实验一 可变分区存储管理

**学号：515030910067 姓名： 杨超琪 日期： 2018.3.29**

**一、实验题目**

编写一个C程序，用char \*malloc(unsigned size)函数向系统申请一次内存空间（如size=1000，单位为字节），用循环首次适应法addr = (char \*)lmalloc(unsigned size) 和lfree(unsigned size,char \* addr)模拟UNIX可变分区内存管理，实现对该内存区的分配和释放管理。

**二、算法设计思路**

循环首次适应算法是首次适应算法的变种。在分配内存空间时，不再每次从表头（链首）开始查找，而是从上次找到空闲区的下一个空闲开始查找，直到找到第一个能满足要求的空闲区为止，并从中划出一块与请求大小相等的内存空间分配给作业。该算法能使内存中的空闲区分布得较均匀。

循环首次适应法包括**分配算法**、**回收算法**两大模块，分配算法指的是将内存分配给作业的算法，而回收算法指的是在作业完成后，操作系统回收内存的算法。但循环首次适应算法比首次适应算法好的地方是使用链表或者数组结构存储空闲分区的地址与空间大小，并且在分配内存时，循环链表或数组，找到第一个可适应的分区分配给作业。这种方法的好处是：（1）首先可以将内存中的空闲分区分布得比较均匀；（2）再循环分配空闲分区时，先分配的作业先结束，等到循环到这个分区时，可能作业已完成，明显提高分配查找速度。

在这个实验中，我们使用双向链表来存储空闲分区的列表，因为链表结构比较灵活，易于处理分区的增多与减少，并且设置了头指针与尾指针。下面具体介绍了分配算法和回收算法模块：

**分配算法：**即char \*lmalloc(unsigned size)，分配算法中，为了查找可用的第一个分区，首先我们需要一个可以从头遍历到为的指针。在每次分配时，总是从起始查找指针所指的表项开始查找，第一次找到大于或等于申请大小的空闲区时，就按所需大小分配给作业，此处分为三种情形：

1）如果当前空闲分区大小大于作业所需大小，则从分区起始地址开始，给作业分配所需大小，并且将该空闲分区的头指针往下移，同时空闲分区相应减小。

2）如果当前空闲分区大小等于作业所需大小，则直接把该分区划分给作业，同时，将上一个空闲分区与下一个空闲分区连接起来，继续构成循环链表。

3）如果当前空闲分区大小小于作业所需大小，则依照链表方向，指针移动。

如果查找操作循环一遍以后，并未找到合适的可用分区，则说明当前作业不可装入内存，返回错误信息，跳出程序。

**回收算法：**即bool lfree(unsigned size,char \*addr)，在这里，由于我们采用的是双向链表存储，我们依然采用循环的方式来释放我们的分区。其实在分区地址制定了以后，如果使用数组存储，可能有比循环遍历更好的方式来找到释放的地址，在这里不做讨论。在我们的双向链表存储结构的回收过程中，主要可以分为四种情形：

1）情况a：释放区仅与前空闲区相连。合并释放区与前空闲区，前空闲区的结点地址不变，大小变为原大小与释放区大小之和。

2）情况b：释放区与前、后空闲区均相连。将三块空闲区合并成一块，前空闲区的结点地址不变，前空闲区的大小变为三块空闲区之和，删除原后空闲区的结点，释放内存空间，并对各自的前驱分区与后继分区进行修改。

3）情况c：释放区仅与后空闲区相连。合并释放区与后空闲区，后空闲区的结点地址变为释放区的起始地址，空闲区的大小变为原大小与释放区大小之和。

4）情况d：释放区与前、后空闲区均不相连。需要申请一个新的分区节点，调用malloc函数向内存申请相应大小的内存，并对其地址、大小、直接后继与直接前驱结点进行赋值，并分别修改前、后空闲区的直接后继和直接前驱结点。

