
本科生实验报告

学生姓名：　丁晓琪

学生学号：　22336057

专业名称：　计科



# 1.实验要求

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。在实现了malloc和free后，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。最后将结果截图并说说你是怎么做的

# 2. 实验过程

1. 添加malloc和free的系统调用
2. 实现堆池(`Heap_pool`)类（堆池内管理分配）
    - 分配字节：先在进程堆池中现有的分配作为堆的页寻找是否有符合要求的空间（页内采用链表管理的首次适应的动态内存分配），如果没有合适的连续空间，将再分配一页给堆池。
    - 释放字节：在堆池中找到释放空间所在页，然后在页内释放该空间
3. 实现`malloc`和`free`的系统服务处理函数

# 3.关键代码

### 3.1 添加系统调用

1. `malloc`：
   - 参数：分配字节数（`bytes_size`）
   - 返回：分配到的进程用户空间中的虚拟地址
2. `free`：
   - 参数：需要释放空间的起始地址(`begin_address`),释放的字节数(`bytes_size`)
   - 显示：是否成功释放

```
systemService.setSystemCall(5, (int)syscall_malloc);
systemService.setSystemCall(6, (int)syscall_free);
int malloc(int bytes_size) {
    return asm_system_call(5, (int)bytes_size);
}

int syscall_malloc(int bytes_size) {
    return programManager.malloc(bytes_size);
}
int free(int begin_address,int bytes_size) {
    return asm_system_call(6, (int)begin_address,(int)bytes_size);
}

int syscall_free(int begin_address,int bytes_size) {
    return programManager.free(begin_address,bytes_size);//两个参数都是int类型
}
```

### 3.2 实现堆池

1. **数据结构**
   - 首次适应的链表类`Fit_List`：
     - 作用：实现页内字节为粒度的首次适应动态分配管理
     - 链表项`Fit_ListItem`：

```
struct Fit_ListItem
{
    int begin_address=-1; //空闲孔开始地址，为-1代表该孔完全没被任何进程使用
    int size=0;           //孔的大小
    int is_allocate=0;    //为0代表该孔分配某进程且管理的空间空闲，为1代表该孔分配给某进程且管理的空间被使用
    Fit_ListItem *previous=0;
    Fit_ListItem *next=0;
};
```

     - 链表类`Fit_List`：

       每个链表管理一个页

```
class Fit_List
{
public:
    Fit_ListItem start;  //头节点，是head指针指向的节点
    Fit_ListItem* head;  //头节点的指针
    Fit_ListItem one;    //第一个节点
    char* start_address; //该链表管理的页的虚拟开始地址
public:
    // 初始化List
    Fit_List();
    // 显式初始化List
    void initialize();
    ......
    int allocate(int size);           //在页内分配size大小的字节，返回分配位置的页内偏移
    Fit_ListItem* find_fit(int size); // 返回第一个能容纳下需要大小的页内位置的节点
    void release(int start_address,int size);//释放start_address（页内偏移）处的size个字节
    Fit_ListItem* find_release(int start_address);
    void print_allocate();            //打印该页的分配孔洞的情况
    Fit_ListItem* find_hole();
};
```

该链表是根据页的动态内存分配（lab7）改造优化的，下面展示对初始化的改造点（由于`allocate`和`release`都是由lab7的函数做出微调得来，这里不多做赘述）

`initialize`:

[1] 这里的没被使用过的`item`是所有进程共享的，也就是所有链表共享的，所以在链表初始化的时候不能对`item`初始化，否则会破坏其他堆页的动态分配的信息。

[2]这里的第一个节点`one`的开始地址为0,是页内相对偏移，只有指定了分配到的物理页的虚拟地址才会对它的起始地址`start_address`进行赋值

```
Fit_ListItem item[1000];
void Fit_List::initialize()
{
    head=&start;
    head->previous = 0;//有头节点的
    head->size=0; //不知道给了地址能不能这样搞
    //  for(int i=0;i<1000;i++){
    //堆item的初始化...
    // }
    //初始化孔洞表
    one.size=4096;//这个是一个页的大小
    one.is_allocate=0;
    one.previous=head;
    one.next=0;
    one.begin_address=0;
    head->next=&one;
}
```

