JOS-Lab-2 实验报告

熊伟伦
5120379076
azardf4yy@gmail.com

2014年10月11日 - 10月15日

Contents

1 前言

该报告描述了我在lab2实验的过程中遇到的问题与解决的方法，介绍了lab2的整体结构。指导中问题的解答参考上传的压缩包中的answers-lab2.txt文件。

2 概括

这个实验需要实现kern/pmap.c中的关于内存管理的函数，包括物理页的分配，虚拟内存的管理。所有需要完成的代码均在kern/pmap.c文件中。需要参考inc/x86.h，inc/mmu.h，inc/string.h，inc/queue.h，inc/types.h，inc/memlayout.h，kern/pmap.h中的相关常量，宏，函数。

尤其是inc/mmu.h，kern/pmap.h，分别包含了页表定义的大部分常量，和页表指针，物理地址，虚拟地址的转换函数，宏。

3 物理页管理

这一部分根据Exercise的要求，需要实现页表结构在实际物理内存中的管理，所以暂时不需要关注虚拟地址，线性地址。

3.1 创建页表结构

与这一部分相关的源代码还有inc/memlayout.h，在其中定义了存储物理页结构的关键数据结构-Page：

inc/memlayout.h

```
164 struct Page {
165     struct Page *pp_link;
166     uint16_t pp_ref;
167 };
```

我把注释去掉了。根据注释，这是一个单项链表结构，其中pp_link指向链表的下一个单元的地址，pp_ref则表示页表结构中映射到该段物理页的entry数量。

在Lab1中，内核被载入到了0x00010000（1MB）的位置处，程序使用了一个指针end来标记内核数据在内存中最后的位置。因此，我们的物理页表目录和页表存在end标记后按4096byte对齐的位置。

下面的常量是整个页表结构的关键。

kern/pmap.c

```
13 // These variables are set by i386_detect_memory()
14 size_t npages;
```

```
15 static size_t npages_basemem;
16
17 // These variables are set in mem_init()
18 pde_t *kern_pgdir;
19 struct Page *pages;
20 static struct Page *page_free_list;
21 static struct Page chunck_list;
```

其中npages和npages_basemem保存的是整个系统能够使用的总页数和其中的基础内存页数。他们通过调用i386_detect_memory()函数得到数值，可以把他们看做常量。

根据输出可知npages=16639，npages_basemem=160，对应的内存空间分别为66556K和640K。两者相减就是扩展内存部分的数值了。

接着看代码，kern_pgdir保存了页表目录的入口，是虚拟地址，将其减去0xF0000000就是存入cr3的页表目录的物理地址了。pages则是目录索引的页表的入口，page_free_list是未分配的页表项的合集，用链表存储，chunck_list书在连续页表回收的地方用到的。

kern/pmap.c

```
142 kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
143 memset(kern_pgdir, 0, PGSIZE)
```

可知为kern_pgdir分配了一个页大小的空间，也就是4096byte，同时进行了初始化，期中memset的第二个参数的格式是int，所以直接传0即可。

接下来实现boot_alloc()

kern/pmap.c

```
84  static void *
85  boot_alloc(uint32_t n)
86  {
87      static char *nextfree;
88      char *result;
89      if (!nextfree) {
90          extern char end[];
91          nextfree = ROUNDUP((char *) end, PGSIZE);
92      }
93      nextfree = ROUNDUP(nextfree, PGSIZE);
94      result = KADDR(PADDR(nextfree));
95      if (n > 0) {
96          nextfree += n;
97          // PADDR will check if VA >= KERNBASE
98          // KADDR will check if PA < npages*PGSIZE
99          KADDR(PADDR(nextfree));
100         nextfree = ROUNDUP(nextfree, PGSIZE);
101     }
102     return result;
103 }
```

nextfree是一个全局变量，保存了接下来可以alloc的空间，当然，整个函数只用在物理地址层面上的内存管理。函数会对其到下一个页大小的可用的地址，并将地址返回。需要注意的是注释中需要判断nextfree是否在合法的虚拟地址空间上，在宏PADDR和KADDR中会做该检测，因此第99行就实现了这个检测功能。

