Jos Lab 2: Memory Management Lab Report

王勇杰 5120379068

2014.10.27

注：**lab** 简介以及 **questions** 都在此文档中，可通过目录直接跳转。

Contexts

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.

A. boot_alloc()函数

```
104     // LAB 2: Your code here.
105     if (n==0)   return nextfree;
106     if (n>0)
107     {
108         uint32_t roundup_n = ROUNDUP(n, PGSIZE);
109         if (PADDR(nextfree + roundup_n) > (npages * PGSIZE))
110         {
111             panic("error in boot_alloc: out of memory\n");
112             return NULL;
113         }
114         char *tem = nextfree;
115         nextfree += roundup_n;
116         return tem;
117     }
118
119     return NULL;
120 }
```

这一个函数主要是 JOS 载入虚拟内存时调用的。在代码中第 108 行将需要申请的内存空间进行 PGSIZE（4K）对齐。在 109 行由于调用此函数时地址空间已经为虚拟地址，因此要调用 PADDR 将其转化为物理地址，也就是需要的最大物理地址空间。此时如果超过了物理内存的最大地址空间，则报错并返回 NULL。如果在物理内存空间内，则将 nextfree 向后推移 size 个单位，并将原先的 nextfree 返回。

B. mem_init()

```
162     // Your code goes here:
163     pages = boot_alloc(npages * sizeof(struct Page));
```

这段代码为在使用页表前，使用 boot_alloc 函数开设存储所有页表的空间。

C. page_init()

```
267     size_t i;
268     pages[0].pp_ref = 1;
269
270     page_free_list = NULL;
271     for (i = 1; i < npages_basemem; i++)
272     {
273         pages[i].pp_ref = 0;
274         pages[i].pp_link = page_free_list;
275         page_free_list = &pages[i];
276     }
277
278     for (i = IOPHYSMEM/PGSIZE; i < EXTPHYSMEM/PGSIZE; ++i)
279         pages[i].pp_ref = 1;
280
281     for (; i < PADDR(boot_alloc(0))/PGSIZE; ++i)
282         pages[i].pp_ref = 1
283     for (; i < npages; ++i)
284     {
285         pages[i].pp_ref = 0;
286         pages[i].pp_link = page_free_list;
287         page_free_list = &pages[i];
288     }
```

此函数代码根据注释由 4 部分组成。

1、 在 268 行将 page 0 标记为已使用。
2、 270 行初始化 page_free_list（便于标记 freelist 的结尾），而后 271-276 行代码将第 1 个到第 npages_basemem 个 page 进行初始化，并加入 page_free_list 链表
3、 278-279 行代码将 IO 部分的内存空间标记为已使用
4、 281-282 行代码先将内核在内存内的使用的空间标记为已使用（使用 boot_alloc(0)来获取内核分配的物理地址空间中下一个空闲地址），然后 283-288 行代码再将接下来所有的 pages 初始化并加入 page_free_list。

最后形成的链表如下：（page[]变量中由 pp_link 指向下一个）

page_free_list→page[npages-1]→page[npages-2]→.....(去除 IO 及内核所用的 pages)→page[2] →page[1] →NULL

D. page_alloc()

```
301  struct Page *
302  page_alloc(int alloc_flags)
303  {
304      // Fill this function in
305
306      if (page_free_list == NULL)
307          return NULL;
308
309      Page *tem = page_free_list;
310      page_free_list = page_free_list->pp_link;
311      tem->pp_link = NULL;
312      if (alloc_flags & ALLOC_ZERO)
313          memset(page2kva(tem),'\0',PGSIZE);
314      return tem;
315
316  }
```

这个函数实现的是分配一块物理地址。首先在 306-307 行判断内存是否使用完了，如果是返回 NULL。如果没用完，则取出 page_free_list 中的第一块，在按照 alloc_flag 的要求初始化后返回。

E. page_free()

```
355  void
356  page_free(struct Page *pp)
357  {
358      pp->pp_link = page_free_list;
359      page_free_list = pp;
360      // Fill this function in
361  }
```

