电 子 科 技 大 学

作 业 报 告

学生姓名：邢薪达　　学 号：202222080340　　指导教师：刘杰彦

学生 E-mail：　1756775636@qq.com

Linux 下内存分配和释放

第一章 需求分析

1.1 总体要求：

在 Linux 环境下，采用 C 或 C++语言实现一个堆分配器，即实现内存分配和释放的功能。

1.2 基本要求：

- 实现的内存分配函数名为 MyMalloc，内存释放函数名为 MyFree，这两个函数原型同 malloc 和 free。
- 使用 mmap 函数从内核分配整个堆空间，而 MyMalloc 和 MyFree 管理该空间
  - 即用这个堆空间模拟内存，每次分配和释放在这个空间内进行
  - 如空间不够，再通过 mmap 重新分配堆空间，重复上述工作。
- 要考虑内存碎片问题

- 例如可以考虑采用内存池技术来管理

- 要考虑并发性问题，即多个线程同时分配和释放内存的场景
  - 例如可以采用互斥技术
  - 互斥的粒度不宜过粗，如不宜针对每个 malloc 和 free 直接加锁，而应当将锁的粒度放到对指定内存池元素的分配和回收上

第二章 总体设计

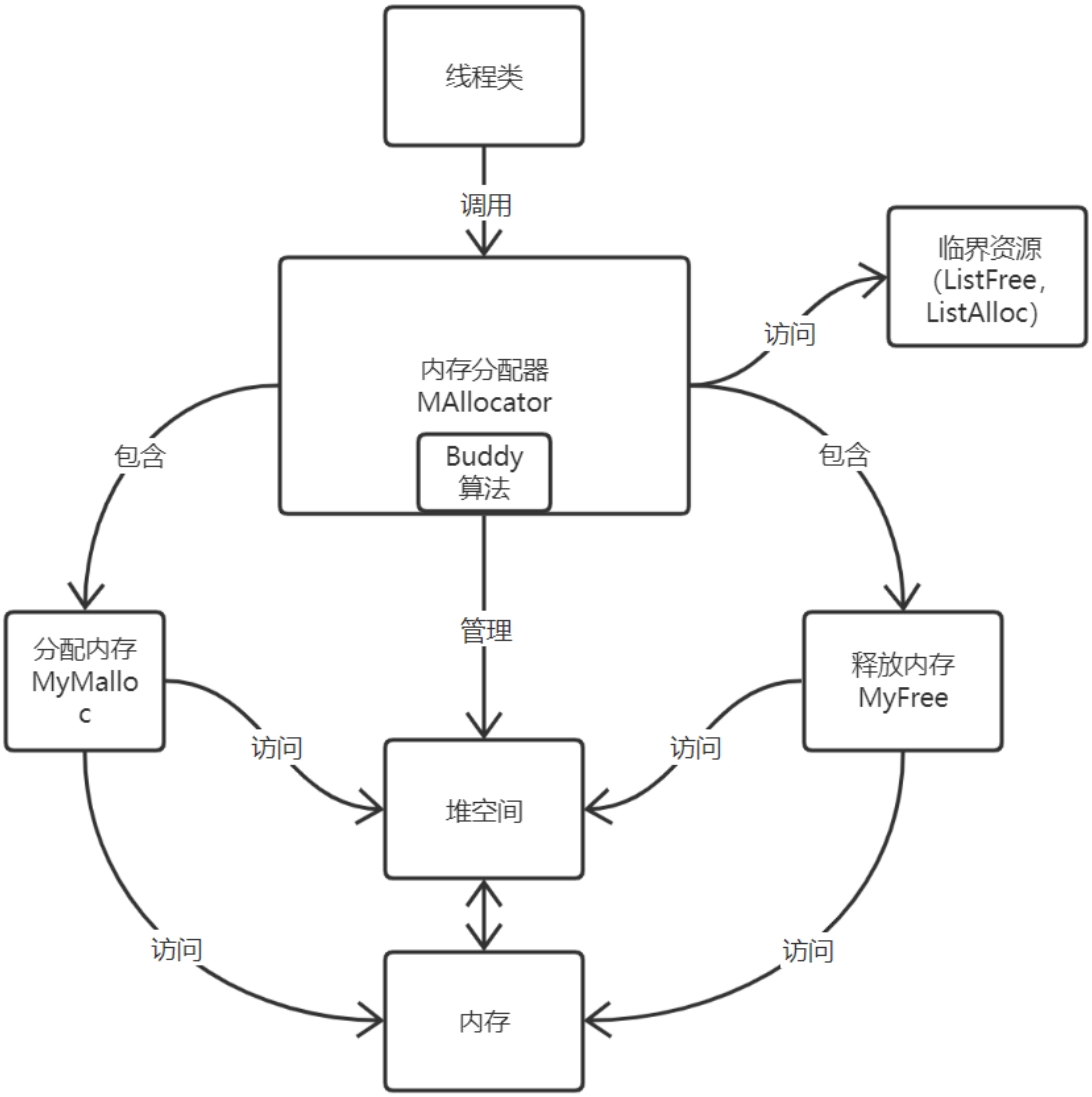
2.1 对作业目标的个人理解

1) 要点 1：第一次分配内存时，从内核一次性 mmap 出 2MB 大小内存空间作为整个堆空间，之后的分配与释放操作都在这个堆空间内进行。
2) 要点 2：经过长时间的分配如果第一次分配的 2MB 空间不足，再次 mmap 2MB 空间大小，即当前共使用了 4MB 内存，依此类推。
3) 要点 3：申请内存超出 256KB 时，直接使用 mmap 分配内存。释放时也直接通过 munmap 释放。
4) 要点 4：经过长时间分配与回收，产生的内存碎片需要合并处理。
5) 要点 5：多个线程可以同时分配和释放内存。（未完成，比较遗憾）