如果回收操作循环一遍以后，并未找到合适的可回收分区，则说明当前作业释放失败，返回错误信息，跳出程序。

可以看出，循环首次适应算法不管是在分配算法上还是在回收算法上，都与较首次适应算法好，因为其可以循环查找，只要找到第一个可适应的分区，即可完成分配任务。这种方法的好处是：（1）首先可以将内存中的空闲分区分布得比较均匀；（2）再循环分配空闲分区时，先分配的作业先结束，等到循环到这个分区时，可能作业已完成，明显提高分配查找速度。同时，由于我们使用的是双向循环链表，在完成分配与回收任务时，不管是分区合并，还是产生新分区，都只与局部链表有关，而不需像数组一样进行批量移动。

**三、模块设计、功能和接口说明**

本实验包含四个主要模块，分别是：分配算法模块，回收算法模块，分区显示模块与主函数模块。

本程序的顶层设计框架如下图：

**分配算法模块：**即char \*lmalloc(unsigned size)，输入参数为所需内存的大小，返回的结果为所分配的内存的其实地址。该模块的动能是分配内存。

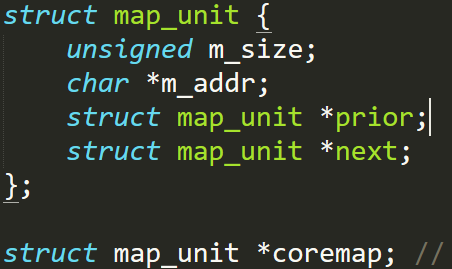
**回收算法模块：**即bool lfree(unsigned size, char \*addr)，输入参数为所释放的内存的大小与释放区的起始地址，返回的结果为布尔变量，成功与否。该模块的动能是给定起始地址与内存大小，释放内存。

**分区显示模块：**即void traverse()，无输入参数，也无返回参数，该模块的功能是格式化显示内存中空闲分区的列表。

**主函数：**即int main()，主函数是连接整个程序过程的纽带，首先其初始化双向链表，接受从终端输入的命令，进行解析，然后调用上述三个模块与用户进行交互，最后释放所有申请空间，交还给内存。

**四、重要数据结构和变量说明**

**1、双向链表结点**：空闲分区表的存储结构



此结构体定义了双向链表的结点，其中的m\_size为无符号整型，保存空闲分区的大小；m\_addr为字符型指针，保存空闲分区的起始地址指针；prior与next分别为在双向链表中，该节点的前驱结点与后继节点的指针。其中 coremap为双向链表当前正在操纵的结点。

**2、头结点**：双向链表的头指针





head表示的是双向链表的头指针，其初始化的时候，使用malloc函数从内存中申请一个地址。在后续进行内存空间遍历、分配算法与回收算法时，头指针都是起到了起点的作用，初始化时，其空闲分区大小为0。

**3、尾结点**：双向链表的尾指针





tail表示的是双向链表的尾指针，其初始化的时候，使用malloc函数从内存中申请一个地址。在后续进行内存空间遍历、分配算法与回收算法时，尾指针都是起到了终点的作用，初始化时，其空闲分区大小为0，初始地址为头指针地址加虚拟内存大小10000。

**4、当前结点**：双向链表的尾指针







coremap表示的是双向链表的当前指针，其初始化的时候，使用malloc函数从内存中申请一个地址。在后续进行内存空间遍历、分配算法与回收算法时，coremap都是起到了至关重要的作用，初始化时，其空闲分区大小为10000，初始地址与头指针地址一致。

**5、用户输入**：用户命令



c变量在我们整个程序中代表了用户的输入，可取f（回收）、m（分配）、q（退出程序）三个指令，利用getchar()函数来获取。

**五、测试方法与分析**

**1、测试方法：**

本实验采用白盒法的路径测试方法，测试路径包含lmalloc的“循环”、“首次”， lfree的与邻近空闲分区联系的四种情况，并包含一些必要的出错处理路径。

程序中设定的内存空闲分区大小为10000，在此基础上进行存储管理的测试： 具体操作如下：

m 2000 // allocate first time

m 10000 // allocate failed

f 200 300 // free (d)

f 1000 500 // free (a)

f 100 1900 // free (c)