这个函数是释放一个 page。由于 pp->pp_ref 已经为 0 了，因此只需将 pp 加入 page_free_list 即可。

在完成上述函数后，运行 JOS，进行结果如下：

```
warning! The value %n argument pointed to has been overflowed!
chnum1: -1
Physical memory: 66556K available, base = 640K, extended = 655
32K
check_page_alloc() succeeded!
kernel panic at kern/pmap.c:759: assertion failed: page_insert
(kern_pgdir, pp1, 0x0, PTE_W) < 0
```

Exercise 2

In the file kern/pmap.c, you must implement code for the following functions.

```
page_alloc_npages(int alloc_flags, int n)
page_free_npages(struct Page *pp, int n)
```

check_n_pages(), called from mem_init(), tests your page table management routines. You should make sure it reports success before proceeding.

在这个练习中需要实现分配连续的地址空间，因此上个练习中的 page_free 函数需要进行修改，在释放一个 page 后 page_free_list 队列中的页仍然需要按照顺序排列，这样的话在释放 page 时就需要将其插入到 page_free_list 中切当的地方。修改后的 page_free 代码如下：

```
355  void
356  page_free(struct Page *pp)
357  {
358      // Fill this function in
359      struct Page *tem = page_free_list;
360      if (pp > tem)
361      {
362          pp->pp_link = page_free_list;
363          page_free_list = pp;
364      }
365      else
366      {
367          while (pp < (tem->pp_link))
368              tem = tem->pp_link;
369          pp->pp_link = tem->pp_link;
370          tem->pp_link = pp;
371      }
372  }
```

由于在 page_free_list 中是大地址指向小地址，因此插入 pp 是也要按照这一顺序插入。

那么现在在做 page_alloc_npages 时，只需检索 page_free_list 中是否有连续的空间即可。

以下即为 page_alloc_npages 的代码，后面将会详细说明这个函数的实现思想。

```
331 struct Page *
332 page_alloc_npages(int alloc_flags, int n)
333 {
334     // Fill this function
335     int i;
336     if (n <= 0) return NULL;
337     struct Page* pfl = page_free_list;
338     struct Page* last_pfl = NULL;
339     while (pfl != NULL)
340     {
341         struct Page* tem = pfl;
342         int flag = 0;
343         for (i=0; i<n-1; ++i)
344         {
345             if (tem->pp_link == NULL) {flag = 1; break;}
346             if ((tem - tem->pp_link) != 1) {flag = 1; break;}
347             tem = tem->pp_link;
348         }
349 
350         if (flag == 1)
351         {
352             last_pfl = tem;
353             pfl = tem->pp_link;
354         }
355         else
356         {
357             //find n pages
358             pfl = tem;
359             if (last_pfl == NULL)
360                 page_free_list = tem->pp_link;
361             else
362                 last_pfl->pp_link = tem->pp_link;
363 
364             for (i=0; i<n-1; ++i)
365             {
366                 tem->pp_link = tem + 1;
367                 tem = tem + 1;
368             }
369             tem->pp_link = NULL;
370             tem = pfl;
371             if (alloc_flags & ALLOC_ZERO)
372                 for (i=0;i<n;++i)
373                 {
374                     memset(page2kva(tem),'\0',PGSIZE);
375                     tem = tem->pp_link;
376                 }
377             return pfl;
378         }
379 
380     }
381     return NULL;
382 }
```

在 339 行大循环即为寻找连续 n 块内存，如果没找到就会跳至 381 行返回 NULL。

343-348 为判断是否有当前 pfl 指向的是否为连续的 n 块空闲内存，如果不是会把 flag 置为 1。

352-353 为 pfl 指向的不是连续空间，此时更新 pfl 并退回大循环开头向后再次寻找。

358-377 为已经找到连续的 n 块，此时 pfl 指向的为最高地址的一块，这一部分将对一些地址空间进行修改。359-362 将得到的 n 块地址从 page_free_list 中移除。364-368 行将 n 块内存链接由高地址指向低地址改为低地址指向高地址，并在 369 行将最后一个页的 pp_link 置为 NULL。371-376 则根据要求初始化所选的页。

```
389  int
390  page_free_npages(struct Page *pp, int n)
391  {
392      // Fill this function
393      if (check_continuous(pp,n) == 0) return -1;
394      struct Page* tem = pp;
395      struct Page* last_tem = NULL;
396      int i;
397      for (i=0; i<n; ++i)
398      {
399          last_tem = tem->pp_link;
400          page_free(tem);
401          tem = last_tem;
402      }
403      return 0;
404  }
```

