



本科生实验报告

实验课程: 操作系统原理实验

实验名称: Course Projects (选做)

专业名称: 计算机科学与技术

学生姓名: 熊彦钧

学生学号: 23336266

实验地点: 实验楼 B203

实验成绩:

报告时间: 2025 年 7 月 5 日

Section 1 实验概述

- Malloc/free 的实现

(1) 参考、复现、测试并详细分析 malloc/free 实现的思路和代码（70 分）；

(2) 加分项（20 分）：下面的代码没有做线程/进程同步和互斥处理，因此是线程不安全的。同学们可以加入进行同步互斥的代码，以保证动态内存分配时的线程安全。

Section 2 实验步骤与实验结果

----- 实验任务 1 （对应 assignment1） -----

- 任务要求： 参考、复现、测试并详细分析 malloc/free 的实现思路和代码。

- 实验步骤：首先我们根据实验指导的思路，复现 malloc/free 的实现。

在 lab7 中，我们实现了以页为粒度的动态内存分配机制。这样的机制实现起来是简单的，因为内存分配的基本单位是固定的，我们就无需考虑由于长度不固定的内存分配单元带来的外部碎片等问题。虽然我们通过固定内存分配单元的长度解决了外部碎片的问题，但是由于一个页的长度是 4KB。在一般情况下，我们用不到这么大的内存，因此会产生较多的内部碎片。此时，我们不得不缩小内存分配的粒度来减少内部碎片。所以，我们需要实现以字节为粒度的动态内存分配机制。

前面提到，一个任意长度的内存分配单元是不好管理的。即使我们这里实现的是以字节为粒度的动态内存分配，实际上我们分配的内存单元同样是固定的。只不过这个固定的内存单元不只是一个页，而是若干个固定长度，这些长度都是形如 2^N 的形式，如 16 字节、32 字节、64 字节、128 字节、256 字节、512 字节、1024 字节。这些固定长度的内存单元有一个名字，叫做 arena。此时，当我们希望分配 size 个字节的内存的时候，我们需要找到一个 N，满足

$$2^{N-1} < \text{size} \leq 2^N$$

也就是从小到大搜索，找到第一个恰好不小于 size 的 arena。找到这样一个 arena 后，我们便返回 arena 的起始地址作为分配的结果。特别地，当没有 arena 能够包含 size 个字节时，我们就分配连续的 M 个页，使得

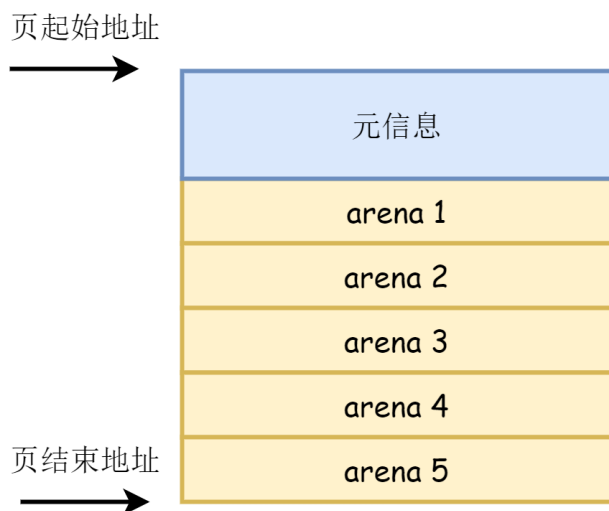
$$(M-1) \times 4096 < \text{size} \leq M \times 4096$$

返回这 M 个页的起始地址作为分配的结果。

以上就是实现以字节为粒度动态内存分配的基本思想，我们现在来实现之。首先，我们需要定义 arena 的类型（在 include/ByteMemoryManager.h 中）。

```
6 enum ArenaType
7 {
8     ARENA_16,
9     ARENA_32,
10    ARENA_64,
11    ARENA_128,
12    ARENA_256,
13    ARENA_512,
14    ARENA_1024,
15    ARENA_MORE
16 };
```

现在，我们需要解决的问题是如何在页内存分配的基础上分配出 arena。不难想到，我们可以把一个页划分成一个个的 arena。但是，arena 也是有大小之分的。为了方便管理，我们不妨规定一个页中划分出来的 arena 的大小都是相同的。此时，给定一个页的地址，我们需要知道这个页中划分出来的 arena 的大小。所以，我们需要在这个页的开头保存一些元信息，如 arena 的类型，可分配的 arena 的数量等，如下所示。



