

实验五 内存

马逸君 17300180070

第一部分 Physical Page Management

阅读kern/pmap.c中的mem_init()函数的代码(到check_page();这句代码为止), 介绍整个流程以及其中boot_alloc(), page_init(), page_alloc(), page_free()这几个函数的作用。

mem_init() (节选) : 用于建立起两级页表的结构。

其工作过程是, 最开始探测本机物理内存的大小, 根据得到的总页面数为页目录(两级页表的第一级)申请空间并填充0初始化, 并通过在页目录中插入自身给予用户代码直接读取二级页表的能力; 然后为二级页表申请一个PageInfo类型的数组, 填充0并调用page_init()对其初始化; 最后进行错误检查。

源码分析:

```
1 // 建立两级页表。kern_pgdir是该页表根部的线性地址(虚拟地址)。
2 // 该函数只建立地址空间的内核部分(亦即地址<=UTOP的部分), 用户部分稍后建立。
3 // 权限设置: UTOP到ULIM, 用户可读不可写; ULIM以上, 用户不可读不可写。
4 void
5 mem_init(void)
6 {
7     ... // 函数开头定义了两个变量, 但我们需要分析的这段代码段中没有用到, 故略去。
8
9     i386_detect_memory();
10    // 该函数用于得到本机拥有的内存大小。
11    // 包括总页数和基础内存(base memory)的页数。
12
13    kern_pgdir = (pde_t *) boot_alloc(PGSIZE); // 创建初始的页目录(两级页表的一
级)
14    memset(kern_pgdir, 0, PGSIZE); // 并填充0初始化
15
16    // 该行代码在页目录中插入自身, 给予了从页目录访问自身的能力。
17    kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
18    // 与PTE_U进行“或”运算, 使得该页表项的present bit为真
19    // 与PTE_P进行“或”运算, 赋予用户读取页目录权限
20
21    // 用boot_alloc()分配一个由npages个PageInfo组成的动态数组, 用指针pages表示。
22    // 内核用这个数组来跟踪维护物理页面; 每个物理页面在该数组中都有一个对应的PageInfo对象。
23    pages = (struct PageInfo *) boot_alloc(npages * sizeof(struct
PageInfo));
24    memset(pages, 0, npages * sizeof(struct PageInfo));
25    // 对每个PageInfo的各项成员填充0初始化
26
27    // 至此我们已经分配了初始的内核数据结构, 可以开始初始化这些分页结构了
(page_init())。
28    // 一旦完成, 以后所有的内存管理都使用page_*()函数, 不再使用boot_alloc()。
29    page_init();
30
31    // 下面执行一些检查。这些函数也定义在pmap.c中。
32    check_page_free_list(1);
33    // 检测空闲页面链表无错误。参数1表示仅检查低地址部分。
```

```

34     check_page_alloc();
35     // 测试物理页面分配器无错误，即page_alloc(), page_free(), page_init()。
36     check_page();
37     // 测试page_insert()和page_remove()无错误。
38
39     // 至此，虚拟内存设置完毕。
40
41     ...
42 }

