

Project 2 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。

1. 实验要求

1. 自行设计并实现 `malloc` 和 `free`。
2. 在实现了 `malloc` 和 `free` 后，需要自行提供测例来测试 `malloc` 和 `free`。根据测试方法和输出结果来解释自己程序的正确性。
3. 最后将结果截图并详细分析 `malloc` 和 `free` 的实现思路。

2. 实验过程

2.1 设计并实现malloc和free

1. 创建 `MallocManager` 类，并在其中实现 `malloc`，`free` 等相关功能。
2. 利用信号量 `Semaphore` 类，实现 `malloc` 和 `free` 的线程互斥。
3. 分别为内核线程和用户进程声明 `MallocManager`，为 `malloc`，`free` 等函数设置对应的系统调用函数，从而可以在用户进程中调用 `malloc` 和 `free`。
4. 在 `setup.cpp` 中的用户进程中调用相关函数。

2.2 测试malloc和free

1. 测试 `malloc` 和 `free` 并截图
2. 分析实验结果的正确性及实验思路

3. 关键代码

本实验的主要思路参考在网上查到的如下教程（https://wiki-prog.infoprepa.epita.fr/images/0/04/Malloc_tutorial.pdf）内容：

4.2 How to represent block information

So what we need is a small block at the begining of each chunk containing the extra-information, called meta-data. This block contains at least a pointer to the next chunk, a flag to mark free chunks and the size of the data of the chunk. Of course, this block of information is before the pointer returned by malloc.

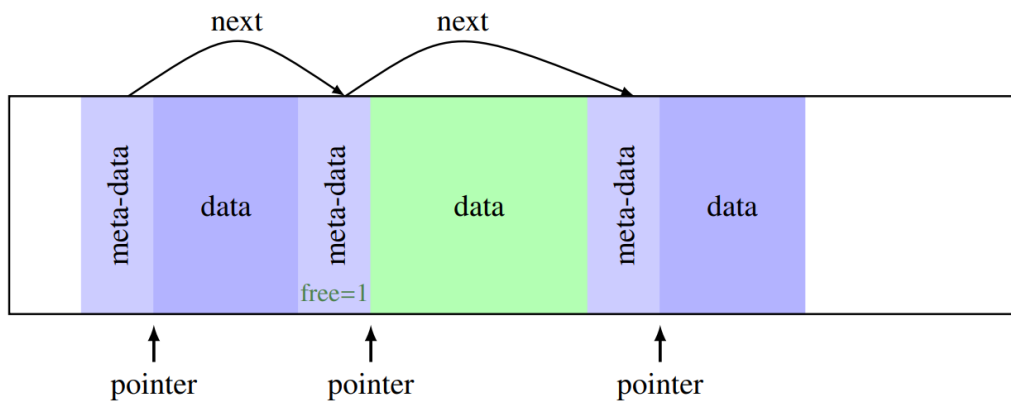


Figure 3: Heap's Chunks Structures

该思路主要内容是将连续的在连续的内存空间中分出 meta-data 和 data，data 也就是 malloc 返回的指针指向的数据块，每个 meta-data 存有它后面的 data 块的信息。

主要的实现步骤如下：

3.1 MallocManager的声明

MallocManager 在 malloc.h 中的声明如下：

```
//malloc.h

#ifndef MALLOC_H
#define MALLOC_H

#include "os_type.h"
#include "stdio.h"
#include "sync.h"

struct block {
    size_t size;
    bool available;
    struct block *next;
};

class MallocManager{
private:
    char * memory_blocks;
    struct block *freeList;
    Semaphore semaphore;
public:
    void initialize();
    void split(struct block*, size_t);
    void *malloc(size_t);
    void merge();
    void free(void *);
};
```

```
#endif
```

MallocManager 中实现了 malloc, free 等功能, block 结构体即为前面所讲的 meta-data 块, malloc 所需的连续内存为 memory_blocks 指针指向的内存块 (通过 allocatePages 获得), freeList 是将 memory_blocks cast 成 struct block * 类型后的指针, 方便内存分配。

3.2 MallocManager的初始化

MallocManager 的初始化很简单, 在 initialize 函数中实现, 代码如下:

```
void MallocManager::initialize() {
    memory_blocks = 0;
    semaphore.initialize(1);
    //printf("memory_blocks: %x\n", memory_blocks);
    //printf("MallocManager: Initialized!\n");
}
```