元信息定义如下（在 include/ByteMemoryManager.h 中）。

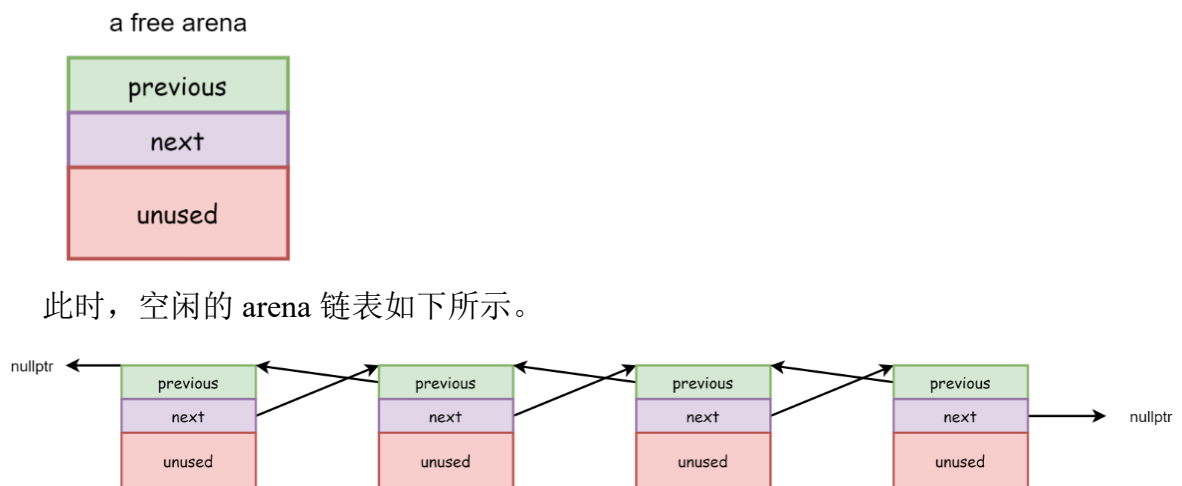
```
18 struct Arena
19 {
20     ArenaType type; // Arena的类型
21     int counter; // 如果是ARENA_MORE, 则counter表明页的数量,
22                 // 否则counter表明该页中的可分配arena的数量
23 };
```

在一开始分配 `size` 长度的字节的时候，内核中是不存在任何可分配的 `arena` 的。此时，我们需要申请一个页，然后在这个页的开头写入元信息，此时，该页中可以分配的内存大小减小到 `4096 - sizeof(Arena)`。

接着，我们计算出 `size` 长度对应的 `arena` 的大小，然后将页的剩余内存划分成相等大小的 `arena`，然后返回第一个 `arena` 的起始地址即可。剩余的这些 `arena` 会被放入一个双向链表中，作为空闲的 `arena`。当空闲 `arena` 链表存在空闲的 `arena` 时，我们直接从这个链表中取出一个 `arena` 返回即可。由于我们的 `arena` 的类型一共有 7 种，从 16 到 1024，因此会有 7 个空闲的 `arena` 链表。既然是一个链表，那么链表中的每一项都需要包含指向链表下一项的指针和指向上一项的指针，即 `MemoryBlockListItem`（在 `include/ByteMemoryManager.h` 中）。

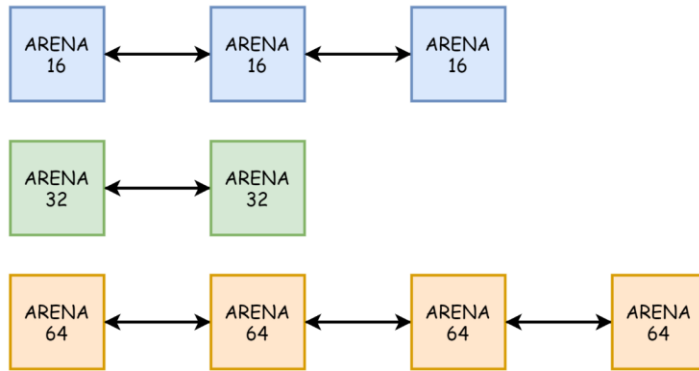
```
25 struct MemoryBlockListItem
26 {
27     MemoryBlockListItem *previous, *next;
28 };
```

