第九章

在上一章我们对我们的内存做了简单的介绍，给出了内存的检测。这一章我们就开始实现对内存的管理。

如果要管理全部的内存信息，首先就要记录下全部的信息，但想想，如果要记录全部的内存信息，实在是太费内存了！假设只使用0和1来表示这一字节是否使用，那么就要用掉八分之一的内存了！！（一个字节有8位）。而对这个表的维护又需要不少的内存。简直太可怕了。事实上，我们并不需要记录那么多的信息，我们都是把内存分成块（block），按照块进行管理的。在诸多的考虑之下，这个块的大小使用4kb。

好了，那么我们怎么管理内存呢？管理内存需要比较好的算法，要尽可能兼顾到小文件和大文件的内存申请，尽量保证我们物理内存的连续性。Linux上比较著名的算法就是伙伴算法，slab算法了，前者应用于大文件，后者应用于小文件。但我比较懒。。。这里懒得去做了（写了这么多内容了。。。很累的诶，有机会我会在附录里添加）这里只做一个非常智障非常低级的管理：用一个栈来实现。分配的时候我们把栈指针增加，用完了就减。可以看出来这种分配只能针对单一任务。（有没有搞错，操作系统竟然不支持多任务！）

我们直接在上一章的pmm.c里面加入这部分函数好了。代码如下：

/pmm/pmm.c

1. #include "multiboot.h"
2. #include "common.h"
3. #include "print.h"
4. #include "pmm.h"
5. // 物理内存页面管理的栈
6. static uint32\_t pmm\_stack[PAGE\_MAX\_SIZE+1];
7. // 物理内存管理的栈指针
8. static uint32\_t pmm\_stack\_top;
9. // 物理内存页的数量
10. uint32\_t phy\_page\_count=0;
11. void show\_memory\_map()
12. {
13. uint32\_t mmap\_addr = glb\_mboot\_ptr->mmap\_addr;
14. uint32\_t mmap\_length = glb\_mboot\_ptr->mmap\_length;
15. printk("Memory map:\n");
16. mmap\_entry\_t \*mmap = (mmap\_entry\_t \*)mmap\_addr;
17. for (mmap = (mmap\_entry\_t \*)mmap\_addr; (uint32\_t)mmap < mmap\_addr + mmap\_length; mmap++) {
18. printk("base\_addr = 0x%x%x, length = 0x%x%x, type = 0x%x\n",
19. (uint32\_t)mmap->base\_addr\_high, (uint32\_t)mmap->base\_addr\_low,
20. (uint32\_t)mmap->length\_high, (uint32\_t)mmap->length\_low,
21. (uint32\_t)mmap->type);
22. }
23. }
24. void init\_pmm()
25. {
26. //把multiboot的信息传递给了mmap
27. mmap\_entry\_t \*mmap\_start\_addr = (mmap\_entry\_t \*)glb\_mboot\_ptr->mmap\_addr;
28. mmap\_entry\_t \*mmap\_end\_addr = (mmap\_entry\_t \*)glb\_mboot\_ptr->mmap\_addr + glb\_mboot\_ptr->mmap\_length;
30. //指针变量，在for循环里用作循环指示，对mmap里的内存段进行遍历
31. mmap\_entry\_t \*map\_entry;
32. //for循环，遍历我们上一章给出的mmap
33. for (map\_entry = mmap\_start\_addr; map\_entry < mmap\_end\_addr; map\_entry++) {
34. // 如果是可用内存 ( 按照协议，1 表示可用内存，其它数字指保留区域 )
35. if (map\_entry->type == 1 && map\_entry->base\_addr\_low == 0x100000) {
37. // 把内核结束位置到结束位置的内存段，按页存储到页管理栈里
38. // 最多支持512MB的物理内存
39. uint32\_t page\_addr = map\_entry->base\_addr\_low + (uint32\_t)(kern\_end - kern\_start);
40. uint32\_t length = map\_entry->base\_addr\_low + map\_entry->length\_low;
41. //while循环，用来在当前地址空间进行计数，按照4K页大小统计，把信息写入我们的phy\_page\_count里
42. while (page\_addr < length && page\_addr <= PMM\_MAX\_SIZE) {
43. pmm\_free\_page(page\_addr);
44. page\_addr += PMM\_PAGE\_SIZE;
45. phy\_page\_count++;
46. }
47. }
48. }
49. }
50. uint32\_t pmm\_alloc\_page()
51. {
52. error\_detect(pmm\_stack\_top != 0, "out of memory");
54. uint32\_t page = pmm\_stack[pmm\_stack\_top--];
55. return page;
56. }
57. void pmm\_free\_page(uint32\_t p)
58. {
59. error\_detect(pmm\_stack\_top != PAGE\_MAX\_SIZE, "out of pmm\_stack");
61. pmm\_stack[++pmm\_stack\_top] = p;
62. }

