第十一章

上一章我们说到，虚拟内存的管理可以分成四部分，**第一个是Higher-Half Kernel，第二个是虚拟内存的寻址与地址映射，第三个是内存管理的初始化，包括建立内存分配与释放等函数，第四个是页错误处理。**

本来可以在上一章都实现的，但一下子放太多的东西，比较复杂，所以我把后两项内容的实现放在这一章。首先来看看我们内核管理的初始化。我们先来修改一下我们的头文件，为它增加一些内容：

/include/vmm.h

.........

1. // 初始化虚拟内存管理
2. void init\_vmm();
3. // 更换当前的页目录
4. void switch\_pgd(uint32\_t pd);
5. // 使用 flags 指出的页权限，把物理地址 pa 映射到虚拟地址 va
6. void map(pgd\_t \*pgd\_now, uint32\_t va, uint32\_t pa, uint32\_t flags);
7. // 取消虚拟地址 va 的物理映射
8. void unmap(pgd\_t \*pgd\_now, uint32\_t va);
9. // 如果虚拟地址 va 映射到物理地址则返回 1
10. // 同时如果 pa 不是空指针则把物理地址写入 pa 参数
11. uint32\_t get\_mapping(pgd\_t \*pgd\_now, uint32\_t va, uint32\_t \*pa);
12. // 页错误中断的函数处理
13. void page\_fault(pt\_regs \*regs);
14. #endif // INCLUDE\_VMM\_H

首先我们接着51行开始，增加了虚拟内存初始化的函数。

55行是内核切换的函数，用于我们从临时页表切换到我们进入内核后的页表。

58行是一个地址映射的函数。可以根据我们申请到的物理地址，把它映射到虚拟地址上。

61行是取消地址映射的函数。

65行是根据虚拟地址去找它对应的物理地址。

68行是我们页错误的函数处理。

然后我们看看具体实现的函数吧：

/vmm/vmm.c

1. #include "idt.h"
2. #include "string.h"
3. #include "print.h"
4. #include "vmm.h"
5. #include "pmm.h"
6. // 内核页目录区域
7. pgd\_t pgd\_kern[PGD\_SIZE] \_\_attribute\_\_ ((aligned(PAGE\_SIZE)));
8. // 内核页表区域
9. static pte\_t pte\_kern[PTE\_COUNT][PTE\_SIZE] \_\_attribute\_\_ ((aligned(PAGE\_SIZE)));
10. void init\_vmm()
11. {
12. // 0xC0000000 这个地址在页目录的索引
13. uint32\_t kern\_pt\_first\_idx = PGD\_INDEX(PAGE\_OFFSET);
15. uint32\_t i, j;
16. //这里是从内核处开始，我们映射了512MB的地址到页目录里
17. for (i = kern\_pt\_first\_idx, j = 0; i < PTE\_COUNT + kern\_pt\_first\_idx; i++, j++) {
18. // 此处是内核虚拟地址，MMU需要物理地址，所以减去偏移，下同
19. pgd\_kern[i] = ((uint32\_t)pte\_kern[j] - PAGE\_OFFSET) | PAGE\_PRESENT | PAGE\_WRITE;
20. }
21. uint32\_t \*pte = (uint32\_t \*)pte\_kern;
23. // 不映射第 0 页，便于跟踪 NULL 指针
24. for (i = 1; i < PTE\_COUNT \* PTE\_SIZE; i++) {
25. pte[i] = (i << 12) | PAGE\_PRESENT | PAGE\_WRITE;
26. }
27. uint32\_t pgd\_kern\_phy\_addr = (uint32\_t)pgd\_kern - PAGE\_OFFSET;
28. // 注册页错误中断的处理函数 ( 14 是页故障的中断号 )
29. register\_interrupt\_handler(14, &page\_fault);
30. switch\_pgd(pgd\_kern\_phy\_addr);
31. }
32. void switch\_pgd(uint32\_t pd)
33. {
34. asm volatile ("mov %0, %%cr3" : : "r" (pd));
35. }
36. void map(pgd\_t \*pgd\_now, uint32\_t va, uint32\_t pa, uint32\_t flags)
37. {
38. uint32\_t pgd\_idx = PGD\_INDEX(va);
39. uint32\_t pte\_idx = PTE\_INDEX(va);
41. pte\_t \*pte = (pte\_t \*)(pgd\_now[pgd\_idx] & PAGE\_MASK);
42. if (!pte) {
43. pte = (pte\_t \*)pmm\_alloc\_page();
44. pgd\_now[pgd\_idx] = (uint32\_t)pte | PAGE\_PRESENT | PAGE\_WRITE;
45. // 转换到内核线性地址并清 0
46. pte = (pte\_t \*)((uint32\_t)pte + PAGE\_OFFSET);
47. bzero(pte, PAGE\_SIZE);
48. } else {
49. // 转换到内核线性地址
50. pte = (pte\_t \*)((uint32\_t)pte + PAGE\_OFFSET);
51. }
52. pte[pte\_idx] = (pa & PAGE\_MASK) | flags;
53. // 通知 CPU 更新页表缓存
54. asm volatile ("invlpg (%0)" : : "a" (va));
55. }
56. void unmap(pgd\_t \*pgd\_now, uint32\_t va)
57. {
58. uint32\_t pgd\_idx = PGD\_INDEX(va);
59. uint32\_t pte\_idx = PTE\_INDEX(va);
60. pte\_t \*pte = (pte\_t \*)(pgd\_now[pgd\_idx] & PAGE\_MASK);
61. if (!pte) {
62. return;
63. }
64. // 转换到内核线性地址
65. pte = (pte\_t \*)((uint32\_t)pte + PAGE\_OFFSET);
66. pte[pte\_idx] = 0;
67. // 通知 CPU 更新页表缓存
68. asm volatile ("invlpg (%0)" : : "a" (va));
69. }
70. uint32\_t get\_mapping(pgd\_t \*pgd\_now, uint32\_t va, uint32\_t \*pa)
71. {
72. uint32\_t pgd\_idx = PGD\_INDEX(va);
73. uint32\_t pte\_idx = PTE\_INDEX(va);
74. pte\_t \*pte = (pte\_t \*)(pgd\_now[pgd\_idx] & PAGE\_MASK);
75. if (!pte) {
76. return 0;
77. }
79. // 转换到内核线性地址
80. pte = (pte\_t \*)((uint32\_t)pte + PAGE\_OFFSET);
81. // 如果地址有效而且指针不为NULL，则返回地址
82. if (pte[pte\_idx] != 0 && pa) {
83. \*pa = pte[pte\_idx] & PAGE\_MASK;
84. return 1;
85. }
86. return 0;
87. }

