

操作系统原理实验报告

**实验名称:** 实验九 Course Projects

**授课教师：** 张青

**学生姓名:** 曾慧蕾

**学生学号:** 21307358

1. **实验要求**

**Assignment 1 malloc/free的实现：**

我们已经实现了以页为粒度的动态内存分配和释放。但是，我们在程序中使用的往往是以字节为粒度的动态内存管理机制，即我们可以分配和释放任意字节长度的内存。在本项目中，同学们需要实现系统调用malloc和free。malloc用于分配任意字节的内存，free用于释放任意字节的内存。

方案：参考实现

* 参考、复现、测试并详细分析以下实现思路和代码（70分）。
* 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。

1. **实验过程+关键代码+实验结果**

本项目代码基于lab8-task2补充实现。

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个字节的内存时的时候，我们需要找到一个N，满足

2^(N-1)<size<=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)\*4096<size<=M\*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。下面是具体的实现代码。

首先，定义好arena的类型：

enum ArenaType{

    ARENA\_16,

    ARENA\_32,

    ARENA\_64,

    ARENA\_128,

    ARENA\_256,

    ARENA\_512,

    ARENA\_1024,

    ARENA\_MORE

};

元信息定义如下：

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

};

接着，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。

struct MemoryBlockListItem{

    MemoryBlockListItem \*previous, \*next;

};

为了实现方便，我们不妨定义一个类MemoryManager来做动态内存分配。

// 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。

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;

    }

}

以下是内存分配的函数。

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;

}

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。

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;

}

接下来实现内存释放。

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。

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;

        memeoryManager.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);

        }

    }

}

由于进程有自己的内存空间，我们修改PCB，在PCB中加入ByteMemeoryManager。

struct PCB

{

    ...

    ByteMemoryManager byteMemoryManager; // 进程内存管理者

    ...

};

最后实现malloc和free的系统调用处理函数，同学们需要自行加入两个系统调用malloc和free

void \*malloc(int size);

void free(void \*address);

extern ByteMemoryManager kernelByteMemoryManager;

void \*malloc(int size){

    return (void\*)asm\_system\_call(1,size);

}

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);

    }

}

void free(void \*address){

    asm\_system\_call(2,(int)address);

}

void syscall\_free(void \*address)

{

    PCB \*pcb = programManager.running;

    if (pcb->pageDirectoryAddress)

    {

        pcb->byteMemoryManager.release(address);

    }

    else

    {

        kernelByteMemoryManager.release(address);

    }

}

并在setup.cpp中定义好两个系统调用

...

    // 设置1号系统调用

    systemService.setSystemCall(1, (int)syscall\_malloc);

    // 设置2号系统调用

    systemService.setSystemCall(2, (int)syscall\_free);

...

并编写好测试用例（在setup.cpp中）：

void first\_thread(void \*arg)

{

    void \*addr = malloc(20);

    printf("first\_thread get addr0(20) %x\n", addr);

    void \*addr1 = malloc(30);

    printf("first\_thread get addr1(30) %x\n",addr1);

    void \*addr2 = malloc(40);

    printf("first\_thread get addr2(40) %x\n",addr2);

    free(addr);

    printf("first\_thread free addr0(20) %x\n",addr);

    void \*addr3 = malloc(20);

    printf("first\_thread get addr3(20) %x\n",addr3);

    free(addr1);

    free(addr2);

    free(addr3);

}

void second\_thread(void \*arg)

{

    void \*addr = malloc(20);

    printf("second\_thread get addr(20) %x\n",addr);

    free(addr);

}

void first\_process(){

    printf("start process\n");

    void \*addr = malloc(20);

    printf("first\_process get addr0(20) %x\n", addr);

    void \*addr2 = malloc(30);

    printf("first\_process get addr1(30) %x\n", addr2);

    asm\_halt();

}

void first\_thread(void \*arg){

    printf("start first\_thread\n");

    programManager.executeThread(thread1, nullptr, "first", 1);

    programManager.executeThread(second\_thread, nullptr, "second", 1);

    programManager.executeProcess((const char \*)first\_process, 1);

    programManager.executeProcess((const char \*)first\_process, 1);

    asm\_halt();

}

现在对以上代码做出改进。由于上面的代码中并没有做线程、进程同步和互斥处理，因此线程是不安全的。现在可以在其中涉及到动态内存分配的部分加入同步互斥的代码，从而保证线程安全。

1. 在分配页时加入锁机制。文件为memory.cpp

#include "sync.h"

Semaphore sem;

void MemoryManager::initialize(){

    sem.initialize(1);

    ...

}

int MemoryManager::allocatePages(enum AddressPoolType type, const int count){

    sem.P();

    ...

    sem.V();

    return virtualAddress;

}

由于新编写的类MemoryManager中也涉及到动态内存分配，因此也要添加锁机制。在bytememory.cpp中：

#include "sync.h" //lock

Semaphore bm\_sem;

void ByteMemoryManager::initialize(){

    bm\_sem.initialize(1);

    ...

}

void \*ByteMemoryManager::allocate(int size){

    else{

        // 每次取出内存块链表中第一个

        bm\_sem.P();

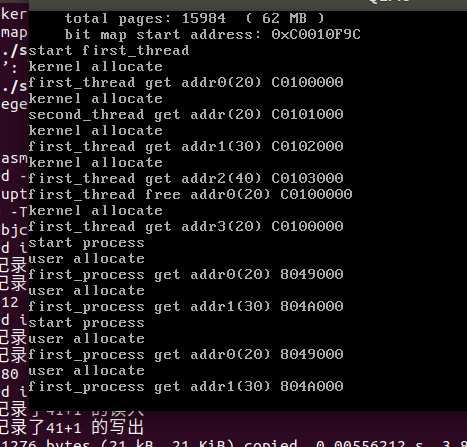
        ...

        bm\_sem.V();

    }

}

最终运行结果如下所示：



Malloc（）分配的地址没有重复的。可见在添加了锁机制之后成功实现互斥。

1. **总结**

本次实验让我了解了malloc和free的实现机制，并且对于动态内存管理机制有了更深刻的理解，对于锁机制的运用也更加熟练。