(一) kmalloc.c

**//void \*kmalloc(size\_t size, gfp\_t flags)**

size是要分配内存的大小

flags指定内存分配时的行为标志，例如GFP\_KERNEL用于可以睡眠的情况，而GFP\_ATOMIC用于不能睡眠的中断上下文。这些标志帮助内核决定如何处理内存不足的情况以及是否允许睡眠等待内存。

**//void kfree(const void \*ptr);**

ptr是之前kmalloc调用返回的指针，指向要释放的内存区域。

**//int printk(const char \*fmt, ...);**

这里，printk函数接受一个格式化字符串（fmt）和一系列变长参数（...），这些参数根据fmt中指定的格式进行格式化输出。与标准C库中的printf类似，它允许插入格式化占位符（如 %d、%s 等）来表示整数、字符串等变量类型。

printk还支持消息级别，允许开发者指定日志消息的严重性。这通常通过在格式字符串前加上一个日志级别宏来实现，例如KERN\_INFO、KERN\_WARNING等，尽管这些宏不是函数声明的一部分，但在使用printk时是常见的实践。

1. #include <linux/module.h>  //提供了定义模块所需的基本元素
2. #include <linux/slab.h>   //包含了kmalloc和kfree函数声明
4. MODULE\_LICENSE("GPL");   //模块遵循的许可协议，这里是GNU General Public License (GPL)
6. unsigned **char** \*kmallocmem1;  //将用来存储通过kmalloc分配的内存地址
7. unsigned **char** \*kmallocmem2;
9. **static** **int** \_\_init mem\_module\_init(**void**)
10. {
11. printk("Start kmalloc!\n");
12. kmallocmem1 = (unsigned **char**\*)kmalloc(1024, GFP\_KERNEL); // 强转成unsigned char\*方便进行字节级操作
13. **if** (kmallocmem1 != NULL){
14. printk(KERN\_ALERT "kmallocmem1 addr = %lx\n", (unsigned **long**)kmallocmem1);
15. }**else**{
16. printk("Failed to allocate kmallocmem1!\n");
17. }
18. kmallocmem2 = (unsigned **char** \*)kmalloc(8192, GFP\_KERNEL);
19. **if** (kmallocmem2 != NULL){
20. printk(KERN\_ALERT "kmallocmem2 addr = %lx\n", (unsigned **long**)kmallocmem2);
21. }**else**{
22. printk("Failed to allocate kmallocmem2!\n");
23. }
24. **return** 0;
25. }
27. **static** **void** \_\_exit mem\_module\_exit(**void**)
28. {
29. kfree(kmallocmem1);  //分别释放之前通过kmalloc分配的两块内存
30. kfree(kmallocmem2);
31. printk("Exit kmalloc!\n");
32. }
34. module\_init(mem\_module\_init);  //注册上述定义的初始化和退出函数
35. module\_exit(mem\_module\_exit);
36. vmalloc.c

**// void \*vmalloc(unsigned long size);**

vmalloc函数接收一个unsigned long size参数，表示请求分配的虚拟内存大小（以字节为单位），并返回一个指向分配的内存区域的指针（类型为void \*）。如果分配失败，它将返回NULL。

**//void vfree(const void \*addr);**

vfree函数接收一个之前由vmalloc分配的内存区域的指针const void \*addr作为参数，并负责释放这块内存。传入的地址应当是有效的，并且是由vmalloc分配的，否则行为是未定义的。

