第十章

上一章我们实现了一个极其简单的物理内存管理。而除了物理内存以外，现代计算机广泛采用了**虚拟内存**，虚拟内存是由硬件和软件所包含的特殊的存储器寻址方案。 它允许非连续的物理内存以连续内存的形式被访问。

虚拟内存关键的地方在于**虚拟地址空间**（Virtual Address Space，VAS）的实现，VAS是给程序的地址空间，与物理内存无关。 也就是说每个程序都有自己独立的地址空间。 一方面，可以确保一个程序无法访问另一个程序，因为它们正在使用不同的地址空间。另一方面，因为VAS是虚拟的，与物理内存无关，所以它允许使用其他存储方式，如磁盘驱动器，把它当做内存来用。 也就是说，它允许我们使用更多的“内存”，这就能解决内存不够用的问题了。另一个好处是对于程序来讲，我们可以直接让它从0x0000:0000基址开始使用内存，这样能解决内存碎片和程序重定位相关的一些问题。所以虚拟内存确实是一个非常好的手段，但注意，这里说到的一些实现不只是虚拟内存，还涉及到了分页的功能。

虚拟内存是由内核通过**内存管理单元**（Memory Management Unit，MMU）实现的。内存管理单元，也被叫做分页内存管理单元（Paged Memory Management Unit，PMMU），它放在处理器和存储控制器之间，存储器控制器的主要功能是将存储器地址转换为物理存储器位置，而MMU的则是将虚拟存储地址转换为存储地址以供存储控制器使用。

这里多次提到了**分页**（paging），那么什么是分页呢。前面的内容讨论过简单的内存分段，我们说过，内存分段是为了兼容而不得不做的一项措施。分页本身与分段并不冲突，即便我们开启了分页机制，为了兼容，我们还是需要分段的存在。分页和我们上一章实现的内存管理有点类似的地方，都是把内存划分为4K的块（通过页面大小扩展，PSE，也可以使用更大的页），然后进行管理。但不同于上一章我们简单的分配与释放，分页的管理是精准而又具体的，我们需要维护我们的页表，来获取页面的信息，包括使用情况，权限等内容。同时，我们还要维护我们的页目录，来获取当前页表的情况，以及可以申请或者释放的内存。

分页是实现虚拟内存管理的重要部分，所以这一章我们同时讲分页和虚拟内存的情况。

先来看看页表项（Page Table Entries，PTE）吧，一个页表项描述了4K的内存情况，用4字节的信息来存储。各位的情况如下：

Bit 0 (P): Present flag

0: Page is not in memory

1: Page is present (in memory)

Bit 1 (R/W): Read/Write flag

0: Page is read only

1: Page is writable

Bit 2 (U/S):User mode/Supervisor mode flag

0: Page is kernel (supervisor) mode

1: Page is user mode. Cannot read or write supervisor pages

Bits 3-4 (RSVD): Reserved by Intel

Bit 5 (A): Access flag. Set by processor

0: Page has not been accessed

1: Page has been accessed

Bit 6 (D): Dirty flag. Set by processor

0: Page has not been written to

1: Page has been written to

Bits 7-8 (RSVD): Reserved

Bits 9-11 (AVAIL): Available for use

Bits 12-31 (FRAME): Frame address

好了，现在我们有了一个页表项了，那我们把我们的页表项汇总起来，就是我们的页表了。这个页表结构很简单了，因为页表项大小是4字节，我们可以直接定义成uint32\_t类型的就好，那么我们的页表就是一个uint32\_t类型的数组喽，这个数组我们放1024个元素。所以一个页表管理了4k x 1024 = 4MB的空间。上一章我们提到，我们要管理512MB的内存，所以512MB / 4MB = 128，我们需要128个页表。

呃，但128个页表。。。。是不是还是有点多？而且注意到我们在页表项里的地址可只有20位（FRAME ADDRESS），这个好像不能遍历我们的4G空间吧。而且，虽然页表不大，但在4G空间下，也得4MB的数据结构来存储我们的页表，这可是针对每个程序的诶！也就是说我们要是有1024个程序的话，光页表就要占满整个内存了，所以我们建立**多级页表**。在使用中，每个进程都会被分配一个页目录，但是只有被实际使用页表才会被分配到内存里面。一级页表需要一次分配所有页表空间，两级页表则可以在需要的时候再分配页表空间。我们这里的一级页表指的就是我们的页目录：

