Os lab2:Memory Management

陈炜栋 11300240057

November 23, 2013

Contents

1 Physical Page Management

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.

- 在exercise 1 里面我们需要实现5个函数，其中mem_init()只需要完成第一个检查点。首先看mem_init(),i386_用CMOScalls 获取 base memory 和 extended memory的大小，从而计算出总的页数，npage.
- 在这个时候首先调用boot_alloc(PGSIZE)分配第一个页的大小给也目录，并对页目录进行初始化0.于是我们天下boot_alloc()函数。对齐将要分配的空间，判断是否月结，然后反悔一个nextfree指向的空间。
- 接下来，在虚拟内存UVPT位置设置一个系统页目录的映射，指向的就是页目录本身的屋里地址。开放炜用户只读。在 inc/memlayout.c中，我们可以看到UVPT对应的是0xef0000到0xefc0000的PTSIZE的空间，但是其实只用到了PGSIZE的大小。
- 再接下来，就是对整个页表的空间分配和页面初始化。
- 页面初始化其实就是把所有可分配的屋里地址链成一个链表，page_free_list以供之后的页面分配。正如hint里写到的，这里不能分配的有：
  - 页目录本身，也就是pages[0],这个页已经拿去放置页目录。
  - [IOPHYSMEM¡EXTPHYSMEM)的IO hole,这个原来被用来放置BIOS，VGA，RAM。但是到了32位系统后，BIOS被换到了屋里内存的最顶上的地方。
  - [EXTPHYSMEM,boot_alloc(0))的空间，这里放置了Kernel，以及紧随其后分配给也表的空间。

Figure 1: kern/pmap.c : boot_alloc

```
81 boot_alloc(uint32_t n)
82 {
83     static char *nextfree;  // virtual address of next byte of free memory
84     char *result;
85
86     // Initialize nextfree if this is the first time.
87     // 'end' is a magic symbol automatically generated by the linker,
88     // which points to the end of the kernel's bss segment:
89     // the first virtual address that the linker did *not* assign
90     // to any kernel code or global variables.
91     if (!nextfree) {
92         extern char end[];
93         nextfree = ROUNDUP((char *) end, PGSIZE);
94     }
95
96     // Allocate a chunk large enough to hold 'n' bytes, then update
97     // nextfree.  Make sure nextfree is kept aligned
98     // to a multiple of PGSIZE.
99     //
100    // LAB 2: Your code here.
101
102    if (n == 0) return nextfree;
103    result = nextfree;
104    nextfree += n;
105    nextfree = ROUNDUP( (char *) nextfree,PGSIZE);
106
107    return result;
108
109    return NULL;
110 }
```

Figure 2: kern/pmap.c : mem_init

```
138     //////////////////////////////////////////////////////////////////////
139     // Recursively insert PD in itself as a page table, to form
140     // a virtual page table at virtual address UVPT.
141     // (For now, you don't have understand the greater purpose of the
142     // following line.)
143
144     // Permissions: kernel R, user R
145     kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
146
147     //////////////////////////////////////////////////////////////////////
148     // Allocate an array of npages 'struct PageInfo's and store it in 'pages'.
149     // The kernel uses this array to keep track of physical pages: for
150     // each physical page, there is a corresponding struct PageInfo in this
151     // array.  'npages' is the number of physical pages in memory.
152     // Your code goes here:
153     pages = (struct PageInfo *) boot_alloc(sizeof(struct PageInfo) * npages);
154
```

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 paging 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.

通过exercise所给的资料，chapter5 讲了线性地址的格式，以及线性地址到物理地址 的转化。这与我们构建的二级也表结构是相对应的：

Figure 3: kern/pmap.c : page_init

```
244 page_init(void)
245 {
246     // The example code here marks all physical pages as free.
247     // However this is not truly the case.  What memory is free?
248     //  1) Mark physical page 0 as in use.
249     //     This way we preserve the real-mode IDT and BIOS structures
250     //     in case we ever need them.  (Currently we don't, but...)
251     //  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
252     //     is free.
253     //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
254     //     never be allocated.
255     //  4) Then extended memory [EXTPHYSMEM, ...).
256     //     Some of it is in use, some is free. Where is the kernel
257     //     in physical memory?  Which pages are already in use for
258     //     page tables and other data structures?
259     //
260     // Change the code to reflect this.
261     // NB: DO NOT actually touch the physical memory corresponding to
262     // free pages!
263     size_t i;
264     for (i = 1; i< npages_basemem; i++){
265         pages[i].pp_ref = 0;
266         pages[i].pp_link = page_free_list;
267         page_free_list = &pages[i];
268     }
269     // some are used! so we start from unalloc space. why ZhangChi's report ignore this???
270     size_t fs = ((int) boot_alloc(0) - KERNBASE) / PGSIZE;
271     for (i = fs; i < npages; i++) {
272         pages[i].pp_ref = 0;
273         pages[i].pp_link = page_free_list;
274         page_free_list = &pages[i];
275     }
276 }
```