```

值得讨论的是第17行的这句代码：

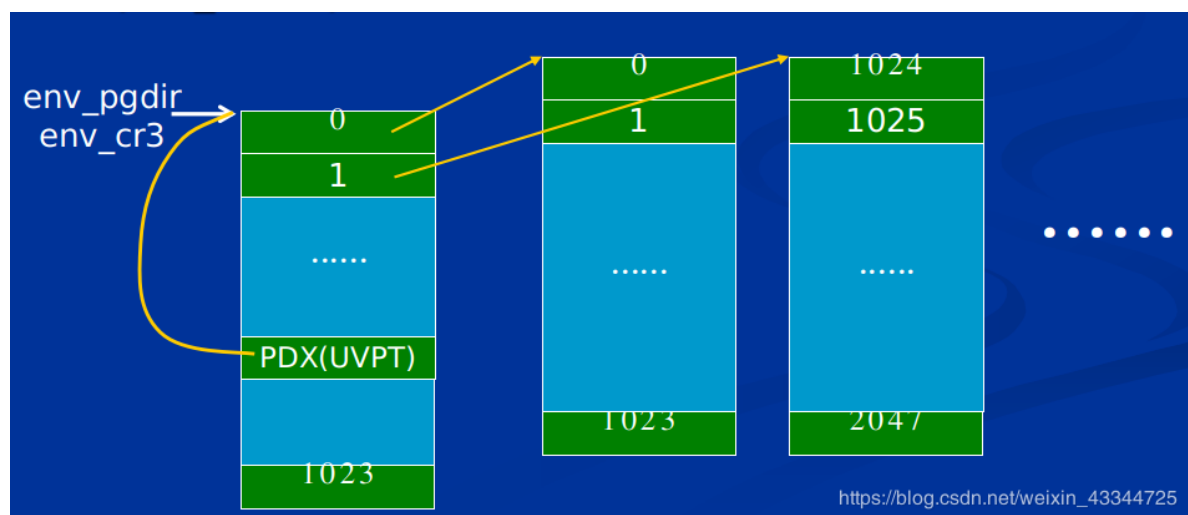
```

1 | kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;

```

此语句上方原有的注释（翻译）：将页目录本身当作一个页表，递归地插入页目录中，以在地址UVPT处构成一个伪页表。（权限：内核可读，用户可读）

这行语句实际上给予了用户程序读取任一页面的页表项的方法，即访问uvpt[i]即可。页表目录、二级页表和各虚拟页面对应关系如图所示[1]：



阅读代码和有关资料可知，访问uvpt[i]时的寻址方式为：



作为对比，通常的寻址方式为：



boot_alloc(): 启动(boot)过程中使用的简易物理内存分配器。

其工作过程是，维护一个全局静态指针nextfree表示内存下一个可用字节的虚拟地址（首次调用boot_alloc()时初始化），每次调用boot_alloc(n)时分配足够容纳n字节个页面的连续物理内存，并更新nextfree指针，若内存不足则报错退出。

源码分析：

```

1 | // 这是一个非常简易的物理内存分配器，仅当JOS还在建立其虚拟内存系统的过程中使用。
2 | // “简易”的特点表现在：只分配处于空闲区域首部的连续页面，且不会回收。

```

```

3 // page_alloc()才是真正的完备的物理内存分配器。
4 //
5 // 若参数n>0, 则分配足够容纳n字节个页面的连续物理内存, 不会初始化该内存区域, 返回一个内核
  虚拟地址。
6 // 若n==0, 返回下一个可用页面的地址, 不分配任何内存。
7 // 若内存不足, boot_alloc()会panic(报错, 且整个xv6退出运行)。
8 //
9 // 该函数必须只有在初始化过程中、在page_free_list建立完成前才能使用。
10 static void *
11 boot_alloc(uint32_t n)
12 {
13     static char *nextfree; // 下一字节可用(free)内存的虚拟地址
14     char *result; // 返回值
15
16     // 若是首次调用该函数(nextfree == 0), 则此处初始化nextfree变量。
17     if (!nextfree) {
18         extern char end[];
19         // 此处的'end'是一个由链接器自动生成的魔数(magic symbol),
20         // 它指向的是内核的bss(???)段的末尾,
21         // 也就是首个未被链接器分配给内核代码或全局变量的虚拟地址。
22         nextfree = ROUNDUP((char *) end, PGSIZE);
23         // 宏ROUNDUP(a, n)用于得到不小于a的最小的n的倍数(定义于inc/types.h)
24         // 该行语句的含义是,
25     }
26
27     // 分配足够容纳n字节个页面的连续物理内存,
28     // 然后更新nextfree指针(会确保nextfree被对齐到PGSIZE的整数倍)。
29     result = nextfree; // 分配内存的首地址被返回
30     nextfree = ROUNDUP(nextfree+n, PGSIZE); // 对齐nextfree指针
31     if((uint32_t)nextfree - KERNBASE > (npages*PGSIZE)) // 若可用空间不足
32         panic("Out of memory!\n"); // 报错退出
33     return result;
34
35 }