该初始化函数将 memory_blocks 初始化为空指针 (用于后面 malloc 时判断是否第一次 malloc), 并且将信号量设置为1, 用于实现线程间互斥。

这里我们也可以直接给 MallocManager 分配一个页的内存并且初始化 freeList, 但是由于不是每个进程 (或者内核) 都会用到 malloc 函数, 所以我们不浪费这一部分内存, 留到首次 malloc 时再做这一步。

3.3 malloc的实现

malloc 的实现如下:

```
void *MallocManager::malloc(size_t num_bytes) {
    if ( num_bytes > 4084 ){
        printf("Not enough memory to malloc!\n");
        return 0;
    }
    semaphore.P();
    if ( memory_blocks == 0 ){
        PCB *pcb = programManager.running;
        AddressPoolType poolType = (pcb->pageDirectoryAddress) ?
AddressPoolType::USER : AddressPoolType::KERNEL;
        memory_blocks = (char *) memoryManager.allocatePages(poolType, 1);
        freeList = (struct block *)memory_blocks;
        freeList->size = PAGE_SIZE - sizeof(struct block);
        freeList->available = true;
        freeList->next = 0;
    }
    struct block *curr;
    void *result;
    curr = freeList;
    while (curr->next != 0 && (curr->size < num_bytes || curr->available == 0))
    {
        curr = curr->next;
    }
    if ( curr->available && curr->size == num_bytes) {
        curr->available = false;
        result = (void *) (curr + 1);
    }
```

```

    }
    else if ( curr->available && curr->size > num_bytes + sizeof(struct block))
    {
        split(curr, num_bytes);
        result = (void *) (curr + 1);
    }
    else {
        result = 0;
        printf("malloManager: Haiyaa, not enough memory!\n");
    }
    semaphore.V();
    return result;
}

```

malloc 函数首先检查 malloc 传入的字节数有多大，如果大于一个页能够分配的字节数，就打印提示信息，并返回空指针。

接着，malloc 函数检查是否第一次 malloc，若是第一次 malloc 则为 MallocManager 分配一页内存，并做好 freeList 相关的初始化。

最后，该函数遍历 freeList 中的所有 meta-data 块，找到一个满足条件的 meta-data 块（以及其对应的内存块），若找到的 meta-data 块 size 与需要 malloc 的内存块大小相同，就直接把 available 设置为 false，若大于需要 malloc 的内存块，则 split 出一块大小正好的内存块。split 函数的实现如下：

```

void MallocManager::split(struct block *slot, size_t size) {
    //printf("%x %x\n", slot, size);
    struct block *new_block = (struct block *) ((char *) (slot + 1) + size);
    //printf("%x\n", new_block);
    new_block->size = (slot->size) - size - sizeof(struct block);
    new_block->available = true;
    new_block->next = slot->next;
    slot->size = size;
    slot->available = false;
    slot->next = new_block;
}

```

3.4 free的实现

free 的实现比较简单，先将对应的 meta-data 块设置为 available，再用 merge 函数，遍历所有的 meta-data 块，把连续的可用内存块合并即可。代码如下：

```

void MallocManager::free(void *ptr) {
    semaphore.P();
    if ((void *)memory_blocks <= ptr && ptr <= (void *) (memory_blocks +
PAGE_SIZE)) {
        struct block *curr = (struct block *) (ptr-1);
        curr->available = true;
        merge();
        printf("mallocManager: One block freed!\n");
    }
    else{
        printf("mallocManager: Pointer invalid!\n");
    }
    semaphore.V();
}

```

其中 `merge()` 函数的实现如下：

```

void MallocManager::merge() {
    struct block *curr;
    curr = freeList;
    while (curr && curr->next) {
        if (curr->available && curr->next->available) {
            curr->size += curr->next->size + sizeof(struct block);
            curr->next = curr->next->next;
            printf("mallocManager: Merged two blocks!\n");
        }
        curr = curr->next;
    }
}