Figure 4: kern/pmap.c : page_alloc

```
288 page_alloc(int alloc_flags)
289 {
290     // Fill this function in
291     struct PageInfo * retPageInfo = page_free_list;
292     if (page_free_list != NULL){
293         page_free_list = page_free_list->pp_link;
294         if (alloc_flags & ALLOC_ZERO){
295             memset(page2kva(retPageInfo),'\0',PGSIZE);
296         }
297     }
298
299     return retPageInfo;
300 }
```

Figure 5: kern/pmamp.c : page_free

```
307 page_free(struct PageInfo *pp)
308 {
309     // Fill this function in
310     pp->pp_link = page_free_list;
311     page_free_list = pp;
312 }
```

Figure 5-8. Format of a Linear Address

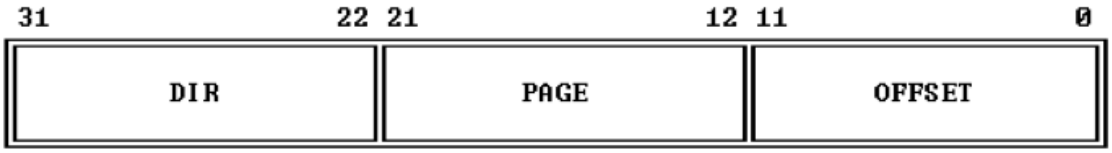
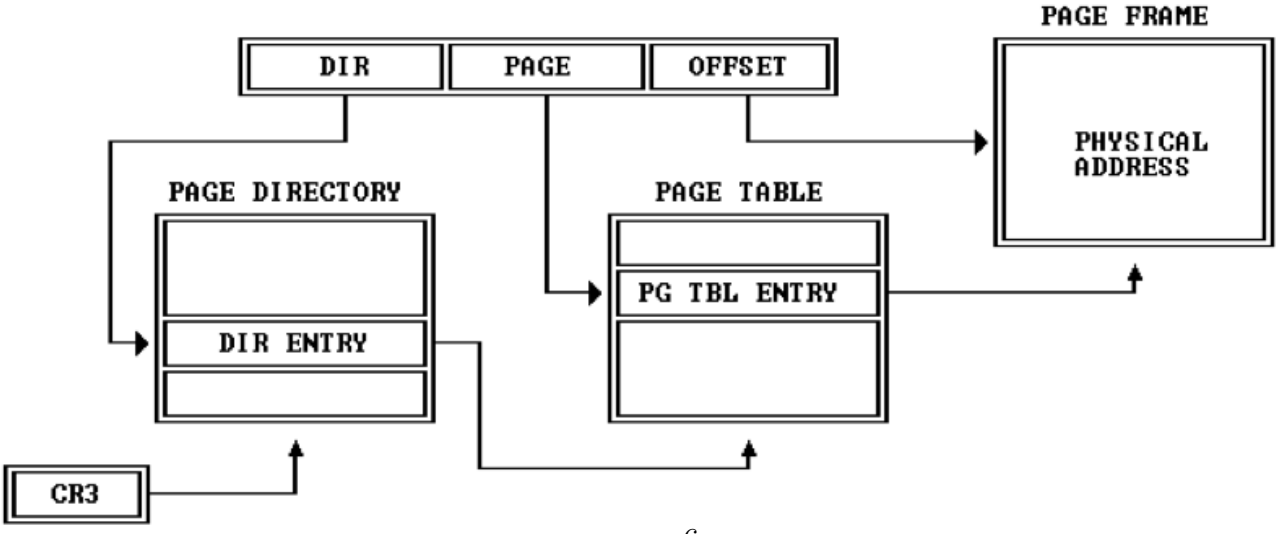


Figure 5-9. Page Translation



2.1 Virtual, Linear, and Physical Address

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 memory are mapped and with what permissions.

这个就是介绍了Qemu的连个命令，作为对我们设置的物理内存和虚拟内存之间的对应关系做一个确认。

Question

1. Assuming that the following JOS kernel code is correct, what type should variable `x` have, `uintptr_t` or `physaddr_t`?