这一部分中，我们第7行首先定义了一个页面管理的栈，用来存放我们的页面的数量。10行定义了对这个栈进行指向的栈指针。13行是我们要统计的内存页的数量。15行还是我们的show\_memory\_map函数，上一章有介绍。在31行我们定义了对内存初始化的函数，这个函数首先把multiboot的信息传递给了mmap，然后我们对mmap进行了遍历，找出了我们要使用的内存部分。然后用一个while循环对这部分内存进行了计算。这里的注释写的比较详细，应该能看懂。后面我们还定义了一个分配函数，一个释放函数，也都是超级简单的。

看下头文件吧：

/include/pmm.h

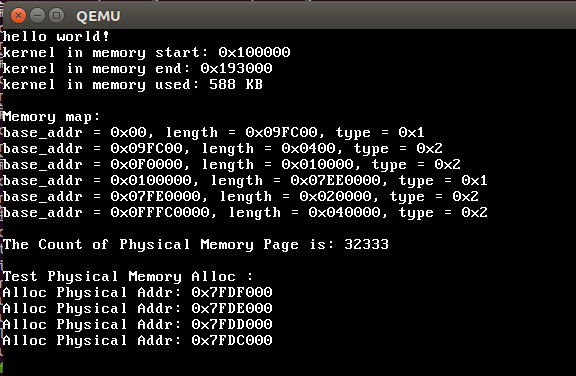
1. #ifndef INCLUDE\_PMM\_H
2. #define INCLUDE\_PMM\_H
3. #include "multiboot.h"
4. // 线程栈的大小
5. #define STACK\_SIZE 8192
6. // 支持的最大物理内存(512MB)
7. #define PMM\_MAX\_SIZE 0x20000000
8. // 物理内存页框大小，4KB
9. #define PMM\_PAGE\_SIZE 0x1000
10. // 最多支持的物理页面个数
11. #define PAGE\_MAX\_SIZE (PMM\_MAX\_SIZE/PMM\_PAGE\_SIZE)
12. // 页掩码 按照 4096 对齐地址
13. #define PHY\_PAGE\_MASK 0xFFFFF000
14. #define error\_detect(x,info) \
15. do { \
16. if (!(x)) { \
17. printk("\*\*\* memory error: %s\n", (char\*)info); \
18. while(1) \
19. } \
20. } while (0)
21. // 内核文件在内存中的起始和结束位置
22. // 在链接器脚本中要求链接器定义
23. extern uint8\_t kern\_start[];
24. extern uint8\_t kern\_end[];
25. // 动态分配物理内存页的总数
26. extern uint32\_t phy\_page\_count;
27. // 输出 BIOS 提供的物理内存布局
28. void show\_memory\_map();
29. // 初始化物理内存管理
30. void init\_pmm();
31. // 返回一个内存页的物理地址
32. uint32\_t pmm\_alloc\_page();
33. // 释放申请的内存
34. void pmm\_free\_page(uint32\_t p);
35. #endif // INCLUDE\_PMM\_H

都很简单是吧。我们直接修改entry.c，看一下效果吧：

/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. init\_timer(200);
15. //asm volatile("sti");
16. printk("kernel in memory start: 0x%x\n",kern\_start);
17. printk("kernel in memory end: 0x%x\n",kern\_end);
18. printk("kernel in memory used: %d KB\n\n",(kern\_end-kern\_start+1023)/1024);
20. show\_memory\_map();
21. init\_pmm();
23. printk("\nThe Count of Physical Memory Page is: %d\n\n", phy\_page\_count);
24. uint32\_t allc\_addr = NULL;
25. printk("Test Physical Memory Alloc :\n");
26. allc\_addr = pmm\_alloc\_page();
27. printk("Alloc Physical Addr: 0x%x\n", allc\_addr);
28. allc\_addr = pmm\_alloc\_page();
29. printk("Alloc Physical Addr: 0x%x\n", allc\_addr);
30. allc\_addr = pmm\_alloc\_page();
31. printk("Alloc Physical Addr: 0x%x\n", allc\_addr);
32. allc\_addr = pmm\_alloc\_page();
33. printk("Alloc Physical Addr: 0x%x\n", allc\_addr);
34. return 0;
35. }

运行结果如下：



可以看出来，我们每调用一次pmm\_alloc\_page();内存就会减少4K。

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

├── kernel

│   └── print.c

├── libs

│   ├── common.c

│   └── string.c

├── Makefile

├── pmm

│   └── pmm.c

└── scripts

├── gdbinit

└── kernel.ld

9 directories, 27 files