这个东西解释起来可要费一番口舌了。

7行处是我们的内核页目录，PGD\_SIZE是1024，也就是说我们建立了一个可以访问4G内存的页目录。

11行是页表，PTE\_COUNT是128，PTE\_SIZE是1024，所以这是个可以映射512MB的页表。

然后我们就初始化我们的这个页表，建立起映射关系。

在init\_vmm里，16行是我们找出3GB处对应的页目录，18行是3GB对应的页表项。然后我们以此为映射。

20行开始我们做了页目录的映射，简单说，就是把128个页表分配给了页目录。注意pgd\_kern[i] = ((uint32\_t)pte\_kern[j] - PAGE\_OFFSET) | PAGE\_PRESENT | PAGE\_WRITE; 因为我们在linker script里虽然说的是把我们的**段放到高地址**，但实际上存放的**物理地址还是在低地址**，只不过**虚拟地址是高地址**，一定要搞清楚这个。（虽然物理地址是低地址，但这并没有关系，因为RAM访问的速度都是相同的，而这里的低地址也只是针对实模式进入的时候而言的，实际上建立了虚拟地址和分页后，我们就再也看不到物理地址了，也就没有了物理地址高低之分）

所以我们pte\_kern[j]指的是第一个页表（我们总共有128个页表）的地址，但这个是虚拟地址，而真实的物理地址我们还是非常清楚的（因为页表就是我们建立的，我们当然知道物理地址是怎么映射的了！）我们知道我们吧.text段放到了**虚拟地址的高地址**，所以这里的变量都是高地址。而真实的位置我们也在.text段那里提到了，AT(ADDR(.text) - 0xC0000000)，所以物理地址还是在低地址处，减去偏移量，就是变量的物理地址了。

这里提到了MMU。MMU（memory-management unit）是硬件，叫做内存管理单元，它做的一大工作就是使用TLB（Translation lookaside buffer）把虚拟地址翻译成物理地址。而***TLB建立的基础，就是我们维护的这个页目录和页表***，通过CR0来控制。

接下来，我们25行到30行，是对我们的页表建立映射的过程，相当于把物理地址的低512MB逐项分配给了我们的128个页表。

感觉怪怪的是吧，我们只有512MB的内存，却要把内核放在3GB的地方，但别忘了我们还要虚拟内存。512MB是我们的物理内存（实际上也没这么多，这个也是虚拟的，因为我们现在在qemu下。如果试图分配过多的内存会发生错误的），而3GB是我们的虚拟地址，虚拟地址是我们设定的。

