第八章

从这一章开始，我们又要进行下一个big project了。就是我们的内存管理。

那么，既然要管理它，首先我们得知道内存是什么才行。那么内存是什么呢，我们这里指的内存是存储在计算机随机存取存储器(Random Access Memory ，RAM)中的抽象存储块。物理内存在RAM中“存储”的方式取决于系统使用的RAM类型。例如，动态随机存取存储器(Dynamic Random Access Memory ，DRAM)存储用电容器(capacitor)。 电容器是在有限时间内存储电流的电子设备。 它存储电流，表示二进制1，无电流，表示二进制0。

但我们的存储器不是直接接入计算机的，物理内存(RAM，SRAM，DRAM等)需要特定类型的存储器控制器(Memory Controller)与处理器和系统总线进行接口。内存控制器不仅提供了一种通过软件读取和写入内存位置的方法，同时负责持续刷新RAM芯片，以确保其保留其信息(比如DRAM，电容会自动放电的)。

存储器控制器包含多路复用器(Multiplexer)和解复用器(Demultiplexer)电路，用于选择RAM芯片以及地址总线中的地址。 通过这些部件，处理器就能够通过地址总线发送内存地址来读取特定内存。

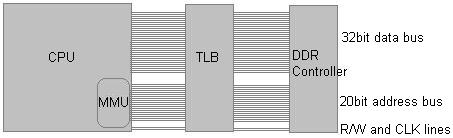
存储器控制器以顺序的方式选择RAM芯片内的位置。如果我们访问的物理内存位置大于系统中的内存总量，则不会发生任何事情。可以向该内存位置写入一个值并将其读回来 ——但获得的是数据总线上的残余数据。物理地址空间中也可能有缺陷。例如，如果RAM芯片在插槽1和3中，在插槽2中没有RAM芯片。所以结论就是**无论读取还是写入到不存在的内存处，都不会有任何事情发生。**如果去读，可能读到的是总线上残余的垃圾信息，而写的话则什么也不会发生。

对于内存基本的内容就介绍这么多。接下来我们来看点实际的东西——物理内存空间(Physical Address Space，PAS)。

物理地址空间，PAS，指的是处理器使用的（并由存储控制器转换）用于指代存储在物理存储器（RAM）中的8位数据（即，一个字节）的地址空间。这个定义有点拗口，其实很简单，一个内存地址只是存储控制器为一个数据字节选择的一个数字，比如地址0代表了物理内存的第一个字节。而我们把这些数字的地址汇总起来，就是所谓的物理地址空间。

好了，现在有了地址空间了，处理器就可以按照这个表来调用内存了。但这是硬件层面的，我们的软件如何使用内存呢。这就要说说**寻址模式**(Addressing Mode)了, 寻址模式是由处理器来管理软件如何访问物理地址空间的一个抽象。其实此前我们已经见过两种寻址模式了，一个是实模式下的段:偏移（segment:offset）寻址，一个是保护模式下的描述符:偏移（descriptor:offset）寻址。

说完这些了，那我们看看硬件上的寻址吧。在附录1中我们对系统总线讲了一部分内容了，并且说到了一点内存工作的方法。这里先给一张图：



这里面，TLB指的是翻译后备缓冲区（Translation Lookaside Buffer），而DDR controller就是我们的存储控制器了。这里有32位的数据总线，20位的地址总线，而控制总线我们暂时只需要R/W和CLK这两条线就先。

这里面的TLB仅仅当分页开启时才有用，它会对我们的地址做另一个翻译。

先不看TLB。在实际中，如果有一个写操作的话，处理器将RW引脚设置为高电平（相当于逻辑1）。 这将告诉与其连接的设备将进行写入操作。 处理器将I/O控制线重置为低电平（逻辑0）。 这确保I/O子系统忽略该命令（意思是它不是IN / OUT端口指令，而是内存控制器。因为我们之前说过，内存里也映射了很多硬件设备的端口），然后，处理器将要写入的地址复制到地址总线上，并将数据写入数据总线。 因为这些线间接连接到存储器控制器，所以存储器控制器能够看到它是写操作。 所以，所有的存储器控制器需要做的是使用其解复用器电路来转换地址总线上的存储地址，来对应要使用的RAM芯片，并将线性偏移字节转换为RAM芯片存储器空间。 存储器控制器然后将数据从数据总线复制到该位置，并在下一个时钟信号刷新存储器状态。