在经过详细阅读学习给出的参考代码 malloc.c，以及结合最后一

堂课老师的讲解与网上的资料，我对内存如何分配管理有了更加深入的理解。最后本人选择采用 buddy 伙伴堆算法进行内存管理。

2.2 程序架构图



第三章 详细设计与实现

3.1 重要的数据结构作为临界区资源

1. Memory 结构体，用来表示 buddy 算法的每一块内存。详细说明

如下图。

```
//内存块地址信息结构体
typedef struct Memory
{
    void * address;//内存块地址
    unsigned long size;//内存块大小
    unsigned long FromMmapBit;//是否是从mmap直接申请的
    Memory* pre;//前一内存块
    Memory* next;//后一内存块
    pthread_mutex_t mutex;//本块互斥访问
    Memory(void * _address, size_t _size) : address(_address), size(_size)
    {
        FromMmapBit = 0;
        pre = nullptr;
        next = nullptr;
        pthread_mutex_init(&mutex, NULL);
    }
}Memory;
```

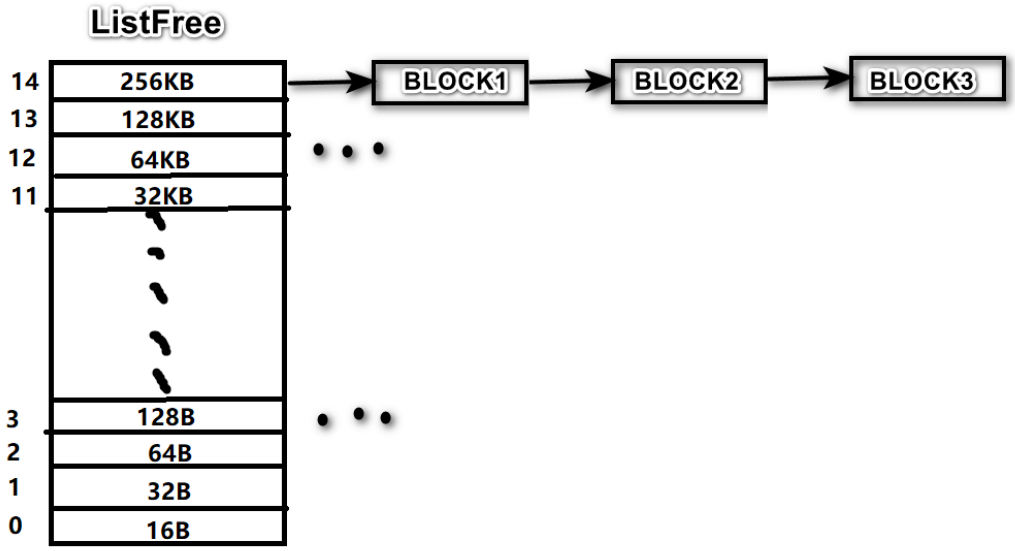
2. 空闲块链表数组 ListFree[15]：使用该逻辑数据结构作为空闲内存块管理实际申请的物理内存。数组的每个元素的链表头结点为 BuddyNode 类型，链表后续结点为 Memory 类型。

```
//buddy空闲内存块数组链表
extern BuddyNode * ListFree[15];//ListFree[0]保存16B 2^4大小的块
```

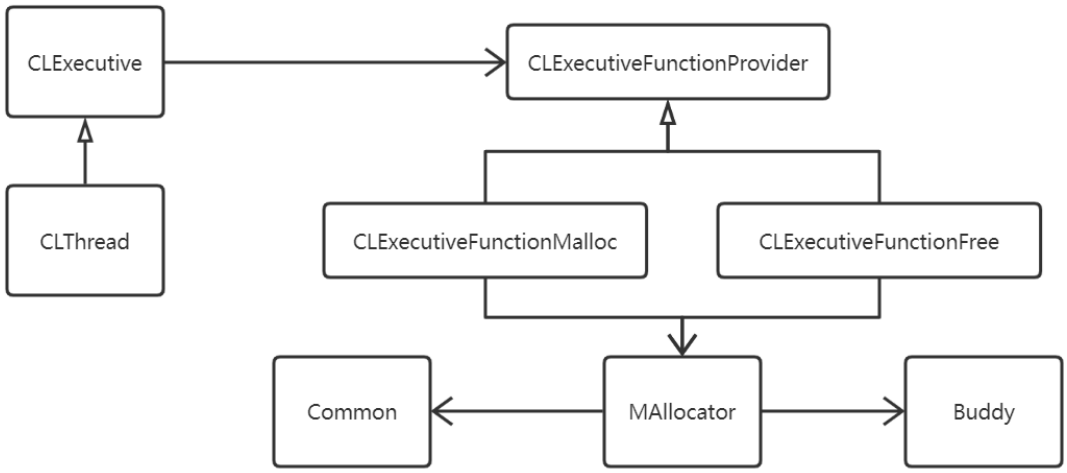
BuddyNode：

```
typedef struct BuddyNode
{
    unsigned long size;
    Memory* head;
}BuddyNode;
```

