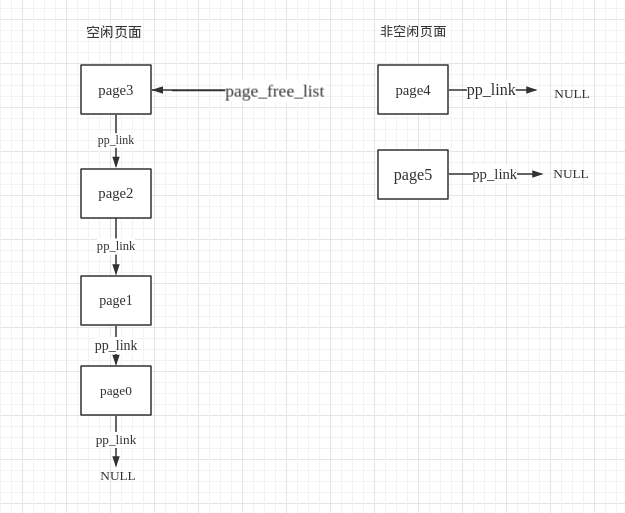
Lab2

**练习1：**



Boot\_alloc:

1. result = nextfree;
2. nextfree = ROUNDUP(nextfree+n, PGSIZE);
3. **return** result;

每次调用都返回nextfree，然后根据参数n更新nextfree的值，使其指向下一个空闲地址处。

mem\_init()调用boot\_alloc()，将返回值赋给全局变量kern\_pgdir，kern\_pgdir保存的是内核页目录的物理地址。

Mem\_init:

1. pages = (**struct** PageInfo\*)boot\_alloc(**sizeof**(**struct** PageInfo) \* npages); //分配足够大的空间(PGSIZE的倍数)保存pages数组
2. memset(pages, 0, **sizeof**(**struct** PageInfo) \* npages);

Page\_init:

1. **size\_t** i;
2. **size\_t** io\_hole\_start\_page = (**size\_t**)IOPHYSMEM / PGSIZE;
3. **size\_t** kernel\_end\_page = PADDR(boot\_alloc(0)) / PGSIZE;
4. **for** (i = 0; i < npages; i++) {
5. **if** (i == 0) {
6. pages[i].pp\_ref = 1;
7. pages[i].pp\_link = NULL;
8. } **else** **if** (i >= io\_hole\_start\_page && i < kernel\_end\_page) {
9. pages[i].pp\_ref = 1;
10. pages[i].pp\_link = NULL;
11. } **else** {
12. pages[i].pp\_ref = 0;
13. pages[i].pp\_link = page\_free\_list;
14. page\_free\_list = &pages[i];
15. }
16. }

这里初始化pages中的每一项，建立page\_free\_list链表

// 已使用的物理页包括如下几部分：

// 1）第一个物理页是IDT所在，需要标识为已用

// 2）[IOPHYSMEM, EXTPHYSMEM)称为IO hole的区域，需要标识为已用。

// 3）EXTPHYSMEM是内核加载的起始位置，终止位置可以由boot\_alloc(0)给出（理由是boot\_alloc()分配的内存是内核的最尾部），这块区域也要标识

! boot\_alloc返回的是虚拟地址，需要转为物理地址

Page\_alloc:

1. **struct** PageInfo \*
2. page\_alloc(**int** alloc\_flags)
3. {
4. // Fill this function in
5. **struct** PageInfo \*one;
6. **if**(page\_free\_list == NULL)
7. **return** NULL;
8. one = page\_free\_list;
9. page\_free\_list = one->pp\_link;
10. one->pp\_link = NULL;
12. **if**(alloc\_flags & ALLOC\_ZERO)
13. memset(page2kva(one), 0, PGSIZE);
14. **return** one;
15. }

从page\_free\_list指向的链表中取一个PageInfo结构返回，根据参数alloc\_flags决定要不要将这块内存初始化为0

Page\_free:

1. **void**
2. page\_free(**struct** PageInfo \*pp)
3. {
4. // Fill this function in
5. // Hint: You may want to panic if pp->pp\_ref is nonzero or
6. // pp->pp\_link is not NULL.
7. **if**(pp->pp\_ref == 0 && pp->pp\_link == NULL){
8. pp->pp\_link = page\_free\_list;
9. page\_free\_list =pp;
11. }
12. **else**{
13. panic("This oage can't be free\n");
14. }
15. }