虽然看起来内容很多，但实际上并不是十分麻烦。而读操作与这个类似，不说了。

前面对我们的内存算是有了一个比较详细的介绍了，接下来我们说说内存的管理吧。

要管理我们的内存，首先得对内存有一个了解，所以现在要做的第一件事情就是检测我们的内存，了解它的大小，分布等情况。如何检测内存呢？放一张图：



熟悉吧，注意第五行开始的内容，就是我们内存的最基本信息了。所以想一下就明白了，对内存的检测当然是在BIOS就要进行的，所以我们可以通过BIOS函数来实现对内存的检测。但所幸，我们直接使用了GRUB来加载我们的内核，这就提供了很大的方便，因为麻烦的内容GRUB已经帮我们完成了。

第二章最后我们提到，“系统信息和启动信息块的线性地址保存在 EBX中(相当于一个指针)” 之前我们没有介绍这个信息，那么这个信息都有什么呢。这个信息的结构如下：

+-------------------+

0 | flags | (required)

+-------------------+

4 | mem\_lower | (present if flags[0] is set)

8 | mem\_upper | (present if flags[0] is set)

+-------------------+

12 | boot\_device | (present if flags[1] is set)

+-------------------+

16 | cmdline | (present if flags[2] is set)

+-------------------+

20 | mods\_count | (present if flags[3] is set)

24 | mods\_addr | (present if flags[3] is set)

+-------------------+

28 - 40 | syms | (present if flags[4] or

| | flags[5] is set)

+-------------------+

44 | mmap\_length | (present if flags[6] is set)

48 | mmap\_addr | (present if flags[6] is set)

+-------------------+

52 | drives\_length | (present if flags[7] is set)

56 | drives\_addr | (present if flags[7] is set)

+-------------------+

60 | config\_table | (present if flags[8] is set)

+-------------------+

64 | boot\_loader\_name | (present if flags[9] is set)

+-------------------+

68 | apm\_table | (present if flags[10] is set)

+-------------------+

72 | vbe\_control\_info | (present if flags[11] is set)

76 | vbe\_mode\_info |

80 | vbe\_mode |

82 | vbe\_interface\_seg |

84 | vbe\_interface\_off |

86 | vbe\_interface\_len |

+-------------------+

可以看到，这个信息量很大啊23333。但弱水三千，我们只取一瓢饮。先说flag，它表示了Multiboot信息结构中其他字段的存在和有效性。 引导加载程序必须将所有尚未定义的位设置为零。 应该忽略操作系统不了解的任何设置位。 因此，flag也用作版本指示器，允许将来扩展Multiboot信息结构，而不会破坏任何内容。