ListFree 结构示意图。数组元素类型是 BuddyNode，每层使用双向链表连接的空闲块类型是 Memory 类型，



3.2 系统类图



其中 CLThread 课上实现的线程类，继承自 CLExecutive，CLExecutive 依赖于 CLExecutiveFunctionProvider。可以多态性定义不同功能的线程。

CLExecutiveFunctionMalloc，CLExecutiveFunctionMalloc 分别调用 MAllocator 的 MyMalloc 和 MyFree 方法。

MAllocator：内存分配器类，依赖 Buddy 伙伴算法类，以及 Common 全局资源（临界区）类。在该类中实现 MyMalloc 和 MyFree 方法。

3.3 Buddy 内存管理算法的实现。

后面通过自顶向下的叙述方式展开。

1. MyMalloc：封装好的总的内存分配接口函数。

   主要流程：

   a) 排除非法内存大小输入

   b) 判断大小是否大于 256KB 阈值，大于直接通过 mmap 分配，标记是从 mmap 直接分配的，并将该内存块插入 ListAlloc 中。

   c) 判断空闲块链表是否为空，不空则直接在 ListFree 空闲块链表数组上通过 buddy 算法选择合适的内存块并插入 ListAlloc 中。

   d) 空闲块链表为空，则先申请 8 *256KB（2MB）个空闲块，再通过 buddy 算法进行分配。

   关键代码：

```
//申请空间
void* MAllocator::MyMalloc(size_t size,int threadCount)
{
    //单线程测试需要返回内存地址
    void* getAddress = nullptr;
    //非法输入
    if(size <= 0)
        return nullptr;
    //判断大小，超过 256kb 阈值，使用 mmap 直接分配
    else if(size > THRESHOLD_FOR_MMAP)
    {
        //mmap 是多线程安全的，可以在两个线程里同时调用 mmap 这个函数。不用加锁
        int r =addToListAlloc(MallocBymmap(size),&getAddress);
```

```
        if(r == 0)
        {
            return getAddress;
        }
    }
    //否则使用 buddy 空闲块分配
    else
    {
        //是否存在可用 buddy 内存块 =i 意味着第 i 层存在空闲块，根据申请大小判断，
直接从对应大小层往上找
        if(IsListFreeEmpty(size) != 0)
        {
            //使用 buddy 算法分配空闲块
            int r =addToListAlloc(BuddyAllocBlock(size),&getAddress);
            if(r == 0)
            {
                return getAddress;
            }
        }
        //空闲 buddy 内存块不够（初次申请内存，或者多次分配后无可用内存），找不
到空闲 buddy 内存块。
        else
        {
            //mmap 一个 2M 大小的空间，并拆分为 8 个 256kb，buddy 块，插入
ListFree[14]
            BuddyBlockAllocMmap2MB();
            //然后再使用 buddy 算法分配空闲块
            int r =addToListAlloc(BuddyAllocBlock(size),&getAddress);
            if(r == 0)
            {
                return getAddress;
            }
        }
    }
    return nullptr;
}
```

1) MallocBymmap 方法:使用 mmap 申请内存，将内存地址，大小等信息记录到 Memory 结构体中，返回。

```
Memory * MallocBymmap(size_t size)
{
    //以 4KB 为最小单位申请，向上取整
    size = ALIGN_UP_4KB(size);
```

```
    //返回申请到的内存起始地址
   void *addr = mmap(NULL, size, PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS,
-1, 0);
    if(addr == MAP_FAILED)
    {
        //申请失败，内存不足
        printf("MallocBymmap 申请大小{%ld}B 失败\n",size);
        return NULL;
    }
    Memory *p = new Memory(addr,size);
    p->FromMmapBit = 1;
    return p;
};
```

2) **addToListAlloc：将空闲块插入 ListAlloc 链表，简单的单链表插入。过于基础，不浪费篇幅。**

3) **IsListFreeEmpty：判断空闲块表是否有合适的空闲块，并且若第 I 层为空，继续向上一层查找，直到返回有合适块的层号。其中 BELONG_TO_N_LAYER 函数可以根据块大小，快速判断出该块应该属于第 I 层。**

```
//是否存在空闲 buddy 内存块 =i 意味着存在空闲块，根据申请大小判断，直接从对应大
小层往上找
int IsListFreeEmpty(int size)
{
    //对所有访问 ListFree 或者 ListAlloc 的地方上读锁
    //locker->lockRead();
    for(int i = BELONG_TO_N_LAYER(size);i<=14;i++)
    {
        //如果当前层链表为空，继续往上找,否则说明有合适大小的空闲块
        if(ListFree[i]->head->next == nullptr)
            continue;
        else
            return i;
    }
    //所有层都没有合适的空闲块
    return 0;
}
```