练习2：

练习3：

问题：



虚拟地址，在程序里面，任何指针都是虚拟地址

练习4：

// 线性地址分为如下三部分

//

// +--------10------+-------10-------+---------12----------+

// | Page Directory | Page Table | Offset within Page |

// | Index | Index | |

// +----------------+----------------+---------------------+

// \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/

// \---------- PGNUM(la) ----------/

//

// 页目录和页表的一些常量定义

#define NPDENTRIES 1024 //每个页目录的页目录项数目为1024

#define NPTENTRIES 1024 //每个页表的页表项数目也为1024

#define PGSIZE 4096 // 页大小为4096B，即4KB

#define PGSHIFT 12 // log2(PGSIZE)

#define PTSIZE (PGSIZE\*NPTENTRIES) // 一个页目录项映射内存大小，4MB

#define PTSHIFT 22 // log2(PTSIZE)

#define PTXSHIFT 12

#define PDXSHIFT 22

// 页号

#define PGNUM(la) (((uintptr\_t) (la)) >> PTXSHIFT)

// 页目录项索引(高10位)

#define PDX(la) ((((uintptr\_t) (la)) >> PDXSHIFT) & 0x3FF)

// 页表项索引（中间10位）

#define PTX(la) ((((uintptr\_t) (la)) >> PTXSHIFT) & 0x3FF)

// 页内偏移

#define PGOFF(la) (((uintptr\_t) (la)) & 0xFFF)

// 由索引构造线性地址

#define PGADDR(d, t, o) ((void\*) ((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))

// 由PageInfo结构得到页面物理地址

static inline physaddr\_t

page2pa(struct PageInfo \*pp)

{

return (pp - pages) << PGSHIFT;

}

// 由物理地址得到PageInfo结构体

static inline struct PageInfo\*

pa2page(physaddr\_t pa)

{

if (PGNUM(pa) >= npages)

panic("pa2page called with invalid pa");

return &pages[PGNUM(pa)];

}

// 与 page2pa 类似，只不过返回的是 PageInfo 结构 pp 所对应的物理页面的内核首地址(虚拟地址)

static inline void\*

page2kva(struct PageInfo \*pp)

{

return KADDR(page2pa(pp));

}

pagedir\_walk：作用：给定pgdir，指向一个页目录，该函数返回一个指针指向虚拟地址va对应的页表条目(PTE)。最后返回PTE的地址的时候，需要返回PTE地址对应的虚拟地址

1. pte\_t \*
2. pgdir\_walk(pde\_t \*pgdir, **const** **void** \*va, **int** create)
3. {
4. // Fill this function in
5. pde\_t\* pde\_ptr = pgdir + PDX(va);
6. **if** (\*pde\_ptr == 0) {                              //页表还没有分配
7. **if** (create) {
8. //分配一个页作为页表
9. **struct** PageInfo \*pp = page\_alloc(1);
10. **if** (pp == NULL) {
11. **return** NULL;
12. }
13. pp->pp\_ref++;
14. \*pde\_ptr = (page2pa(pp)) | PTE\_P | PTE\_U | PTE\_W;   //更新页目录项
15. } **else** {
16. **return** NULL;
17. }
18. }
20. **return** (pte\_t \*)KADDR(PTE\_ADDR(\*pde\_ptr)) + PTX(va);        //转为pte\_t\*类型，KADDR返回的的是void\*类型。
21. }

P —— Present，判断对应物理页面是否存在，存在为1，否则为0;

W —— Write，该位用来判断对所指向的物理页面是否可写，1可写，0不可写；

U —— User，该位用来定义页面的访问者应该具备的权限。为1是User权限即可，如果为 0，

boot\_map\_region（）将[va, va+size)对应的虚拟地址空间映射到物理地址空间[pa, pa+size)。

pgdir:页目录指针

va:虚拟地址

size:大小

pa:物理地址

perm:权限

1. **static** **void**
2. boot\_map\_region(pde\_t \*pgdir, **uintptr\_t** va, **size\_t** size, physaddr\_t pa, **int** perm)
3. {
4. **size\_t** pgs = size / PGSIZE;
5. **if** (size % PGSIZE != 0) {
6. pgs++;
7. }                            //计算总共有多少页
8. **for** (**int** i = 0; i < pgs; i++) {
9. pte\_t \*pte = pgdir\_walk(pgdir, (**void** \*)va, 1);//获取va对应的PTE的地址
10. **if** (pte == NULL) {
11. panic("boot\_map\_region(): out of memory\n");
12. }
13. \*pte = pa | PTE\_P | perm; //修改va对应的PTE的值
14. pa += PGSIZE;             //更新pa和va，进行下一轮循环
15. va += PGSIZE;
16. }
17. }