- 堆池`Heap_pool`
  - 作用： 每个进程都有自己的堆池，用于在用户内存空间管理堆
  - 成员：包含管理堆池中的页的链表的数组`resources`和堆池中页的数量`pool_size`

```
class Heap_pool
{
public:
    Fit_List resources[50];//最多分配50页当成堆池
    int pool_size; //堆池中页的数量
public:
    // 初始化地址池
    void initialize();
    // 从地址池中分配count个连续页，成功则返回第一个页的地址，失败则返回-1
    int allocate( int size);
    // 释放若干页的空间
    void release(const int address,int size);
    void print();
};
```

2. **初始化** `void Heap_pool::initialize()`

   初始化`pool_size`为0和对链表数组`resources`初始化

```
void Heap_pool::initialize()
{
    pool_size=0;
    for(int i=0;i<50;i++){
        resources[i].initialize();
    }
    // one.begin_address=startAddress;  //这个留到分配堆页的时候做
}
```

3. **分配** `int Heap_pool::allocate(const int size)`

   - 思路：先从现存页中查找，如果现存页不满足请求，申请再分配一页进入堆池，再在新分配页中查找满足请求的位置
   - 怎么在现在进程的用户空间中申请页：直接调用内存管理的`allocatePages`函数，指定分配空间为用户空间即可。这个函数会先在现在运行的进程`running`的虚拟用户空间里面分配页，再从总的用户物理帧池中分配物理页，建立虚拟页和物理页联系的页表项，最终返回该页的虚拟地址。

```
int Heap_pool::allocate(const int size)
{
    //首先先看现存堆池能否满足要求//这里假设申请的字节一次性不超过一页
    int address=0;
    for(int i=0;i<pool_size;i++){
       address=resources[i].allocate(size)+
(int)resources[i].start_address;
       if(address!=-1) return address;
    }
    if(size==50){
        printf("full error!\n");
        return -1;//堆池满了，不成功
    }//不满足则分配新页进入堆池
    else{//堆池为0
       //先申请分配一个用户空间的页
      char *startAddress= (char
*)memoryManager.allocatePages(AddressPoolType::USER,1);//给的就是running进程的页的虚拟地址
```

```
        resources[pool_size].start_address=startAddress;//给管理链表赋值页的开始地址
        address=(int)startAddress+resources[pool_size].allocate(size);//分配
        pool_size++;
        return address;
    }
}
```

4. **释放**`void Heap_pool::release(const int address,int size)`
   - 遍历堆池中的页，找到要释放的空间的所在页，然后通过管理页的链表释放
   - 注意：传进来的`address`是虚拟地址，而要传进`release`里面的地址是页内偏移，需要减去页的`start_address`

```

void Heap_pool::release(const int address,int size)
{
    for(int i=0;i<pool_size;i++){
        int re_address=(int)resources[i].start_address;
        if(address>=re_address&& address<re_address+4096){
            resources[i].release(address-re_address,size);
            break;
        }
    }
    //resources.release(address,size);
}
```

5. **PCB中添加**`Heap_pool`**的属性**，并且记得在创建函数`int ProgramManager::executeThread`中对堆池初始化
6. **清除堆池void Heap_pool::clear()**
   - 作用：放在进程的结束函数处，清除堆池
   - 直接把孔洞的相关分配信息恢复初始化

```
void Heap_pool::clear(){
      for(int i=0;i<pool_size;i++){
        resources[i].clear();
    }
}
void Fit_List::clear(){
    Fit_ListItem *temp = head->next;
    while (temp)
    {
        temp->begin_address=-1;
        temp->is_allocate=0;
        temp = temp->next;
    }
}
```