4) **BuddyAllocBlock：通过 buddy 算法分配空间**

a) 首先，根据申请块大小查询满足条件的最小层数，并直接取出该层第一块内存 A。

b) 根据当前层 M 和该块大小原本应该在第 N 层，做差 M-N，将 A 逐层分块，最终返回大小最合适的块。

```
//使用 buddy 算法分配空闲块，输入需求的大小。
Memory* BuddyAllocBlock(size_t size)
{
    //直接去第 layer 层找,前面已经判断过必定有合适大小的块,直接取出当前层第一块
    int layer = IsListFreeEmpty(size);
    //把 target 卸下来，分块后删除
    //删除块修改了三个块，当前块，前一块，后一块，都加锁
    Memory* target =  ListFree[layer]->head->next;
    void* address = target->address;
    {
        ListFree[layer]->head->next = target->next;
        //还剩下不止一块时，下一块指向头结点
        if(target->next !=nullptr)
        {
            target->next->pre = ListFree[layer]->head;
        }
        target->next = nullptr;
        target->pre = nullptr;
    }
    //删除一开始的 target 结点
    delete target;
    //申请大小本应在第 origin_layer 层，现在跨了 layer - originLayer 层，向下分块。
    //比如 14 层是 256kb 12 层是 64kb，跨了两层，向下分两次块
    int originLayer = BELONG_TO_N_LAYER(size);
    Memory * blockDivided = nullptr;
    Memory* blockLeft= nullptr;
    for(int i = layer-1; i >= originLayer;i--)
    {
        //分成两块后剩下的一块，连在当前层的头。另一块继续分块
        blockLeft = new Memory(address,ListFree[i]->size);
        //第 2 块地址
        address = (char*)address + ListFree[i]->size;
        //第一块连接本层
       {
            blockLeft->next = ListFree[i]->head->next;
```

```
            //当前层不为空
            if(ListFree[i]->head->next !=nullptr)
                ListFree[i]->head->next->pre = blockLeft;
            ListFree[i]->head->next=blockLeft;
            blockLeft->pre = ListFree[i]->head;
        }
    }

    //0 0   循环 1 块起始地址 0   大小 128  下一块起始地址 addr = 128
    //      循环 2 块起始地址 128 大小 64   下一块起始地址 addr = 192
    //循环结束后,将最终另一块分配出去
    blockDivided = new Memory(address,ListFree[originLayer]->size);

    return blockDivided;
};
```

2. MyFree：封装好的总的 Free 函数。关键代码如下。

   主要流程：

   a) 首先通过输入的内存物理地址在 ListAlloc 链表上寻找对应的块。

   b) 如果该块是通过 mmap 直接分配，再通过 mummap 直接释放。

   c) 否则，使用 buddy 算法合并到 FreeList 表中。

```
//释放内存块，有兄弟就向上合并。如果是 mmap 直接分配的，直接释放;threadCount 记录第几个线程访问
void MAllocator:: MyFree(void* address,int threadCount)
{
    //找到要释放的块
    Memory* findToFree = ListAlloc;
    Memory* prefindToFree = findToFree->pre;
    while(findToFree->next!=nullptr)
    {
        prefindToFree = findToFree;
        //跳过头结点
        findToFree = findToFree->next;
        if(findToFree->address == address)
        {
            //删除块修改了三个块，当前块，前一块，后一块，都加锁
            //从分配链表中删除
            {
```

```
                prefindToFree->next = findToFree->next;
                //还剩下不止一块时，下一块指向头结点
                if(findToFree->next != nullptr)
                {
                    findToFree->next->pre = prefindToFree;
                }
                findToFree->pre = nullptr;
                findToFree->next = nullptr;
            }
            //不是从 mmap 直接获取的
            if(findToFree->FromMmapBit == 0)
            {
                //合并到空闲块链表数组
                int tempSize = findToFree->size;
                int layer = BuddyMergeToFreeList(findToFree);
                return;
            }
            //mmap 直接获取的直接 munmap 掉,且对应的 Memory 块也删除
            else
            {
                int temp = findToFree->size;
                munmap((void*)findToFree->address, findToFree->size);
                delete findToFree;
                return;
            }
        }
    }
}
```

1) **BuddyMergeToFreeList：通过 Buddy 算法合并。**

   主要流程：

   a) **首先判断该块应该插在 FreeList 的第几层**

   b) **判断该块在该层是否存在兄弟块，通过遍历该层内存块结点，通过两两块地址相减，判断是否相差该层块的大小来判断是否是兄弟。**

   c) **如果没有兄弟块，直接插入这一层**

   d) **如果有兄弟，将两个块合并，保留低地址块，删除高地址**

块，并递归调用自身，将合并后的块传到更高一层，继续向上查找是否有兄弟块。

```
//合并到空闲块链表
int BuddyMergeToFreeList(Memory *MemoryToMerge)
{
    //递归调用这部分
    //根据块大小判断是第几层的块
    int layer = BELONG_TO_N_LAYER(MemoryToMerge->size);
    //判断该层是否有兄弟块
    int flag = 0;
    Memory * temp = ListFree[layer]->head;
    while(temp->next != nullptr)
    {
        temp = temp->next;
        long x = (char *)(MemoryToMerge->address)-(char *)(temp->address);
        if((x < 0)&& (x+ListFree[layer]->size) == 0)
        {
            flag = 1;
            break;
        }
        //temp 是低地址块
        else if((x > 0)&& (x-ListFree[layer]->size) == 0)
        {
            flag = 2;
            break;
        }
    }
    //没有，直接头插，插入该层，返回
    //或者到第 14 层也应该直接插入
    if(flag == 0||layer ==14)
    {
        //MemoryToMerge 是低地址块
        Memory * temp = ListFree[layer]->head;
        {

            MemoryToMerge->next = temp->next;
            if(temp->next !=nullptr)
                temp->next->pre = MemoryToMerge;
            temp->next = MemoryToMerge;
            MemoryToMerge->pre = temp;
        }
        return layer;
```

```
    }
    //有兄弟块，保留低地址块结点，删除高地址块结点，修改低地址块结点大小，将其
插入下一层
    else
    {
        //不论 temp 是不是低地址块都要先把 temp 从 layer 层卸下来
        Memory * tempPre =temp->pre;
        tempPre->next = temp->next;
        if(temp->next != nullptr )
        {
            temp->next->pre = tempPre;
        }
        temp->pre = nullptr;
        temp->next = nullptr;
        //(flag == 1) MemoryToMerge 是低地址块,，后将 MemoryToMerg 大小扩大后
送入下一层
        if(flag == 1)
        {

            MemoryToMerge->size += temp->size;
            delete temp;
            return BuddyMergeToFreeList(MemoryToMerge);
        }
        //(flag == 2) temp 是低地址块，大小扩大后送入下一层，删除 MemoryToMerge
        else if(flag == 2)
        {
            temp->size += MemoryToMerge->size;
            delete MemoryToMerge;
            return BuddyMergeToFreeList(temp);
        }
    }
    return -1;
};
```

