

Lab 2: Memory Management

December 24, 2019

摘要

本次实验取自 MIT 6.828 Lab2，共分为三个部分。第一部分学习并编写物理页面管理。第二部分学习并修改虚拟内存相关组件。第三部分学习内存地址空间。

关键词：JOS、内存页面、虚拟内存

1 作业三

1.1 boot_alloc()

用来申请 n 个字节的空間（会被 4K 对齐），返回申请好的空间的首地址，如果 $n==0$ ，则返回当前未分配的空間的首地址。

```
1 static void *
2 boot_alloc(uint32_t n)
3 {
4     static char *nextfree;
5     char *result;
6
7     if (!nextfree) {
8         extern char end[];
9         nextfree = ROUNDUP((char*)end, PGSIZE);
10    }
11    if (n==0)
12        return nextfree;
```

```

13     result = nextfree;
14     nextfree += n;
15     nextfree = ROUNDUP((char*)nextfree, PGSIZE);
16     return result;
17 }

```

这里，end 代表 bss 段的末尾，即当前未分配空间的首部；ROUNDUP 函数实现向上取整倍数，用来 4K 对齐。

当 $n > 0$ 时，先将 nextfree，即当前未分配空间的首地址赋值给 result，再将 nextfree 向后加 4K 对齐的 n 个字节，这时 result 就是申请好的空间的首地址。

1.2 mem_init()

大部分函数都在这个函数里被调用，需要添加的是为 pages 申请空间并初始化的代码。代码如下：

```

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

```

这里 boot_alloc() 的参数是页数乘每页大小（结构体而非实际页），正好是需要申请的空间大小。

1.3 page_init()

负责 page 结构体和内存空闲链表的初始化。根据注释的要求实现代码如下：

```

1 void
2 page_init(void)
3 {
4     size_t i;
5     for (i = 0; i < npages; i++) {

```

```

6      if(i == 0) {
7          pages[i].pp_ref = 1;
8          pages[i].pp_link = NULL;
9      } else if (i >= 1 && i < npages_basemem) {
10         pages[i].pp_ref = 0;
11         pages[i].pp_link = page_free_list;
12         page_free_list = &pages[i];
13     } else if (i >= IOPHYSMEM / PGSIZE && i < EXTPHYSMEM/
14                PGSIZE) {
15         pages[i].pp_ref = 1;
16         pages[i].pp_link = NULL;
17     } else if (i >= EXTPHYSMEM / PGSIZE &&
18                i < ((int)(boot_alloc(0)) - KERNBASE) /
19                PGSIZE) {
20         pages[i].pp_ref = 1;
21         pages[i].pp_link = NULL;
22     } else {
23         pages[i].pp_ref = 0;
24         pages[i].pp_link = page_free_list;
25         page_free_list = &pages[i];
26     }
27 }

```

这里的 `pages` 是 `PageInfo` 结构体数组，`PageInfo` 有两个成员：`struct PageInfo *pp_link` 和 `uint16_t pp_ref`，`pp_ref` 指出当前页的被引次数，`pp_link` 指向下一个空闲页。

进行初始化时，当这个页是空闲页时，前者指向下一个空闲页，否则为空。即所有空闲页通过链表连接在一起，占用页则没有联系；但事实上全部的页（空闲页，占用页）相邻地分布在 `boot_alloc` 函数所申请的空間里，并通过指针进行访问。同时，需要保证第一个对象的 `pp_link` 为空，

空闲页的 `pp_ref` 置 0、占用页置 1，以及 `page_free_list` 永远指向表头。

1.4 `page_alloc()`

这个函数是为了分配一个物理页，如果当前有空闲的页，那么将其返回，并且找到下一个空闲的页。

```
1 struct PageInfo *
2 page_alloc(int alloc_flags)
3 {
4     if (page_free_list == NULL)
5         return NULL;
6     struct PageInfo* page = page_free_list;
7     page_free_list = page->pp_link;
8     page->pp_link = 0;
9     if (alloc_flags & ALLOC_ZERO)
10         memset(page2kva(page), 0, PGSIZE);
11     return page;
12 }
```

