第十二章

前面基本上实现了虚拟内存以及分页，但前面的内存分配都是按照页进行分配的，这样比较容易造成内存碎片。我们这一章来实现一个好一点的分配函数，也就是堆管理。

我们用一个很简单的侵入式链表（intrusive list）来实现一下。

先看看头文件：

/include/heap.h

1. #ifndef INCLUDE\_HEAP\_H\_
2. #define INCLUDE\_HEAP\_H\_
3. #include "types.h"
4. // 堆起始地址
5. #define HEAP\_START 0xE0000000
6. // 内存块管理结构
7. typedef
8. struct header {
9. struct header \*prev; // 前后内存块管理结构指针
10. struct header \*next;
11. uint32\_t allocated : 1; // 该内存块是否已经被申请
12. uint32\_t length : 31; // 当前内存块的长度
13. } header\_t;
14. // 初始化堆
15. void init\_heap();
16. // 内存申请
17. void \*kmalloc(uint32\_t len);
18. // 内存释放
19. void kfree(void \*p);
20. // 测试内核堆申请释放
21. void test\_heap();
22. #endif // INCLUDE\_HEAP\_H\_

内容挺简单的，但注意在管理结构里我们用到了位域定义，uint32\_t allocated : 1意思是只使用它1个bit，uint32\_t length : 31是说使用它的31个bit，这样两个定义加起来正好32bit，既方便了变量的调用，同时也能节省空间。这里我们把虚拟堆的起始地址定义在了0xE000 0000，这个是内核没有用到的空闲区域，大概在3.5G以上。

接下来看具体实现，这里除了上面的接口函数，还包括一些内部函数：

/heap/heap.c

1. #include "print.h"
2. #include "pmm.h"
3. #include "vmm.h"
4. #include "heap.h"
5. // 申请内存块
6. static void alloc\_chunk(uint32\_t start, uint32\_t len);
7. // 释放内存块
8. static void free\_chunk(header\_t \*chunk);
9. // 切分内存块
10. static void split\_chunk(header\_t \*chunk, uint32\_t len);
11. // 合并内存块
12. static void glue\_chunk(header\_t \*chunk);
13. static uint32\_t heap\_max = HEAP\_START;
14. // 内存块管理头指针
15. static header\_t \*heap\_first;
16. void init\_heap()
17. {
18. heap\_first = 0;
19. }
20. void \*kmalloc(uint32\_t len)
21. {
22. // 所有申请的内存长度加上管理头的长度
23. // 因为在内存申请和释放的时候要通过该结构去管理
24. len += sizeof(header\_t);
25. header\_t \*cur\_header = heap\_first;
26. header\_t \*prev\_header = 0;
27. //这一部分主要是在释放内存后，再分配时候用到的。
28. while (cur\_header) {
29. // 如果当前内存块没有被申请过而且长度大于待申请的块
30. if (cur\_header->allocated == 0 && cur\_header->length >= len) {
31. // 按照当前长度切割内存
32. split\_chunk(cur\_header, len);
33. cur\_header->allocated = 1;
34. // 返回的时候必须将指针挪到管理结构之后
35. return (void \*)((uint32\_t)cur\_header + sizeof(header\_t));
36. }
37. // 逐次推移指针
38. prev\_header = cur\_header;
39. cur\_header = cur\_header->next;
40. }
41. uint32\_t chunk\_start;
42. // 第一次执行该函数则初始化内存块起始位置
43. // 之后根据当前指针加上申请的长度即可
44. if (prev\_header) {
45. chunk\_start = (uint32\_t)prev\_header + prev\_header->length;
46. } else {
47. chunk\_start = HEAP\_START;
48. heap\_first = (header\_t \*)chunk\_start;
49. }
50. // 检查是否需要申请内存页
51. alloc\_chunk(chunk\_start, len);
52. cur\_header = (header\_t \*)chunk\_start;
53. cur\_header->prev = prev\_header;
54. cur\_header->next = 0;
55. cur\_header->allocated = 1;
56. cur\_header->length = len;
58. if (prev\_header) {
59. prev\_header->next = cur\_header;
60. }
61. return (void\*)(chunk\_start + sizeof(header\_t));
62. }
63. void kfree(void \*p)