页目录项（Page Directory Entries，PDE），页目录表和页表的结构非常类似。 同样是1024个为一个数组，页目录项也有自己的格式：

Bit 0 (P): Present flag

0: Page is not in memory

1: Page is present (in memory)

Bit 1 (R/W): Read/Write flag

0: Page is read only

1: Page is writable

Bit 2 (U/S):User mode/Supervisor mode flag

0: Page is kernel (supervisor) mode

1: Page is user mode. Cannot read or write supervisor pages

Bit 3 (PWT):Write-through flag

0: Write back caching is enabled

1: Write through caching is enabled

Bit 4 (PCD):Cache disabled

0: Page table will not be cached

1: Page table will be cached

Bit 5 (A): Access flag. Set by processor

0: Page has not been accessed

1: Page has been accessed

Bit 6 (D): Reserved by Intel

Bit 7 (PS): Page Size

0: 4 KB pages

1: 4 MB pages

Bit 8 (G): Global Page (Ignored)

Bits 9-11 (AVAIL): Available for use

Bits 12-31 (FRAME): Page Table Base address

关于具体的结构与转换，我们先不说，我们先说说虚拟内存的管理吧。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。**

首先我们来看第一部分，也就是Higher-Half Kernel，这个指的是把内核映射到虚拟内存的高地址区域，现代操作系统，包括windows和linux，都是这样实现的，这样实现的目的主要还是兼容，为了运行V86的程序。

Visual 8086模式，是保护模式的一个子模式，CPU以模拟16位实模式运行。v86个任务在用户模式（user mode）和实模式地址限制（0xffff：0xffff），大约1MB + 64k线性地址下运行。

Linux内核是把内核放到3GB以上的地址空间里， 我们也这样实现。但实现Higher-Half Kernel会给我们的页表建立带来一些麻烦。

因为要改变内核的位置，有了先前的经验，我们首先想到的肯定是修改linker script了，但有一个问题在于，GRUB加载内核是从1MB开始的，所以我们必须把内核的开始放在1MB的地方。为了与GRUB妥协，我们只能把一些必要的代码还是放在1MB处，并且从1MB先开始设置我们的临时页表，然后我们从高地址（3GB处）跳入我们的内核。再重新载入页表。这是一个解决的方法，也是我们要采用的。此外，还有两套方案：

1. 设计bootloader让它把内核直接放到它要执行的虚拟地址处，而物理地址依旧保持在1MB处。比如我们直接把内核放到3GB处，但实际物理上它的位置还是1MB处。这个实现起来不太容易，主要因为我们使用的是GRUB，而不是自己写的bootloader。
2. 第二个方案是Tim Robinson's提出的一个trick，通过GDT的回卷实现，因为对GDT的寻址大于 0xffffffff会发生回卷，所以我们相当于实现了X + 3GB = 1MB，这个要改变的就是GDT的基址了。

为了简单易行，我们不做这么trick 的东西，我们还是老老实实做两次页表好了。首先修改我们的kernel.ld：

/script/kernel.ld

1. ENTRY(start)
2. SECTIONS
3. {
4. PROVIDE( kern\_start = 0xC0100000 );
6. . = 0x100000;
7. .init.text :
8. {
9. \*(.init.text)
10. . = ALIGN(4096);
11. }
12. .init.data :
13. {
14. \*(.init.data)
15. . = ALIGN(4096);
16. }
18. . += 0xC0000000;
19. .text : AT(ADDR(.text) - 0xC0000000)
20. {
21. \*(.text)
22. . = ALIGN(4096);
23. }
24. .data : AT(ADDR(.data) - 0xC0000000)
25. {
26. \*(.data)
27. \*(.rodata)
28. . = ALIGN(4096);
29. }
30. .bss : AT(ADDR(.bss) - 0xC0000000)
31. {
32. \*(.bss)
33. . = ALIGN(4096);
34. }
35. .stab : AT(ADDR(.stab) - 0xC0000000)
36. {
37. \*(.stab)
38. . = ALIGN(4096);
39. }
40. .stabstr : AT(ADDR(.stabstr) - 0xC0000000)
41. {
42. \*(.stabstr)
43. . = ALIGN(4096);
44. }
45. PROVIDE( kern\_end = . );
47. /DISCARD/ : { \*(.comment) \*(.eh\_frame) }
48. }