```

page_init(): 用于初始化二级页表(具体而言, 初始化pages数组和空闲页面链表page_free_list)。

其工作过程是, 算出在boot_alloc()中分配了哪些页面, 将这些已分配的页面和保留区域(0号页面和I/O保留区域)的页面在pages数组中标记为已使用(引用计数置为1), 将剩余页面标记为可使用(引用计数置为0)并依次插入page_free_list。

源码分析:

```

1 // 本函数初始化页结构和page_free_list。
2 //
3 // pages数组每一项是一个物理页面的页面信息(类型为struct PageInfo)。
4 // 页面采用引用计数(reference counted), 并将空闲页面组织成链表page_free_list。
5 //
6 // 本函数执行完毕后, 千万不要再使用boot_alloc(), 务必只使用page_alloc()、
  page_free()
7 // 来通过page_free_list分配及回收物理内存。
8 //
9 void
10 page_init(void)
11 {
12     // 关于内存地址的划分:

```

```

13 // 1. 将0号物理页面标为使用中，以保留实模式的中断描述符表(IDT)和BIOS，以备不时之
    需。
14 // 2. 基础内存(base memory)的剩余部分[PGSIZE, npages_basemem * PGSIZE)空闲
    可用。
15 // 3. 接下来的I/O保留区域(IO hole)[IOPHYSMEM, EXTPHYSMEM)不能分配。
16 // 4. 扩展内存(extended memory)定义为[EXTPHYSMEM, ...)，可以使用。
17
18 // 但扩展内存区域有部分内存已经被boot_alloc()分配出去，
19 // 从boot_alloc()函数的定义可知，这些内存是连续的，集中在扩展内存的首部，而且未
    被回收，
20 // 我们通过boot_alloc(0)取得空闲区域的首地址，就可以计算已分配的页数。
21
22 size_t i; // 循环变量。(size_t是定义在<stddef.h>文件中的一个类型，用于表示数组
    大小)
23 page_free_list = NULL; // 将空闲页面链表page_free_list初始化为空
24
25 int num_alloc = ((uint32_t)boot_alloc(0) - KERNBASE) / PGSIZE;
26 // num_alloc: 扩展内存区域已被分配的页数
27 int num_iohole = 96; // num_iohole: I/O保留区域(IO hole)占用的页数
28
29 for(i=0; i<npages; i++) // 遍历页表
30 {
31     if(i==0)
32     {
33         pages[i].pp_ref = 1; // 如上所述，将0号物理页面标为已在使用中
34     }
35     else if(i >= npages_basemem && i < npages_basemem + num_iohole +
    num_alloc)
36     {
37         pages[i].pp_ref = 1;
38         // 将I/O保留区域的页面和在boot_alloc()中已分配的页面标为已使用
39     }
40     else
41     {
42         pages[i].pp_ref = 0; // 其他页面均标为空闲
43         pages[i].pp_link = page_free_list;
44         // pp_link的定义: 当前页在空闲链表上的下一页的指针
45         page_free_list = &pages[i]; // 这两行语句将该页插到page_free_list链
    表的首部
46     }
47 }
48
49 }

```

值得讨论的是函数原本注释中的这样一句话：

```

1 // Pages are reference counted, and free pages are kept on a linked list.

```

此处reference counted是什么意思呢？查阅相关资料[2] [3]得知，引用计数(reference counting)是一种广泛使用的内存管理机制，给每项资源维护一个引用计数(reference count)，维护的方法是资源被获取时增加，被释放时减少。

page_alloc()：分配一个物理内存页面。

其工作过程是，首先检查是否还有空闲页面，若没有则返回空地址；取出空闲页面链表(page_free_list)表头的页面，清空其pp_link成员，若参数指定需填充0初始化则给该页面填充0，最后返回该页面。