紧跟在flag下面的，就是我们的内存信息了，分别定义了我们内存的最低值和最高值，而44到52字节则给出了我们的内存映射的情况。

我们来以此为基础，建立这个结构体：

/include/multiboot.h

1. #ifndef INCLUDE\_MULTIBOOT\_H\_
2. #define INCLUDE\_MULTIBOOT\_H\_
3. #include "types.h"
4. typedef
5. struct multiboot\_t {
6. uint32\_t flags; // Multiboot 的版本信息
7. /\*\*
8. \* 从 BIOS 获知的可用内存
9. \*
10. \* mem\_lower和mem\_upper分别指出了低端和高端内存的大小，单位是K。
11. \* 低端内存的首地址是0，高端内存的首地址是1M。
12. \* 低端内存的最大可能值是640K。
13. \* 高端内存的最大可能值是最大值减去1M。但并不保证是这个值。
14. \*/
15. uint32\_t mem\_lower;
16. uint32\_t mem\_upper;
17. uint32\_t boot\_device; // 指出引导程序从哪个BIOS磁盘设备载入的OS映像
18. uint32\_t cmdline; // 内核命令行
19. uint32\_t mods\_count; // boot 模块列表
20. uint32\_t mods\_addr;
22. /\*\*
23. \* ELF 格式内核映像的section头表、每项的大小、一共有几项以及作为名字索引的字符串表。
24. \* 它们对应于可执行可连接格式（ELF）的program头中的shdr\_\* 项（shdr\_num等）。
25. \* 所有的section都会被载入，ELF section头的物理地址域指向所有的section在内存中的位置。
26. \*/
27. uint32\_t num;
28. uint32\_t size;
29. uint32\_t addr;
30. uint32\_t shndx;
31. /\*\*
32. \* 以下两项指出保存由BIOS提供的内存分布的缓冲区的地址和长度
33. \* mmap\_addr是缓冲区的地址，mmap\_length是缓冲区的总大小
34. \* 缓冲区由一个或者多个下面的大小/结构对 mmap\_entry\_t（size实际上是用来跳过下一个对的）组成
35. \*/
36. uint32\_t mmap\_length;
37. uint32\_t mmap\_addr;
39. uint32\_t drives\_length; // 指出第一个驱动器结构的物理地址
40. uint32\_t drives\_addr; // 指出第一个驱动器这个结构的大小
41. uint32\_t config\_table; // ROM 配置表
42. uint32\_t boot\_loader\_name; // boot loader 的名字
43. uint32\_t apm\_table; // APM 表
44. uint32\_t vbe\_control\_info;
45. uint32\_t vbe\_mode\_info;
46. uint32\_t vbe\_mode;
47. uint32\_t vbe\_interface\_seg;
48. uint32\_t vbe\_interface\_off;
49. uint32\_t vbe\_interface\_len;
50. } \_\_attribute\_\_((packed)) multiboot\_t;
51. typedef
52. struct mmap\_entry\_t {
53. uint32\_t size; // 留意 size 是不含 size 自身变量的大小
54. uint32\_t base\_addr\_low;
55. uint32\_t base\_addr\_high;
56. uint32\_t length\_low;
57. uint32\_t length\_high;
58. uint32\_t type;
59. } \_\_attribute\_\_((packed)) mmap\_entry\_t;
60. // 声明全局的 multiboot\_t \* 指针
61. extern multiboot\_t \*glb\_mboot\_ptr;
62. #endif // INCLUDE\_MULTIBOOT\_H\_

在boot.s里有：GLOBAL glb\_mboot\_ptr以及mov [glb\_mboot\_ptr], ebx。前者早声明了我们的外部变量glb\_mboot\_ptr，而后者把ebx中结构体的信息传递给了这个变量。所以我们这个在multiboot.h的67行，只需要extern multiboot\_t \*glb\_mboot\_ptr;一个声明就可以直接使用这些信息了，很方便，不是吗。

下面的mmap\_entry\_t就是我们的内存映射的结构了。它实际上是这样的：

+-------------------+

-4 | size |

+-------------------+

0 | base\_addr |

8 | length |

16 | type |

+-------------------+

Flag的第6位有效时，指示了包含由BIOS提供的机器的内存映射的缓冲器的地址和长度。 mmap\_addr 存储了我们内存映射的地址，mmap\_length是长度。

这个结构体中，其中'size'是以字节表示的的关联结构的大小，可以大于20个字节的最小值。 'base\_addr'是起始地址。 “length”是以字节为单位的存储区大小。 “type”是表示的地址的特性，其中值1指示可用的RAM，其他值指示保留区域。

而在我们的multiboot.h中，我们把base\_addr和length都拆成了两段。因为我们定义的变量类型是4字节的。

好了，我们用这样一个函数来打印物理内存段：

/pmm/pmm.c

1. #include "multiboot.h"
2. #include "print.h"
3. #include "pmm.h"
4. void show\_memory\_map()
5. {
6. uint32\_t mmap\_addr = glb\_mboot\_ptr->mmap\_addr;
7. uint32\_t mmap\_length = glb\_mboot\_ptr->mmap\_length;
8. printk("Memory map:\n");
9. mmap\_entry\_t \*mmap = (mmap\_entry\_t \*)mmap\_addr;
10. for (mmap = (mmap\_entry\_t \*)mmap\_addr; (uint32\_t)mmap < mmap\_addr + mmap\_length; mmap++) {
11. printk("base\_addr = 0x%x%x, length = 0x%x%x, type = 0x%x\n",
12. (uint32\_t)mmap->base\_addr\_high, (uint32\_t)mmap->base\_addr\_low,
13. (uint32\_t)mmap->length\_high, (uint32\_t)mmap->length\_low,
14. (uint32\_t)mmap->type);
15. }
16. }