page\_insert 使va映射到pp对应的物理页处

1. **int**
2. page\_insert(pde\_t \*pgdir, **struct** PageInfo \*pp, **void** \*va, **int** perm)
3. {
4. // Fill this function in
5. pte\_t \*pte = pgdir\_walk(pgdir, va, 1);    //拿到va对应的PTE地址，如果va对应的页表还没有分配，则分配一个物理页作为页表
6. **if** (pte == NULL) {
7. **return** -E\_NO\_MEM;
8. }
9. pp->pp\_ref++;                                       //引用加1
10. **if** ((\*pte) & PTE\_P) {                               //当前虚拟地址va已经被映射过，需要先释放
11. page\_remove(pgdir, va);
12. }
13. physaddr\_t pa = page2pa(pp); //将PageInfo结构转换为对应物理页的首地址
14. \*pte = pa | perm | PTE\_P;    //修改PTE
15. pgdir[PDX(va)] |= perm;
17. **return** 0;
18. }

page\_lookup 返回虚拟地址va对应的物理地址的页面page

1. **struct** PageInfo \*
2. page\_lookup(pde\_t \*pgdir, **void** \*va, pte\_t \*\*pte\_store)
3. {  .0
4. // Fill this function in
5. **struct** PageInfo \*pp;
6. pte\_t \*pte =  pgdir\_walk(pgdir, va, 0);         //如果对应的页表不存在，不进行创建
7. **if** (pte == NULL) {
8. **return** NULL;
9. }
10. **if** (!(\*pte) & PTE\_P) {
11. **return** NULL;
12. }
13. physaddr\_t pa = PTE\_ADDR(\*pte);                 //va对应的物理
14. pp = pa2page(pa);                               //物理地址对应的PageInfo结构地址
15. **if** (pte\_store != NULL) {
16. \*pte\_store = pte;
17. }
18. **return** pp;
19. }

page\_remve 解除va的映射关系。

1. **void**
2. page\_remove(pde\_t \*pgdir, **void** \*va)
3. {
4. // Fill this function in
5. pte\_t \*pte\_store;
6. **struct** PageInfo \*pp = page\_lookup(pgdir, va, &pte\_store); //获取va对应的PTE的地址以及pp结构
7. **if** (pp == NULL) {    //没有映射
8. **return**;
9. }
10. page\_decref(pp);    //将pp->pp\_ref减1，如果pp->pp\_ref为0，需要释放该PageInfo结构（将其放入page\_free\_list链表中）
11. \*pte\_store = 0;    //将PTE清空
12. tlb\_invalidate(pgdir, va); //失效化TLB缓存
13. }

练习5：

1. //////////////////////////////////////////////////////////////////////
2. // Map 'pages' read-only by the user at linear address UPAGES
3. // Permissions:
4. //    - the new image at UPAGES -- kernel R, user R
5. //      (ie. perm = PTE\_U | PTE\_P)
6. //    - pages itself -- kernel RW, user NONE
7. // Your code goes here:
8. // 将虚拟地址的UPAGES映射到物理地址pages数组开始的位置
9. boot\_map\_region(kern\_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE\_U);
10. //////////////////////////////////////////////////////////////////////
11. // Use the physical memory that 'bootstack' refers to as the kernel
12. // stack.  The kernel stack grows down from virtual address KSTACKTOP.
13. // We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
14. // to be the kernel stack, but break this into two pieces:
15. //     \* [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
16. //     \* [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
17. //       the kernel overflows its stack, it will fault rather than
18. //       overwrite memory.  Known as a "guard page".
19. //     Permissions: kernel RW, user NONE
20. // Your code goes here:
21. // 'bootstack'定义在/kernel/entry.
22. boot\_map\_region(kern\_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE\_W);
24. //////////////////////////////////////////////////////////////////////
25. // Map all of physical memory at KERNBASE.
26. // Ie.  the VA range [KERNBASE, 2^32) should map to
27. //      the PA range [0, 2^32 - KERNBASE)
28. // We might not have 2^32 - KERNBASE bytes of physical memory, but
29. // we just set up the mapping anyway.
30. // Permissions: kernel RW, user NONE
31. // Your code goes here:
32. boot\_map\_region(kern\_pgdir, KERNBASE, 0xffffffff - KERNBASE, 0, PTE\_W);

