

一、实验要求：

实现以字节为粒度的动态内存管理机制，可以分配和释放任意字节长度的内存。

二、实验内容：

实现：

为实现以字节为粒度的内存的申请，我们引入了 arena 的概念。我们分配一个页的大小，每个页的头部都有描述符，其余部分为可分配的内存。将其余部分的内存平均分成大小相等的若干块，这些块就是 arena。块的大小可以是 16、32、64、128、256、512、1024。当申请的内存大小大于 1024 时，我们分配整个页给用户。因为当申请的内存大小大于 1024 MB 时，我们不能将剩下的内存划分为两个大于 1024 KB 的内存块。因此只能分配整个页。

描述符定义如下：

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

为有序管理 Arena，我们用 MemoryBlockListItem 来管理。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为相等大小的arena，然后返回第一个 arena的起始地址。剩余的 arena 将会被放在一个双向链表中，作为空闲的 arena。当空闲的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 进行返回。

MemoryBlockListItem 定义如下：

```
struct MemoryBlockListItem
{
    MemoryBlockListItem *previous, *next;
};
```

为了实现方便，定义一个用于动态内存分配的类 `class ByteMemoryManager`。

在实现之前我们需要在进程的PCB中添加这个管理类。如下：

```
struct PCB
{
    ...
    ByteMemoryManager byteMemoryManager; // 进程内存管理者
    ...
};
```

并在 `int ProgramManager::executeProcess(const char *filename, int priority)` 函数中添加 `byteMemoryManager` 的初始化语句。

```
int ProgramManager::executeProcess(const char *filename, int priority)
{
    ...
    // 找到刚刚创建的PCB
    PCB *process = ListItem2PCB(allPrograms.back(), tagInAllList);
    process->byteMemoryManager.initialize();
    ...
};
```

对于每个由进程创建的线程，他们共享进程的 `byteMemoryManager`。所有内核线程共享一个 `ByteMemoryManager`。

因此我们在 `setup.cpp` 中声明一个 `ByteMemoryManager kernelByteMemoryManager;`，并进行初始化 `kernelByteMemoryManager.initialize();`，然后在 `os_modules.h` 中声明 `extern ByteMemoryManager kernelByteMemoryManager;`。

`class ByteMemoryManager` 的定义如下：

```
// 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);
};
```

`MEM_BLOCK_TYPES` 为内存块的类型数目，`minSize` 为 7 种类型中规格最小的块的大小。

`arenaSize[MEM_BLOCK_TYPES]` 记录每种类型对于的内存块大小，`arenas[MEM_BLOCK_TYPES]` 为每种类型的 `arena` 内存块指针。

初始化函数实现如下：

```
ByteMemoryManager::ByteMemoryManager()
{
    initialize();
}
```

```

void ByteMemoryManager::initialize()
{
    int size = minSize;
    for (int i = 0; i < MEM_BLOCK_TYPES; ++i)
    {
        arenas[i] = nullptr;
        arenaSize[i] = size;
        size = size << 1;
    }
}

```

刚开始时，还没有进行动态内存分配，故每种类型的 arena 内存块指针为 `nullptr`。初始化 arena 的块大小分别为 16、32、64、128、512、1024。

malloc 函数的实现如下：

```

void *ByteMemoryManager::allocate(int size)
{
    int index = 0;
    while (index < MEM_BLOCK_TYPES && arenaSize[index] < size)
        ++index;

    PCB *pcb = programManager.running;
    AddressPoolType poolType = (pcb->pageDirectoryAddress) ?
    AddressPoolType::USER : AddressPoolType::KERNEL;
    void *ans = nullptr;

    if (index == MEM_BLOCK_TYPES)
    {
        // 上取整
        int pageAmount = (size + sizeof(Arena) + PAGE_SIZE - 1) / PAGE_SIZE;

        ans = memoryManager.allocatePages(poolType, pageAmount);

        if (ans)
        {
            Arena *arena = (Arena *)ans;
            arena->type = ArenaType::ARENA_MORE;
            arena->counter = pageAmount;
        }
    }
    else
    {
        //printf("---ByteMemoryManager::allocate----\n");
        if (arenas[index] == nullptr)
        {
            if (!getNewArena(poolType, index))
                return nullptr;
        }

        // 每次取出内存块链表中的第一个内存块
        ans = arenas[index];
        arenas[index] = ((MemoryBlockListItem *)ans)->next;
    }
}

```

```

        if (arenas[index])
        {
            (arenas[index])->previous = nullptr;
        }

        Arena *arena = (Arena *)((int)ans & 0xfffff000);
        --(arena->counter);
        //printf("---ByteMemoryManager::allocate----\n");
    }

    return ans;
}

```

传进的参数是要申请的字节数，我们通过参数找到需要分配的内存块大小是多少，如果大于 1024，我们分配一个页，否则分配能满足需求的最小内存块。

找到合适大小的块之后，查看申请内存的是线程还是进程，是用户态还是内核态。然后我们再决定在内核的池中进行分配还是在用户态的池中进行分配。

然后我们确认对应 size 的 arena 空闲列表是否为空，如果不为空，则找到空闲的 arena，否则通过函数 `bool getNewArena(AddressPoolType type, int index)` 申请一个页进行分块，然后再分配。