64. {
65. // 指针回退到管理结构，并将已使用标记置 0
66. header\_t \*header = (header\_t\*)((uint32\_t)p - sizeof(header\_t));
67. header->allocated = 0;
68. // 粘合内存块
69. glue\_chunk(header);
70. }
71. void alloc\_chunk(uint32\_t start, uint32\_t len)
72. {
73. // 如果当前堆的位置已经到达界限则申请内存页
74. // 必须循环申请内存页直到有到足够的可用内存
75. while (start + len > heap\_max) {
76. uint32\_t page = pmm\_alloc\_page();
77. map(pgd\_kern, heap\_max, page, PAGE\_PRESENT | PAGE\_WRITE);
78. heap\_max += PAGE\_SIZE;
79. }
80. }
81. void free\_chunk(header\_t \*chunk)
82. {
83. if (chunk->prev == 0) {
84. heap\_first = 0;
85. } else {
86. chunk->prev->next = 0;
87. }
88. // 空闲的内存超过 1 页的话就释放掉
89. while ((heap\_max - PAGE\_SIZE) >= (uint32\_t)chunk) {
90. heap\_max -= PAGE\_SIZE;
91. uint32\_t page;
92. get\_mapping(pgd\_kern, heap\_max, &page);
93. unmap(pgd\_kern, heap\_max);
94. pmm\_free\_page(page);
95. }
96. }
97. void split\_chunk(header\_t \*chunk, uint32\_t len)
98. {
99. // 切分内存块之前得保证之后的剩余内存至少容纳一个内存管理块的大小
100. if (chunk->length - len > sizeof (header\_t)) {
101. header\_t \*newchunk = (header\_t \*)((uint32\_t)chunk + len);
102. newchunk->prev = chunk;
103. newchunk->next = chunk->next;
104. newchunk->allocated = 0;
105. newchunk->length = chunk->length - len;
106. chunk->next = newchunk;
107. chunk->length = len;
108. }
109. }
110. void glue\_chunk(header\_t \*chunk)
111. {
112. // 如果该内存块后面有链内存块且未被使用则拼合
113. if (chunk->next && chunk->next->allocated == 0) {
114. chunk->length = chunk->length + chunk->next->length;
115. if (chunk->next->next) {
116. chunk->next->next->prev = chunk;
117. }
118. chunk->next = chunk->next->next;
119. }
120. // 如果该内存块前面有链内存块且未被使用则拼合
121. if (chunk->prev && chunk->prev->allocated == 0) {
122. chunk->prev->length = chunk->prev->length + chunk->length;
123. chunk->prev->next = chunk->next;
124. if (chunk->next) {
125. chunk->next->prev = chunk->prev;
126. }
127. chunk = chunk->prev;
128. }
129. // 假如该内存后面没有链表内存块了直接释放掉
130. if (chunk->next == 0) {
131. free\_chunk(chunk);
132. }
133. }
134. void test\_heap()
135. {
136. printk("Test kmalloc() && kfree() now ...\n\n");
137. void \*addr1 = kmalloc(50);
138. printk("kmalloc 50 byte in 0x%x\n", addr1);
139. void \*addr2 = kmalloc(500);
140. printk("kmalloc 500 byte in 0x%x\n", addr2);
141. void \*addr3 = kmalloc(5000);
142. printk("kmalloc 5000 byte in 0x%x\n", addr3);
143. void \*addr4 = kmalloc(50000);
144. printk("kmalloc 50000 byte in 0x%x\n\n", addr4);
145. printk("free mem in 0x%x\n", addr1);
146. kfree(addr1);
147. printk("free mem in 0x%x\n", addr2);
148. kfree(addr2);
149. printk("free mem in 0x%x\n", addr3);
150. kfree(addr3);
151. printk("free mem in 0x%x\n\n", addr4);
152. kfree(addr4);
153. }