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

答：应该是uintptr_t 因为value是一个指针，可以被解引用，而在JOS中，解引用默认按照虚拟地址来解释，如果，这是一个物理地址，解引用会被当做虚拟地址的。这样子不可能得到正确的值

2.2 Reference counting

这部分主要描述了一下JOS里面简单的内存惯例几只，我们用一个struct pageInfo来存储对于一个页的信息。

- pp_link为一个指针，指向下一个空闲的页，当前页在被使用时，这个指针将没有任何作用，只有当前页不被使用时，挂在page_free_list底下时才有作用。
- pp_ref为一个整数，记录了当前页被引用的次数。当pp_ref = 0时，该页将被释放，挂回page_free_list

2.3 Page Table Management

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.

接下来就是继续写几个函数。

- pgdir_walk()对于给定的页目录pgdir,虚拟地址va,找到va所指向的页的page table entry,如果带了create = true 的参数，那么在找不到页表时，我们将调用page_alloc分配一个新的页给所查找的也表。
- 在查找一个页的时候，我们将经过页目录和页表的双重检查，因此在第一级目录的时候可以方框权限
- boot_map_region()则对[va,va+size) 到 [pa,pa+size)进行银蛇，把虚拟内存到屋里内存的映射保存在也表中。
- page_insert()把一个屋里页pp映射到内存页va,并加上权限 perm — PTE_P.这里需要考虑一个特殊情况，即重复插入同一个物理页。我没有进行特殊处理，只是将之前的页表删除掉，然后再重新添加。
- page_lookup 和pgdir_walk有点像
- page_remove()把一个虚拟地址上的页面移出也表，注意这个屋里页的引用只是减一，并不一定会减到零。

Figure 6: kern/pmap.c : pgdir_walk

```
348 pgdir_walk(pde_t *pgdir, const void *va, int create)
349 {
350     // Fill this function in
351     pte_t * pt_addr_v;
352     struct PageInfo *pg;
353 
354     if (pgdir[PDX(va)] & PTE_P){
355         pt_addr_v = (pte_t *) KADDR(PTE_ADDR(pgdir[PDX(va)]));
356         return &pt_addr_v[PTX(va)];
357     } else {
358         if (create == 0){
359             return NULL;
360         } else {
361             if ((pg = page_alloc(1)) == NULL){
362                 return NULL;
363             } else {
364                 pg->pp_ref = 1;
365                 memset(KADDR(page2pa(pg)),0,PGSIZE);
366                 pgdir[PDX(va)] = page2pa(pg);
367                 pgdir[PDX(va)] = pgdir[PDX(va)] | PTE_U | PTE_W | PTE_P;
368                 pt_addr_v = (pte_t *) KADDR(PTE_ADDR(pgdir[PDX(va)]));
369 
370                 return &pt_addr_v[PTX(va)];
371 
372             }
373         }
374     }
375 
376     return NULL;
377 }
```

Figure 7: kern/pmap.c : mem_init

```
171     //////////////////////////////////////////////////////////////////////
172     // Map 'pages' read-only by the user at linear address UPAGES
173     // Permissions:
174     //    - the new image at UPAGES -- kernel R, user R
175     //      (ie. perm = PTE_U | PTE_P)
176     //    - pages itself -- kernel RW, user NONE
177     // Your code goes here:
178     n = ROUNDUP( npages * sizeof(struct PageInfo), PGSIZE);
179     boot_map_region(kern_pgdir,UPAGES,n,PADDR(pages),PTE_P | PTE_U);
180
181     //////////////////////////////////////////////////////////////////////
182     // Use the physical memory that 'bootstack' refers to as the kernel
183     // stack.  The kernel stack grows down from virtual address KSTACKTOP.
184     // We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
185     // to be the kernel stack, but break this into two pieces:
186     //     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
187     //     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
188     //       the kernel overflows its stack, it will fault rather than
189     //       overwrite memory.  Known as a "guard page".
190     //     Permissions: kernel RW, user NONE
191     // Your code goes here:
192
193     boot_map_region(kern_pgdir,KSTACKTOP-KSTKSIZE,KSTKSIZE,PADDR(bootstack),PTE_W | PTE_P);
194
195     //////////////////////////////////////////////////////////////////////
196     // Map all of physical memory at KERNBASE.
197     // Ie.  the VA range [KERNBASE, 2^32) should map to
198     //      the PA range [0, 2^32 - KERNBASE)
199     // We might not have 2^32 - KERNBASE bytes of physical memory, but
200     // we just set up the mapping anyway.
201     // Permissions: kernel RW, user NONE
202     // Your code goes here:
203
204     boot_map_region(kern_pgdir,KERNBASE,0x10000000,0,PTE_W | PTE_P);
```

Figure 8: kern/pmap.c : boot_map_region

```
389 static void
390 boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
391 {
392     // Fill this function in
393     unsigned int i;
394     pte_t * pg;
395     size = ROUNDUP(size,PGSIZE);
396     for (i = 0;i<size;i += PGSIZE){
397         pg = pgdir_walk(pgdir,(void *) (va + i) ,1);
398         *pg = (pa + i) | perm | PTE_P;
399 
400     }
401 }
```