源码分析:

```
1 // 分配一个物理内存页面。
2 // 如果参数alloc_flags中指定填充0的那个控制位为1, 亦即alloc_flags & ALLOC_ZERO (常
   量)
3 // == 1, 我们就将返回的物理页面全部用'\0'字节填充。
4 // 如果可用空间不足, 返回空地址NULL。
5 //
6 // 本函数不维护每个页面的引用计数(reference count), 这是调用者的工作。调用者可以显式地
   给
7 // 引用计数加一, 也可以调用page_insert()。
8 struct PageInfo *
9 page_alloc(int alloc_flags)
10 {
11     struct PageInfo *result; // 分配结果
12     if (page_free_list == NULL)
13         return NULL; // 没有空闲页面了, 返回NULL
14
15     result = page_free_list; // 取空闲页面链表(page_free_list)头部的第一个页面
16     page_free_list = result->pp_link; // 该页面被取出, 设置下一个页面为新的表头
17     result->pp_link = NULL; // 该页面将被分配, 将其pp_link清空
18     // pp_link的定义: 当前页在空闲链表上的下一页的指针
19
20     if (alloc_flags & ALLOC_ZERO) // 如上所述, 如果参数中指定需要给分配的内存区域填
   充0
21         memset(page2kva(result), 0, PGSIZE); // 填充0初始化
22
23     return result;
24
25 }
```

page_free(): 释放一个物理内存页面, 将其插入空闲页面链表。

工作过程是, 首先检查欲释放的页面是否引用计数为0, 以及是否已经在空闲页面链表中; 若检查无误, 则将其插到空闲页面链表头部。

源码分析:

```
1 // 将一个页面归还到空闲页面链表。
2 // 调用之前, 须确保将释放的页面的引用计数为0。
3 void
4 page_free(struct PageInfo *pp)
5 {
6     assert(pp->pp_ref == 0); // 检查将要释放的页面是否引用计数为0
7     assert(pp->pp_link == NULL); // 检查是否试图释放一个已经是空闲状态的页面
8
9     // 将该页面插入空闲页面链表(page_free_list)
10    pp->pp_link = page_free_list;
11    page_free_list = pp;
12 }
```

第二部分 Virtual Memory

编写管理页表的操作 (依旧是在kern/pmap.c中), 完成page_lookup()函数和page_remove()函数。

为了完成这两个函数的设计，首先梳理一遍二级页表的工作原理：

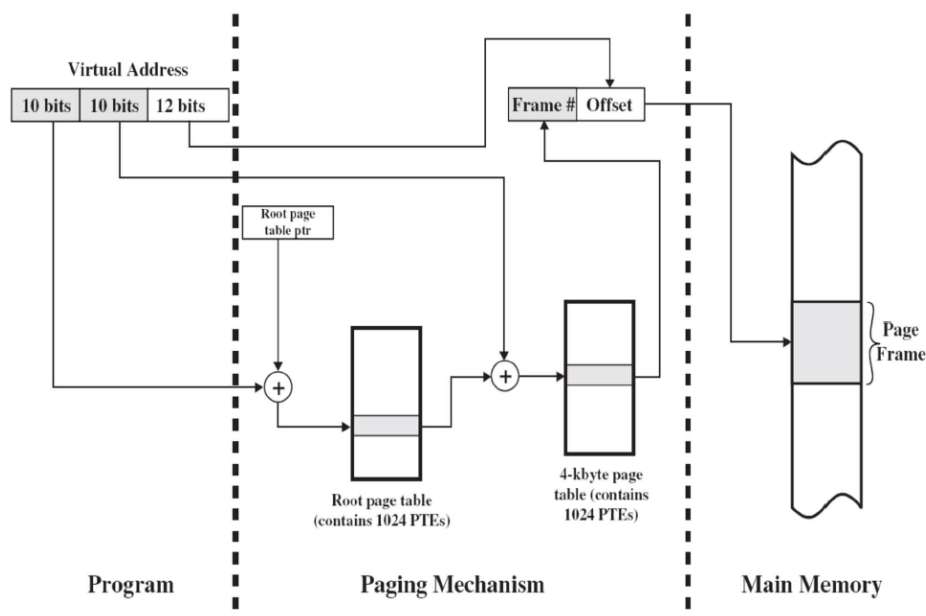


Figure 8.5 Address Translation in a Two-Level Paging System

如图[4]，在二级页表机制下，一个虚拟地址转换到物理地址需要进行两层转换，先取出地址高位部分的页目录索引，在页目录中查询得到页表号，再取出地址中间的页表索引，在对应页表中查询得到页框号，页框号与地址低位部分组合得到物理地址。

