# 操作内存管理

## 1. 基本信息

| 姓名 | 学号 | 班级 | 选择哪个算法 | 是否编译成功 | 是否运行结果正确（逻辑也要正确） |
|---|---|---|---|---|---|
| 庄佳强 | 202121331104 | 计算2114 | 首次适应 | 是 | 是 |

## 2. 记录内存空间使用情况

```
typedef struct memory_Block {
    void* start;  //初始地址
    void* end;    //结束地址
    size_t size;  //大小
    struct memory_Block* next;
    int num;     //进程号
}memoryBlock;

//定义空闲内存块结构体
typedef struct free_Block
{
    void* start;  //初始地址
    void* end;    //结束地址
    size_t size;  //大小
    struct free_Block* next;  //下一个空闲块
}FreeBlock;


//定义全局变量，表示整体的内存空间
void* memory_pool;
size_t memory_size;

//打印内存空闲情况和使用情况
void print_memory_status() {
    printf("内存情况:\n");
    printf("总共内存:%zu bytes\n",memory_size);
    size_t used=0, free=0;
    memoryBlock* memoryblock=used_list;
    FreeBlock* freeblock=free_list;
    //算出使用内存整体的量
    while (memoryblock != NULL) {
        used = used + memoryblock->size;
        memoryblock = memoryblock->next;
    }
    //同理
    while (freeblock != NULL) {
        free = free + freeblock->size;
        freeblock = freeblock->next;
    }
    printf("使用的内存:%zu bytes\n", used);
```

```
    printf("空闲的内存:%zu bytes\n", free);
    printf("使用的内存块:\n");
    memoryblock = used_list;
    freeblock = free_list;
    while (memoryblock != NULL) {
        printf("编号%d的进程，开始地址:%p,结尾地址:%p 大小:%zu bytes\n", memoryblock-
>num,memoryblock->start, memoryblock->end, memoryblock->size);
        memoryblock = memoryblock->next;
    }
    printf("空闲的内存块:\n");
    while (freeblock != NULL) {
        printf("开始地址:%p,结尾地址:%p 大小:%zu bytes\n", freeblock->start,
freeblock->end, freeblock->size);
        freeblock = freeblock->next;
    }
}
```

我把空闲分区和使用分区分开记录，由内存池分配一个空间。把未使用的内存存放记录在空闲内存块链表中，把使用的内存存放记录在使用的链表中。

要访问内存使用情况时只需要遍历链表，讲链表的中元素的值相加就可以就可以算出空闲分区的内存量和使用的内存量。再可以一次打印出每次进程的使用情况。

例:

```
内存情况:
总共内存:1024 bytes
使用的内存:249 bytes
空闲的内存:775 bytes
使用的内存块:
编号0的进程，开始地址:000001AC0FF83C90,结尾地址:000000000FF83D89 大小:249 bytes
空闲的内存块:
开始地址:000000000FF83D89,结尾地址:000000000FF84090 大小:775 bytes
```

# 3. 记录空闲分区

空闲分区链表

```
//定义空闲内存块结构体
typedef struct free_Block
{
    void* start;  //初始地址
    void* end;    //结束地址
    size_t size;  //大小
    struct free_Block* next;  //下一个空闲块
}FreeBlock;

//初始化空闲内存链表
void init_free_list() {
    free_list = malloc(sizeof(FreeBlock));
    if (free_list == NULL) {
        printf("2.失败");
        exit(EXIT_FAILURE);
    }
```

```
    free_list->start = memory_pool;  //开始为内存池的头地址
    free_list->end = (int)memory_pool +memory_size;//结尾为内存池的尾地址
    free_list->size = memory_size;
    free_list->next = NULL;//暂时没有分配

}
```

通过链表来记录空闲区的情况。

# 4. 内存分配算法

## 内存分配：首次适配

```
//分配内存
void* alloc(size_t size, int num) {
    FreeBlock* pre = NULL;
    FreeBlock* cur = free_list;
    while (cur != NULL && cur->size < size) {//寻找符合要求的空闲内存块
        pre = cur;
        cur = cur->next;
    }
    if (cur != NULL) {
        //找到了足够的空间。直接分配
        memoryBlock* block = malloc(sizeof(memoryBlock));
        if (block == NULL) {
            printf("4.失败");
            exit(EXIT_FAILURE);
        }
        block->start = cur->start;
        block->end = (int)cur->start + size;
        block->size = size;
        block->num = num;
        block->next = used_list;//头插法
        used_list = block;
        cur->start = block->end;
        cur->size = cur->size - size;
        printf("编号%d的进程，开始地址:%p,结尾地址:%p 大小:%zu bytes存入内存\n", block-
>num, block->start, block->end, block->size);
        if (cur->size == 0) {
            if (pre == NULL)
                free_list = cur->next;//头改为下一个
            else
                pre->next = cur->next;//跳过cur
            free(cur);
        }

        //更新页表链表
        Page_Table* ppre = NULL;
        Page_Table* pcur = page_table;
        while (pcur != NULL && pcur->num != block->num) {//寻找链表是否有这个进程的页
表
            ppre = pcur;
            pcur = pcur->next;
        }
        if (pcur != NULL) {//有添加
            for (size_t i = 0; i < page_table_size; i++) {
```

```
                if (pcur->page[i] == 0) {//如果页为空
                    pcur->page[i] = block->start;//存入物理地址
                    printf("物理地址%p的内存块的地址存储在第%zu页中\n", block->start,
i);
                    break;
                }
            }
        }
        else {//没有新建
            Page_Table* page_Table_new = malloc(sizeof(Page_Table));//分配内存
            memset(page_Table_new->page, 0, sizeof(page_Table_new->page));//页表设
置为0
            page_Table_new->num = block->num;//写入编号
            page_Table_new->page[0] = block->start;//写入开始地址
            page_Table_new->next = page_table;//头插
            page_table = page_Table_new;
            printf("物理地址%p的内存块的地址存储在第0页中\n", block->start);
        }
        return block->start;//返回开始地址
    }
    else {
        printf("编号%d的进程 大小:%zu bytes存入内存\n",num,size);
        printf("没有足够的空间，使用最先适配算法从使用的链表中释放一个最早分配的内存块释放
\n");
        //没有足够的空间，使用最先适配算法从使用的链表中释放一个最早分配的内存块释放
        memoryBlock* mprev = NULL;
        memoryBlock* mcur = used_list;
        //头插法的头在尾部
        while (mcur != NULL && mcur->size < size) {//找到头部
            mprev = mcur;
            mcur = mcur->next;
        }
        if (mcur != NULL) {//调用删除操作
            dealloc(mcur->start);//调用内存释放函数
            return alloc(size, num);//再次尝试
        }

    }
}
```