32行是我们找出页目录的物理地址，有了前面的内容，这个就能理解了吧。

然后注册页错误的处理函数，我们在中断那几章做过类似的工作了，如果不懂可以回头看看。

最后，把我们的分页转换到我们高地址的分页！上一章建立的临时页表现在可以忘掉啦！

40行是页表转换的函数switch\_pgd，我们用汇编实现，比较简单，直接更新cr3寄存器就行。

45行是映射函数map，它可以使用 flags 指出的页权限，把物理地址 pa 映射到虚拟地址 va。也就是说，我们可以给定任意一个虚拟地址（在512MB以内），并把它和物理地址相互映射。这个函数很简单，首先根据我们传入的虚拟地址va，计算它的页目录和页表的偏移，然后按照PAGE\_MASK的页对齐查看它是否存在，如果不存在，我们就给它分配一块物理内存，然后写入我们的页表，页目录，最后有bzero置0。如果存在，直接增加我们的偏移量，转换成虚拟地址。然后写入flags权限，并更新页表缓存。

72行的unmap函数是取消虚拟地址 va 的物理映射，也就是把对应的页表置为0。

92行的get\_mapping是查看虚拟地址是否有映射，有映射的话把这个映射的物理地址传递给pa参数并返回1，没有映射的话返回0。

接下来看看我们的错误处理函数：

/vmm/page\_fault.c

1. #include "vmm.h"
2. #include "print.h"
3. #include "idt.h"
4. void page\_fault(pt\_regs \*regs)
5. {
6. uint32\_t cr2;
7. asm volatile ("mov %%cr2, %0" : "=r" (cr2));
8. printk("Page fault at 0x%x, virtual faulting address 0x%x\n", regs->eip, cr2);
9. printk("Error code: %x\n", regs->err\_code);
10. // bit 0 为 0 指页面不存在内存里
11. if ( !(regs->err\_code & 0x1)) {
12. printk("Because the page wasn't present.\n");
13. }
14. // bit 1 为 0 表示读错误，为 1 为写错误
15. if (regs->err\_code & 0x2) {
16. printk("Write error.\n");
17. } else {
18. printk("Read error.\n");
19. }
20. // bit 2 为 1 表示在用户模式打断的，为 0 是在内核模式打断的
21. if (regs->err\_code & 0x4) {
22. printk("In user mode.\n");
23. } else {
24. printk("In kernel mode.\n");
25. }
26. // bit 3 为 1 表示错误是由保留位覆盖造成的
27. if (regs->err\_code & 0x8) {
28. printk("Reserved bits being overwritten.\n");
29. }
30. // bit 4 为 1 表示错误发生在取指令的时候
31. if (regs->err\_code & 0x10) {
32. printk("The fault occurred during an instruction fetch.\n");
33. }
34. while (1);
35. }