在本次lab的代码中，也提供了对虚拟地址的解构，如下（来自mmu.h，由本人翻译）：

```
1 // 线性地址（虚拟地址）'la'的结构分为三个部分：
2 //
3 // +-----10-----+-----10-----+-----12-----+
4 // |      页目录索引      |      页表索引      |      页内偏移量      |
5 // +-----+-----+-----+
6 // \--- PDX(la) --/ \--- PTX(la) --/ \--- PGOFF(la) ----/
7 // \----- PGNUM(la) -----/
8 //
9 // 宏PDX、PTX、PGOFF、PGNUM用于解构线性地址（虚拟地址），如上图所示。
10 // 若要从PDX(la)、PTX(la)和PGOFF(la)三个部分重构线性地址la，使用宏PGADDR：
11 // PGADDR(PDX(la), PTX(la), PGOFF(la))。
```

这样一来就可以很容易地写出这两个函数了：

```
1 struct PageInfo *
2 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
3 {
4     pte_t* entry = pgdir_walk(pgdir, va, 0);
5     if (entry == NULL) return NULL;
6     if (pte_store != NULL) *pte_store = entry;
7     return pa2page(*entry);
8 }
```

这个函数的思路是，用虚拟地址va查询二级页表得到物理地址（具体过程在pgdir_walk()中完成，这个函数的思路是先查页目录得到va对应的二级页表地址，再查二级页表得到va对应的表项），再利用pa2page()得到物理地址对应的PageInfo*。

```
1 void
2 page_remove(pde_t *pgdir, void *va)
3 {
4     struct PageInfo * pp = page_lookup(pgdir, va, NULL);
5     pte_t * entry = pgdir_walk(pgdir, va, 0);
6     if (pp == NULL) return;
7
8     //cprintf("page_remove before decref: %d %u\n", pp->pp_ref, pp);
9     page_decref(pp);
10    //cprintf("page_remove after decref: %d %u\n", pp->pp_ref, pp);
11    tlb_invalidate(pgdir, va);
12    if (entry != NULL) *entry = 0;
13 }
```

这个函数的思路是（其实函数原本的注释中已经清楚地说明了我们需要实现的这个函数的工作步骤），利用虚拟地址va找到对应的物理页面的PageInfo*（若找不到，说明va没有映射任何物理页面，则什么都不做，直接返回），将物理页面的引用计数减1且若减到0则放回空闲页面链表（这通过调用给出的page_decref()来完成），刷新TLB，最后清空va对应的页表项。

—————以上为答案，以下为附加内容—————

值得一提的是，一开始我写错了page_lookup函数，写成了这个样子：

```
1 struct PageInfo *
2 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
3 {
4     ...
5     return pa2page(entry[PTX(va)]);
6 }
```

启动qemu时，系统提示在这一行出现了assertion fail（为更清楚展现出错处，遂将其邻近代码也一并贴上来了，出错的是以下四行中的最后一行）：

```
1 // should be able to map pp2 at PGSIZE because it's already there
2 assert(page_insert(kern_pgdir, pp2, (void*) PGSIZE, PTE_W) == 0);
3 assert(check_va2pa(kern_pgdir, PGSIZE) == page2pa(pp2));
4 assert(pp2->pp_ref == 1);
```

我采用的办法是输出调试，在文档中查找了所有对pp_ref进行修改的地方，在相关代码前后（check_page()、page_insert()、page_remove()三处）都添加了调试信息的输出，例如：

```
1 void
2 page_remove(pde_t *pgdir, void *va)
3 {
4     ...
5     cprintf("page_remove before decref: %d %u\n", pp->pp_ref, pp);
6     page_decref(pp);
7     cprintf("page_remove after decref: %d %u\n", pp->pp_ref, pp);
8     ...
9 }
```