我们将表头，也就是 `page_free_list` 指向的对象从链表上取出（`pp_link` 置 0）并分配，在需要的情况下将新分配的页置 0。

1.5 `page_free()`

该函数释放一个页，当 `pp_ref` 不为 0（页面仍被使用）时，以及当 `pp_link` 不为空（页面已经被释放）时，需要提出 `panic`。

```
1 void
2 page_free(struct PageInfo *pp)
3 {
4     if (pp->pp_ref != 0)
5         panic("This page is still been used.");
6     if (pp->pp_link != 0)
7         panic("This page has been freed");
```

```

8     pp->pp_link = page_free_list;
9     page_free_list = pp;
10    return;
11 }

```

2 问题三

x 是 `uintptr_t` 类型。

因为这里变量 `value` 使用了 `*` 操作符解析地址，即先转为指针再解析引用的方式，是虚拟地址。所以变量 `x` 也应该是虚拟地址也就是 `uintptr_t` 类型。

3 作业四

为了补全代码，需要了解以下两个宏：

```

1 #define PADDR(kva) __paddr(__FILE__, __LINE__, kva)
2 #define KADDR(pa) __kaddr(__FILE__, __LINE__, pa)

```

`PADDR()` 接受一个内核虚拟地址，并返回相应的物理地址；`KADDR()` 接受一个物理地址并返回相应的内核虚拟地址。由于 JOS 将物理地址 0 映射到虚拟地址 `0xf0000000`，因此两个宏实际的操作是将地址加上或减去这个数。

3.1 `pgdir_walk()`

`pgdir_walk()` 用于查找虚拟地址对应的页表项地址，传入参数分别为页目录项指针、线性地址和一个用于判断页目录项不存在时是否进行创建参数。返回值为页表项指针，在页表不存在且 `create==false` 或创建页表失败时，返回 `NULL`。

查找页表项地址之前首先要获得页目录地址，对页表是否存在进行判断，如果页目录存在，直接进行映射；如果不存在，则新建一个页表后再进行映射。创建页表时使用 `page_alloc` 分配新的页表页，并为新建的物理页设置页目录时添加上权限位。查找过程如下图 5.1：