3.4 多线程交互管理

因时间紧张，能力有限等原因，互斥量机制调试失败，最终选择以多线程安全的 **MAllocator** 单例实现伪并发（虽是多个线程同时在运行，实际上还是串行执行的，性能损失极大）。

MAllocator 单例类如下，通过 **getInstance** 时使用互斥量，保证即使多线程也只有一个 **MAllocator** 实例。

```
class MAllocator
{
public:
    static MAllocator *getInstance();//获取MAllocator对象指针
    void* MyMalloc(size_t size,int threadCount);//申请空间
    void MyFree(void* address,int threadCount); //释放空间
private:
    MAllocator();
    ~MAllocator();
    static pthread_mutex_t *getCreatingMAllocatorMutex();//获取创建内存分配器的互斥量
private:

    //用于MAllocator实例的创建，保证只创建一个MAllocator实例
    static pthread_mutex_t *m_pMutexForCreatingMAllocator;
    static MAllocator* m_MAllocator;


};
```

第四章 测试

需要说明测试方案、测试代码的设计实现方案，以及测试结果，如截图等；注意测试结果中必须有截图，截图中必须包括主机名，主机名必须是自己姓名的汉语拼音

4.1 测试环境

```
(base) xingxinda@xingxinda:~/public/LinuxCode/work$ cat /proc/version
Linux version 5.15.68.1-microsoft-standard-WSL2 (oe-user@oe-host) (x86_64-msft-linux-gcc
 (GCC) 9.3.0, GNU ld (GNU Binutils) 2.34.0.20200220) #1 SMP Mon Sep 19 19:14:52 UTC 2022
```

4.2 测试方案：

1. 方案一：MyMalloc 单线程申请 5 块，大小分别为：16B,256B，16KB，128KB，257KB。
    a) 申请结果，其中 257KB 直接通过 mmap 进行分配。

```
(base) xingxinda@xingxinda:~/public/LinuxCode/work$ ./test
Malloc线程1 块不够了，重新分配2M空间，使用buddy算法分配块大小16B成功

Malloc线程1 使用buddy算法分配块大小256B成功

Malloc线程1 使用buddy算法分配块大小16384B成功

Malloc线程1 使用buddy算法分配块大小131072B成功

Malloc线程1 mmap直接分配块大小263168B成功
```