```

3.5 实现线程间的同步和互斥

（这一部分在前面的代码中已经体现了，单独列出以显示实现过程逻辑）在 `MallocManager` 的声明中添加一个信号量 `semaphore`，并在 `intitalize` 函数中将其初始化为1，在 `malloc` 和 `free` 函数头尾加上 `P` 和 `V` 操作，就实现了线程之间的互斥。

至此我们就实现了 `malloc` 和 `free` 的基本功能。剩余的工作如下：

3.6 实现系统调用

实现了 `malloc` 和 `free` 后，就要为其实现系统调用函数（虽然在我的 `qemu` 上似乎用户进程直接调用了内核函数也不会出现闪退等情况），代码如下：

```

void * malloc(size_t num_bytes){
    return (void *)asm_system_call(6, (int)num_bytes);
}

void * syscall_malloc(size_t num_bytes){
    PCB *pcb = programManager.running;
    if (pcb->pageDirectoryAddress)
    {
        return pcb->mallocManager.malloc(num_bytes);
    }
    return kernelMallocManager.malloc(num_bytes);
}

```

```

void free( void * ptr ){
    asm_system_call(7, (int)ptr);
}

void syscall_free(void * ptr){
    PCB *pcb = programManager.running;
    if (pcb->pageDirectoryAddress)
    {
        return pcb->mallocManager.free(ptr);
    }
    return kernelMallocManager.free(ptr);
}

```

调用 `MallocManager::malloc` 时，需要判断是内核线程还是用户进程，并根据判断结果调用不同的 `MallocManager` 的函数。

这些函数要记得在 `syscall.h` 中进行声明，不然会报错。

以上用到的各个 `ManagerManager` 在下面声明：

3.7 setup_kernel函数与其他设置

在 `setup.cpp` 中声明一个内核线程共用的 `MallocManager`，并在 `setup_kernel` 函数中对其进行初始化，并且设置系统调用，代码如下：

```

//malloc
MallocManager kernelMallocManager;

extern "C" void setup_kernel() {
    ...
    // set system call 6
    systemService.setSystemCall(6, (int)syscall_malloc);
    // set system call 7
    systemService.setSystemCall(7, (int)syscall_free);

    // 内存管理器
    memoryManager.initialize();

    // mallocManager
    kernelMallocManager.initialize();
    ...
}

```

同时还要记得在 `os_modules.h` 中声明该 `MallocManager`，如下：

```
extern MallocManager kernelMallocManager;
```

至于各个用户进程的自己的 `MallocManager`，我们将它放在 `PCB` 中，如下：

```

struct PCB
{
    ...
    MallocManager mallocManager; // MallocManager
    ...
};

```

在 `exeuteProcess` 中进行初始化:

```

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

    ...
    process->mallocManager.initialize();

    interruptManager.setInterruptStatus(status);

    return pid;
}