1. #include <linux/module.h>
2. #include <linux/vmalloc.h>
4. MODULE\_LICENSE("GPL");
6. unsigned **char** \*vmallocmem1;
7. unsigned **char** \*vmallocmem2;
8. unsigned **char** \*vmallocmem3;
10. **static** **int** \_\_init mem\_module\_init(**void**)
11. {
12. printk("Start vmalloc!\n");
13. vmallocmem1 = (unsigned **char**\*)vmalloc(8192);   //0x2000
14. **if** (vmallocmem1 != NULL){
15. printk("vmallocmem1 addr = %lx\n", (unsigned **long**)vmallocmem1);
16. }**else**{
17. printk("Failed to allocate vmallocmem1!\n");
18. }
19. vmallocmem2 = (unsigned **char**\*)vmalloc(1048576);  //0x100000
20. **if** (vmallocmem2 != NULL){
21. printk("vmallocmem2 addr = %lx\n", (unsigned **long**)vmallocmem2);
22. }**else**{
23. printk("Failed to allocate vmallocmem2!\n");
24. }
25. vmallocmem3 = (unsigned **char**\*)vmalloc(67108864);  //0x4000000
26. **if** (vmallocmem3 != NULL){
27. printk("vmallocmem3 addr = %lx\n", (unsigned **long**)vmallocmem3);
28. }**else**{
29. printk("Failed to allocate vmallocmem3!\n");
30. }
31. **return** 0;
32. }
34. **static** **void** \_\_exit mem\_module\_exit(**void**)
35. {
36. vfree(vmallocmem1);
37. vfree(vmallocmem2);
38. vfree(vmallocmem3);
39. printk("Exit vmalloc!\n");
40. }
42. module\_init(mem\_module\_init);
43. module\_exit(mem\_module\_exit);
44. request\_region.c

**// struct source**

在Linux内核编程的上下文中，struct resource是一种特殊的数据结构，用来描述系统资源，如内存区域、I/O端口范围等。这个结构体通常包含资源的起始地址、结束地址、资源类型以及一些标志位等信息，用于管理系统中的硬件资源分配。

具体到成员，struct resource可能包含但不限于以下字段：

start: 资源的起始地址（例如，内存起始地址或I/O端口起始编号）。

end: 资源的结束地址（例如，内存结束地址或I/O端口结束编号）。

name: 资源的名称，便于识别。

flags: 标志位集合，表示资源的状态和特性，如是否已分配等。

parent, sibling, child: 这些指针用于构成资源树，帮助管理资源之间的层次关系。

**//struct resource \*request\_region(unsigned long start, unsigned long len, const char \*name);**

功能：此函数用于向内核请求一段连续的I/O端口地址范围，从start地址开始，长度为len个端口。name参数是一个描述这段端口用途的字符串，用于调试和错误报告。

返回值：如果成功，返回一个指向struct resource的指针；如果失败（比如端口已经被其他驱动占用），则返回NULL。

用途：确保某个驱动程序独占使用一段I/O端口，防止资源冲突。

**//void release\_region(unsigned long start, unsigned long len);**

功能：释放之前通过request\_region申请的I/O端口地址范围。参数start和len需与之前调用request\_region时使用的相同。

用途：当驱动程序不再需要使用某段I/O端口时，应当调用此函数释放资源，以便其他驱动或后续的请求可以使用这些端口。

1. #include <linux/module.h>
2. #include <asm/io.h>
3. #include <linux/ioport.h>
5. MODULE\_LICENSE("GPL");
7. **struct** resource \* myregion;
9. **static** **int** \_\_init mem\_module\_init(**void**)
10. {
11. printk("Start request region!\n");
13. myregion = request\_region(22222, 10,"ve");
15. **if** (myregion != NULL){
16. printk("it's ok for %lld .",myregion->start);
18. }**else**{
19. printk("Failed to request region!\n");
20. }
22. **return** 0;
23. }
25. **static** **void** \_\_exit mem\_module\_exit(**void**)
26. {
27. release\_region(22222, 10);
28. printk("Exit request\_region!\n");
29. }
31. module\_init(mem\_module\_init);
32. module\_exit(mem\_module\_exit);