问题2：

| **Entry(PDENo.)** | **Base Virtual Address** | **Points to (logically):** |
| --- | --- | --- |
| 1023 | 0xfff00000 | Page table for top 4MB of phys memory |
| … | … | Page table for top 4MB of phys memory |
| 960 | 0xf0000000 | Page table for top 4MB of phys memory |
| 959 | 0xefff8000 | Kernel Stack & Invalid Memory |
| 958 | 0 | NULL |
| 957 | 0xef400000 | User read-only virtual page table |
| 956 | 0xef000000 | Read-only copies of the Page structures |
| 955 | NULL | NULL |
| … | NULL | NULL |
| 0 | 0x00000000 | 0 |

问题3：靠PTE\_U来保护

问题4：UPAGES总共有4M的内存来存储pages，也就是总共可以存4M/8Byte=0.5M个页面，总共的内存大小为0.5M\*4K=2G，所以总共2G内存最大。

问题5：2G内存的话，总共页面数就是0.5M个，pages结构体(PageInfo)的大小就是0.5M\*8Byte=4M，page director是4K, 由于pages的数目为0.5M,所以page table是0.5M\*4byge=2M，所以总共是6M+4k

问题6：jmp \*%eax之后

entry\_pgdir也把VA映射到了PA

之后kern\_pgdir会被加载，va [0, 4M)会被丢弃。

挑战2：

字符串处理函数，把string转成地址

1. uint32\_t xtoi(**char**\* buf) {
2. uint32\_t res = 0;
3. buf += 2; //0x...
4. **while** (\*buf) {
5. **if** (\*buf >= 'a') \*buf = \*buf-'a'+'0'+10;
6. res = res\*16 + \*buf - '0';
7. ++buf;
8. }
9. **return** res;
10. }

输出函数：

1. **void** pprint(pte\_t \*pte) {
2. cprintf("PTE\_P: %x, PTE\_W: %x, PTE\_U: %x\n",
3. \*pte&PTE\_P, \*pte&PTE\_W, \*pte&PTE\_U);
4. }

Showmapping：

1. **int**
2. showmappings(**int** argc, **char** \*\*argv, **struct** Trapframe \*tf)
3. {
4. **if** (argc == 1) {
5. cprintf("Usage: showmappings 0xbegin\_addr 0xend\_addr\n");
6. **return** 0;
7. }
8. uint32\_t begin = xtoi(argv[1]);
9. uint32\_t end = xtoi(argv[2]);
10. cprintf("begin: %x, end: %x\n", begin, end);
11. **for** (; begin <= end; begin += PGSIZE) {
12. pte\_t \*pte = pgdir\_walk(kern\_pgdir, (**void** \*) begin, 1);
13. **if** (!pte) panic("boot\_map\_region panic, out of memory");
14. **if** (\*pte & PTE\_P) {
15. cprintf("page %x with ", begin);
16. pprint(pte);
17. } **else** cprintf("page not exist: %x\n", begin);
18. }
19. **return** 0;
20. }

Setm:

1. **int** setm(**int** argc, **char** \*\*argv, **struct** Trapframe \*tf) {
2. **if** (argc == 1) {
3. cprintf("Usage: setm 0xaddr [0|1 :clear or set] [P|W|U]\n");
4. **return** 0;
5. }
6. uint32\_t addr = xtoi(argv[1]);
7. pte\_t \*pte = pgdir\_walk(kern\_pgdir, (**void** \*)addr, 1);
8. cprintf("%x before setm: ", addr);
9. pprint(pte);
10. uint32\_t perm = 0;
11. **if** (argv[3][0] == 'P') perm = PTE\_P;
12. **if** (argv[3][0] == 'W') perm = PTE\_W;
13. **if** (argv[3][0] == 'U') perm = PTE\_U;
14. **if** (argv[2][0] == '0')  //clear
15. \*pte = \*pte & ~perm;
16. **else**    //set
17. \*pte = \*pte | perm;
18. cprintf("%x after  setm: ", addr);
19. pprint(pte);
20. **return** 0;
21. }

Showvm:

1. **int** showvm(**int** argc, **char** \*\*argv, **struct** Trapframe \*tf) {
2. **if** (argc == 1) {
3. cprintf("Usage: showvm 0xaddr 0xn\n");
4. **return** 0;
5. }
6. **void**\*\* addr = (**void**\*\*) xtoi(argv[1]);
7. uint32\_t n = xtoi(argv[2]);
8. **int** i;
9. **for** (i = 0; i < n; ++i)
10. cprintf("VM at %x is %x\n", addr+i, addr[i]);
11. **return** 0;
12. }