```

至此, 我们就实现了 `malloc` 和 `free` 啦。

4. 实验结果

4.1 MallocManager初始化

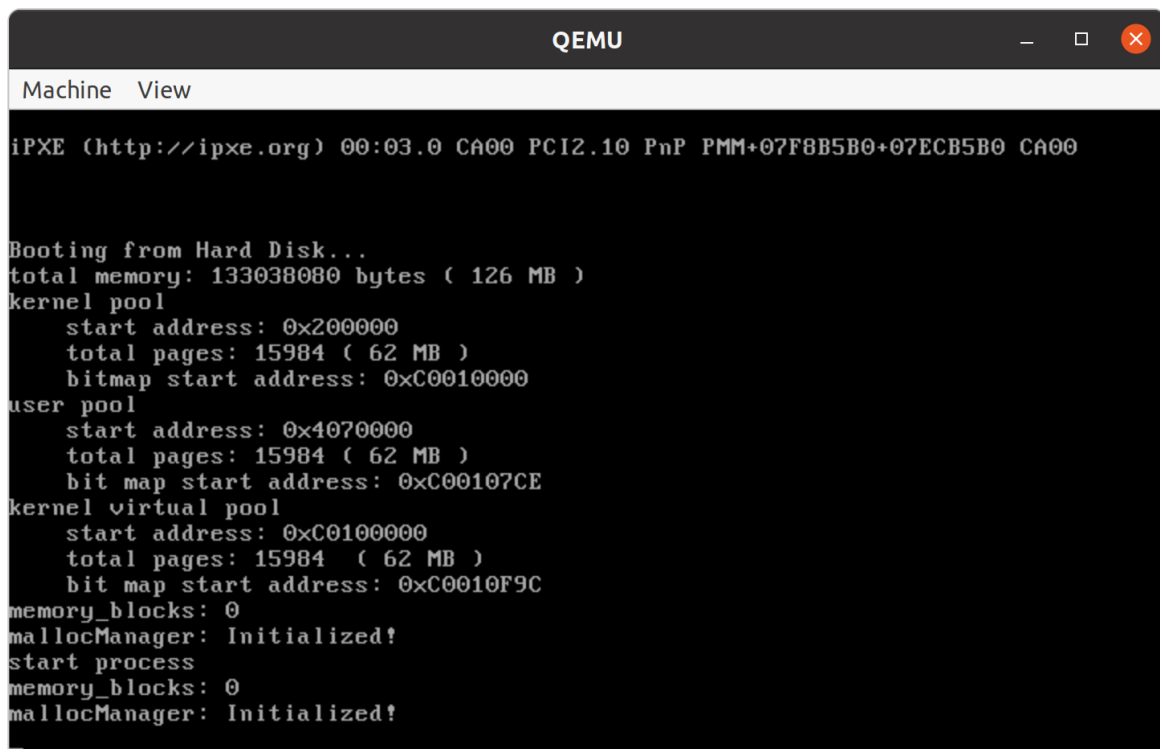
在 `initialize` 函数中加入如下 `printf`, 打印相关信息:

```

void MallocManager::initialize() {
    memory_blocks = 0;
    semaphore.initialize(1);
    printf("memory_blocks: %x\n", memory_blocks);
    printf("mallocManager: Initialized!\n");
}

```

运行结果如下:

A screenshot of a QEMU terminal window. The title bar says 'QEMU' with standard window controls. The terminal output shows the boot process: 'iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B5B0+07ECB5B0 CA00', 'Booting from Hard Disk...', 'total memory: 133038080 bytes (126 MB)', 'kernel pool' with its address and page details, 'user pool' with its address and page details, 'kernel virtual pool' with its address and page details, 'memory_blocks: 0', 'mallocManager: Initialized!', 'start process', 'memory_blocks: 0', and 'mallocManager: Initialized!'.