回到mem_init()函数，程序接下来还会alloc页表的空间：

kern/pmap.c

```
160 size_t page_size = sizeof(struct Page);
161 pages = boot_alloc(npages*page_size);
162 memset(pages, 0, npages*page_size);
```

由于Page包含一个32位的指针和一个uint32_t变量，所以page_size变量大小是8，npages前面得知是16639，所以这一次分配了16639*8byte大小的空间，也就是133112byte，也就是32.498个页大小的空间，所以需要33个完整的页。因此整个页表目录和页表总共占用了34个页大小的空间，我们输出进行验证。

得到物理地址，kern_pgdir=0x0011A000，pages=0x0011B000，下一个可分配的页boot_alloc(0)=0x0013C000。确实符合我们的计算。

接下来进入page_init()函数

3.2 页表结构初始化

kern/pmap.c

```
249 void
250 page_init(void)
251 {
252     uint32_t i;
253     page_free_list = NULL;
254     for (i = 1; i < npages_basemem; i++) {
255         pages[i].pp_ref = 0;
256         pages[i].pp_link = page_free_list;
257         page_free_list = &pages[i];
258     }
259     for (i = PGNUM(PADDR(boot_alloc(0))); i < npages; i++) {
260         pages[i].pp_ref = 0;
261         pages[i].pp_link = page_free_list;
262         page_free_list = &pages[i];
263     }
264 }
```

根据注释，我们将描述整个可用物理内存的页表分隔成两大块加入到page_free_list中，第一个循环加载的是0都640K的物理地址空间，即Low Memory。第二个循环加载的是从页表后面开始的可以用空间，直到最大的可用物理内存。

值得注意的是模仿注释中的样例，我们这种写法链表头指向的是物理地址最高的页表，然后依次降低。我输出了第二个循环的开始的i的值，为316，即从第2页到160页，从第317页到最后一页，都是可以分配的页。开头第一页的Boot，中间的VGA显存，扩展ROMs，BIOS，Kernel，页表目录和页表则不能分配。

接下来还有page_alloc()和page_free()两个函数需要实现。

3.3 单个页表分配和释放

kern/pmap.c

```
298 struct Page *
299 page_alloc(int alloc_flags)
300 {
301     // Fill this function in
302     struct Page* alloc_page = NULL;
303 
304     if (page_free_list) {
305         alloc_page = page_free_list;
306         page_free_list = page_free_list->pp_link;
307         alloc_page->pp_link = NULL;
308         // if ALLOC_ZERO flag true, set all empty
309         if (alloc_flags & ALLOC_ZERO)
310             memset(page2kva(alloc_page), 0, PGSIZE);
311     }
312 
313     return alloc_page;
314 }
```

page_alloc()函数的实现十分简单。就是将page_free_list链表的第一节从链表中剥离出来，并且将其返回。如果alloc_flags要求将分配的页清空，就调用memset()函数，该函数通过page2kva()函数根据页表项的指针返回对应的虚拟地址。在前面说过了，memset的第二个参数是int类型，所以0对应了空字符，清空的长度显然就是一个PGSIZE的大小。

kern/pmap.c

```
435 void
436 page_free(struct Page *pp)
437 {
438     // Fill this function in
439     if (pp->pp_ref == 0) {
440         pp->pp_link = page_free_list;
441         page_free_list = pp;
442     }
443 }
```

page_free()函数如上所示，先判断还有没有虚拟页映射到该物理页，如果没有，则将该物理页加入到page_free_list的链表头。实现之后check_alloc()通过。

3.4 多个页表的分配，释放和重新分配

对应于Exercise2，需要实现page_alloc_npages和page_free_npages两个函数。主要是优化大空间分配，不需要多次调用page_alloc函数。下面是page_alloc_npages函数的代码。