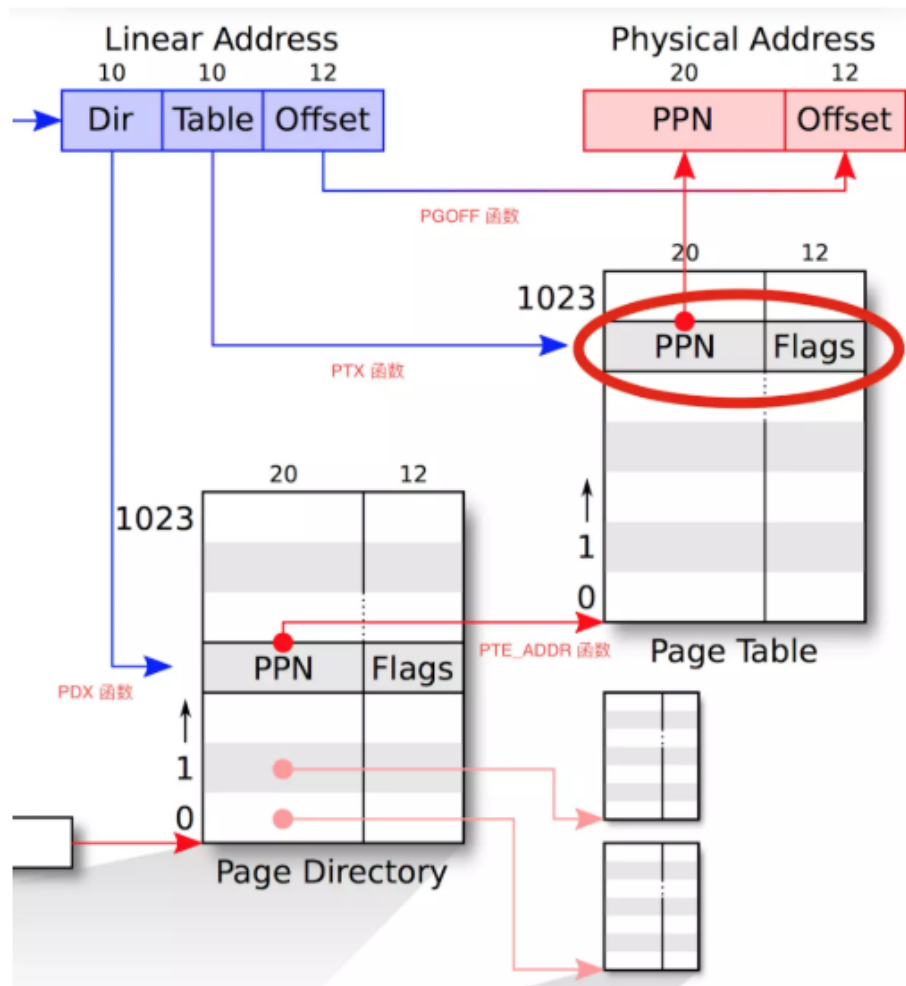


图 3.1: 查找过程

具体代码如下：

```

1 pte_t *
2 pgdir_walk(pde_t *pgdir, const void *va, int create)
3 {
4     // Fill this function in
5     pde_t *pt = pgdir + PDX(va);
6     pte_t *pt_addr_v;
7

```

```

8      if (*pt & PTE_P) {
9          pt_addr_v = (pte_t*)KADDR(PTE_ADDR(*pt));
10         return pt_addr_v + PTX(va);
11     } else {
12         struct PageInfo *newpt = page_alloc(ALLOC_ZERO);
13         if (create == 1 && newpt != 0) {
14             memset(page2kva(newpt), 0, PGSIZE);
15             newpt->pp_ref++;
16             *pt = PADDR(page2kva(newpt)) | PTE_U | PTE_W |
17                 PTE_P;
18             pt_addr_v = (pte_t*)KADDR(PTE_ADDR(*pt));
19             return pt_addr_v + PTX(va);
20         }
21     }
22     return NULL;
23 }

```

3.2 boot_map_region()

boot_map_region() 将虚拟地址 [va, va+size) 映射到物理地址 [pa, pa+size), 传入参数中, size 是 PGSIZE 的倍数, va 和 pa 都是 page-aligned。

反复调用 pgdir_walk, 获取页表地址, 其中 va 类型是 uintptr_t, 调用 pgdir_walk 时需要将其转换为 void*, 并对获取到的页表地址使用权限位 perm|PTE_P。

具体代码如下:

```

1 static void
2 boot_map_region(pde_t *pgdir, uintptr_t va, size_t size,
3                 physaddr_t pa, int perm)
4 {
5     int offset;
6     pte_t *pt;

```

```

6     for (offset = 0; offset < size; offset += PGSIZE) {
7         pt = pgdir_walk(pgdir, (void*)va, 1);
8         *pt = pa | perm | PTE_P;
9         pa += PGSIZE;
10        va += PGSIZE;
11    }
12 }

```

3.3 page_lookup()

page_lookup() 用于查找线性地址 va 对应的物理页面，找到就返回这个物理页，否则返回 NULL。参数为页目录指针、线性地址和指向页表指针的指针。

利用 pgdir_walk() 查找 va 对应的物理页面，如果 pte_store 不为零，则在其中存储此页的 pte 地址。

具体代码如下：

```

1 struct PageInfo *
2 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
3 {
4     pte_t *pte = pgdir_walk(pgdir, va, 0);
5     if (pte_store != 0) {
6         *pte_store = pte;
7     }
8     if (pte != NULL && (*pte & PTE_P)) {
9         return pa2page(PTE_ADDR(*pte));
10    }
11    return NULL;
12 }

```


3.4 page_remove()

page_remove() 在虚拟地址 “va” 处取消对物理页面的映射，如果该地址没有物理页面，就什么都不做。

通过 page_lookup 获得物理页，如果物理页存在，利用 page_decreef 进行删除工作，然后将 va 地址的页表项设为 0，并对有效性进行验证。

具体代码如下：

```
1 void
2 page_remove(pde_t *pgdir, void *va)
3 {
4     pte_t *pte;
5     struct PageInfo *page = page_lookup(pgdir, va, &pte);
6     if (page) {
7         page_decreef(page);
8         *pte = 0;
9         tlb_invalidate(pgdir, va);
10    }
11 }
```

3.5 page_insert()

page_insert() 建立一个虚拟地址与物理页的映射，将页面管理结构 pp 所对应的物理页面分配给线性地址 va。传入参数为页目录指针、页描述结构体指针、线性地址，和权限，建立映射成功，返回 0，失败返回-E_NO_MEM（内存不足）