如图所示，内核与用户 `MallocManager` 分别完成了初始化，`memory_blocks` 指针初始化为空指针。

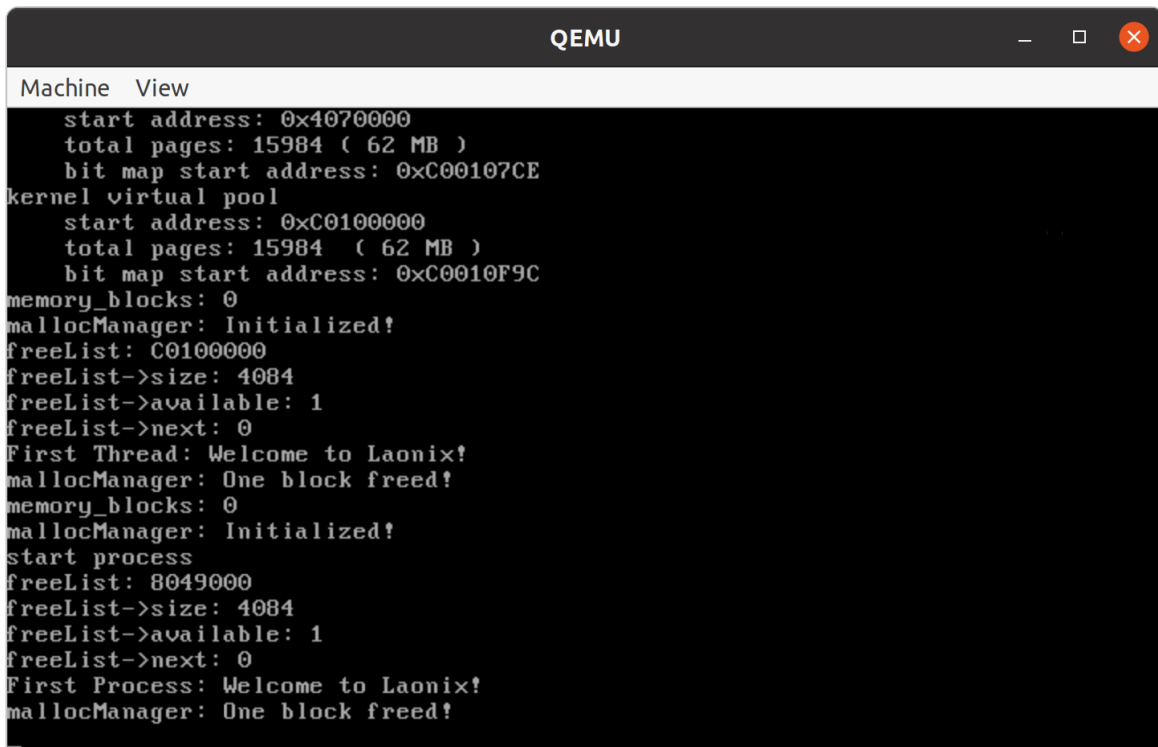
4.2 malloc与free函数

编写测试代码如下：

```
void first_process() {
    printf("start process\n");
    char *p = (char*)malloc(100 * sizeof(char));
    char *string = "First Process: welcome to Laonix!\n";
    for ( int i = 0 ; i < 35; i++ ){
        p[i] = string[i];
    }
    printf("%s", p);
    free(p);
}

void first_thread(void *arg) {
    char *p = (char*)malloc(100 * sizeof(char));
    char *string = "First Thread: welcome to Laonix!\n";
    for ( int i = 0 ; i < 34; i++ ){
        p[i] = string[i];
    }
    printf("%s", p);
    free(p);
    programManager.executeProcess((const char *)first_process, 1);
    asm_halt();
}
```

运行结果如下：



```
Machine View
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
memory_blocks: 0
mallocManager: Initialized!
freeList: C0100000
freeList->size: 4084
freeList->available: 1
freeList->next: 0
First Thread: Welcome to Laonix!
mallocManager: One block freed!
memory_blocks: 0
mallocManager: Initialized!
start process
freeList: 8049000
freeList->size: 4084
freeList->available: 1
freeList->next: 0
First Process: Welcome to Laonix!
mallocManager: One block freed!
```

如图所示，由于是第一次 `malloc`，两个 `MallocManager` 初始化了 `freeList`，打印出了内存块地址，大小为 4084（`PAGE_SIZE = 4096` 减去一个 meta-data block 的大小），目前为 `available` 状态，`next` 为 `null`。

