远程进程socket通信实验

张侨伦

5140809064

#### （一） 实验目的

1．系统v的IPC通信机制只支持同一台计算机内的进程间通信。BSD UNIX，系统V和Solaris等操作系统所提供的Socket（套接字）是最重要的进程通信工具，可支持广域网上异种机器之间的通信，包括各类UNIX系统和使用最为广泛的个人计算机。通过本实验能使学生进一步理解套接字通信的基本原理和机制，通过编制一个实用的异机socket通信程序，深入掌握套接字编程的方法和细节。

2．通过实验对Socket有进一步的感性认识，掌握系统的套接字编程。

#### （二）  实验题目

1．分别编一个客机程序和服务器程序，首先建立客户程序与服务器之间正确的socket连结。

2. 利用send和recv函数，客户程序将一个较长的文本文件（如几k字节）中的数据发送给服务器。要求服务器全部正确地接收到所有的数据（一个也不能少），并将其存入一个文件。

#### （三） 实验要求

1. 给出数据结构的说明

2. 画出程序框图

3. 在源程序中加入注释

4. 说明程序的测试结果。

#### （四） 程序框图

**（1）概要设计**

整个系统的概要设计如下图1。程序开始运行时，使用malloc函数分配空间。然后接受输入命令，执行对应的分配内存区和释放内存区的操作。同时打印当前的空闲内存区的地址和大小。当输入的命令不为分配内存区或者回收内存区的时候，程序退出。



图1 系统的基本框架

**（2）算法思想**

程序接受用户输入对应的分配内存区和回收内存区的指令。

在分配内存区时，程序会检查使用循环首次适应法查找是否有合适的存储区。当有合适的存储区时，分配存储区。没有合适的存储区时，提示分配错误。

在回收内存区的时候，存在四种情况，如下图2所示。这些分配情况在于待释放区前后相邻的区域是空闲区还是分配区。如果是空闲区，就将待释放区和空闲区连在一起。同时，如果两侧都是空闲区，同时有待释放区后方的那个区域是循环首次适应法下次开始查找的位置，那么需要更换存储该位置的变量。

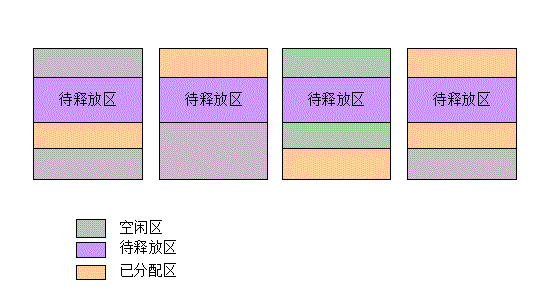


图2 回收内存区

**（3）模块的详细设计和功能**

**模块Lmalloc()**

功能：分配内存

设计：首先使用循环首次适应法查找分配的内存块。如果查找失败，返回NULL。然后判断查找到的内存块的大小和申请的内存块的大小的关系。如果两者大小相同，那么将它前后的空闲内存块相连并释放这块内存。如果大小不同，则改变查找到的内存块的大小和起始地址。

**模块Lfree()**

功能：释放内存

设计：首先确定释放的内存块的位置。

如果释放的内存块位于当前空闲的内存块的前方，那么新建一个空闲存储区，将这个内存块插入空闲存储区表的最前部。如果释放的内存块位于空闲存储区表的最后一个空闲存储区的后方，那么新建一个空闲存储区，将这个内存块插入空闲存储区表的后方。

如果不满足上面的位置，那么确认释放的内存块位于那两个空闲存储块之间。然后进行释放。释放的时候需要判断该内存块是否有相邻的空闲内存块。如果有相邻的空闲内存块那么把它和相邻的存储区块相连，并相应地更改空闲存储区表。