`MemoryBlockListItem` 放在哪里呢？既然链表中存放的 `arena` 是空闲的，那么我们就把这 `MemoryBlockListItem` 放到每一个空闲的 `arena` 中，这样我们就无需额外地开辟空间存放链表了，如下所示。



此时，空闲的 `arena` 链表如下所示。

每一种类型的 `arena` 都会有一个空闲链表，如下所示。



以上便是内存分配的思路，为了实现方便，我们不妨定义一个类
MemoryManager 来做动态内存分配（在 include/ByteMemoryManager.h 中）。

```

// MemoryManager是在内核态调用的内存管理对象
class ByteMemoryManager
{
private:
    // 16, 32, 64, 128, 256, 512, 1024
    static const int MEM_BLOCK_TYPES = 7; // 内存块的类型数目
    static const int minSize = 16; // 内存块的最小大小
    int arenaSize[MEM_BLOCK_TYPES]; // 每种类型对应的内存块大小
    MemoryBlockListItem *arenas[MEM_BLOCK_TYPES]; // 每种类型的arena内存块的指针

public:
    ByteMemoryManager();
    void initialize();
    void *allocate(int size); // 分配一块地址
    void release(void *address); // 释放一块地址

private:
    bool getNewArena(AddressPoolType type, int index);
};

```

以下是初始化函数（在 src/utils/ByteMemoryManager.cpp 中）。

```

11 ByteMemoryManager kernelByteMemoryManager;
12
13 ByteMemoryManager::ByteMemoryManager()
14 {
15     initialize();
16 }
17
18 void ByteMemoryManager::initialize()
19 {
20     int size = minSize;
21     for (int i = 0; i < MEM_BLOCK_TYPES; ++i)
22     {
23         arenas[i] = nullptr;
24         arenaSize[i] = size;
25         size = size << 1;
26     }
27 }

```

以下是内存分配的函数（在 src/utls/ByteMemoryManager.cpp 中，因篇幅问题而省略）。

```
29 void *ByteMemoryManager::allocate(int size)
30 {
31     int index = 0;
32     while (index < MEM_BLOCK_TYPES && arenaSize[index] < size)
33         ++index;
34
35     PCB *pcb = programManager.running;
36     AddressPoolType poolType = (pcb->pageDirectoryAddress) ? AddressPoolType::USER :
AddressPoolType::KERNEL;
37     void *ans = nullptr;
38
39     if (index == MEM_BLOCK_TYPES)
40     {
41         // 上取整
42         int pageAmount = (size + sizeof(Arena) + PAGE_SIZE - 1) / PAGE_SIZE;
43
44         ans = (void*)memoryManager.allocatePages(poolType, pageAmount);
45
46         if (ans)
47         {
48             Arena *arena = (Arena *)ans;
49             arena->type = ArenaType::ARENA_MORE;
50             arena->counter = pageAmount;
51         }
52     }
53     else
54     {
```

首先，函数通过循环查找第一个能满足请求大小(size)的内存块类型(index)，使用预定义的 arenaSize 数组进行比较。

随后，函数通过检查运行中进程的页目录地址，判断是用户空间(USER)还是内核空间(KERNEL)的内存池。

如果请求大小超过所有预定义内存块类型：系统则会①先计算需要的页数（包括 Arena 头结构和向上取整）②然后调用底层页分配器分配连续物理页③设置 Arena 头信息（标记为 ARENA_MORE 类型并记录页数）。