Figure 9: kern/pmap.c : page_lookup

```
463 struct PageInfo *
464 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
465 {
466     // Fill this function in
467     pte_t * pg;
468     pg = pgdir_walk(pgdir,va,0);
469 
470     if (pg == NULL){
471         return NULL;
472     }
473     if (pte_store != NULL){
474         *pte_store = pg;
475     }
476     return pa2page(*pg);
477 }
```

Figure 10: kern/pmamp.c : page_remove

```
493 page_remove(pde_t *pgdir, void *va)
494 {
495     // Fill this function in
496     struct PageInfo *pg;
497     pte_t *p_pte;
498     pg = page_lookup(pgdir,va,&p_pte);
499     if (pg == NULL){
500         return;
501     } else {
502         page_decref(pg);
503     }
504     if (p_pte != NULL){
505         *p_pte = 0;
506     }
507
508     tlb_invalidate(pgdir,va);
509
510 }
```

Figure 11: kern/pmamp.c : page_insert

```
429 page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
430 {
431     // Fill this function in
432     pte_t *pte;
433     pte = pgdir_walk(pgdir,va,1);
434     if (pte == NULL){
435         return -E_NO_MEM;
436     } else {
437         pp->pp_ref ++;
438
439         if ( (*pte) & (PTE_P) ) {
440             page_remove(pgdir,va);
441         }
442
443         *pte = page2pa(pp) | PTE_P | perm;
444     }
445
446     tlb_invalidate(pgdir,va);
447     return 0;
448 }
```

3 Kernel Address Space

3.1 Permissions and Fault Isolation

3.2 Initializing the Kernel Address Space

> **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.

完成mem_init() 接下来的部分具体看Figure 2 & 7

- 把pages映射到UPAGES处
- Kernel Stack的映射 boostack
- 映射[KERNBASE,2$\hat{}$32)的部分。这部分是256M

下图就是结果了。

```
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
running JOS: (1.1s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

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]

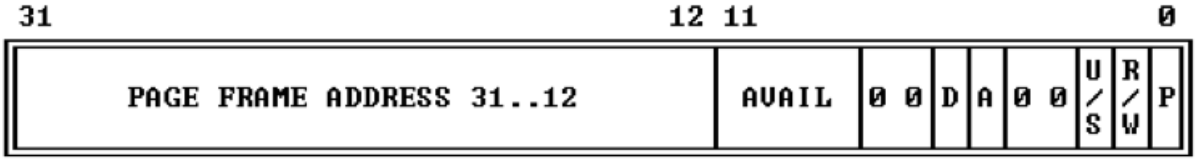
3. (From Lecture 3) We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?
4. What is the maximum amount of physical memory that this operating system can support? 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?

答：

2. 这边对应exercise 5的三块代码。

3. 因为有保护位，PTE_W,PTE_U两个bit来表示权限，当用户程序试图去访问Kernel的时候会因为权限不够而报错。详细的定义见下图

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.

4. UPAGES存放的是系统pages的空间，系统分配的虚拟空间是PTSIZE,也就是4MB.

   每个PageInfo,8Byte.一个PageInfo对应4KB 的空间，所以从空就是4MB/8Byte*4KB = 2G.

5. pgdir一个页面加上，二级目录的1024页面。所以总共是（1+1024）*4K 的空间占用

   如果说lab最多使用256MB 那么即使 （64+1）*4KB = 260KB。

6. 如下图所示，60 62开启了KERNBASE的映射，而eip还用的是低位置的，在68行的jmp指令之后才跳到高位置上去。

```
59      # Turn on paging.
60      movl    %cr0, %eax
61      orl $(CR0_PE|CR0_PG|CR0_WP), %eax
62      movl    %eax, %cr0
63
64      # Now paging is enabled, but we're still running at a low EIP
65      # (why is this okay?).  Jump up above KERNBASE before entering
66      # C code.
67      mov $relocated, %eax
68      jmp *%eax
```

   查看kern/entrypgdir.c 中的27 28定义的entry_pgdir可以看到，不仅错了[KERNBASE,KERNBASE+4MB)到映射，也做了[0,4MB)到[0,4MB)的映射，差别就是低位置的映射没有开启写的权限。

```
20 __attribute__((__aligned__(PGSIZE)))
21 pde_t entry_pgdir[NPDENTRIES] = {
22     // Map VA's [0, 4MB) to PA's [0, 4MB)
23     [0]
24         = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,
25     // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
26     [KERNBASE>>PDXSHIFT]
27         = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W
28 };
29
```