**（4）接口说明**

**- 函数char \* lmalloc(unsigned size)**

输入为想要分配的内存大小。

返回为分配的空闲存储区。

**- 函数int lfree(unsigned size, char \* addr)**

输入为想要释放的大小，和想要释放的起始地址，输出为是否释放成功的标志。如果输出0，释放失败，输出非0，释放成功。

#### （五）重要数据结构和变量的说明

**（1）重要的数据结构**

- IP地址

IP有两种结构体，可以互相转换。

struct sockaddr {

unsigned short sa\_family; /\**Address family(eg. AF\_INET) \**/

char sa\_data[14]; /*\* Family-specific address information \**/

};

struct in\_addr {

unsigned long s\_addr; /*\* Internet address (32 bits) \**/

}

struct sockaddr\_in {

unsigned short sin\_family; /*\* Internet protocol (AF\_INET) \**/

unsigned short sin\_port;

struct in\_addr sin\_addr; /*\* Internet address (32 bits) \**/

char sin\_zero[8]; /*\* Not used \**/

}

**（2）重要的变量**

- struct map \*coremap

使用双向链表存放空闲存储区表

- struct map \*start;

存放循环首次适应法上次查找到的位置的下一个位置。

- char \* mem\_begin;

分配的整个内存区域的起始地址

- char \* mem\_end;

分配的整个内存区域的结束地址

#### （六）测试方法和测试结果的分析

在两台机器之间测试程序。比较服务器收到的文件中的数据和客户机原先文件中数 据 是否完全相同。文本文件测试完成后，再用二进制文件测试。

1. 测试实验要求1

对于信号的处理步骤如下：

- 一个事件产生一个信号

- 系统立即停止这个进程

- 信号的handler执行

- 进程恢复执行

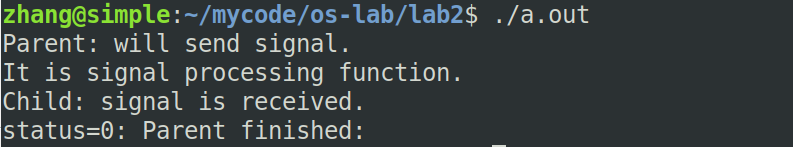


图1 测试要求1

1. 测试lmalloc的首次

首先分配60,10,20的空间，然后回收20的空间。这时候空闲的区域地址分别为大小为40和大小为10，这时候重新分配5的空间，这时符合的位置有两个。选择第一个查找到的。这时候剩下的空间就如下图4所示。