如果申请的内存大小大于 1024 字节，则以页为单位分配，用 `pageAmount = (size + sizeof(Arena) + PAGE_SIZE - 1) / PAGE_SIZE` 进行判断所需要的页的数量。

`bool getNewArena(AddressPoolType type, int index)` 函数的实现如下：

```

bool ByteMemoryManager::getNewArena(AddressPoolType type, int index)
{
    void *ptr = memoryManager.allocatePages(type, 1);

    if (ptr == nullptr)
        return false;

    // 内存块的数量
    int times = (PAGE_SIZE - sizeof(Arena)) / arenaSize[index];
    // 内存块的起始地址
    int address = (int)ptr + sizeof(Arena);

    // 记录下内存块的数据
    Arena *arena = (Arena *)ptr;
    arena->type = (ArenaType)index;
    arena->counter = times;
    // printf("---ByteMemoryManager::getNewArena: type: %d, arena->counter: %d\n", index, arena->counter);

    MemoryBlockListItem *prevPtr = (MemoryBlockListItem *)address;
    MemoryBlockListItem *curPtr = nullptr;
    arenas[index] = prevPtr;
    prevPtr->previous = nullptr;
    prevPtr->next = nullptr;
    --times;

    while (times)
    {
        address += arenaSize[index];
    }
}

```

```

        curPtr = (MemoryBlockListItem *)address;
        prevPtr->next = curPtr;
        curPtr->previous = prevPtr;
        curPtr->next = nullptr;
        prevPtr = curPtr;
        --times;
    }

    return true;
}

```

我们从内存中分配一个页，然后将页划分成第 index 大的块。可以划分的块的数量为页的大小减去 Arena 在页的首部所占的大小除以块的大小： $(\text{PAGE_SIZE} - \text{sizeof}(\text{Arena})) / \text{arenaSize}[\text{index}]$ ，然后将空闲的 Arena 块串成空闲链表。

free 函数的实现如下：

```

void ByteMemoryManager::release(void *address)
{
    // 由于Arena是按页分配的，所以其首地址的低12位必定0，
    // 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
    Arena *arena = (Arena *)((int)address & 0xfffff000);

    if (arena->type == ARENA_MORE)
    {
        int address = (int)arena;

        memoryManager.releasePage(address, arena->counter);
    }
    else
    {
        MemoryBlockListItem *itemPtr = (MemoryBlockListItem *)address;
        itemPtr->next = arenas[arena->type];
        itemPtr->previous = nullptr;

        if (itemPtr->next)
        {
            itemPtr->next->previous = itemPtr;
        }

        arenas[arena->type] = itemPtr;
        ++(arena->counter);

        // 若整个Arena被归还，则清空分配给Arena的页
        int amount = (PAGE_SIZE - sizeof(Arena)) / arenaSize[arena->type];
        // printf("---ByteMemoryManager::release---: arena->counter: %d, amount: %d\n", arena->counter, amount);

        if (arena->counter == amount)
        {
            // 将属于Arena的内存块从链上删除
            while (itemPtr)
            {
                if ((int)arena != ((int)itemPtr & 0xfffff000))
                {

```

```

        itemPtr = itemPtr->next;
        continue;
    }

    if (itemPtr->previous == nullptr) // 链表中的第一个节点
    {
        arenas[arena->type] = itemPtr->next;
        if (itemPtr->next)
        {
            itemPtr->next->previous = nullptr;
        }
    }
    else
    {
        itemPtr->previous->next = itemPtr->next;
    }

    if (itemPtr->next)
    {
        itemPtr->next->previous = itemPtr->previous;
    }

    itemPtr = itemPtr->next;
}

memeoryManager.releasePage((int)address, 1);
}
}
}

```

因为页的大小是 4 kB，所以我们可以通过 `(Arena *)((int)address & 0xfffff000)` 找到需要释放的内存地址所在的页。找到页地址后，我们就可以通过页开头保存的元信息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。

特殊地，当这个页的所有 arena 块被释放完时，我们不将最后一个释放的 arena 块放入空闲链表，而要释放整个页。

实现完底层的 malloc 和 free 后，我们利用这两个函数实现系统调用处理函数。两个系统调用处理函数的实现在 `syscall.cpp` 中。

```

void *syscall_malloc(int size)
{
    PCB *pcb = programManager.running;
    if (pcb->pageDirectoryAddress)
    {
        // 每一个进程有自己的ByteMemoryManager
        return pcb->byteMemoryManager.allocate(size);
    }
    else
    {
        // 所有内核线程共享一个ByteMemoryManager
        return kernelByteMemoryManager.allocate(size);
    }
}

```

对于每个申请内存的实体，我们先判断它是内核线程还是进程，然后在相应的 `ByteMemoryManager` 中分配内存块。

```
void syscall_free(void *address)
{
    PCB *pcb = programManager.running;
    if (pcb->pageDirectoryAddress)
    {
        pcb->byteMemoryManager.release(address);
    }
    else
    {
        kernelByteMemoryManager.release(address);
    }
}
```