(四) request\_mem\_region.c

**//int request\_mem\_region(unsigned long start, unsigned long len, const char \*name);**

功能：该函数用于请求分配一段连续的物理内存区域，从start地址开始，长度为len个字节。name参数是一个描述这片内存区域用途的字符串，有助于调试和资源跟踪。此调用会检查所请求的内存区域是否已被其他设备或驱动占用，如果没有被占用，则标记这片区域为已分配状态，防止其他请求重叠使用。

返回值：成功时返回0，失败（例如区域已经被占用）则返回非零值

**//void release\_mem\_region(unsigned long start, unsigned long len);**

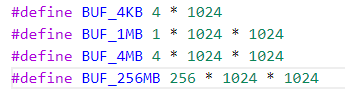
功能：释放之前通过request\_mem\_region分配的内存资源区域。参数start和len必须与当初分配时使用的相同。调用此函数表明驱动程序不再需要访问这片内存区域，从而允许其他驱动或组件重新分配和使用它。

1. #include <linux/module.h>
2. #include <asm/io.h>
3. #include <linux/ioport.h>
5. MODULE\_LICENSE("GPL");
7. **struct** resource \* myregion;
9. **static** **int** \_\_init mem\_module\_init(**void**)
10. {
11. printk("Start request men region!\n");
13. myregion = request\_mem\_region(994115584, 10, "mem\_module");
15. **if** (myregion != NULL){
16. printk("it's ok for %lld .",myregion->start);
18. }**else**{
19. printk("Failed to request region!\n");
20. }
22. **return** 0;
23. }
25. **static** **void** \_\_exit mem\_module\_exit(**void**)
26. {
27. release\_mem\_region(994115584, 10);
28. printk("Exit request\_men\_region!\n");
29. }
31. module\_init(mem\_module\_init);
32. module\_exit(mem\_module\_exit);

(五)emalloc.c

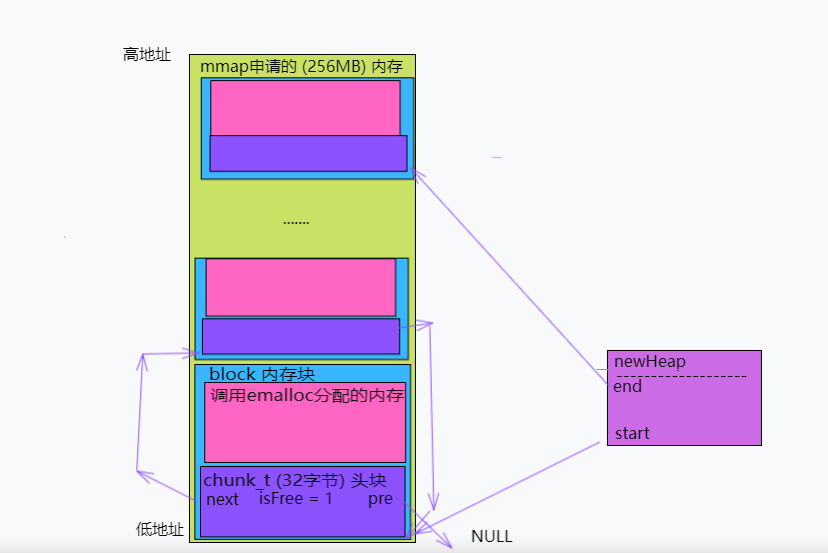
任务要求是编写自己的内存管理小程序，要求设计实现一个简单的内存管理程序，能够在256MB 的内存块内，支持 4KB 内存、1MB 内存、 4MB 内存三类固定长度内存管理的频繁分配和回收。 **（UBuntu20.04系统）**