可以看出，我们还是从1MB处开始我们的内容，这里有两个段，一个是.init.text另一个是.init.data。但我们的内核kern\_start被我们放到了0xC0100000处，包括更多的段。

既然我们修改了内核函数，那么我们的boot.s也是势必要修改的了：

/boot/boot.s

1. MBOOT\_HEADER\_MAGIC equ 0x1BADB002
2. MBOOT\_PAGE\_ALIGN equ 1 << 0
3. MBOOT\_MEM\_INFO equ 1 << 1
4. MBOOT\_HEADER\_FLAGS equ MBOOT\_PAGE\_ALIGN | MBOOT\_MEM\_INFO
5. MBOOT\_CHECKSUM equ - (MBOOT\_HEADER\_MAGIC + MBOOT\_HEADER\_FLAGS)
7. GLOBAL start ; 内核代码入口
8. GLOBAL mboot\_ptr\_tmp ;向外部声明的struct multiboot\*变量
9. EXTERN kern\_entry ; 主函数入口
10. [BITS 32] ; 内核以32位形式编译
11. section .init.text ; 临时代码段
12. align 4 ; 4字节对齐
13. Multi\_boot\_header: ; 声明变量，只用Multiboot Header必须的那些部分
14. dd MBOOT\_HEADER\_MAGIC
15. dd MBOOT\_HEADER\_FLAGS
16. dd MBOOT\_CHECKSUM
17. start: ; 汇编的指令开始的地方
18. cli ; 此时还没有设置好保护模式的中断处理，所以必须关闭中断
19. mov esp, STACK\_TOP ; 设置内核栈地址
20. mov ebp, 0 ; 帧指针修改为 0
21. mov [mboot\_ptr\_tmp], ebx ; 将 ebx 中存储的指针存入全局变量
22. call kern\_entry ; 调用内核入口函数
23. section .init.data
24. stack: times 1024 db 0
25. STACK\_TOP equ $-stack-1
26. mboot\_ptr\_tmp: dd 0

这里主要的改动首先是把我们存放的段改了，代码放在了.init.text段中，而变量不再放在.bss段中，而是放在了.init.data段中，因为我们用完这部分代码后它们就没用了。另一个变化在于把原来的global\_mboot\_ptr改成了mboot\_ptr\_tmp。

接下来我们先来看看要用到的头文件，在里面有一些信息需要我们注意的：

/include/vmm.h

1. #ifndef INCLUDE\_VMM\_H
2. #define INCLUDE\_VMM\_H
3. #include "types.h"
4. // 内核的偏移地址
5. #define PAGE\_OFFSET 0xC0000000
6. //页表项的第1位，存在位
7. #define PAGE\_PRESENT 0x1
8. //页表项的第2位，读写位
9. #define PAGE\_WRITE 0x2
10. //页表项的第3位，权限位
11. #define PAGE\_USER 0x4
12. // 虚拟分页大小
13. #define PAGE\_SIZE 4096
14. // 页掩码，用于 4KB 对齐
15. #define PAGE\_MASK 0xFFFFF000
16. // 获取一个地址的页目录项
17. #define PGD\_INDEX(x) (((x) >> 22) & 0x3FF)
18. // 获取一个地址的页表项
19. #define PTE\_INDEX(x) (((x) >> 12) & 0x3FF)
20. // 获取一个地址的页內偏移
21. #define OFFSET\_INDEX(x) ((x) & 0xFFF)
22. // 页目录数据类型
23. typedef uint32\_t pgd\_t;
24. // 页表数据类型
25. typedef uint32\_t pte\_t;
26. // 页目录成员数
27. #define PGD\_SIZE (PAGE\_SIZE/sizeof(pte\_t))
28. // 页表成员数
29. #define PTE\_SIZE (PAGE\_SIZE/sizeof(uint32\_t))
30. // 映射 512MB 内存所需要的页表数
31. #define PTE\_COUNT 128
32. // 内核页目录区域
33. extern pgd\_t pgd\_kern[PGD\_SIZE];
34. #endif // INCLUDE\_VMM\_H