算法中会遍历空闲的内存块，寻找一个大小符合要求的内存块。如果找到了，则直接分配；否则，使用最先适配算法从已使用的内存块中找到一个最早分配的内存块进行回收。

如果找到了大小符合要求的空闲内存块，则直接分配该内存块。先会创建一个新的内存块存放消息包括开始地址，结束地址等等，并将其添加到已使用的内存块链表中。然后，更新空闲内存块的信息，并将该内存块的起始地址存储到页表中。最后，返回该内存块的起始地址。

如果没有找到符合要求的空闲内存块，则使用首次适配算法从已使用的内存块中找到一个最先分配的内存块进行回收。先会将该内存块的信息添加到空闲内存块链表中，并将其从已使用的内存块链表中删除。然后，更新页表中该内存块的信息，并使用递归再次尝试分配内存。

## 虚拟内存分配：分页

```
//定义页表,用于把逻辑地址转化为物理地址
//一个进程一张页表
typedef struct page_Table
{
    int num;//进程号
    void* page[10];//页表内容
    struct page_Table* next;//下一个指针域
}Page_Table;
size_t page_table_size;//页表大小



//初始化页表
void init_page_table(size_t size) {
    page_table_size = size;
    page_table = NULL;

}

//分配完内存后
Page_Table* pcur = page_table;
        while (pcur != NULL && pcur->num != block->num) {
            ppre = pcur;
            pcur = pcur->next;
        }
        if (pcur != NULL) {
            for (size_t i = 0; i < page_table_size; i++) {
                if (pcur->page[i] == 0) {//如果页为空
                    pcur->page[i] = block->start;//存入物理地址
                    printf("物理地址%p的内存块的地址存储在第%zu页中\n", block->start,
i);
                    break;
                }
            }
        }
        else {
            Page_Table* page_Table_new = malloc(sizeof(Page_Table));//分配内存
            memset(page_Table_new->page, 0, sizeof(page_Table_new->page));//页表设
置为0
            page_Table_new->num = block->num;//写入编号
            page_Table_new->page[0] = block->start;//写入开始地址
            page_Table_new->next = page_table;//头插
            page_table = page_Table_new;
            printf("物理地址%p的内存块的地址存储在第0页中\n", block->start);
        }


//释放使用的内存空间后，释放页表内容
Page_Table* ppre = NULL;
        Page_Table* pcur = page_table;
        while (pcur != NULL && pcur->num != cur->num) {
            ppre = pcur;
            pcur = pcur->next;
        }
        if (pcur != NULL) {
```

```
            for (size_t i = 0; i < page_table_size; i++)
            {
                if (pcur->page[i] == cur->start) {
                    pcur->page[i] =0;
                    printf("编号%d的进程,物理地址%p的内存块的地址从第%zu页中释放\n",
cur->num, cur->start, i);
                    break;
                }
            }
            int flag = 0;
            for (size_t i = 0; i < page_table_size; i++) {
                if (pcur->page[i] != 0)
                    flag = 1;
            }
            if (flag == 0) {
                if (ppre = NULL) {
                    page_table = pcur->next;
                }
                else {
                    ppre->next = pcur->next;
                }
                free(pcur);
            }
        }



//打印页表
Page_Table* pcur = page_table;
    while (pcur != NULL) {
        printf("%d的页表：\n", pcur->num);
        for (size_t i = 0; i < page_table_size; i++) {
            if (pcur->page[i] == NULL) {}
            //printf("Page %zu free\n", i);
            else
                printf("Page %zu alloc (开始地址：%p)\n", i, pcur->page[i]);
        }
        printf("\n");
        pcur = pcur->next;
    }
}
```