kern/pmap.c

```
435 struct Page *
436 page_alloc_npages(int alloc_flags, int n)
437 {
438     // Fill this function
439 
440     uint32_t i;
441     uint32_t find_flag = 0;
442     uint32_t alloc_zero_flag = alloc_flags & ALLOC_ZERO;
443     struct Page* find_head = page_free_list;
444     struct Page* ahead_find = NULL;
445     struct Page* t_find;
446 
447     if (n <= 0 || page_free_list == NULL)
448         return NULL;
449 
450     // search n cotinuous physical pages
451     while(1) {
452         t_find = find_head;
453         for (i = 0; i < n; i++) {
454             // is physical pages continuous?
455             if ((page2pa(t_find) - page2pa(t_find->pp_link))
456                 != PGSIZE)
457                 break;
458             t_find = t_find->pp_link;
459             find_flag = (i == n-1);
460         }
461         // if find n continuous physical pages
462         if (find_flag)
463             break;
464         ahead_find = find_head;
465         find_head = find_head->pp_link;
466         // if out of free memory, return NULL
467         if (!find_head)
468             return NULL;
469     }
470 
471 
472     // ready to alloc
473     t_find = find_head;
474     for (i = 0; i < n; i++) {
475         if (alloc_zero_flag)
476             memset(page2kva(t_find), 0, PGSIZE);
477         if (i != n-1)
478             t_find = t_find->pp_link;
479     }
480 
481 
```

```
482     // remove from page_free_list
483     // if find_head == page_free_list
484     if (ahead_find == NULL)
485         page_free_list = t_find->pp_link;
486     // find_head != page_free_list
487     else
488         ahead_find->pp_link = t_find->pp_link;
489     t_find->pp_link = NULL;
490
491
492     // reverse, for check_continuous(), list order by physical
493     struct Page* next_find = NULL;
494     struct Page* prev_find = NULL;
495     t_find = find_head;
496     while(t_find) {
497         next_find = t_find->pp_link;
498         t_find->pp_link = prev_find;
499         prev_find = t_find;
500         t_find = next_find;
501     }
502     // return alloc list head
503     return prev_find;
504 }
```

这个函数较为复杂，首先扫描page_free_list，需要找到连续的n个块，要满足这n个块的物理地址连续。我的代码不是十分优化，时间消耗比较大，但为了代码的简单化，满足了功能就好。

接着将这n个页表项从page_free_list中剥离出来，需要判断是否是链表头或者是链表中部，然后重新组织page_free_list链表。但是根据page_free函数，只是简单的将页表项加到链表头，所以会出现明明有连续的物理页表项可以分配，但由于它们在链表中不是连续的，所以不会分配，同样是为了代码的简单化，我只是实现了注释中所描述的逻辑。所以这些不合理都无视掉了。

最后，还需要对剥离出来的一段Page链表进行逆转，因为在check_continuous函数中它判断得到的物理页entry是按链表顺序物理地址逐渐增大的，而在page_alloc中的样例恰恰相反，因此我们得到的n个entry是按物理地址逐渐递减的，所以需要逆转一下链表。

kern/pmap.c

```
1004 static int
1005 check_continuous(struct Page *pp, int num_page)
1006 {
1007     struct Page *tmp;
1008     int i;
1009     for( tmp = pp, i = 0; i < num_page - 1;
1010         tmp = tmp->pp_link, i++ )
1011     {
1012         if(tmp == NULL)
1013         {
```

```
1014                return 0;
1015            }
1016            if( (page2pa(tmp->pp_link) - page2pa(tmp)) != PGSIZE )
1017            {
1018                return 0;
1019            }
1020        }
1021        return 1;
1022    }
```

page_free_npages由于要求比较诡异，所以实现比较简单。注释要求将释放的链表存入chunck_list，但是整个pmap.c没有其他任何地方操作这个链表。我依然就按照注释的要求实现了这个函数。

kern/pmap.c

```
405 int
406 page_free_npages(struct Page *pp, int n)
407 {
408     // Fill this function
409     struct Page* t_chunck;
410
411     // check if continuous physical page
412     if (check_continuous(pp, n) == 0)
413         return -1;
414
415     // add list to chunck list, list order by physical
416     if (chunck_list.pp_link == NULL) {
417         chunck_list.pp_link = pp;
418     } else {
419         t_chunck = chunck_list.pp_link;
420         while(1) {
421             if (t_chunck->pp_link != NULL)
422                 t_chunck = t_chunck->pp_link;
423             else {
424                 t_chunck->pp_link = pp;
425                 break;
426             }
427         }
428     }
429     return 0;
430 }
```