同样，我们先判断释放内存块的主体是内核线程还是进程。如果是进程，则在 PCB 的 `ByteMemoryManager` 中释放内存块，否则在全局的 `kernelByteMemoryManager` 中释放内存块。

然后我们对系统调用处理函数进行封装。先在 `syscall.h` 中声明两个系统调用函数：

```
void *malloc(int size);
void free(void *address);
```

然后在 `syscall.cpp` 中实现封装：

```
void *malloc(int size){
    return syscall_malloc( size);
}

void free(void *address){
    syscall_free(address);
}
```

最后，我们在 `setup.cpp` 中加入两个系统调用：

```
extern "C" void setup_kernel()
{
    ...
    systemService.setSystemCall(5,(int)malloc);
    systemService.setSystemCall(6,(int)free);
    ...
}
```

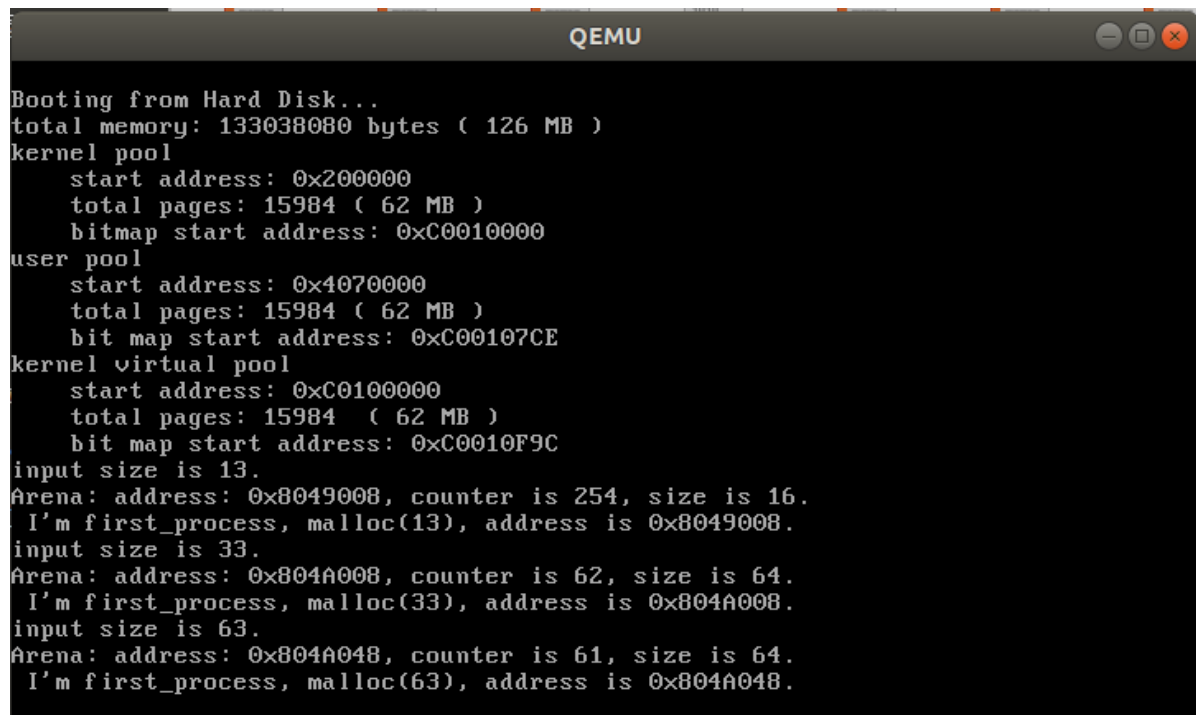
这样我们就可以通过系统调用，去申请和释放内存了。

```

void first_process()
{
    int pid ;
    int retval;
    PCB *parent = programManager.running;
    int addr1 = asm_system_call(5, 13);
    printf("I'm first_process, malloc(13), address is 0x%x.\n", addr1);
    int addr2 = asm_system_call(5, 33);
    printf("I'm first_process, malloc(33), address is 0x%x.\n", addr2);
    int addr3 = asm_system_call(5, 63);
    printf("I'm first_process, malloc(63), address is 0x%x.\n", addr3);
    asm_halt();
}

```

结果:



```

QEMU

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x2000000
  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
input size is 13.
Arena: address: 0x8049008, counter is 254, size is 16.
I'm first_process, malloc(13), address is 0x8049008.
input size is 33.
Arena: address: 0x804A008, counter is 62, size is 64.
I'm first_process, malloc(33), address is 0x804A008.
input size is 63.
Arena: address: 0x804A048, counter is 61, size is 64.
I'm first_process, malloc(63), address is 0x804A048.