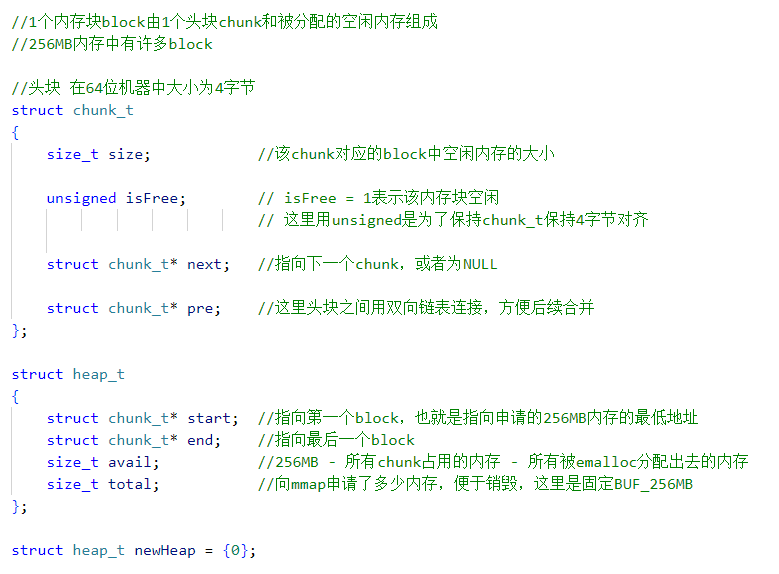
1) 常数定义



这里先定义一下要用到的常数256MB、4KB、1MB、4MB

2）相关数据结构

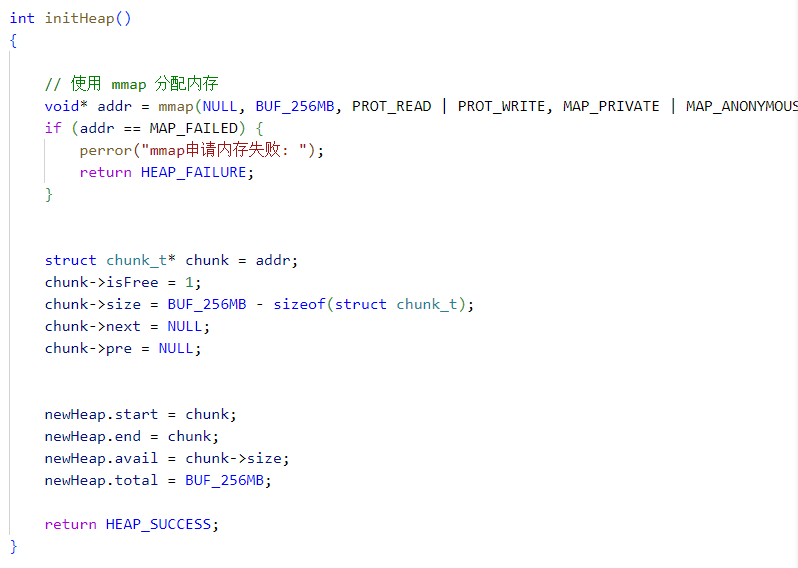


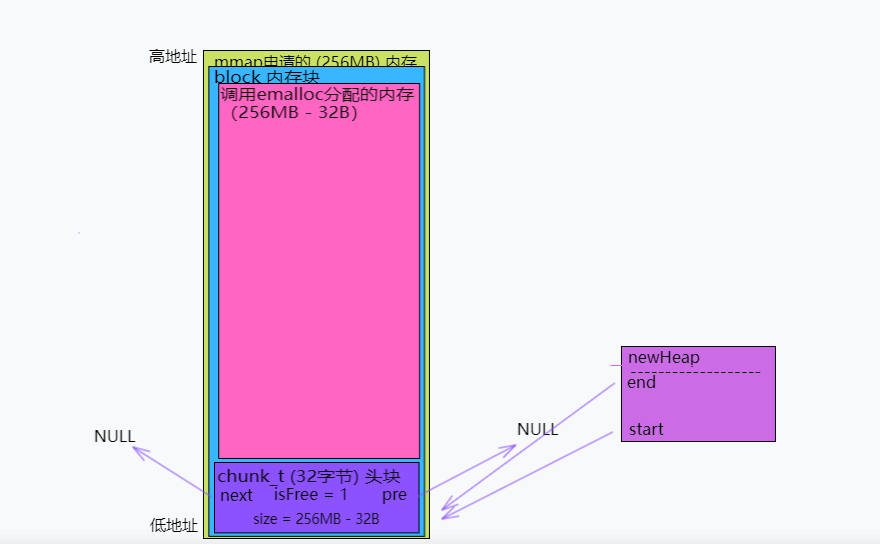


这里需要注意，这里使用的是双向链表连接chunk，方便后续efree的时候合并。