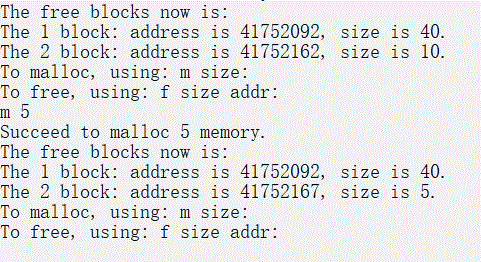


图4 测试lmalloc的首次

1. 测试lfree：两边都是已分配区

首先分配60,20,10的空间，然后释放中间20的空间。结果如下图5所示。

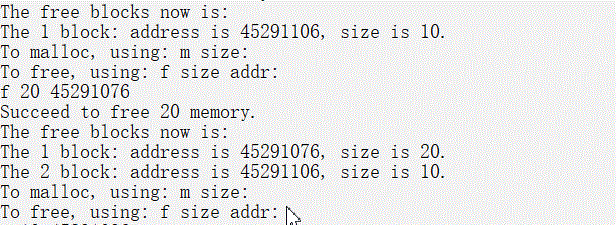


图5 测试lfree两边都是已分配区

1. 测试lfree：两边都是未分配区

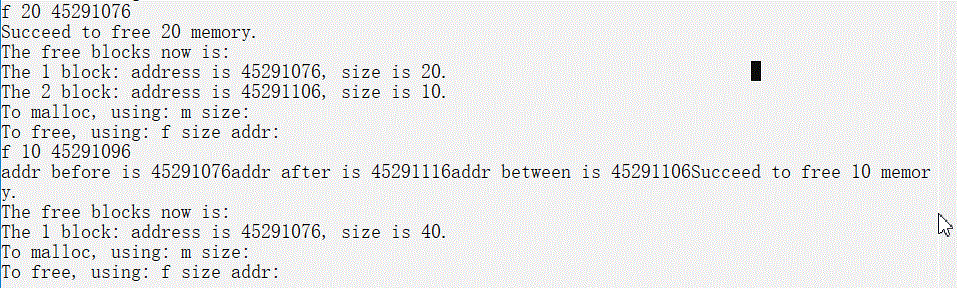


图6 测试lfree两边都是未分配区

1. 测试lfree：前面是未分配区

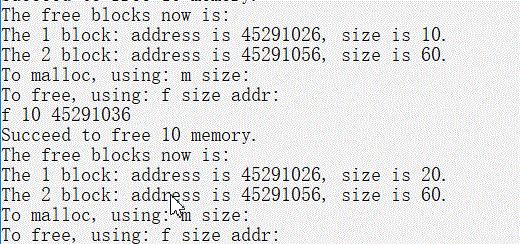


图7 测试lfree前面是未分配区

1. 测试lfree：后面是未分配区

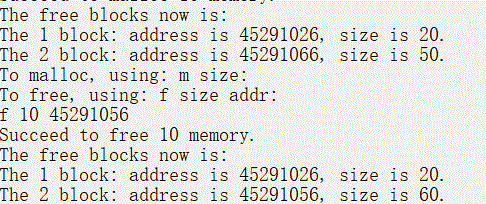


图8 测试后面是未分配区

#### （七）本次上机经验及体会

在上机过程中碰到了如下的问题，但最终都通过了思考和网上查找解决方法得以解决。在这个过程中锻炼了我对C语言的熟悉程度以及搜索问题的能力。碰到的问题如下：

1. while循环有一个循环条件没有写对，导致后来查找的时候出现了错误。
2. 在比较地址大小的时候声明了一个int型的变量存放指针地址，导致比较地址大小出现了错误。应该声明为unsigned int类型。
3. 在比较地址的时候，不小心把size和结构体里面的m\_size混用，导致出现了一些问题。
4. 调试程序的时候发现如果出现释放内存区域时，如果两边都是空闲区域，这种情况下，start指针的地址会出现问题。因为如果start指针指向的就是释放的内存区域的后方，这样会导致start指向的区域错误。需要对这种情况进行判断。然后如果是这种情况，start指针的地址需要进行改动。

#### （八）程序代码

程序代码如下所示：

#include <stdio.h>

#include <stdlib.h>

struct map {

unsigned m\_size;

char \*m\_addr;

struct map \*next, \*prior;

};

struct map \*coremap, \*start;

char \* mem\_begin;

char \* mem\_end;

// 分配内存空间

char \* lmalloc(unsigned size) {

// 最后返回前要记得更改start的值

struct map \*tmp = start;

// unsigned int address\_int;

do {

if (tmp->m\_size == size) {

(tmp->prior)->next = tmp->next;

(tmp->next)->prior = tmp->prior;

// free tmp

//address\_int = (unsigned int) (tmp->m\_addr);

free(tmp);

start = tmp->next;

return tmp;

}

else if (tmp->m\_size > size) {

start = tmp->next;

// reduce and alloc new

//struct map \*

tmp->m\_addr = tmp->m\_addr + size;

tmp->m\_size = tmp->m\_size - size;

start = tmp->next;

return tmp->m\_addr;

}

tmp = tmp->next;

} while (tmp != start);

start = tmp->next;

return NULL;

}

// 回收内存空间

int lfree(unsigned size, char \* addr)

{

struct map \* tmp = coremap;

unsigned int bd = 0;

unsigned int ad = 0;

unsigned int addr\_start\_before;

unsigned int addr\_start\_after;

unsigned int addr\_start\_between;

// if the freed block is at the beginning

// if it is out of the memory range, return 0

if (addr < mem\_begin || (addr+size-1) > mem\_end) {

return 0;

}

if (coremap->m\_addr > addr) {

if ((addr + size) < coremap->m\_addr) {

struct map \* block\_new = (struct map \*) malloc(sizeof(struct map));

block\_new->m\_addr = addr;

block\_new->m\_size = size;

block\_new->next = coremap;

block\_new->prior = coremap->prior;

(block\_new->prior)->next = block\_new;

(block\_new->next)->prior= block\_new;

coremap = block\_new;

return coremap->m\_addr;

}

else if ((addr + size) == coremap->m\_addr) {

coremap->m\_addr = addr;

coremap->m\_size = coremap->m\_size + size;

return coremap->m\_addr;

}

else

return 0;

}

// if the freed block is at the last

if (coremap->prior->m\_addr < addr) {

if ((coremap->prior->m\_addr + size) < addr) {

struct map \* block\_new = (struct map \*) malloc(sizeof(struct map));

block\_new->m\_addr = addr;

block\_new->m\_size = size;

block\_new->next = coremap;

block\_new->prior = coremap->prior;

(block\_new->prior)->next = block\_new;