```

可以看到在同个进程内申请13MB内存时, Arena的块大小是16MB, 起始地址是0x8049008, 申请了一块之后还剩254块。

之后该进程又分别申请33MB和63MB的内存, 这两个内存大小都在 32MB ~ 64MB之间, 所以分配的是 64MB 的内存块, 因而他们共享一个arena。分配完 33 MB 后, arena 剩下62块, 然后再分配 63 MB 内存大小, arena 剩下 61块。

内存为 33 MB 的地址为 0x804A008, 比 13 MB内存的其实地址 0x8049008 多 0x1000, 大小正好为 4k 。63 MB 的地址为 0x804a048, 与 33 MB 的地址相差 0x40, 大小正好为 64。

创建第二个进程, 此进程每次申请 1025 MB 的内存, 申请 3 次。如下:


```

void second_process()
{
    int pid ;
    int retval;
    PCB *parent = programManager.running;
    int addr1 = asm_system_call(5, 1025);
    printf("I'm second_process, malloc(1025), address is 0x%x.\n", addr1);
    int addr2 = asm_system_call(5, 1025);
    printf("I'm second_process, malloc(1025), address is 0x%x.\n", addr2);
    int addr3 = asm_system_call(5, 1025);
    printf("I'm second_process, malloc(1025), address is 0x%x.\n", addr3);
    asm_halt();
}

```

创建第二个线程和第三个线程，这两个线程分别由第一个线程创建，他们分别申请两次内存，一次是13 MB，一次是 33 MB。如下：

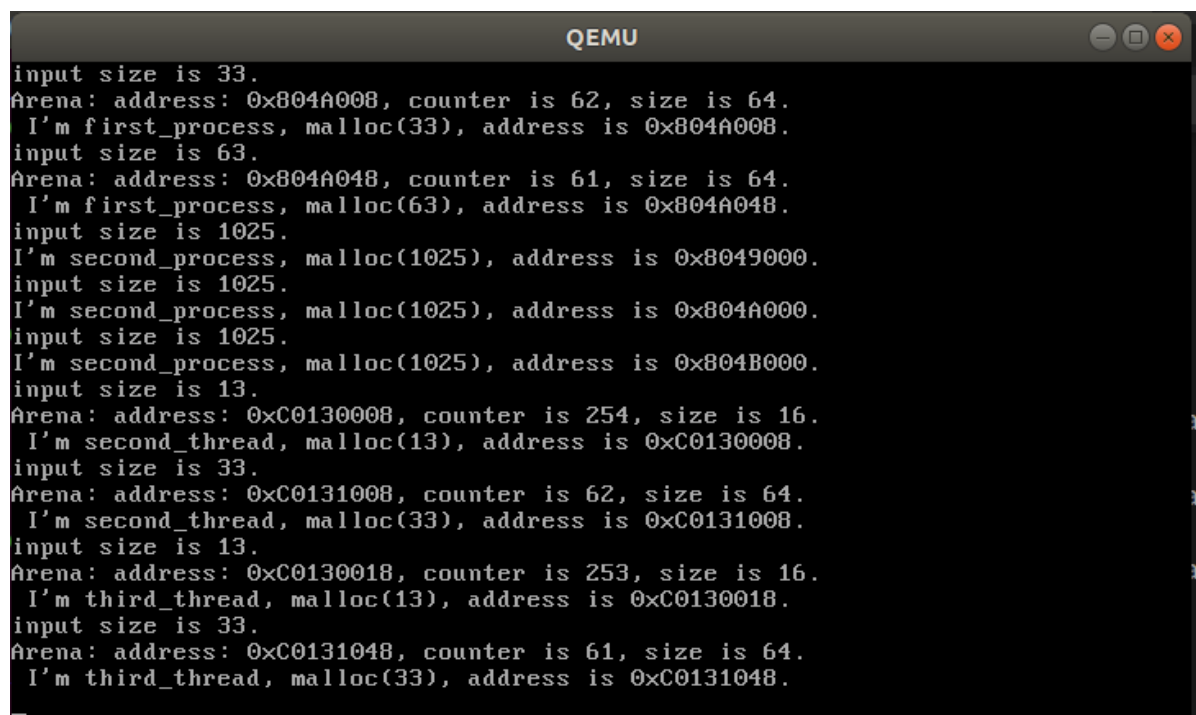
```

void second_thread(void *arg)
{
    int addr1 = asm_system_call(5, 13);
    printf("I'm second_thread, malloc(13), address is 0x%x.\n", addr1);
    int addr2 = asm_system_call(5, 33);
    printf("I'm second_thread, malloc(33), address is 0x%x.\n", addr2);
    asm_halt();
}

void third_thread(void* arg)
{
    int addr1 = asm_system_call(5, 13);
    printf("I'm third_thread, malloc(13), address is 0x%x.\n", addr1);
    int addr2 = asm_system_call(5, 33);
    printf("I'm third_thread, malloc(33), address is 0x%x.\n", addr2);
    asm_halt();
}

