



本科生实验报告

实验课程: _____ 操作系统 _____

实验名称: _____ malloc free _____

专业名称: _____ 计算机科学与技术 _____

学生姓名: _____ 凌国明 _____

学生学号: _____ 21307077 _____

实验地点: _____ 教室 _____

实验成绩: _____

报告时间: _____ 2023. 07. 08 _____

1. 实验过程

- 1) 通过定义系统调用,使得 malloc 和 free 在内核态中运行。首先在 syscall.h 中声明, 在 syscall.call 中实现。

```
// 第5个系统调用, move cursor
void move_cursor(int i, int j);
void syscall_move_cursor(int i, int j);

// malloc and free
int malloc(int bytes);
bool free(int addr, int bytes);

// malloc and free
int malloc(int bytes){
    return memoryManager.allocateBytes(bytes);
}
bool free(int addr, int bytes){
    return memoryManager.releaseBytes(addr, bytes);
}
```

并在 setup.cpp 中设置系统调用, 使系统调用号 6 和 7 分别对应两个函数

```
// 设置6号系统调用
systemService.setSystemCall(6, (int)malloc);
// 设置7号系统调用
systemService.setSystemCall(7, (int)free);
```

- 2) 在 MemoryManager 中声明和实现相应的函数

```
int allocateBytes(int bytes);

bool releaseBytes(int addr, int bytes);
```

- 3) 实现 `allocateBytes`，分两种情况讨论，第一种情况，申请大小小于一页，且刚好有符合要求的“空位”；第二种情况，申请大小大于一页，或者申请大小小于一页但没有符合要求的“空位”。

采用 `list` 来记录一页中的空位（空位理解为一页的内部碎片，但是这个碎片未被使用，所以可以用于新一轮分配）

```
struct ListItem
{
    int start;
    int size;
    ListItem *previous;
    ListItem *next;
};
```

```
class MemoryManager
{
public:
    // 可管理的内存容量
    int totalMemory;
    // 内核物理地址池
    AddressPool kernelPhysical;
    // 用户物理地址池
    AddressPool userPhysical;
    // 内核虚拟地址池
    AddressPool kernelVirtual;
    // 页内空位表
    List empty_list;
```

最开始 `empty_list` 为空。当申请一些页面，且最后一页的空间未被完全使用时，产生内部碎片。可以用一个 `list` 把内部碎片记录下来，用于下次的小额字节内存分配。

申请一些页面后，页面内部碎片无法避免，这个时候 `empty_list` 记录着所有页内部碎片，当 `malloc` 的字节数小于一页大小时：如果有空位符合大小要求，则可以将这部分碎片分配给新的申请需求；如果没有符合大小要求的空位，则要申请新的一页，并将新产生的碎片记录到表中。当 `malloc` 的字节数大于一页大小时，直接分配适量页面，并将最后一页的空缺记录到表中。

```

int MemoryManager::allocateBytes(int bytes){
    // 申请大小 小于 一页
    if(bytes < PAGE_SIZE){
        // 在已分配的页中找空位, 如果找到则分配, 返回。
        ListItem *temp = &empty_list.head;
        int counter = 0;
        while (temp)
        {
            temp = temp->next;
            // 该页中的空位满足大小要求
            if(temp->size >= bytes){
                int addr = temp->start;
                int size = temp->size;
                // 去除这个空位
                empty_list.erase(counter);
                // 增添新的空位
                ListItem a;
                a.start = addr + bytes;
                a.size = size - bytes;
                empty_list.push_back(&a);
                printf("malloc 0 pages, start addr %x\n", addr);
                printf("empty_block_start %x, bytes count %d\n\n", a.start, a.size);
                return addr;
            }
            ++counter;
        }
    }
}