如果请求的大小未超过预定义内存块类型，但是对应索引的 arena 链表为空，则会先调用 getNewArena 获取新的 arena。

当 arena 链表不为空的时候，系统会①从链表头部取出内存块；②更新链表指针（移除取出的块）；③通过地址对齐找到所属 arena，并递减其计数器。

最后，函数返回分配的内存地址指针，失败时返回 nullptr。

如果空闲 arena 链表为空，则需要从内核中分配一个页，写入元信息，然后将该页划分成一个个 arena，有一个专门的函数处理这个过程，叫 getNewArena（在 src/utls/ByteMemoryManager.cpp）。

```

79 bool ByteMemoryManager::getNewArena(AddressPoolType type, int index)
80 {
81     void *ptr = (void*)memoryManager.allocatePages(type, 1);
82
83     if (ptr == nullptr)
84         return false;
85
86     // 内存块的数量
87     int times = (PAGE_SIZE - sizeof(Arena)) / arenaSize[index];
88     // 内存块的起始地址
89     int address = (int)ptr + sizeof(Arena);
90
91     // 记录下内存块的数据
92     Arena *arena = (Arena *)ptr;
93     arena->type = (ArenaType)index;
94     arena->counter = times;
95     // printf("---ByteMemoryManager::getNewArena: type: %d, arena->counter: %d\n", index,
96     arena->counter);
97
98     MemoryBlockListItem *prevPtr = (MemoryBlockListItem *)address;
99     MemoryBlockListItem *curPtr = nullptr;
100     arenas[index] = prevPtr;
101     prevPtr->previous = nullptr;
102     prevPtr->next = nullptr;
103     --times;
104
105     while (times)
106     {
107         address += arenaSize[index];
108         curPtr = (MemoryBlockListItem *)address;
109         prevPtr->next = curPtr;
110         curPtr->previous = prevPtr;
111         curPtr->next = nullptr;
112         prevPtr = curPtr;
113         --times;
114     }
115
116     return true;
117 }

```

getNewArena 函数的大致流程如下：

- ① 内存分配：调用 memoryManager.allocatePages() 分配一页内存（PAGE_SIZE），如果分配失败返回 false；
- ② 初始化 Arena 头信息：计算该页内可容纳的内存块数量：(PAGE_SIZE - Arena 头结构大小)/指定大小的内存块，并设置 Arena 头信息：记录内存块类型和数量；
- ③ 构建内存块链表：从分配的内存区域起始地址（跳过 Arena 头）开始，将剩余空间划分为多个内存块，并将这些内存块通过 MemoryBlockListItem 结构连接成双向链表，最后将链表头指针保存在 Arenas 数组中对应索引位置；
- ④ 返回结果：初始化成功后，函数返回 true。

接下来我们来实现内存释放（在 src/Utils/ByteMemoryManager.cpp 中，由于篇幅问题，这里我们只展示部分代码）。

内存释放非常简单，当我们释放一个动态分配的内存地址时，我们可以找到这个地址所在的页。然后通过这个页开头保存的元信息就能够找到这个地址对应的 arena 类型。最后将这个 arena 放到对应的空闲 arena 链表即可。

特别地，当一个页内的所有 arena 都被释放时，我们需要释放这个页。

```
118 void ByteMemoryManager::release(void *address)
119 {
120     // 由于Arena是按页分配的，所以其首地址的低12位必定0，
121     // 其中划分的内存块的高20位也必定与其所在的Arena首地址相同
122     Arena *arena = (Arena *)((int)address & 0xfffff000);
123
124     if (arena->type == ARENA_MORE)
125     {
126         int address = (int)arena;
127
128         PCB *pcb = programManager.running;
129         AddressPoolType poolType = (pcb->pageDirectoryAddress) ? AddressPoolType::USER :
AddressPoolType::KERNEL;
130
131         memoryManager.releasePages(poolType, address, arena->counter);
132     }
133     else
134     {
135         MemoryBlockListItem *itemPtr = (MemoryBlockListItem *)address;
136         itemPtr->next = arenas[arena->type];
137         itemPtr->previous = nullptr;
138
139         if (itemPtr->next)
140         {
141             itemPtr->next->previous = itemPtr;
142         }
143
144         arenas[arena->type] = itemPtr;
145         ++(arena->counter);
146
147         // 若整个Arena被归还，则清空分配给Arena的页
148         int amount = (PAGE_SIZE - sizeof(Arena)) / arenaSize[arena->type];
149         // printf("---ByteMemoryManager::release---: arena->counter: %d, amount: %d\n", arena-
>counter, amount);
150     }
```