```

运行结果：



```

QEMU
input size is 33.
Arena: address: 0x804A008, counter is 62, size is 64.
I'm first_process, malloc(33), address is 0x804A008.
input size is 63.
Arena: address: 0x804A048, counter is 61, size is 64.
I'm first_process, malloc(63), address is 0x804A048.
input size is 1025.
I'm second_process, malloc(1025), address is 0x8049000.
input size is 1025.
I'm second_process, malloc(1025), address is 0x804A000.
input size is 1025.
I'm second_process, malloc(1025), address is 0x804B000.
input size is 13.
Arena: address: 0xC0130008, counter is 254, size is 16.
I'm second_thread, malloc(13), address is 0xC0130008.
input size is 33.
Arena: address: 0xC0131008, counter is 62, size is 64.
I'm second_thread, malloc(33), address is 0xC0131008.
input size is 13.
Arena: address: 0xC0130018, counter is 253, size is 16.
I'm third_thread, malloc(13), address is 0xC0130018.
input size is 33.
Arena: address: 0xC0131048, counter is 61, size is 64.
I'm third_thread, malloc(33), address is 0xC0131048.

```

```

I'm second_process, malloc(1025), address is 0x8049000.
input size is 1025.
I'm second_process, malloc(1025), address is 0x804A000.
input size is 1025.
I'm second_process, malloc(1025), address is 0x804B000.
input size is 13.
Arena: address: 0xC0130008, counter is 254, size is 16.
I'm second_thread, malloc(13), address is 0xC0130008.
input size is 33.
Arena: address: 0xC0131008, counter is 62, size is 64.
I'm second_thread, malloc(33), address is 0xC0131008.
input size is 13.
Arena: address: 0xC0130018, counter is 253, size is 16.
I'm third_thread, malloc(13), address is 0xC0130018.
input size is 33.
Arena: address: 0xC0131048, counter is 61, size is 64.
I'm third_thread, malloc(33), address is 0xC0131048.

```

分析:

可以看到，第二个进程每次申请 1025 MB 内存时，每次得到的地址差都是 0x1000 MB, 即分配的是一整个页，这和设计是一致的。

当第二个线程申请 13 MB 内存的地址是 0xC0130008，剩下的内存块有254，申请 33 MB 的地址是 0xC0131008, 剩下的内存块有 62。

当第三个线程申请 13 MB 内存时，地址为 0xC0130018, 比 0xC0130008 多 0x10，刚好是16，此时剩下 253 块可用。说明两个线程共享同一个 arena。同样的道理，第三个线程申请 33 MB 的内存时，也和第二个线程共享同一个 arena，因此第三个线程申请完后，还剩 61 块可用。

因此我们证明了 malloc 工作的正确性。

测试 free 函数的正确性:

见 assignment2.

我们创建两个进程进行测试。

第一个进程做如下动作:

- ① 申请 13 MB 的内存。
- ② 释放 13 MB 的内存。
- ③ 申请 33 MB 的内存。
- ④ 申请 63 MB 的内存。
- ⑤ 释放 63 MB 的内存。
- ⑥ 释放 33 MB 的内存。

实现如下:

```

void first_process()
{
    int pid ;
    int retval;
    PCB *parent = programManager.running;
    int addr1 = asm_system_call(5, 13);
    printf("I'm first_process, malloc(13), address is 0x%x.\n", addr1);
    printf("I'm first_process, free address 0x%x.\n",addr1);
    asm_system_call(6,addr1);
}

```

```

int addr2 = asm_system_call(5, 33);
printf("I'm first_process, malloc(33), address is 0x%x.\n", addr2);
int addr3 = asm_system_call(5, 63);
printf("I'm first_process, malloc(63), address is 0x%x.\n", addr3);
printf("I'm first_process, free address 0x%x.\n",addr3);
asm_system_call(6,addr3);
printf("I'm first_process, free address 0x%x.\n",addr2);
asm_system_call(6,addr2);
while(true){
    int a = 0;
}
asm_halt();
}

```

第二个进程做如下动作：

- ① 第一次申请 1025 MB 的内存。
- ② 第二次申请 1025 MB 的内存。
- ③ 释放第一次申请的 1025 MB 的内存。
- ④ 第三次申请 1025 MB 的内存。
- ⑤ 释放第二次申请的 1025 MB 的内存。
- ⑥ 释放第三次申请的 1025 MB 的内存。

实现如下：

```

void second_process()
{
    int pid ;
    int retval;
    PCB *parent = programManager.running;
    int addr1 = asm_system_call(5, 1025);
    printf("I'm second_process, malloc(1025), address is 0x%x.\n", addr1);
    int addr2 = asm_system_call(5, 1025);
    printf("I'm second_process, malloc(1025), address is 0x%x.\n", addr2);
    printf("I'm second_process, free address 0x%x.\n",addr1);
    asm_system_call(6,addr1);
    int addr3 = asm_system_call(5, 1025);
    printf("I'm second_process, malloc(1025), address is 0x%x.\n", addr3);
    printf("I'm second_process, free address 0x%x.\n",addr2);
    asm_system_call(6,addr2);
    printf("I'm second_process, free address 0x%x.\n",addr3);
    asm_system_call(6,addr3);
    asm_halt();
}