定义了页表和页表大小。页表是一个链表，每个节点代表一个进程的页表。每个节点包括进程号、页表内容和下一个节点的指针。页表大小表示页表中有多少个页。

之后初始化页表，设置一个表头。

分配完内存后，先遍历页表链表，如果找到了对应进程的页表，则在页表中查找空闲页。如果找到了一个空闲页，则将该内存块的起始地址存储在该页中，并输出该内存块存储在哪个页中。如果没有找到对应进程的页表，则创建一个新的页表节点。并将该节点的页表内容设置为0。然后，将该节点的进程号设置为num ，将该内存块的起始地址存储在该节点的第一个页中， 并将该节点插入到页表的头部。同时，输出该内存块存储在哪个页中。

释放内存后，遍历页表并找到对应进程的页表，如果找到了对应进程的页表，则在页表中查找该内存块对应的页，并将该页释放。如果该页表中没有其他内存块，则删除该页表节点。之后会遍历该页表的所有页，如果发现该页表中还有其他内存块，则将flag 设置为1。如果flag仍然为0 ，则说明该页表中没有其他内存块此时，将该节点从链表中删除，并释放该节点的内存。

```
//释放内存

void dealloc(void* ptr) {
    memoryBlock* pre = NULL;
    memoryBlock* cur = used_list;
    while (cur!=NULL&&cur->start!=ptr)
    {
        pre = cur;
        cur = cur->next;
    }
    if (cur != NULL) {
        //找到了要释放的内存块
        if (pre == NULL)//链表头
            used_list = cur->next;
        else
            pre->next = cur->next;
        FreeBlock* freeblock = malloc(sizeof(FreeBlock));
        if (freeblock == NULL) {
            printf("6.失败");
            exit(EXIT_FAILURE);
        }
        //回收回来的内存为碎片的
        printf("编号%d的进程，开始地址:%p,结尾地址:%p 大小:%zu bytes结束进程\n", cur-
>num, cur->start, cur->end, cur->size);
        freeblock->start = (int)cur->start;
        freeblock->end = cur->end;
        freeblock->size = cur->size;
        freeblock->next = free_list;//头插法
        free_list = freeblock;
        //更新页表

        /*for (size_t i = 0; i < page_table_size; i++) {
            if (page_table[i] == cur->start) {
                page_table[i] = NULL;
                printf("编号%d的进程,物理地址%p的内存块的地址从第%zu页中释放\n", cur-
>num,cur->start,i);
                break;
            }
        }*/
        Page_Table* ppre = NULL;
        Page_Table* pcur = page_table;
        while (pcur != NULL && pcur->num != cur->num) {
            ppre = pcur;
            pcur = pcur->next;
        }
        if (pcur != NULL) {
            for (size_t i = 0; i < page_table_size; i++)
            {
                if (pcur->page[i] == cur->start) {
                    pcur->page[i] =0;
```

```
                    printf("编号%d的进程,物理地址%p的内存块的地址从第%zu页中释放\n",
cur->num, cur->start, i);
                    break;
                }
            }
            int flag = 0;
            for (size_t i = 0; i < page_table_size; i++) {
                if (pcur->page[i] != 0)
                    flag = 1;
            }
            if (flag == 0) {
                if (ppre = NULL) {
                    page_table = pcur->next;
                }
                else {
                    ppre->next = pcur->next;
                }
                //free(pcur);
            }
        }
        //合并相邻的空闲内存块
        FreeBlock* pre = NULL;
        FreeBlock* cur = free_list;
        FreeBlock* temp = free_list;
        while (temp != NULL) {
            while (cur != NULL) {
                if (pre!=NULL&&temp->end == cur->start)//判断是否相连在一起
                {
                    temp->end = cur->end;
                    temp->size = temp->size + cur->size;
                    pre->next = cur->next;
                    free(cur);
                    cur = pre->next;
                }
                else
                {
                    pre = cur;
                    cur = cur->next;
                }
            }
            temp = temp->next;
            pre = NULL;
            cur = free_list;
        }

    }
}
//结束进程后，清空内存
void dealloc_all(int num) {
    memoryBlock* cur = used_list;
    while (cur != NULL  && cur->num != num)
    {
        cur = cur->next;
    }
    if (cur != NULL) {
        dealloc(cur->start);
        dealloc_all(num);
    }
```

```
}
```

内存清空的算法我分为两种模式，一种是指清除一个内存块，减少内存负担，一种是进程结束，释放掉所有内存。

改算法中会遍历已使用的内存块，寻找要释放的内存块。如果找到了，则将其从已使用的内存块链表中删除，并将其信息添加到空闲内存块链表中。否者结束。

找到之后会将要释放的内存块的信息，新建一个空闲内存块，把如开始地址，结束地址和大小添加空闲内存块中，在把内存块头插到空闲内存块链表中，并更新算出页表中该内存块的信息。

接着要合并相邻的内存块。算法会遍历空闲内存块链表，合并相邻的空闲内存块。具体来说，该部分代码会判断相邻的两个空闲内存块是否相连在一起，如果是，则将它们合并成一个更大的空闲内存块，修改开始地址和结束地址和大小。