这个函数的大致过程如下：

- ①确定所属 Arena：通过地址的低 12 位清零操作（(int)address & 0xfffff000）找到内存块所属的 Arena 首地址；
- ②函数通过判断，处理不同类型的内存释放：
 - a 如果是大内存块，函数会根据当前进程类型（用户/内核）确定内存池类型，并直接调用 releasePages 释放整个 Arena 占用的所有页面；
 - b 如果是普通类型的内存，函数会将被释放的内存块重新插入对应类型的空闲链表头部，并递增 Arena 的可用块计数器；系统还会根据计算该 Arena 的理论最大块数来检查是否整改 Arena 已空闲，如果该 Arena 所有块均已归还，系统会遍历链表移除所有属于该 Arena 的内存块，并调用 releasePages 释放整个 Arena 占用的页面。
- ③系统还会通过检查当前运行进程的页目录地址确定是用户空间还是内核空间内存，调用函数释放空间相应的内存。

由于进程有自己的内存空间，我们修改 PCB，在 PCB 中加入 ByteMemeoryManager（在 include/thread.h 中）。


```

20 struct PCB
21 {
22     int *stack; // 栈指针, 用于调度时保存esp
23     char name[MAX_PROGRAM_NAME + 1]; // 线程名
24     enum ProgramStatus status; // 线程的状态
25     int priority; // 线程优先级
26     int pid; // 线程pid
27     int ticks; // 线程时间片总时间
28     int ticksPassedBy; // 线程已执行时间
29     ListItem tagInGeneralList; // 线程队列标识
30     ListItem tagInAllList; // 线程队列标识
31
32     int pageDirectoryAddress; // 页目录表地址
33     AddressPool userVirtual; // 用户程序虚拟地址池
34     int parentPid; // 父进程pid
35     int retValue; // 返回值
36
37     ByteMemoryManager byteMemoryManager; // 进程内存管理者
38 };

```

最后实现 malloc 和 free 的系统调用处理函数, 即自行加入两个系统调用 malloc 和 free (在 include/syscall.h 和 src/kernel/syscall.cpp 中)。

```

34 // 第5个系统调用, malloc
35 void *malloc(int size);
36 void *syscall_malloc(int size);
37
38 // 第6个系统调用, free
39 void free(void *address);
40 void syscall_free(void *address);
41
65 void *syscall_malloc(int size)
66 {
67     PCB *pcb = programManager.running;
68     if (pcb->pageDirectoryAddress)
69     {
70         // 每一个进程有自己的ByteMemoryManager
71         return pcb->byteMemoryManager.allocate(size);
72     }
73     else
74     {
75         // 所有内核线程共享一个ByteMemoryManager
76         return kernelByteMemoryManager.allocate(size);
77     }
78 }
79
80 void free(void *address)
81 {
82     asm_system_call(6, (int)address);
83 }
84
85 void syscall_free(void *address)
86 {
87     PCB *pcb = programManager.running;
88     if (pcb->pageDirectoryAddress)
89     {
90         pcb->byteMemoryManager.release(address);
91     }
92     else
93     {
94         kernelByteMemoryManager.release(address);
95     }
96 }

```

其中, kernelByteMemoryManager 定义为一个 ByteMemoryManager 的全局

变量。

随后，我们还要在系统调用注册表中注册这两个函数（在 setup.cpp 中）。

```
124 // 设置4号系统调用
125 systemService.setSystemCall(4, (int)syscall_wait);
126 // 设置5号系统调用
127 systemService.setSystemCall(5, (int)syscall_malloc);
128 // 设置6号系统调用
129 systemService.setSystemCall(6, (int)syscall_free);
```

至此，我们就完成了 malloc/free 函数的实现，接下来我们在 setup.cpp 中编写测试函数，测试 malloc/free 的功能。

```
79 void first_thread(void *arg)
80 {
81     printf("\nstart thread\n");
82     printf("test malloc/free in thread.\n");
83
84     // 先在线程中测试内存分配
85     char *str = (char *)malloc(32);
86     if (str) {
87         printf("Thread successfully malloc memory: 0x%x\n", (uint32)str);
88         free(str);
89         printf("Thread successfully free memory.\n");
90     }
91
92     programManager.executeProcess((const char *)first_process, 1);
93
94     asm_halt();
95 }

31 void first_process()
32 {
33     printf("\nstart process\n");
34     printf("test malloc/free in process\n");
35
36     char *str = (char *)malloc(32);
37     if (str)
38     {
39         printf("Process successfully malloc memory, address: 0x%x\n", (uint32)str);
40
41         const char *message = "23336266 jhinx!";
42         int len = 0;
43         while (message[len]) len++;
44
45         if (len < 32)
46         {
47             for (int i = 0; i <= len; i++)
48             {
49                 str[i] = message[i];
50             }
51         }
52
53         printf("Allocated memory content: %s\n", str);
54         free(str);
55         printf("Process successfully free memory.\n");
56     }
57     else
58     {
59         printf("Process malloc failed\n");
60     }
61
62     // 测试大内存分配
63     printf("Trying to allocate large memory...\n");
64     char *largeStr = (char *)malloc(1024 * 1024); // 1MB
65     if (largeStr)
66     {
67         printf("Process successfully malloc large memory, address: 0x%x\n", (uint32)largeStr);
68         free(largeStr);
69         printf("Process successfully free large memory.\n");
70     }
71     else
72     {
73         printf("Process malloc large memory failed.\n");
74     }
}
```