```

运行结果：

```
QEMU
bit map start address: 0xC0010F9C
Arena: address: 0x8049008, counter is 254, size is 16.
I'm first_process, malloc(13), address is 0x8049008.
I'm first_process, free address 0x8049008.
After free, areana counter is 255.
return the page.
Arena: address: 0x8049008, counter is 62, size is 64.
I'm first_process, malloc(33), address is 0x8049008.
Arena: address: 0x8049048, counter is 61, size is 64.
I'm first_process, malloc(63), address is 0x8049048.
I'm first_process, free address 0x8049048.
After free, areana counter is 62.
I'm first_process, free address 0x8049008.
After free, areana counter is 63.
return the page.
I'm second_process, malloc(1025), address is 0x8049000.
I'm second_process, malloc(1025), address is 0x804A000.
I'm second_process, free address 0x8049000.
Address is 8049000. Release page.
I'm second_process, malloc(1025), address is 0x8049000.
I'm second_process, free address 0x804A000.
Address is 804A000. Release page.
I'm second_process, free address 0x8049000.
Address is 8049000. Release page.
```

分析:

可以看到, **第一个进程**申请 13 MB 的内存时, 分配了块大小为 16 MB 的 arena, 起始地址是 0x8049008, 分配完后还剩 254 块。然后释放 13 MB 内存, 释放后, 还剩 255 块, 此时归还整个页。

然后申请 33 MB 的内存, 分配了一个块大小为 64 MB 的 arena, 起始地址是 0x8049008. 可以知道此时分配给 13 MB 的内存已经被回收, 不然这里的内存起始地址不会是 0x8049008. 分配完后还剩 62 块。

然后再申请 63 MB 的内存, 块大小仍然是 64 MB. 起始地址是 0x8049048, 与 0x8049008 正好相差 64MB. 分配完后还剩 61 块。说明 63 MB 的内存与 33 MB 的内存瓜分的是同一个 arena。

然后归还 63 MB 的内存, 归还后剩余 62 块。

然后归还 33 MB 的内存, 归还后剩余 63块, 此时归还整个页。

第二个进程第一次申请 1025 MB, 于是分配了整个页, 起始地址是 0x8049000.

第二次申请 1025 MB, 于是又分配了整个页, 起始地址是 0x804A000.

然后释放第一次申请的 1025 MB, 于是释放整个页。

然后第三次申请 1025 MB, 于是又分配了整个页, 注意到此时的地址是 0x8049000, 和第一次申请 1025 MB 的地址一样, 说明第一次申请的 1025 MB 内存已被释放。

然后分别回收第二次申请的 1025 MB 和第三次申请的 1025 MB 内存, 归还两个页。

因此说明了 free函数的正确性。

没加锁线程是不安全的:

在 `memory.h` 中添加变量 `int flag`, 此变量用于制造不稳定因素, 模拟线程不安全的情况:

```
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
    int flag;
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);
};

```

在memory.cpp中的void *ByteMemoryManager::allocate(int size)修改为一下实现，模拟线程不安全的情况：

```

void *ByteMemoryManager::allocate(int size)
{
    // printf("input size is %d.\n",size);
    int index = 0;
    while (index < MEM_BLOCK_TYPES && arenaSize[index] < size){
        //printf("arenaSize is %d.\n",arenaSize[index] );
        ++index;
    }

    PCB *pcb = programManager.running;
    AddressPoolType poolType = (pcb->pageDirectoryAddress) ?
AddressPoolType::USER : AddressPoolType::KERNEL;
    void* ans = nullptr ;
    //the difference
    if( flag % 2 == 1 ){
        int delay = 0xffffffff;
        while(delay -- ){
            flag ++;
        }
    }
    if (index == MEM_BLOCK_TYPES)
    {
        // 上取整
        int pageAmount = (size + sizeof(Arena) + PAGE_SIZE - 1) / PAGE_SIZE;

        ans = (void*)memoryManager.allocatePages(poolType, pageAmount);

        if (ans)
        {
            Arena *arena = (Arena *)ans;
            arena->type = ArenaType::ARENA_MORE;
            arena->counter = pageAmount;
        }
    }
    else
    {
        //printf("---ByteMemoryManager::allocate----\n");
        if (arenas[index] == nullptr)
        {
            if (!getNewArena(poolType, index)){
                return nullptr;
            }
        }
    }
}

```

```

    }

    // 每次取出内存块链表中的第一个内存块
    ans = arenas[index];
    arenas[index] = ((MemoryBlockListItem *)ans)->next;

    if (arenas[index])
    {
        (arenas[index])->previous = nullptr;
    }

    Arena *arena = (Arena *)((int)ans & 0xfffff000);
    --(arena->counter);
    printf("Arena: address: 0x%x, counter is %d, size is %d.\n ",
    (int)ans, arena->counter, 8 << (index+1));
    //printf("---ByteMemoryManager::allocate----\n");
}
//printf("ByteMemoryManager::allocate address is %x.\n", (int)ans);
return ans;
}

```

(助教可以ctr + F “the difference”，查看修改的地方。)