```

首先, 如果 malloc 的字节数小于一页大小, 则遍历 empty_list 记录的所有空位 (代表着页面内部碎片), 如果找到一个符合大小要求的空位, 则分配内存, 并将空位切割成更小的空位。这里找空位相当于用了首次适应的方法, 找 list 中第一个符合大小要求的空位。分配后, 新的空位起始地址为 空位起始地址+申请字节数, 新的空位大小为 原始空位大小-申请大小。注意这里即使大小为 0 也会插入 list, 这是没有考虑到的地方, 可以增加一次判断: 如果大小为 0, 则不插入。

```

// 申请的大小 大于 一页 或 已有的空位都不符合要求
int addr = allocatePages(AddressPoolType::USER, ceil(bytes, PAGE_SIZE) );
// 将最后一页的空位放入表中
ListItem a;
a.start = addr + bytes;
a.size = PAGE_SIZE - bytes % PAGE_SIZE;
empty_list.push_back(&a);
printf("malloc %d pages, start addr %x\n", ceil(bytes, PAGE_SIZE), addr);
printf("empty_block_start %x, bytes count %d\n\n", a.start, a.size);
return addr;

```

如果申请大小大于一页, 或者申请大小小于一页但没有符合要求的空位, 则申请新的页面, 申请的页面数量为 申请字节数量 / 页面大小 再向上取整。将最后一页的碎片加入 list。同样可以增加碎片大小为 0 的判断。

4) free, 注意对应 malloc 的两种情况,

```
bool MemoryManager::releaseBytes(int addr, int bytes){
    // 跨页
    if((addr - 0x8049000) % PAGE_SIZE == 0){
        releasePages(AddressPoolType::USER, addr, ceil(bytes, PAGE_SIZE));
        ListItem *temp = &empty_list.head;
        int counter = 0;
        while (temp)
        {
            temp = temp->next;
            if(temp->start == addr + bytes){
                // 去除这个空位
                empty_list.erase(counter);
                break;
            }
            ++counter;
        }
        printf("free addr %x, free pages %d\n\n", addr, ceil(bytes, PAGE_SIZE));
        return true;
    }
}
```

首先是 malloc 的第二种情况, 直接申请页面。这里直接释放页面, 并遍历 list 中的所有空位, 找到满足要求的空位, 并去除。这里的 $\text{addr}-k$ 代表着 free 的地址和空间起始地址的差值。如果这个差值模页面大小为 0 的话, 说明这是一个页面的起始地址, 那么可以将这些页面都进行释放操作。

```
    else if((addr - 0x8049000) % PAGE_SIZE > bytes){
        ListItem a;
        a.start = addr;
        a.size = bytes;
        empty_list.push_back(&a);
        printf("free addr %x, free pages 0\n\n", addr);
        return true;
    }
    printf("free false\n\n");
    return false;
}
```

然后这里对应 malloc 的第一种情况, 申请的字节数小于一页的大小。这里 $\text{bytes} < (\text{addr}-k)$ 模页面大小的判断, 是为了判断释放的字节是否跨页, 如果跨页则不符合 malloc 中的操作, free 失败。如果不跨页, 说明符合 malloc 的操作, 是合法的, 此时增加空位即可。