b) 此时的 ListFree 空闲块情况图

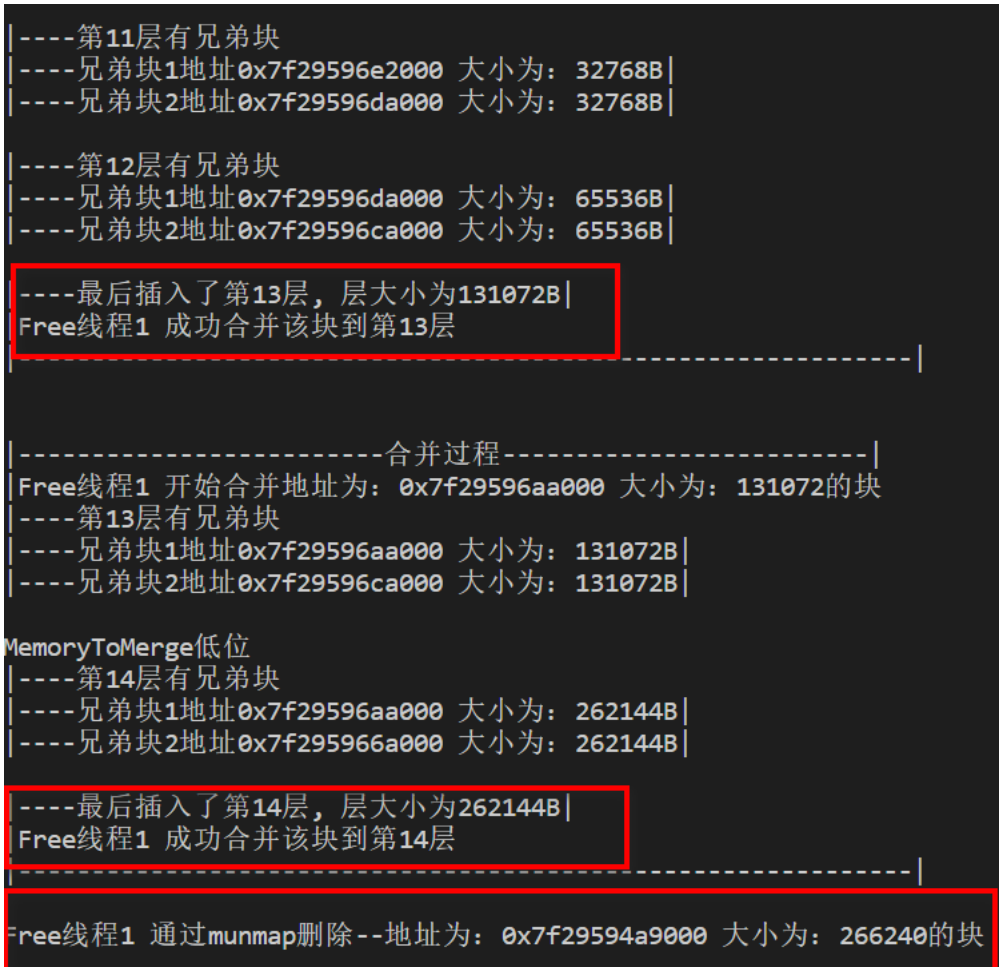
```
|-----------------------------------------------------|
|FreeList空闲块链表数组                                |
|--第14层，本层块大小262144B----------------------------|
|----第1块,地址：0x7f07843f5000，块大小：262144B|
|----第2块,地址：0x7f07843b5000，块大小：262144B|
|----第3块,地址：0x7f0784375000，块大小：262144B|
|----第4块,地址：0x7f0784335000，块大小：262144B|
|----第5块,地址：0x7f07842f5000，块大小：262144B|
|----第6块,地址：0x7f07842b5000，块大小：262144B|
|----第7块,地址：0x7f0784275000，块大小：262144B|
|-----------------------------------------------------|

|--第12层，本层块大小65536B-----------------------------|
|----第1块,地址：0x7f0784455000，块大小：65536B|
|-----------------------------------------------------|

|--第11层，本层块大小32768B-----------------------------|
|----第1块,地址：0x7f0784465000，块大小：32768B|
|-----------------------------------------------------|

|--第9层，本层块大小8192B-------------------------------|
|----第1块,地址：0x7f0784471000，块大小：8192B|
|-----------------------------------------------------|

|--第8层，本层块大小4096B-------------------------------|
|----第1块,地址：0x7f0784473000，块大小：4096B|
|-----------------------------------------------------|

|--第7层，本层块大小2048B-------------------------------|
|----第1块,地址：0x7f0784474000，块大小：2048B|
|-----------------------------------------------------|

|--第6层，本层块大小1024B-------------------------------|
|----第1块,地址：0x7f0784474800，块大小：1024B|
|-----------------------------------------------------|

|--第5层，本层块大小512B--------------------------------|
|----第1块,地址：0x7f0784474c00，块大小：512B|
|-----------------------------------------------------|

|--第3层，本层块大小128B--------------------------------|
|----第1块,地址：0x7f0784474f00，块大小：128B|
|-----------------------------------------------------|

|--第2层，本层块大小64B---------------------------------|
|----第1块,地址：0x7f0784474f80，块大小：64B|
|-----------------------------------------------------|

|--第1层，本层块大小32B---------------------------------|
|----第1块,地址：0x7f0784474fc0，块大小：32B|
|-----------------------------------------------------|

|--第0层，本层块大小16B---------------------------------|
|----第1块,地址：0x7f0784474fe0，块大小：16B|
```

c) ListAlloc 已分配块结果

2. 方案二　在 方案一基础上开始回收内存，单线程申请 3 块（由于报告篇幅限制，只申请 3 块。）大小 256B，128KB，257KB, 同时开始回收内存。

   结果如图：可以看到兄弟块合并过程，以及 257KB 超出范围，直接 munmap 掉。

```
(base) xingxinda@xingxinda:~/public/LinuxCode/work$ ./test
Malloc线程1 块不够了，重新分配2M空间，使用buddy算法分配块大小256B成功

Malloc线程1 使用buddy算法分配块大小131072B成功

Malloc线程1 mmap直接分配块大小263168B成功


|------------------------合并过程------------------------|
|Free线程1 开始合并地址为：0x7f29596e9f00 大小为：256的块
|----第4层有兄弟块
|----兄弟块1地址0x7f29596e9f00 大小为：256B|
|----兄弟块2地址0x7f29596e9e00 大小为：256B|

|----第5层有兄弟块
|----兄弟块1地址0x7f29596e9e00 大小为：512B|
|----兄弟块2地址0x7f29596e9c00 大小为：512B|

|----第6层有兄弟块
|----兄弟块1地址0x7f29596e9c00 大小为：1024B|
|----兄弟块2地址0x7f29596e9800 大小为：1024B|

|----第7层有兄弟块
|----兄弟块1地址0x7f29596e9800 大小为：2048B|
|----兄弟块2地址0x7f29596e9000 大小为：2048B|

|----第8层有兄弟块
|----兄弟块1地址0x7f29596e9000 大小为：4096B|
|----兄弟块2地址0x7f29596e8000 大小为：4096B|

|----第9层有兄弟块
|----兄弟块1地址0x7f29596e8000 大小为：8192B|
|----兄弟块2地址0x7f29596e6000 大小为：8192B|

|----第10层有兄弟块
|----兄弟块1地址0x7f29596e6000 大小为：16384B|
|----兄弟块2地址0x7f29596e2000 大小为：16384B|

|----第11层有兄弟块
|----兄弟块1地址0x7f29596e2000 大小为：32768B|
```