线程2和线程3分别如下：

```

void second_thread(void *arg)
{
    printf("I'm second_thread, malloc(13).\n ");
    int addr1 = asm_system_call(5, 13);
    printf("Thread2: address of 13 MB is 0x%x.\n", addr1);
    printf("I'm second_thread, malloc(33). \n ");
    int addr2 = asm_system_call(5, 33);
    printf("Thread2: address of 33 MB is 0x%x.\n", addr2);
    asm_halt();
}

void third_thread(void* arg)
{
    printf("I'm third_thread, malloc(13). \n");
    int addr1 = asm_system_call(5, 13);
    printf("Thread3: address of 13 MB is 0x%x.\n", addr1);
    printf("I'm third_thread, malloc(33).\n ");
    int addr2 = asm_system_call(5, 33);
    printf("Thread3: address of 33 MB is 0x%x.\n", addr2);
    asm_halt();
}

```

然后在第一个线程中先执行线程2再执行线程3。

运行结果：

```
QEMU
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
I'm second_thread, malloc(13).
I'm third_thread, malloc(13).
Arena: address: 0xC0100008, counter is 254, size is 16.
Thread3: address of 13 MB is 0xC0100008.
I'm third_thread, malloc(33).
Arena: address: 0xC0101008, counter is 62, size is 64.
Thread3: address of 33 MB is 0xC0101008.
Arena: address: 0xC0100018, counter is 253, size is 16.
Thread2: address of 13 MB is 0xC0100018.
I'm second_thread, malloc(33).
Arena: address: 0xC0101048, counter is 61, size is 64.
Thread2: address of 33 MB is 0xC0101048.
```

通过模拟，可以看到当多个线程都要申请内存时，线程之间会存在竞争。

加锁实现：

在 `memory.h` 中引入头文件 `"sync.h"`，然后加入两个自旋锁：

```
SpinLock user;
SpinLock kernel;
```

在 `memory.cpp` 中实现两个函数，分别是 `Lock` 和 `Unlock`：

```
void Lock(enum AddressPoolType type) {
    if ( type == AddressPoolType::KERNEL ){
        kernel.lock();
    }
    else{
        user.lock();
    }
}

void Unlock(enum AddressPoolType type) {
    if ( type == AddressPoolType::KERNEL ){
        kernel.unlock();
    }
    else{
        user.unlock();
    }
}
```

然后我们就可以利用这两个自旋锁对内核线程和进程的内存申请进行互斥了。

在 `void *ByteMemoryManager::allocate(int size)` 中加锁：

```

void *ByteMemoryManager::allocate(int size)
{
    // printf("input size is %d.\n",size);
    int index = 0;
    while (index < MEM_BLOCK_TYPES && arenaSize[index] < size){
        //printf("arenaSize is %d.\n",arenaSize[index] );
        ++index;
    }

    PCB *pcb = programManager.running;
    AddressPoolType poolType = (pcb->pageDirectoryAddress) ?
AddressPoolType::USER : AddressPoolType::KERNEL;
    void* ans = nullptr ;

    //lock
    memoryManager.Lock(poolType);
    if( flag % 2 == 1 ){
        int delay = 0xffffffff;
        while(delay -- ){}}
        flag ++;
    }
    if (index == MEM_BLOCK_TYPES)
    {
        // 上取整
        int pageAmount = (size + sizeof(Arena) + PAGE_SIZE - 1) / PAGE_SIZE;

        ans = (void*)memoryManager.allocatePages(poolType, pageAmount);

        if (ans)
        {
            Arena *arena = (Arena *)ans;
            arena->type = ArenaType::ARENA_MORE;
            arena->counter = pageAmount;
        }
    }
    else
    {
        //printf("---ByteMemoryManager::allocate----\n");
        if (arenas[index] == nullptr)
        {
            if (!getNewArena(poolType, index)){
                //unlock
                memoryManager.Unlock(poolType);
                return nullptr;
            }
        }

        // 每次取出内存块链表中的第一个内存块
        ans = arenas[index];
        arenas[index] = ((MemoryBlockListItem *)ans)->next;

        if (arenas[index])
        {
            (arenas[index])->previous = nullptr;
        }

        Arena *arena = (Arena *)((int)ans & 0xfffff000);
        --(arena->counter);
    }
}

```



```

        printf("Arena: address: 0x%x, counter is %d, size is %d.\n ",
(int)ans,arena->counter,8 << (index+1));
        //printf("---ByteMemoryManager::allocate----\n");
    }
    //printf("ByteMemoryManager::allocate address is %x.\n",(int)ans);
    //Unlock
    memoryManager.Unlock(poolType);
    return ans;
}