简单的说明下这个函数，首先验证传入的pp和n是否真实的指向一个物理地址连续的页表项，然后将整个链表插入到chunck_list的末尾。

接下来是page_realloc_npages函数，比较复杂。

kern/pmap.c

```
450 struct Page *
451 page_realloc_npages(struct Page *pp, int old_n, int new_n)
452 {
```

```
453     // Fill this function, all free
454     if (new_n == 0) {
455         page_free_npages(pp, old_n);
456         return NULL;
457     }
458 
459     // assume pp_ref will be added by caller
460     // need more physical pages
461     if (old_n < new_n) {
462         uint32_t more_n = new_n - old_n;
463         uint32_t can_add_flag = 1;
464         uint32_t i = 0;
465 
466         // if can add to tail?
467         for (i = old_n; i < new_n; i++){
468             // empty and not over limit
469             if ((pp+i)->pp_ref == 0 && pp+i < pages+npages ) {
470                 can_add_flag = 0;
471                 break;
472             }
473         }
474 
475         // if can't add to tail, free and alloc
476         if (can_add_flag == 0){
477             struct Page* new_alloc;
478             new_alloc = page_alloc_npages(ALLOC_ZERO, new_n);
479             memmove(page2kva(new_alloc), page2kva(pp),
480                     old_n*PGSIZE);
481             page_free_npages(pp, old_n);
482             return new_alloc;
483         }
484 
485         // if can add
486         // first, remove those from page_free_list
487         struct Page* find_pp;
488         while(page_free_list > pp && page_free_list <=
489             pp+more_n) {
490             page_free_list = page_free_list->pp_link;
491         }
492         find_pp = page_free_list;
493         while(find_pp != NULL && find_pp->pp_link != NULL) {
494             if (find_pp->pp_link > pp && find_pp->pp_link <=
495                 pp+more_n) {
496                 find_pp->pp_link = find_pp->pp_link->pp_link;
497             }
498             find_pp = find_pp->pp_link;
499         }
500 
501         // second, add to old alloc page list
502         for (i = 0; i < more_n; i++) {
503             (pp+i)->pp_link = pp+i+1;
504         }
505         (pp+more_n)->pp_link = NULL;
506 
507         // third, init new memory
508         memset(page2kva(pp+1), 0, more_n*PGSIZE);
```

```
509
510         // fourth, return pp
511         return pp;
512
513     } else if (old_n > new_n) {
514         // free last pages
515         page_free_npages(pp+new_n, old_n-new_n);
516         (pp+new_n-1)->pp_link = NULL;
517         return pp;
518
519     } else {
520         // old_n == new_n, do nothing
521         return pp;
522     }
523 }
```

这个函数比较长，在此说明下它的功能。首先454行判断new_n是否为0，如果是就全部free掉，这点练习的说明中并没有写。然后在461行判断是否new_n大于old_n。先说明new_n更小的情况：

如515行所示，直接free掉pp+new_n后的物理页表项，然后将已经分配的链表的最后一块的pp_link置为NULL表明这一块到达了链表末尾，十分简单。如果old_n==new_n，则什么都不做返回pp。

如果new_n比old_n大，则进入462行的代码。首先判断后面直到new_n个的页表项是否都为空，如果不是就直接alloc，memmove，free(lib/string.c中说用memmove取代memcpy)。如果可以直接扩展就先将这些要被扩展的未分配页表项从page_free_list中移除，这一步也分两部分，先判断链表头是否属于需要移除的部分，再判断链表内部是否需要移除。第二步是将扩展的页表项块加入到pp的链表中，第三步初始化扩展的物理页(练习中没有说明这一步，属于我自己判断做的)。最后返回pp地址。

kern/pmap.c

```
172     check_page_free_list(1);
173     check_page_alloc();
174     check_page();
175     check_n_pages();
```

做完这些后可以通过check_n_pages函数，虽然mem_init中有些不合理的是这个测试函数在check_page后，而check_page在后面才会实现，因此我调换了两个测试函数的顺序先测试下多个物理页的操作函数是否正确，也许由于page_realloc_npages函数是助教老师新加的，并没有在grade脚本中进行相应测试，我自行写了些测试判断了正确性。