# 6. 测试

**（1）产生测试数据**

随机为4个进程分别分配和释放内存10次以上，即随机产生10组以上数据：（进程Pi 分配内存大小） 或者 （进程Pi结束）

这个测试要求我还是没怎么懂，就按我的理解来。我先构建4个进程，然后随机分配内存大小给四个进程中随机一个，重复10次，其中要是空间不足会自动执行首次适用算法，然后打印出内存状态和页表，最后我在手动结束一个进程来展示效果。

```
int main() {
    init_memory_pool(1024);//初始化内存池
    init_free_list();//初始化空闲列表
    init_used_list();//初始化使用列表
    init_page_table(10);//初始化页表
    void* ptr[4]; //模拟进程
    srand((unsigned int)time(NULL));
    int size,num;
    for (int i = 0; i < 10; i++) {
        size= rand() % 300 + 1;//随机内存
        num = rand() % 4;//随机进程号
        ptr[num]=alloc(size, num);
        print_memory_status();
        printf("\n");
        printf("\n");
    }
    dealloc(ptr[0]);//清除一个内存块
    dealloc_all(1);//清除进程所有内存块
    print_memory_status();
    printf("\n");
    print_page_table();
    printf("通过页表找内存块，删除所有的使用内存\n");
    Page_Table* cur=page_table;
    while (cur!= NULL) {
        for (size_t i = 0; i < page_table_size; i++) {
            if (cur->page[i] == 0) {}
            else
                dealloc(cur->page[i]);
```

```
        }
        cur = cur->next;
    }
    print_memory_status();
    print_page_table();

}
```

结果

```
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes存入内存
物理地址0000023DE1773C60的内存块的地址存储在第0页中
内存情况:
总共内存:1024 bytes
使用的内存:42 bytes
空闲的内存:982 bytes
使用的内存块:
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1774060 大小:982 bytes


编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes存入内存
物理地址FFFFFFFFE1773C8A的内存块的地址存储在第0页中
内存情况:
总共内存:1024 bytes
使用的内存:251 bytes
空闲的内存:773 bytes
使用的内存块:
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1774060 大小:773 bytes


编号3的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E69 大小:270 bytes存入内存
物理地址FFFFFFFFE1773D5B的内存块的地址存储在第1页中
内存情况:
总共内存:1024 bytes
使用的内存:521 bytes
空闲的内存:503 bytes
使用的内存块:
编号3的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E69 大小:270 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1774060 大小:503 bytes


编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F95 大小:300 bytes存入内存
物理地址FFFFFFFFE1773E69的内存块的地址存储在第1页中
内存情况:
总共内存:1024 bytes
使用的内存:821 bytes
空闲的内存:203 bytes
使用的内存块:
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F95 大小:300 bytes
```

```
编号3的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E69 大小:270 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1774060 大小:203 bytes


编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes存入内存
物理地址FFFFFFFFE1773F95的内存块的地址存储在第2页中
内存情况:
总共内存:1024 bytes
使用的内存:827 bytes
空闲的内存:197 bytes
使用的内存块:
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F95 大小:300 bytes
编号3的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E69 大小:270 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE1774060 大小:197 bytes


编号0的进程 大小:210 bytes存入内存
没有足够的空间，使用最先适配算法从使用的链表中释放一个最早分配的内存块释放
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F95 大小:300 bytes结束进程
编号0的进程,物理地址FFFFFFFFE1773E69的内存块的地址从第1页中释放
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes存入内存
物理地址FFFFFFFFE1773E69的内存块的地址存储在第1页中
内存情况:
总共内存:1024 bytes
使用的内存:737 bytes
空闲的内存:287 bytes
使用的内存块:
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes
编号3的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E69 大小:270 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773F3B,结尾地址:FFFFFFFFE1773F95 大小:90 bytes
开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE1774060 大小:197 bytes


编号2的进程 大小:267 bytes存入内存
没有足够的空间，使用最先适配算法从使用的链表中释放一个最早分配的内存块释放
编号3的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E69 大小:270 bytes结束进程
编号3的进程,物理地址FFFFFFFFE1773D5B的内存块的地址从第1页中释放
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E66 大小:267 bytes存入内存
物理地址FFFFFFFFE1773D5B的内存块的地址存储在第0页中
内存情况:
总共内存:1024 bytes
使用的内存:734 bytes
空闲的内存:290 bytes
使用的内存块:
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E66 大小:267 bytes
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes
```

```
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773E66,结尾地址:FFFFFFFFE1773E69 大小:3 bytes
开始地址:FFFFFFFFE1773F3B,结尾地址:FFFFFFFFE1773F95 大小:90 bytes
开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE1774060 大小:197 bytes


编号0的进程，开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE177401F 大小:132 bytes存入内存
物理地址FFFFFFFFE1773F9B的内存块的地址存储在第2页中
内存情况:
总共内存:1024 bytes
使用的内存:866 bytes
空闲的内存:158 bytes
使用的内存块:
编号0的进程，开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE177401F 大小:132 bytes
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E66 大小:267 bytes
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773E66,结尾地址:FFFFFFFFE1773E69 大小:3 bytes
开始地址:FFFFFFFFE1773F3B,结尾地址:FFFFFFFFE1773F95 大小:90 bytes
开始地址:FFFFFFFFE177401F,结尾地址:FFFFFFFFE1774060 大小:65 bytes


编号2的进程 大小:214 bytes存入内存
没有足够的空间，使用最先适配算法从使用的链表中释放一个最早分配的内存块释放
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E66 大小:267 bytes结束进程
编号2的进程,物理地址FFFFFFFFE1773D5B的内存块的地址从第0页中释放
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E31 大小:214 bytes存入内存
物理地址FFFFFFFFE1773D5B的内存块的地址存储在第0页中
内存情况:
总共内存:1024 bytes
使用的内存:813 bytes
空闲的内存:211 bytes
使用的内存块:
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E31 大小:214 bytes
编号0的进程，开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE177401F 大小:132 bytes
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773E31,结尾地址:FFFFFFFFE1773E69 大小:56 bytes
开始地址:FFFFFFFFE1773F3B,结尾地址:FFFFFFFFE1773F95 大小:90 bytes
开始地址:FFFFFFFFE177401F,结尾地址:FFFFFFFFE1774060 大小:65 bytes


编号1的进程 大小:267 bytes存入内存
没有足够的空间，使用最先适配算法从使用的链表中释放一个最早分配的内存块释放
分配失败
内存情况:
总共内存:1024 bytes
使用的内存:813 bytes
空闲的内存:211 bytes
使用的内存块:
```

```
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E31 大小:214 bytes
编号0的进程，开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE177401F 大小:132 bytes
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773E31,结尾地址:FFFFFFFFE1773E69 大小:56 bytes
开始地址:FFFFFFFFE1773F3B,结尾地址:FFFFFFFFE1773F95 大小:90 bytes
开始地址:FFFFFFFFE177401F,结尾地址:FFFFFFFFE1774060 大小:65 bytes


编号0的进程，开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE177401F 大小:132 bytes结束进程
编号0的进程,物理地址FFFFFFFFE1773F9B的内存块的地址从第2页中释放
内存情况:
总共内存:1024 bytes
使用的内存:681 bytes
空闲的内存:343 bytes
使用的内存块:
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E31 大小:214 bytes
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE1774060 大小:197 bytes
开始地址:FFFFFFFFE1773E31,结尾地址:FFFFFFFFE1773E69 大小:56 bytes
开始地址:FFFFFFFFE1773F3B,结尾地址:FFFFFFFFE1773F95 大小:90 bytes

2的页表：
Page 0 alloc（开始地址：FFFFFFFFE1773D5B）

0的页表：
Page 0 alloc（开始地址：FFFFFFFFE1773C8A）
Page 1 alloc（开始地址：FFFFFFFFE1773E69）

3的页表：
Page 0 alloc（开始地址：0000023DE1773C60）
Page 2 alloc（开始地址：FFFFFFFFE1773F95）

通过页表找内存块，删除所有的使用内存
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E31 大小:214 bytes结束进程
编号2的进程,物理地址FFFFFFFFE1773D5B的内存块的地址从第0页中释放
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes结束进程
编号0的进程,物理地址FFFFFFFFE1773C8A的内存块的地址从第0页中释放
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes结束进程
编号0的进程,物理地址FFFFFFFFE1773E69的内存块的地址从第1页中释放
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes结束进程
编号3的进程,物理地址0000023DE1773C60的内存块的地址从第0页中释放
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes结束进程
编号3的进程,物理地址FFFFFFFFE1773F95的内存块的地址从第2页中释放
内存情况:
总共内存:1024 bytes
使用的内存:0 bytes
空闲的内存:1024 bytes
使用的内存块:
空闲的内存块:
开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1774060 大小:203 bytes
```

```
开始地址:FFFFFFFFE1773C60,结尾地址:FFFFFFFFE1773F95 大小:821 bytes
```