利用 pgdir_walk() 查找该虚拟地址对应的页表项，查找失败，返回-E_NO_MEM，否则，对查找结果进行判断，如果映射到与之前相同的物理页，应调用 page_remove 删除之前的映射关系，重新建立映射。如果映射到与之前不同的物理页面，则将 va 对应的页表项中的物理地址重新赋值为 pp 所对应的物理页面的首地址。建立映射时应将对应的页表项的权限（低 12 位）设置成 PTE_P & perm，插入成功，-pp->pp_ref 递增。

具体代码如下：

```

1  int
2  page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int
   perm)
3  {
4      pte_t *pte = pgdir_walk(pgdir, va, 1);
5      if (!pte)
6          return -E_NO_MEM;
7      if (*pte & PTE_P) {
8          if (PTE_ADDR(*pte) == page2pa(pp)) {
9              tlb_invalidate(pgdir, va);
10             pp->pp_ref--;
11         } else {
12             page_remove(pgdir, va);
13         }
14     }
15     *pte = page2pa(pp) | perm | PTE_P;
16     pp->pp_ref++;
17     pgdir[PDX(va)] |= perm;
18     return 0;
19 }

```

对映射到的物理页进行区分会出一些问题，则不进行区分，直接删除之前的映射关系，对函数进行调整，代码如下：

```

1  int
2  page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int
   perm)
3  {
4      pte_t *pte = pgdir_walk(pgdir, va, 1);
5      if (!pte)
6          return -E_NO_MEM;
7      pp->pp_ref++;

```

```

8      if (*pte & PTE_P)
9          page_remove(pgdir, va);
10     *pte = page2pa(pp) | perm | PTE_P;
11     return 0;
12 }

```

4 作业五

按要求分别加入三处代码如下：

```

1 boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U
   );
2
3 boot_map_region_large(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE,
   PADDR(bootstack), PTE_W);
4
5 boot_map_region(kern_pgdir, KERNBASE, -KERNBASE, 0, PTE_W);

```

从 `mem_init()` 中的注释信息中，我们得知需使用 `boot_map_region()` 函数。

`boot_map_region()` 的作用是将一段连续的虚拟地址映射到它所对应的物理地址中。

观察这个函数的参数：第一个是页面的目录，第二个是虚拟地址，第三是映射范围大小，第四是对应物理地址，第五是赋予的权限。

第一部分的映射，起点是 `UPAGES`，大小是 `PTSIZE`，物理地址是 `PADDR(pages)`，权限是 `PTE_U|PTE_P`，这里不写 `PTE_P` 是因为函数内部已经自己默认赋予此权限了。

```

1 // UVPT  ——> +-----+ 0xef400000
2 //          /          RO  PAGES          / R-/R-  PTSIZE
3 // UPAGES ——> +-----+ 0xef000000

```

第二部分的映射，起点是 KSTACKTOP-KSTKSIZE，大小是 KSTKSIZE，物理地址是 PADDR(bootstack)，权限是 PTE_W，这里不写 PTE_P 是因为函数内部已经自己默认赋予此权限了。内核堆栈的虚拟地址范围是 [KSTACKTOP-PTSIZE, KSTACKTOP)，但是我们只把 [KSTACKTOP-KSTKSIZE, KSTACKTOP) 这部分映射关系加入到页表中，[KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) 这部分不进行映射。

第三部分的映射，虚拟地址范围是 [KERNBASE,4G]，物理地址范围是 [0,232-KERNBASE]。权限是 PTE_W 。

1	// 4 Gig	————>	+	—————	+
2	//		/		/ RW/--
3	//		~~~~~		
4	//		:	.	:
5	//		:	.	:
6	//		:	.	:
7	//		/~~~~~		/ RW/--
8	//		/		/ RW/--
9	//		/ Remapped Physical Memory		/ RW/--
10	//		/		/ RW/--
11	// KERNBASE,	————>	+	—————	+ 0xf0000000

5 问题四

5.1 第一问

假设下图描述的是系统的页目录表，哪些条目（行）已经被填充了？它们是怎样进行地址映射的？它们所指向的位置在哪里？请尽可能完善这张表的内容。

根据 memlayout.h 中的布局图，我们填写该表如下表 7.1：

Entry	Base Virtual Address	Points to (logically)
1023	0xffc0000	Page table for top 4MB of phys memory
1022	0xff80000
...		
960	0xf0000000	Page table for bottom 4MB of phys memory
959	0xefc00000	Current page table, kernel RW
958	0xef800000	Kernel stack
957	0xef400000	Current page table kernel R-, user R-
956	0xef000000	User pages
...		
2	0x00800000	
1	0x00400000	
0	0x00000000	Empty Memory