代码运行结果如下：

```

start thread
test malloc/free in thread.
Thread successfully malloc memory: 0xC0100008
Thread successfully free memory.

start process
test malloc/free in process
Process successfully malloc memory, address: 0x8049008
Allocated memory content: 23336266 jhinx!
Process successfully free memory.
Trying to allocate large memory...
Process successfully malloc large memory, address: 0x8049000
Process successfully free large memory.

```

我们可以看到，系统成功实现内存的分配和释放，并且内存释放后没有出现系统错误。malloc/free 函数实现成功。

----- 实验任务 2 （对应 assignment2） -----

- 任务要求： 加入进行同步互斥的代码，保证动态内存分配时代线程安全。
- 实验步骤：

①分析错误的原因（接下来的代码在 assignment2（error）中）：在上面的代码中，如果我们发现 malloc 和 free 函数没有任何的同步互斥机制，如果多个进程同时申请同一块内存，这块内存会被同时分配给这些进程，从而导致内存错误。

接下来我们复现一下这个可能的错误，我们先修改 ByteMemoryManager::allocate 函数，延长它分配内存的时间，让它在分配内存的过程中被系统调度走：

```

62         // 每次取出内存块链表中的第一个内存块
63         ans = arenas[index];
64
65         //add allocation delay
66         int delay=0xffffffff;
67         while(delay--);
68
69         arenas[index] = ((MemoryBlockListItem *)ans)->next;
70

```

与此同时，我们在 setup.cpp 中多引入一个进程来申请内存，制造两个进程同时申请内存的情况：

```

77 void second_thread(void *arg)
78 {
79     printf("\nsecond thread\n");
80     char *str = (char *)malloc(32);
81     if (str) {
82         printf("Thread 2 successfully malloc memory: 0x%x\n", (uint32)str);
83         free(str);
84         printf("Thread 2 successfully free memory.\n");
85     }
86 }

```

运行程序，我们发现程序会触发中断错误，这说明此时线程中动态内存分配时是不安全的：

```
Unhandled interrupt happened, halt...

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C

start thread
test malloc/free in thread.

second thread
Thread 1 successfully malloc memory: 0xC0100008
Thread 1 successfully free memory.
```

②解决方案（代码在 assignment2 (correct) 中）：回顾学过的知识，我们知道想要解决同步互斥的问题，我们可以引入互斥锁，同一时间只允许一个进程对动态内存进行操作（包括申请和释放），只有进程完全获得内存或者完全释放内存后，才允许下一个进程进行操作。

根据这个思路，我们进行如下修改：

首先我们在 ByteMemoryManager.h 中添加互斥锁的声明：

```
35 private:
36     // 16, 32, 64, 128, 256, 512, 1024
37     static const int MEM_BLOCK_TYPES = 7;           // 内存块的类型数目
38     static const int minSize = 16;                 // 内存块的最小大小
39     int arenaSize[MEM_BLOCK_TYPES];                 // 每种类型对应的内存块大小
40     MemoryBlockListItem *arenas[MEM_BLOCK_TYPES]; // 每种类型的arena内存块的指针
41     SpinLock memoryLock; // mutex锁
```

随后，我们在 ByteMemoryManager.cpp 中实现这个锁：

```
18 void ByteMemoryManager::initialize()
19 {
20     // 初始化mutex锁
21     memoryLock.initialize();
22
23     int size = minSize;
24     for (int i = 0; i < MEM_BLOCK_TYPES; ++i)
25     {
26         arenas[i] = nullptr;
27         arenaSize[i] = size;
28         size = size << 1;
29     }
30 }
```