在 page_free_npages 中，根据注释需要使用 chunck_list，但是我没发现 chunck_list 有任何用处。在 lab 介绍中说可以使用 chunck_list 存储连续的页，但是在这个练习中我不知道它是存储多长连续的页，假如我先分配了连续 4 页，然后释放，再分配 5 页，此时 chunck_list 并没有用处，反而可能过多地消耗内存空间。所以在这个函数中我就把需要释放的每一个页表都通过调用修改过的 page_free，这样在释放后连续的页在 page_free_list 内也是连续的。

而后重新编译运行 JOS，得到如下结果。

```
warning! The value %n argument pointed to has been overflowed!
chnum1: -1
Physical memory: 66556K available, base = 640K, extended = 655
32K
check_page_alloc() succeeded!
check_n_pages() succeeded!
kernel panic at kern/pmap.c:824: assertion failed: page_insert
(kern_pgdir, pp1, 0x0, PTE_W) < 0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Exercise 2(2)

You may know realloc() in C program. Sometimes, applications want to shrink and enlarge the memory they alrealy allocated. A simple way is to allocate a new one and use memcpy() to fill it with data, then free the old one. However, allocator can provide a fast way to support this function. Please implement the following function which changes the allocated size from old_n pages to new_n pages.

```
page_realloc_npages(struct Page *pp, int old_n, int new_n);
```

这个练习是完成 realloc 函数，这个函数共分成 4 个部分来实现，具体如下：

```
433  struct Page *
434  page_realloc_npages(struct Page *pp, int old_n, int new_n)
435  {
436      // Fill this function
437      struct Page *tem = pp;
438      int i;
439      if (old_n == new_n) return pp;
440      if (old_n > new_n)
441      {
442          for (i=0; i<new_n; ++i) tem = tem->pp_link;
443          page_free_npages(tem, old_n - new_n);
444          return pp;
445      }
446      for (i=0; i<old_n-1; ++i) tem = tem->pp_link;
447      struct Page *target = tem + new_n - old_n;
448      struct Page *pfl = page_free_list;
449      struct Page *last_pfl = NULL;
```

第一部分为 439 行，如果新旧大小一致则直接返回

第二部分为 440-445，如果大小减小，则直接释放后面多余的空间。

第三部分为 446-481，如果大小变大则会现在这一部分检测原先地址的后面内存空间是否为空。首先在 456-461 行扫描 free_page_list，查看是否在原先地址最后一块的后面有连续的(new_n-old_n)块空余空间，如果没有则会进入第四部分，如果有则在 462-477 行对这块空间进行连接和初始化。初始化过程中，464-467 行将要分配的空间从 free_page_list 中删除，469-475 行将这些空间按地址从小到大连接，并接在原先 pp 指向的 old_n 块空间后面。

第四部分为 483-486 行，这一部分说明原空间后面已经没有足够的连续空闲内存空间，因此会重新分配 new_n 的内存，而后将原空间内的内容复制到新的内存空间内，再把原内存空间 free 掉。其中描述中提到使用 memcpy 来拷贝内存内的数据，但是在内核中并没有这个函数，查看代码在 string.h 文件中找到如下的注释，因此并不存在 memcpy 的函数，而改用了 memmove 这个函数。

```
17  /* no memcpy - use memmove instead */
18  void *  memmove(void *dst, const void *src, size_t len);
```