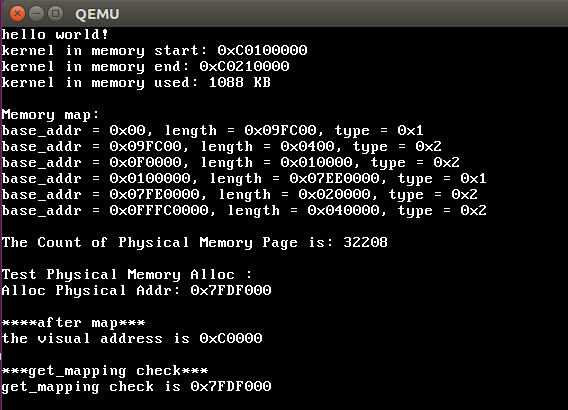
这里也没什么可说的，这些错误代码都是可以查到的。

最后修改我们的entry.c，运行一下：

/entry.c

1. #include "console.h"
2. #include "string.h"
3. #include "print.h"
4. #include "gdt.h"
5. #include "idt.h"
6. #include "timer.h"
7. #include "pmm.h"
8. #include "vmm.h"
9. // 内核初始化函数
10. void kern\_init();
11. // 开启分页机制之后的 Multiboot 数据指针
12. multiboot\_t \*glb\_mboot\_ptr;
13. // 开启分页机制之后的内核栈
14. char kern\_stack[STACK\_SIZE];
15. // 内核使用的临时页表和页目录
16. // 该地址必须是页对齐的地址，内存 0-640KB 肯定是空闲的
17. \_\_attribute\_\_((section(".init.data"))) pgd\_t \*pgd\_tmp = (pgd\_t \*)0x1000;
18. \_\_attribute\_\_((section(".init.data"))) pgd\_t \*pte\_low = (pgd\_t \*)0x2000;
19. \_\_attribute\_\_((section(".init.data"))) pgd\_t \*pte\_hign = (pgd\_t \*)0x3000;
20. // 内核入口函数
21. \_\_attribute\_\_((section(".init.text"))) void kern\_entry()
22. {
23. pgd\_tmp[0] = (uint32\_t)pte\_low | PAGE\_PRESENT | PAGE\_WRITE;
24. pgd\_tmp[PGD\_INDEX(PAGE\_OFFSET)] = (uint32\_t)pte\_hign | PAGE\_PRESENT | PAGE\_WRITE;
25. // 映射内核虚拟地址 4MB 到物理地址的前 4MB
26. int i;
27. for (i = 0; i < 1024; i++) {
28. pte\_low[i] = (i << 12) | PAGE\_PRESENT | PAGE\_WRITE;
29. }
30. // 映射 0x00000000-0x00400000 的物理地址到虚拟地址 0xC0000000-0xC0400000
31. for (i = 0; i < 1024; i++) {
32. pte\_hign[i] = (i << 12) | PAGE\_PRESENT | PAGE\_WRITE;
33. }
35. // 设置临时页表
36. asm volatile ("mov %0, %%cr3" : : "r" (pgd\_tmp));
37. uint32\_t cr0;
38. // 启用分页，将 cr0 寄存器的分页位置为 1 就好
39. asm volatile ("mov %%cr0, %0" : "=r" (cr0));
40. cr0 |= 0x80000000;
41. asm volatile ("mov %0, %%cr0" : : "r" (cr0));
43. // 切换内核栈
44. uint32\_t kern\_stack\_top = ((uint32\_t)kern\_stack + STACK\_SIZE) & 0xFFFFFFF0;
45. asm volatile ("mov %0, %%esp\n\t"
46. "xor %%ebp, %%ebp" : : "r" (kern\_stack\_top));
47. // 更新全局 multiboot\_t 指针
48. glb\_mboot\_ptr = mboot\_ptr\_tmp + PAGE\_OFFSET;
49. // 调用内核初始化函数
50. kern\_init();
51. }
52. void kern\_init()
53. {
54. init\_gdt();
55. init\_idt();
56. console\_clear();
57. printk("hello world!\n");
58. init\_timer(200);
60. //asm volatile("sti");
61. printk("kernel in memory start: 0x%x\n",kern\_start);
62. printk("kernel in memory end: 0x%x\n",kern\_end);
63. printk("kernel in memory used: %d KB\n\n",(kern\_end-kern\_start+1023)/1024);
65. show\_memory\_map();
66. init\_pmm();
67. init\_vmm();
69. printk("\nThe Count of Physical Memory Page is: %d\n\n", phy\_page\_count);
70. uint32\_t allc\_addr = NULL;
71. //测试用的一个虚拟地址
72. uint32\_t allc\_va = 0xC0000;
73. uint32\_t get\_map\_check;
75. printk("Test Physical Memory Alloc :\n");
76. allc\_addr = pmm\_alloc\_page();
77. printk("Alloc Physical Addr: 0x%x\n", allc\_addr);
79. printk("\n\*\*\*\*after map\*\*\*\n");
80. map(pgd\_kern, allc\_va, allc\_addr, 0);
81. printk("the visual address is 0x%x\n",allc\_va);
83. printk("\n\*\*\*get\_mapping check\*\*\*\n");
84. get\_mapping(pgd\_kern, allc\_va, &get\_map\_check);
85. printk("get\_mapping check is 0x%x\n",get\_map\_check);
87. while (1) {
88. asm volatile ("hlt");
89. }
90. }

结果如下：



当前的目录：

jf@ubuntu:~/OS\_jf$ tree

.

├── boot

│   └── boot.s

├── drivers

│   ├── console.c

│   └── timer.c

├── entry.c

├── floppy.img

├── gdt

│   ├── gdt.c

│   └── gdt\_s.s

├── idt

│   ├── idt.c

│   └── idt\_s.s

├── include

│   ├── common.h

│   ├── console.h

│   ├── gdt.h

│   ├── idt.h

│   ├── multiboot.h

│   ├── pmm.h

│   ├── print.h

│   ├── string.h

│   ├── timer.h

│   ├── types.h

│   ├── vargs.h

│   └── vmm.h

├── kernel

│   └── print.c

├── libs

│   ├── common.c

│   └── string.c

├── Makefile

├── pmm

│   └── pmm.c

├── scripts

│   ├── gdbinit

│   └── kernel.ld

└── vmm

├── page\_fault.c

└── vmm.c

10 directories, 30 files