**扩展的内存管理机制设计实验及测试**

**环境：**Tornado、VxSim仿真

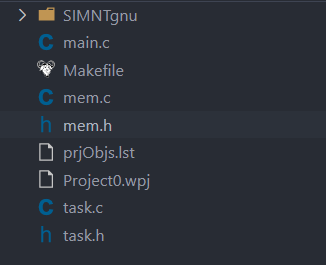
**实现目的：**

**1** 实现内存管理机制

2 通过进程通信测试内存管理机制（申请内存）

**实验内容：**

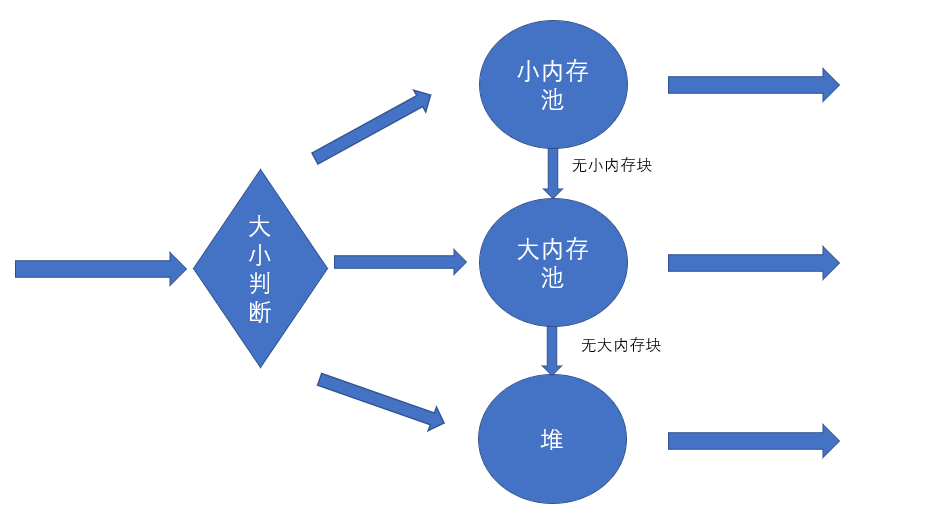
将本次实验代码分为3部分，分别为内存管理部分，task部分以及主程序入口。分别对应下图mem.c，mem.h、task.c,task.h和main.c。



**1 内存管理部分**

在本部分实现静态预分配内存池，分为大内存块和小内存块，根据请求内存大小自动分配内存块，当请求大小大于大内存块时，向堆申请内存。因此需要定义大小内存块管理结构，和堆内存的管理方法。

申请内存过程如下图所示。



外部传入内存请求，传入需要的字节大小，之后通过条件判断去按类型请求内存块，当内存块不够时，请求更大的内存块。

在头文件中进行宏定义。

|  |  |
| --- | --- |
| 宏名称 | 含义 |
| SMALLBLOCKSIZE | 小内存块大小 |
| BIGBLOCKSIZE | 大内存块大小 |
| SMALLBLOCKNUM | 小内存块数量 |
| BIGBLOCKNUM | 大内存块数量 |
| BLOCKBUSY | 内存块占用状态 |
| BLOCKAVAILABLE | 内存块空闲状态 |
| HEAP | 内存类型标识（堆） |
| SMALLBLOCK | 内存类型标识（小内存块） |
| BIGBLOCK | 内存类型标识（大内存块） |

定义大小内存块的字节数。

定义管理结构，包含大小内存池的基地址，使用数量，以及状态数组，表示占用或空闲（在这里使用数组，同时也可以使用链表方式）。

堆内存管理采用链表方式存储，定义节点结构体，存储分配的内存起始地址，大小和下一个内存块地址。

/\* define small memory block \*/

typedef struct

{

    char data[**SMALLBLOCKSIZE**]; /\* define the data area(small\_block) \*/

}small\_block;

/\* define big memory block \*/

typedef struct

{

    char data[**BIGBLOCKSIZE**]; /\* define the data area(big\_block) \*/

}big\_block;

/\* define small block management struct \*/

typedef struct

{

    small\_block \*base\_add; /\* small blocks' base address \*/

    int used\_num; /\* the number of used small blocks \*/

    char status[**SMALLBLOCKNUM**]; /\* the status of small block \*/

}small\_block\_manage;

/\* define big block management struct \*/

typedef struct

{

    big\_block \*base\_add; /\* big blocks' base address \*/

    int used\_num; /\* the number of used big blocks \*/

    char status[**BIGBLOCKNUM**]; /\* the status of big block \*/

}big\_block\_manage;

/\*

    if the memory size need bigger than BIGBLOCKSIZE, need get it from heap.

    define a link list to manage the memory(>BIGBLOCKSIZE) allocated from the heap

\*/

/\* define the link node \*/

typedef struct heap\_add\_node

{

    int \* heap\_add; /\* the address in heap of the mermory \*/

    int size; /\* memory size \*/

    struct heap\_add\_node \*next\_node; /\* link to next heap\_add\_node \*/

}heap\_add\_node;