2. 实验结果

```
void first_process()
{
    int addr = malloc(2000);
    free(addr, 2000);

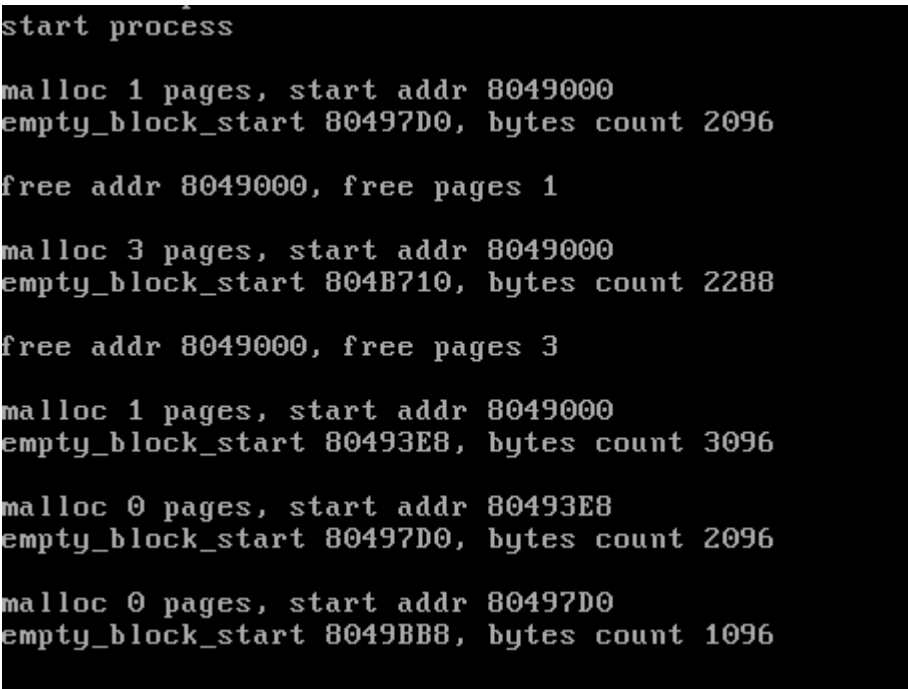
    addr = malloc(10000);
    free(addr, 10000);

    addr = malloc(1000);
    addr = malloc(1000);
    addr = malloc(1000);

}

void first_thread(void *arg)
{
    printf("start process\n\n");
    programManager.executeProcess((const char *)first_process, 1);
    asm_halt();
}
```

在 first_process 中申请，效果如下



```
start process

malloc 1 pages, start addr 8049000
empty_block_start 80497D0, bytes count 2096

free addr 8049000, free pages 1

malloc 3 pages, start addr 8049000
empty_block_start 804B710, bytes count 2288

free addr 8049000, free pages 3

malloc 1 pages, start addr 8049000
empty_block_start 80493E8, bytes count 3096

malloc 0 pages, start addr 80493E8
empty_block_start 80497D0, bytes count 2096

malloc 0 pages, start addr 80497D0
empty_block_start 8049BB8, bytes count 1096
```

先申请 2000 字节，小于一页（4096），没有符合的空位，直接申请一页。申请后直接释放，释放成功。

申请 10000 字节，大于一页（4096），申请 3 页，最后一页有 2288 字节的空位。申请后释放，释放成功，空位列表释放空位成功。

连续申请 3 次 1000 字节，先在 0x804900 处申请 1000，然后在 0x80493E8 后有 3096 个字节的空位；再申请 1000 字节，起始地址为 0x80493E8，更新空位起始地址为 0x497D0，空位大小为 2096 字节；最后申请 1000 字节，更新空位起始地址为 0x8049BB8，空位大小为 1000 字节。基本满足了字节级别的分配，利用好了页面的内部碎片。

3. 总结

这次实验的大致思路是采用 list 来记录一页中的空位（空位理解为一页的内部碎片，但是这个碎片未被使用，所以可以用于新一轮分配）。然后分两种情况讨论 malloc 的实现，并针对这两种情况实现 free。

虽然基本实现了字节级别的内存分配与回收，基本可以利用页面的内部碎片。但是还有以下的缺点：通常每个进程间的地址空间是独立的，这里因为时间原因，没有实现进程间的独立分配，只是把基本逻辑写了出来。还有就是分情况讨论的劣势：考虑的情况较少，导致页面碎片的利用率还不够高，可以再讨论更多的情况，以实现更高的内存利用率。

经过这个学期的操作系统实验，我受益匪浅。理论与实践相结合，我对保护模式、中断机制、内核线程、并发与锁、请求分页，OS 双模态的认识从模糊到深入，理解了操作系统是如何统筹硬件，更好服务用户，对计算机科学的认识更加深入。在这个过程中，我还掌握了 qemu 和 gdb 这两个工具，qemu 用于模拟各种处理器架构，可以用于操作系统运行模拟；gdb 与 nasm 混合使用，可以进行源码级别的断点调试，可以设置断点、步进、查看当前寄存器状态等等，通过这个工具，可以很好地进行系统的调试，方便找出问题所在，也更利于实现原理的理解。总而言之，本学期的操作系统实验使我受益匪浅。