**（2）解释结果**

每一次内存分配或释放，内存的示意图是怎样的。给出4组分析即可。

```
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes存入内存
物理地址0000023DE1773C60的内存块的地址存储在第0页中
内存情况:
总共内存:1024 bytes
使用的内存:42 bytes
空闲的内存:982 bytes
使用的内存块:
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1774060 大小:982 bytes
```

一开始没有进程，编号为3的进程进入，显示内存情况，再打印出使用的内存块和空闲的内存块，最后打印出页表。

```
编号3的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E69 大小:270 bytes存入内存
物理地址FFFFFFFFE1773D5B的内存块的地址存储在第1页中
内存情况:
总共内存:1024 bytes
使用的内存:521 bytes
空闲的内存:503 bytes
使用的内存块:
编号3的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E69 大小:270 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1774060 大小:503 bytes
```

在这里中，编号为3的进程追加了内存，理应给他分配内存。

```
编号2的进程 大小:267 bytes存入内存
没有足够的空间，使用最先适配算法从使用的链表中释放一个最早分配的内存块释放
编号3的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E69 大小:270 bytes结束进程
编号3的进程,物理地址FFFFFFFFE1773D5B的内存块的地址从第1页中释放
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E66 大小:267 bytes存入内存
物理地址FFFFFFFFE1773D5B的内存块的地址存储在第0页中
内存情况:
总共内存:1024 bytes
使用的内存:734 bytes
空闲的内存:290 bytes
使用的内存块:
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E66 大小:267 bytes
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes
空闲的内存块:
开始地址:FFFFFFFFE1773E66,结尾地址:FFFFFFFFE1773E69 大小:3 bytes
开始地址:FFFFFFFFE1773F3B,结尾地址:FFFFFFFFE1773F95 大小:90 bytes
```

```
开始地址:FFFFFFFFE1773F9B,结尾地址:FFFFFFFFE1774060 大小:197 bytes
```