```
450     while (pfl > pp)
451     {
452         if (pfl == target)
453         {
454             struct Page* temp = pfl;
455             int flag = 0;
456             for (i=0; i<(new_n-old_n-1); ++i)
457             {
458                 if (temp->pp_link == NULL) {flag = 1; break;}
459                 if ((temp - temp->pp_link) != 1) {flag = 1; break;}
460                 temp = temp->pp_link;
461             }
462             if (flag == 0)
463             {
464                 if (page_free_list == target)
465                     page_free_list = temp->pp_link;
466                 else
467                     last_pfl->pp_link = temp->pp_link;
468 
469                 for (i=0; i<(new_n-old_n-1); ++i)
470                 {
471                     temp->pp_link = temp + 1;
472                     temp = temp + 1;
473                 }
474                 tem->pp_link = pfl;
475                 temp->pp_link = NULL;
476                 return pp;
477             }
478         }
479         last_pfl = pfl;
480         pfl = pfl->pp_link;
481     }
483     struct Page *new_list = page_alloc_npages(ALLOC_ZERO, new_n);
484     memmove(page2kva(new_list),page2kva(pp),old_n*PGSIZE);
485     page_free_npages(pp,old_n);
486     return new_list;
```

2. Virtual Memory

Exercise 3

主要为了解分页地址转换和页保护相关的内容。

2.1 Virtual, Linear, and Physical Addresses

Exercise 4

这一练习为了解 qemu 中 gdb 的一些操作。

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;
```

由于这段代码实在 kernel 中运行，而运行到 kernel 时使用的已经为虚拟地址，因此我认为 x 应为 uintptr_t 类型。

2.2 Reference counting

这里讲到了内存页的应用计数，当计数为 0 时会释放该页。而当使用 page_alloc 时由于此函数不会对引用计数+1，因此在调用后要自行修改应用计数。

2.3 Page Table Management

Exercise 5

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.

A. pgdir_walk()

```
524  pte_t *
525  pgdir_walk(pde_t *pgdir, const void *va, int create)
526  {
527      // Fill this function in
528      pde_t *pde = pgdir + PDX(va);
529      if (PTE_P & *pde)
530          return (pte_t *) (KADDR( PTE_ADDR(*pde) )) + PTX(va);
531      if (create == 1)
532      {
533          struct Page *pp = page_alloc(ALLOC_ZERO);
534          if(pp == NULL) return NULL;
535          pp->pp_ref += 1;
536          *pde = page2pa(pp) | PTE_U | PTE_W | PTE_P;
537          return (pte_t *) (KADDR( PTE_ADDR(*pde) )) + PTX(va);;
538      }
539      return NULL;
540  }
```

这个函数为给定一个虚拟地址 va，找到相应的物理地址。首先在 528 算出二级页表的对应项，并在 529 行判断其最后一位的有效性。如果存在说明已经有了这项页表，那么直接算出物理地址返回即可。否则在 531 行看是否允许创建页表，并在 533-536 行创建。

B. boot_map_region()

```
552  static void
553  boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa
554  {
555      // Fill this function in
556      pte_t *pte;
557      size_t i;
558      for(i = 0; i < size; i += PGSIZE)
559      {
560          pte = pgdir_walk(pgdir, (void *)(va+i), 1);
561          *pte = PTE_ADDR(pa+i) | perm | PTE_P;
562      }
563  }
```

这个函数是将连续的虚拟内存映射到连续的物理内存，将每一个 PGSIZE 分别一一映射即可。

C. page_lookup()

```
621  struct Page *
622  page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
623  {
624      // Fill this function in
625      pte_t *pte = pgdir_walk(pgdir, va, 0);
626      if (pte == NULL) return NULL;
627      if (pte_store != 0) *pte_store = pte;
628      if (PTE_P & (*pte)) return pa2page(PTE_ADDR(*pte));
629      return NULL;
630  }
```

这个函数为查找一个虚拟地址对应的 Page，这个函数使用 pgdir_walk 就可以实现。626 行判断是否有虚拟地址对应的表项，627 行判断是否要储存 PTE，628 行判断该二级页表项是否有效，如果有效就返回页地址。