发现page_remove里找到的物理页面PageInfo*和pp2不相等。遂得出结论：问题出在page_lookup()中。我又认真阅读了check_va2pa()（虚拟地址向物理地址转换的一个检测函数）和pgdir_walk()的代码，发现pgdir_walk()找到的就是页表项本身而不是页表项所在的那个页面，对页表项直接解引用（"*entry"）得到的就是va对应的物理地址了。遂作如下修改：

```
1 struct PageInfo *
2 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
3 {
4     ...
5     return pa2page(*entry);
6 }
```

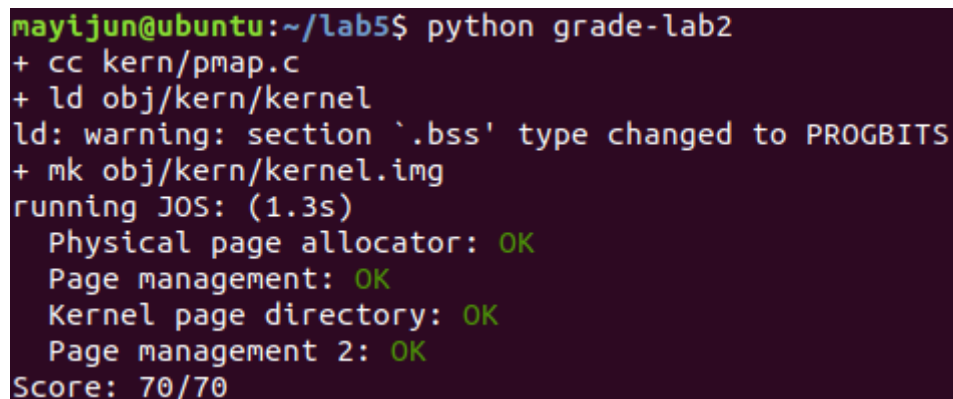
问题解决。

第三部分 Kernel Address Space

（依旧是在kern/pmap.c中）将mem_init()中三处注释有 "//Your code goes here:" 的地方补全。（第一处已经写好）

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

运行评分程序，所有测试项全部通过。



```
mayijun@ubuntu:~/lab5$ python grade-lab2
+ cc kern/pmap.c
+ ld obj/kern/kernel
ld: warning: section '.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
running JOS: (1.3s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

—————以上为答案，以下为附加内容—————

一开始我又写丑了，导致make qemu时qemu无限重启（屏幕上的内容不断清空又重新运行）。当时我是这么写的：

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

最初遇到这个以前从未遇到过的无限重启问题我是很慌的，但我随后发现，每次重启都是运行到同一处代码时触发的，都是在屏幕上显示"check_page() succeeded!"之后开始重启，结合我们这次的任务，基本可以确定是mem_init()中调用check_page()后出现的，很可能就是我填入的这些代码导致的。

这样一来，罪魁祸首就清楚了。我采用了和上一题一样的调试方式：


```
1 boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
2 cprintf("boot map region 1 succeeded!\n");
3
4 boot_map_region(kern_pgdir, KSTACKTOP-KSTACKSIZE, KSTACKSIZE, PADDR(bootstack),
5 PTE_W);
6 cprintf("boot map region 2 succeeded!\n");
7
8 boot_map_region(kern_pgdir, KERNBASE, 0xffffffff, 0, PTE_W);
9 cprintf("boot map region 3 succeeded!\n");
```

这一次，无限重启前可以看到的最后一条消息变成了"boot map region 2 succeeded!"。这时情况已经很清楚了，是第三条语句导致的错误。我重新读了一遍题目，将其修改为：

```
1 boot_map_region(kern_pgdir, KERNBASE, 0xffffffff-KERNBASE, 0, PTE_W);
```

问题解决。

[1] https://blog.csdn.net/weixin_43344725/article/details/89382013

MIT-JOS系列：用户态访问页表项详解

[2] https://ranjitjhala.github.io/static/verifying_reference_counted_implementations.pdf

Verifying Reference Counted Implementations

[3] https://en.wikipedia.org/wiki/Reference_counting

Reference Counting - Wikipedia

[4] <https://elearning.fudan.edu.cn/courses/16555/files?preview=153666>

OS Chapter 7 - Memory Management