这里因为没有空闲内存区中没有足够的内存，使用首次适应算法给他分配了一块足够的内存，同时更新使用的内存块和空闲内存块。

```
编号1的进程，开始地址:FFFFFFFF99B52A10,结尾地址:FFFFFFFF99B52AAD 大小:157 bytes结束进程
编号1的进程,物理地址FFFFFFFF99B52A10的内存块的地址从第1页中释放
编号1的进程，开始地址:FFFFFFFF99B52D22,结尾地址:FFFFFFFF99B52D2D 大小:11 bytes结束进程
编号1的进程,物理地址FFFFFFFF99B52D22的内存块的地址从第5页中释放
编号1的进程，开始地址:FFFFFFFF99B52D01,结尾地址:FFFFFFFF99B52D22 大小:33 bytes结束进程
编号1的进程,物理地址FFFFFFFF99B52D01的内存块的地址从第4页中释放
编号1的进程，开始地址:0000020699B529C0,结尾地址:FFFFFFFF99B52A10 大小:80 bytes结束进程
编号1的进程,物理地址0000020699B529C0的内存块的地址从第0页中释放
内存情况:
总共内存:1024 bytes
使用的内存:516 bytes
空闲的内存:508 bytes
使用的内存块:
编号0的进程，开始地址:FFFFFFFF99B52D2D,结尾地址:FFFFFFFF99B52D63 大小:54 bytes
编号2的进程，开始地址:FFFFFFFF99B52BDF,结尾地址:FFFFFFFF99B52D01 大小:290 bytes
编号0的进程，开始地址:FFFFFFFF99B52B33,结尾地址:FFFFFFFF99B52BDF 大小:172 bytes
空闲的内存块:
开始地址:FFFFFFFF99B529C0,结尾地址:FFFFFFFF99B52B33 大小:371 bytes
开始地址:FFFFFFFF99B52D01,结尾地址:FFFFFFFF99B52D2D 大小:44 bytes
开始地址:FFFFFFFF99B52D63,结尾地址:FFFFFFFF99B52DC0 大小:93 bytes
```

这是类外一个例子中,执行了`dealloc_all(1);`所有删除了所有的进程1的内存块，实现了进程结束，释放所有的内存。

```
剩下的:
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E31 大小:214 bytes
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes

通过页表找内存块，删除所有的使用内存
编号2的进程，开始地址:FFFFFFFFE1773D5B,结尾地址:FFFFFFFFE1773E31 大小:214 bytes结束进程
编号2的进程,物理地址FFFFFFFFE1773D5B的内存块的地址从第0页中释放
编号0的进程，开始地址:FFFFFFFFE1773C8A,结尾地址:FFFFFFFFE1773D5B 大小:209 bytes结束进程
编号0的进程,物理地址FFFFFFFFE1773C8A的内存块的地址从第0页中释放
编号0的进程，开始地址:FFFFFFFFE1773E69,结尾地址:FFFFFFFFE1773F3B 大小:210 bytes结束进程
编号0的进程,物理地址FFFFFFFFE1773E69的内存块的地址从第1页中释放
编号3的进程，开始地址:0000023DE1773C60,结尾地址:FFFFFFFFE1773C8A 大小:42 bytes结束进程
编号3的进程,物理地址0000023DE1773C60的内存块的地址从第0页中释放
编号3的进程，开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1773F9B 大小:6 bytes结束进程
编号3的进程,物理地址FFFFFFFFE1773F95的内存块的地址从第2页中释放
内存情况:
总共内存:1024 bytes
使用的内存:0 bytes
空闲的内存:1024 bytes
使用的内存块:
空闲的内存块:
开始地址:FFFFFFFFE1773F95,结尾地址:FFFFFFFFE1774060 大小:203 bytes
开始地址:FFFFFFFFE1773C60,结尾地址:FFFFFFFFE1773F95 大小:821 bytes
```

这里使用了页表来找内存，通过逻辑地址找到了物理地址，然后删除了所有的内存块，添加到空闲内存块。

但这里还有一点小bug,无法彻底的把空闲内存块合并成一个完整的，因为在遍历的时候没办法彻底的遍历，因为这两个开始地址和结束地址在链表顺序中是反着的，所有如果想要遍历到，好的解法就得使用双向链表，但这要几乎重构结构体，那同时要重构很多东西，太麻烦了，这里实验先略过，这也警醒我下次要考虑多一些东西。

# 7.整体代码

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<string.h>


//定义内存块结构体
typedef struct memory_Block {
    void* start;  //初始地址
    void* end;    //结束地址
    size_t size;  //大小
    struct memory_Block* next;
    int num;      //进程号
}memoryBlock;

//定义空闲内存块结构体
typedef struct free_Block
{
    void* start;  //初始地址
    void* end;    //结束地址
    size_t size;  //大小
    struct free_Block* next;  //下一个空闲块
}FreeBlock;

//定义页表,用于把逻辑地址转化为物理地址
//一个进程一张页表
typedef struct page_Table
{
    int num;
    void* page[10];
    struct page_Table* next;
}Page_Table;
size_t page_table_size;

//定义全局变量，表示整体的内存空间
void* memory_pool;
size_t memory_size;