使用 `malloc` 申请了 100 字节的内存，然后在其中存入了一个字符串，最后 `free` 掉内存。屏幕正确打印出了结果。

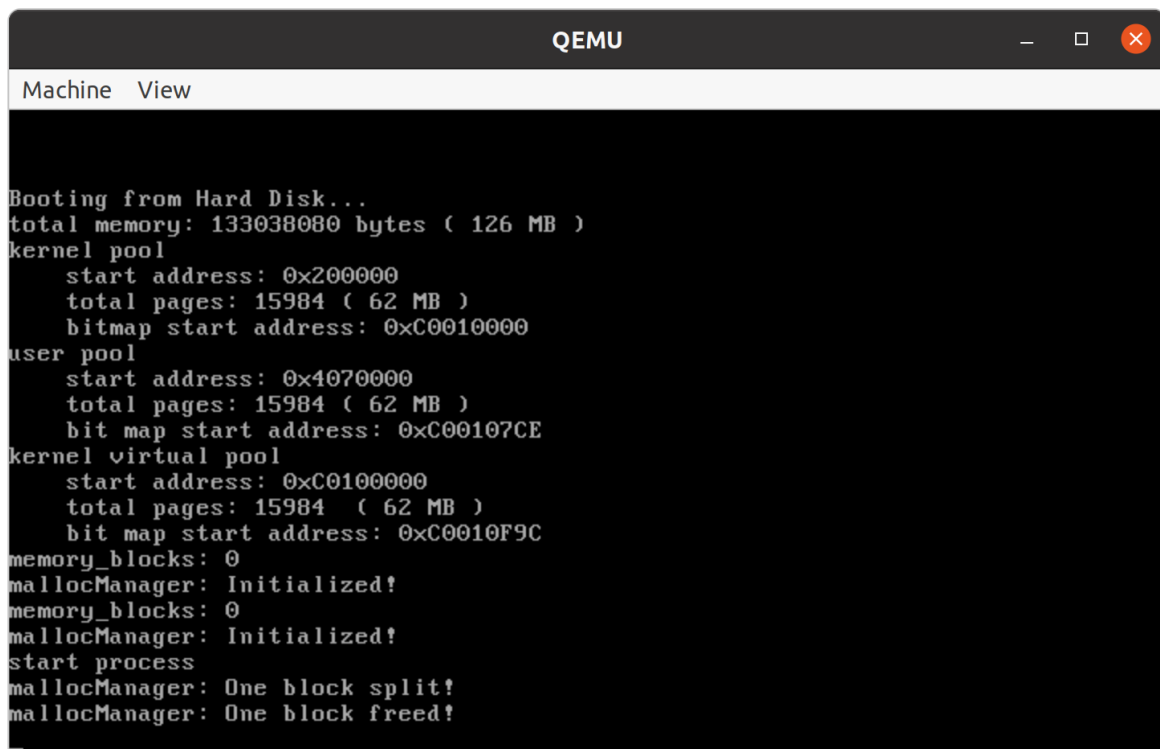
4.3 split和merge函数

编写测试代码如下：

```
void first_process() {
    printf("start process\n");
    char *q = (char*)malloc(100 * sizeof(char));
    free(q);
}
```

并且在 `merge` 和 `split` 函数中加入了 `printf` 打印信息。

运行结果如下：

A screenshot of a QEMU terminal window. The window has a title bar with 'QEMU' and standard window controls. Below the title bar is a menu bar with 'Machine' and 'View'. The main area is a black terminal with white text. The text shows the booting process from a hard disk, memory initialization for kernel and user pools, and the initialization of a mallocManager. The logs include details about memory addresses, page counts, and bitmap start addresses. It also shows memory blocks being split and freed by the mallocManager.

```
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
memory_blocks: 0
mallocManager: Initialized!
memory_blocks: 0
mallocManager: Initialized!
start process
mallocManager: One block split!
mallocManager: One block freed!
```

如图所示，程序首先 `malloc` 了一块 100 字节的内存，`malloc` 函数利用 `split` 函数分别分出这一内存块。

然后我们 `free` 掉内存，释放时调用了 `merge` 函数，将这一块被 `free` 的内存和在它后面的空闲内存合并成一块连续的可用内存。

5. 实验总结

5.1 实验中出现的問題及解决方案

1. make时出现“error: ‘mallocManager’ was not declared in this scope; did you mean ‘MallocManager’?”报错

解决方案：仔细检查代码，发现需要在 `os_modules` 中声明 `mallocManager`，如下：

```
extern MallocManager mallocManager;
```

这样才不会出现报错。

另外在写代码的过程中对于其他的函数等的实现，也要记得在相应的 `.h` 文件中加上声明。

2. 指针所指的地址计算问题

注意指针的地址计算和指针的类型有关，例如以下代码：

```
struct block *curr = (struct block *)ptr-1;
```

说的是先把 `ptr` cast 成 `struct block*` 类型的指针再减小，由于一个 `struct block` 占用 12 字节，那么后面的 `-1` 就是减去 12 字节。

而以下代码：

```
struct block *curr = (struct block *) (ptr-1);
```

则是先把地址先减小再 cast 成 `struct block *` 类型。由于 `ptr` 原来是 `void *` 类型，所以减一的时候是减去一个字节，而不是像前面的例子那样减去 12 个字节。这里要尤其小心。

5.2 实验感想

本次实验是这学期最后一次os实验，又是第一次在没有课程指导教程下的第一次实验，所以开始实现时感觉比较具有挑战性。实验过程中查阅了不少资料，找到了这种比较简洁的 `malloc` 和 `free` 实现方法。遇到了不少困难和bug也在请教老师和助教的过程中——解决。

非常感谢这一学期来老师和助教的悉心指导，能够让我们比理论课更加深入地了解操作系统内部的原理与实现，真的非常不容易。我也要再接再厉，在未来继续努力学习相关领域的课程与内容。