链表并不是空闲链表，而是所以chunk（Used和Free都有）连接而成，保证newHeap.start永远指向256MB的最低地址，newHeap.end永远指向256MB的最高地址，中间就像糖葫芦一样顺序连接，各个block在其中是顺序的，这里更像是由不同长度元素组成的数组。

1. 初始化newHeaap

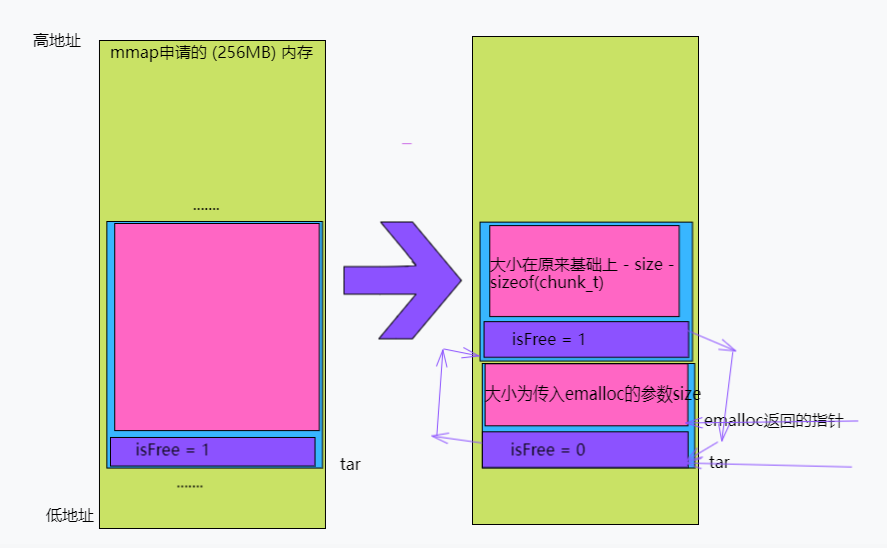




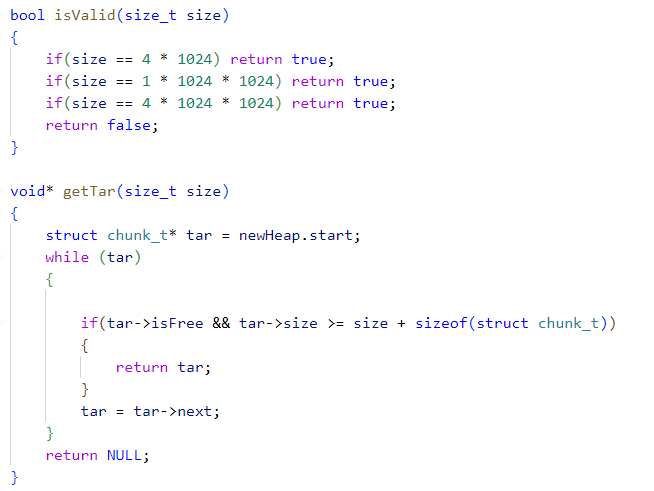
初始化完了之后newHeap中只剩下1个clock，还剩下可用内存256MB-32B