//定义已使用内存链表的头指针和空闲内存链表的头指针
memoryBlock* used_list;
FreeBlock* free_list;
Page_Table* page_table;
```

```
void dealloc(void* ptr);

//初始化内存池
void init_memory_pool(size_t size) {
    memory_size = size;
    memory_pool = malloc(memory_size);//存放分配空间初始地址
    if (memory_pool == NULL) {
        printf("1.失败");
        exit(EXIT_FAILURE);
    }
}
//初始化空闲内存链表
void init_free_list() {
    free_list = malloc(sizeof(FreeBlock));
    if (free_list == NULL) {
        printf("2.失败");
        exit(EXIT_FAILURE);
    }
    free_list->start = memory_pool;  //开始为内存池的头地址
    free_list->end = (int)memory_pool +memory_size;//结尾为内存池的尾地址
    free_list->size = memory_size;
    free_list->next = NULL;//暂时没有分配

}

//初始化使用链表
void init_used_list() {
    used_list = NULL;
}

//初始化页表
void init_page_table(size_t size) {
    page_table_size = size;
    page_table = NULL;

}

//分配内存
void* alloc(size_t size, int num) {
    FreeBlock* pre = NULL;
    FreeBlock* cur = free_list;
    while (cur != NULL && cur->size < size) {//寻找符合要求的空闲内存块
        pre = cur;
        cur = cur->next;
    }
    if (cur != NULL) {
        //找到了足够的空间。直接分配
        memoryBlock* block = malloc(sizeof(memoryBlock));
        if (block == NULL) {
            printf("4.失败");
            exit(EXIT_FAILURE);
        }
        block->start = cur->start;
        block->end = (int)cur->start + size;
        block->size = size;
        block->num = num;
        block->next = used_list;//头插法
```

```
        used_list = block;
        cur->start = block->end;
        cur->size = cur->size - size;
        printf("编号%d的进程，开始地址:%p,结尾地址:%p 大小:%zu bytes存入内存\n", block->num, block->start, block->end, block->size);
        if (cur->size == 0) {
            if (pre == NULL)
                free_list = cur->next;//头改为下一个
            else
                pre->next = cur->next;//跳过cur
            free(cur);
        }

        //更新页表
        //for (size_t i = 0; i < page_table_size; i++) {
        //  if (page_table[i] == NULL) {//如果页为空
        //      page_table[i] = block->start;//存入物理地址
        //      printf("物理地址%p的内存块的地址存储在第%zu页中\n", block->start, i);
        //      break;
        //  }
        //}
        Page_Table* ppre = NULL;
        Page_Table* pcur = page_table;
        while (pcur != NULL && pcur->num != block->num) {
            ppre = pcur;
            pcur = pcur->next;
        }
        if (pcur != NULL) {
            for (size_t i = 0; i < page_table_size; i++) {
                if (pcur->page[i] == 0) {//如果页为空
                    pcur->page[i] = block->start;//存入物理地址
                    printf("物理地址%p的内存块的地址存储在第%zu页中\n", block->start, i);
                    break;
                }
            }
        }
        else {
            Page_Table* page_Table_new = malloc(sizeof(Page_Table));//分配内存
            memset(page_Table_new->page, 0, sizeof(page_Table_new->page));//页表设置为0
            page_Table_new->num = block->num;//写入编号
            page_Table_new->page[0] = block->start;//写入开始地址
            page_Table_new->next = page_table;//头插
            page_table = page_Table_new;
            printf("物理地址%p的内存块的地址存储在第0页中\n", block->start);
        }
        return block->start;//返回开始地址
    }
    else {
        printf("编号%d的进程 大小:%zu bytes存入内存\n",num,size);
        printf("没有足够的空间，使用最先适配算法从使用的链表中释放一个最早分配的内存块释放\n");
        //没有足够的空间，使用最先适配算法从使用的链表中释放一个最早分配的内存块释放
        memoryBlock* mprev = NULL;
        memoryBlock* mcur = used_list;
        //头插法的头在尾部
        while (mcur != NULL && mcur->size < size) {//找到头部
```

```
            mprev = mcur;
            mcur = mcur->next;
        }
        if (mcur != NULL) {//调用删除操作
            dealloc(mcur->start);
            return alloc(size, num);//再次尝试
        }
        else {
            printf("分配失败\n");
        }
    }

}


//释放内存

void dealloc(void* ptr) {
    memoryBlock* pre = NULL;
    memoryBlock* cur = used_list;
    while (cur!=NULL&&cur->start!=ptr)
    {
        pre = cur;
        cur = cur->next;
    }
    if (cur != NULL) {
        //找到了要释放的内存块
        if (pre == NULL)//链表头
            used_list = cur->next;
        else
            pre->next = cur->next;
        FreeBlock* freeblock = malloc(sizeof(FreeBlock));
        if (freeblock == NULL) {
            printf("6.失败");
            exit(EXIT_FAILURE);
        }
        //回收回来的内存为碎片的
        printf("编号%d的进程，开始地址:%p,结尾地址:%p 大小:%zu bytes结束进程\n", cur-
>num, cur->start, cur->end, cur->size);
        freeblock->start = (int)cur->start;
        freeblock->end = cur->end;
        freeblock->size = cur->size;
        freeblock->next = free_list;//头插法
        free_list = freeblock;
        //更新页表