声明以下的函数。

|  |  |
| --- | --- |
| 函数名 | 功能 |
| void cal\_mem(int\* base\_mem); | 计算大小内存池初始地址 |
| int mem\_init(int\* init\_mem); | 内存初始化 |
| void free\_mem(int \*free\_add); | 释放内存 |
| int\* malloc\_mem\_type(int size,int type); | 按类型分配内存（递归方式） |
| int\* malloc\_mem(int size); | 申请内存，使用malloc\_mem\_type |
| void free\_list(); | 释放所有申请的堆内存 |
| void traverse\_list(); | 遍历申请的堆内存 |
| void show\_mem(); | 显示内存使用情况 |

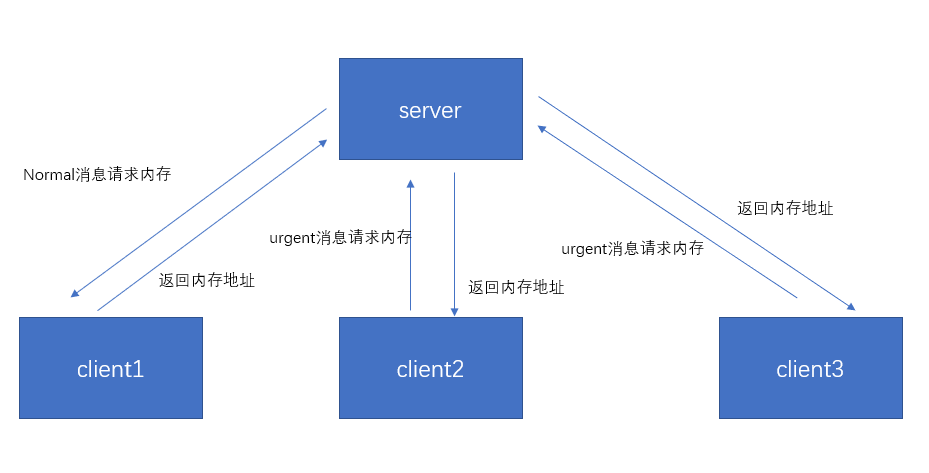
**防止内存泄露：**

注意C语言中无析构函数，在free\_mem函数中，若释放的是堆内存，不仅仅需要释放管理节点，即heap\_add\_node，还要释放该节点内的指针所指向的内存，即heap\_add指向的内存。在申请内存时，即malloc\_mem\_type函数中，若申请到内存，但添加heap\_add\_node到链表中失败时，也要按申请内存的顺序反序释放内存，防止内存泄漏。

**互斥操作：**

大小内存池以及堆是共享的，因此申请时需要互斥访问，需要设置3个信号量，这里为sem\_small, sem\_big, sem\_heap。

**2 task部分**



创建4个task模拟上图过程，client1,client2,client3分别向server发送请求，server分配内存将其起始地址返回。其中client2和client3会使用urgent消息，为保证urgent消息先进先出，需要设置二值信号量,这里为sem\_urgent\_msg。

void **server**();

void **client1**();

void **client2**();

void **client3**();

void **task\_run**();

定义消息结构体，请求消息REQ和应答消息RES

typedef struct{

    char pri;

    int pid;

    int size;

    int \*mem\_add;

} REQ;

typedef struct{

    int ack;

    int \*mem\_add;

} RES;

REQ中，pri表示该消息优先级，’u’为urgent,’n’为normal,pid标志客户端身份，size表示请求内存大小,mem\_add表示内存地址（若需要free内存，可以将pri设置为’f’配合此地址释放）。

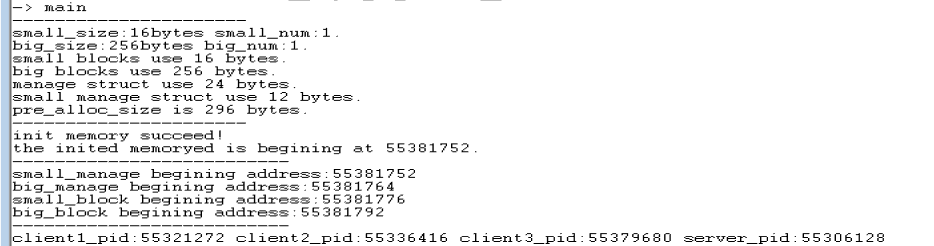
RES中，ack表示响应状态，1为成功,0为失败。mem\_add为分配的内存首地址。

在设计发送urgent消息时，需要semTake信号量sem\_urgent\_msg，当服务器处理完毕时，将其释放，防止新的urgent消息抢占在未处理的urgent消息之前，导致无法先进先出。

**3 主程序部分**

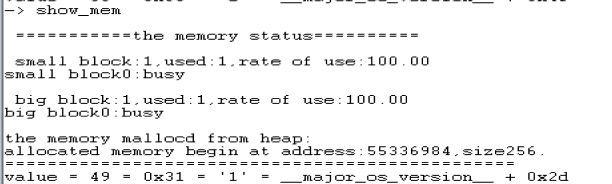
该部分主要调用相关函数，如调用mem\_init初始化内存，接着调用task\_run，启动4个任务。在本次实验中，将大内存块和小内存块分别设置为1块，3个client可以分别获得到不同的内存。

在shell中启动main函数：



上图分别输出了内存块大小数量等相关信息，初始内存的地址，以及各个内存池起始地址，以及4个进程的pid。

在main中已经调用了task\_run函数，接着在shell中调用show\_mem函数查看内存分配情况。



以下为windview图，表明各个进程获得/释放信号量已经发送/接受消息的顺序。