1. Emalloc





这里给出判断输入内存是不是规定的4KB、1MB、4MB的isvild函数和找到目标blcok的getTar函数



5）efree

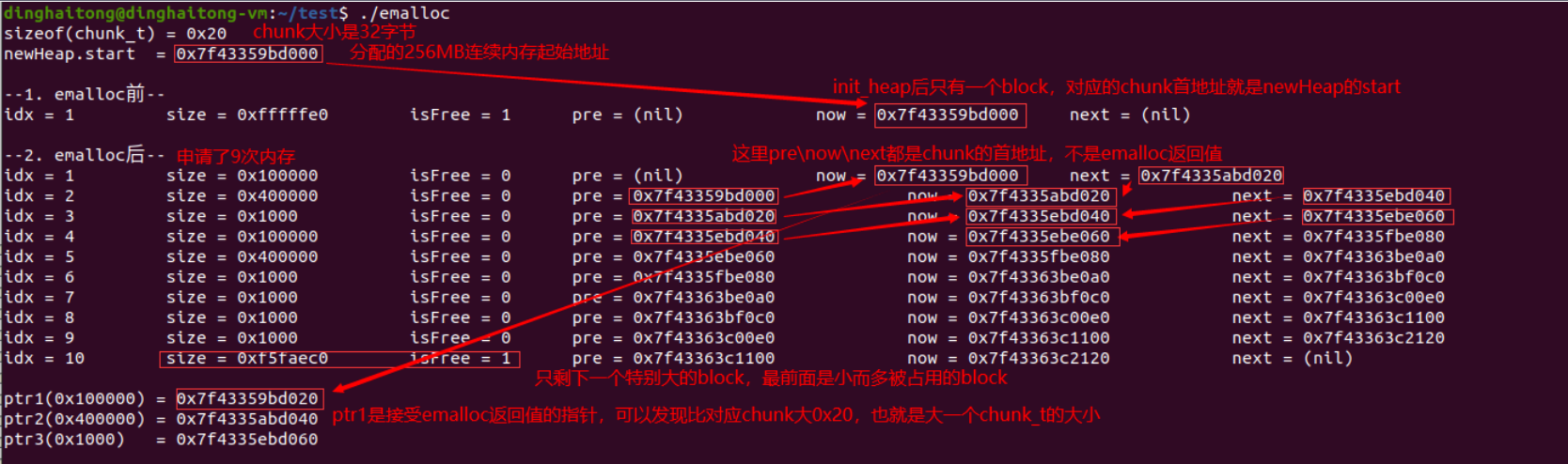
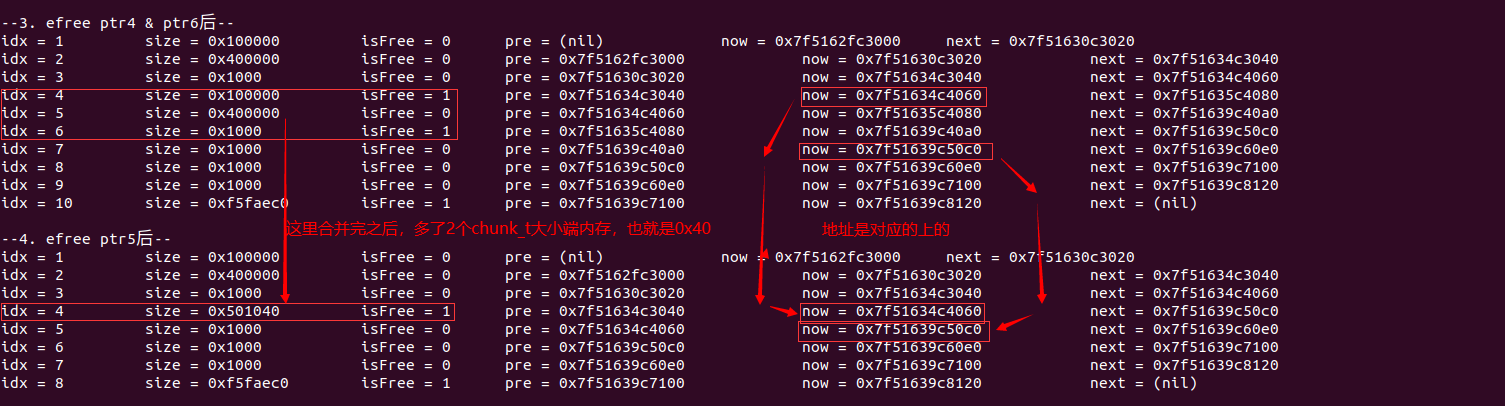