        /*for (size_t i = 0; i < page_table_size; i++) {
            if (page_table[i] == cur->start) {
                page_table[i] = NULL;
                printf("编号%d的进程,物理地址%p的内存块的地址从第%zu页中释放\n", cur-
>num,cur->start,i);
                break;
            }
        }*/
        Page_Table* ppre = NULL;
        Page_Table* pcur = page_table;
        while (pcur != NULL && pcur->num != cur->num) {
            ppre = pcur;
```

```
            pcur = pcur->next;
        }
        if (pcur != NULL) {
            for (size_t i = 0; i < page_table_size; i++)
            {
                if (pcur->page[i] == cur->start) {
                    pcur->page[i] =0;
                    printf("编号%d的进程,物理地址%p的内存块的地址从第%zu页中释放\n",
cur->num, cur->start, i);
                    break;
                }
            }
            int flag = 0;
            for (size_t i = 0; i < page_table_size; i++) {
                if (pcur->page[i] != 0)
                    flag = 1;
            }
            if (flag == 0) {
                if (ppre == NULL) {
                    page_table = pcur->next;
                }
                else {
                    ppre->next = pcur->next;
                }
                //free(pcur);
            }
        }
        //合并相邻的空闲内存块
        FreeBlock* pre = NULL;
        FreeBlock* cur = free_list;
        FreeBlock* temp = free_list;
        while (temp != NULL) {
            while (cur != NULL) {
                if (pre!=NULL&&temp->end == cur->start)//判断是否相连在一起
                {
                    temp->end = cur->end;
                    temp->size = temp->size + cur->size;
                    pre->next = cur->next;
                    free(cur);
                    cur = pre->next;
                }
                else
                {
                    pre = cur;
                    cur = cur->next;
                }
            }
            temp = temp->next;
            pre = NULL;
            cur = free_list;
        }

    }


}
//结束进程后，清空内存
void dealloc_all(int num) {
```

```
    memoryBlock* cur = used_list;
    while (cur != NULL  && cur->num != num)
    {
        cur = cur->next;
    }
    if (cur != NULL) {
        dealloc(cur->start);
        dealloc_all(num);
    }

}
//打印内存空闲情况和使用情况
void print_memory_status() {
    printf("内存情况:\n");
    printf("总共内存:%zu bytes\n",memory_size);
    size_t used=0, free=0;
    memoryBlock* memoryblock=used_list;
    FreeBlock* freeblock=free_list;
    while (memoryblock != NULL) {
        used = used + memoryblock->size;
        memoryblock = memoryblock->next;
    }
    while (freeblock != NULL) {
        free = free + freeblock->size;
        freeblock = freeblock->next;
    }
    printf("使用的内存:%zu bytes\n", used);
    printf("空闲的内存:%zu bytes\n", free);
    printf("使用的内存块:\n");
    memoryblock = used_list;
    freeblock = free_list;
    while (memoryblock != NULL) {
        printf("编号%d的进程，开始地址:%p,结尾地址:%p 大小:%zu bytes\n", memoryblock-
>num,memoryblock->start, memoryblock->end, memoryblock->size);
        memoryblock = memoryblock->next;
    }
    printf("空闲的内存块:\n");
    while (freeblock != NULL) {
        printf("开始地址:%p,结尾地址:%p 大小:%zu bytes\n", freeblock->start,
freeblock->end, freeblock->size);
        freeblock = freeblock->next;
    }
}

//打印页表
void print_page_table() {
    /*printf("page table:\n");*/
    //for (size_t i = 0; i < page_table_size; i++) {
    //  if (page_table[i] == NULL){}
    //      //printf("Page %zu free\n", i);
    //  else
    //      printf("Page %zu alloc (开始地址：%p)\n", i,page_table[i]);
    //}

    Page_Table* pcur = page_table;
    while (pcur != NULL) {
        printf("%d的页表：\n", pcur->num);
        for (size_t i = 0; i < page_table_size; i++) {
```

```
            if (pcur->page[i] == 0) {}
            //printf("Page %zu free\n", i);
            else
                printf("Page %zu alloc (开始地址：%p)\n", i, pcur->page[i]);
        }
        printf("\n");
        pcur = pcur->next;
    }
}

int main() {
    init_memory_pool(1024);//初始化内存池
    init_free_list();//初始化空闲列表
    init_used_list();//初始化使用列表
    init_page_table(10);//初始化页表
    void* ptr[4]; //模拟进程
    srand((unsigned int)time(NULL));
    int size,num;
    for (int i = 0; i < 10; i++) {
        size= rand() % 300 + 1;//随机内存
        num = rand() % 4;//随机进程号
        ptr[num]=alloc(size, num);
        print_memory_status();
        printf("\n");
        printf("\n");
    }
    dealloc(ptr[0]);//清除一个内存块
    dealloc_all(1);//清除进程所有内存块
    print_memory_status();
    printf("\n");
    print_page_table();
    printf("通过页表找内存块，删除所有的使用内存\n");
    Page_Table* cur=page_table;
    while (cur!= NULL) {
        for (size_t i = 0; i < page_table_size; i++) {
            if (cur->page[i] == 0) {}
            else
                dealloc(cur->page[i]);

        }
        cur = cur->next;
    }
    print_memory_status();
    print_page_table();

}
```