m 7500

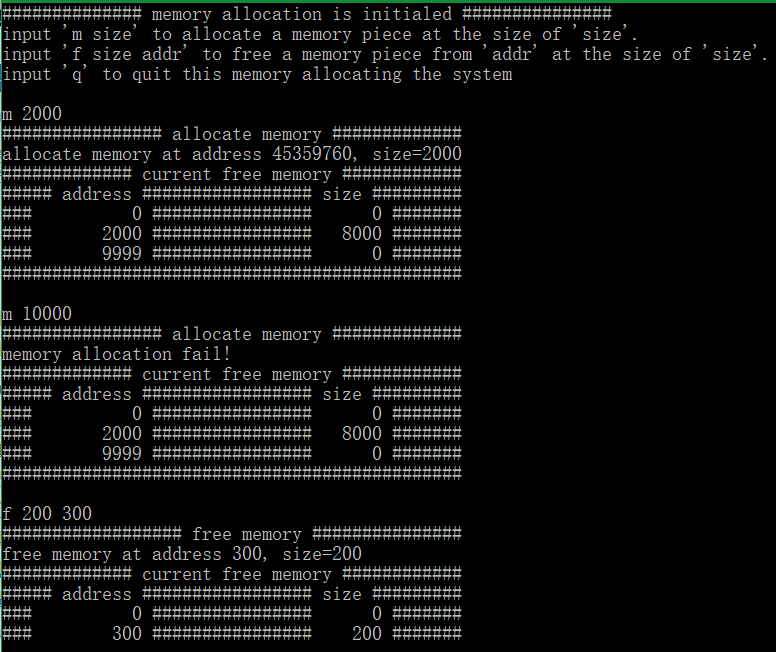
m 1000 // allocate rotary

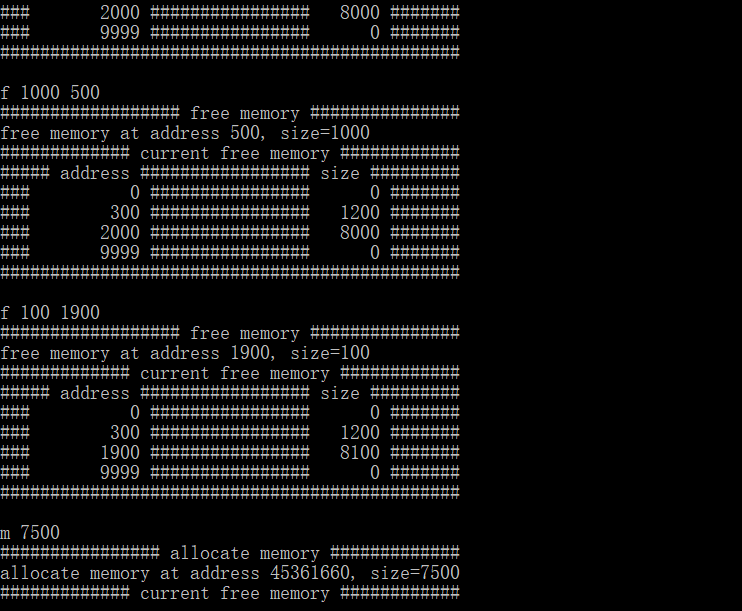
f 7900 1500 // free (b)