(block\_new->next)->prior= block\_new;

return block\_new->m\_addr;

}

else if ((coremap->prior->m\_addr + size) == addr) {

coremap->prior->m\_size = coremap->prior->m\_size + size;

return coremap->prior->m\_addr;

}

else

return 0;

}

// bd = before->m\_addr, ad = after->m\_addr

// (bd + size) > addr || ad < (addr + size) return 0;

// if the freed block is not at the beginning

/\*

do {

if ((tmp->m\_addr < addr) && ((tmp->next)->m\_addr > addr)) {

break;

}

} while (tmp != start);

\*/

while (tmp->next->m\_addr < addr && tmp->next != coremap) {

tmp = tmp->next;

}

if (tmp->next == coremap) return 0;

// 接下来就是检查tmp->next是不是正好是在两个中间的一个值

if ((tmp->m\_addr > addr) || ((tmp->next)->m\_addr <= addr)) {

return 0;

}

bd = tmp->m\_addr;

ad = tmp->next->m\_addr;

if ((bd + tmp->m\_size) > addr || ad < (addr + size)) {

return 0;

}

// first case 两边都没有

if ( (bd + tmp->m\_size) < addr && ad > (addr + size) ) {

struct map \* block\_new = (struct map \*) malloc(sizeof(struct map));

block\_new->m\_addr = addr;

block\_new->m\_size = size;

block\_new->next = tmp->next;

block\_new->prior = tmp->prior;

(block\_new->prior)->next = block\_new;

(block\_new->next)->prior= block\_new;

return block\_new->m\_addr;

}

// second case

// 前后都合并进来

if ( (bd + tmp->m\_size) == addr && ad == (addr + size)) {

// help set start

addr\_start\_before = tmp->m\_addr;

addr\_start\_after = tmp->next->m\_addr + size;

addr\_start\_between = start->m\_addr;

tmp->m\_size = tmp->m\_size + tmp->next->m\_size + size;

tmp->next = tmp->next->next;

free(tmp->next->prior);

tmp->next->prior = tmp;

printf("addr before is %u", addr\_start\_before);

printf("addr after is %u", addr\_start\_after);

printf("addr between is %u", addr\_start\_between);

if (addr\_start\_before <= addr\_start\_between && addr\_start\_between <= addr\_start\_after)

start = tmp;

return tmp->m\_addr;

}

// third case

// 前面的合并进来

if ((bd + tmp->m\_size) == addr && ad > (addr + size)) {

tmp->m\_size = tmp->m\_size + size;

return tmp->m\_addr;

}

// fourth case

if ((bd + tmp->m\_size) < addr && ad == (addr + size)) {

tmp->next->m\_addr = addr;

tmp->next->m\_size = tmp->next->m\_size + size;

return tmp->next->m\_addr;

}

return 1;

}

void show\_free\_memory() {

struct map \* tmp = coremap;

int num = 1;

// printf("\n start address is %u\n", start->m\_addr);

// printf("\n coremap is %u \n", coremap->m\_addr);

printf("The free blocks now is: \n");

do {

printf("The %i block: address is %u, size is %i.\n", num, tmp->m\_addr, tmp->m\_size);

num = num + 1;

tmp = tmp->next;

} while (tmp != coremap);

}

int main()

{

int mem\_size = 100;

int opt\_size = 0;

char c;

char \* addr;

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

coremap->m\_size = mem\_size;

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

coremap->next = coremap;

coremap->prior = coremap;

mem\_begin = coremap->m\_addr;

mem\_end = coremap->m\_addr + mem\_size - 1;

start = coremap;

do {

show\_free\_memory();

printf("To malloc, using: m size: \n");

printf("To free, using: f size addr: \n");

do

c = getchar();

while (c=='\n'||c=='\t'||c==' ');

if (c == 'm') {

scanf("%i", &opt\_size);

if (lmalloc(opt\_size))

printf("Succeed to malloc %i memory.\n", opt\_size);

else

printf("Failed to malloc %i memory.\n", opt\_size);

}

else if (c == 'f') {

scanf("%i %u", &opt\_size, &addr);

if (lfree(opt\_size, addr))

printf("Succeed to free %i memory.\n", opt\_size);

else

printf("Failed to free %i memory.\n", opt\_size);

}

} while ((c == 'm' ) || (c == 'f'));

return 0;

}