结合注释相信可以看懂大部分了。

第7行是我们的内核偏移地址，我们内核从3GB处开始。

第10行到第16行是我们对于页表项的几个特殊位的设置，我们在前面的内容里提到过了页表项的各个位所表示的含义。首先，存在位设为1，才表示该页表项是可用的；读写位设为1，表示该页表项既可以读取，又可以进行修改；权限位设为1，表示它具有用户权限，而不是内核权限。

第18行指的是我们分页大小为4K

第21行用于4K对齐，因为4K是0x1000，所以一个地址与0xFFFFF000做并（&）操作，就可以实现4K对齐。

第24行开始就是我们多级页表的内容啦。前面的内容没有详细讲，这里接着前面的内容详细说说，这个就是我们的**虚拟内存的寻址**了。

页表和页目录的建立其实并不只是为了管理的方便，它们还具有提供虚拟地址转换的功能，什么意思呢。比如我们给出一个0xC0000000的虚拟，我们要和物理的低地址1MB（0x100000）映射，那么如何把它映射到我们的页表呢？

我们实际上是把虚拟地址分成了三段：32位，分为高10位，中10位与低12位。高10位用作页目录的索引，中10位用作页表的索引，低12位是页偏移量：

|  |  |  |
| --- | --- | --- |
| 1100 0000 00 | 0000 0000 00 | 0000 0000 0000 |
| 高10位 | 中10位 | 低12位 |
| 页目录索引，等于768 | 页表索引，等于0, | 页偏移，等于0 |

所以我们要做的是去第769个页表找第一个页，然后把偏移量和我们要映射的地址相加，所以这样就是把3GB的虚拟内存映射到物理1MB内存的过程。

我们继续，在34行到37行我们定义了我们的页目录项类型和页表项类型。

40行到43行定义了页表和页目录成员数。都是1024个（结合上面的10位索引，能明白为什么采用1024作为成员个数了吧。因为1111 1111 11就是1023）

46行是512MB需要的页表数

49行是我们在调到3GB空间之后重新定义的页目录。

有了这部分内容，我们就对我们要实现的工作心里有数多了。先看看我们的entry.c吧，看看我们如何实现：

/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();
68. printk("\nThe Count of Physical Memory Page is: %d\n\n", phy\_page\_count);
69. uint32\_t allc\_addr = NULL;
70. printk("Test Physical Memory Alloc :\n");
71. allc\_addr = pmm\_alloc\_page();
72. printk("Alloc Physical Addr: 0x%x\n", allc\_addr);
73. allc\_addr = pmm\_alloc\_page();
74. printk("Alloc Physical Addr: 0x%x\n", allc\_addr);
75. allc\_addr = pmm\_alloc\_page();
76. printk("Alloc Physical Addr: 0x%x\n", allc\_addr);
77. allc\_addr = pmm\_alloc\_page();
78. printk("Alloc Physical Addr: 0x%x\n", allc\_addr);
80. while (1) {
81. asm volatile ("hlt");
82. }
83. }

啊哈，这个可复杂了呢。首先我们定义了kern\_init()，这个就是我们的内核函数了，也是我们要从3GB开始的内核。在boot.s里我们把开始加载的函数是kern\_entry()，但这个是在1MB的低地址处的，所以我们又加了这个kern\_init()。

14行是开启分页后的指针，glb\_mboot\_ptr，我们随后会把它放到高地址处的。

17行是我们3GB开始的内核的栈，其中STACK\_SIZE是在上一章中，我们的pmm.h里声明的。

21行开始我们声明了我们的临时用的页表和页目录，分别放在了0x1000，0x2000，0x3000地址处，并声明它们在.init.data段里。

26行是我们kern\_entry()函数，并声明它在.init.text段里。进入函数后，28行和29行我们映射了两个页表到页目录里，为什么仅仅是两个而不是全部的内存呢？因为我们内核很小，有这两个就足够了，毕竟，一个页表就可以使用4MB的内存了。