这个函数并不复杂，首先把我们multiboot的信息传递给了mmap，然后对mmap的信息输出。

有了这部分信息了，我们还需要知道内核本身加载到物理内存的位置信息，这块内存必须是物理内存管理所保留的。不然把我们的内核覆盖了就不好玩了。这个问题好像不好解决，尽管我们可以用readelf以及objdump这样的命令获得很多信息，但实际加载的时候，我们是无法预料的，必须找一个和内核加载相关的文件才行。从这个角度来想，我们这里唯一和内核加载相关的，就是我们的链接文件了，所以解决办法很简单，直接在链接文件里声明两个定义就行：

/script/linker.ld

..........

1. . = 0x100000;
2. PROVIDE( kern\_start = . );
3. .text :

..........

1. .stabstr :
2. {
3. \*(.stabstr)
4. . = ALIGN(4096);
5. }
6. PROVIDE( kern\_end = . );

.........

这里我们在内核开始的时候，声明了一个变量叫kern\_start，它的值就是当前地址。内核结束的时候，我们声明了一个变量叫kern\_end，它的值同样是当前地址。这样我们就获得了内核的开始和结束的地址了。

此时我们的头文件如下：

/include/pmm.h

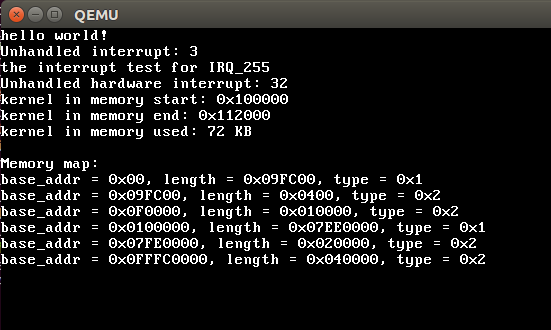
1. #ifndef INCLUDE\_PMM\_H
2. #define INCLUDE\_PMM\_H
3. #include "multiboot.h"
4. // 内核文件在内存中的起始和结束位置
5. // 在链接器脚本中要求链接器定义
6. extern uint8\_t kern\_start[];
7. extern uint8\_t kern\_end[];
8. // 输出 BIOS 提供的物理内存布局
9. void show\_memory\_map();
10. #endif // INCLUDE\_PMM\_H

修改我们的entry.c，看看效果：

/entry.c

1. #include"console.h"
2. #include"print.h"
3. #include"gdt.h"
4. #include"idt.h"
5. #include"timer.h"
6. #include"pmm.h"
7. int kern\_entry()
8. {
9. init\_gdt();
10. init\_idt();
11. console\_clear();
12. printk("hello world!\n");
13. asm volatile ("int $0x3");
14. asm volatile ("int $0xFF");
15. asm volatile ("int $0x20");
16. init\_timer(200);
18. //asm volatile("sti");
19. printk("kernel in memory start: 0x%x\n",kern\_start);
20. printk("kernel in memory end: 0x%x\n",kern\_end);
21. printk("kernel in memory used: %d KB\n\n",(kern\_end-kern\_start+1023)/1024);
23. show\_memory\_map();
24. return 0;
25. }

编译运行，结果如下：



这里的内容就是我们使用的内存以及内存映射了，但注意这个内存映射是我们虚拟机的配置，我们的虚拟机用的是默认配置。

第一部分可以看到，我们当前的内容大概用了72kb，第二部分是我们的内存映射表。

现在的目录如下：

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

├── kernel

│   └── print.c

├── libs

│   ├── common.c

│   └── string.c

├── Makefile

├── pmm

│   └── pmm.c

└── scripts

├── gdbinit

└── kernel.ld

9 directories, 27 files

http://www.brokenthorn.com/Resources/OSDev17.html

https://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Boot-information-format