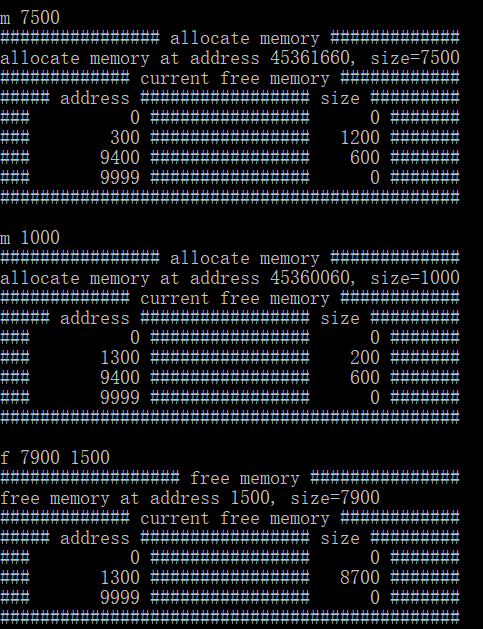
f 200 1200 // free failed

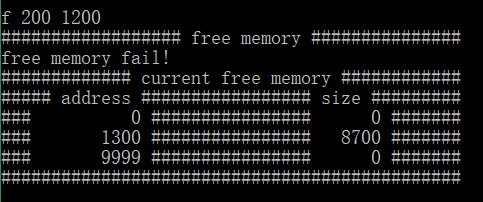
q

得到输出结果如下：











**2、测试分析**

我们通过上述的测试，对包含lmalloc的“循环”、“首次”，lfree的与邻近空闲分区联系的四种情况，同时还有包含一些必要的出错处理路径：lmalloc申请空间出错与lfree回收出错进行测试，发现我们的代码在所有情况下都能做出正确的反应，或者是应急处理。

测试通过后，我们采用输入I/O转向，将预先设计好的所有输入次序写入输入文件inputfile.txt，再用: lab\_1.exe<inputfile.txt>result.txt即可实现。

并最后将程序、输出结果文件和输入命令文件合并成progres.txt文件。实现方法是在cmd中调用命令: copy lab\_1.c+inputfile.txt+result.txt progres.txt。

**六、程序及测试的改进与体会**

**1、存在不足及展望：**

1）在这里我们是使用双向链表来存储管理空闲分区，但是做完实验以后，我发下无论是在分配空闲分区，还是在回收分区，抑或是在遍历空闲分区列表的时候，我们都只用到了链表单向的信息。所以我觉得我们或许还可以用一种更加简便的单链表来存储管理空闲分区，当然这只是一种猜想。

2）在做实验的过程中，我遇到了一个问题，（char \*）与（char \*）不能相加，或许是我对于C语言的指针和指针指向的值这两个概念有些混淆，最终把两者都暴力改成（int）类型就没有bug了，但是这个问题仍然存在我的脑海里，即使在网上搜索类似的知识，仍然没有搞懂，相信如果有更多接触C语言的机会，这个问题将会被解决。

3）第三个问题产生在最后一步，当我在inputfile.txt中输入调试命令加上备注时，程序运行的结果不一样，可能是cmd在执行exe的时候，//符号并不视作注释符号吧，之后在删除了注释后，程序正常运行。

4）其实现在这个程序还存在个问题，当用户输入回收内存的命令时，三个字符间必须有且仅有一个空格，如果用户在其中输入多个空格，I/O输入就会出错。因为在程序中我们是利用了简单的几行代码来进行这样的I/O控制，因此有局限性，若增加几行代码，相信可以解决这个问题。

**2、体会**

这其实是我写的第一个C语言的程序，之前学过C++，虽然语法与C语言较为类似，但是一些库的名字与关键的输入输出，特别是变量类型上，C++与C语言还是有很大的区别的。做完这个实验我深切地体会到了自己如果要走上开发岗位，特别是底层系统的开发，在这些基础细节上还是挺欠缺的。

这次实验让我惊讶的是，我发现C语言的运行效率真的是极高的，与用户交互的速度非常快，这是我在Python，Java等语言中没有体会过的。如果能熟练并精通C语言将会是自己未来发展一个极大的优势。

言归正传，不得不感叹循环首次适应法的思想是很赞的，在这次实验中，我能用C语言，利用指针与链表工具独立完成内存管理系统的编写，还是一次蛮棒的体验！

**七、源代码及其注释**

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

// declare a struct to store the memory piece

struct map\_unit {

unsigned m\_size;

char \*m\_addr;

struct map\_unit \*prior;

struct map\_unit \*next;

};

struct map\_unit \*coremap; // current memory piece

struct map\_unit \*head; // head pointer

struct map\_unit \*tail; // tail pointer

// allocate memory

char \*lmalloc(unsigned size){

struct map\_unit \*current\_p = coremap;

do{

// if current piece is ok

if (current\_p->m\_size>=size){

current\_p->m\_size -= size;

current\_p->m\_addr += size;

// if current piece is used up

if (current\_p->m\_size==0){

current\_p->prior->next = current\_p->next;

current\_p->next->prior = current\_p->prior;