这一部分代码虽然很多，但就是链表的一些操作，如果认真学了数据结构，这个应该不怎么难，或者画一画这一过程，就明白了。毕竟，我总不能做个gif图吧23333.

为了方便，最后那个测试函数我们也直接写在这里了，而不是entry.c。

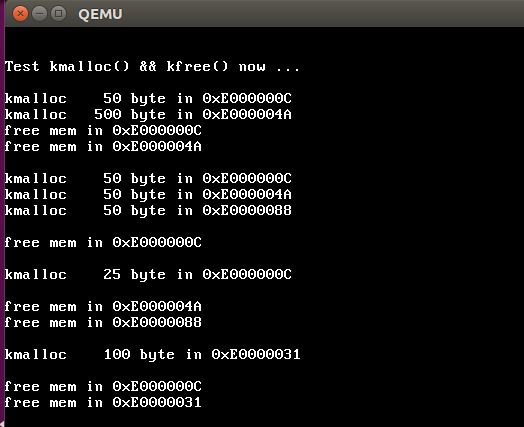
我们修改下entry.c试一试：

/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. #include "heap.h"
10. // 内核初始化函数
11. void kern\_init();
12. // 开启分页机制之后的 Multiboot 数据指针
13. multiboot\_t \*glb\_mboot\_ptr;
14. // 开启分页机制之后的内核栈
15. char kern\_stack[STACK\_SIZE];
16. // 内核使用的临时页表和页目录
17. // 该地址必须是页对齐的地址，内存 0-640KB 肯定是空闲的
18. \_\_attribute\_\_((section(".init.data"))) pgd\_t \*pgd\_tmp = (pgd\_t \*)0x1000;
19. \_\_attribute\_\_((section(".init.data"))) pgd\_t \*pte\_low = (pgd\_t \*)0x2000;
20. \_\_attribute\_\_((section(".init.data"))) pgd\_t \*pte\_hign = (pgd\_t \*)0x3000;
21. // 内核入口函数
22. \_\_attribute\_\_((section(".init.text"))) void kern\_entry()
23. {
24. pgd\_tmp[0] = (uint32\_t)pte\_low | PAGE\_PRESENT | PAGE\_WRITE;
25. pgd\_tmp[PGD\_INDEX(PAGE\_OFFSET)] = (uint32\_t)pte\_hign | PAGE\_PRESENT | PAGE\_WRITE;
26. // 映射内核虚拟地址 4MB 到物理地址的前 4MB
27. int i;
28. for (i = 0; i < 1024; i++) {
29. pte\_low[i] = (i << 12) | PAGE\_PRESENT | PAGE\_WRITE;
30. }
31. // 映射 0x00000000-0x00400000 的物理地址到虚拟地址 0xC0000000-0xC0400000
32. for (i = 0; i < 1024; i++) {
33. pte\_hign[i] = (i << 12) | PAGE\_PRESENT | PAGE\_WRITE;
34. }
36. // 设置临时页表
37. asm volatile ("mov %0, %%cr3" : : "r" (pgd\_tmp));
38. uint32\_t cr0;
39. // 启用分页，将 cr0 寄存器的分页位置为 1 就好
40. asm volatile ("mov %%cr0, %0" : "=r" (cr0));
41. cr0 |= 0x80000000;
42. asm volatile ("mov %0, %%cr0" : : "r" (cr0));
44. // 切换内核栈
45. uint32\_t kern\_stack\_top = ((uint32\_t)kern\_stack + STACK\_SIZE) & 0xFFFFFFF0;
46. asm volatile ("mov %0, %%esp\n\t"
47. "xor %%ebp, %%ebp" : : "r" (kern\_stack\_top));
48. // 更新全局 multiboot\_t 指针
49. glb\_mboot\_ptr = mboot\_ptr\_tmp + PAGE\_OFFSET;
50. // 调用内核初始化函数
51. kern\_init();
52. }
53. void kern\_init()
54. {
55. init\_gdt();
56. init\_idt();
57. console\_clear();
58. printk("hello world!\n");
59. init\_timer(200);
61. //asm volatile("sti");
62. printk("kernel in memory start: 0x%x\n",kern\_start);
63. printk("kernel in memory end: 0x%x\n",kern\_end);
64. printk("kernel in memory used: %d KB\n\n",(kern\_end-kern\_start+1023)/1024);
66. show\_memory\_map();
67. init\_pmm();
68. init\_vmm();
70. printk("\nThe Count of Physical Memory Page is: %d\n\n", phy\_page\_count);
71. uint32\_t allc\_addr = NULL;
73. printk("Test Physical Memory Alloc :\n");
74. allc\_addr = pmm\_alloc\_page();
75. printk("Alloc Physical Addr: 0x%x\n\n\n", allc\_addr);
77. init\_heap();
78. test\_heap();
80. while (1) {
81. asm volatile ("hlt");
82. }
83. }

这里几乎没做什么修改，仅仅加了头文件，堆的初始化以及堆的测试。

看看运行结果：



啊哈，前面的内容都被覆盖了，没关系，前面都是正常的。我们看看我们的测试函数吧：

首先我们尝试分配了50byte和500byte，然后把它们释放。50byte+12byte（我们的header\_t大小是12byte）等于62byte，也就是3E，再加上我们开始的C，3E+C刚好是4A，一起顺利！

下面三个分配不说了，然后我们释放了0xE000000C之后，我们分配了25byte，那么现在应该是C+19（25byte）+C（12byte的header\_t）=31，没错，就是我们的0xE0000031。所以这里的内存分配，释放以及拼合都是正确的。

此时的目录：

1. jf@ubuntu:~/OS\_jf$ tree
2. .
3. ├── boot
4. │   └── boot.s
5. ├── drivers
6. │   ├── console.c
7. │   └── timer.c
8. ├── entry.c
9. ├── floppy.img
10. ├── gdt
11. │   ├── gdt.c
12. │   └── gdt\_s.s
13. ├── heap
14. │   └── heap.c
15. ├── idt
16. │   ├── idt.c
17. │   └── idt\_s.s
18. ├── include
19. │   ├── common.h
20. │   ├── console.h
21. │   ├── gdt.h
22. │   ├── heap.h
23. │   ├── idt.h
24. │   ├── multiboot.h
25. │   ├── pmm.h
26. │   ├── print.h
27. │   ├── string.h
28. │   ├── timer.h
29. │   ├── types.h
30. │   ├── vargs.h
31. │   └── vmm.h
32. ├── kernel
33. │   └── print.c
34. ├── libs
35. │   ├── common.c
36. │   └── string.c
37. ├── Makefile
38. ├── pmm
39. │   └── pmm.c
40. ├── scripts
41. │   ├── gdbinit
42. │   └── kernel.ld
43. └── vmm
44. ├── page\_fault.c
45. └── vmm.c
46. 11 directories, 32 files

http://stackoverflow.com/questions/3361145/intrusive-lists