### 3.3实现系统服务处理函数

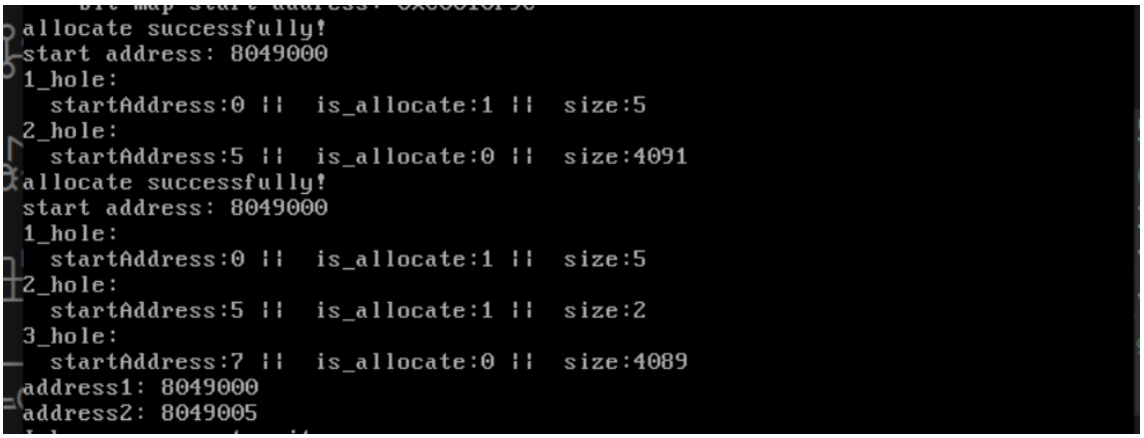
直接调用堆池的相关处理函数，并且打印分配信息

```
int ProgramManager::malloc(int bytes_size){
    int address=running->heap.allocate(bytes_size);
    printf("allocate successfully!\n");
    running->heap.print();
    return address;
}
int ProgramManager::free(int begin_address,int bytes_size){
    running->heap.release(begin_address,bytes_size);
    printf("free successfully!\n");
    running->heap.print();
    return 1;
}
```

# 4. 实验结果

1. 一个堆池页中分配两处

```
void first_process()
{
     int address1=malloc(5);
     // int address2=malloc(4095);
     int address3=malloc(2);
     printf("address1: %x\n",(char* )address1);
     printf("address2: %x\n",(char*) address3);
     //printf("address3: %x\n",(char*) address3);
     //free((int)address1,1);
}
```




2. 现有堆池页不能满足申请的情况,重新分配新页

```
void first_process()
{
     int address1=malloc(5);
     int address2=malloc(4095);
     int address3=malloc(2);
     printf("address1: %x\n",(char* )address1);
     printf("address2: %x\n",(char*) address2);
     //printf("address3: %x\n",(char*) address3);
     //free((int)address1,1);
}
```

```
start address: 8049000
1_hole:
  startAddress:0 ||  is_allocate:1 ||  size:5
2_hole:
  startAddress:5 ||  is_allocate:1 ||  size:2
3_hole:
  startAddress:7 ||  is_allocate:0 ||  size:4089
start address: 804A000
1_hole:
  startAddress:0 ||  is_allocate:1 ||  size:4095
2_hole:
  startAddress:4095 ||  is_allocate:0 ||  size:1
address1: 8049000
address2: 804A000
```

3. 两个进程进行malloc操作

```
void first_process()
{
     int address1=malloc(5);
     printf("address1: %x\n",(char* )address1);
}
void second_process()
{
     int address1=malloc(100);
     printf("address1: %x\n",(char* )address1);
}
```

```
    bit map start address: 0xC0010F9C
allocate successfully!
start address: 8049000
1_hole:
  startAddress:0 ||  is_allocate:1 ||  size:5
2_hole:
  startAddress:5 ||  is_allocate:0 ||  size:4091
address1: 8049000
I have no parent,exit
allocate successfully!
start address: 8049000
1_hole:
  startAddress:0 ||  is_allocate:1 ||  size:100
2_hole:
  startAddress:100 ||  is_allocate:0 ||  size:3996
address1: 8049000
```