然后，我们要在 `allocate` 函数和 `release` 函数中调用这个锁，包括加锁和解锁：

对 `allocate` 函数加锁：

```
32 void *ByteMemoryManager::allocate(int size)
33 {
34     // 获取锁，保护整个分配过程
35     memoryLock.lock();
36
37     int index = 0;
38     while (index < MEM_BLOCK_TYPES && arenaSize[index] < size)
39         ++index;
40 }
```

`allocate` 函数有两处地方可以解锁，首先是当大内存分配失败的时候，函数会立刻返回，此时需要解锁以允许后续的进程获得锁：

```
60
61     //printf("---ByteMemoryManager::allocate----\n");
62     if (arenas[index] == nullptr)
63     {
64         if (!getNewArena(poolType, index))
65         {
66             memoryLock.unlock();
67             return nullptr;
68         }
69     }
70 }
```

其次是在函数的末尾，无论动态内存是否分配成功，都需要进行解锁。

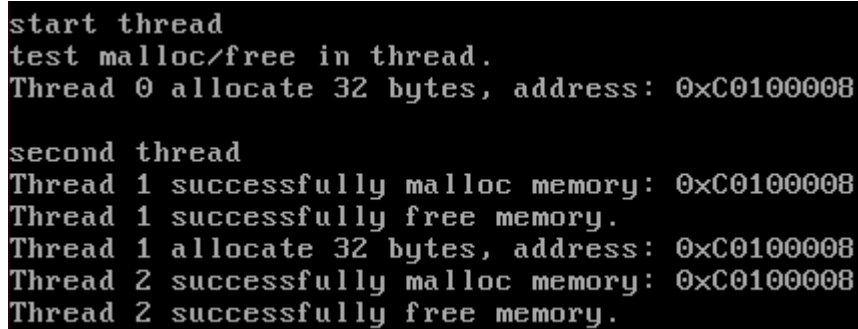
```
88     Arena *arena = (Arena *)((int)ans & 0xfffff000);
89     --(arena->counter);
90     //printf("---ByteMemoryManager::allocate----\n");
91 }
92
93 memoryLock.unlock();
94 return ans;
95 }
```

接下来是 `release` 函数，由于不存在分配失败的情况，只要进程能正常释放内存，最后就会解锁：

```
136 void ByteMemoryManager::release(void *address)
137 {
138     memoryLock.lock();
139
140     // 由于Arena是按页分配的，所以其首地址的低12位必定0，
141     // 其中划分的内存块的高20位也必定与其所在的Arena首地址相同
142     Arena *arena = (Arena *)((int)address & 0xfffff000);
```

```
205
206         memoryManager.releasePages(poolType,(int)arena, 1);
207     }
208 }
209 memoryLock.unlock();
210 }
```

我们运行和刚刚完全一样的测试样例，发现此时系统已经可以正常分配和回收动态内存了：



```
start thread
test malloc/free in thread.
Thread 0 allocate 32 bytes, address: 0xC0100008

second thread
Thread 1 successfully malloc memory: 0xC0100008
Thread 1 successfully free memory.
Thread 1 allocate 32 bytes, address: 0xC0100008
Thread 2 successfully malloc memory: 0xC0100008
Thread 2 successfully free memory.
```

综上所述，我们通过引入互斥锁解决进程共享动态内存的同步互斥问题，进程的安全性得到了提高，实验任务完成。

Section 5 实验总结与心得体会

在这一个选做实验中，我们基于 lab7 的页面内存管理，将动态内存分配的粒度缩小到了一个字节，实现了一个更加强大的动态内存管理机制，这无疑是一个质的飞跃。与此同时，我们还运用了前面学到的锁机制，解决了进程申请动态内存时候可能面临的同步互斥问题，提高了系统的安全性。

至此，本学期的操作系统实验就正式结束了，在接下来的学习中，我们可能还会运用到操作系统实验的有关知识，或者会学习到更高级的操作系统相关知识，希望本学期学到的知识可以作为“巨人的肩膀”，为往后的学习生涯打下坚实的基础。

最后，感谢这学期给过我任何帮助的老师、助教，你们的答疑解惑对我来说无疑是巨大的帮助。

Section 6 附录：代码清单

代码请见附件。