1. 验证





调用测试函数，查看结果



这里给出全部代码：

1. #include <stdio.h>
2. #include <sys/mman.h>
3. #include <stdbool.h>
5. #define BUF\_4KB 4 \* 1024
6. #define BUF\_1MB 1 \* 1024 \* 1024
7. #define BUF\_4MB 4 \* 1024 \* 1024
8. #define BUF\_256MB 256 \* 1024 \* 1024
10. #define HEAP\_SUCCESS 1
11. #define HEAP\_FAILURE -1
13. //1个内存块block由1个头块chunk和被分配的空闲内存组成
15. //头块 在64位机器中大小为4字节
16. **struct** chunk\_t
17. {
18. **size\_t** size;            //该chunk对应的block中空闲内存的大小
20. unsigned isFree;        // isFree = 1表示该内存块空闲
21. // 这里用unsigned是为了保持chunk\_t保持4字节对齐
23. **struct** chunk\_t\* next;   //指向下一个chunk，也就是
25. **struct** chunk\_t\* pre;    //这里头块之间用双向链表连接，方便后续合并
26. };
28. **struct** heap\_t
29. {
30. **struct** chunk\_t\* start;
31. **struct** chunk\_t\* end;
32. **size\_t** avail;
33. **size\_t** total;
34. };
36. **struct** heap\_t newHeap = {0};
38. **bool** isValid(**size\_t** size)
39. {
40. **if**(size == 4 \* 1024) **return** **true**;
41. **if**(size == 1 \* 1024 \* 1024) **return** **true**;
42. **if**(size == 4 \* 1024 \* 1024) **return** **true**;
43. **return** **false**;
44. }
46. **void**\* getTar(**size\_t** size)
47. {
48. **struct** chunk\_t\* tar = newHeap.start;
49. **while** (tar)
50. {
52. **if**(tar->isFree && tar->size >= size + **sizeof**(**struct** chunk\_t))
53. {
54. **return** tar;
55. }
56. tar = tar->next;
57. }
58. **return** NULL;
59. }
61. **int** initHeap()
62. {
64. // 使用 mmap 分配内存
65. **void**\* addr = mmap(NULL, BUF\_256MB, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);
66. **if** (addr == MAP\_FAILED) {
67. perror("mmap申请内存失败: ");
68. **return** HEAP\_FAILURE;
69. }

72. **struct** chunk\_t\* chunk = addr;
73. chunk->isFree = 1;
74. chunk->size = BUF\_256MB - **sizeof**(**struct** chunk\_t);
75. chunk->next = NULL;
76. chunk->pre = NULL;

79. newHeap.start = chunk;
80. newHeap.end = chunk;
81. newHeap.avail = chunk->size;
83. **return** HEAP\_SUCCESS;
84. }
86. **void**\* emalloc(**size\_t** size)
87. {
88. **if** (!isValid(size))
89. {
90. printf("对不起，您申请的空间不满足要求，仅支持4KB、1MB、4MB大小的内存申请\n");
91. **return** NULL;
92. }
93. **if** (size > newHeap.avail)
94. {
95. printf("对不起，您申请的空间已经超过已有空闲空间\n");
96. **return** NULL;
97. }
99. **struct** chunk\_t\* tar = getTar(size);
100. **if**(tar == NULL)
101. {
102. printf("对不起，已有空间不足\n");
103. **return** NULL;
104. }
106. **struct** chunk\_t\* newChunk = (**struct** chunk\_t\*)(tar + 1 + size/**sizeof**(**struct** chunk\_t) );