4. 释放：释放第二个堆池页的`address2`位置的1000个字节

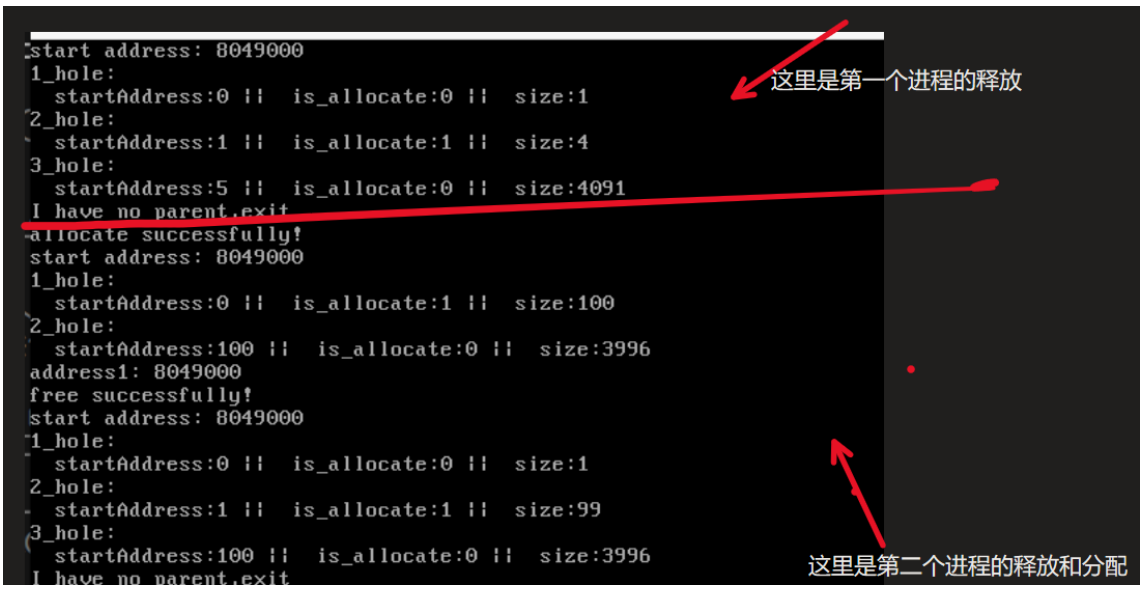
```c
void first_process()
{
     int address1=malloc(5);
     int address2=malloc(4095);
     int address3=malloc(2);
     printf("address1: %x\n",(char* )address1);
     printf("address2: %x\n",(char*) address2);
     //printf("address3: %x\n",(char*) address3);
     free((int)address2,1000);
}
```

```
address1: 8049000
address2: 804A000
free successfully!
start address: 8049000
1_hole:
  startAddress:0 ||  is_allocate:1 ||  size:5
2_hole:
  startAddress:5 ||  is_allocate:0 ||  size:4091
start address: 804A000
1_hole:
  startAddress:0 ||  is_allocate:0 ||  size:1000
2_hole:
  startAddress:1000 ||  is_allocate:1 ||  size:3095
3_hole:
  startAddress:4095 ||  is_allocate:0 ||  size:1
```

5. 两个进程的释放

```c
void first_process()
{
     int address1=malloc(5);
     printf("address1: %x\n",(char* )address1);
     free((int)address1,1);
}
void second_process()
{
     int address1=malloc(100);
     printf("address1: %x\n",(char* )address1);
    free((int)address1,1);

}
```



6. 两个进程的说明：两个进程分配到的虚拟地址虽然是一样的，但是物理地址不同，堆池也不会相同

# 5.总结

- 这是在lab7以页为单位的动态内存分配的基础上改造为以字节为单位，用到了首次适应和堆池的方式。
- 实现上遇到的问题：主要为链表访问到未经定义的空间出现的问题
  - 链表项指针没有初始化指向有效空间
  - 在函数内定义局部的链表项变量，离开函数后变量空间被释放，访问无效
  - 在头文件定义全局变量数据，多次引用头文件，导致全局变量被重复定义
- 可以优化的地方：如果请求的空间超过一个页，可以连续分配多个物理页，但是只返回第一个页的虚拟地址（虚拟地址是连续的）。释放时可以切成不超过一页的大小的片段逐个释放