeable Present)
0xf0100000       0x00100000      /WP
0xf0101000       0x00101000      /WP

% 改变一个页的权限位
K> setpermission 7 0xf0100000
Set permission:
Setting permission to: User Writeable Present
K> showmappings 0xf0100000
Show mappings:
VA               PA              PERM (User Writeable Present)
0xf0100000       0x00100000      UWP

% 恢复这几个页的权限位
K> setpermission 3 0xf0100000 0xf0101000
Set permission:
Setting permission to: Not-user Writeable Present
K> showmappings 0xf0100000 0xf0101000
Show mappings:
VA               PA              PERM (User Writeable Present)
0xf0100000       0x00100000      /WP
0xf0101000       0x00101000      /WP

% 显示这几个页对应的物理页面信息
K> pageinfo 256 2
Page info:
Page index       Physical address        Reference count
256              0x00100000              1
257              0x00101000              1

% 分别通过物理地址和虚拟地址查看Kernel最开始一段的内容
K> dumpmemory p 0x00100000 16
Dump memory:
Dumping memory starting from physical address 0xf0100000 16 words:
1badb002 00000000 e4524ffe 7205c766 34000004 6000b812 220f0011 c0200fd8
0100010d c0220f80 10002fb8 bde0fff0 00000000 116000bc 0002e8f0 feeb0000
K> dumpmemory v 0xf0100000 16
Dump memory:
Dumping memory starting from virtual address 0xf0100000 16 words:
1badb002 00000000 e4524ffe 7205c766 34000004 5000b812 220f0011 c0200fd8
0100010d c0220f80 10002fb8 bde0fff0 00000000 115000bc 0002e8f0 feeb0000
```

Reference

- www.cnblogs.com/fatsheep9146/category/769143.html
- https://pdos.csail.mit.edu/6.828/2016/labs/lab1/