表 5.1: 问题 4-1

5.2 第二问

我们已经将内核和用户环境放在同一地址空间内。为什么用户的程序不能读取内核的内存？有什么具体的机制保护内核空间吗？

因为虚拟内存被 ULIM 和 UTOP 划分为不同的段，(ULIM, 4GB) 只允许内核读写，(UTOP, ULIM] 只允许内核和用户读取，而 [0x0, UTOP] 允许内核与用户读写。

我们通过页（目录）表中设置保护位的机制来确保这点，包括 PTE_W (writeable) 和 PTE_U (user)。

5.3 第三问

JOS 最大支持多大的物理内存，为什么？

4GB。因为页目录表一共包含了 1024 个指针，每个指向一个页表，而每个页表包含 1024 个指针，每个指向一个物理页面，其大小为 4096bytes。因此总大小为 $1024 * 1024 * 4096B = 4GB$ 。

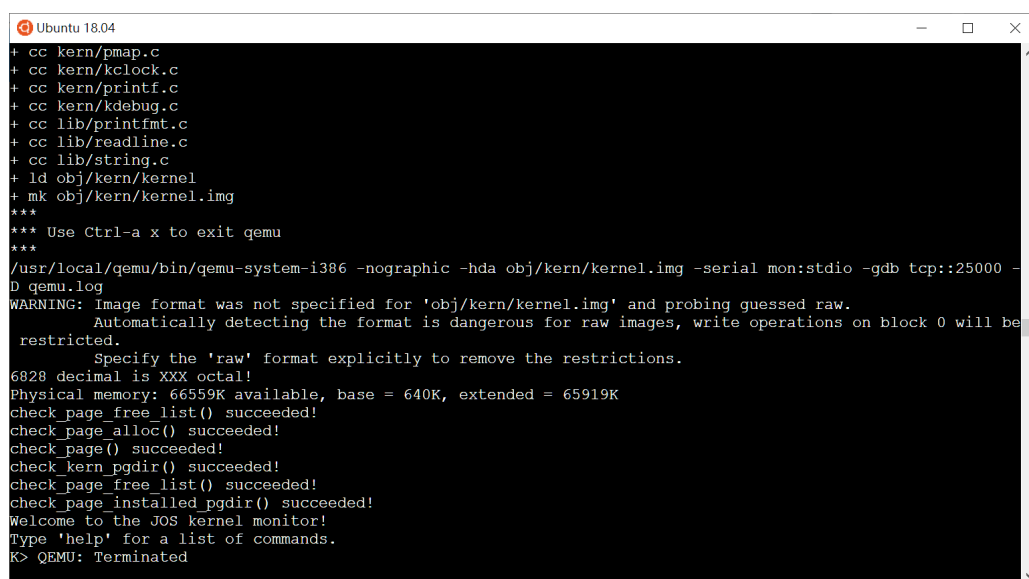
5.4 第四问

如果我们的硬件配置了可以支持的最大的物理内存，那么管理内存空间的开销是多少？这一开销是怎样划分的？

易得 $\text{sizeof PageInfo} = 8 \text{ bytes}$ ，因此 8MB 用于页面信息，4MB 用于页表，4KB 用于页目录表。

6 实验结果

执行 `make grade`，得到结果如下：



```
Ubuntu 18.04
+ cc kern/pmap.c
+ cc kern/kclock.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
***
*** Use Ctrl-a x to exit qemu
***
/usr/local/qemu/bin/qemu-system-i386 -nographic -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::25000 -D qemu.log
WARNING: Image format was not specified for 'obj/kern/kernel.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
6828 decimal is XXX octal!
Physical memory: 66559K available, base = 640K, extended = 65919K
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> QEMU: Terminated
```

图 6.2: lab2 实验结果-make qemu

```
Ubuntu 18.04
./grade-lab2
make[1]: Entering directory '/home/tsukimori/lab/lab1/src/lab1_1'
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/pmap.c
+ cc kern/kclock.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/tsukimori/lab/lab1/src/lab1_1'
running JOS: (1.5s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70

[tsukimori@DESKTOP-QL2F93P 18:48 #0] - [~/lab/lab1/src/lab1_1]
$
```

图 6.3: lab2 实验结果-make grade