结果恢复初始状态

3. 方案三：多线程（伪）随机大小内存大规模分配释放。最大块 257KB，最小块 16B，两个 Malloc 线程，两个 Free 线程，每个

Malloc 线程以随机大小申请 1000 次。

由于结果太长，只放一部分截图。

测试代码：

```
//方案三  大规模 多线程（伪）申请释放内存
CLExecutive *pThreadMalloc=new CLThread(new CLExecutiveFunctionMalloc());
CLExecutive *pThreadMalloc2=new CLThread(new CLExecutiveFunctionMalloc());
CLExecutive *pThreadFree=new CLThread(new CLExecutiveFunctionFree());
CLExecutive *pThreadFree2=new CLThread(new CLExecutiveFunctionFree());
int max = 257 * 1024;//最大申请块范围
int min = 16 ;//最小申请块范围
int size = 1000;//每个线程申请个数
int threadCount = 1;
pThreadMalloc->Run(new Args(max,min,size,threadCount));
pThreadMalloc2->Run(new Args(max,min,size,threadCount+1));
pThreadFree->Run(new Args(max,min,size,threadCount));
pThreadFree2->Run(new Args(max,min,size,threadCount+1));
pThreadMalloc->WaitForDeath();
pThreadMalloc2->WaitForDeath();
pThreadFree->WaitForDeath();
pThreadFree2->WaitForDeath();
```

a) 伪多线程结果：

```
(base) xingxinda@xingxinda:~/public/LinuxCode/work$ ./test

线程ID: 139992876742400

线程ID: 139992859956992

线程ID: 139992868349696
Malloc线程1 块不够了，重新分配2M空间，使用buddy算法分配块大小9591B成功

Malloc线程1 使用buddy算法分配块大小49273B成功

Malloc线程1 使用buddy算法分配块大小97411B成功


线程ID: 139992781289216

|-------------------------合并过程-------------------------|
|Free线程2 开始合并地址为: 0x7f52a0301000 大小为: 16384的块
|----第10层有兄弟块
|----兄弟块1地址0x7f52a0301000 大小为: 16384B|
|----兄弟块2地址0x7f52a02fd000 大小为: 16384B|
```

```
|----最后插入了第14层，层大小为262144B|
|Free线程2 成功合并该块到第14层
|-------------------------------------------------------|

Malloc线程1 使用buddy算法分配块大小41057B成功

Malloc线程2 块不够了，重新分配2M空间，使用buddy算法分配块大小56278B成功

Malloc线程2 使用buddy算法分配块大小11535B成功

Malloc线程2 使用buddy算法分配块大小120922B成功

Malloc线程2 使用buddy算法分配块大小223484B成功
```

b) 运行后 FreeList 结果

```
|-------------------------------------------------------|
|FreeList空闲块链表数组                                  |
|--第14层，本层块大小262144B----------------------------|
|----第1块,地址：0x7f463d33e000，块大小：262144B|
|----第2块,地址：0x7f463d13e000，块大小：262144B|
|----第3块,地址：0x7f464c624000，块大小：262144B|
|----第4块,地址：0x7f463fc00000，块大小：262144B|
|----第5块,地址：0x7f463c43e000，块大小：262144B|
|----第6块,地址：0x7f463d0be000，块大小：262144B|
|----第7块,地址：0x7f463efc0000，块大小：262144B|
|----第8块,地址：0x7f463d0fe000，块大小：262144B|
|----第9块,地址：0x7f463d07e000，块大小：262144B|
|----第10块,地址：0x7f464c344000，块大小：262144B|
|----第11块,地址：0x7f464c5c4000，块大小：262144B|
|----第12块,地址：0x7f463fcc0000，块大小：262144B|
|----第13块,地址：0x7f463fa40000，块大小：262144B|
|----第14块,地址：0x7f463d63e000，块大小：262144B|
|----第15块,地址：0x7f463ed3e000，块大小：262144B|
|----第16块,地址：0x7f463d27e000，块大小：262144B|
|----第17块,地址：0x7f463d23e000，块大小：262144B|
|----第18块,地址：0x7f463e4de000，块大小：262144B|
|----第19块,地址：0x7f463fd00000，块大小：262144B|
|----第20块,地址：0x7f463ff40000，块大小：262144B|
|----第21块,地址：0x7f463ff80000，块大小：262144B|
|----第22块,地址：0x7f463f040000，块大小：262144B|
|----第23块,地址：0x7f463ecbe000，块大小：262144B|
|----第24块,地址：0x7f463e43e000，块大小：262144B|
```