32行到40行，我们把这两个页表分别映射了对应的物理地址。

43行是设置临时页表，这些对于控制寄存器的操作我们在附录1里面有介绍。

48到50行设置了cr0的最高位，加载了我们刚才设置好的在cr3里的页目录。

此时，我们就进入了分页模式了。

53行我们重新设置了我们的栈，并在54行到55行更新了我们的esp。

58行我们把我们在boot.s里定义的mboot\_ptr\_tmp转移到了glb\_mboot\_ptr中，把它放在了高地址处。

终于做完了这些准备工作了，我们在61行跳入我们的高地址内核函数。

除此之外，还有一些要修改的代码，在multiboot.h里加上我们新声明的指针：

/include/multiboot.h

.........

1. // 声明全局的 multiboot\_t \* 指针
2. //内核未开启分页的指针，放在低地址处
3. extern multiboot\_t \*mboot\_ptr\_tmp;
4. //开启分页之后的指针，放在高地址处
5. extern multiboot\_t \*glb\_mboot\_ptr;

.........

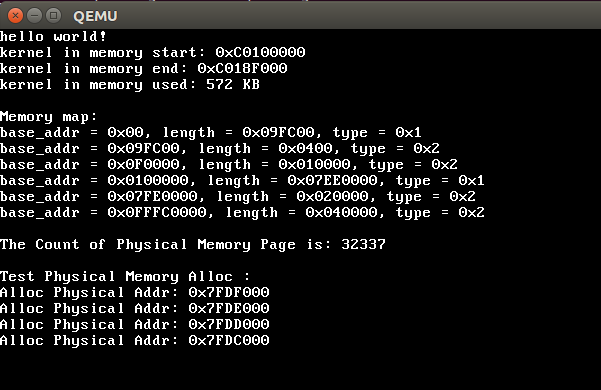
在console.c里面修改我们显存的起始地址：

/driver/console.c

1. #include "console.h"
2. #include "common.h"
3. #include "vmm.h"
4. // VGA 的显示缓冲的起点是 0xB8000
5. static uint16\_t \*video\_memory = (uint16\_t \*)(0xB8000 + PAGE\_OFFSET);

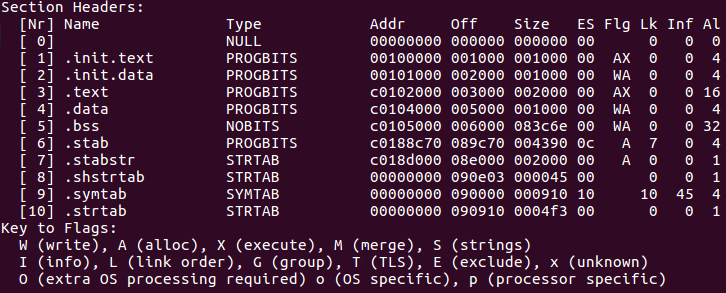
.........

我们来试试这个临时的分页好不好用吧，试着运行一下：



和我们上一章完成的内容差不多。但注意我们内核放置的位置已经变了。

执行readelf -a jf\_kernel 看一下我们段的情况：



后面还有我们变量的情况，可以看出来已经按照高地址实现了。

此时的目录：

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

9 directories, 28 files

<https://en.wikipedia.org/wiki/Paging>

http://www.brokenthorn.com/Resources/OSDev18.html

<http://blog.csdn.net/lsjseu/article/details/9016335>

<http://blog.xiaohansong.com/2015/10/05/Linux%E5%86%85%E5%AD%98%E5%AF%BB%E5%9D%80%E4%B9%8B%E5%88%86%E9%A1%B5%E6%9C%BA%E5%88%B6/>

<http://wiki.osdev.org/Virtual_8086_Mode>

<http://wiki.osdev.org/Higher_Half_Kernel>

http://www.osdever.net/tutorials/