4 虚拟内存

4.1 地址转换

关于逻辑地址，虚拟地址，线性地址，物理地址的含义和转换，在上课，以及网上很多地方都有很详细的介绍，因此我就不多加描述了。比较值得注意的是，由于遗留原因，x86的页模式实际上也要进过段模式的转化，合成段页模式。就像实模式表示20位的地址一样，只不过这里是32位的逻辑地址且直接表示为虚拟地址。

练习中还说明了一些QEMU可以用于查看虚拟地址物理地址对应关系的操作。总之就是让我们先彻底区分系统中哪些是虚拟地址，哪些是物理地址。

在代码中除了physaddr_t表示的类型其余都是虚拟地址，而跟物理地址有关的函数主要是PADDR，KADDR，page2pa，pa2page。

对于Question1，显然是虚拟地址，因为变量x和value是在程序中创建的。

4.2 引用计数

主要说明了被映射的物理page的pp_ref会增加，超过UTOP的虚拟内存地址地址都是与内核有关，不会被free，除非重启关机。

主要参考inc/memlayout.h中的图。图比较长我就不贴了。再后面的内容中再详细介绍这张图的意义。

4.3 页表管理

这里要求实现5个函数，完成虚拟地址管理。

kern/pmap.c

```
555 pte_t *
556 pgdir_walk(pde_t *pgdir, const void *va, int create)
557 {
558     // Fill this function in
559     struct Page* new_page;
560     pde_t* pde = pgdir + PDX(va);
561     pte_t* pte;
562 
563     // has created
564     if (*pde & PTE_P) {
565         pte = (pte_t*)KADDR(PTE_ADDR(*pde));
566         return pte + PTX(va);
567     }
568 
569     // need create
570     if (create == 0) {
571         return NULL;
572     } else {
```

```
573             new_page = page_alloc(ALLOC_ZERO);
574             if (new_page == NULL) {
575                 return NULL;
576             } else {
577                 new_page->pp_ref++;
578                 *pde = page2pa(new_page) | PTE_P | PTE_W |
579                         PTE_U;
580                 pte = (pte_t*)KADDR(PTE_ADDR(*pde));
581                 return pte + PTX(va);
582             }
583         }
584 }
```

这个函数根据输入的虚拟地址va，返回它在二级页表(即页目录的下一级)对应的entry，也就是它所在的物理页。

根据create参数，如果物理页不存在(564行判断页目录中entry的PTE_P位)，则需要为这个虚拟地址分配物理页。

步骤是先创建一个物理页，然后将物理页对应的PTE传给页表保存并分配权限，包括P存在，W可写，U用户可读。然后将pte+PTX(va)返回给用户，即为对应的物理页表位置。

kern/pmap.c

```
595 static void
596 boot_map_region(pde_t *pgdir, uintptr_t va, size_t size,
597                 physaddr_t pa, int perm)
598 {
599     // Fill this function in
600     uint32_t n = size/PGSIZE;
601     uint32_t i;
602     pte_t* pte;
603     for (i = 0; i < n; i++) {
604         pte = pgdir_walk(pgdir, (void*)va, 1);
605         *pte = pa | perm | PTE_P;
606         va += PGSIZE;
607         pa += PGSIZE;
608     }
609 }
```

将从va开始的size个byte映射到pa，使用pgdir_walk函数实现十分简单。

kern/pmap.c

```
670 struct Page *
671 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
672 {
673     // Fill this function in
674     pte_t* pte = pgdir_walk(pgdir, va, 0);
675
676     if (pte_store)
677         *pte_store = pte;
```

```
678     if ((pte != NULL) && (*pte & PTE_P))
679         return pa2page(PTE_ADDR(*pte));
680     return NULL;
681 }
```