```

当判断出我们应该在哪个内存池中进行分配时，我们就可以进行加锁了。

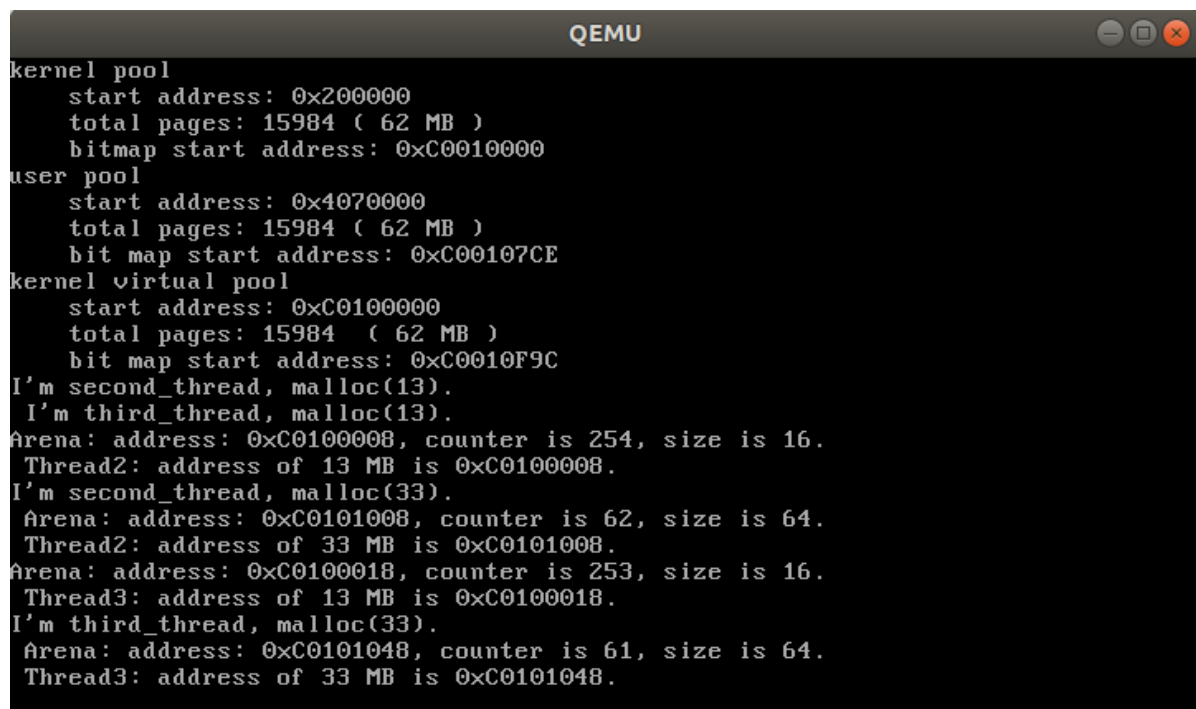
每当分配完内存块，进行返回时，我们就可以进行解锁。

(助教在找锁加在什么地方时可以先找找注释。)

因为以分配出去的内存块不是公共资源，因此不会引起线程会进程的竞争，因此我们无需在 `release` 函数中加锁，进行互斥。

至此，我已经实现了线程/进程同步和互斥处理，因此此时线程是安全的了。

测试：



```

QEMU
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
I'm second_thread, malloc(13).
I'm third_thread, malloc(13).
Arena: address: 0xC0100008, counter is 254, size is 16.
Thread2: address of 13 MB is 0xC0100008.
I'm second_thread, malloc(33).
Arena: address: 0xC0101008, counter is 62, size is 64.
Thread2: address of 33 MB is 0xC0101008.
Arena: address: 0xC0100018, counter is 253, size is 16.
Thread3: address of 13 MB is 0xC0100018.
I'm third_thread, malloc(33).
Arena: address: 0xC0101048, counter is 61, size is 64.
Thread3: address of 33 MB is 0xC0101048.

```

有线程2和线程3分配到的内存地址可以看到，只要线程2先拿到锁（线程2先输出“I'm second_thread, malloc(13)”语句），线程3就不会先申请到内存块。因此验证了锁工作的有效性。

三、实验感想：

此次实验让我对 `malloc` 和 `free` 函数的实现有了更深的理解。同时也对内存的分块有了更进一步的认识。

- 1、内存分为两个空间，一个是用户空间，一个是内核空间。
- 2、在分配和释放内存时，我们都需要先明确该线程/进程是属于哪个空间的，确定了之后，我们才能在相应的空间里进行内存分配。

- 3、在明确了在哪个空间进行内存分配后，我们要明确需要分配的块大小是多少，如果块大小大于 1024 字节，我们只能以页为单位进行分配，否则我们要寻找大于所需 size 的内存的块进行分配。
- 4、若所需要的块的 arena 的空闲链表为空，我们需要分配一个页，创建一个非空的空闲链表，然后进行分配。
- 5、回收内存块时，需要先找到内存地址对应的页地址，再根据页的首部信息，找到对应的信息，然后进行内存的回收。
- 6、在实现互斥时，由于有两个内存空间——用户内存空间和内核态内存空间，因此我在 `memory.h` 中声明了两个自旋锁 `user` 和 `kernel`，用于这两个空间分配时线程/进程的互斥。
- 7、因此在动态内存分配时，一旦我们确认了在哪个空间进行内存分配，我们就要进行加锁，直到返回前才进行解锁。这样我们就实现了两个空间下线程/进程动态申请内存的互斥。