}

return current\_p->m\_addr - size;

}else{

// point to the next piece

current\_p = current\_p->next;

}

}while(current\_p!=coremap);

return NULL;

}

// set the memory free

bool lfree(unsigned size,char \*addr){

struct map\_unit \*current\_p = head;

do{

// situation a

if ((current\_p->m\_addr+current\_p->m\_size==addr)

&&(current\_p->next->m\_addr>addr+size)){

current\_p->m\_size += size;

return true;

}

// situation b

else if ((current\_p->m\_addr+current\_p->m\_size==addr)

&&(current\_p->next->m\_addr==addr+size)){

current\_p->m\_size += size + current\_p->next->m\_size;

current\_p->next = current\_p->next->next;

free(current\_p->next->prior); // free memory

current\_p->next->prior = current\_p;

return true;

}

// situation c

else if ((current\_p->m\_addr+current\_p->m\_size<addr)

&&(current\_p->next->m\_addr==addr+size)){

current\_p->next->m\_addr -= size;

current\_p->next->m\_size += size;

return true;

}

// situation d

else if ((current\_p->m\_addr+current\_p->m\_size<addr)

&&(current\_p->next->m\_addr>addr+size)){

struct map\_unit \*new\_map=(struct map\_unit\*)malloc(sizeof(struct map\_unit));

new\_map->m\_addr = addr;

new\_map->m\_size = size;

new\_map->next = current\_p->next;

new\_map->prior = current\_p;

current\_p->next = new\_map;

new\_map->next->prior = new\_map;

return true;

}

else{

current\_p = current\_p->next;

}

}while(current\_p!=head);

return false;

}

// traverse the memory piece list

void traverse(){

struct map\_unit \*current\_p=head;

printf("############# current free memory ############\n");

printf("##### address ################# size #########\n");

int t = 1;

do{

printf("### %10u ################ %6u #######\n",current\_p->m\_addr-head->m\_addr,current\_p->m\_size);

current\_p = current\_p->next;

t++;

}while(current\_p!=head&& t<10);

printf("##############################################\n\n");

}

int main(){

// apply for three pointer

coremap = (struct map\_unit\*)malloc(sizeof(struct map\_unit));

head = (struct map\_unit\*)malloc(sizeof(struct map\_unit));

tail = (struct map\_unit\*)malloc(sizeof(struct map\_unit));

// apply for 10000 memory size

coremap->m\_addr = (char \*)malloc(10000);

head->m\_addr = coremap->m\_addr;

tail->m\_addr = coremap->m\_addr + 9999;

// initialize the size

coremap->m\_size = 10000;

head->m\_size = 0;

tail->m\_size = 0;

// connection

coremap->next = tail;

coremap->prior = head;

head->next = coremap;

head->prior = tail;

tail->next = head;

tail->prior = coremap;

char c; // store the command

unsigned size;

printf("############## memory allocation is initialed ###############\n");

printf("input 'm size' to allocate a memory piece at the size of 'size'. \n");

printf("input 'f size addr' to free a memory piece from 'addr' at the size of 'size'. \n");

printf("input 'q' to quit this memory allocating the system \n\n");

do{ // filter out the '\t' '\n' ' '

while (!(c=getchar()));

// allocate memory

if (c=='m'){

char \*allo\_addr;

scanf("%u",&size);

printf("################ allocate memory #############\n");

allo\_addr = lmalloc(size);

if (allo\_addr){

printf("allocate memory at address %u, size=%u\n",allo\_addr,size);

traverse();

}else{

printf("memory allocation fail!\n");

traverse();

}

}

// free memory

else if (c=='f'){

char \*addr;

scanf("%u %u",&size, &addr);

printf("################## free memory ###############\n");

if (lfree(size, (char \*)(head->m\_addr)+(int)(addr))){

printf("free memory at address %u, size=%u\n",addr,size);

traverse();

}else{

printf("free memory fail!\n");

traverse();

}

}

}while(c!='q');

free(coremap);

free(head);

free(tail);

printf("################### good bye! #################\n\n");

return 0;

}