D. page_remove()

```
647  void
648  page_remove(pde_t *pgdir, void *va)
649  {
650      // Fill this function in
651      pte_t* pte;
652      struct Page* pp = page_lookup(pgdir, va, &pte);
653      if (pp != NULL)
654      {
655          *pte = 0;
656          page_decref(pp);
657          tlb_invalidate(pgdir, va);
658      }
659  }
```

这个函数为删除一个页，首先通过 page_lookup 找到相对应页，然后 655 行将指向该页的页表项删除，656 减小页表的引用计数，657 在 TLB 中置为 invalidate。

E. page_insert()

```
589  int
590  page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
591  {
592      // Fill this function in
593      pte_t *pte = pgdir_walk(pgdir, va, 0);
594      if(pte == NULL)
595      {
596          pte = pgdir_walk(pgdir, va, 1);
597          if(pte == NULL) return -E_NO_MEM;
598          pp->pp_ref += 1;
599      }
600      else
601      {
602          pp->pp_ref += 1;
603          if(*pte & PTE_P) page_remove(pgdir, va);
604      }
605
606      *pte = PTE_ADDR(page2pa(pp)) | perm | PTE_P;
607      return 0;
608  }
```

这个函数为将一个虚拟地址映射到物理地址上，具体操作为在 593 行先判断该虚拟地址是不有已经挂载了页，如果没有则在 596 行分配一个页并在 606 行对其做映射。如果已经有了挂在就先在 603 行取消映射并在 606 行重新映射。其中由于在取消映射时如果新旧 Page 为同一个，那么在做 page_remove 时会将引用数减一，那么此时 pp->pp_ref 会为 0，这个 page 会被释放，这个结果并不是我们想看到的，因此我先在 602 行将 pp_ref+1，这样改变一下增加 pp_ref 的顺序就可以避免这一问题的发生。

完成上述函数后运行 qemu 如下：

```
Physical memory: 66556K available, base = 640K, extended = 655
32K
check_page_alloc() succeeded!
check_n_pages() succeeded!
check_page() succeeded!
kernel panic at kern/pmap.c:835: assertion failed: check_va2pa
(pgdir, UPAGES + i) == PADDR(pages) + i
Welcome to the JOS kernel monitor!
```

3. Kernel Address Space

3.1 Permissions and Fault Isolation

主要说明了 kernel 和 user 在不同内存地址范围内的权限，这样以保证用户不会使用到 kernel 的内存空间以至于发生系统崩溃。

3.2 Initializing the Kernel Address Space

Exercise 6

> 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.

这个练习就增加了 3 句话，根据注释分别对 pages，boot stack 以及 physical memory 做了虚拟地址的映射。实现后 qemu 显示如下：

```
warning! The value %n argument pointed to has been overflowed!
chnum1: -1
Physical memory: 66556K available, base = 640K, extended = 655
32K
check_page_alloc() succeeded!
check_n_pages() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

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	0xFFC00000	Page table for top 4MB of phys memory
1022	0xFF800000	Page table for next 4MB of phys memory
...	...	...
960	0xF0000000	KERNBASE
959	0xEFC00000	VPT
958	0xEF800000	ULIM
957	0xEF400000	UVPT
956	0xEF000000	UPAGES
955	0xEEC00000	UTOP
...	...	...
2	0x00800000	UTEXT
1	0x00400000	UTEMP
0	0x00000000	0

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?

因为在页表中使用了权限保护，用户只能访问 PTE_U 位为 1 的项。当用户需要访问内存地址时，会在虚拟地址翻译的时候进行检验，如果是只能 kernel 访问，则访问会被拒绝。

Question 4

> What is the maximum amount of physical memory that this operating system can support? Why?

因为 remapped physical memory 为从 0xF0000000-0xFFFFFFFF,共 256MB。

Question 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?

使用 1 个 page directory 以及 1024 个 page table 即能映射 4G 的物理内存，所需的内存空间为 (1+1024)*4K=4M+4K，总共需要 4MB+4KB 的空间。

Question 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?

在 relocated 之后。

因为在初始时虚拟地址的 0-4M 和 KERNBASE 到 KERNBASE+4M 都映射到了物理地址的 0-4M 的空间。

因为 loader 不能直接载入 extended mem。