……

```
|----第467块,地址: 0x7f4638c79000, 块大小: 262144B|
|----第468块,地址: 0x7f4638c39000, 块大小: 262144B|
|-----------------------------------------------------|

|--第13层, 本层块大小131072B---------------------------|
|----第1块,地址: 0x7f463c29c000, 块大小: 131072B|
|----第2块,地址: 0x7f463e51e000, 块大小: 131072B|
|----第3块,地址: 0x7f463e4be000, 块大小: 131072B|
|----第4块,地址: 0x7f464c4e4000, 块大小: 131072B|
|----第5块,地址: 0x7f463f790000, 块大小: 131072B|
|----第6块,地址: 0x7f464c6a4000, 块大小: 131072B|
|----第7块,地址: 0x7f463c23c000, 块大小: 131072B|
|----第8块,地址: 0x7f463ff10000, 块大小: 131072B|
|----第9块,地址: 0x7f463dc8e000, 块大小: 131072B|
|----第10块,地址: 0x7f463fe00000, 块大小: 131072B|
|----第11块,地址: 0x7f464c404000, 块大小: 131072B|
|----第12块,地址: 0x7f463bc1a000, 块大小: 131072B|
|----第13块,地址: 0x7f463bbba000, 块大小: 131072B|
|----第14块,地址: 0x7f46391d9000, 块大小: 131072B|
|----第15块,地址: 0x7f463fe60000, 块大小: 131072B|
|----第16块,地址: 0x7f4639179000, 块大小: 131072B|
|----第17块,地址: 0x7f463969a000, 块大小: 131072B|
|----第18块,地址: 0x7f46395fa000, 块大小: 131072B|
|----第19块,地址: 0x7f464c604000, 块大小: 131072B|
|----第20块,地址: 0x7f463fd60000, 块大小: 131072B|
|-----------------------------------------------------|

|--第12层, 本层块大小65536B----------------------------|
|----第1块,地址: 0x7f463f7b0000, 块大小: 65536B|
|----第2块,地址: 0x7f463f780000, 块大小: 65536B|
|----第3块,地址: 0x7f463ff00000, 块大小: 65536B|
|----第4块,地址: 0x7f463dc7e000, 块大小: 65536B|
|----第5块,地址: 0x7f463dcae000, 块大小: 65536B|
|----第6块,地址: 0x7f463ff30000, 块大小: 65536B|
|-----------------------------------------------------|

|--第11层, 本层块大小32768B----------------------------|
|----第1块,地址: 0x7f463ff28000, 块大小: 32768B|
|-----------------------------------------------------|

|--第10层, 本层块大小16384B----------------------------|
|----第1块,地址: 0x7f463e14a000, 块大小: 16384B|
|----第2块,地址: 0x7f463ff22000, 块大小: 16384B|
|----第3块,地址: 0x7f464c6b8000, 块大小: 16384B|
|-----------------------------------------------------|
```

…

c) 以及最后的 ListAlloc 为空，所有分配的内存都回收了。

```
|--第12层，本层块大小65536B----------------------------|
|----第1块,地址：0x7f463f7b0000，块大小：65536B|
|----第2块,地址：0x7f463f780000，块大小：65536B|
|----第3块,地址：0x7f463ff00000，块大小：65536B|
|----第4块,地址：0x7f463dc7e000，块大小：65536B|
|----第5块,地址：0x7f463dcae000，块大小：65536B|
|----第6块,地址：0x7f463ff30000，块大小：65536B|
|-----------------------------------------------------|

|--第11层，本层块大小32768B----------------------------|
|----第1块,地址：0x7f463ff28000，块大小：32768B|
|-----------------------------------------------------|

|--第10层，本层块大小16384B----------------------------|
|----第1块,地址：0x7f463e14a000，块大小：16384B|
|----第2块,地址：0x7f463ff22000，块大小：16384B|
|----第3块,地址：0x7f464c6b8000，块大小：16384B|
|-----------------------------------------------------|

|--第9层，本层块大小8192B-----------------------------|
|----第1块,地址：0x7f464c6b6000，块大小：8192B|
|----第2块,地址：0x7f463ff26000，块大小：8192B|
|-----------------------------------------------------|

|--第8层，本层块大小4096B-----------------------------|
|----第1块,地址：0x7f463ff21000，块大小：4096B|
|----第2块,地址：0x7f464c6b5000，块大小：4096B|
|-----------------------------------------------------|

|-----------------------------------------------------|
|FreeAlloc已分配块链表                                  |
|-----------------------------------------------------|
```

通过结果分析可以得出绝大部分内存回收后最后都合并到上限 256KB 内存块（共 468 块）上了，产生的内存碎片很少，第八层往下没有一块内存碎片，效果还是十分不错的。

附录

给出代码和测试代码，注释行数要超过总行数的20%

LearningNotes/linux 高级编程 at main · xxdznl/LearningNotes (github.com)

代码可以直接运行，测试代码在 main 里，依次取消 main 函数的三个方案注释就好。

遗憾：

课上讲了那么久，多线程互斥并发运行，自己写的时候还是一脸懵逼，调不通程序，由于时间关系，多线程交互运行没能实现，很遗憾。

心路：

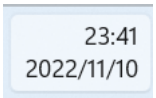
看着 deadline 一天天逐步逼近，内心十分不是滋味。

从 11 月 6 日晚开始赶大作业，，一开始丝毫没有头绪，花了一天半时间仔细看了给的案例 malloc.c 以及课上讲的代码，对内存分配和多线程产生初步的认识，有了模糊的感念。但每当要敲代码时又举步维艰。

11 月 8 日，又花一天时间查询资料，学习 buddy 算法，在 github 上到处找源码。真的寝食难安！

终于在 11 月 9 日早上，正上着数学课，在纸上画着大体框架，要用的数据结构，突然就思如泉涌，赶忙翘课回寝室，把整体逻辑代码框架写出来。

经过 9 日，10 日两天连续奋战，终于在 10 号写完程序、修改 bug、进行测试，10 号晚上极限写完报告。真的太难了！



感谢刘老师，感谢 **csdn** 上那位学长，感谢 **github** 上的不知名国际友人，感谢这几天拼命的自己！