查找给定的虚拟地址va，返回对应的页表的page，如果有pte_store，则将对应的pte的位置保存于此。

kern/pmap.c

```
698 void
699 page_remove(pde_t *pgdir, void *va)
700 {
701     // Fill this function in
702     pte_t* pte;
703     struct Page* pp = page_lookup(pgdir, va, &pte);
704 
705     if (pp != NULL) {
706         *pte = 0;
707         page_decref(pp);
708         tlb_invalidate(pgdir, va);
709     }
710 }
```

page_remove()函数也十分简单，删除给定va对应的页表，注意需要清空TLB对应的项，防止其他指令通过TLB访问已删除的页表。

kern/pmap.c

```
634 int
635 page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
636 {
637     // Fill this function in
638     pte_t *pte = pgdir_walk(pgdir, va, 1);
639 
640     // not exist and can't create
641     if (pte == NULL)
642         return -E_NO_MEM;
643 
644     // if exist
645     if (*pte & PTE_P) {
646         if (PTE_ADDR(*pte) == page2pa(pp)) {
647             tlb_invalidate(pgdir, va);
648             pp->pp_ref--;
649         } else {
650             // page_remove will decrease pp_ref
651             page_remove(pgdir, va);
652         }
653     }
654     *pte = page2pa(pp) | perm | PTE_P;
655     pp->pp_ref++;
656     return 0;
657 }
```

这个函数将虚拟地址va映射到pp对应的页表上，如果没有分配就直接映射va，如果页表已分配先判断是否映射va相同的虚拟地址，如果是就只删除对应的TLB，如果不是就直接remove，两种情况在后面统一重新写入页表。为了一致性，我们在已分配的第一种情况也减减pp_ref为了后面加不需要再立flag进行判断。

完成这些后check_page()函数通过。

5 内核地址空间

根据inc/memlayout.h中的图可知，UTOP以下是用户环境能使用的虚拟内存，ULIM以上是内核才有限权的空间，期间是只有Read限权的空间，保存了内核相关的一系列代码，不可修改。

现在开始填充mem_init函数，

kern/pmap.c

```
187 boot_map_region(kern_pgdir, UPAGES,
188                 ROUNDUP(npages*page_size, PGSIZE),
189                 PADDR(pages), PTE_P | PTE_U);
```

根据注释，这一段将最开始物理内存写入状态的pages映射到UPAGES处，即UPAGES到UVPT存储的是pages。

kern/pmap.c

```
200 boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE,
201                 KSTKSIZE, PADDR(bootstack),
202                 PTE_P | PTE_W);
```

这一段将bootstack映射到了KSTACKTOP的下方，属于内核代码。

kern/pmap.c

```
210 boot_map_region(kern_pgdir, KERNBASE, ~KERNBASE+1,
211                 0, PTE_P | PTE_W);
```

这一段应该就是夏老师上课讲的，虚拟内存上方又将内核代码全部映射了一遍。

至此，根据注释，lab的测试全部通过。对虚拟内存空间的分配有了一个大致的了解，还是有很多细节每太搞明白。

Question2中，答案参考answers-lab2.txt。简单说下。

第2题根据inc/memlayout.h中的图进行填写即可，还有部分需要计算跟程序输出。

第3题题页表有限权位，用户不能修改内核处数据。

第4题根据上面写的UPAGES处的映射可计算，UPAGES到UVPT有一个PTSIZE的大小，一个struct Page是8个byte，总支持空间是PTSIZE/8*PGSIZE=2147483648Byte=2GB。

第5题，根据理想设计一共有1个PGSIZE大小的页目录和1024个页表去支持4GB的内存，开销为4MB空间，就是上面一题分配的大小。而实际上获取的npages为16639，所以只需要17个页表，页目录就只需要17*4个byte，一共是17*4+17*PGSIZE=69700Byte=68.07KB。这里还不包括struct Page*链表维护等空间。

第6题，在kern/entry.S的67行之后EIP基于kernbase，因为kern/entrypgdir中同时将两块虚拟都内都映射到了同一个物理内存空间(这个lab1好像回答过了)

6 总结

至此这个lab算写完了，花的时间远远少于lab1，可能主要原因是lab1需要了解整个系统的结构，第一次看JOS的代码，而且还有大量的汇编。而这个lab专注于虚拟内存，物理内存部分，知识点较lab1更少，但对于虚拟内存空间的分配，我依然有很多不解以及觉得不合理的地方。