109. newChunk->size = tar->size - size - **sizeof**(**struct** chunk\_t);
110. newChunk->isFree = 1;
111. tar->size = size;
112. tar->isFree = 0;
114. newChunk->next = tar->next;
115. **if**(tar != newHeap.end)
116. {
117. tar->next->pre = newChunk;
118. }
119. **else**
120. {
121. newHeap.end = newChunk;
122. }
124. tar->next = newChunk;
125. newChunk->pre = tar;
127. newHeap.avail -= **sizeof**(**struct** chunk\_t) + size;
129. **return** (**void**\*)(tar + 1);

132. }
134. **void** efree(**void**\* ptr)
135. {
136. //这里由于ptr是void\*，做地址乘法时，基数是1
137. **struct** chunk\_t\* tar = (**struct** chunk\_t\*)(ptr - **sizeof**(**struct** chunk\_t));
139. newHeap.avail += tar->size + **sizeof**(**struct** chunk\_t);
141. **struct** chunk\_t\* n = tar->next;
142. **struct** chunk\_t\* p = tar->pre;
143. tar->isFree = 1;
145. **if**(n->isFree)
146. {
147. tar->size += n->size + **sizeof**(**struct** chunk\_t);
148. n->next->pre = tar;
149. tar->next = n->next;
150. }
152. **if**(p->isFree)
153. {
154. p->size += tar->size + **sizeof**(**struct** chunk\_t);
155. tar->next->pre = p;
156. p->next = tar->next;
157. }
159. }
161. **void** eprint()
162. {
163. **struct** chunk\_t\* tar = newHeap.start;
164. **int** idx = 1;
165. **while** (tar)
166. {
167. printf("idx = %d \tsize = %#lx   \tisFree = %u\tpre = %p     \t now = %p   \t next = %p\n", idx++, tar->size, tar->isFree, tar->pre, tar, tar->next);
168. tar = tar->next;
169. }
170. printf("\n");
172. }
174. **void** test() // 测试emalloc
175. {
176. printf("sizeof(chunk\_t) = %#x\n", (**int**)**sizeof**(**struct** chunk\_t));
177. printf("newHeap.start  = %p\n\n", newHeap.start);
179. printf("--1. emalloc前--\n");
180. eprint();
182. printf("--2. emalloc后--\n");
184. **void**\* ptr1 = emalloc(BUF\_1MB);
185. **void**\* ptr2 = emalloc(BUF\_4MB);
186. **void**\* ptr3 = emalloc(BUF\_4KB);
187. **void**\* ptr4 = emalloc(BUF\_1MB);
188. **void**\* ptr5 = emalloc(BUF\_4MB);
189. **void**\* ptr6 = emalloc(BUF\_4KB);
190. **void**\* ptr7 = emalloc(BUF\_4KB);
191. **void**\* ptr8 = emalloc(BUF\_4KB);
192. **void**\* ptr9 = emalloc(BUF\_4KB);
193. eprint();
195. printf("ptr1(%#x) = %p\n", BUF\_1MB, ptr1);
196. printf("ptr2(%#x) = %p\n", BUF\_4MB, ptr2);
197. printf("ptr3(%#x)   = %p\n\n", BUF\_4KB, ptr3);
199. printf("--3. efree ptr4 & ptr6后--\n");
200. efree(ptr4);
201. efree(ptr6);
203. eprint();
205. printf("--4. efree ptr5后--\n");
206. efree(ptr5);
207. eprint();





214. }
216. **int** main()
217. {
218. **if**(initHeap() == HEAP\_FAILURE)
219. {
220. printf("堆申请失败\n");
221. **return** -1;
222. }
224. test();
225. munmap(newHeap.start, newHeap.total);
227. **return** 0;
228. }